Technical Communications of the 28th International Conference on Logic Programming

ICLP 2012, September 4–8, 2012, Budapest, Hungary

Edited by

Agostino Dovier
Vítor Santos Costa

Editors

Agostino Dovier
Dipartimento di Matematica e Informatica
Università di Udine
agostino.dovier@uniud.it

Vítor Santos Costa
DCC/Faculdade de Ciências
Universidade do Porto
vsc@dcc.fc.up.pt

ACM Classification 1998
D.1.6 Logic Programming, D.2 Software Engineering, F.4.1 Mathematical Logic, I.2.4 Knowledge Representation Formalisms and Methods, I.2.8 Problem Solving, Control Methods, and Search

ISBN 978-3-939897-43-9

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/978-3-939897-43-9.

Publication date
September, 2012

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 .

License

This work is licensed under a Creative Commons Attribution-NoDerivs 3.0 Unported license: http://creativecommons.org/licenses/by-nd/3.0/legalcode.
In brief, this license authorizes each and everybody to share (to copy, distribute and transmit) the work under the following conditions, without impairing or restricting the authors' moral rights:

- Attribution: The work must be attributed to its authors.
- No derivation: It is not allowed to alter or transform this work.

The copyright is retained by the corresponding authors.

Digital Object Identifier: 10.4230/LIPIcs.ICLP.2012.i

ISBN 978-3-939897-43-9 **ISSN 1868-8969** **http://www.dagstuhl.de/lipics**

LIPIcs – Leibniz International Proceedings in Informatics

LIPIcs is a series of high-quality conference proceedings across all fields in informatics. LIPIcs volumes are published according to the principle of Open Access, i.e., they are available online and free of charge.

Editorial Board

- Susanne Albers (Humboldt University Berlin)
- Chris Hankin (Imperial College London)
- Deepak Kapur (University of New Mexico)
- Michael Mitzenmacher (Harvard University)
- Madhavan Mukund (Chennai Mathematical Institute)
- Wolfgang Thomas (RWTH Aachen)
- Vinay V. (Chennai Mathematical Institute)
- Pascal Weil (*Chair*, University Bordeaux)
- Reinhard Wilhelm (Saarland University, Schloss Dagstuhl)

ISSN 1868-8969

www.dagstuhl.de/lipics

To the Association for Logic Programming

Contents

Introduction to the Technical Communications of the 28th International Conference on Logic Programming Special Issue
Agostino Dovier and Vítor Santos Costa xvii

Invited Contribution

Simulation Unification: Beyond Querying Semistructured Data
François Bry and Sebastian Schaffert 1

Knowledge Representation, Learning, and ASP

Modeling Machine Learning and Data Mining Problems with FO(·)
Hendrik Blockeel, Bart Bogaerts, Maurice Bruynooghe, Broes De Cat, Stef De Pooter, Marc Denecker, Anthony Labarre, Jan Ramon, and Sicco Verwer 14

Answering Why and How questions with respect to a frame-based knowledge base: a preliminary report
Chitta Baral, Nguyen Ha Vo, and Shanshan Liang 26

Applying Machine Learning Techniques to ASP Solving
Marco Maratea, Luca Pulina, and Francesco Ricca 37

An Answer Set Solver for non-Herbrand Programs: Progress Report
Marcello Balduccini 49

Stable Models of Formulas with Generalized Quantifiers (Preliminary Report)
Joohyung Lee and Yunsong Meng 61

Using Answer Set Programming in the Development of Verified Software
Florian Schanda and Martin Brain 72

Generating Event-Sequence Test Cases by Answer Set Programming with the Incidence Matrix
Mutsunori Banbara, Naoyuki Tamura, and Katsumi Inoue 86

Concurrency and FLP

Towards Testing Concurrent Objects in CLP
Elvira Albert, Puri Arenas, and Miguel Gómez-Zamalloa 98

Visualization of CHR through Source-to-Source Transformation
Slim Abdennadher and Nada Sharaf 109

Static Type Inference for the Q language using Constraint Logic Programming
Zsolt Zombori, János Csorba, and Péter Szeredi 119

Improving Lazy Non-Deterministic Computations by Demand Analysis
Michael Hanus 130

Technical Communications of the 28th International Conference on Logic Programming (ICLP'12).
Editors: A. Dovier and V. Santos Costa

LIPICS Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

The additional difficulties for the automatic synthesis of specifications posed by logic features in functional-logic languages
Giovanni Bacci, Marco Comini, Marco A. Feliú, and Alicia Villanueva 144

A Concurrent Operational Semantics for Constraint Functional Logic Programming
Rafael del Vado Vírseda, Fernando Pérez Morente, and Marcos Miguel García Toledo 156

Answer Set Programming

Surviving Solver Sensitivity: An ASP Practitioner's Guide
Bryan Silverthorn, Yuliya Lierler, and Marius Schneider 164

aspeed: ASP-based Solver Scheduling
Holger Hoos, Roland Kaminski, Torsten Schaub, and Marius Schneider 176

Answer Set Solving with Lazy Nogood Generation
Christian Drescher and Toby Walsh 188

Lazy Model Expansion by Incremental Grounding
Broes De Cat, Marc Denecker, and Peter Stuckey 201

Unsatisfiability-based optimization in clasp
Benjamin Andres, Benjamin Kaufmann, Oliver Matheis, and Torsten Schaub 212

An FLP-Style Answer-Set Semantics for Abstract-Constraint Programs with Disjunctions
Johannes Oetsch, Jörg Pührer, and Hans Tompits 222

Reconciling Well-Founded Semantics of DL-Programs and Aggregate Programs
Jia-Huai You, John Morris, and Yi Bi 235

Preprocessing of Complex Non-Ground Rules in Answer Set Programming
Michael Morak and Stefan Woltran 247

Foundations

Two-Valued Logic Programs
Vladimir Lifschitz 259

Possibilistic Nested Logic Programs
Juan Carlos Nieves and Helena Lindgren 267

A Tarskian Informal Semantics for Answer Set Programming
Marc Denecker, Yuliya Lierler, Miroslaw Truszczynski, and Joost Vennekens 277

Paving the Way for Temporal Grounding
Felicidad Aguado, Pedro Cabalar, Martín Diéguez, Gilberto Pérez, and Concepción Vidal 290

Logic + control: An example
Wlodzimierz Drabent 301

Deriving a Fast Inverse of the Generalized Cantor N-tupling Bijection
Paul Tarau 312

On the Termination of Logic Programs with Function Symbols
Sergio Greco, Francesca Spezzano, and Irina Trubitsyna 323

Logic Programming in Tabular Allegories
Emilio Jesús Gallego Arias and James B. Lipton 334

Applications

Tabling for infinite probability computation
Taisuke Sato and Philipp Meyer 348

ASP at Work: An ASP Implementation of PhyloWS
Tiep Le, Hieu Nguyen, Enrico Pontelli, and Tran Cao Son 359

CHR for Social Responsibility
Veronica Dahl, Bradley Coleman, J. Emilio Miralles, and Erez Maharshak 370

A Logic Programming approach for Access Control over RDF
Nuno Lopes, Sabrina Kirrane, Antoine Zimmermann, Axel Polleres, and Alessandra Mileo 381

LOG-IDEAH: ASP for Architectonic Asset Preservation
Viviana Novelli, Marina De Vos, Julian Padget, and Dina D'Ayala 393

Extending $\mathcal{C}+$ with Composite Actions for Robotic Task Planning
Xiaoping Chen, Guoqiang Jin, and Fangkai Yang 404

Improving Quality and Efficiency in Home Health Care: an application of Constraint Logic Programming for the Ferrara NHS unit
Massimiliano Cattafi, Rosa Herrero, Marco Gavanelli, Maddalena Nonato, Federico Malucelli, and Juan José Ramos 415

A Flexible Solver for Finite Arithmetic Circuits
Nathaniel Wesley Filardo and Jason Eisner 425

Doctoral Consortium

Software Model Checking by Program Specialization
Emanuele De Angelis 439

Temporal Answer Set Programming
Martín Diéguez 445

A Gradual Polymorphic Type System with Subtyping for Prolog
Spyros Hadjichristodoulou 451

ASP modulo CSP: The clingcon system
Max Ostrowski 458

An ASP Approach for the Optimal Placement of the Isolation Valves in a Water Distribution System
Andrea Peano 464

Answer Set Programming with External Sources
Christoph Redl 469

Together, Is Anything Possible? A Look at Collective Commitments for Agents
Ben Wright 476

List of Authors

Slim Abdennadher
German University in Cairo
Egypt
slim.abdennadher@guc.edu.eg

Felicidad Aguado
Coruña University
Spain
aguado@udc.esn

Elvira Albert
Complutense University of Madrid
Spain
elvira@clip.dia.fi.upm.es

Benjamin Andres
University of Potsdam
Germany
bandres@cs.uni-potsdam.de

Puri Arenas-Sanchez
Complutense University of Madrid
Spain
puri@sip.ucm.es

Giovanni Bacci
Univ. of Udine, DIMI
Italy
giovanni.bacci@uniud.it

Marcello Balduccini
Kodak Research Laboratories
USA
marcello.balduccini@gmail.com

Mutsunori Banbara
Information Science and Technology Center,
Kobe University, Japan
banbara@kobe-u.ac.jp

Chitta Baral
Arizona State University
AZ, USA
chitta@asu.edu

Yi Bi
Tianjin University
China
thalian.bi@gmail.com

Hendrik Blockeel
Department of Computer Science,
KU Leuven, Belgium
hendrik.blockeel@cs.kuleuven.be

Bart Bogaerts
Department of Computer Science,
KU Leuven Belgium
bart.bogaerts@cs.kuleuven.be

Martin Brain
University of Bath
UK
mjb@cs.bath.ac.uk

Maurice Bruynooghe
Department of Computer Science,
KU Leuven, Belgium
maurice.bruynooghe@cs.kuleuven.be

François Bry
Institute for Informatics,
Ludwig-Maximilians University of Munich
Germany
bry@lmu.de

Pedro Cabalar
Coruña University
Spain
cabalar@udc.es

Massimiliano Cattafi
University of Ferrara
Italy
massimiliano.cattafi@unife.it

Xiaoping Chen
University of Science and Technology
of China
xpchen@ustc.edu.cn

Bradley Coleman
Simon Fraser University
Canada
bradley@proxydemocracy.org

Marco Comini
University of Udine, DIMI
Italy
marco.comini@uniud.it

Technical Communications of the 28th International Conference on Logic Programming (ICLP'12).
Editors: A. Dovier and V. Santos Costa

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

János Csorba
Budapest University of Technology and Economics, Hungary
csorba@cs.bme.hu

Dina D'Ayala
University of Bath
UK
d.f.d'ayala@bath.ac.uk

Veronica Dahl
Simon Fraser University
Canada
veronica@cs.sfu.ca

Emanuele de Angelis
University "G. d'Annunzio" of Chieti-Pescara,
Italy
deangelis@sci.unich.it

Broes De Cat
Department of Computer Science,
KU Leuven, Belgium
Broes.DeCat@cs.kuleuven.be

Stef De Pooter
Department of Computer Science,
KU Leuven, Belgium
stef.depooter@cs.kuleuven.be

Marina De Vos
University of Bath
UK
mdv@cs.bath.ac.uk

Rafael del Vado Vírseda
Universidad Complutense de Madrid. Spain
rdelvado@sip.ucm.es

Marc Denecker
KU Leuven
Belgium
marcd@cs.kuleuven.be

Martín Diéguez
Coruña University
Spain
martin.dieguez@udc.es

Włodzimierz Drabent
IPI PAN Warszawa
Poland
drabent@ipipan.waw.pl

Christian Drescher
NICTA and UNSW
Australia
christian.drescher@nicta.com.au

Jason Eisner
Johns Hopkins University
USA
jason@cs.jhu.edu

Marco A. Feliú
Universitat Politécnica de Valéncia, DSIC
Spain
mfeliu@dsic.upv.es

Nathaniel Wesley Filardo
Johns Hopkins University
USA
nwf@cs.jhu.edu

Emilio Jesús Gallego Arias
Universidad Politécnica de Madrid
Spain
egallego@babel.ls.fi.upm.es

Marcos Miguel García Toledo
Complutense University of Madrid
Spain
solomiyo@gmail.com

Marco Gavanelli
University of Ferrara
Italy
marco.gavanelli@unife.it

Miguel Gómez-Zamalloa
Complutense University of Madrid
Spain
mzamalloa@clip.dia.fi.upm.es

Sergio Greco
University of Calabria
Italy
greco@deis.unical.it

Spyros Hadjichristodoulou
Computer Science Department,
Stony Brook University
NY, USA
shadjichrist@cs.stonybrook.edu

Nguyen Ha Vo
Arizona State University
AZ, USA
Nguyen.H.Vo@asu.edu

Michael Hanus
CAU Kiel
Germany
mh@informatik.uni-kiel.de

Rosa Herrero
Universitat Autònoma de Barcelona
Spain
rherrero.math@gmail.com

Holger Hoos
University of British Columbia
Canada
hoos@cs.ubc.ca

Katsumi Inoue
National Institute of Informatics
Japan
ki@nii.ac.jp

Guoqiang Jin
University of Science and Technology of China
abxeeled@mail.ustc.edu.cn

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

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

Sabrina Kirrane
Digital Enterprise Research Institute and Storm Technology
Ireland
sabrina.kirrane@deri.org

Anthony Labarre
Department of Computer Science, KU Leuven, Belgium
anthony.labarre@cs.kuleuven.be

Tiep Le
New Mexico State University
NM, USA
tile@cs.nmsu.edu

Joohyung Lee
Arizona State University
AZ, USA
joolee@asu.edu

Shanshan Liang
Arizona State University
AZ, USA
Shanshan.Liang@asu.edu

Yuliya Lierler
The University of Texas at Austin
TX, USA
yuliya@cs.utexas.edu

Vladimir Lifschitz
University of Texas
TX, USA
vl@cs.utexas.edu

Helena Lindgren
Department of Computing Science, Umeå University
Sweden
helena@cs.umu.se

James B. Lipton
Wesleyan University
USA
jlipton@wesleyan.edu

Nuno Lopes
Digital Enterprise Research Institute
Ireland
nuno.lopes@deri.org

Erez Maharshak
Simon Fraser University
Canada
erez@proxydemocracy.org

Federico Malucelli
Politecnico di Milano, DEI
Italy
malucell@elet.polimi.it

Marco Maratea
DIST, University of Genova
Italy
marco@dist.unige.it

Oliver Matheis
University of Potsdam
Germany
ollbert@gmx.de

Yunsong Meng
Arizona State University
AZ, USA
Yunsong.Meng@asu.edu

Philipp Meyer
Technical University Munich
Germany
meyerphi@in.tum.de

Alessandra Mileo
Digital Enterprise Research Institute
Ireland
alessandra.mileo@deri.org

Emilio Miralles
Simon Fraser University
Canada
emiralle@sfu.ca

Michael Morak
University of Oxford
UK
michael.morak@gmail.com

John Morris
University of Alberta
Canada
morris2@ualberta.ca

Hieu Nguyen
New Mexico State University
NM, USA
nhieu@cs.nmsu.edu

Juan Carlos Nieves
Department of Computing Science, Umeå University
Sweden
jcnieves@cs.umu.se

Maddalena Nonato
University of Ferrara
Italy
maddalena.nonato@unife.it

Vivana Novelli
University of Bath
UK
v.i.novelli@bath.ac.uk

Johannes Oetsch
Vienna University of Technology
Austria
oetsch@kr.tuwien.ac.at

Max Ostrowski
Institut für Informatik, Universität Potsdam
Germany
ostrowsk@cs.uni-potsdam.de

Julian Padget
University of Bath
UK
jap@cs.bath.ac.uk

Andrea Peano
Università degli Studi di Ferrara, EnDiF
Italy
andrea.peano@unife.it

Gilberto Pérez
University of Corunna
Spain
gperez@udc.es

Fernando Pérez Morente
Universidad Complutense de Madrid, DSIC
Spain
fperezmo@fdi.ucm.es

Axel Polleres
Siemens AG Österreich
DERI, National University of Ireland, Galway
Austria and Ireland
axel@polleres.net

Enrico Pontelli
New Mexico State University
NM, USA
epontell@cs.nmsu.edu

Jörg Pührer
Vienna University of Technology
Austria
puehrer@kr.tuwien.ac.at

Luca Pulina
Univ. of Sassari, POLCOMING
Italy
lpulina@uniss.it

Jan Ramon
Department of Computer Science
KU Leuven, Belgium
jan.ramon@cs.kuleuven.be

Juan José Ramos Gonzalez
Universitat Autònoma de Barcelona
Spain
JuanJose.Ramos@uab.es

Christoph Redl
Institute of Information Systems, TU Vienna
Austria
redl@kr.tuwien.ac.at

Francesco Ricca
Department of Mathematics,
University of Calabria, Italy
ricca@mat.unical.it

Taisuke Sato
Tokyo Institute of Technology
Japan
sato@mi.cs.titech.ac.jp

Florian Schanda
Altran Praxis
UK
florian.schanda@altran-praxis.com

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

Sebastian Schaffert
Salzburg Research Forschungsgesellschaft
Austria
sebastian.schaffert@salzburgresearch.at

Marius Schneider
University of Potsdam
Germany
manju@cs.uni-potsdam.de

Nada Sharaf
German University in Cairo
Egypt
nada.hamed@guc.edu.eg

Bryan Silverthorn
The University of Texas at Austin
TX, USA
bsilvert@cs.utexas.edu

Tran Cao Son
New Mexico State University
NM, USA
tson@cs.nmsu.edu

Francesca Spezzano
University of Calabria, DEIS
Italy
fspezzano@deis.unical.it

Peter Stuckey
National ICT Australia, Victoria Laboratory
Australia
peter.stuckey@nicta.com.au

Péter Szeredi
Budapest University of Technology
and Economics, Hungary
szeredi@gmail.com

Naoyuki Tamura
Information Science and Technology Center,
Kobe University, JAPAN
tamura@kobe-u.ac.jp

Paul Tarau
University of North Texas
TX, USA
ptarau@gmail.com

Hans Tompits
Vienna University of Technology
Austria
tompits@kr.tuwien.ac.at

Irina Trubitsyna
University of Calabria, DEIS
Italy
irina@deis.unical.it

Mirek Truszczynski
Computer Science Department,
University of Kentucky
KY, USA
mirek@cs.uky.edu

Joost Vennekens
KU Leuven
Belgium
joost.vennekens@cs.kuleuven.be

Sicco Verwer
Radboud Universiteit Nijmegen,
Institute for Computing and
Information Sciences
Belgium
siccoverwer@gmail.com

Concepcion Vidal
University of Coruña
Spain
concepcion.vidalm@udc.es

Alicia Villanueva
Universitat Politécnica de Valéncia, DSIC
Spain
villanue@dsic.upv.es

Toby Walsh
NICTA and UNSW
Australia
toby.walsh@nicta.com.au

Stefan Woltran
Vienna University of Technology
Austria
woltran@dbai.tuwien.ac.at

Ben Wright
Department of Computer Science, New
Mexico State University
NM, USA
bwright@cs.nmsu.edu

Fangkai Yang
Department of Computer Science,
The University of Texas at Austin
TX, USA
fkyang@cs.utexas.edu

Jia-Huai You
Department of Computing Science,
University of Alberta, Edmonton
Canada
you@cs.ualberta.ca

Antoine Zimmermann
École des Mines de Saint-Étienne
France
antoine.zimmermann@emse.fr

Zsolt Zombori
Budapest University of Technology
and Economics, Hungary
zombori@cs.bme.hu

Introduction to the Technical Communications of the 28th International Conference on Logic Programming Special Issue

Agostino Dovier[1] and Vítor Santos Costa[2]

1 **Dipartimento di Matematica e Informatica**
University of Udine (Italy)
agostino.dovier@uniud.it

2 **CRACS-INESC TEC & Dep. de Ciência de Computadores, FCUP,**
Universidade do Porto, Portugal
vsc@dcc.fc.up.pt

Abstract

We are proud to introduce this special issue of LIPIcs — Leibniz International Proceedings in Informatics, dedicated to the technical communications accepted for the 28th International Conference on Logic Programming (ICLP).

1998 ACM Subject Classification D.1.6 Logic Programming, I.2.3 Deduction and Theorem proving/Logic programming

Keywords and phrases Logic Programming, Organization Details

Digital Object Identifier 10.4230/LIPIcs.ICLP.2012.xvii

The ICLP meetings started in Marseille in 1982 and since then constitute the main venue for presenting and discussing work in the area of logic programming. We contributed to ICLP for the first time in 1991. The first guest-editor had a paper on logic programming with sets, and the second had two papers on the parallel implementation of the Andorra model. Since then, we continued pursuing research in this exciting area and ICLP has always been the major venue for our work. Thus, when the ALP EC committee kindly invited us for chairing the 2012 edition we were delighted to accept.

We particularly appreciate the honor and responsability of organising ICLP in Budapest. Hungary has had a central role both in implementation and in the application of logic programming. Indeed, the role of Hungary in general in Computer Science is widely recognized, and organizing this meeting in the town of John von Neumann, one of the “talent-scouts” of Turing, in the centenary of the birth of the latter, is just another reason for justifying the fact that the fascinating Budapest is the unique town to host ICLP twice.

Publishing the technical communications as LIPIcs paper is a joint initiative taken by the Association for Logic Programming and of the Dagstuhl Research Online Publication Server (DROPS). The goal is to allow a fast preliminary publication for research contributions that are not yet ready for a journal publication but, on the other hand, deserves to be presented at the ICLP. Quality is ensured by an anonymous refereeing process (at least three reviewers per paper), and by an active and very much participating program committee. The approach was first experimented in 2010, and has had favorable feedback since.

This year, ICLP sought contributions in all areas of logic programming, including but not restricted to:

© Agostino Dovier and Vítor Santos Costa;
licensed under Creative Commons License ND

Technical Communications of the 28th International Conference on Logic Programming (ICLP'12).
Editors: A. Dovier and V. Santos Costa; pp. xvii–xxi

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

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

In response to the call for papers we received 102 abstracts, 90 of which remained as complete submissions. Of these, 81 were submitted as full papers and 9 as technical communications. Each paper was reviewed by at least three anonymous program committee members, selected by the program chairs. Sub-reviewers were allowed. After discussion, involving the whole program committee, and a second round of revision for some papers, 20 papers have been selected for immediate journal publication in a special issue of Theory and Practice of Logic Programming (TPLP). 37 papers instead have been judged to deserve a slot for a short presentation at the Meeting and a "technical communication" publication in this Volume of the Leibniz International Proceedings in Informatics (LIPIcs) series, published on-line through the Dagstuhl Research Online Publication Server (DROPS).

The whole set of accepted papers includes 36 technical papers, 12 application papers, 5 system and tool papers, and 4 papers submitted directly as technical communications.

The Conference program was honored to include contributions from three keynote speakers and from a tutorialist. Two invited speakers come from industry, namely Ferenc Darvas from *CompuDrug International, Inc.* Sedona, Arizona, and *ComGrid Kft*, Budapest (two companies using computer science techniques for chemistry), and Mike Elston from *SecuritEase* (an Australian company developing stock brokering tools). Moreover, Jan Wielemaker, of the VU University Amsterdam, presented an history of the first 25 years of SWI Prolog, one of the major (and free) Prolog releases. Tutorialist Viviana Mascardi from University of Genova (Italy) introduced us to the hot topic of "Logic-based Agents and the Semantic Web".

The first ICLP Conference was organized 30 years to this year, in Marseille. During those 30 years, ICLP has been a major venue in Computer Science. In order to acknowledge some of the major contributions that have been fundamental to the success of LP as a field, the ALP executive committee decided that ICLP should recognize the most influential papers presented in the ICLP and ILPS conferences (ILPS was another major meeting in logic programming, organized until 1998), that, 10 and 20 years onwards, have been shown to be a major influence in the field. As program co-chairs of ICLP2012, we were the first to be charged with this delicate task. We included papers from ICLP 1992 and ILPS 1992, 20 years onwards, and of ICLP 2002, 10 years onwards. Our procedure was to use bibliometric information in a first stage, and to use our own personal criteria in a second stage, if necessary. Given that this is the first time this award was given we also considered 1991, and 2001 papers. Although there are an impressive number of excellent papers in 1991 and 1992, one paper emerges with an outstanding record of roughly 600 citations. Further, the paper clearly has a very major influence in the field. The paper is

- Michael Gelfond and Vladimir Lifschitz: Representing Actions in Extended Logic Programming. JICSLP 1992: 559-573

The 10 years onward analysis again produced a group of excellent papers (as expected, the number of citations was stricly less than for 20 years old papers). In this case choosing the winner in a very short list was more difficult. Ackowledging their influence over the very active field of Web Databases and Semantic Web, our selection went to:

- François Bry and Sebastian Schaffert: Towards a Declarative Query and Transformation Language for XML and Semistructured Data: Simulation Unification. ICLP 2002: 255-270

We therefore invited these authors for an invited talk in a special session at the meeting. François Bry and Sebastian Schaffert also contributed to this iussue with a survey paper, entitled *Simulation Unication: Beyond Querying Semistructured Data.*

Since the first edition in 2005, organized by Enrico Pontelli, the Doctoral Consortium has been organized at each ICLP meeting. This event is designed for doctoral students working in areas related to logic programming, with a particular emphasis to students interested in pursuing a career in academia. The Doctoral Consortium aims to provide students with an opportunity to present and discuss their research directions, their thesis proposal, and to obtain feedback from the major experts in the field. This year the doctoral consortium organization has been coordinated by Marco Gavanelli and Stefan Woltran, and seven thesis proposals have been considered deserving of presentation. A survey of these proposals is part of this volume.

Together, this LIPIcs volume and the TPLP special issue constitute the proceedings of ICLP12. The list of the 20 accepted full papers appearing (sorted by alphabetical order) in the corresponding TPLP special issue follows:

- Disjunctive Datalog with Existential Quantifiers: Semantics, Decidability, and Complexity Issues. *Mario Alviano, Wolfgang Faber, Nicola Leone, and Marco Manna*
- Towards Multi-Threaded Local Tabling Using a Common Table Space. *Miguel Areias and Ricardo Rocha*
- Module Theorem for the General Theory of Stable Models. *Joseph Babb and Joohyung Lee*
- Typed Answer Set Programming Lambda Calculus and Corresponding Inverse Lambda Algorithms. *Chitta Baral, Juraj Dzifcak, Marcos Gonzalez, and Aaron Gottesman*
- D-FLAT: Declarative Problem Solving Using Tree Decompositions and Answer-Set Programming. *Bernhard Bliem, Michael Morak, and Stefan Woltran*
- An Improved Proof-Theoretic Compilation of Logic Programs. *Iliano Cervesato*
- Annotating Answer-Set Programs in LANA. *Marina De Vos, Doga Gizem Kisa, Johannes Oetsch, Jörg Pührer, and Hans Tompits*
- SMCHR: Satisfiability Modulo Constraint Handling Rules. *Gregory Duck*
- Conflict-driven ASP Solving with External Sources. *Thomas Eiter, Michael Fink, Thomas Krennwallner, and Christoph Redl*
- Multi-threaded ASP Solving with clasp. *Martin Gebser, Benjamin Kaufmann, and Torsten Schaub*
- Model Checking with Probabilistic Tabled Logic Programming. *Andrey Gorlin, C. R. Ramakrishnan, and Scott Smolka*
- Diagrammatic confluence for Constraint Handling Rules. *Rémy Haemmerlé*

- Inference in Probabilistic Logic Programs with Continuous Random Variables. *Muhammad Islam, C.R. Ramakrishnan, and I.V. Ramakrishnan*
- Relational Theories with Null Values and Non-Herbrand Stable Models. *Vladimir Lifschitz, Karl Pichotta, and Fangkai Yang*
- The Relative Expressiveness of Defeasible Logics. *Michael Maher*
- Compiling Finite Domain Constraints to SAT with BEE. *Amit Metodi and Michael Codish*
- Lightweight Compilation of (C)LP to JavaScript. *Jose F. Morales, Rémy Haemmerlé, Manuel Carro, and Manuel Hermenegildo*
- ASP modulo CSP: The clingcon system. *Max Ostrowski and Torsten Schaub*
- Annotation of Logic Programs for Independent AND-Parallelism by Partial Evaluation. *German Vidal*
- Efficient Tabling of Structured Data with Enhanced Hash-Consing. *Neng-Fa Zhou and Christian Theil Have*

We would like to take this opportunity to acknowledge and thank the other ICLP organisers. Without their work and support this event would not have been possible. We would like to start with the General chair Péter Szeredi (Budapest Univ. of Technology and Economics), and all the organizing chairs, namely the Workshop Chair Mats Carlsson (SICS, Sweden), the Doctoral Consortium Chairs Marco Gavanelli (Univ. of Ferrara) and Stefan Woltran (Vienna University of Technology), the Prolog Programming Contest Chair Tom Schrijvers (Universiteit Gent), the Publicity Chair Gergely Lukácsy (Cisco Systems Inc.), and the Web Manager: János Csorba (Budapest Univ. of Technology and Economics). Thanks also to Alessandro Dal Palù for allowing us to publish his pictures of Budapest on the website. We benefited from material and advice kindly given by last year's program chairs Michael Gelfond and John Gallagher. Thank you very much!

On behalf of the whole LP community, we would like to thank all authors who have submitted a paper, the 41 members of the program committee: Elvira Albert (U.C. Madrid), Sergio Antoy (Portland State Univ.), Marcello Balduccini (Kodak Research Laboratories), Manuel Carro (Technical University of Madrid (UPM)), Michael Codish (Ben Gurion Univ.), Veronica Dahl (Simon Fraser Univ.), Marina De Vos (Univ. of Bath), Alessandro Dal Palù (Universita degli Studi di Parma), Bart Demoen (K.U. Leuven), Thomas Eiter (T.U. Wien), Esra Erdem (Sabanci University), Thom Frῧwirth (Univ. of Ulm), Andrea Formisano (Univ. of Perugia), Maria Garcia de la Banda (Monash Univ.), Marco Gavanelli (University of Ferrara), Hai-Feng Guo (Univ. of Nebraska, Omaha), Gopal Gupta (Univ. of Texas, Dallas), Katsumi Inoue (National Inst. of Informatics, Japan), Angelika Kimmig (K.U. Leuven), Joohyung Lee (Arizona State University), Evelina Lamma (Univ. of Ferrara), Nicola Leone (University of Calabria), Yuliya Lierler (Univ. of Kentucky), Boon Thau Loo (Univ. of Pennsylvania), Michael Maher (R.R.I., Sydney), Alessandra Mileo (DERI Galway), Jose Morales (U.P. Madrid), Enrico Pontelli (New Mexico State Univ.), Gianfranco Rossi (Univ. of Parma), Beata Sarna-Starosta (Cambian, Vancouver), Torsten Schaub (Univ. of Potsdam), Tom Schrijvers (Universiteit Gent), Fernando Silva (Univ. of Porto), Tran Cao Son (New Mexico State University), Terrance Swift (Univ. Nova de Lisboa), Péter Szeredi (Budapest Univ. of Technology and Economics), Francesca Toni (Imperial College London), Mirek Truszczynski (University of Kentucky), Germán Vidal (U.P. of Valencia), Stefan Woltran (Vienna University of Technology), and Neng-Fa Zhou (CUNY, New York).

A particular thanks goes to the 96 external referees, namely: Alicia Villanueva, Amira Zaki, Ana Paula Tomás, Andrea Bracciali, Antonis Bikakis, Antonis Kakas, Brian Devries, C. R. Ramakrishnan, Chiaki Sakama, Christoph Redl, Christopher Mears, Dale

Miller, Daniel De Schreye, Daniela Inclezan, David Brown, Demis Ballis, Dimitar Shterionov, Dragan Ivanovic, Evgenia Ternovska, Fabio Fioravanti, Fabrizio Riguzzi, Fangkai Yang, Fausto Spoto, Feliks Kluźniak, Francesco Calimeri, Francesco Ricca, Fred Mesnard, Gianluigi Greco, Giovanni Grasso, Gregory Duck, Gregory Gelfond, Inês Dutra, Jesus M. Almendros-Jimenez, Joost Vennekens, Juan Manuel Crespo, Julio Mariño, Kyle Marple, Marco Alberti, Marco Maratea, Mario Alviano, Mário Florido, Marius Schneider, Martin Gebser, Masakazu Ishihata, Massimiliano Cattafi, Matthias Knorr, Maurice Bruynooghe, Max Ostrowski, Michael Bartholomew, Michael Hanus, Michael Morak, Minh Dao-Tran, Mutsunori Banbara, Naoki Nishida, Naoyuki Tamura, Neda Saeedloei, Nicola Capuano, Nicolas Schwind, Noson Yanofsky, Nysret Musliu, Orkunt Sabuncu, Pablo Chico De Guzmán Paolo Torroni, Paul Tarau, Peter James Stuckey, Peter Schüller, Philipp Obermeier, Puri Arenas-Sanchez, Rémy Haemmerlé, Rafael Del Vado Virsela, Ricardo Rocha, Richard Min, Robert Craven, Roland Kaminski, Samir Genaim, Sandeep Chintabathina, Santiago Escobar, Sara Girotto, Sean Policarpio, Simona Perri, Slim Abdennadher, Sofia Gomes, Stefania Costantini, Stefano Bistarelli, Thomas Krennwallner, Thomas Ströder, Tomoya Tanjo, Torben Mogensen, Umut Oztok, Valerio Senni, Victor Marek, Victor Pablos Ceruelo, Wolfgang Dvořák, Wolfgang Faber, Yana Todorova, and Yunsong Meng.

Throughout this period, we could always rely on ALP. Our gratitude goes to the ALP president Gopal Gupta, to the Conference chair Manuel (Manolo) Carro, and to all the ALP Executive committe members. We already thanked the invited speakers and the tutorialist above, but we would like to stress here our thank to them. A particular thank goes to Marc Herbstritt from Dagstuhl, for the support in publication of this special issue. Similarly, David Tranah and Ilkka Niemelä deserve our thanks for their kindness and their precious support in all TPLP publication stages.

Our thanks also go to the the sponsors of the meeting, namely the Association for Logic Programming (ALP), the Artificial Intelligence Section of the John von Neumann Computer Society, the Aquincum Institute of Technology (AIT) of Budapest, Alerant System Inc, and Google (female researchers grant). VSC would like to acknowledge funding by the ERDF (European Regional Development Fund) through the COMPETE Programme and by FCT (Portuguese Foundation for Science and Technology) within projects HORUS (PTDC/EIA-EIA/100897/2008) and LEAP (PTDC/EIA-CCO/112158/2009). Finally, a well-deserved thank you goes to Easychair developers and managers. This amazing free software allowed us to save days of low level activities. Similarly, the joint work of the two co-chairs would have been extremely more difficult and expensive without the Dropbox and Skype services.

September 2012
Agostino Dovier and Vítor Santos Costa
Program Committee Chairs and Guest Editors

Simulation Unification: Beyond Querying Semistructured Data

François Bry[1] and Sebastian Schaffert[2]

1 **Institute for Informatics, Ludwig-Maximilians University of Munich, Germany**
http://pms.ifi.lmu.de

2 **Salzburg Research Forschungsgesellschaft, Austria**
http://www.salzburgresearch.at/

Abstract

This article first reminds of simulation unification, a non-standard unification proposed at the 18th International Conference on Logic Programming (ICLP 2002) for making logic programming capable of querying semistructured data on the Web. This article further argues that, beyond querying semistructured data on the Web, simulation unification has a potential for Web querying of multimedia data and semantic metadata and for Web searching of data of all kinds.

1998 ACM Subject Classification D3.3 Language Constructs and Features

Keywords and phrases Simulation Unification, (Semantic) Web Querying

Digital Object Identifier 10.4230/LIPIcs.ICLP.2012.1

1 Introduction

This article is devoted to simulation unification, a non-standard unification which has been introduced in 2002 at the 18th International Conference on Logic Programming (ICLP 2002) with the article titled “Towards a Declarative Query and Transformation Language for XML and Semistructured Data: Simulation Unification” [15] and the long version [16] of that article. Simulation unification has been specified in more detail two years later, in 2004, in the doctoral thesis “Xcerpt: A Rule-Based Query and Transformation Language for the Web” [30] of Sebastian Schaffert.

This article recalls simulation unification and argues that it has a so far unexploited potential for Web querying of multimedia and semantic data as well as for Web searching of data of all kinds.

This article is structured as follows. After this introduction, Section 2 describes the context in which and why we developed simulation unification. Section 3 is a brief, and simplified, reminder of simulation unification. Section 4 is devoted to works related to simulation unification. Section 5 discusses how simulation unification could be applied to querying multimedia and semantic data and to searching. Section 6 is a conclusion.

2 What Led to Simulation Unification

At the beginning of the 90es of the 20th century, as the Web became a common medium, many computer scientists first did not fully realised what impact the Web would have on their areas of research. This was the case amongst others of the query answering community. At the end of the 90es, that community hastily investigated Web query languages, what resulted in XQuery [10], a “recommendation” of the W3C, so as to keep an hold on data access. This community celebrated XQuery amongst others for its roots in functional programming,

© François Bry and Sebastian Schaffert;
licensed under Creative Commons License ND
Technical Communications of the 28th International Conference on Logic Programming (ICLP'12).
Editors: A. Dovier and V. Santos Costa; pp. 1–13

Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

and promoted it as a worthy descendant of SQL [20], the query language which had greatly contributed to the success of relational databases.

We took the enthusiasm for XQuery with a skepticism rooted at a logic programming practice. We found that XQuery was difficult to program with; we thought it would often yield inelegant and therefore costly to maintain programms and we guessed that it would require complicated runtime systems. Since, these intuitions have been amply confirmed and XQuery is no longer the subject of much enthusiasm.

The study reported about in [28] had hinted at the potential of logic programming for querying semistructured data. That article shows that restricting XPath [21], the data selection sub-language of XQuery, to its forwards axes, that is, to so-called Forward XPath, does not restrict the data selection language's expressivity. Since a Forward XPath expression basically amounts to a logic atom, a link between logic programming an Web querying was established. The afore mentioned article [28] has received some attention because it makes it possible to restrict formal investigations on XPath to XPath Forward what gives rise to significant simplifications. Surprisingly, that the restriction to XPath Forward also, and for the same reasons, gives rise to simpler queries and therefore eases both, programming and query evaluation, has been rarely noticed.

Pattern-based queries for Web data had been proposed with the Web query languages UnQL [19] and XML-QL [22] what suggested a full unification binding variables in the two terms considered instead of a pattern matching binding variables in only in the pattern.

These two observations led us to simulation unification, a technique that makes logic programming as convenient for querying semistructured data as for querying relational data.

Since, search engines, other tools the importance of which has not been immediately understood within the query answering community, have considerably reduced the need for Web query languages. Indeed, data are no longer only queried for but also, and mostly, search for. In this article, we arge that, beyond querying semistructured data, simulation unification also has a potential for both, Web querying of multimedia and semantic data and Web searching of data of all kinds.

3 What is Simulation Unification?

Given two terms t_1 and t_2 simulation unification [15, 16, 30] determines, if possible, a most general unifier σ for the variables in t_1 and t_2 such that every ground instances of $t_1\sigma$ simulates in a ground instance of $t_2\sigma$.

Simulation unification is based on an adaption of graph simulation to terms aimed at representing, selecting (or querying) and constructing XML data. Simulation unification's principles are relatively simple. The syntactical richness necessary for an easy expression of data selections and construction makes it, however, complicated.

In the remainder of the current section 3, rooted graph simulation is introduced in Section 3.1, database terms, query terms and construct terms in Section 3.2, term simulation and answers to query terms in Section 3.3 and simulation unification in Section 3.4.

3.1 Rooted Graph Simulation

Simulation, also called graph simulation, has been studied in [26, 27]. A term t_1 (seen as a graph G_1) simulates in a term t_2 (seen as a graph G_2) if there is a mapping of the nodes of G_1 (that is of the subterms of t_1) in the nodes of G_2 (that is, the subterms of t_2) which preserves the edges (that is, subterm nesting). Simulation is similar to, though more general

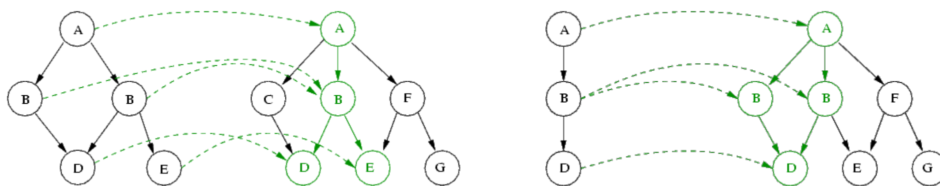


Figure 1 Two simulations (with respect to node label equality) [15].

than, graph homomorphism because it allows two nodes of the one graph being mapped to a single node of the other graph and vice versa.

In general, there might be more than one simulation between two graphs. Therefore, so-called minimal simulations are considered.

Figure 1 from [15] gives two examples of simulations. In each of these two examples a node of the left graph is mapped into a node of the right graph if their labels are identical. Such simulations are simulations with respect to label identity. More generally, a simulation can be defined with respect to any preorder relation (amongst other order and equivalence relations). In Section 5, we argue that considering other relations than label equality makes simulation unification convenient for querying multimedia and semantic data on the Web as well as for searching for data of all kinds on the Web.

The following definition from [30] which refines that of [15] is inspired from [26, 27]. A (directed) rooted graph $G = (V, E, r)$ consists in a set V of vertices (or nodes), a set E of edges (that is, ordered pairs of vertices), and a selected vertex r, called the root of G, from which each vertex of G is accessible.

▶ **Definition 1** (Rooted graph simulation with respect to a preorder relation $\sim$ [30]). Let $G_1 = (V_1, E_1, r_1)$ and $G_2 = (V_2, E_2, r_2)$ be rooted graphs and $\sim \subseteq V_1 \times V_2$ a preorder relation. A relation $\mathcal{S} \subseteq V_1 \times V_2$ is a *rooted simulation* of G_1 in G_2 with respect to $\sim$ if:

1. $r_1 \ \mathcal{S} \ r_2$.
2. If $v_1 \ \mathcal{S} \ v_2$, then $v_1 \sim v_2$.
3. If $v_1 \ \mathcal{S} \ v_2$ and $(v_1, v_1', i) \in E_1$, then there exists $v_2' \in V_2$ such that $v_1' \ \mathcal{S} \ v_2'$ and $(v_2, v_2', j) \in E_2$

A rooted simulation $\mathcal{S}$ of G_1 in G_2 with respect to $\sim$ is *minimal* if there are no rooted simulations $\mathcal{S}'$ of G_1 in G_2 with respect to $\sim$ such that $\mathcal{S}' \subset \mathcal{S}$ (and $S \neq S'$).

Graph simulation conveys well how the Web is queried. Web queries are mostly incomplete specifications of data striven for that are convenientely answered by data items containing more than the query specifies and allowing that distinct parts of the query are answered by the same data. The relevance of graph simulation for Web querying has been first pointed out in [19, 22].

3.2 Database Terms, Query Terms and Construct Terms

3.2.1 Database terms

Database terms are an abstraction of XML documents and a generalisation of the ground terms of logic. Database terms are similar to logic ground terms except that the arity of a function symbol, called "label", is not fixed but variable, and that the order of the arguments of a function symbol might not be compelling.

A database term whith a root labelled l and ordered children $t_1, \ldots, t_n$ is denoted $l[t_1, \ldots, t_n]$. A databasse term with a root labelled l and unordered children $t_1, \ldots, t_n$ is denoted $l\{t_1, \ldots, t_n\}$.

Cyles, possible in XML documents though hypertext links and ID-IDREF references, are allowed in database terms but not considered in the following for the sake of briefness. A database terms without cycles can be seen as a tree, a database term with cycles as a rooted graph.

3.2.2 Query terms

Query terms are patterns specifying selections of ground terms terms. They are similar to logic atoms except that they can express incompleteness in breath and depth and that a variable X in a query term can be restricted.

In a query term,

- the brackets [ ] and { } require answers with no more ordered respectively unordered subterms than the query term;
- double brackets [[ ]] and {{ }} accept answers having more ordered respectively unordered subterms than the query term;
- a variable X can be constrained to some query terms Q using $X \leadsto Q$, where $\leadsto$ is read “as";
- $X \leadsto$ *desc* t, read “X descendant t”, is used to express that X is bound to a term containing a subterm t at an unspecified depth.

Multiple constraints for a a same variable are allowed. Figure 2 hints at the semantics of query terms formally specified in [17, 18, 30].

Constraining variables (with $\leadsto$) might results in cyclic constraints that cannot be answered by database terms because database terms are finite. A variable X is said to depend on a variable Y in a query term t if $X \leadsto t_1$ is a subterm of t and Y is a subterm of t_1. A query term t is said to be variable well-formed if it contains no variables $X_0, \ldots, X_n$ $(n \geq 1)$ such that $X_0 = X_n$ and for all $i = 1, \ldots n$ X_i depends on X_{i-1} in t. Only variable well-formed query terms are considered in the following.

A query term is ground if it contains no variables (and therefore no $\leadsto$ and no desc).

Further constructs such as “option” and “except” might occur in query terms so as to ease the expression of some queries [30, 11, 31]. They are not considered in the following for the sake of briefness.

3.2.3 Construct terms

Construct terms serve to re-assemble the values which are specified in query terms by variables, so as to form new database terms. Thus,[ ], { } and variables may occur in construct terms but neither [[ ]], nor {{ }}, nor $\leadsto$. In a construct term, a variable might be preceded by “all” meaning that all values, or bindings, for this variables are to be gathered.

Rules combine construct terms and query terms in the manner of logic programming: A rule head is a construct term; a rule body is build up from query terms, conjunctions, disjunctions, and negations.

Like in [15], simulation unification is defined below under the simplifying assumption that {} and {{}} are the only kinds of braces. The complete definition is given in [30].

Query terms	Possible answers	No answers
$a[[b, c\{d, e\}, f]]$	$a[b, c\{d, e, g\}, f]$ $a[b, c\{d, e, g\}, f\{g, h\}]$ $a[b, c\{d, e\{g, h\}, g\}, f\{g, h\}]$ $a[b, c[d, e], f]$	$a\{b, c\{d, e\}, f, g\}$ $a[b, c\{d, e\}, f, g]$ $a\{b, c\{d, e\}, f\}$
$a[desc\ f[c, d], b]$	$a[f[c, d], b]$ $a[g[f[c, d]], b]$ $a[g[f[c, d], h], b]$ $a[g[g[f[c, d]]], b]$ $a[g[g[f[c, d], h], i], b]$	$a[b]$ $a[g, b[f[c, d]]]$
$a[X \leadsto b[c, d], Y, e]$	$a[b[c, d], f, e]$ X *bound to* $b[c, d]$ Y *bound to* f $a[b[c, d], f[g, h], e]$ X *bound to* $b[c, d]$ Y *bound to* $f[g, h]$	$a[c, f, e]$ $a[b[c], f, e]$ $a[h[b, c], f, e]$
$a\{X \leadsto b\{c\}, X \leadsto b\{d\}\}$	$a\{b\{c, d\}\}$ X *bound to* $b\{c, d\}$	$a\{b\{c\}\}$ $a[b[c], f, e]$
$a[X \leadsto b\{c\}, X \leadsto f\{d\}]$	*none*	$a[b\{c\}]$ $a[f\{d\}]$ $a[b\{c\}, f\{d\}]$
$a\{\{\}\}$	a	$a\{b\}$ $a\{b, c\}$ $a[b]$ $a[b, c]$

Figure 2 Query terms.

3.3 Term Simulation and Answers

Substitutions and grounding substitutions are defined as usual except that they assign construct terms, but no query terms, to variables. Instances and ground instances of query and construct terms are defined as usual except that an instance of $X \leadsto t$ is defined as an instance of X (that is, ignoring $\leadsto t$). $\leadsto$ and desc induce constraints on variables and subterms of a query term. Instances of a query that fulfill these constraints are called allowed instances. Only allowed instances are considered in the following.

Simulation of a graph G_1 into a graph G_2 is adapted into the simulation of a ground query term Q into a ground construct term t by paying the necessary attention to the brackets $\{\}$ and $\{\{\}\}$. Ground term simulation is then extended to query and construct terms with variables as follows: A query term Q simulates into a construct term t, denote $Q \preceq t$, if there exists a substitution σ such that every ground instance of $Q\sigma$ simulates into a ground instance of $t\sigma$.

An answer to a query term Q is a database term t such that an allowed instance of Q simulates in t. As usual, substitution (so-called answer substitutions) are associated with term answers. Because of the construct desc serving to express subterm constraints and in contrast to classical logic programming, answers cannot be fully defined by answer substitutions.

3.4 Simulation Unification

Simulation unification is a non-deterministic algorithm for solving equations of the form $Q \preceq t$, read Q simulates in t, on query terms Q and construct terms t. It is based on the following term decomposition rules – see [15, 16, 30] for a detailed description of the non-deterministic algorithm. The outcome of simulation unification, if it succeeds, is a finite set of substitutions called simulation unifier.

▶ **Definition 2** (Term Decomposition Rules). Let l (with or without indices) denote a label. Let t^1 and t^2 (with or without indices) denote query terms.

- **Root Elimination:**

  (1) $l \preceq l\{t^2_1, \ldots, t^2_m\} \Leftrightarrow true$ if $m \geq 1$
  $l \preceq l\{\{\}\} \Leftrightarrow true$

  (2) $l\{t^1_1, \ldots, t^1_n\} \preceq l \Leftrightarrow false$ if $n \geq 1$
  $l\{t^1_1, \ldots, t^1_n\} \preceq l\{\{\}\} \Leftrightarrow false$ if $n \geq 1$

  (3) Let Π be the set of (total) functions $\{t^1_1, \ldots, t^1_n\} \to \{t^2_1, \ldots, t^2_m\}$:
  $l\{t^1_1, \ldots, t^1_n\} \preceq l\{t^2_1, \ldots, t^2_m\} \Leftrightarrow \bigvee_{\pi\in\Pi} \bigwedge_{1\leq i\leq n} t^1_i \preceq \pi(t^1_i)$
  if $n \geq 1$ and $m \geq 1$

  (4) $l_1\{t^1_1, \ldots, t^1_n\} \preceq l_2\{t^2_1, \ldots, t^2_m\} \Leftrightarrow false$ if $l_1 \neq l_2$ and $n \geq 0$ and $m \geq 0$

- **$\rightsquigarrow$ Elimination:**
  $X \rightsquigarrow t^1 \preceq t^2 \;\Leftrightarrow\; t^1 \preceq t^2 \;\wedge\; t^1 \preceq X \;\wedge\; X \preceq t^2$
- **Descendant Elimination:**
  $desc\ t^1 \preceq l_2\{t^2_1, \ldots, t^2_m\} \Leftrightarrow t^1 \preceq l_2\{t^2_1, \ldots, t^2_m\} \vee \bigvee_{1\leq i\leq m} desc\ t^1 \preceq t^2_i$
  if $m \geq 0$

Simulation unification is sound and complete for the notion of answer recalled above [15, 16, 30] . Like standard unification, simulation unification is symmetric since it can bind variables in the two terms considered. Unlike standard unification, however, it is asymmetric in the sense that it does not make the two terms considered equal, but instead makes the one simulate into the other what in general is no symmetrical relationship.

4 Work Related to Simulation Unification

How simulation unification relates to classical involved forms of unifications is addressed in [15] as follows:

> Several unification methods have been proposed that, like simulation unification, process flexible terms or structures, notably feature unification [1, 34] and associative-commutative-unification, short AC-unification, [23]. Simulation unification differs from feature unification in several aspects (discussed in [16]). Simulation unification might remind of theory unification [2]. The significant difference between both is that simulation unification is based upon an order relation, while theory unification refers to a congruence relation.

Simulation unification offers a decidable alternative to equational unification [2]. We argue in the following Section 5 that novel forms of simulation unification based on verious

embedding or similarity relationships would be useful in querying multimedia and semantic data as well as in searching data of all kinds.

Simulation unification has been developed for the textual query language and its visual companion visXcerpt. Between 2002 and 2006, research prototypes of Xcerpt and visXcerpt have been presented at database, Web, Semantic Web, logic programming and visual programming conferences [14, 6, 7, 31, 3, 13, 4, 5].

A subsumption referring to simulation unification, called simulation subsumption, and its use for query optimization have been introduced in [12]. Simulation subsumption expreses query containement for queries based on simulation unification. Simulation subsumption is useful for the query optimization, in particular for verifying the termination of recursive queries.

5 Beyond Querying Semistructured Data

Since the publication of our original article in 2002, the Web has undergone several major developments. First, the Semantic Web effort with its underlying technologies RDF and OWL has gained much momentum with the emergence of "Linked Data" as a means to publish semi-structured data using a uniform model for data representation and interlinking between datasets. Second, while the Web of 2002 was still mainly a static, text-based Web, the Web of 2012 is interactive and mostly consists of multimedia content. And third, with the success of Social Software, the amount of content and data on the Web has grown tremendously, making effective and efficient Web search more and more important. In the following, we will briefly describe how our ideas concerning simulation unification are more important than ever for addressing typical problems in these areas.

5.1 Generalising Simulation Unification

Simulation unification gives rise to queries retrieving structural sub-patterns within XML data. This can be generalized to other kind of data in two complementary ways:

- The first generalisation would build upon an "embedding" relationship on the data considered which, like simulation unification, would not be symmetric.
- The second generalisation would build upon a "similarity" relationship on the data considered which, in contrast to simulation unification, would be symmetric.

In the following sections, we describe how these two generalisations could help addressing open problems in several other areas.

5.2 RDF, RDFS and OWL

The Resource Description Framework, or "RDF" [37], is the primary model for publishing data on the Semantic Web. At its core, it defines a graph model where vertices represent Web resources (identified by URIs) or literal values and edges (so-called "triples") represent typed relations between Web resources. RDF also defines a number of different serialization formats for this graph data, e.g. RDF/XML, Turtle, or N-Triples. Schema information about an RDF graph can be defined using the schema languages RDFS [36] or OWL [35]. Both are capable of representing ontological knowledge about the schema in addition to specifying possible relations and are based on some form of logics.

Querying RDF Data. An important aspect of RDF is querying the graph data contained in a dataset. Typical RDF queries for example express RDF subgraphs to be found in the data queried. Given the graph model underlying RDF, pattern-based querying

of RDF is natural. In fact, the most widely used RDF query language SPARQL [39] uses so-called “triple patterns” specifying edges to look for in the dataset. Variables in triple patterns are bound to values when matching a pattern in the same style as in other logic programming languages.

While SPARQL is already a well designed and widely established query language, it is currently only defined in terms of a query algebra similar to the *relational algebra* and is not offering a *declarative calculus*. A query approach based on *(unrooted) simulation unification* could provide such a calculus for SPARQL in a style similar to the relational calculus behind SQL and Datalog. It would thus allow for a more declarative semantics and advanced reasoning services over RDF by opening up RDF querying to logic programming approaches. Note that this would also give rise to expressing RDFS and OWL ontology semantics in terms of logic programming rules. Querying RDF data corresponds to the “embedding relationship” of simulation unification described above.

Matching RDF Datasets. On the Semantic Web with many independent data publishers a common challenge is so-called “schema alignment” or “data alignment”. In schema (or data) alignment, the goal is to create mappings between two different schemas (or datasets) to allow better interoperability and exchange of the data. A common way of doing schema alignment is to map concepts from the two schemas that are “similar” regarding different criteria.

For example, both schemas might define their own “Student” concept but with slightly different properties:

- Schema 1 defines a *Student* with *full name*, *email* and *inscription number*
- Schema 2 defines a *Student* with *first name* and *last name*, as well as *email*

Schema alignment could map between the Students of Schema 1 and 2 based on the name of the concept, the shared property *email*, and the similarity between *first name*/*last name* on the one hand and *full name* on the other hand.

A lot of research has been undertaken to investigate automatic means to carry out schema alignment. Nevertheless, many problems in this area today remain unsolved [33]. When representing the different attributes of a concept in terms of a graph structure, simulation unification in the second generalisation (“similarity relationship”) could offer a new option for identifying similar concepts by trying to find a maximal simulation between the graphs representing two concepts.

5.3 Linked Data

Linked Data [8, 25] is a recent development within the Semantic Web effort to publish datasets of various sizes on the Web for anyone to use and combine, using the technologies developed in the Semantic Web context (mainly URIs and RDF). In his initial announcement, Tim Berners-Lee described four “Linked Data Principles” [8]:

1. Use URIs as names for things
2. Use HTTP URIs so that people can look up those names.
3. When someone looks up a URI, provide useful information, using the standards (RDF, SPARQL)
4. Include links to other URIs. so that they can discover more things.

Since then, numerous datasets have been made available under these principles. As of September 2011, the known part of the “Web of Data” consists of about 300 datasets from various domains with more than 30 billion triples. Moreover, these datasets are connected with each other with about 500 million RDF links [9].

Conceptually, the Linked Data Cloud (or "Web of Data") can be seen as an RDF graph structure over distributed information systems. Since no dataset has information about the full graph, non-local parts can only be discovered by following RDF links that span across different servers.

Even though resources on Linked Data servers are typically interlinked and thus conceptually integrate data from many different sources, querying such data is still very cumbersome. The main reason is that existing query languages for RDF like SPARQL are rather dataset-centric and do not easily query over distributed or even unknown sources. There are currently four approaches to address this issue:

- a central index harvests the Web for RDF data and stores it in a central repository and offers it for querying, e.g. using SPARQL. This approach is followed e.g. by Sindice,[1] which offers a public SPARQL endpoint.
- a query is distributed over several query endpoints and the results are then combined. This approach is proposed in the SPARQL 1.1 Federation Extension [38].
- accepting the incompleteness of the results returned by the query and trying to improve the recall by different heuristics, as proposed e.g. by Hartig et.al. [24]

The first two approaches have obvious disadvantages: a central repository is not always recent and a single point of failure, while explicit federated queries are cumbersome to write and need exact information on how and where to access the SPARQL endpoint. They also require that all queried servers implement the SPARQL 1.1 Federation Extensions. The third approach is in our opinion not very user friendly, since the user cannot easily determine whether the results he will get are complete or not and important enterprise decisions might depend on that information.

In [32], we therefore proposed a path-based approach that is more suitable for querying Linked Data. However, a path language only allows binding one variable at a time (the "end" of the path) and is therefore rather limited in its expressivity and performance. Rooted simulation unification as described previously for XML could give rise to a novel kind of query language for Linked Data that does not share the problems of SPARQL and goes beyond the expressivity of simple path navigations. A query pattern could use a context resource from a local dataset as query root, follow links to other remote datasets and then bind multiple variables at the same time, reducing the number of network requests and providing a convenient way for formulating a query.

As an example, consider users publishing their basic profile information using the FOAF (friend-of-a-friend) vocabulary. Each user publishes on his website an RDF file with his name and email address, as well as links (`foaf:knows`) to the FOAF files of his friends. A query based on simulation unification could then select the first name, last name and email addresses of each friend in a single query by starting at the local FOAF file, following `foaf:knows`, and binding the three variables at the same time. Such a query is currently neither possible using SPARQL nor using a path-based approach.

5.4 Multimedia

An embedding realtionship can be specified for multimedia data expressing that, for example, a given visual pattern can be found in a picture or in a video. Such a relationship can be defined in terms of either geometrical image recognition algorithms, of features extracted from the multimedia content, or of symbolic metadata associated with picture. Rather

[1] `http://sindice.com/`

different embedding relationships can be thought of that would fullfil the needs of different applications. For example:

- Existing metadata could be queried. In the scenario described in [32], we are working with cliff-diving videos provided by Red Bull that are accompanied with precise descriptions of the scene, transcripts of interviews, as well as music cue sheets and general metadata about a video (persons, locations, editor, description). A query based on Simulation Unification could query for all videos with a certain person at a certain location.
- Multimedia information extraction could automatically extract faces of persons (e.g. using an Eigenfaces algorithm) as well as prominent structures (e.g. edges with sharp contrast) from a large collection of images and videos and store them as features. Simulation Unification could be used to provide a query with some sample features (the face of a person and a tower in the background) and the be evaluated over the image collection to retrieve matching images.

Similarity relationships are often used in retrieving multimedia data. Indeed, multimedia applications require to retrieve images similar to some given images. Image similarity can be specified in many different manners, with and without simliarity threshold to be fulfiled by the selected data.

5.5 Web Search and Enterprise Search

With the tremendous increase in content, Web Search and Enterprise Search are nowadays the most important way of finding and accessing information. The most important difference between Web Search and Enterprise Search is that Web Search can make use of the hyperlinks between documents (e.g. in Google's PageRank [29]) and the novelty of documents, while enterprise content is typically not connected and novelty is not necessarily a good measure for relevance.

Web Search and Enterprise Search could benefit from generalisations of simulation based on emebedding or similarity relationships in the following typical search tasks:

- *Search:* Both Web and Enteprise Search build in its core build upon the occurring of a words, or phrase, or of an ordered list of words or phrases in documents. Such a relationship could be replaced by embedding or similarity relationships for multimedia or semantic data of the afore mentioned kind. This would result in multimedia and semantic search engines at the the cost of indexing a well-chosen selection of patterns. For example, this would allow searches like “the fantasy book with the blue cover”.
- *Grouping:* Search results often contain many similar documents, e.g. different versions of the same document in an enterprise setting. When displaying the search results, such documents should be grouped and displayed together. Detecting such groups can be a very hard task. A simulation unification for similarity relationships could be used for clustering similar documents based on various document properties.
- *Ranking:* Ranking of search results in the result list is the real art of search engines. For example, Google considers over 300 features in their ranking algorithm to determine the relevance of documents with respect to the search query and the user context (e.g. location, previous searches, social networking profile). When so many aspects are taken into account, simulation unification could provide a conceptual framework for calculating the similarity between search results and the query and user context.

6 Conclusion

Thias article has first recalled what led its authors to develop simulation unification for querying semistructured data on the Web. Simulation unification has been presenteed in 2002 at the 18th International Conference on Logic Programming (ICLP 2002) [15], in the long version [16] of that article, and in more detail in the doctoral thesis [30].

This article then has given a brief reminder of simulation unification as presented in the afore mentioned ICLP 2002 article.

Finally, this article has suggested novel directions for Web and Semantic Web research building upon the idea of simulation unification and generalising it in various manners.

Generalising simulation unification as suggested in this article would anchor logic programming in promising fields of research of considerable practical importance: Querying and Web search for multimedia and semantic data.

Acknowledgements

The authors are thanksful to the program comittee of the 28th International Conference on Logic Programming (ICLP 2012) and to its chairs, Agostino Dovier and Vitor Vitor Santos Costa, for having selected their article [15] amongst the mot influencial logic programming publications of the last decade and for their invitation to present the content of the present article at ICLP 2012.

References

1. Hassan Aït-Kaci, Andreas Podelski, and Seth Copen Goldstein. Order-Sorted Theory Unification. Technical report, digital – Paris Research Laboratory, 1993.
2. Franz Baader and Wayne Snyder. Unification theory. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*. Elsevier Science Publishers, 1999.
3. Sacha Berger, François Bry, Oliver Bolzer, Tim Furche, Sebastian Schaffert, and Christoph Wieser. Xcerpt and visXcerpt: Twin Query Languages for the Semantic Web (Demonstration). In *Proceedings of 3rd International Semantic Web Conference (ISWC)*, 2004. `http://pms.ifi.lmu.de/publikationen/PMS-FB/PMS-FB-2004-23.pdf`.
4. Sacha Berger, François Bry, Oliver Bolzer, Tim Furche, Sebastian Schaffert, and Christoph Wieser. Querying the standard and Semantic Web using Xcerpt and visXcerpt (Demonstration). In *Proceedings of the 2nd European Semantic Web Conference (ESWC)*, 2005.
5. Sacha Berger, François Bry, Tim Furche, Benedikt Linse, and Andreas Schröder. Beyond XML and RDF: The Versatile Web Query Language Xcerpt (Poster Paper). In *Proceedings of 15th International World Wide Web Conference (WWW)*, pages 1053–1054, 2006. `http://pms.ifi.lmu.de/publikationen/PMS-FB/PMS-FB-2006-21/PMS-FB-2006-21.pdf`.
6. Sacha Berger, François Bry, and Sebastian Schaffert. A Visual Language for Web Querying and Reasoning. In *Proceedings of Workshop on Principles and Practice of Semantic Web Reasoning (PPSWR)*, number 2901 in LNCS. Springer, 2003. `http://pms.ifi.lmu.de/publikationen/PMS-FB/PMS-FB-2003-6.pdf`.
7. Sacha Berger, François Bry, Sebastian Schaffert, and Christoph Wieser. Xcerpt and visXcerpt: From Pattern-Based to Visual Querying of XML and Semistructured Data (Demonstration). In *Proceedings of 29th Intl. Conference on Very Large Data Bases (VLDB)*. `http://pms.ifi.lmu.de/publikationen/PMS-FB/PMS-FB-2003-2.pdf`.
8. Tim Berners-Lee. Linked Data, 2006.

9 Chris Bizer, Anja Jentzsch, and Richard Cyganiak. State of the lod cloud. Technical report, Freie Universität Berlin / DERI, NUI Galway, http://www4.wiwiss.fu-berlin.de/lodcloud/state/, September 2011.

10 Scott Boag, Don Chamberlin, Mary F. Fernández, Daniela Florescu, Jonathan Robie, and Jérôme Siméon. *XQuery 1.0: An XML Query Language (Second Edition)*. W3C Recommendation, December 2010. `http://http://www.w3.org/TR/xquery/`.

11 Oliver Bolzer, François Bry, Tim Furche, Sebastian Kraus, and Sebastian Schaffert. Development of Use Cases, Part I – Illustrating the Functionality of a Versatile Web Query Language. Technical report, Institute for Informatics, Ludwig-Maximilian University of Munich, 2004. `http:/pms.ifi.lmu.de/publikationen/PMS-FB/PMS-FB-2005-23.pdf`.

12 François Bry, Tim Furch, and Benedikt Linse. Simulation Subsumption or Déjà vu on the Web. In Diego Calvanese and Georg Lausen, editors, *Proceedings of the 2nd International Conference on Web Reasoning and Rule Systems (RR)*, number 5341 in LNCS, pages 28–42. Springer, 2008. `http://pms.ifi.lmu.de/publikationen/PMS-FB/PMS-FB-2008-8/PMS-FB-2008-8-paper.pdf`.

13 François Bry, Paula-Lavinia Pătrânjan, and Sebastian Schaffert. Poster Presentation: Xcerpt and XChange: Logic Programming Languages for Querying and Evolution on the Web. In *Proceedings of the 20th International Conference on Logic Programming (ICLP)*, number 3132 in LNCS. Springer, 2004. `http://pms.ifi.lmu.de/publikationen/PMS-FB/PMS-FB-2004-11.pdf`.

14 François Bry and Sebastian Schaffert. A Gentle Introduction into Xcerpt, a Rule-based Query and Transformation Language for XML. In *Proceedings of International Workshop on Rule Markup Languages for Business Rules on the Semantic Web (RuleML)*, 2002. `http:/pms.ifi.lmu.de/publikationen/PMS-FB/PMS-FB-2002-11.pdf`.

15 François Bry and Sebastian Schaffert. Towards a Declarative Query and Transformation Language for XML and Semistructured Data: Simulation Unification. In Peter J. Stuckey, editor, *Proceedings of the 18th International Conference on Logic Programming (ICLP)*, number 2401 in LNCS, pages 255–270. Springer, 2002. Short version of [16], `http://pms.ifi.lmu.de/publikationen/PMS-FB/PMS-FB-2002-2_short.pdf`.

16 François Bry and Sebastian Schaffert. Towards a Declarative Query and Transformation Language for XML and Semistructured Data: Simulation Unification. Research Report PMS-FB-2002-2, Institute for Informatics, Ludwig-Maximilian University of Munich, 2002. Long version of [15], `http:/pms.ifi.lmu.de/publikationen/PMS-FB/PMS-FB-2002-2.pdf`.

17 François Bry and Sebastian Schaffert. An Entailment Relation for Reasoning on the Web. In *Proceedings of Rules and Rule Markup Languages for the Semantic Web (RuleML)*. Springer, 2003. `http://pms.ifi.lmu.de/publikationen/PMS-FB/PMS-FB-2003-5.pdf`.

18 François Bry, Sebastian Schaffert, and Andreas Schröder. A contribution to the Semantics of Xcerpt, a Web Query and Transformation Language. In *Applications of Declarative Programming and Knowledge Management – Proceedings of 15th International Conference on Applications of Declarative Programming and Knowledge Management and 18th Workshop on Logic Programming (INAP/WLP)*. `http:/pms.ifi.lmu.de/publikationen/PMS-FB/PMS-FB-2004-5.pdf`.

19 Peter Buneman, Mary Fernandez, and Dan Suciu. UnQL: A Query Language and Algebra for Semistructured Data Based on Structural Recursion. *VLDB Journal*, 9(1):76–110, 2000.

20 Donald D. Chamberlin and Raymond F. Boyce. SEQUEL: A Structured English Query Language. In *Proceedings of the SIGMOD Workshop (SIGMOD)*, volume 1. ACM.

21 James Clark and Steve DeRose. *XML Path Language (XPath) Version 1.0*. W3C Recommendation, November 1999. `http://www.w3.org/TR/xpath/`.

22 Alin Deutsch, Mary Fernandez, Daniela Florescu, Alon Levy, and Dan Suciu. *XML-QL: A Query Language for XML*. W§C Submission, 1998.

23 François Fages. Associative-commutative Unification. In R. E. Shostak, editor, *Proceedings of the Seventh International Conference on Automated Deduction (Napa, CA)*, volume 170, pages 194–208, Berlin, 1984. Springer-Verlag.

24 Olaf Hartig, Christian Bizer, and Johann-Christoph Freytag. Executing SPARQL Queries over the Web of Linked Data. In *Proc. 8th International Semantic Web Conference (ESWC20099)*, Washington DC, USA, 2009.

25 Tom Heath and Christian Bizer. *Linked Data: Evolving the Web into a Global Data Space. Synthesis Lectures on the Semantic Web: Theory and Technology, 1:1*. Morgan & Claypool, 1st edition, 2011.

26 Monika R. Henzinger, Thomas A. Henzinger, and Peter W. Kopke. Computing Simulations on Finite and Infinite Graphs. Technical report, Computer Science Department, Cornell University, July 1996.

27 Robin Milner. An Algebraic Definition of Simulation between Program. Technical Report CS-205, Computer Science Department, Stanford University, 1971. Stanford Aritifical Intelligence Project, Memo AIM-142.

28 Dan Olteanu, Holger Meuss, Tim Furche, and François Bry. Xpath: Looking forward. In *XML-Based Data Management and Multimedia Engineering – Proceedings of the EDBT EDBT 2002 Workshops XMLDM, MDDE, and YRWS, Revised Papers*, volume 2490 of *Lecture Notes in Computer Science*, pages 109–127. Springer, 2002. `http://pms.ifi.lmu.de/publikationen/PMS-FB/PMS-FB-2002-4.pdf`.

29 Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999. Previous number = SIDL-WP-1999-0120.

30 Sebastian Schaffert. *Xcerpt: A Rule-Based Query and Transformation Language for the Web.* Doctoral dissertation, Institute for Informatics, Ludwig-Maximilian University of Munich, October 2004. `http://pms.ifi.lmu.de/publikationen/dissertationen/PMS-DISS-2004-1/Schaffert_Sebastian.pdf`.

31 Sebastian Schaffert and François Bry. Querying the Web Reconsidered: A Practical Introduction to Xcerpt. In *Proceedings of Extreme Markup Languages 2004*, 2004. `http:/pms.ifi.lmu.de/publikationen/PMS-FB/PMS-FB-2004-7.pdf`.

32 Sebastian Schaffert, Thomas Kurz, Dietmar Glachs, Christoph Bauer, Fabian Dorschel, and Manuel Fernandez. The linked media framework: Integrating and interlinking enterprise media content and data. In *Proceedings of I-Semantics 2012*, 2012.

33 Pavel Shvaiko and Jérôme Euzenat. Ontology matching: state of the art and future challenges. In *IEEE Transactions on knowledge and data engineering*, 2012.

34 Gert Smolka. Feature Constraint Logics for Unification Grammars. *Journal of Logic Programming*, 12:51–87, 1992.

35 W3 Consortium. *OWL Web Ontology Language*, February 2004. W3C Recommendation, `http://www.w3.org/TR/owl-ref/`.

36 W3 Consortium. *RDF Vocabulary Description Language 1.0: RDF Schema*, February 2004. W3C Recommendation, `http://www.w3.org/TR/rdf-schema/`.

37 W3 Consortium. *Resource Description Framework*, February 2004. W3C Recommendation, `http://www.w3.org/TR/rdf-primer/`.

38 W3 Consortium. *SPARQL 1.1 Federation Extensions (W3C Working Draft)*, June 2010. `http://www.w3.org/TR/sparql11-federated-query/`.

39 W3 Consortium. *SPARQL 1.1 Query (W3C Working Draft)*, May 2011. `http://www.w3.org/TR/sparql11-query/`.

Modeling Machine Learning and Data Mining Problems with FO(·)*

Hendrik Blockeel, Bart Bogaerts, Maurice Bruynooghe, Broes De Cat, Stef De Pooter, Marc Denecker, Anthony Labarre, Jan Ramon,[1] and Sicco Verwer[2]

1 **Department of Computer Science, KU Leuven**
`firstname.secondname@cs.kuleuven.be`

2 **Radboud Universiteit Nijmegen, Institute for Computing and Information Sciences**
`siccoverwer@gmail.com`

Abstract

This paper reports on the use of the FO(·) language and the IDP framework for modeling and solving some machine learning and data mining tasks. The core component of a model in the IDP framework is an FO(·) theory consisting of formulas in first order logic and definitions; the latter are basically logic programs where clause bodies can have arbitrary first order formulas. Hence, it is a small step for a well-versed computer scientist to start modeling. We describe some models resulting from the collaboration between IDP experts and domain experts solving machine learning and data mining tasks. A first task is in the domain of stemmatology, a domain of philology concerned with the relationship between surviving variant versions of text. A second task is about a somewhat similar problem within biology where phylogenetic trees are used to represent the evolution of species. A third and final task is about learning a minimal automaton consistent with a given set of strings. For each task, we introduce the problem, present the IDP code and report on some experiments.

1998 ACM Subject Classification D.1.6 [Logic Programming], F.4.1 [Mathematical Logic]: Computational logic, I.2.4 [Knowledge Representation Formalisms and Methods]

Keywords and phrases Knowledge representation and reasoning, declarative modeling, logic programming, knowledge base systems, FO(·), IDP framework, stemmatology, phylogenetic tree, deterministic finite state automaton.

Digital Object Identifier 10.4230/LIPIcs.ICLP.2012.14

1 Introduction

Researchers in machine learning and data mining are often confronted with problems for which no standard algorithms are applicable. Here we explore a few of these problems. They can be abstracted as graph problems and are NP-complete. This means that algorithms inherently involve search and that heuristics are needed to guide the search towards solutions. Doing this in a procedural language is complex and cumbersome; this is the kind of application for which high level modeling languages can be very useful. Under such a paradigm, a model specifies the format of the data, the function to be optimized and a set of constraints to be satisfied. The model together with a given problem instance is handed over to a solver which

* This work was supported by BOF project GOA/08/008 and by FWO Vlaanderen.

© Hendrik Blockeel, Bart Bogaerts, Maurice Bruynooghe, Broes De Cat, Stef De Pooter, Marc Denecker, Anthony Labarre, Jan Ramon, and Sicco Verwer;
licensed under Creative Commons License ND

Technical Communications of the 28th International Conference on Logic Programming (ICLP'12).
Editors: A. Dovier and V. Santos Costa; pp. 14–25
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

produces a solution. Several modeling languages exist in the field of Constraint Programming; the Zinc language [16] is a good example. The Answer Set Programming (ASP) paradigm can also be considered a modeling language; many solvers exist, examples are the systems described in [15, 9, 21]. Another such modeling language is FO$(\cdot)$ [6]. Problem solving with such modeling languages makes use of powerful solvers that perform propagation to squeeze the search space each time a choice is made. They relieve the programmer from encoding such propagation in procedural code. By default, these solvers use heuristics which are not problem-specific, but even so they often outperform procedural solutions.

This paper explores the use of the FO$(\cdot)$ language and its incarnation in the IDP framework for solving some machine learning and data mining problems. The core component of a model in the IDP framework is an FO$(\cdot)$ theory consisting of formulas in first order logic and definitions; the latter are basically logic programming clauses with arbitrary first order formulas in the body. The necessary background on FO$(\cdot)$ is given in Section 2.

Section 3 solves a task in the domain of a stemmatology, a part of philology that studies the relationship between surviving variant versions of a text. Section 4 discusses a problem about phylogenetic trees as used in biology. Whereas the first two tasks are new, in Section 5, it is investigated how well FO$(\cdot)$ performs on a standard machine learning task, namely the learning of a minimal deterministic finite state automaton (DFA) that is consistent with a given set of accepted and rejected strings.

In all of these problems, model expansion [17] —expanding a partially given structure into a complete structure that is a model of a theory— is the core computational task. Sometimes, a model that is minimal according to some criterion is required.

2 FO$(\cdot)$ and the IDP framework

2.1 FO$(\cdot)$

The term FO$(\cdot)$ is used to denote a family of extensions of first order logic (FO). In this text, the focus lies on FO$(\cdot)^{\texttt{IDP}}$, the instances supported by the IDP framework. FO$(\cdot)^{\texttt{IDP}}$ extends FO with (among others) *types*, *arithmetic*, *aggregates*, *partial functions* and *inductive definitions*. This section recalls the aspects of FO$(\cdot)$ that are necessary for a good understanding of the rest of the paper; more information can be found in [23] and [3].

A specification in FO$(\cdot)^{\texttt{IDP}}$ consists of a number of logical components, namely *vocabularies, structures, terms*, and *theories*. A vocabulary declares the symbols to be used (contrary to Prolog, the first character of a symbol has no bearing on its kind); a structure is a database with input knowledge; a term declared as a separate component represents a value to be optimized; a theory consists of FO formulas and inductive definitions. An *inductive definition* is a set of *rules* of the form $\forall \bar{x} : P(\bar{x}) \leftarrow \varphi(\bar{x})$. where φ is an FO$(\cdot)^{\texttt{IDP}}$ formula. As argued in [6], the intended meaning of all common forms of definitions is captured by the well-founded semantics [22] which extends the least model semantics of Prolog's definite clauses to rule sets with negation. An FO$(\cdot)^{\texttt{IDP}}$ *formula* differs from FO formulas in several ways. Firstly, FO$(\cdot)^{\texttt{IDP}}$ is a many-sorted logic: every variable has an associated *type* and every type an associated domain. Moreover, it is order-sorted: types can be subtypes of others. Secondly, besides the standard terms of FO, FO$(\cdot)^{\texttt{IDP}}$ also has aggregate terms: functions over a set of domain elements and associated numeric values which map to the sum, product, cardinality, maximum or minimum value of the set.

We write $\mathcal{M} \models T$ to denote that structure $\mathcal{M}$ satisfies theory T. With $x^{\mathcal{M}}$, we denote the interpretation of x under $\mathcal{M}$, where x can be a formula or a term.

2.2 The IDP framework

The IDP framework [5] combines a declarative specification, in $\mathrm{FO}(\cdot)^{\mathtt{IDP}}$, with imperative manipulation of the specification via the Lua [13] scripting language. Such an interaction makes it a *Knowledge Base System* (KBS), as it allows one to reuse the same declarative knowledge for a range of inference tasks such as *model expansion*, *optimization*, *verification*, *symmetry breaking*, *grounding*, etc. For an in-depth treatment of the framework and the supported inferences, we refer to [3].

In this paper, we focus on the inference tasks *model expansion* and *model minimization*. The task of model expansion is, given a vocabulary V, a theory T over V and a partial structure S over V (at least interpreting all types), to find an interpretation $\mathcal{M}$ that satisfies T and expands S, i.e., $\mathcal{M}$ is a model of the theory and the input structure S is a subset of the model. Such a task is represented as $\langle V, T, S \rangle$.

The task of model minimization, represented as $\langle V, T, S, t \rangle$ with V, T and S as above and t a term, is to find a model $\mathcal{M}$ of T that expands S such that for all other models $\mathcal{M}$' expanding S, $t^{\mathcal{M}} \leq t^{\mathcal{M}'}$.

The IDP framework allows users to specify $\mathrm{FO}(\cdot)^{\mathtt{IDP}}$ problem descriptions. Such a problem description consists of logical and procedural components. The basic overall structure of the various logical components is as in the following schema.

```
vocabulary V    { ... }      theory T: V     { ... }
term t: V       { ... }      structure S: V { ... }
```

The first component defines a vocabulary V. The other components define respectively a theory T, a term t and a structure S. They all refer to the vocabulary V for the symbols they use. In general, several vocabularies can be defined, eventually, one vocabulary extending another.

We use IDP syntax in the examples throughout the paper. Each IDP operator has an associated logical operator, the main (non-obvious) operators being: `&`($\wedge$), `|`($\vee$), $\sim$($\neg$), `!`($\forall$), `?`($\exists$), `<=>`($\equiv$), $\sim$=($\neq$).

The procedural component consists of procedures, coded in Lua, that provide the interface between the user and the logical components. Examples will be shown in the next sections.

3 Stemmatology

3.1 The task

The Oxford English Dictionary defines stemmatics, or stemmatology, as "the branch of study concerned with analyzing the relationship of surviving variant versions of a text to each other, especially so as to reconstruct a lost original." A stemma is a kind of "family tree" of a set of manuscripts that indicates which manuscripts have been copied from which other manuscripts, and which manuscript is the original source. It may include both extant (currently existing and available) and non-extant ("lost") manuscripts. The stemma is not necessarily a tree: sometimes a manuscript has been copied partially from one manuscript, and partially from another, in which case the manuscript has multiple parents. Hence, a stemma is in general a connected directed acyclic graph with a single root [1]; we use the term CRDAG (connected rooted DAG) for it.

While constructing a stemma has some similarities with constructing a phylogenetic tree in biology, the algorithms of that domain do not fit the stemmatological context well and specific algorithms are developed [2].

The problem studied here assumes that a CRDAG representing a stemma is given, as well as feature data about (some of) the manuscripts. More specifically, for each location where variation is observed in the manuscripts, the data includes a feature that indicates which variant a particular manuscript has. Note that, in practice, it is highly unlikely that exactly the same variant originated multiple times independently; when a variant occurs in multiple manuscripts, it is reasonable to assume there was one ancestor, common to all of these, where the variant occurred for the first time (the "source" of the variant)[1]. Therefore, we say that the feature is consistent with the stemma if it is possible to indicate for each variant a single manuscript that may have been the origin of that variant. Since for some manuscripts the value of the feature is not known, checking consistency boils down to assigning a variant to each node in the CRDAG in such a way that, for each variant, the nodes having that variant form a CRDAG themselves. Using colors to denote the value of a variant, this property is captured by the following definition.

▶ **Definition 1** (Color-connected)**.** Two nodes x and y in a colored CRDAG are *color-connected* if a node z exists (z can be one of x and y) such that there is a directed path from z to x, and one from z to y, and all nodes on these paths (including z, x, y) have the same color.

Given a partially colored CRDAG, the *color-connected problem* is to complete the coloring such that every pair of nodes of the same color is color-connected.

3.2 An IDP solution

A pair of researchers in stemmatology attempted to develop a search free algorithm. They wrote 370 lines of perl and used a graph library in the background. While it worked for their benchmarks, they were worried about the completeness of their approach. After abstracting the problem as the color-connected problem, we proved that it was NP-complete (hence requires search) and constructed a solvable example for which their algorithm claimed no solution exists. We also worked on an IDP solution. After several iterations, we arrived at the following simple solution which turned out to be faster than the (incomplete) procedural algorithm on the benchmark set. It is shown in Listing 1.

The vocabulary part introduces two types (`manuscript` and `color`), two functions and one predicate. The function `colorOf` maps a manuscripts to its color and the function `sourceOf` maps a color to the manuscript that is the source of the feature. The predicate `copiedBy` is used to represent the CRDAG of the stemma in the input structure.

The theory part compactly represents the color-connectedness property by a single constraint: when the source of the color of a manuscript (`x`) is not equal to the manuscript itself then there must exist a manuscript (`y`) with the same color that has been copied by `x`.

The Lua code of the procedure `process` (omitted, 60 lines) processes the stemma data and builds the input structure for `copiedBy`. It then iterates over the features, partially builds the structure for the function `colorOf` and calls the procedure `check`, passing all structures in the variable `feature`. The latter procedure calls the model expansion and returns the result to `process` which reports them to the user.

Our largest benchmark so far is the Heinrichi data set [18]. This stemma about old Finnish texts includes 48 manuscripts, 51 `copiedBy` tuples and information about 1042 features. Processing all features takes 12 seconds with the IDP system while it took 25 seconds with the original procedural code. Our solution is integrated in the toolset of [20].

---

[1] For some features, e.g., the spelling of a particular word, this does not hold.

Listing 1 Description of the connected-coloring problem using IDP.

```
vocabulary V {
  type manuscript
  type color
  copiedBy(manuscript,manuscript)
  colorOf(manuscript): color
  sourceOf(color): manuscript
}
theory T : V {
  ! x : x ~= sourceOf(colorOf(x))
     => ? y : copiedBy(y,x) & colorOf(x) = colorOf(y).
}
procedure check(feature) {
  return sat(T,feature) // checks existence of a model
}
procedure process(stemmafilename,samplefilename) {
read the stemma data and build a structure for copiedBy
for each feature {
   read the given colors and build a partial structure for colorOf
   call check(feature)
   report the results }
}
```

4 Minimum common supergraphs of partially labelled trees

Phylogenetic trees, extensively surveyed by [7], are the traditional tool for representing the evolution of a given set of species. However, there exist situations in which a tree representation is inadequate. One reason is the presence of evolutionary events that cannot be displayed by a tree: genes may be duplicated, transferred or lost, and recombination events (i.e., the breaking of a DNA strand followed by its reinsertion into a different DNA molecule) as well as hybridisation events (i.e., the combination of genetic material from several species) are known to occur. A second reason is that even when evolution is indeed tree-like, there are cases in which a relatively large number of tree topologies might be "equally good" according to some chosen criteria, and not enough information is available to discriminate between those trees. One solution that has been proposed to address the latter issue is the use of *consensus trees*, where the idea is to find a tree that represents a compromise between the given topologies; another approach, on which we focus here, consists in building a network that is compatible with all topologies of interest. A somewhat loose description of the variant we are interested in, which will be stated in a more formal way below, is to find the smallest graph that contains a given set of evolutionary trees. For more information about those *phylogenetic networks*, see the recent book by [12] and the online, up-to-date annotated bibliography maintained by [8].

4.1 The problem

The studied problem is about the evolution of a fixed set of m given species. The input is a set of phylogenetic trees, each tree showing a plausible relationship between the species. All trees have n ($> m$) nodes, m of them are labeled with the name of the species (typically, in the leaves, but also internal nodes can be labeled). Given $n - m$ extra names, the labeling of each tree can be extended into a full labeling. The completely labeled trees then induce pairs

of labels, whose union yields a graph over the set of n names. The task is to find a network with a minimum number of edges. Here, we formulate the problem as a slightly more general graph problem where we do not fix the size of the initial labeling.

▶ **Definition 2** (Common supergraph of partially labeled n-graphs)**.** Given is a set S of n names and a set of graphs $\{G_1, G_2, \ldots, G_t\}$ where each graph $G_i = (V, E_i, \mathcal{L}_i)$ has n vertices and is partially labeled with an injective function $\mathcal{L}_i : V \to S$. A graph (S, ES) is a *common supergraph* of $\{G_1, G_2, \ldots, G_t\}$ if there exists, for each i, a bijection $\mathcal{L}'_i : V \to S$ that extends $\mathcal{L}_i$ and such that, for each edge $\{v, w\}$ of E_i: $\{\mathcal{L}'_i(v), \mathcal{L}'_i(w)\} \in ES$.

A *minimum* common supergraph (S, ES) is a common supergraph such that $|ES'| \geq |ES|$ for all common supergraphs (S, ES').

Note that every labeling function $\mathcal{L}'_i$ induces an injection $E_i \to ES$, hence the name common supergraph. Figure 1 shows two partially labeled 7-graphs, along with two of their common supergraphs. G_1 is a minimum common supergraph since T_1 and T_2 are not isomorphic and G_1 has only one more edge than each of T_1 and T_2. G_2 is not a minimum common supergraph since it has more edges than G_1.

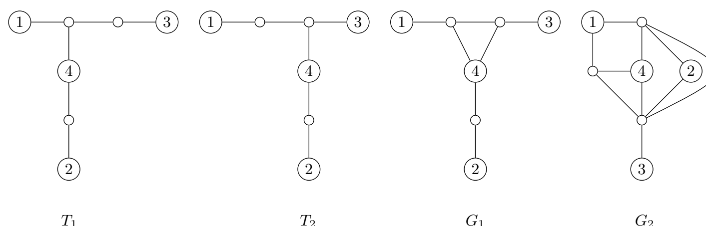


Figure 1 Two 7-graphs T_1 and T_2, a minimum common supergraph G_1, and a common supergraph G_2 that is not minimum.

Now, we can consider the following decision problem: Given a set of partially labeled n-graphs, can the labelings be completed such that the n-graphs have a common supergraph with at most k edges? It is proven in [14] that this problem is NP-hard, even if the n-graphs are trees with all leaves labeled.

4.2 The IDP solution

Listing 2 shows a simple model inspired by [14]. The labeling is declared as a function from nodes to the names (it is partly specified in the input structure). The only constraint of the theory forces the function to be bijective. The common supergraph over the names induced by the labeling is given by the arc atoms. As the minimization is on the number of such atoms, some care is required. Either one should make arc a symmetric relation or one should pay attention to the direction, e.g., by ensuring $x < y$ in arc(x,y) (every type is ordered in FO$(\cdot)^{\texttt{IDP}}$ and provided of a < predicate). The latter is done here as the former gives a somewhat larger grounding.

A feature of the shown solution is that the terms label(t,x) and label(t,y) each have two occurrences in the rules defining arc. The current grounder associates a distinct variable with each occurrence. One can avoid this by replacing the head of the definition by arc(lx,ly) and by adding lx=label(t,x) and ly=label(t,y) to the body. This has a dramatic effect on the size of the grounding and on the solving time; e.g., the grounding is reduced from 620798 to 6024 lines and the solving time from 144s to 8 s on a problem with 5 trees of 8 vertices (4 leaves).

Listing 2 Modelling CS-PLT in the IDP format.

```
vocabulary CsPltVoc {
  type tree
  type vertex
  type name // Isomorphic to vertex
  edge(tree,node,node)  // trees, given in input structure
  arc(name,name) // The induced network
  label(tree,node): name // the labeling,
                    //partially given in the input structure
}
theory CsPltTheory: CsPltVoc {
  { // induced network
    arc(label(t,x),label(t,y)) <- edge(t,x,y) &
                                  label(t,x) < label(t,y).
    arc(label(t,x),label(t,y)) <- edge(t,y,x) &
                                  label(t,x) < label(t,y).
  }
  ! t c : ?1 n : label(t,n) = c. // label function is bijective
}
term SizeOfSupergraph: CsPltVoc { #{ x y : arc(x,y) } }
procedure main() {
  print(minimize(CsPltTheory,CsPltStructure,SizeOfSupergraph)[1])
}
```

The solving time is exponential in the number of nodes and the program becomes impractical on real-world problems, even if the best solution found so far is returned when some time budget is exceeded. However, the versatility of the IDP system allowed us to experiment with various strategies for greedily searching an approximate solution. This led to the following quite natural solution that performed very well, with respect to both running time and quality of the solution.

1. Find a minimum common supergraph (MCS) for every pair of trees.
2. Pick the smallest MCS (say G) and remove the two trees that are the input for G.
3. Find an MCS between G and every remaining tree.
4. Replace G by an MCS with minimum size, remove the tree that is the input for this MCS and go back to step 3 if any tree remains.

Steps 1 and 3 of this simple procedure are performed by IDP using a model very similar to that of Listing 2 (see [14] for the actual model). This greedy approach works very well. Indeed, for large instances and a fixed time budget, the exact method runs out of time and returns a suboptimal solution while the greedy method completes and returns a solution that, although suboptimal, is typically much smaller.

5 Learning deterministic finite state automata

A third task is about learning a *deterministic finite state automaton* (DFA). The goal is to find a (non-unique) smallest DFA that is consistent with a given set of positive and negative examples. It is one of the best studied problems in grammatical inference [4], has many application areas, and is known to be NP-complete [10]. Recently [11] won the 2010 Stamina DFA learning competition [19] by reducing the DFA learning problem to a SAT problem and running an off-the-shelf SAT solver. Here we explore whether an $\mathrm{FO}(\cdot)^{\mathtt{IDP}}$ formalization can compete with this competition winner.

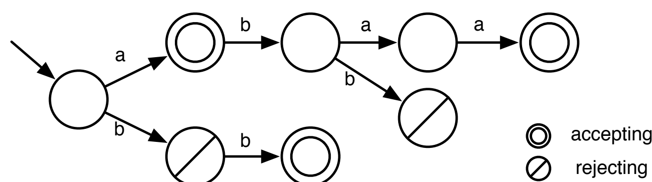


Figure 2 An augmented prefix tree acceptor (APTA) for $S = (S^+ = \{a, abaa, bb\}, S^- = \{abb, b\})$. The start state is the root of the APTA.

5.1 The problem

A *deterministic finite state automaton* (DFA) is a directed graph consisting of a set of *states* Q (nodes) and labeled *transitions* T (directed edges). The root is the start state and any state can be an *accepting state*. In each state, there is exactly one transition for each symbol. A DFA can be used to *generate* or *accept* sequences of symbols (strings) using a process called *DFA computation*. When accepting strings, the symbols of the input string determine a path through the graph. When the final state is an accepting state, the string is accepted, otherwise it is rejected.

Given a pair of finite sets of positive example strings S^+ and negative example strings S^-, (the *input sample*), the goal of *DFA identification* (or *learning*) is to find a (non-unique) *smallest* DFA A that is *consistent* with $S = \{S^+, S^-\}$, i.e., every string in S^+ is accepted, and every string in S^- is rejected by A. Typically, the size of a DFA is measured by $|Q|$, the number of states it contains.

5.2 The solution

Most DFA learning algorithms use a form of state-merging. First, a a tree-shaped automaton called the *augmented prefix tree acceptor* (APTA), is constructed. As can be seen in Figure 2, the APTA accepts the positive examples and rejects the negative ones. State-merging merges states under the constraint that the automaton remains deterministic (at most one transition/label in each state) and that accepting and rejecting states cannot be merged.

States of the final automaton are thus equivalence classes of states of the APTA. Calling the states of the final automaton colors, the problem becomes that of finding a coloring of the states of the APTA that is consistent with the input sample. This is also the approach taken by [11]; they formulate constraints expressing which pairs of states are incompatible, and abstract the problem as a graph, with as states the states of the APTA and as links the incompatible pairs. The problem is now a conventional graph coloring problem and they use a clever SAT encoding to solve it. Here we construct a direct model in $\mathrm{FO}(\cdot)^{\mathtt{IDP}}$. But before doing so, we have to consider one more aspect. For really large problems, the SAT formulation was too big (hundreds of colors, resulting in over 100.000.000 clauses) [11]. To get around such problems, they used a greedy heuristic procedural method to identify a clique of pairwise incompatible states in the APTA. For states in such a clique, the colors can be fixed in advance. The effect is to break some symmetries and to reduce the size of the problem. We assume here that the states of the clique are already colored in the input structure.

The $\mathrm{FO}(\cdot)^{\mathtt{IDP}}$ DFA learning theory is depicted in Listing 3. The types `state`, `label`, the function `trans`, and the predicates `acc` and `rej` describe the given input samples (and hence the APTA). Note that `trans` is partial as the input samples do not define all transitions.

Listing 3 Modelling DFA in the IDP format.

```
vocabulary dfaVoc {
  type state // states used in APTA
  type label // symbols triggering transitions
  type color // available states for resulting automaton
  partial trans(state,label): state // transitions defining APTA
  acc(state) // accepting states of APTA
  rej(state) // rejecting states of APTA
  colorOf(state): color // fixed in input for colors in clique
  // the resulting automaton:
  partial colorTrans(color,label): color // transitions
  accColor(color) // accepting states
}
theory dfaTheory : dfaVoc {
  ! x : acc(x) => accColor(colorOf(x)).
  ! x : rej(x) => ~accColor(colorOf(x)).
  // trans induces colorTrans:
  ! x l z : trans(x,l)=z => colorTrans(colorOf(x),l)=colorOf(z).
}
term nbColorsUsed: dfaVoc { #{ x : (? y : ColorOf(y) = x ) } }
procedure main() {
  stdoptions.symmetry = 1 //detect and break symmetries
  print(minimize(dfaTheory, simple, nbColorsUsed)[1])
}
```

The states of the resulting automaton are elements of the type `color`. Its transitions are described by the function `colorTrans`. This function is also declared as a partial function. To construct a complete DFA from the result, `colorTrans` has to be made total by mapping the missing transitions to a hidden "sink" state. The function `colorOf` maps the states of the APTA on the states (colors) of the final automaton. Finally, the predicate `accColor` describes the accepting states of the resulting automaton.

The theory expresses two constraints on `accColor`: accepting states of the APTA must and rejecting states cannot be mapped to an accepting state of the final automaton. The third constraint states that each transition on the APTA induces a transition between colors. The term nbColorsUsed counts the number of states (colors) of the resulting automaton and is used for minimization. Instead of minimizing the number of states, one could as well minimize other properties such as the number of transitions, depth of the model, the size of loops, etc. They are also easy to formalize in $\mathrm{FO}(\cdot)^{\mathtt{IDP}}$. This makes the resulting DFA learning tool very suitable for application in different problem domains such as software engineering or bioinformatics where other optimization criteria are preferred.

In order to test the performance of the IDP translation, we ran it on the benchmark set of [11]. We compare IDP with two versions of the encoding in [11]: an unoptimized plain encoding (but with the symmetry breaking clique), and an optimized version (with extra symmetry breaking, unit literal propagation, but without redundant clauses). The experiment is not on the minimization problem but on the problem of constructing a DFA with a fixed set of states.

IDP, with the symmetry breaking option on, is significantly faster than the plain SAT encoding (not for the easy problems where the IDP time is dominated by the approximately one second grounding time, a time not needed when the problem is directly encoded in SAT). For example the maximum runtime of an instance in IDP is approximately 1400 seconds while one instance takes over 70000 seconds to solve in the plain encoding. The IDP

translation is however outperformed by the optimized version of the direct SAT translation. In the optimized encoding, the longest recorded runtime is slightly above 100 seconds. In [11] an even better time is obtained by including extra redundant clauses. It is an interesting question whether the performance gap can be closed by adding redundant constraints or by parameter tuning of the SAT solver.

6 Conclusion

We have described three NP-hard problems together with their solution with FO(·) and the IDP framework. The first problem is in the domain of stemmatology. We developed an IDP solution that outperformed the dedicated procedural code of a researcher in the field. We proved the problem is NP-complete and constructed problem instances on which the original code errs. The resulting program is a useful tool for the researchers and is integrated in [20]. In a trivial extension we made the `colorOf` function partial; then only those manuscripts are colored as necessary for making the coloring consistent. This gives useful insight to the philologist. Another planned variation does not enforce a unique source for each color, but minimizes the number of sources. This can provide additional insight when the data are in disagreement with the hypothesized stemma.

The second problem addressed the construction of a minimal common supergraph out of a given set of phylogenetic trees. The use of $\mathrm{FO}(\cdot)^{\mathtt{IDP}}$ allowed the authors of [14] to quickly explore various approaches and to arrive at an approximate method that gives good results.

These two applications illustrate the versatility of FO(·) for solving a new problem. The third application compares an FO(·) formalization with a state of the art solution for the NP-complete problem of learning a DFA. While we observe a performance gap with a highly tuned competition winner, our solution performs better than the initial encoding of [11]. On the other hand, the FO(·) formalization took much less effort to develop and offers a lot more flexibility, e.g., to change the optimization criterion. The application is also a good benchmark for further improving the IDP system.

We hope these applications inspire others to try out the IDP framework. It is a small step for computer scientists knowledgeable about logic and Prolog. While our solutions look deceivingly simple, a word of caution is in place. A first solution is hardly ever the best solution; be convinced that it can be done simpler. Simpler not only means a more concise and elegant model but also, almost always, a better performance. Try to break up complex constraints in simpler ones, requiring less variables.

A common beginners misconception we observed, is to use one function (or relation) for information partially given in the input structure and to use another function that extends the partial function into a total one while that same function can serve by declaring it total (the default for functions) and stating that the input structure is partial.

We also observed a very useful programming pattern. In each of our applications, some equivalence class over some given elements is to be constructed. Representing this relationships as a function from the elements to the set of equivalence classes is an excellent choice (the function `colorOf` in stemmatology and in DFA learning, the function `label` in the phylogenetic trees).

Acknowledgements Caroline Macé and Tara Andrews brought some of the authors in touch with stemmatology and Tara explained them the working of the procedural code.

References

1 T. Andrews and C. Macé. Beyond the tree of texts: Graph methods for stemmatic analysis. *In preparation*, 2012.

2 P. Baret, C. Macé, P. Robinson, C. Peersman, R. Mazza, J. Noret, E. Wattel, Van Mulken M., Robinson P., A. Lantin, P. Canettieri, V. Loreto, H. Windram, M. Spencer, C. Howe, M. Albu, and A. Dress. Testing methods on an artificially created textual tradition. In *The evolution of texts: Confronting stemmatological and genetical methods*, pages 255–283. Istituti editoriali e poligrafici internazionali, Pisa, 2006.

3 Bart Bogaerts, Broes De Cat, Stef De Pooter, and Marc Denecker. The IDP framework reference manual. http://dtai.cs.kuleuven.be/krr/software/idp3/documentation.

4 Colin de la Higuera. A bibliographical study of grammatical inference. *Pattern Recognition*, 38(9):1332–1348, 2005.

5 Stef De Pooter, Johan Wittocx, and Marc Denecker. A prototype of a knowledge-based programming environment. In *International Conference on Applications of Declarative Programming and Knowledge Management*, 2011.

6 Marc Denecker and Eugenia Ternovska. A logic of nonmonotone inductive definitions. *ACM Transactions on Computational Logic (TOCL)*, 9(2):Article 14, 2008.

7 Joseph Felsenstein. *Inferring Phylogenies.* Sinauer Associates, Sunderland, MA, 2004.

8 Philippe Gambette. Who is who in phylogenetic networks: Articles, authors and programs. Published electronically at http://www.atgc-montpellier.fr/phylnet, 2010.

9 Martin Gebser, Benjamin Kaufmann, André Neumann, and Torsten Schaub. *clasp*: A conflict-driven answer set solver. In Chitta Baral, Gerhard Brewka, and John S. Schlipf, editors, *LPNMR*, volume 4483 of *LNCS*, pages 260–265. Springer, 2007.

10 E. Mark Gold. Complexity of automaton identification from given data. *Information and Control*, 37(3):302–320, 1978.

11 Marijn Heule and Sicco Verwer. Exact DFA identification using SAT solvers. In *Grammatical Inference: Theoretical Results and Applications, ICGI 2010*, pages 66–79, 2010.

12 Daniel H. Huson, Regula Rupp, and Celine Scornavacca. *Phylogenetic Networks: Concepts, Algorithms and Applications.* Cambridge University Press, November 2010.

13 Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. Lua – an extensible extension language. *Software: Practice and Experience*, 26(6):635–652, 1996.

14 Anthony Labarre and Sicco Verwer. Merging partially labelled trees: hardness and an efficient practical solution. *In preparation*, 2012.

15 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*, 7:499–562, 2002.

16 Kim Marriott, Nicholas Nethercote, Reza Rafeh, Peter J. Stuckey, Maria Garcia de la Banda, and Mark Wallace. The design of the Zinc modelling language. *Constraints*, 13(3):229–267, 2008.

17 David G. Mitchell and Eugenia Ternovska. A framework for representing and solving NP search problems. In Manuela M. Veloso and Subbarao Kambhampati, editors, *AAAI*, pages 430–435. AAAI Press / The MIT Press, 2005.

18 T. Roos and T. Heikkilä. Evaluating methods for computer-assisted stemmatology using artificial benchmark data sets. *Literary and Linguistic Computing*, 24(4):417–433, 2009.

19 The StaMinA competition, Learning regular languages with large alphabets. http://stamina.chefbe.net/, 2010.

20 Stemmaweb, a collection of tools for analysis of collated texts. http://byzantini.st/stemmaweb/, 2012.

21 Tommi Syrjänen and Ilkka Niemelä. The smodels system. In Thomas Eiter, Wolfgang Faber, and Mirosław Truszczyński, editors, *LPNMR*, volume 2173 of *LNCS*, pages 434–438. Springer, 2001.

22 Allen Van Gelder, Kenneth A. Ross, and John S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, 1991.

23 Johan Wittocx, Maarten Mariën, and Marc Denecker. The IDP system: a model expansion system for an extension of classical logic. In Marc Denecker, editor, *LaSh*, pages 153–165, 2008.

Answering Why and How questions with respect to a frame-based knowledge base: a preliminary report

Chitta Baral, Nguyen Ha Vo, and Shanshan Liang

School of Computing, Informatics, and Decision Systems Engineering
Arizona State University, Tempe, Arizona, USA
`chitta@asu.edu, nguyen.h.vo@asu.edu, shanshan.liang@asu.edu`

Abstract

Being able to answer questions with respect to a given text is the cornerstone of language understanding and at the primary school level students are taught how to answer various kinds of questions including why and how questions. In the building of automated question answering systems the focus so far has been more on factoid questions and comparatively little attention has been devoted to answering why and how questions. In this paper we explore answering why and how questions with respect to a frame-based knowledge base and give algorithms and ASP (answer set programming) implementation to answer two classes of questions in the Biology domain. They are of the form: "How are X and Y related in the process Z?" and "Why is X important to Y?"

1998 ACM Subject Classification D.1.6 Logic Programming, H.3.4 Question-answering (fact retrieval) systems, I.2.4 Frames and scripts

Keywords and phrases answer set programming, frame based knowledge representation, question answering.

Digital Object Identifier 10.4230/LIPIcs.ICLP.2012.26

1 Introduction

In recent years question answering (QA) has become more prominent via efforts such as the Google Knowledge Graph [11] and systems such as Watson [7]. However, most question answering efforts remain focused on factoid questions; a notable exception being navigational "How" questions answered by Siri.

"How" and "Why" questions are important types of questions that are introduced to students at primary school level in their reading and comprehension classes. At the school level answering why questions involves finding the reason or cause of a thing that happened and answering how questions involves finding the way something is done. Answering such questions become more elaborate in Biology where some researchers suggest [15] three kinds of answers to "Why" questions: teleological answer about effects, proximate answers about immediate causes and evolutionary answers based on natural selection; while others [16] propose an even more elaborate categorization of questions and answers such as: How is X used (asked for the biological role/function), How does X work (asked for physiological explanation), and Why does X has a certain item/behavior (asked for the functional significance of certain biological roles). In the literature [1] "How" questions have been referred to as procedural questions.

At present automatic answering of "Why" and "How" questions with respect to large text corpuses [12] are based on factoid extraction where answers are located by looking for

© Chitta Baral, Nguyen HaVo, and S.Liang;
licensed under Creative Commons License ND
Technical Communications of the 28th International Conference on Logic Programming (ICLP'12).
Editors: A. Dovier and V. Santos Costa; pp. 26–36

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

associate words and phrases such as "because of" and "causes". In this paper we take a different approach. Instead of answering "Why" and "How" questions with respect to natural language text [3] , we explore answering them with respect to a frame based knowledge base. Our motivation behind that is to first formalize the notion of answers to such questions; I.e., define what are answers to "Why" and "How" questions with respect to a knowledge base.

We use the frame based biology knowledge base AURA [5] and while identifying several question forms we focus on two specific question forms as a start: "How are X and Y related in the process Z?" and "Why is X important to Y?" Looking at examples in the frame based knowledge representation in AURA we define the notion of an *event description graph* and formalize the answers to our two question types with respect to such graphs. We then give an answer set programming formalization of the reasoning process to find the answers (and thus give an implementation) and conclude with future research directions. Our answer set programming formalization builds up on our earlier work [4] to reason with frame based knowledge using answer set programming.

2 Background

2.1 Frame-Based Knowledge Base

The basic aspects of a frame-based knowledge base (KB) is to represent classes and objects (instances). For classes, the most important information is the class hierarchy. For example[1], the highest class in the AURA[2] [5] hierarchy is "Thing", with two children classes "Entity" and "Event". "Entity" can have descendent classes such as "Cell", "Sunlight", "Sugar" that are biological entities, while "Event" can have descendent classes such as "Photosynthesis", "Mitosis" that are biological processes. We also need to represent objects that may belong to the same classes (share the same basic features), but have their own specific properties. To represent the shared features amongst objects (in order to prevent repetitive encoding of the same set of knowledge entries), "prototypes" of classes are encoded and during reasoning they are cloned by all the objects from that class. The KB normally supports the encoding of multiple inheritance, meaning that a class need to inherit from all of its ancestor classes in the class hierarchy. In this case, in order to obtain the full information for an object, the object needs to clone from all the prototypes of the class it belongs to, as well as the prototypes of all its ancestor classes. When merging the information together, the process of "unification" [6] is introduced to make sure that any conflicts are dealt with properly.

In general, although there is a large body of knowledge bases that use the frame based approach [8], there hasn't been much research on how to use the knowledge encoded in frames declaratively, especially in the realm of question answering applications. In our earlier work [4] we investigated how to utilize the KB for answering "what" questions in a declarative way, as opposed to the procedural approach adopted by the original AURA system. There we give an abstract definition of a KB, and a declarative implementation of "clone and unify". From here on, whenever we refer to an object we use the complete information for that object (after the cloning and unification process), which is obtained by the declarative implementation mentioned earlier.

[1] Note that the various examples mentioned in this paper are from the AURA knowledge base, some with slight modifications.

[2] The AURA knowledge base is a frame-based KB developed manually by knowledge experts. AURA contains large amount of frames describing biology concepts and biology processes, and has been used to answer a wide variety of "what" questions [5].

To the best of our knowledge, there has been little research on answering "How" and "Why" questions with respect to frame-based knowledge bases. The main goal of this paper is to provide insight on how frame based KB can be used to answer some "Why" and "How" questions. To do that we use an "abstract view" of the KB that allows a better illustration of the semantics behind the KB and how they can be used for QA purposes.

2.2 Answer Set Programming

We use Answer Set Programming (ASP) [10] as our knowledge representation language for its strong theoretical foundation [2], expressiveness, the availability of various solvers [9, 14, 13] and its earlier use in the declarative implementation of "clone and unify".

An ASP program is a collection of rules of the form:

$$a \leftarrow a_1, \ldots, a_m, \textit{not } b_1, \ldots, \textit{not } b_n$$

where a, $a_1, ..., a_m$ and $b_1, ..., b_n$ are atoms. The rule reads as "a is true if $a_1 ... a_m$ are all known to be true and $b_1 ... b_n$ can be assumed to be false". The semantics of answer set programs are defined using answer sets (earlier called stable models).

3 Answering two Why/How Questions

As mentioned earlier, in this paper we consider two particular types of Why and How questions: "How are X and Y related in process Z?" and "Why is X important to Y?".

Let us illustrate them with respect to a knowledge base about the process of photosynthesis. The following component of an event description graph (to be formally defined later) expresses the knowledge about photosynthesis.

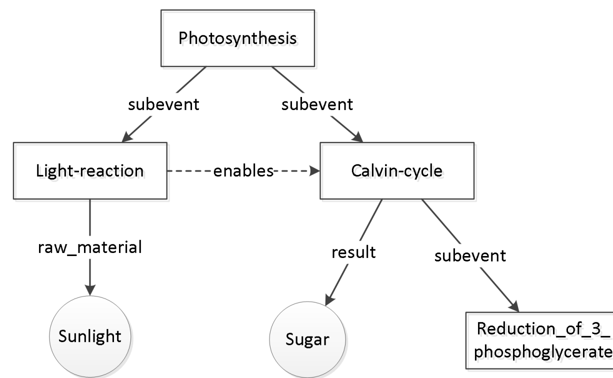


Figure 1 The event description graph of photosynthesis. Events and processes are depicted by rectangles and circles respectively. Compositional edges are represented by solid lines and behavioral edges by dashed lines.

Now consider the "How" question: How are sunlight and sugar related in photosynthesis?

An intuitive answer to this question is: *Photosynthesis has two subevents: light reaction and calvin cycle. The light reaction needs sunlight as its raw material, and later enables the calvin cycle which produces sugar as the result.* This answer can be obtained from the graph in Fig. 1 constructed from the frame based knowledge base AURA by using the information that "raw material", "enables", and "result" are the key slots used by AURA.

Now let us consider the "Why" question: Why is sunlight important to photosynthesis?

An intuitive answer to this question is: *Sunlight is the raw material of light reaction thus sunlight is important to light reaction; light reaction is an important sub-event of photosynthesis; therefore sunlight is also important for photosynthesis.* This answer can be obtained from the graph in Fig. 1 when augmented with information about "importance". Following is such an augmented graph.

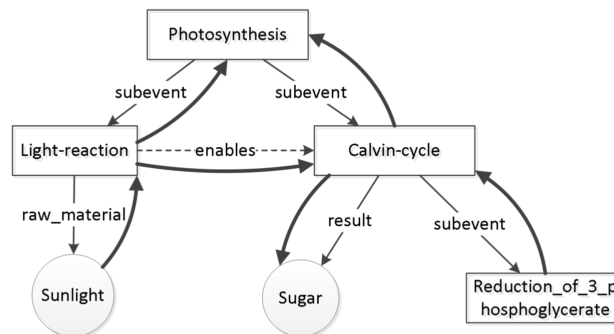


Figure 2 The event description graph of photosynthesis with the "important edges" marked by bold arrow.

Using the augmented graph we need to follow the "important" edges that link "sunlight" to photosynthesis.

The above examples suggest close relationships between the answers of why and how questions and the graph representation of processes. In the following we give a formal representation of processes as graphs, define some generic operations on the graphs and use them in formulating answers to our two kinds of why and how questions.

3.1 Knowledge Bases of Biological Processes

In the frame representation that we use in [4] the Knowledge Base has the generic encoding format: $has(X, S, V)$, where X can be either a class or an object, S refers to a "slot", which describes the property of X, and V is the value for that slot. While the KB may contain a large amount of information, we do not need all of that for our specific types of question answering. Thus we consider and define a simplified view of the KB through the notion of Event Description Graphs.

There are two important aspects of a Knowledge Base of Biological Processes: Events and Entities. Each biological process is a event, which can often be broken down to several sub-events (and sub-events of sub-events). Entities can be involved in the processes as raw materials, results, bases, objects, etc.[3] Using that we now define Event Description Graphs.

▶ **Definition 1.** An Event Description Graph is a directed graph with two types of nodes: event nodes and entity nodes; two types of directed edges: compositional edges and behavioral edges; and a special node referred as main event node or root node which has no incoming edge. An Event Description Graph satisfies the following conditions:

1. All other nodes beside the root are reachable from the root via compositional edges.
2. There are no directed cycle of only compositional edges.

[3] For a complete list such relations (slot names), please refer to the Slot Dictionary in the Component Library (http://www.cs.utexas.edu/ mfkb/RKF/tree/).

3. There are no directed cycle of only behavioral edges.
4. There are no outgoing edges from the entity nodes.

We use $EDG(Z)$ to denote the Event Description Graph with root Z.

"Event nodes" and "entity nodes" represent biological entities and biological processes, respectively. The "compositional edges" and "behavioral edges" are categorized based on specific event-event and event-entity relations. Table 1 shows some example relations that can be viewed as compositional and behavioral edges. For event-to-event relation, only the "sub-event" relation is viewed as a compositional edge, while others are viewed as behavioral edges. All the event-to-entity relations are considered compositional edges.

Each Event Description Graph describes its root event which is a biological process defined in the KB. As all the sub-events are also biological processes, the subgraph with a sub-event as root and that contains all the accessible nodes/edges from that sub-event is considered the Event Description Graph for that sub-event.

Table 1 The slot names indicating "compositional"/"behavioral" edges.

Category	Type	Slot names
Event-to-Event	compositional	sub-event
Event-to-Event	behavioral	next_event, enables, causes, prevents...
Event-to-Entity	compositional	raw_material, result, site, location, base, agent...
Event-to-Entity	behavioral	(null)

A *cpath* from a node X to a node Y in $EDG(Z)$, denoted as $cpath(X, Y)$, is a path consisting of only compositional edges. Similarly, a $bpath(X, Y)$ is a path consisting of only behavioral edges, and an $ipath(X, Y)$, is a path consisting of only "important edges". While $cpath(X, Y)$ and $bpath(X, Y)$ reflect how X and Y are connected compositionally or behaviorally, sometimes we need to add richer semantic information such as an edge being important which is then used to define $ipath(X, Y)$. Intuitively we say that there is an "important edge" from X to Y iff Y can not function properly without X. The following Table 2 shows several functionally important relations.

Table 2 The slot names indicating "functional importance".

Category	Slot names
Entity-to-Event	raw_material, site, base
Event-to-Event(explicit)	enables, causes, regulates, prevents, subevent
Event-to-Event (implicit)	(sample rule) the result of E1 is the raw_material of E2
Event-to-Entity	result

3.2 Answers to two types of Why/How Questions

In this subsection we will formally define the answers to two types of Why and How questions. We will illustrate the definitions and algorithms using the following event description graph.

Given the event description graph of process 1 in Figure 3 consider answering the question "How are process 8 and entity 10 related in process 1?". Intuitively, it seems the answer should only contain important information to understand the relation between 8 and 10 such as: compositional path from process 3 to process 8 through process 5 and compositional path from 3 to 10 through 7 to explain compositional relations between 8 and 10; behavioral

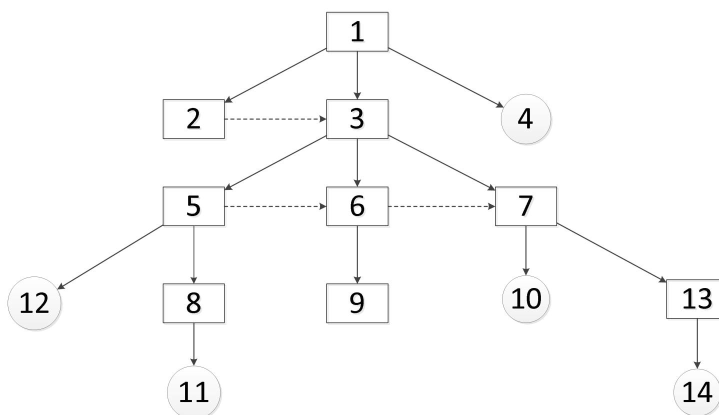


Figure 3 Event Description Graph of process 1. Events and processes are depicted by rectangles and circles respectively. Compositional edges are represented by solid lines and behavioral edges by dashed lines.

path from 5 to 7 through 6 to explain behavioral relations; and compositional edge from 1 to 3 and then to 6 to clarify process 6. Information about process/entity 2, 4, 9, 11, 12, 13 and 14 can be omitted since they are not important for the connection between 8 and 10. Following the above intuition, we formally define the answer for question "How are X and Y related in process Z" as the graph denoted by $MIN_EDG^{Z}_{X,Y}$ defined as follows. First, $LCA(X, Y)\, in\, EDG(Z)$, denotes the lowest common ancestor of X and Y in $EDG(Z)$.

▶ **Definition 2.** Given an event description graph $EDG(Z)$ and two nodes X and Y in that graph, $MIN_EDG^{Z}_{X,Y}$ is the subgraph of $EDG(Z)$ consisting of the following:

- The set of nodes $V_X \cup V_Y \cup V_{behavioral} \cup V_{ZT}$, where $T = LCA(X, Y)$ in $EDG(Z)$; V_x and V_y are the set of nodes in $cpath(T, X)$ and $cpath(T, Y)$ respectively; $V_{behavioral}$ is the set of nodes on any $bpath(X', Y')$, where $X' \in V_x$, and $Y' \in V_y$; and V_{ZT} is the set of nodes in $cpath(Z, T)$.
- The set of edges consisting of the union of the edges obtained from $EDG(T)$ by removing all edges that connect to the nodes in $EDG(T) \setminus V$ and the edges in $cpath(Z, T)$ from Z to T in $EDG(Z)$.

The path $cpath(Z, T)$ from Z to T helps clarify what T is with respect to the process Z. With respect to relating 8 and 10 in Figure 3, this $cpath(Z, T)$ is the path from process 1 to process 3.

▶ **Example 3.** Let us consider the photosynthesis example. $LCA(sunlight, sugar)$ in $EDG(photosynthesis)$ is photosynthesis itself. Using definition 2 we have: V_x = *{photosynthesis, light_reaction, sunlight}*, $V_y = \{photosynthesis, calvin_cycle, sugar\}$, and $V_{behavioral} = \{\}$.

$MIN_EDG(photosynthesis)^{photosynthesis}_{sunlight,sugar}$ thus has nodes V = *{photosynthesis, light_reaction, calvin_cycle, sunlight, sugar}* and edges: E = *{(photosynthesis, light_reaction), (photosynthesis, calvin_cycle), (light_reaction, sunlight), (calvin_cycle, sugar), (light_reaction, calvin_cycle)}*.

This subgraph expresses the answer to the question "How are sunlight and sugar related in photosynthesis?". We can also answer the question "How are sunlight and sugar related?", by finding the *MIN_EDG* for sunlight and sugar in the entire KB, rather than in the Event Description Graph of photosynthesis.

Now let us consider the question: "Why is X important to Y?" In order to answer it we need both $MIN_EDG(event)^{T}_{X,Y}$ and the notion of path, where *event* is the ancestor of all events in the KB. Using them we have the following definition.

▶ **Definition 4.** The answer for "Why is X important to Y?" is the combination of: (i) $MIN_EDG(event)^{T}_{X,Y}$ where $T = LCA(X,Y)\, in\, EDG(event)$ and (ii) $ipath(X,Y)$.

4 ASP Encodings for General Reasoning Rules

In this section we discuss the encoding for all the defined components discussed in the previous section.

4.1 Encoding the Semantics of Slots

We encode the slot names that indicates a compositional/behavioral edges (Table 1) as follows:

```
cedge(subevent; raw_material; result; site; location; base; agent).
bedge(next_event; enables; causes; prevents).
iedge(raw_material; site; base; subevent).
iedge(enables; causes; regulates; supports; prevents; result).
```

4.2 Compositional-Connected, Behavioral-Connected, and Importantly-Connected

The following rules define "directly-compositionally-connected" (dcconnects), "directly-behaviorally-connected" (dbconnects) and "directly-importantly-connected" (diconnects).

```
dcconnects(X, Y) :-  has(X, S, Y), event(X), cedge(S).
dbconnects(X, Y) :-  has(X, S, Y), event(X), event(Y), bedge(S).
diconnects(X, Y) :-  has(X, S, Y), iedge(S).
```

The predicates *cconnect*, *bconnect* and *iconnect* denoting "compositionally-connected", "behaviorally-connected" and "importantly-connected" are transitive closures of "dcconnects", "dbconnects", and "dcconnects" respectively and are defined in the standard way.

4.3 Cpath, Bpath, and Ipath

We can utilize the above defined relations to enumerate all the nodes on the compositional/behavioral path from a node to another. We define $cpath(A, Z, I, C)$ which means C is the Ith node in the path from A to Z.

```
cpath(A, Z, 0, A)   :- cconnects(A, Z).
cpath(A, Z, T+1, C) :- cpath(A, Z, T, B), dcconnects(B,C), step(T),
                       cconnects(C, Z).
```

We similarly define *bpath* and *ipath* and use them.

4.4 Finding Common Ancestor

Now we encode the rules for finding common ancestor for X and Y. The first two rules encode the special cases where either X is the ancestor of Y or Y is the ancestor of X. The 3rd rule means that Z is a common ancestor of X and Y if Z cconnects to both X and Y.

```
common_ancestor(X,X,Y) :- cconnects(X, Y), X != Y.
common_ancestor(Y,X,Y) :- cconnects(Y, X), X != Y.
common_ancestor(Z,X,Y) :- cconnects(Z, X), cconnects(Z, Y), X != Y.
```

Following the algorithm, the next step is to find the lowest-common-ancestor. We say that $Z1$ is not a lowest common ancestor if there exist another common ancestor $Z2$ which is a descendant of $Z1$ ($Z1$ cconnects to $Z2$). And then we can define the lowest-common-ancestor(lcs) using default negation.

```
not_lcs(Z1, X, Y) :- common_ancestor(Z1,X,Y), common_ancestor(Z2,X,Y),
                      Z1 != Z2, cconnects(Z1, Z2).
lcs(Z, X, Y)      :- common_ancestor(Z, X, Y), not not_lcs(Z, X, Y).
```

4.5 Correctness of the General Reasoning Rules

Proposition 1. Z is the lowest common ancestor of X and Y w.r.t. the KB of process P iff: $lcs(Z, X, Y)$ is entailed by the program described above.

5 ASP Encoding of How/Why Question and Answering

In this section we present the encoding for the general reasoning rules used in answering the how and why questions. We provide the template for encoding both the questions and the answers in a generic and easy-to-expand fashion. Our encoding is sufficient for a large list of questions. However, there are questions that themselves encompass a complicated semantic meaning, which needs additional representations that are beyond the scope of this work.

5.1 Question Encoding

To encode the semantics in the questions properly, we use the following template. Each question has a *QID*, *Type*, *Category*, two *Parameters*, and optionally the *Scope*.

```
question(QID).                  has(QID, type, Type).
has(QID, category, Category).  has(QID, param1, XClass).
has(QID, param2, YClass).      has(QID, scope, ScopeClass).
```

In the following we illustrate the encodings for the questions "How are sunlight and sugar related in photosynthesis?" and "Why is sunlight important to photosynthesis?", respectively.

```
question(q1).                         question(q2).
has(q1, type, how).                   has(q2, type, why).
has(q1, category, relation).          has(q2, category, important_to).
has(q1, param1, sunlight).            has(q2, param1, sunlight).
has(q1, param2, sugar).               has(q2, param2, photosynthesis).
has(q1, scope, photosynthesis).
```

5.2 Answer Graph

According to the definitions of the answers for how and why questions, we use $_answer_graph(Q, Z, X, Y)$ to denote the answer $MIN_EDG(Scope)^{Z}_{X,Y}$ of question Q. The rule head $_answer_graph(Q, Z, X, Y)$ denotes the answer as a graph with root Z, and two descendant X and Y, in which X, Y and Z are instances of XClass, YClass, and ScopeClass. Z is the lowest common ancestor for X and Y.

```
_answer_graph(Q, Z, X, Y) :-
question(Q),
has(Q, type, how),
has(Q, category, relation),
has(Q, param1, XClass),
has(Q, param2, YClass),
has(Q, scope, ScopeClass),
has(X, instance_of, XClass),
has(Y, instance_of, YClass),
has(Z, instance_of, ScopeClass),
lcs(Z, X, Y).
```

```
_answer_graph(Q, Z, X, Y) :-
question(Q),
has(Q, type, why),
has(Q, category, important_to),
has(Q, param1, XClass),
has(Q, param2, YClass),
XClass != YClass,
has(X, instance_of, XClass),
has(Y, instance_of, YClass),
iconnects(X, Y),
lcs(Z, X, Y).
```

Similar to the “How” question, for the “Why” question we also find the lowest common ancestor (without the scope information) Z of X and Y to form the answer graph, while enforcing that there must exist an ipath from X to Y.

5.3 Obtaining Complete Answer: Output All Nodes/Edges in the Answer Graph and Answer Path

We use *_answer_node(Q, AnswerGraph, node, E)* to denote all the nodes E in the *AnswerGraph*. The first two rules encode that if the question has a answer graph (Q, Z, X, Y), then all the nodes E on the compositional path from both Z to X and Z to Y will be answer nodes. The 3rd rule encodes that all the nodes on the behavioral paths linking every pair of nodes on the compositional paths are also answer nodes. The last rule encodes that all the nodes on compositional paths from the scope of the question to Z (to clarify the role of Z with respect to the given scope) are also answer nodes. Note that Scope, a prototype of ScopeClass, is the instance of the ScopeClass class. In our DB, prototype is always defined for each class.

```
_answer_node(Q, _answer_graph(Q, Z, X, Y), node, E) :-
    _answer_graph(Q, Z, X, Y), cpath(Z, X, T, E), step(T).

_answer_node(Q, _answer_graph(Q, Z, X, Y), node, E) :-
    _answer_graph(Q, Z, X, Y), cpath(Z, Y, T, E), step(T).

_answer_node(Q, AnswerGraph, node, E) :-
    _answer_node(Q, AnswerGraph, node, X),
    _answer_node(Q, AnswerGraph, node, Y),
    bpath(X, Y, T, E), step(T).

_answer_node(Q, _answer_graph(Q, Z, X, Y), node, E) :-
    _answer_graph(Q, Z, X, Y), has(Q, scope, ScopeClass),
    has(Scope, prototype_of, ScopeClass), cpath(Scope, Z, T, E), step(T).
```

Next the final answer is the collection of nodes and edges in the answer graph and appropriate rules are written for that. For lack of space we skip the propositions that relate the earlier definition of an answer with the answer obtained using the ASP rules.

6 Conclusion, Discussion and Future work

With good progress in information retrieval, natural language processing, speech recognition and associated fields, question answering systems are becoming a reality. However, most question answering systems are about factoid questions. But various applications, such as building intelligent tutoring systems need more general form of question answering, especially involving why and how questions. To develop systems that can answer why and how questions with respect to text, we first need to be clear about correct answers to why and how questions in a more formal setting. In other words, we need to develop a formal theory of answers to why and how questions. Towards that end, we made a start in this paper with focus on why and how question answering with respect to a structured knowledge base. We developed an abstract notion of an event description graph and used that to formalize answers with respect to two kinds of why and how questions. We then gave an ASP implementation of our formalization. The motivation behind using ASP is that as a prerequisite to answering questions with respect to a frame based knowledge base we need to implement issues such as inheritance and cloning in making inferences about facts of the form $has(X, S, Y)$. In an earlier paper we showed how ASP can be used to implement inheritance and cloning. Hence our use of ASP in this paper. Moreover we are not aware of any other declarative implementation or formalization of cloning in any other language.

Although, so far in this paper we only considered two kinds of why and how questions, our approach generalizes beyond those two to additional types. Below, we give a couple of examples on that. In the future we will consider additional types of why and how questions.

1. To answer questions of the form, "How does X occur?", we just need to define an answer graph as:

```
_answer_graph(Q, X, First_subevent_of_X, Last_subevent_of_X), ...
```

in which the first and last subevents of X can be easily obtained if all the subevents are properly ordered using the "next_event" relation.

2. Similarly, to answer questions of the form, "How does X produce Y?", the answer graph is defined as:

```
_answer_graph(Q, X, null, Y), ...
```

so that only the cpath from X to Y is in the answer graph, and this chain of reaction is "how X produces Y" if the last event in the chain has Y as result.

3. Similarly, to answer questions of the form, "Why does X have Property Y?", the answer graph is defined as:

```
_answer_graph(Q, X, Y, Subevent_of_X_that_involves_Y), ...
```

where for each subevents of X that involves Y, we generate an answer graph and output "why is Y important for that subevent".

References

1. F. Aouladomar. Towards answering procedural questions. page 21. Proc. of KRAQ'05, an IJCAI05 workshop., 2005.
2. C. Baral. *Knowledge representation, reasoning and declarative problem solving.* Cambridge University Press, 2003.
3. C. Baral, S. Liang, and V. Nguyen. Towards deep reasoning with respect to natural language text in scientific domains. *DeepKR Workshop*, 2011.
4. Chitta Baral and Shanshan Liang. From knowledge represented in frame-based languages to declarative representation and reasoning via asp. *13th International Conference on Principles of Knowledge Representation and Reasoning*, 2012.
5. Vinay K. Chaudhri, Peter E. Clark, Sunil Mishra, John Pacheco, Aaron Spaulding, and Jing Tien. Aura: Capturing knowledge and answering questions on science textbooks. Technical report, SRI International, 2009.
6. P. Clark, B. Porter, and B.P. Works. Km: The knowledge machine 2.0: Users manual, 2004.
7. D. Ferrucci, E. Brown, J. Chu-Carroll, J. Fan, D. Gondek, A.A. Kalyanpur, A. Lally, J.W. Murdock, E. Nyberg, J. Prager, et al. Building watson: An overview of the deepqa project. *AI Magazine*, 31(3):59–79, 2010.
8. R. Fikes and T. Kehler. The role of frame-based representation in reasoning. *Communications of the ACM*, 28(9):904–920, 1985.
9. M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and S. Thiele. A user's guide to gringo, clasp, clingo, and iclingo. *November*, 77:78–80, 2008.
10. M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In R. Kowalski and K. Bowen, editors, *Logic Programming: Proc. of the Fifth Int'l Conf. and Symp.*, pages 1070–1080. MIT Press, 1988.
11. Google Knowledge Graph. `http://www.google.com/insidesearch/features/search/knowledge.html`.
12. R. Higashinaka and H. Isozaki. Corpus-based question answering for why-questions. *Proc. of IJCNLP*, 1:418–425, 2008.
13. 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 (TOCL)*, 7(3):499–562, 2006.
14. I. Niemelä and P. Simons. Smodels—an implementation of the stable model and well-founded semantics for normal logic programs. *Logic Programming and Nonmonotonic Reasoning*, pages 420–429, 1997.
15. Tom Shellberg. Teaching how to answer 'why'questions about biology. *The American Biology Teacher*, 63(1):16–19, 2012/06/17 2001.
16. A. Wouters. The functional perspective of organismal biology. *Current Themes in Theoretical Biology*, pages 33–69, 2005.

Applying Machine Learning Techniques to ASP Solving

Marco Maratea[1], Luca Pulina[2], and Francesco Ricca[3]

1 **DIBRIS, Università degli Studi di Genova**
Viale F.Causa 15, 16145 Genova, Italy
marco@dist.unige.it

2 **POLCOMING, Università degli Studi di Sassari**
Viale Mancini 5, 07100 Sassari, Italy
lpulina@uniss.it

3 **Dipartimento di Matematica, Università della Calabria**
Via P. Bucci, 87030 Rende, Italy
ricca@mat.unical.it

Abstract

Having in mind the task of improving the solving methods for Answer Set Programming (ASP), there are two usual ways to reach this goal: (i) extending state-of-the-art techniques and ASP solvers, or (ii) designing a new ASP solver from scratch. An alternative to these trends is to build on top of state-of-the-art solvers, and to apply machine learning techniques for choosing automatically the "best" available solver on a per-instance basis.

In this paper we pursue this latter direction. We first define a set of cheap-to-compute syntactic features that characterize several aspects of ASP programs. Then, given the features of the instances in a *training* set and the solvers performance on these instances, we apply a classification method to inductively learn algorithm selection strategies to be applied to a *test* set. We report the results of an experiment considering solvers and training and test sets of instances taken from the ones submitted to the "System Track" of the 3rd ASP competition. Our analysis shows that, by applying machine learning techniques to ASP solving, it is possible to obtain very robust performance: our approach can solve a higher number of instances compared with any solver that entered the 3rd ASP competition.

1998 ACM Subject Classification D.1.6 Logic Programming, I.2.4 Knowledge Representation Formalisms and Methods, I.2.6 Learning

Keywords and phrases Answer Set Programming, Automated Algorithm Selection, Multi-Engine solvers

Digital Object Identifier 10.4230/LIPIcs.ICLP.2012.37

1 Introduction

Having in mind the task of improving the robustness, i.e., the ability to perform well across a wide set of problem domains, and the efficiency, i.e., the quality of solving a high number of instances, of solving methods for Answer Set Programming (ASP) [13, 27, 30, 26, 14, 3], it is possible to extend existing state-of-the-art techniques implemented in ASP solvers, or design from scratch a new ASP system with powerful techniques and heuristics. An alternative to these trends is to build on top of state-of-the-art solvers, leveraging on a number of efficient ASP systems, e.g., [36, 22, 24, 10, 28, 21, 36], and applying machine learning techniques for inductively choosing, among a set of available ones, the "best" solver on the basis of the characteristics, called *features*, of the input program. This approach falls

© Marco Maratea, Luca Pulina, and Francesco Ricca;
licensed under Creative Commons License ND

Technical Communications of the 28th International Conference on Logic Programming (ICLP'12).
Editors: A. Dovier and V. Santos Costa; pp. 37–48

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

in the framework of the *algorithm selection problem* [34]. Related approaches, following a per-instance selection, have been exploited for solving propositional satisfiability (SAT), e.g., [40], and Quantified SAT (QSAT), e.g., [32] problems. In ASP, an approach for selecting the "best" CLASP internal configuration is followed in [9], while another approach that imposes learned heuristics ordering to SMODELS is [2].

In this paper we pursue this direction, and design a multi-engine approach to ASP solving. We first define a set of cheap-to-compute syntactic features that describe several characteristics of ASP programs, paying particular attention to ASP peculiarities. We then compute such features for the grounded version of all benchmark submitted to the "System Track" of the 3rd ASP Competition [5] falling in the "*NP*" and "*Beyond NP*" categories of the competition: this track is well suited for our study given that (*i*) contains many ASP instances, (*ii*) the language specification, ASP-Core, is a common ASP fragment such that (*iii*) many ASP systems can deal with it.

Then, starting from the features of the instances in a *training* set, and the solvers performance on these instances, we apply the "Nearest-neighbor" classification method to inductively learn general algorithm selection strategies to be applied to a *test* set. We perform an analyses that consider as test set the instances evaluated to the 3rd ASP competition.

Our experiments show that it is possible to obtain a very robust performance, by solving a higher number of instances than all the solvers that entered the 3rd ASP competition and DLV [22].

The paper is structured as follow. Section 2 contains preliminaries about ASP and classification methods. Section 3 then describes our benchmarks setting, in terms of dataset and solvers employed. Section 4 defines how features and solvers have been selected, and presents the classification methods employed. Section 5 shows the performance analysis, while Section 6 and 7 end the paper with discussion about related work and conclusions, respectively.

2 Preliminaries

In this section we recall some preliminary notions concerning answer set programming and machine learning techniques for algorithm selection.

2.1 Answer Set Programming

Answer Set Programming (ASP) [13, 27, 30, 26, 14, 3] is a declarative programming formalism proposed in the area of non-monotonic reasoning and logic programming. The idea of ASP is to represent a given computational problem by a logic program whose answer sets correspond to solutions, and then use a solver to find those solutions [26].

In the following, we recall both the syntax and semantics of ASP. The presented constructs are included in ASP-Core [5], which is the language specification that was originally introduced in the 3rd ASP Competition [5] as well as the one employed in our experiments (see Section 3). Hereafter, we assume the reader is familiar with logic programming conventions, and refer the reader to [14, 3, 12] for complementary introductory material on ASP, and to [4] for obtaining the full specification of ASP-Core.

2.1.1 Syntax

A variable or a constant is a *term*. An *atom* is $p(t_1, ..., t_n)$, where p is a *predicate* of arity n and $t_1, ..., t_n$ are terms. A *literal* is either a *positive literal* p or a *negative literal* `not` p, where

p is an atom. A *(disjunctive) rule* r is of the form:

$$a_1 \;\vee\; \cdots \;\vee\; a_n \;\text{:-}\; b_1, \cdots, b_k,\; \texttt{not}\;\; b_{k+1}, \cdots,\; \texttt{not}\;\; b_m.$$

where $a_1, \ldots, a_n, b_1, \ldots, b_m$ are atoms. The disjunction $a_1 \vee \ldots \vee a_n$ is the *head* of r, while the conjunction $b_1, \ldots, b_k, \texttt{not}\ b_{k+1}, \ldots, \texttt{not}\ b_m$ is the *body* of r. We denote by $H(r)$ the set of atoms occurring in the head of r, and we denote by $B(r)$ the set of body literals. A rule s.t. $|H(r)| = 1$ (i.e., $n = 1$) is called a *normal rule*; if the body is empty (i.e., $k = m = 0$) it is called a *fact* (and the :- sign is omitted); if $|H(r)| = 0$ (i.e., $n = 0$) is called a *constraint*. A rule r is *safe* if each variable appearing in r appears also in some positive body literal of r.

An *ASP program* $\mathcal{P}$ is a finite set of safe rules. A $\texttt{not}$-free (resp., $\vee$-free) program is called *positive* (resp., *normal*). A term, an atom, a literal, a rule, or a program is *ground* if no variable appears in it.

2.1.2 Semantics

Given a program $\mathcal{P}$, the *Herbrand Universe* $U_{\mathcal{P}}$ is the set of all constants appearing in $\mathcal{P}$, and the *Herbrand Base* $B_{\mathcal{P}}$ is the set of all possible ground atoms which can be constructed from the predicates appearing in $\mathcal{P}$ with the constants of $U_{\mathcal{P}}$. Given a rule r, $Ground(r)$ denotes the set of rules obtained by applying all possible substitutions from the variables in r to elements of $U_{\mathcal{P}}$. Similarly, given a program $\mathcal{P}$, the *ground instantiation* of $\mathcal{P}$ is $Ground(\mathcal{P}) = \bigcup_{r \in \mathcal{P}} Ground(r)$.

An *interpretation* for a program $\mathcal{P}$ is a subset I of $B_{\mathcal{P}}$. A ground positive literal A is true (resp., false) w.r.t. I if $A \in I$ (resp., $A \notin I$). A ground negative literal $\texttt{not}\ A$ is true w.r.t. I if A is false w.r.t. I; otherwise $\texttt{not}\ A$ is false w.r.t. I.

The answer sets of a program $\mathcal{P}$ are defined in two steps using its ground instantiation: First the answer sets of positive disjunctive programs are defined; then the answer sets of general programs are defined by a reduction to positive ones and a stability condition.

Let r be a ground rule, the head of r is true w.r.t. I if $H(r) \cap I \neq \emptyset$. The body of r is true w.r.t. I if all body literals of r are true w.r.t. I, otherwise the body of r is false w.r.t. I. The rule r is *satisfied* (or true) w.r.t. I if its head is true w.r.t. I or its body is false w.r.t. I.

Given a *ground positive* program P_g, an *answer set* for P_g is a subset-minimal interpretation A for P_g such that every rule $r \in P_g$ is true w.r.t. A (i.e., there is no other interpretation $I \subset A$ that satisfies all the rules of P_g).

Given a *ground* program P_g and an interpretation I, the (Gelfond-Lifschitz) *reduct* [14] of P_g w.r.t. I is the positive program P_g^I, obtained from P_g by (i) deleting all rules $r \in P_g$ whose negative body is false w.r.t. I, and (ii) deleting the negative body from the remaining rules of P_g.

An answer set (or stable model) of a general program $\mathcal{P}$ is an interpretation I of $\mathcal{P}$ such that I is an answer set of $Ground(\mathcal{P})^I$.

As an example consider the program $\mathcal{P} = \{\ a \vee b \,\text{:-}\, c.,\ b \,\text{:-}\,\texttt{not}\ a, \texttt{not}\ c.,\ a \vee c \,\text{:-}\,\texttt{not}\ b.,\ k \,\text{:-}\, a.,\ k \,\text{:-}\, b.\ \}$ and $I = \{b, k\}$. The reduct $\mathcal{P}^I$ is $\{a \vee b \,\text{:-}\, c.,\ b.\ k \,\text{:-}\, a.,\ k \,\text{:-}\, b.\}$. I is an answer set of $\mathcal{P}^I$, and for this reason it is also an answer set of $\mathcal{P}$.

2.2 Multinomial classification for Algorithm Selection

With regard to empirically hard problems, there is rarely a best algorithm to solve a given combinatorial problem, while it is often the case that different algorithms perform well on different problem instances. Among the approaches for solving this problem, in this work we rely on a per-instance selection algorithm in which, given a set of *features* –i.e., numeric

Table 1 Problems and instances considered, coming from the *NP* and *Beyond NP* classes of the 3rd ASP competition.

Problem	Class	#Instances
DisjunctiveScheduling	*NP*	10
GraphColouring	*NP*	60
HanoiTower	*NP*	59
KnightTour	*NP*	10
MazeGeneration	*NP*	50
Labyrinth	*NP*	261
MultiContextSystemQuerying	*NP*	73
Numberlink	*NP*	150
PackingProblem	*NP*	50
SokobanDecision	*NP*	50
Solitaire	*NP*	25
WeightAssignmentTree	*NP*	62
MinimalDiagnosis	*Beyond NP*	551
StrategicCompanies	*Beyond NP*	51

values that represent particular characteristics of a given instance–, it is possible to choose the best algorithm among a pool of them –in our case, tools to solve ASP instances. In order to make such a selection in an automatic way, we model the problem using *multinomial classification* algorithms, i.e., machine learning techniques that allow automatic classification of a set of instances, given instance features.

More in detail, in multinomial classification we are given a set of patterns, i.e., input vectors $X = \{\underline{x}_1, \ldots \underline{x}_k\}$ with $\underline{x}_i \in \mathbb{R}^n$, and a corresponding set of labels, i.e., output values $Y \in \{1, \ldots, m\}$, where Y is composed of values representing the m classes of the multinomial classification problem. In our modeling, the m classes are m ASP solvers. We think of the labels as generated by some unknown function $f : \mathbb{R}^n \to \{1, \ldots, m\}$ applied to the patterns, i.e., $f(\underline{x}_i) = y_i$ for $i \in \{1, \ldots, k\}$ and $y_i \in \{1, \ldots, m\}$. Given a set of patterns X and a corresponding set of labels Y, the task of a multinomial classifier c is to extrapolate f given X and Y, i.e., construct c from X and Y so that when we are given some $\underline{x}^\star \in X$ we should ensure that $c(\underline{x}^\star)$ is equals to $f(\underline{x}^\star)$. This task is called *training*, and the pair (X, Y) is called the *training set*.

3 Benchmark data and Settings

In this section we report some information concerning the benchmark settings employed in this work, which is needed for properly introducing the techniques described in the remainder of the paper. In particular, we report some data concerning: benchmark problems, instances and ASP solvers employed, as well as the hardware platform, and the execution settings for reproducibility of experiments.

3.1 Dataset

The benchmarks considered for the experiments belong to the suite of the 3rd ASP Competition [5]. This is a large and heterogeneous suite of hard benchmarks, which was already

employed for evaluating the performance of state-of-the-art ASP solvers, which are encoded in ASP-Core. That suite includes planning domains, temporal and spatial scheduling problems, combinatorial puzzles, graph problems, and a number of application domains i.e., database, information extraction and molecular biology field.[1] More in detail, we have employed the encodings used in the System Track of the competition, and all the problem instances made *available* (in form of facts) from the contributors of the problem submission stage of the competition, which are available from the competition website [4]. Note that this is a superset of the instances actually selected for running (and, thus *evaluated* in) the competition itself. Hereafter, with *instance* we refer to the complete input program (i.e., encoding+facts) to be fed to a solver for each instance of the problem to be solved.

The techniques presented in this paper are conceived for dealing with propositional programs, thus we have grounded all the mentioned instances by using GrinGo (v.3.0.3) [11] to obtain a setup very close to the one of the competition.We considered only computationally-hard benchmarks, corresponding to all problems belonging to the categories *NP* and *Beyond NP* of the competition. The dataset is summarized in Table 1, which also reports the complexity classification and the number of available instances for each problem.

3.2 Executables and Hardware Settings

We have run all the ASP solvers in our experiments that entered the System Track of the 3rd ASP Competition [4] with the addition of DLV [22] (which did not participate in the competition since it is developed by the organizers of the event). In this way we have covered –to the best of our knowledge– all the state-of-the-art solutions fitting the benchmark settings. In detail, we have run: clasp [10], claspD [7], claspfolio [9], idp [39], cmodels [24], sup [25], Smodels [36], and several solvers from both the lp2sat [20] and lp2diff [21] families, namely: lp2gminisat, lp2lminisat, lp2lgminisat, lp2minisat, lp2diffgz3, lp2difflgz3, lp2difflz3, and lp2diffz3. More in detail, clasp is a native ASP solver relying on conflict-driven nogood learning; claspD is an extension of clasp that is able to deal with disjunctive logic programs, while claspfolio exploits machine-learning techniques in order to choose the best-suited execution options of clasp; idp is a finite model generator for extended first-order logic theories, which is based on *MiniSatID* [28]; Smodels is one of the first robust native ASP solvers that have been made available to the community; DLV [22] is one of the first systems able to cope with disjunctive programs; cmodels exploits a SAT solver as a search engine for enumerating models, and also verifying model minimality whenever needed; sup exploits nonclausal constraints, and can be seen as a combination of the computational ideas behind cmodels and Smodels; the lp2sat family employs several variants (indicated by the trailing g, l and lg) of a translation strategy to SAT and resorts on MiniSat [8] for actually computing the answer sets; the lp2diff family translates programs in difference logic over integers [37] and exploit *Z3* [6] as underlying solver (again, g, l and lg indicate different translation strategies). Solvers were run on the same configuration (i.e., parameter settings) as in the competition.

Concerning the hardware employed and the execution settings, all the experiments were carried out on CyberSAR [29], a cluster comprised of 50 Intel Xeon E5420 blades equipped with 64 bit Gnu Scientific Linux 5.5. Unless otherwise specified, the resources granted to the solvers are 600s of CPU time and 2GB of memory. Time measurements were carried out using the `time` command shipped with Gnu Scientific Linux 5.5.

[1] An exhaustive description of the benchmark problems can be found in [4].

4 Designing a Multi-Engine ASP Solver

The design of a multi-engine solver involves several steps: (*i*) design of (syntactic) features that are both significant for classifying the instances and cheap-to-compute (so that the classifier can be fast and accurate); (*ii*) selection of solvers that are representative of the state of the art (to be able to obtain the best possible performance in any considered instance); and (*iii*) selection of the classification algorithm, and fair design of training and test sets, to obtain a robust and unbiased classifier.

In the following we describe the choices we have made for designing ME-ASP, which is our multi-engine solver for ground ASP programs.

4.1 Features

We consider syntactic features that are cheap-to-compute, i.e., computable in linear time in the size of the input, given that in previous work (e.g., [32]) syntactic features have been profitably used for characterizing (inherently) ground instances. The features that we compute for each ground program are divided into four groups: problems size, balance, "proximity to horn" and ASP-based peculiar features. This categorization is borrowed from [31]. The problem size features are: number of rules r, number of atoms a, ratios r/a, $(r/a)^2$, $(r/a)^3$ and ratios reciprocal a/r, $(a/r)^2$ and $(a/r)^3$. The balance features are: fraction of unary, binary and ternary rules. The "proximity to horn" features are: fraction of horn rules and number of occurrences in a horn rule for each atom. We have added a number of ASP peculiar features, namely: number of true and disjunctive facts, fraction of normal rules and constraints c. Also some combinations, e.g., c/r, are considered for a total of 52 features.

We were able to ground with GRINGO 1425 instances out of a total of 1462 in less than 600s.[2] Our system for extracting features from ground programs can then compute all features (in less than 600s) for 1371 programs: to have an idea of its performance, it can compute all features of a ground program of approximately 20MB in about 4s.

4.2 Solvers selection

The target of our selection is to collect a pool of solvers that is representative of the state-of-the-art solver (SOTA), i.e., considering a problem instance, the oracle that always fares the best among available solvers. In order to do that, we ran preliminary experiments, and we report the results (regarding the *NP* class) in Table 2. Looking at the table, first we notice that we do not report results related to both CLASPD and CLASPFOLIO. Concerning the results of CLASPD, we report that –considering the *NP* class– its performance is subsumed by the performance of CLASP. Considering the performance of CLASPFOLIO, we exclude such system from this analysis because we consider it as a yardstick system, i.e., we will compare its performance against the ones related to ME-ASP.

Looking at Table 2, we can see that only 4 solvers out of 16 are able to solve a noticeable amount of instances *uniquely*, namely CLASP, CMODELS, DLV, and IDP. Concerning *Beyond NP* instances, we report that only three solvers are able to cope with such class of problems, name CLASPD, CMODELS, and DLV. Considering that both CMODELS and DLV are involved in the previous selection, the pool of engines used in ME-ASP will be composed of 5 solvers, namely CLASP, CLASPD, CMODELS, DLV, and IDP.

[2] The exceptions are 10 and 27 instances of DisjunctiveScheduling and PackingProblem, respectively.

Table 2 Results of a pool of ASP solvers on the *NP* instances of the 3rd ASP Competition. The table is organized as follows: Column "**Solver**" reports the solver name, column "**Solved**" reports the total amount of instances solved with a time limit of 600 seconds, and, finally, in column "**Unique**" we report the total amount of instances solved uniquely by the corresponding solver.

Solver	Solved	Unique	Solver	Solved	Unique
CLASP	445	26	LP2DIFFZ3	307	–
CMODELS	333	6	LP2SAT2GMINISAT	328	–
DLV	241	37	LP2SAT2LGMINISAT	322	–
IDP	419	15	LP2SAT2LMINISAT	324	–
LP2DIFFGZ3	254	–	LP2SAT2MINISAT	336	–
LP2DIFFLGZ3	242	–	SMODELS	134	–
LP2DIFFLZ3	248	–	SUP	311	1

4.3 Classification algorithms and training

The classification method employed in our analysis is **Nearest-neighbor** (NN), already considered in [32] in QBF solving: it is a classifier yielding the label of the training instance which is closer to the given test instance, whereby closeness is evaluated using some proximity measure, e.g., Euclidean distance; we use the method described in [1] to store the training instances for fast look-up.

As mentioned in Section 2.2, in order to train the classifier, we have to select a pool of instances for training purpose, i.e., the training set. Concerning such selection, our aim is twofold. On the one hand, we want to compose a training set in order to train a robust model.

As result of the considerations above, we design a training set–TS1 in the following– composed of the 320 instances solved uniquely –without taking into account the instances involved in the competition– by the pool of engines selected in Section 4.2. The rational of this choice is to try to "mask" noisy information during model training.

Our next experiment is devoted to training the classifier, and to assessing its accuracy. Referring to the notation introduced in Section 2.2, even assuming that a training set is sufficient to learn f, it is still the case that different sets may yield a different f. The problem is that the resulting trained classifier may underfit the unknown pattern –i.e., its prediction is wrong– or overfit –i.e., be very accurate only when the input pattern is in the training set. Both underfitting and overfitting lead to poor *generalization* performance, i.e., c fails to predict $f(\underline{x}^*)$ when $\underline{x}^* \neq \underline{x}$. However, statistical techniques can provide reasonable estimates of the generalization error. In order to test the generalization performance, we use a technique known as *stratified 10-times 10-fold cross validation* to estimate the generalization in terms of *accuracy*, i.e., the total amount of correct predictions with respect to the total amount of patterns. Given a training set (X, Y), we partition X in subsets X_i with $i \in \{1, \dots 10\}$ such that $X = \bigcup_{i=1}^{10} X_i$ and $X_i \cap X_j = \emptyset$ whenever $i \neq j$; we then train $c_{(i)}$ on the patterns $X_{(i)} = X \setminus X_i$ and corresponding labels $Y_{(i)}$. We repeat the process 10 times, to yield 10 different c and we obtain the global accuracy estimate.

We finally report the accuracy results related to the experiment described above for our classification method: 92.81%.

Table 3 Results of the various solvers on the grounded instances evaluated at the 3rd ASP competition. ME-ASP(NN) has been trained on the TS1 training set.

Solver	*NP*		*Beyond NP*		**Total**	
	#Solved	Time	#Solved	Time	#Solved	Time
CLASP	60	5132.45	–	–	–	–
CLASPD	–	–	13	2344.00	–	–
CMODELS	56	5092.43	9	2079.79	65	7172.22
DLV	37	1682.76	15	1359.71	52	3042.47
IDP	61	5010.79	–	–	–	–
ME-ASP (NN)	66	4854.78	15	3187.31	81	8042.09
CLASPFOLIO	62	4824.06	–	–	–	–
SOTA	71	5403.54	15	1221.01	86	6624.55

5 Performance analysis

In this section we present the results of the analysis we have performed. We consider the training sets TS1 introduced in Section 4, composed of uniquely solved instances, and as test set the successfully grounded instances evaluated at the 3rd ASP Competition (a total of 88 instances): the goal of this analysis is to test the *efficiency* of our approach on all the evaluated instances when the model is trained on the whole space of the uniquely solved instances.

The results are reported in a table structured as follows: the first column reports the name of a solver, the second, third and fourth columns report the results of each solver on *NP*, *Beyond NP* classes, and on both classes, respectively, in terms of the number of solved instances within the time limit and sum of their solving times (a sub-column is devoted to each of these numbers). About the last column, numbers are reported only for ME-ASP and the engines that have been selected on both classes in Section 4.2 (note that CLASPD always performs worse than CLASP on *NP* instances, and CLASPFOLIO can only handle *NP* instances).

We report the results obtained by running: ME-ASP with the NN classification method introduced in Section 4.3, denoted with ME-ASP(NN) the component engines employed by ME-ASP on each class as explained in Section 4.2, CLASPFOLIO and SOTA, which is the ideal multi-engine solver (considering the engines employed).

We remind the reader that, for ME-ASP, the number of instances on which ME-ASP is run is further limited to the ones for which we were able to compute all features, and its timings include both the time spent for extracting the features from the ground instances, and the time spent by the classifier.

Results are shown in Table 3. We can see that, on problems of the *NP* class, ME-ASP(NN) solves the highest number of instances, 5 more than IDP, 6 more than CLASP and 4 more than CLASPFOLIO, that we remind the fastest solver in the *NP* class that entered the System Track of the competition. On the *Beyond NP* problems, instead, ME-ASP(NN) and DLV solve 15 instances (DLV having best mean CPU time), followed by CLASPD and CMODELS, which solve 13 and 9 instances, respectively. It is interesting to report the overall result of CLASPD, i.e., the overall winner of the System Track of the competition on both *NP* and *Beyond NP* classes: it solves a total of 62 instances (i.e., 52 *NP* instances and 13 *Beyond NP* instances), thus a total of 19 instances less than ME-ASP(NN).

Summarizing, ME-ASP(NN) is the solver that solves the highest number of instances in comparison with (i) its engines, (ii) CLASPFOLIO, i.e., the fastest solver in the *NP* class that entered the System Track of the competition, and (iii) CLASPD, i.e., the overall winner of the System Track of the competition. It is further very interesting to note that its performance is very close to the SOTA solver which, we remind, has the ideal performance that we could expect in these instances with these engines.

6 Related Work

Starting from the consideration that, on empirically hard problems, there is rarely a "global" best algorithm, while it is often the case that different algorithms perform well on different problem instances, Rice [34] defined the algorithm selection problem as the problem of finding an effective, or good, or best algorithm, based on an abstract model of the problem at hand. Along this line, several works have been done to tackle combinatorial problems efficiently. [16, 23] described the concept of "algorithm portfolio" as a general method for combining existing algorithms into new ones that are unequivocally preferable to any of the component algorithms. Most related papers to our work are [40, 32] for solving SAT and QSAT problems. Both [40] and [32] rely on a per-instance analysis, like the one we have performed in this paper: in [32], which is the work closest to our, the goal is to design a multi-engine solver, i.e. a tool that can choose among its engines the one which is more likely to yield optimal results. The approach in [40] has also the ability to compute features on-line, e.g., by running a solver for an allotted amount of time and looking "internally" to solver statistics, with the option of changing the solver on-line: this is a per-instance algorithm portfolio approach. The algorithm portfolio approach is employed also in, e.g., [16] on Constraint Satisfaction and MIP, [35] on QSAT and [15] on planning problems. The advantage of the algorithm portfolio over a multi-engine is that it is possible, by combining algorithms, to reach, in each instance, better performance than the best engine, while this is the bound for a multi-engine solver. On the other hand, an algorithm portfolio needs internal changes in the code of the engines, while the multi-engine treats the engines as black-box, thus no internal modification, even minor, is requested, resulting in higher modularity for this approach: when a new engine is added, there is just the need to update the model. It has to be noticed that both [32] and [40] reached very good results, e.g., AQME, the multi-engine solver implementing the approach in [32] had top performance at the 2007 QBF competition.[3] [33] extends [32] by introducing a self-adaptation of the learned selection policies when the approach fails to give a good prediction.

Other approaches work by designing methods for automatically tuning and configuring the solver parameters: this approach is followed in, e.g., [19, 18] for solving SAT and MIP problems, and [38] for planning problems. An overview can be found in [17]. In ASP, the approach implemented in CLASPFOLIO [9] mixes characteristics of the algorithm portfolio approach with others more similar to this second trend: it works by selecting the most promising CLASP internal configuration on the basis of both static and dynamic features of the input program, the latter obtained by running CLASP for a given amount of time. In CLASPFOLIO, features are extracted by means of the CLASPRE tool. Thus, like the algorithms portfolio approaches, it can compute both static and dynamic features, while trying to automatically configure the "best" CLASP configuration on the basis of the computed features. An alternative approach is followed in the DORS framework of [2], where in the off-line

[3] `http:www.qbflib.org/qbfeval`.

learning phase, carried out on representative programs from a given domain, a heuristic ordering is selected to be then used in SMODELS when solving other programs from the same domain. The target of this work seems to be real-world problem domains where instances have similar structures, and heuristic ordering learned in some (possibly small) instances in the domain can help to improve the performance on other (possibly big) instances.

7 Conclusion

In this paper we have applied machine learning techniques to ASP solving with the goal of developing a fast and robust multi-engine ASP solver. To this end, we have: (*i*) specified a number of cheap-to-compute syntactic features that allow for accurate classification of ground ASP programs; (*ii*) applied a multinomial classification method to learning algorithm selection strategies; (*iii*) implemented these techniques in our multi-engine solver ME-ASP, which is available for download at `http://www.mat.unical.it/ricca/me-asp`. The performance of ME-ASP was assessed on an experiment, which was conceived for checking efficiency of our approach, involving training and test sets of instances taken from the ones submitted to the System Track of the 3rd ASP competition. Our analysis shows that, our multi-engine solver ME-ASP is very robust and efficient, and outperforms both its component engines and state of the art solvers.

Acknowledgements The authors would like to thank Marcello Balduccini for usefuls discussion on his solver DORS.

References

1. D.W. Aha, D. Kibler, and M.K. Albert. Instance-based learning algorithms. *Machine learning*, 6(1):37–66, 1991.
2. Marcello Balduccini. Learning and using domain-specific heuristics in ASP solvers. *AI Communications – The European Journal on Artificial Intelligence*, 24(2):147–164, 2011.
3. Chitta Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving.* Cambridge University Press, Tempe, Arizona, 2003.
4. Francesco Calimeri, Giovambattista Ianni, and Francesco Ricca. The third answer set programming system competition, since 2011. `https://www.mat.unical.it/aspcomp2011/`.
5. Francesco Calimeri, Giovambattista Ianni, Francesco Ricca, Mario Alviano, Annamaria Bria, Gelsomina Catalano, Susanna Cozza, Wolfgang Faber, Onofrio Febbraro, Nicola Leone, Marco Manna, Alessandra Martello, Claudio Panetta, Simona Perri, Kristian Reale, Maria Carmela Santoro, Marco Sirianni, Giorgio Terracina, and Pierfrancesco Veltri. The Third Answer Set Programming Competition: Preliminary Report of the System Competition Track. In *Proc. of LPNMR11.*, pages 388–403, Vancouver, Canada, 2011. LNCS Springer.
6. Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *TACAS*, pages 337–340, 2008.
7. Christian Drescher, Martin Gebser, Torsten Grote, Benjamin Kaufmann, Arne König, Max Ostrowski, and Torsten Schaub. Conflict-Driven Disjunctive Answer Set Solving. In Gerhard Brewka and Jérôme Lang, editors, *Proceedings of the Eleventh International Conference on Principles of Knowledge Representation and Reasoning (KR 2008)*, pages 422–432, Sydney, Australia, 2008. AAAI Press.
8. Niklas Eén and Niklas Sörensson. An Extensible SAT-solver. In *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003*, pages 502–518. LNCS Springer, 2003.

9 Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Torsten Schaub, Marius Thomas Schneider, and Stefan Ziller. A portfolio solver for answer set programming: Preliminary report. In James P. Delgrande and Wolfgang Faber, editors, *Proc. of the 11th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, volume 6645 of *LNCS*, pages 352–357, Vancouver, Canada, 2011. Springer.

10 Martin Gebser, Benjamin Kaufmann, André Neumann, and Torsten Schaub. Conflict-driven answer set solving. In *Twentieth International Joint Conference on Artificial Intelligence (IJCAI-07)*, pages 386–392, Hyderabad, India, January 2007. Morgan Kaufmann Publishers.

11 Martin Gebser, Torsten Schaub, and Sven Thiele. GrinGo : A New Grounder for Answer Set Programming. In Chitta Baral, Gerhard Brewka, and John S. Schlipf, editors, *Logic Programming and Nonmonotonic Reasoning, 9th International Conference, LPNMR 2007, Tempe, AZ, USA, May 15-17, 2007, Proceedings*, volume 4483 of *Lecture Notes in Computer Science*, pages 266–271, Tempe, Arizona, 2007. Springer.

12 Michael Gelfond and Nicola Leone. Logic Programming and Knowledge Representation – the A-Prolog perspective . *Artificial Intelligence*, 138(1–2):3–38, 2002.

13 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.

14 Michael Gelfond and Vladimir Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, 9:365–385, 1991.

15 Alfonso Gerevini, Alessandro Saetti, and Mauro Vallati. An automatically configurable portfolio-based planner with macro-actions: PbP. In Alfonso Gerevini, Adele E. Howe, Amedeo Cesta, and Ioannis Refanidis, editors, *Proc. of the 19th International Conference on Automated Planning and Scheduling*, Thessaloniki, Greece, 2009. AAAI.

16 Carla P. Gomes and Bart Selman. Algorithm portfolios. *Artificial Intelligence*, 126(1-2):43–62, 2001.

17 Holger H. Hoos. Programming by optimization. *Communucations of the ACM*, 55(2):70–80, 2012.

18 Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Automated configuration of mixed integer programming solvers. In Andrea Lodi, Michela Milano, and Paolo Toth, editors, *Proc. of the 7th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, volume 6140 of *LNCS*, pages 186–202, Bologna, Italy, 2010. Springer.

19 Frank Hutter, Holger H. Hoos, Kevin Leyton-Brown, and Thomas Stützle. ParamILS: An automatic algorithm configuration framework. *Journal of Artificial Intelligence Research*, 36:267–306, 2009.

20 Tomi Janhunen. Some (in)translatability results for normal logic programs and propositional theories. *Journal of Applied Non-Classical Logics*, 16:35–86, 2006.

21 Tomi Janhunen, Ilkka Niemelä, and Mark Sevalnev. Computing stable models via reductions to difference logic. In *Proceedings of the 10th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, LNCS, pages 142–154, Postdam, Germany, 2009. Springer.

22 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, July 2006.

23 K. Leyton-Brown, E. Nudelman, G. Andrew, J. Mcfadden, and Y. Shoham. A portfolio approach to algorithm selection. In *In IJCAI-03*, 2003.

24 Yuliya Lierler. Disjunctive Answer Set Programming via Satisfiability. In Chitta Baral, Gianluigi Greco, Nicola Leone, and Giorgio Terracina, editors, *Logic Programming and*

Nonmonotonic Reasoning — 8th International Conference, LPNMR'05, Diamante, Italy, September 2005, Proceedings, volume 3662 of *Lecture Notes in Computer Science*, pages 447–451. Springer Verlag, September 2005.

25 Yuliya Lierler. Abstract Answer Set Solvers. In *Logic Programming, 24th International Conference (ICLP 2008)*, volume 5366 of *Lecture Notes in Computer Science*, pages 377–391. Springer, 2008.

26 Vladimir Lifschitz. Answer Set Planning. In Danny De Schreye, editor, *Proceedings of the 16th International Conference on Logic Programming (ICLP'99)*, pages 23–37, Las Cruces, New Mexico, USA, November 1999. The MIT Press.

27 V. Wiktor Marek and Mirosław Truszczyński. Stable models and an alternative logic programming paradigm. *CoRR*, cs.LO/9809032, 1998.

28 Maarten Mariën, Johan Wittocx, Marc Denecker, and Maurice Bruynooghe. SAT(ID): Satisfiability of propositional logic extended with inductive definitions. In *Proc. of the 11th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, LNCS, pages 211–224, Guangzhou, China, 2008. Springer.

29 A. Masoni, M. Carpinelli, G. Fenu, A. Bosin, D. Mura, I. Porceddu, and G. Zanetti. Cybersar: A lambda grid computing infrastructure for advanced applications. In *Nuclear Science Symposium Conference Record (NSS/MIC), 2009 IEEE*, pages 481–483. IEEE, 2009.

30 Ilkka Niemelä. Logic Programs with Stable Model Semantics as a Constraint Programming Paradigm. In Ilkka Niemelä and Torsten Schaub, editors, *Proceedings of the Workshop on Computational Aspects of Nonmonotonic Reasoning*, pages 72–79, Trento, Italy, May/June 1998.

31 Eugene Nudelman, Kevin Leyton-Brown, Holger H. Hoos, Alex Devkar, and Yoav Shoham. Understanding random SAT: Beyond the clauses-to-variables ratio. In Mark Wallace, editor, *Proc. of the 10th International Conference on Principles and Practice of Constraint Programming (CP)*, Lecture Notes in Computer Science, pages 438–452, Toronto, Canada, 2004. Springer.

32 Luca Pulina and Armando Tacchella. A multi-engine solver for quantified boolean formulas. In Christian Bessiere, editor, *Proc. of the 13th International Conference on Principles and Practice of Constraint Programming (CP)*, Lecture Notes in Computer Science, pages 574–589, Providence, Rhode Island, 2007. Springer.

33 Luca Pulina and Armando Tacchella. A self-adaptive multi-engine solver for quantified boolean formulas. *Constraints*, 14(1):80–116, 2009.

34 John R. Rice. The algorithm selection problem. *Advances in Computers*, 15:65–118, 1976.

35 Horst Samulowitz and Roland Memisevic. Learning to solve QBF. In *Proc. of the 22th AAAI Conference on Artificial Intelligence*, pages 255–260, Vancouver, Canada, 2007. AAAI Press.

36 Patrik Simons, Ilkka Niemelä, and Timo Soininen. Extending and Implementing the Stable Model Semantics. *Artificial Intelligence*, 138:181–234, June 2002.

37 smt-lib-web. The Satisfiability Modulo Theories Library, 2011. http://www.smtlib.org/.

38 Mauro Vallati, Chris Fawcett, Alfonso Gerevini, Holger Hoos, and Alessandro Saetti. Generating fast domain-specific planners by automatically configuting a generic parameterised planner. Working notes of 21st International Conference on Automated Planning and Scheduling (ICAPS-11) – Workshop on Planning and Learning, 2011.

39 Johan Wittocx, Maarten Mariën, and Marc Denecker. The IDP system: a model expansion system for an extension of classical logic. In Marc Denecker, editor, *Logic and Search, Computation of Structures from Declarative Descriptions (LaSh 2008)*, pages 153–165, Leuven, Belgium, November 2008.

40 Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. SATzilla: Portfolio-based algorithm selection for SAT. *JAIR*, 32:565–606, 2008.

An Answer Set Solver for non-Herbrand Programs: Progress Report

Marcello Balducchini[1]

1 **Kodak Research Laboratories**
Eastman Kodak Company
Rochester, NY 14650-2102 USA
marcello.balduccini@gmail.com

Abstract

In this paper we propose an extension of Answer Set Programming (ASP) by non-Herbrand functions, i.e. functions over non-Herbrand domains, and describe a solver for the new language. Our approach stems for our interest in practical applications, and from the corresponding need to compute the answer sets of programs with non-Herbrand functions efficiently. Our extension of ASP is such that the semantics of the new language is obtained by a comparatively small change to the ASP semantics from [8]. This makes it possible to modify a state-of-the-art ASP solver in an incremental fashion, and use it for the computation of the answer sets of (a large class of) programs of the new language. The computation is rather efficient, as demonstrated by our experimental evaluation.

1998 ACM Subject Classification I.2.4 Knowledge Representation Formalisms and Methods

Keywords and phrases Answer Set Programming, non-Herbrand Functions, Answer Set Solving, Knowledge Representation and Reasoning

Digital Object Identifier 10.4230/LIPIcs.ICLP.2012.49

1 Introduction

In this paper we describe an extension of Answer Set Programming (ASP) [8, 12, 2] called ASP{f}, and a solver for the new language.

In logic programming, functions are typically interpreted over the Herbrand Universe, with each functional term $f(x)$ mapped to its own canonical syntactical representation. That is, in most logic programming languages, the value of an expression $f(x)$ is $f(x)$ itself, and thus strictly speaking $f(x) = 2$ is false. This type of functions, the corresponding languages and efficient implementation of solvers is the subject of a substantial amount of research (we refer the reader to e.g. [5, 3, 13]).

When representing certain kinds of knowledge, however, it is sometimes convenient to use functions with *non-Herbrand domains* (*non-Herbrand functions* for short), i.e. functions that are interpreted over domains other than the Herbrand Universe. For example, when describing a domain in which people enter and exit a room over time, it may be convenient to represent the number of people in the room at step s by means of a function $occupancy(s)$ and to state the effect of a person entering the room by means of a statement such as

$$occupancy(S + 1) = O + 1 \leftarrow occupancy(S) = O$$

where S is a variable ranging over the possible time steps in the evolution of the domain.

© Marcello Balduccini;
licensed under Creative Commons License ND

Technical Communications of the 28th International Conference on Logic Programming (ICLP'12).
Editors: A. Dovier and V. Santos Costa; pp. 49–60

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Of course, in most logic programming languages, non-Herbrand functions can still be represented, but the corresponding encodings are not as natural and declarative as the one above. For instance, a common approach consists in representing the functions of interest using relations, and then characterizing the functional nature of these relations by writing auxiliary axioms. In ASP, one would encode the above statement by (1) introducing a relation $occupancy'(s,o)$, whose intuitive meaning is that $occupancy'(s,o)$ holds iff the value of $occupancy(s)$ is o; and (2) re-writing the original statement as a rule

$$occupancy'(S+1, O+1) \leftarrow occupancy'(S,O). \tag{1}$$

The characterization of the relation as representing a function would be completed by an axiom such as

$$\neg occupancy'(S,O') \leftarrow occupancy'(S,O),\ O \neq O'. \tag{2}$$

which intuitively states that $occupancy(s)$ has a unique value. The disadvantage of this representation is that the functional nature of $occupancy'(s,o)$ is only stated in (2). When reading (1), one is given no indication that $occupancy'(s,o)$ represents a function – and, before finding statements such as (2), one can make no assumption about the functional nature of the relations in a program when a combination of (proper) relations and non-Herbrand functions are present.

Various extensions of ASP with non-Herbrand functions exist in the literature. In [4], Quantified Equilibrium Logic is extended with support for equality. A subset of the general language, called FLP, is then identified which can be translated into normal logic programs. Such translation makes it possible to compute the answer sets of FLP programs using ASP solvers. [10] proposes instead the use of second-order theories for the definition of the semantics of the language. Again, a transformation is described, which removes non-Herbrand functions and makes it possible to use ASP solvers for the computation of the answer sets of programs in the extended language. In [11, 14] the semantics is based on the notion of reduct as in the original ASP semantics [8]. For the purpose of computing answer sets, a translation is defined, which maps programs of the language from [11, 14] to constraint satisfaction problems, so that CSP solvers can be used for the computation of the answer sets of programs in the extended language. Finally, the language of CLINGCON [7] extends ASP with elements from constraint satisfaction. The CLINGCON solver finds the answer sets of a program by interleaving the computations of an ASP solver and of a CSP solver.

Our investigation stems for our interest in practical applications, and in particular from the need for a knowledge representation language with non-Herbrand functions that can be used for such applications and that allows for an efficient computation of answer sets. From this point of view, the existing approaches have certain limitations.

The transformations to constraint satisfaction problems used in [11, 14] certainly allow for an efficient computation of answer sets using constraint solving techniques, as demonstrated by the experimental results in [14]. On the other hand, the recent successes of CDCL-based solvers (see e.g. [9]) such as CLASP [6] have shown that for certain domains CSP solvers perform poorly compared to CDCL-based solvers. For practical applications it is therefore important to ensure the availability of a CDCL-based solver as well. Furthermore, as observed in [4], the requirement made in [11, 14] that non-Herbrand functions be total yields some counterintuitive results in certain knowledge representation tasks, which, from our point of view, limits the practical applications of the language. This arguments also holds for

CLINGCON. An additional limitation of CLINGCON is the fact that the interleaved computation it performs carries some overhead.

In both [4] (where functions are partial) and [10] (where functions are total) the computation of the answer sets of a program is obtained by translating the program into a normal logic program, and then using state-of-the-art ASP solving techniques and solvers. Unfortunately, in both cases the translation to normal logic programs causes a substantial growth of the size of the translated (ground) program compared to the original (ground) program. Two, similar and often concurrent reasons exist for this growth. First of all, when a non-Herbrand function is removed and replaced by a relation-based representation, axioms that ensure the uniqueness of value of the function have to be introduced. In [4], for example, when a function $f(\cdot)$ is removed, the following constraint is introduced:

$$\leftarrow holds_f(X,V), holds_f(X,W), V \neq W. \tag{3}$$

As usual, before an ASP solver can be used, this constraint must in turn be replaced by its ground instances, obtained by substituting every variable in it by a constant. This process causes the appearance of $|D_f|^2 \cdot |C_f|$ ground instances, where D_f and C_f are respectively the domain and the co-domain of function f. In the presence of functions with a sizable domain and/or co-domain, the number of ground instances of (3) can grow quickly and impact the performance of the solver rather substantially. Secondly, certain syntactic elements of these extended languages, once mapped to normal logic programs, can also yield translations with large ground instances. Taking again [4] as an example (the transformation in [10] appears to follow the same pattern), consider the FLP rule:

$$p(x) \leftarrow f(x) \# g(x). \tag{4}$$

which intuitively says that $p(x)$ must hold if f and g are defined for x and have different values. During the transformation to normal logic programs, this rule is translated into:

$$p(x) \leftarrow Y \neq Z, holds_f(x,Y), holds_g(x,Z).$$

Similarly to the previous case, the number of ground instances of this rule grows proportionally with $|D_f|^2$, and in the presence of non-Herbrand functions with sizable domains, solver performance can be affected quite substantially. Although one might argue that it is possible to modify an ASP solver to guarantee that (3) is enforced without the need to explicitly specify it in the program, such a solution is unlikely to be applicable in the case of an arbitrary rule such as (4).

In response to these issues, in this paper we define an extension of ASP with non-Herbrand functions, called ASP{f}, that is obtained with a comparatively small modification to the semantics from [8]. The nature of this change makes it possible to modify a state-of-the-art ASP solver in an incremental fashion, and to use it directly for the computation of the answer sets of (a large class of) ASP{f} programs. This prevents the phenomenon of the quadratic growth of the ground instance described above and results in a rather efficient computation, as demonstrated later in the paper.

The rest of the paper is organized as follows. The next two sections describe the syntax and the semantics of the proposed language. In the following section we discuss the topic of knowledge representation with non-Herbrand functions. Next, we describe our ASP{f} solver and report experimental results. Finally, we draw conclusions and discuss future work.

2 The Syntax of ASP{f}

In this section we define the syntax of ASP{f}. To keep the presentation simple, in this paper the version of ASP{f} described here does not allow for Herbrand functions, and thus from now on we drop the "non-Herbrand" attribute. (Allowing for Herbrand functions is straightforward.)

The syntax of ASP{f} is based on a signature $\Sigma = \langle \mathcal{C}, \mathcal{F}, \mathcal{R} \rangle$ whose elements are, respectively, finite sets of *constants*, *function symbols* and *relation symbols*. A *term* is an expression $f(c_1, \ldots, c_n)$ where $f \in \mathcal{F}$, and c_i's are 0 or more constants. An *atom* is an expression $r(c_1, \ldots, c_n)$, where $r \in \mathcal{R}$, and c_i's are constants. The set of all terms (resp., atoms) that can be formed from Σ is denoted by $\mathcal{T}$ (resp., $\mathcal{A}$). A *t-atom* is an expression of the form $f = g$, where f is a term and g is either a term or a constant. We call *seed t-atom* a t-atom of the form $f = v$, where v is a constant. Any t-atom that is not a seed t-atom is a *dependent t-atom*. Thus, given a signature with $\mathcal{C} = \{a, b, 0, 1, 2, 3, 4\}$ and $\mathcal{F} = \{occupancy, seats\}$, expressions $occupancy(a) = 2$ and $seats(b) = 4$ are seed t-atoms, while $occupancy(b) = seats(b)$ is a dependent t-atom.

A *regular literal* is an atom a or its strong negation $\neg a$. A *t-literal* is a t-atom $f = g$ or its strong negation $\neg(f = g)$, which we abbreviate $f \neq g$. A *dependent t-literal* is any t-literal that is not a seed t-atom. A *literal* is a regular literal or a t-literal. A *seed literal* is a regular literal or a seed t-atom. Given a signature with $\mathcal{R} = \{room_evacuated\}$, $\mathcal{F} = \{occupancy, seats\}$ and $\mathcal{C} = \{a, b, 0, \ldots, 4\}$, $room_evacuated(a)$, $\neg room_evacuated(b)$ and $occupancy(a) = 2$ are seed literals (as well as literals); $room_evacuated(a)$ and $\neg room_evacuated(b)$ are also regular literals; $occupancy(b) \neq 1$ and $occupancy(b) = seats(b)$ are dependent t-literals, but they are not regular or seed literals.

A *rule* r is a statement of the form:

$$h \leftarrow l_1, \ldots, l_m, not\ l_{m+1}, \ldots, not\ l_n \tag{5}$$

where h is a seed literal and l_i's are literals. Similarly to ASP, the informal reading of r is that a rational agent who believes $l_1, \ldots, l_m$ and has no reason to believe $l_{m+1}, \ldots, l_n$ must believe h. Given a signature with $\mathcal{R} = \{room_evacuated, door_stuck, room_occupied, room_maybe_occupied\}$, $\mathcal{F} = \{occupancy\}$ and $\mathcal{C} = \{0\}$, the following is an example of ASP{f} rules encoding knowledge about the occupancy of a room:

$r_1 : occupancy = 0 \leftarrow room_evacuated,\ not\ door_stuck.$
$r_2 : room_occupied \leftarrow occupancy \neq 0.$
$r_3 : room_maybe_occupied \leftarrow not\ occupancy = 0.$

Intuitively, rule r_1 states that the occupancy of the room is 0 if the room has been evacuated and there is no reason to believe that the door is stuck. Rule r_2 says that the room is occupied if its occupancy is different from 0. On the other hand, r_3 aims at drawing a weaker conclusion, stating that the room *may* be occupied if there is no explicit knowledge (i.e. reason to believe) that its occupancy is 0.

Given rule r from (5), $head(r)$ denotes $\{h\}$; $body(r)$ denotes $\{l_1, \ldots, not\ l_n\}$; $pos(r)$ denotes $\{l_1, \ldots, l_m\}$; $neg(r)$ denotes $\{l_{m+1}, \ldots, l_n\}$.

A *constraint* is a special type of rule with an empty head, informally meaning that the condition described by the body of the constraint must never be satisfied. A constraint is considered a shorthand of $\bot \leftarrow l_1, \ldots, l_m, not\ l_{m+1}, \ldots, not\ l_n, not\ \bot$, where $\bot$ is a fresh atom.

A *program* is a pair $\Pi = \langle \Sigma, P \rangle$, where Σ is a signature and P is a set of rules. Whenever possible, in this paper the signature is implicitly defined from the rules of Π, and Π is identified with its set of rules. In that case, the signature is denoted by $\Sigma(\Pi)$ and its elements by $\mathcal{C}(\Pi)$, $\mathcal{F}(\Pi)$ and $\mathcal{R}(\Pi)$. A rule r is *positive* if $neg(r) = \emptyset$. A program Π is *positive* if every $r \in \Pi$ is positive. A program Π is also *t-literal free* if no t-literals occur in the rules of Π.

Like in ASP, in ASP{f} too variables can be used in place of constants and terms. The *grounding of a rule* r is the set of all the syntactically valid rules (its *ground instances*) obtained by replacing every variable of r with an element of $\mathcal{C}$. The *grounding of a program* Π is the set of the groundings of the rules of Π. A syntactic element of the language is *ground* if it is variable-free and *non-ground* otherwise.

3 Semantics of ASP{f}

The semantics of a non-ground program is defined to coincide with the semantics of its grounding. The semantics of ground ASP{f} programs is defined below. It is worth noting that the semantics of ASP{f} is obtained from that of ASP in [8] by simply extending entailment to t-literals.

In the rest of this section, we consider only ground terms, literals, rules and programs and thus omit the word "ground." A set S of seed literals is *consistent* if (1) for every atom $a \in \mathcal{A}$, $\{a, \neg a\} \not\subseteq S$; (2) for every term $t \in \mathcal{T}$ and v_1, $v_2 \in \mathcal{C}$ such that $v_1 \neq v_2$, $\{t = v_1, t = v_2\} \not\subseteq S$. Hence, $S_1 = \{p, \neg q, f = 3\}$ and $S_2 = \{q, f = 3, g = 2\}$ are consistent, while $\{p, \neg p, f = 3\}$ and $\{q, f = 3, f = 2\}$ are not. Incidentally, $\{p, \neg q, f = g, g = 2\}$ is not a set of seed literals, because $f = g$ is not a seed literal.

The *value* of a term t w.r.t. a consistent set S of seed literals (denoted by $val_S(t)$) is v iff $t = v \in S$. If, for every $v \in \mathcal{C}$, $t = v \notin S$, the value of t w.r.t. S is *undefined*. The value of a constant $v \in \mathcal{C}$ w.r.t. S ($val_S(v)$) is v itself. For example given S_1 and S_2 as above, $val_{S_2}(f)$ is 3 and $val_{S_2}(g)$ is 2, whereas $val_{S_1}(g)$ is undefined. Given S_1 and a signature with $\mathcal{C} = \{0, 1\}$, $val_{S_1}(1) = 1$.

A seed literal l is *satisfied* by a consistent set S of seed literals iff $l \in S$. A dependent t-literal $f = g$ (resp., $f \neq g$) is *satisfied* by S iff both $val_S(f)$ and $val_S(g)$ are defined, and $val_S(f)$ is equal to $val_S(g)$ (resp., $val_S(f)$ is different from $val_S(g)$). Thus, seed literals q and $f = 3$ are satisfied by S_2; $f \neq g$ is also satisfied by S_2 because $val_{S_2}(f)$ and $val_{S_2}(g)$ are defined, and $val_{S_2}(f)$ is different from $val_{S_2}(g)$. Conversely, $f = g$ is not satisfied, because $val_{S_2}(f)$ is different from $val_{S_2}(g)$. The t-literal $f \neq h$ is also not satisfied by S_2, because $val_{S_2}(h)$ is undefined. When a literal l is satisfied (resp., not satisfied) by S, we write $S \models l$ (resp., $S \not\models l$).

An *extended literal* is a literal l or an expression of the form *not* l. An extended literal *not* l is satisfied by a consistent set S of seed literals ($S \models not\ l$) if $S \not\models l$. Similarly, $S \not\models not\ l$ if $S \models l$. Considering set S_2 again, extended literal *not* $f = h$ is satisfied by S_2, because $f = h$ is not satisfied by S_2.

Finally, a set E of extended literals is satisfied by a consistent set S of seed literals ($S \models E$) if $S \models e$ for every $e \in E$.

We begin by defining the semantics of ASP{f} programs for *positive* programs.

A set S of seed literals is *closed* under positive rule r if $S \models h$, where $head(r) = \{h\}$, whenever $S \models pos(r)$. Hence, set S_2 described earlier is closed under $f = 3 \leftarrow g \neq 1$ and

(trivially) under $f = 2 \leftarrow r$, but it is not closed under $p \leftarrow f = 3$, because $S_2 \models f = 3$ but $S_2 \not\models p$. S is closed under Π if it is closed under every rule $r \in \Pi$.

Finally, a set S of seed literals is an *answer set* of a positive program Π if it is consistent and closed under Π, and is minimal (w.r.t. set-theoretic inclusion) among the sets of seed literals that satisfy such conditions. Thus, the program $\{p \leftarrow f = 2. \quad f = 2. \quad q \leftarrow q.\}$ has one answer sets, $\{f = 2, p\}$. The set $\{f = 2\}$ is not closed under the first rule of the program, and therefore is not an answer set. The set $\{f = 2, p, q\}$ is also not an answer set, because it is not minimal (it is a proper superset of another answer set). Notice that positive programs may have no answer set. For example, the program $\{f = 3 \leftarrow \textit{not } p. \quad f = 2 \leftarrow \textit{not } q.\}$ has no answer set. Programs that have answer sets (resp., no answer sets) are called *consistent* (resp., *inconsistent*).

Positive programs enjoy the following property:

▶ Proposition 1. Every consistent positive ASP{f} program Π has a unique answer set.

Next, we define the semantics of arbitrary ASP{f} programs.

The *reduct* of a program Π w.r.t. a consistent set S of seed literals is the set Π^S consisting of a rule $\textit{head}(r) \leftarrow \textit{pos}(r)$ (the *reduct* of r w.r.t. S) for each rule $r \in \Pi$ for which $S \models \textit{body}(r) \setminus \textit{pos}(r)$.

▶ **Example 1.** Consider a set of seed literals $S_3 = \{g = 3, f = 2, p, q\}$, and program Π_1:

$$r_1 : p \leftarrow f = 2, \textit{not } g = 1, \textit{not } h = 0. \qquad r_2 : q \leftarrow p, \textit{not } g \neq 2.$$
$$r_3 : g = 3. \qquad r_4 : f = 2.$$

and let us compute its reduct. For r_1, first we have to check if $S_3 \models \textit{body}(r_1) \setminus \textit{pos}(r_1)$, that is if $S_3 \models \textit{not } g = 1, \textit{not } h = 0$. Extended literal $\textit{not } g = 1$ is satisfied by S_3 only if $S_3 \not\models g = 1$. Because $g = 1$ is a seed literal, it is satisfied by S_3 if $g = 1 \in S_3$. Since $g = 1 \notin S_3$, we conclude that $S_3 \not\models g = 1$ and thus $\textit{not } g = 1$ is satisfied by S_3. In a similar way, we conclude that $S_3 \models \textit{not } h = 0$. Hence, $S_3 \models \textit{body}(r_1) \setminus \textit{pos}(r_1)$. Therefore, the reduct of r_1 is $p \leftarrow f = 2$. For the reduct of r_2, notice that $\textit{not } g \neq 2$ is not satisfied by S_3. In fact, $S_3 \models \textit{not } g \neq 2$ only if $S_3 \not\models g \neq 2$. However, it is not difficult to show that $S_3 \models g \neq 2$: in fact, $\textit{val}_{S_3}(g)$ is defined and $\textit{val}_{S_3}(g) \neq 2$. Therefore, $\textit{not } g \neq 2$ is not satisfied by S_3, and thus the reduct of Π_1 contains no rule for r_2. The reducts of r_3 and r_4 are the rules themselves. Summing up, $\Pi_1^{S_3}$ is $\{r_1' : p \leftarrow f = 2,\ r_3' : g = 3,\ r_4' : f = 2\}$

Finally, a consistent set S of seed literals is an *answer set* of Π if S is the answer set of Π^S.

▶ **Example 2.** By applying the definitions given earlier, it is not difficult to show that an answer set of $\Pi_1^{S_3}$ is $\{f = 2, g = 3, p\} = S_3$. Hence, S_3 is an answer set of $\Pi_1^{S_3}$. Consider instead $S_4 = S_3 \cup \{h = 1\}$. Clearly $\Pi_1^{S_4} = \Pi_1^{S_3}$. From the uniqueness of the answer sets of positive programs, it follows immediately that S_4 is not an answer set of $\Pi_1^{S_4}$. Therefore, S_4 is not an answer set of Π_1.

4 Knowledge Representation with ASP{f}

In this section we demonstrate the use of ASP{f} for the formalization of key types of knowledge. We start our discussion by addressing the encoding of defaults.

Consider the statements: (1) the value of $f(x)$ is a unless otherwise specified; (2) the value of $f(x)$ is b if $p(x)$ (this example is similar to, and inspired by, one from [10]). These statements

can be encoded in ASP{f} by $P_1 = \{r_1 : f(x) = a \leftarrow not\ f(x) \neq a.,\ r_2 : f(x) = b \leftarrow p(x).\}$. Rule r_1 encodes the default, and r_2 encodes the exception. The informal reading of r_1, according to the description given earlier in this paper, is "if there is no reason to believe that $f(x)$ is different from a, then $f(x)$ must be equal to a".

Extending a common ASP methodology, the choice of value for a non-Herbrand function can be encoded in ASP{f} by means of default negation. Consider the statements (adapted from [10]): (1) the value $f(X)$ is a if $p(X)$; (2) otherwise, the value of $f(X)$ is arbitrary. Let the domain of variable X be given by a relation $dom(X)$, and let the possible values of $f(X)$ be encoded by a relation $val(V)$. A possible ASP{f} encoding of these statements is $\{r_1 : f(X) = a \leftarrow\ p(X),\ dom(X).,\ r_2 : f(X) = V \leftarrow\ dom(X),\ val(V),\ not\ p(X),\ not\ f(X) \neq V.\}$. Rule r_1 encodes the first statement. Rule r_2 formalizes the arbitrary selection of values for $f(X)$ in the default case.

A similar use of defaults is typically associated, in ASP, with the representation of dynamic domains. In this case, defaults are a key tool for the encoding of the law of inertia. Let us show how dynamic domains involving functions can be represented in ASP{f}. Consider a domain including a button b_i, which increments a counter c, and a button b_r, which resets it. At each time step, the agent operating the buttons may press either button, or none. A possible ASP{f} encoding of this domain is:

$$r_1 : val(c, S+1) = 0 \leftarrow\ pressed(b_r, S).$$
$$r_2 : val(c, S+1) = N+1 \leftarrow\ pressed(b_i, S),\ val(c, S) = N.$$
$$r_3 : val(c, S+1) = N \leftarrow\ val(c, S) = N,\ not\ val(c, S+1) \neq val(c, S).$$

Rules r_1 and r_2 are a straightforward encoding of the effect of pressing either button (variable S denotes a time step). Rule r_3 is the ASP{f} encoding of the law of inertia for the value of the counter, and states that the value of c does not change unless it is forced to. For simplicity of presentation, it is instantiated for a particular function, but could be as easily written so that it applies to arbitrary functions from the domain.

Formal results about ASP{f} that are useful for knowledge representation tasks can be found in [1].

5 Computing the Answer Sets of ASP{f} Programs

In this section we describe an algorithm, CLASP{f}, which computes the answer sets of ASP{f} programs. Although CLASP{f} is based on the CLASP algorithm [6], the approach can be easily extended to other ASP solvers. In our description we follow the notation of [6], to which the interested reader can refer for more details on the CLASP algorithm.

As customary, the algorithm operates on ground programs. To keep the presentation simple, we further assume that every program Π considered in this section contains, for every atom a from Π, a constraint $\leftarrow a, \neg a$ (usually this constraint is added automatically by the solver).

Given a literal l, a *signed literal* is an expression of the form $\mathbf{T}l$ or $\mathbf{F}l$. Given a signed literal σ, $\overline{\sigma}$, called the *complement* of σ, denotes $\mathbf{F}l$ if σ is $\mathbf{T}l$, and $\mathbf{T}l$ otherwise. An assignment A over some domain D is a sequence $\langle \sigma_1, \ldots, \sigma_n \rangle$ of signed literals for literals from D. The domain of A is denoted by $dom(A)$. The expression $A \circ B$ denotes the concatenation of assignments A and B. For an assignment A, we denote by A^T the set of literals l such that $\mathbf{T}l$ occurs in A; A^F is instead the set of literals l such that $\mathbf{F}l$ occurs in A.

A *nogood* is a set $\{\sigma_1, \ldots, \sigma_n\}$ of signed literals. An assignment A is a *solution* for a set Δ of nogoods if (1) $A^T \cup A^F = dom(A)$; (2) $A^T \cap A^F = \emptyset$; and (3) for every $\delta \in \Delta$, $\delta \not\subseteq A$. Given

a nogood δ, a signed literal $\sigma \in \delta$ and an assignment A, $\overline{\sigma}$ is called *unit-resulting* for δ w.r.t. A if $\delta \setminus A = \{\sigma\}$ and $\overline{\sigma} \notin A$. *Unit propagation* is the process of iteratively extending A with unit-resulting signed literals until no signed literal is unit-resulting for any nogood in Δ.

At the core of the computation of the answer sets of a program in CLASP{f} is the process of mapping the program to a suitable set of nogoods. Such mapping is described next, beginning with the nogoods already used in CLASP.

Given a program Π, let $lit(\Pi)$ be the set of literals that occur in Π, $seed(\Pi)$ the set of seed literals that occur in Π, and $body(\Pi)$ be the collection of the bodies of the rules of Π. Furthermore, let the expression $body(l)$ denote the set of rules of Π whose head is l.

Given a rule's body $\beta = \{l_1, \dots, l_m, not\ l_{m+1}, \dots, not\ l_n\}$, the expression $\delta(\beta)$ denotes the nogood $\{\mathbf{F}\beta, \mathbf{T}l_1, \dots, \mathbf{T}l_m, \mathbf{F}l_{m+1}, \dots, \mathbf{F}l_n\}$. The expression $\Delta(\beta)$ denotes instead the set of nogoods $\{\{\{\mathbf{T}\beta, \mathbf{F}l_1\}, \dots, \{\mathbf{T}\beta, \mathbf{F}l_m\},\ \{\mathbf{T}\beta, \mathbf{T}l_{m+1}\}, \dots, \{\mathbf{T}\beta, \mathbf{T}l_n\}\}\}$.

Next, given a literal l such that $body(l) = \{\beta_1, \dots, \beta_k\}$, the expression $\Delta(l)$ denotes the set of nogoods $\{\{\mathbf{F}l, \mathbf{T}\beta_1\}, \dots, \{\mathbf{F}l, \mathbf{T}\beta_k\}\}$. Finally, $\delta(l) = \{\mathbf{T}l, \mathbf{F}\beta_1, \dots, \mathbf{F}\beta_k\}$.

Given a program Π, let Δ_Π denote $\{\{\delta(\beta) \mid \beta \in body(\Pi)\} \cup\ \{\delta \in \Delta(\beta) \mid \beta \in body(\Pi)\}\ \cup$ $\{\delta(l) \mid l \in seed(\Pi)\} \cup\ \{\delta \in \Delta(l) \mid l \in lit(\Pi)\}\}$. Intuitively, in Δ_Π, $\delta(l)$ is applied only to seed t-atoms because dependent t-literals do not occur in the head of rules.

It can be shown [6] that Δ_Π can be used to find the answer sets of tight, t-literal free, programs. To find the answer sets of non-tight programs, one needs to introduce *loop nogoods*. For a program Π and some $U \subseteq lit(\Pi)$, expression $EB_\Pi(U)$ denotes the collection of the *external bodies* of U, i.e. $\{body(r) \mid r \in \Pi, head(r) \in U, body(r) \cap U = \emptyset\}$. Given a literal $l \in U$ and $EB_\Pi(U) = \{\beta_1, \dots, \beta_k\}$, the *loop nogood* of l is $\lambda(l, U) = \{\mathbf{F}\beta_1, \dots, \mathbf{F}\beta_k, \mathbf{T}l\}$. The set of loop nogoods for program Π is $\Lambda_\Pi = \bigcup_{U \subseteq lit(\Pi), U \neq \emptyset} \{\lambda(l, U) \mid l \in U\}$. The following property follows from a similar result from [6]:

▶ **Theorem 3.** *For every ASP{f} program* Π that contains no dependent t-literals*,* $X \subseteq lit(\Pi)$ *is an answer set of* Π *iff* $X = A^T \cap lit(\Pi)$ *for a solution* A *for* $\Delta_\Pi \cup \Lambda_\Pi$*.*

Next, we introduce nogoods for the computation of the answer sets of programs containing dependent t-literals. Given a dependent t-literal l of the form $f = g$ (resp., $f \neq g$), a pair of seed t-atoms $f = v$ and $g = w$ formed from $\Sigma(\Pi)$ is a *satisfying pair for* l if $v = w$ (resp., $v \neq w$) and a *falsifying pair for* l otherwise. Let $\{\langle f = v_1, g = w_1\rangle, \dots \langle f = v_k, g = w_k\rangle\}$ be the set of satisfying pairs for l. The expression $\rho^+(l)$ denotes the set of nogoods $\{\{\mathbf{F}l, \mathbf{T}f = v_1, \mathbf{T}g = w_1\}, \dots, \{\mathbf{F}l, \mathbf{T}f = v_k, \mathbf{T}g = v_k\}\}$. Let $\{\langle f = v_1, g = w_1\rangle, \dots \langle f = v_k, g = w_k\rangle\}$ be the set of falsifying pairs for l. The expression $\rho^-(l)$ denotes the set of nogoods $\{\{\mathbf{T}l, \mathbf{T}f = v_1, \mathbf{T}g = w_1\}, \dots, \{\mathbf{T}l, \mathbf{T}f = v_k, \mathbf{T}g = v_k\}\}$. Intuitively the nogoods in $\rho^+(l)$ and $\rho^-(l)$ enforce the truth or falsity of a dependent t-literal when suitable seed t-atoms are true.

Finally, given a dependent t-literal l, let $terms(l)$ denote the set of terms that occur in l, and, for every term f that occurs in l, let $rel(f)$ denote the set of seed t-atoms of the form $f = v$ for some $v \in \mathcal{C}(\Pi)$. Intuitively $rel(f)$ is the set of seed t-atoms that are relevant to the value of term f. The expression $\kappa(l)$ denotes the set of nogoods $\bigcup_{f \in terms(l)} (\{\mathbf{T}l\} \cup \{\mathbf{F}s \mid s \in rel(f)\})$. Intuitively $\kappa(l)$ states that l cannot be true if one of its terms is undefined.

Let $dep(\Pi)$ be the set of dependent t-literals in a program Π. Θ_Π denotes $\{\rho^+(l) \mid l \in dep(\Pi)\} \cup\ \{\rho^-(l) \mid l \in dep(\Pi)\} \cup\ \{\kappa^-(l) \mid l \in dep(\Pi)\}$.

The following condition defines a (rather large) class of ASP{f} programs whose answer sets can be found using Θ_Π. Given a program Π, we say that Π contains a *t-loop* for seed t-atom

l if, in the dependency graph for Π, there is a positive path from l to a t-literal l' such that $terms(l) \cap terms(l') \neq \emptyset$. A program containing a t-loop is for example $f = 2 \leftarrow f \neq 3$. In practice, for most domains from the literature there appear to be t-loop free encodings. The following result characterizes the answer sets of t-loop free programs.

▶ **Theorem 4.** *For every t-loop free ASP{f} program* Π*,* $X \subseteq seed(\Pi)$ *is an answer set of* Π *iff* $X = A^T \cap seed(\Pi)$ *for a solution* A *for* $\Delta_\Pi \cup \Lambda_\Pi \cup \Theta_\Pi$*.*

From a high-level perspective, in the CLASP algorithm the answer sets of ASP programs are computed by iteratively (1) performing unit propagation on the nogoods for the program and (2) non-deterministically assigning a truth value to a signed literal. Unfortunately, performing unit propagation on the nogoods in Θ_Π is inefficient, because in the worst case sets $\rho^+(l)$ and $\rho^-(l)$ exhibit quadratic growth. However, the conditions expressed by those nogoods can be easily checked algorithmically. Let VALUE(f, A) be a function that returns v if signed literal $\mathbf{T}f = v$ occurs in assignment A. Given A and a dependent t-literal $f = g$, unit propagation on $\rho^+(f = g)$ can be performed by checking if VALUE(f, A) = VALUE(g, A) and, if so, by adding $\mathbf{T}f = g$ to A. A similar approach applies to the unit propagation for the other elements of Θ_Π.

Using this technique, unit propagation on the nogoods of Θ_Π can be performed in constant time w.r.t. the number of seed t-atoms in the program. (The reader may be wondering about the cases such as the one in which the truth of $\mathbf{T}f = v$ together with VALUE(f, A) can be used to infer VALUE(g, A). It can be shown that support for this type of scenario can be dropped without affecting the soundness and completeness of the solver.)

Function fLocalPropagation(Π, ∇, A), shown below, iteratively augments the result of unit propagation from CLASP's function LocalPropagation(Π, ∇, A) with the unit-resulting dependent t-literals derived from Θ_Π. The iterations continue until a fixpoint is reached. (Function LocalPropagation(Π, ∇, A) in CLASP computes a fixpoint of unit propagation by adding to assignment A the unit-resulting literals derived from nogoods in Δ_Π and in ∇.)

Function: fLocalPropagation
Input: program Π, set ∇ of nogoods, assignment A
Output: an extended assignment and a set of nogoods
$U \leftarrow \emptyset$
loop
  $B \leftarrow$ LocalPropagation(Π, ∇, A)
  $A \leftarrow$ LocalPropagation$_\Theta(\Pi, \nabla, B)$
  if $A = B$ then return A

The algorithm for nogood propagation from [6] is modified by replacing the call to LocalPropagation by a call to fLocalPropagation. The main algorithm of CLASP{f} is obtained in a similar way from algorithm CDNL-ASP from [6].

6 Experimental Results

To evaluate the performance of the CLASP{f} algorithm, we have compared it with the method for computing the answer sets of programs with non-Herbrand functions used in [4] and [10]. In that method, given a program Π with non-Herbrand functions, (1) all occurrences of

t-literals are replaced by regular ASP literals (e.g. $f = g$ is replaced by $eq(f, g)$), and (2) suitable equality and inequality axioms are added to Π. The answer sets of the resulting program are then computed using an ASP solver. It can be shown that the answer sets of the translation encode the answer sets of Π.

For our comparison we have chosen a planning task in which an agent starts at $(0, 0)$ on a $n \times n$ grid and has the goal of reaching a given position in k steps. The agent can move either up or to the right, by one cell at a time. Concurrent actions are not allowed. To make the task more challenging, the goal position is chosen so that the minimum number of actions needed to achieve the goal is equal to number of steps k. This domain has been selected because, in our experience on practical applications of ASP, solver performance decreases rapidly when parameter n is increased. This decrease in performance is due to the growth in the size of the grounding of the inertia axiom, and we are aware of no general-purpose technique to alleviate this issue in ASP programs.

The ASP{f} formalization, $\Pi_{\text{ASP\{f\}}}$ is show below. Constants k and n are specified at run-time. Symbol / used in the second-to-last rule denotes integer division in the dialect of CLASP.

$$
\begin{array}{l}
step(0..k).\;\; loc(0..n-1).\;\; posx(0) = 0.\;\; posy(0) = 0. \\
posx(S+1) = X + 1 \leftarrow \\
\quad step(S),\; step(S+1),\; loc(X),\; loc(X+1),\; posx(S) = X,\; o(plusx, S). \\
\leftarrow o(plusx, S), posx(S) = n - 1. \\
posy(S+1) = Y + 1 \leftarrow \\
\quad step(S),\; step(S+1),\; loc(Y),\; loc(Y+1),\; posy(S) = Y,\; o(plusy, S). \\
\leftarrow o(plusy, S), posy(S) = n - 1. \\
posx(S+1) = X \leftarrow \\
\quad step(S),\; step(S+1),\; loc(X),\; posx(S) = X,\; not\ posx(S+1) \neq posx(S). \\
posy(S+1) = Y \leftarrow \\
\quad step(S),\; step(S+1),\; loc(Y),\; posy(S) = Y,\; not\ posy(S+1) \neq posy(S). \\
1\{o(plusx, S), o(plusy, S)\}1 \leftarrow step(S),\; S < k. \\
goal \leftarrow posx(k) = k/2,\; posy(k) = k - k/2. \\
\leftarrow not\ goal.
\end{array}
$$

Program Π_{ASP}, omitted to save space, is an ASP encoding of the problem obtained by the usual formalization techniques; it is also equivalent, modulo renaming and reification of relations, to the translation of the formalizations in the languages of [4] and [10]. Table 1 shows a comparison of the time, in seconds, to find one answer set using $\Pi_{\text{ASP\{f\}}}$ and using Π_{ASP}. The results have been obtained for various values of parameters k and n. As the table shows, the time for $\Pi_{\text{ASP\{f\}}}$ is consistently more than an order of magnitude better than of Π_{ASP}, even though the code for the support of non-Herbrand functions in the implementation of CLASP{f} is still largely unoptimized. The CLASP{f} solver used here is an extension of CLINGO 2.0.2. To ensure the fairness of the comparison, the answer sets of the ASP encoding have been computed using CLINGO 2.0.2. The experiments were performed on a computer with an Intel Q6600 processor at 2.4GHz, 1.5GB RAM and Linux Fedora Core 11.

7 Conclusions and Future Work

In this paper we have defined the syntax and semantics of an extension of ASP by non-Herbrand functions. Although the semantics of our language is a comparatively small modification of the semantics of ASP from [8], it allows for an efficient implementation in

Table 1 Performance comparison between $\Pi_{\text{ASP}\{f\}}$ + CLASP{f} and Π_{ASP} + CLINGO.

	$k=3$		$k=5$		$k=7$	
n	$\Pi_{\text{ASP}\{f\}}$	Π_{ASP}	$\Pi_{\text{ASP}\{f\}}$	Π_{ASP}	$\Pi_{\text{ASP}\{f\}}$	Π_{ASP}
100	0.000	0.045	0.011	0.063	0.018	0.108
200	0.016	0.282	0.044	0.467	0.076	0.555
500	0.115	1.919	0.212	3.149	0.458	4.530
1000	0.513	8.273	1.012	13.787	1.766	21.432
1500	1.203	21.300	2.515	37.024	4.626	56.341
2000	2.429	43.092	4.283	70.591	7.712	103.737

ASP solvers, as demonstrated by our experimental comparison with the solving techniques for other languages supporting non-Herbrand functions. Although the language of [11, 14] is also supported by an efficient solver, that solver uses CSP solving techniques rather than ASP solving techniques. Currently, the ASP{f} solving algorithm is only applicable to a (large) subclass of ASP{f} programs. We expect that it will be possible to extend our algorithm to arbitrary programs by introducing additional nogoods.

References

1. Marcello Balduccini. *Correct Reasoning: Essays on Logic-Based AI in Honour of Vladimir Lifschitz*, chapter 3. A “Conservative” Approach to Extending Answer Set Programming with Non-Herbrand Functions, pages 23–39. Lecture Notes in Artificial Intelligence (LNCS). Springer Verlag, Berlin, Jun 2012.
2. Chitta Baral. *Knowledge Representation, Reasoning, and Declarative Problem Solving*. Cambridge University Press, Jan 2003.
3. Sabrina Baselice and Piero A. Bonatti. A Decidable Subclass of Finitary Programs. *Journal of Theory and Practice of Logic Programming (TPLP)*, 10(4–6):481–496, 2010.
4. Pedro Cabalar. Functional Answer Set Programming. *Journal of Theory and Practice of Logic Programming (TPLP)*, 11:203–234, 2011.
5. Francesco Calimeri, Susanna Cozza, Giovanbattista Ianni, and Nicola Leone. Enhancing ASP by Functions: Decidable Classes and Implementation Techniques. In *Proceedings of the Twenty-Fourth Conference on Artificial Intelligence*, pages 1666–1670, 2010.
6. Martin Gebser, Benjamin Kaufmann, Andre Neumann, and Torsten Schaub. Conflict-Driven Answer Set Solving. In Manuela M. Veloso, editor, *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI'07)*, pages 386–392, 2007.
7. Martin Gebser, Max Ostrowski, and Torsten Schaub. Constraint Answer Set Solving. In *25th International Conference on Logic Programming (ICLP09)*, volume 5649, 2009.
8. Michael Gelfond and Vladimir Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, 9:365–385, 1991.
9. Eugene Goldberg and Yakov Novikov. BerkMin: A Fast and Robust Sat-Solver. In *Proceedings of Design, Automation and Test in Europe Conference (DATE-2002)*, pages 142–149, Mar 2002.
10. Vladimir Lifschitz. Logic Programs with Intensional Functions (Preliminary Report). In *ICLP11 Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP11)*, Jul 2011.
11. Fangzhen Lin and Yisong Wang. Answer Set Programming with Functions. In *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning (KR2008)*, pages 454–465, 2008.

12 Victor W. Marek and Miroslaw Truszczynski. *The Logic Programming Paradigm: a 25-Year Perspective*, chapter Stable Models and an Alternative Logic Programming Paradigm, pages 375–398. Springer Verlag, Berlin, 1999.
13 Tommi Syrjänen. Omega-Restricted Logic Programs. In Thomas Eiter, Wolfgang Faber, and Miroslaw Truszczynski, editors, *6th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR01)*, volume 2173 of *Lecture Notes in Artificial Intelligence (LNCS)*, pages 267–279. Springer Verlag, Berlin, 2001.
14 Yisong Wang, Jia-Huai You, Li-Yan Yuan, and Mingyi Zhang. Weight Constraint Programs with Functions. In Esra Erdem, Fangzhen Lin, and Torsten Schaub, editors, *10th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR09)*, volume 5753 of *Lecture Notes in Artificial Intelligence (LNCS)*, pages 329–341. Springer Verlag, Berlin, Sep 2009.

Stable Models of Formulas with Generalized Quantifiers (Preliminary Report)

Joohyung Lee and Yunsong Meng

School of Computing, Informatics, and Decision Systems Engineering
Arizona State University, Tempe, AZ, USA
`joolee@asu.edu, Yunsong.Meng@asu.edu`

Abstract

Applications of answer set programming motivated various extensions of the stable model semantics, for instance, to allow aggregates or to facilitate interface with external ontology descriptions. We present a uniform, reductive view on these extensions by viewing them as special cases of formulas with generalized quantifiers. This is done by extending the first-order stable model semantics by Ferraris, Lee and Lifschitz to account for generalized quantifiers and then by reducing the individual extensions to this formalism.

1998 ACM Subject Classification I.2.4 Knowledge Representation Formalisms and Methods

Keywords and phrases answer set programming, stable model semantics, generalized quantifiers

Digital Object Identifier 10.4230/LIPIcs.ICLP.2012.61

1 Introduction

Applications of answer set programming motivated various recent extensions of the stable model semantics, for instance, to allow aggregates [4, 8, 15], or to facilitate interface with external ontology descriptions [3]. While the extensions were driven by different motivations and applications, a common underlying issue is how to extend the stable model semantics to incorporate "complex atoms," such as "aggregate atoms" and "dl-atoms."

Most extensions involve grounding. For instance, assuming that the domain is $\{1, 2, \dots\}$ the rule

$$q(y) \leftarrow \#\text{COUNT}\{x.p(x, y)\} \geq 2 \tag{1}$$

can be understood as a schema for ground instances

$$q(1) \leftarrow \#\text{COUNT}\{1.p(1,1), 2.p(2,1), \dots\} \geq 2$$
$$q(2) \leftarrow \#\text{COUNT}\{1.p(1,2), 2.p(2,2), \dots\} \geq 2$$
$$\dots$$

Here y is called a "global" variable, and x is called a "local" variable. Replacing a global variable by ground terms increases the number of rules; replacing a local variable by ground terms increases the size of each rule.

Instead of involving grounding, in [10], a simple approach to understanding the meaning of the count aggregate in answer set programming was provided by reduction to first-order formulas under the stable model semantics [6, 7]. For instance, rule (1) can be understood as the first-order formula

$$\forall y(\exists x_1 x_2(p(x_1) \wedge p(x_2) \wedge \neg(x_1 = x_2)) \rightarrow q(y)) \ ,$$

in which quantifiers are introduced to account for local variables in aggregates.

© Joohyung Lee and Yunsong Meng;
licensed under Creative Commons License ND
Technical Communications of the 28th International Conference on Logic Programming (ICLP'12).
Editors: A. Dovier and V. Santos Costa; pp. 61–71

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

An attempt to extend this approach to handle arbitrary nonmonotone aggregates encounters some difficulty, as the quantifiers $\forall$ and $\exists$, like its propositional counterpart $\wedge$ and $\vee$, are "monotone."

It is hinted in [5] that aggregates may be viewed in terms of generalized quantifiers—a generalization of the standard quantifiers, $\forall$ and $\exists$, introduced by Mostowski [13]. We follow up on that suggestion, and extend the stable model semantics by [7] to allow generalized quantifiers.

It turns out that generalized quantifiers are not only useful in explaining the meaning of arbitrary aggregates, but also useful in explaining other recent extensions of the stable model semantics, such as nonmonotonic dl-programs [3]. This allows us to combine the individual extensions in a single language as in the following example.

▶ **Example 1.** We consider an extension of nonmonotonic dl-programs $(\mathcal{T}, \Pi)$ that allows aggregates. For instance, the ontology description $\mathcal{T}$ specifies that every married man has a spouse who is a woman, and similarly for a married woman:

$$\begin{array}{lcl} \mathit{Man} \sqcap \mathit{Married} & \sqsubseteq & \exists \mathit{Spouse}.\mathit{Woman}. \\ \mathit{Woman} \sqcap \mathit{Married} & \sqsubseteq & \exists \mathit{Spouse}.\mathit{Man}. \end{array}$$

The following program Π counts the number of people who are eligible for an insurance discount:

$$\begin{array}{l} \mathit{discount}(x) \leftarrow \mathit{not}\ \mathit{accident}(x), \\ \quad \#\mathit{dl}[\mathit{Man} \uplus \mathit{mm}, \mathit{Married} \uplus \mathit{mm}, \mathit{Woman} \uplus \mathit{mw}, \mathit{Married} \uplus \mathit{mw}; \exists \mathit{Spouse}.\top](x). \\ \mathit{discount}(x) \leftarrow \mathit{discount}(y), \mathit{family}(y, x), \mathit{not}\ \mathit{accident}(x). \\ \mathit{numOfDiscount}(z) \leftarrow \mathrm{COUNT}\langle x.\mathit{discount}(x)\rangle = z. \end{array}$$

The first rule asserts that everybody who has a spouse and has no accident is eligible for a discount. The second rule asserts that everybody who has no accident and has a family member with a discount is eligible for a discount.

The paper is organized as follows. We first review the syntax and the semantics of formulas with generalized quantifiers (GQ-formulas). Next we define stable models of GQ-formulas, and then show the individual extensions of the stable model semantics, such as logic programs with aggregates and/or nonmonotonic dl-atoms, can be viewed as special cases of GQ-formulas.

2 Preliminaries

2.1 Syntax of Formulas with Generalized Quantifiers

We follow the definition of a GQ-formula from [16, Section 5] (that is to say, with Lindström quantifiers [12] without the isomorphism closure condition).

As in first-order logic, a *signature* σ is a set of symbols consisting of *function constants* and *predicate constants*. Each symbol is assigned a nonnegative integer, called the arity. Function constants with arity 0 are called *object constants*, and predicate constants with arity 0 are called *propositional constants*. A *term* is an object variable or $f(t_1, \dots, t_n)$, where f is a function constant in σ of arity n, and t_i are terms. An *atomic formula* is an expression of the form $p(t_1, \dots, t_n)$ or $t_1 = t_2$, where p is a predicate constant in σ of arity n.

We assume a set $\mathbf{Q}$ of symbols for generalized quantifiers. Each symbol in $\mathbf{Q}$ is associated with a tuple of nonnegative integers $\langle n_1, \dots, n_k\rangle$ ($k \geq 0$, and each n_i is ≥ 0), called the *type*. A *GQ-formula (with the set* $\mathbf{Q}$ *of generalized quantifiers)* is defined in a recursive way:

- an atomic formula is a GQ-formula;
- if $F_1, \dots, F_k$ ($k \geq 0$) are GQ-formulas and Q is a generalized quantifier of type $\langle n_1, \dots, n_k \rangle$ in $\mathbf{Q}$, then

  $$Q[\mathbf{x}_1] \dots [\mathbf{x}_k](F_1(\mathbf{x}_1), \dots, F_k(\mathbf{x}_k)) \quad (2)$$

  is a GQ-formula, where each $\mathbf{x}_i$ ($1 \leq i \leq k$) is a list of distinct object variables whose length is n_i.

We say that an occurrence of a variable x in a GQ-formula F is *bound* if it belongs to a subformula of F that has the form $Q[\mathbf{x}_1] \dots [\mathbf{x}_k](F_1(\mathbf{x}_1), \dots, F_k(\mathbf{x}_k))$ such that x is in some $\mathbf{x}_i$. Otherwise the occurrence is *free*. We say that x is *free* in F if F contains a free occurrence of x. A *GQ-sentence* is a GQ-formula with no free variables. Notice that the distinction between free and bound variables is similar to that of global and local variables informally described in the introduction.

We assume that $\mathbf{Q}$ contains a type $\langle\rangle$ quantifier $Q_\bot$, a type $\langle 0 \rangle$ quantifier $Q_\neg$, type $\langle 0, 0 \rangle$ quantifiers $Q_\wedge, Q_\vee, Q_\to$, and type $\langle 1 \rangle$ quantifiers $Q_\forall, Q_\exists$. Each of them corresponds to the standard propositional connectives and quantifiers, $\bot, \neg, \wedge, \vee, \to, \forall, \exists$. These generalized quantifiers will often be written in the familiar form. For example, we write $F \wedge G$ in place of $Q_\wedge[][](F, G)$, and write $\forall x F(x)$ in place of $Q_\forall[x](F(x))$.

2.2 Models of GQ-Formulas

As in first-order logic, an interpretation I of a signature σ consists of a nonempty set U, called the *universe* of I, and a mapping c^I for each constant c in σ. For each function constant f of σ whose arity is n, f^I is an element of U if n is 0, and is a function from U^n to U otherwise. For each predicate constant p of σ whose arity is n, p^I is an element of $\{\mathbf{t}, \mathbf{f}\}$ if n is 0, and is a function from U^n to $\{\mathbf{t}, \mathbf{f}\}$ otherwise. For each generalized quantifier Q of type $\langle n_1, \dots, n_k \rangle$, Q^U is a function from $\mathcal{P}(U^{n_1}) \times \dots \times \mathcal{P}(U^{n_k})$ to $\{\mathbf{t}, \mathbf{f}\}$, where $\mathcal{P}(U^{n_i})$ denotes the power set of U^{n_i}.

▶ **Example 2.** Besides the standard propositional connectives and quantifiers, the following are other examples of generalized quantifiers.

- type $\langle 1 \rangle$ quantifier $Q_{\leq 2}$ such that $Q^U_{\leq 2}(R) = \mathbf{t}$ iff the cardinality of R is ≤ 2; [1]
- type $\langle 1 \rangle$ quantifier $Q_{majority}$ such that $Q^U_{majority}(R) = \mathbf{t}$ iff the cardinality of R is greater than the cardinality of $U \setminus R$;
- type $\langle 2, 1, 1 \rangle$ reachability quantifier Q_{reach} such that $Q^U_{reach}(R_1, R_2, R_3) = \mathbf{t}$ iff there are some $u, v \in U$ such that $R_2 = \{u\}$, $R_3 = \{v\}$, and (u, v) belongs to the transitive closure of R_1.

By σ^I we mean the signature obtained from σ by adding new object constants ξ^*, called *names*, for every element ξ in the universe of I. We identify an interpretation I of σ with its extension to σ^I defined by $I(\xi^*) = \xi$. For any term t of σ^I that does not contain variables, we define recursively the element t^I of the universe that is assigned to t by I. If t is an object constant then t^I is an element of U. For other terms, t^I is defined by the equation

$$f(t_1, \dots, t_n)^I = f^I(t_1^I, \dots, t_n^I)$$

for all function constants f of arity $n > 0$.

Given a GQ-sentence F of σ^I, F^I is defined recursively as follows:

[1] It is clear from the type that R is any subset of U. We will skip such explanation.

- $p(t_1, \dots, t_n)^I = p^I(t_1^I, \dots, t_n^I)$,
- $(t_1 = t_2)^I = (t_1^I = t_2^I)$,
- For a generalized quantifier Q of type $\langle n_1, \dots, n_k \rangle$,

$$(Q[\mathbf{x}_1] \dots [\mathbf{x}_k](F_1(\mathbf{x}_1), \dots, F_k(\mathbf{x}_k)))^I = Q^U((\mathbf{x}_1.F_1(\mathbf{x}_1))^I, \dots, (\mathbf{x}_k.F_k(\mathbf{x}_k))^I),$$

where $(\mathbf{x}_i.F_i(\mathbf{x}_i))^I = \{\boldsymbol{\xi} \in U^{n_i} \mid (F_i(\boldsymbol{\xi}^*))^I = \mathbf{t}\}$.

We assume that, for the standard propositional connectives and quantifiers Q, functions Q^U have the standard meaning:

- $Q^U_\forall(R) = \mathbf{t}$ iff $R = U$; $\quad Q^U_\exists(R) = \mathbf{t}$ iff $R \cap U \neq \emptyset$;
- $Q^U_\wedge(R_1, R_2) = \mathbf{t}$ iff $R_1 = R_2 = \{\epsilon\}$;[2] $\quad Q^U_\vee(R_1, R_2) = \mathbf{t}$ iff $R_1 = \{\epsilon\}$ or $R_2 = \{\epsilon\}$;
- $Q^U_\to(R_1, R_2) = \mathbf{t}$ iff $R_1 = \emptyset$ or $R_2 = \{\epsilon\}$;
- $Q^U_\neg(R) = \mathbf{t}$ iff $R = \emptyset$;
- $Q^U_\bot() = \mathbf{f}$.

We say that an interpretation I *satisfies* a GQ-sentence F, or is a *model* of F, and write $I \models F$, if $F^I = \mathbf{t}$. A GQ-sentence F is *logically valid* if every interpretation satisfies F. A GQ-formula with free variables is said to be *logically valid* if its universal closure is logically valid.

▶ **Example 3.** Let I_1 be an interpretation whose universe is $\{1, 2, 3, 4\}$ and let p be a unary predicate constant such that $p(\xi^*)^{I_1} = \mathbf{t}$ iff $\xi \in \{1, 2, 3\}$. We check that I_1 satisfies GQ-sentence $F = \neg Q_{\leq 2}[x]\ p(x) \to Q_{majority}[y]\ p(y)$ ("if p does not contain at most two elements in the universe, then p contains a majority"). Let I_2 be another interpretation with the same universe such that $p(\xi^*)^{I_2} = \mathbf{t}$ iff $\xi \in \{1\}$. It is clear that I_2 also satisfies F.

We say that a generalized quantifier Q is *monotone in the i-th argument position* if the following holds for any universe U: if $Q^U(R_1, \dots, R_k) = \mathbf{t}$ and $R_i \subseteq R'_i \subseteq U^{n_i}$, then $Q^U(R_1, \dots, R_{i-1}, R'_i, R_{i+1}, \dots, R_k) = \mathbf{t}$. Similarly, we say that Q is *anti-monotone in the i-th argument position* if the following holds for any universe U: if $Q^U(R_1, \dots, R_k) = \mathbf{t}$ and $R'_i \subseteq R_i \subseteq U^{n_i}$, then $Q^U(R_1, \dots, R_{i-1}, R'_i, R_{i+1}, \dots, R_k) = \mathbf{t}$. We call an argument position of Q *monotone (anti-monotone)* if Q is monotone (anti-monotone) in that argument position.

Let M be a subset of $\{1, \dots, k\}$. We say that Q is *monotone in M* if Q is monotone in the i-th argument position for all i in M. It is easy to check that both $Q_\wedge$ and $Q_\vee$ are monotone in $\{1, 2\}$. $Q_\to$ is anti-monotone in $\{1\}$ and monotone in $\{2\}$; $Q_\neg$ is anti-monotone in $\{1\}$. In Example 2, $Q_{\leq 2}$ is anti-monotone in $\{1\}$ and $Q_{majority}$ is monotone in $\{1\}$.

Predicate variables can be added to the language in the usual way as we define the standard second-order logic. Syntactically, n-ary predicate variables are used to form atomic formulas in the same way as n-ary predicate constants. Semantically, these variables range over arbitrary truth-valued functions on U^n.

3 Stable Models of GQ-Formulas

We now define the stable model operator SM with a list of intensional predicates. Let $\mathbf{p}$ be a list of distinct predicate constants $p_1, \dots, p_n$, and let $\mathbf{u}$ be a list of distinct predicate

[2] ϵ denotes the empty tuple. For any interpretation I, $U^0 = \{\epsilon\}$. For I to satisfy $Q_\wedge[][](F, G)$, both $(\epsilon.F)^I$ and $(\epsilon.G)^I$ have to be $\{\epsilon\}$, which means that $F^I = G^I = \mathbf{t}$.

variables $u_1, \dots, u_n$. By $\mathbf{u} \le \mathbf{p}$ we denote the conjunction of the formulas $\forall \mathbf{x}(u_i(\mathbf{x}) \to p_i(\mathbf{x}))$ for all $i = 1, \dots, n$, where $\mathbf{x}$ is a list of distinct object variables of the same length as the arity of p_i, and by $\mathbf{u} < \mathbf{p}$ we denote $(\mathbf{u} \le \mathbf{p}) \wedge \neg(\mathbf{p} \le \mathbf{u})$. For instance, if p and q are unary predicate constants then $(u, v) < (p, q)$ is

$$\forall x(u(x) \to p(x)) \wedge \forall x(v(x) \to q(x)) \wedge \neg\Big(\forall x(p(x) \to u(x)) \wedge \forall x(q(x) \to v(x))\Big).$$

For any GQ-formula F and any list of predicates $\mathbf{p} = (p_1, \dots, p_n)$, expression $\text{SM}[F; \mathbf{p}]$ is defined as

$$F \wedge \neg\exists \mathbf{u}((\mathbf{u} < \mathbf{p}) \wedge F^*(\mathbf{u})), \tag{3}$$

where $F^*(\mathbf{u})$ is defined recursively:

- $p_i(\mathbf{t})^* = u_i(\mathbf{t})$ for any list $\mathbf{t}$ of terms;
- $F^* = F$ for any atomic formula F that does not contain members of $\mathbf{p}$;
- $$\begin{aligned}(Q[\mathbf{x}_1] \dots [\mathbf{x}_k](F_1(\mathbf{x}_1), \dots, F_k(\mathbf{x}_k)))^* = \\ Q[\mathbf{x}_1] \dots [\mathbf{x}_k](F_1^*(\mathbf{x}_1), \dots, F_k^*(\mathbf{x}_k)) \wedge Q[\mathbf{x}_1] \dots [\mathbf{x}_k](F_1(\mathbf{x}_1), \dots, F_k(\mathbf{x}_k)).\end{aligned} \tag{4}$$

When F is a GQ-sentence, the models of $\text{SM}[F; \mathbf{p}]$ are called the $\mathbf{p}$-*stable* models of F: they are the models of F that are "stable" on $\mathbf{p}$. We often simply write $\text{SM}[F]$ in place of $\text{SM}[F; \mathbf{p}]$ when $\mathbf{p}$ is the list of all predicate constants occurring in F, and call $\mathbf{p}$-stable models simply stable models.

▶ Proposition 1. Let $Q[\mathbf{x}_1] \dots [\mathbf{x}_k](F_1(\mathbf{x}_1), \dots, F_k(\mathbf{x}_k))$ be a GQ-formula and let M be a subset of $\{1, \dots, k\}$ such that every predicate constant from $\mathbf{p}$ occurs in some F_j where $j \in M$.

(a) If Q is monotone in M, then

$$\mathbf{u} \le \mathbf{p} \to ((Q[\mathbf{x}_1] \dots [\mathbf{x}_k](F_1(\mathbf{x}_1), \dots, F_k(\mathbf{x}_k)))^* \leftrightarrow Q[\mathbf{x}_1] \dots [\mathbf{x}_k](F_1^*(\mathbf{x}_1), \dots, F_k^*(\mathbf{x}_k)))$$

is logically valid.

(b) If Q is anti-monotone in M, then

$$\mathbf{u} \le \mathbf{p} \to ((Q[\mathbf{x}_1] \dots [\mathbf{x}_k](F_1(\mathbf{x}_1), \dots, F_k(\mathbf{x}_k)))^* \leftrightarrow Q[\mathbf{x}_1] \dots [\mathbf{x}_k](F_1(\mathbf{x}_1), \dots, F_k(\mathbf{x}_k)))$$

is logically valid.

Proposition 1 allows us to simplify the formula $F^*(\mathbf{u})$ in (3) without affecting the models of (3). In formula (4), if Q is monotone in all argument positions, we can drop the second conjunctive term in view of Proposition 1 (a). If Q is anti-monotone in all argument positions, we can drop the first conjunctive term in view of Proposition 1 (b). For instance, recall that each of $Q_\wedge$, $Q_\vee$, $Q_\forall$, $Q_\exists$ is monotone in all its argument positions, and $Q_\neg$ is anti-monotone in $\{1\}$. If F is a standard first-order formula, then (4) can be equivalently rewritten as

- $(\neg F)^* = \neg F$;
- $(F \wedge G)^* = F^* \wedge G^*$; $(F \vee G)^* = F^* \vee G^*$;
- $(F \to G)^* = (F^* \to G^*) \wedge (F \to G)$;
- $(\forall x F)^* = \forall x F^*$; $(\exists x F)^* = \exists x F^*$.

This is almost the same as the definition of F^* given in [7], except for the case $(\neg F)^*$, which is a bit more concise.[3] The only propositional connective which is neither monotone nor anti-monotone in all argument positions is $Q_{\rightarrow}$, for which the simplification does not apply.

Example 3 continued. *For the GQ-sentence F considered earlier,* SM$[F]$ *is*

$$F \wedge \neg\exists u(u < p \wedge F^*(u)) \ , \tag{5}$$

where $F^*(u)$ is equivalent to the conjunction of F and

$$\neg Q_{\leq 2}[x]\ p(x) \rightarrow\ Q_{majority}[y]\ u(y). \tag{6}$$

I_1 considered earlier satisfies (5): it satisfies F, and, for any proper "subset" u of p, it satisfies the antecedent of (6) but not the consequent. Thus it is a stable model of F. On the other hand, we can check that I_2 does not satisfy (5), and is not a stable model.

4 Aggregates as GQ-Formulas

4.1 Formulas with Aggregates

The following definition of a formula with aggregates is from [5], which extends the one from [9] to allow nested aggregates. By a *number* we understand an element of some fixed set **Num**. For example, **Num** is $\mathbf{Z} \cup \{+\infty, -\infty\}$, where $\mathbf{Z}$ is the set of integers. A *multiset* is usually defined as a set of elements along with a function assigning a positive integer, called the *multiplicity*, to each of its elements. An *aggregate function* is a partial function from the class of multisets to **Num**. We assume the presence of some fixed background signature σ_{bg} that contains all numbers. Furthermore, we assume that the interpretation I_{bg} of the background signature is fixed, and interpretes each number as itself.

We consider a signature σ as a superset of σ_{bg}. An *expansion* I of I_{bg} to σ is an interpretation of σ such that

- the universe of I is the same as the universe of I_{bg}, and
- I agrees with I_{bg} on all the constants in σ_{bg}.

First-order formulas with aggregates are defined as an extension of standard first-order formulas by adding the following clause:

-
$$\text{OP}\langle \mathbf{x}_1.F_1, \dots, \mathbf{x}_n.F_n\rangle \succeq b \tag{7}$$

  is *a first-order formula with aggregates*, where
  - OP is a symbol for an *aggregate function* (not from σ);
  - $\mathbf{x}_1, \dots, \mathbf{x}_n$ are nonempty lists of distinct object variables;
  - $F_1, \dots, F_n$ are arbitrary *first-order formulas with aggregates* of signature σ;
  - $\succeq$ is a symbol for a comparison operator (may not necessarily be from σ);
  - b is a term of σ.

[3] $\neg F$ is understood as $F \rightarrow \bot$ in [7], but this difference does not affect stable models. When $\neg$ is a primitive propositional connective as above,

$$\mathbf{u} \leq \mathbf{p} \rightarrow ((F \rightarrow \bot)^*(\mathbf{u}) \leftrightarrow (\neg F)^*(\mathbf{u}))$$

is logically valid.

4.2 Aggregates as GQ-Formulas

Due to the space limit, we refer the reader to [5] for the stable model semantics of formulas with aggregates. We can explain their semantics by viewing it as a special case of the stable model semantics presented here. Following [5], for any set X of n-tuples ($n \geq 1$), let $msp(X)$ ("the multiset projection of X") be the multiset consisting of all ξ_1 such that $(\xi_1, \dots, \xi_n) \in X$ for at least one $(n-1)$-tuple $(\xi_2, \dots, \xi_n)$, with the multiplicity equal to the number of such $(n-1)$-tuples (and to $+\infty$ if there are infinitely many of them). For example, $msp(\{(a,a),(a,b),(b,a)\}) = \{\!\{a,a,b\}\!\}$.

We identify expression (7) with the GQ-formula

$$Q_{(\text{OP},\succeq)}[\mathbf{x}_1]\dots[\mathbf{x}_n][y](F_1(\mathbf{x}_1),\dots,F_n(\mathbf{x}_n), y=b) \ , \tag{8}$$

where, for any interpretation I, $Q^U_{(\text{OP},\succeq)}$ is a function that maps $\mathcal{P}(U^{|\mathbf{x}_1|}) \times \dots \times \mathcal{P}(U^{|\mathbf{x}_n|}) \times \mathcal{P}(U)$ to $\{\mathbf{t},\mathbf{f}\}$ such that $Q^U_{(\text{OP},\succeq)}(R_1,\dots,R_n,R_{n+1}) = \mathbf{t}$ iff

- $\text{OP}(\alpha)$ is defined, where α is the join of the multisets $msp(R_1),\dots,msp(R_n)$,
- $R_{n+1} = \{n\}$, where n is an element of **Num**, and
- $\text{OP}(\alpha) \succeq n$.

▶ **Example 4.** $\{discount(alice), discount(carol), numOfDiscounts(2)\}$ is an Herbrand stable model of the formula

$$\begin{aligned}&discount(alice) \wedge discount(carol)\\ &\quad \wedge \forall z(\text{COUNT}\langle x.discount(x)\rangle = z \rightarrow numOfDiscounts(z)).\end{aligned}$$

The following proposition states that this definition is equivalent to the definition from [5].

▶ Proposition 2. Let F be a first-order sentence with aggregates whose signature is σ, and let $\mathbf{p}$ be a list of predicate constants. For any expansion I of σ_{bg} to σ, I is a $\mathbf{p}$-stable model of F in the sense of [5] iff I is a $\mathbf{p}$-stable model of F in our sense.

5 Nonmonotonic dl-Programs as GQ-Formulas

5.1 Review of Nonmonotonic dl-Programs

Let C be a set of object constants, and let $P_{\mathcal{T}}$ and P_{Π} be disjoint sets of predicate constants. A nonmonotonic *dl-program* [3] is a pair $(\mathcal{T},\Pi)$, where $\mathcal{T}$ is a theory in description logic of signature $\langle C, P_{\mathcal{T}}\rangle$ and Π is a *generalized* normal logic program of signature $\langle C, P_{\Pi}\rangle$ such that $P_{\mathcal{T}} \cap P_{\Pi} = \emptyset$. We assume that Π contains no variables by applying grounding w.r.t. C. A generalized normal logic program is a set of nondisjunctive rules that can contain queries to $\mathcal{T}$ using "dl-atoms." A *dl-atom* is of the form

$$DL[S_1 op_1 p_1, \dots, S_k op_k p_k;\ Query](\mathbf{t}) \quad (k \geq 0), \tag{9}$$

where $S_i \in P_{\mathcal{T}}$, $p_i \in P_{\Pi}$, and $op_i \in \{\uplus, \cup\!\!\!\!-, \cap\!\!\!\!-\}$. $Query(\mathbf{t})$ is a *dl-query* as defined in [3]. A *dl-rule* is of the form

$$a \ \leftarrow \ b_1,\dots,b_m, not\ b_{m+1},\dots, not\ b_n \ , \tag{10}$$

where a is an atom and each b_i is either an atom or a dl-atom. We identify rule (10) with

$$a \ \leftarrow \ B, N \ , \tag{11}$$

where B is $b_1,\dots,b_m$ and N is $not\ b_{m+1},\dots,not\ b_n$. An Herbrand interpretation I *satisfies* a ground atom A *relative to* $\mathcal{T}$ if I satisfies A. An Herbrand interpretation I *satisfies* a ground dl-atom (9) *relative to* $\mathcal{T}$ if $\mathcal{T} \cup \bigcup_{i=1}^k A_i(I)$ entails $Query(\mathbf{t})$, where $A_i(I)$ is

- $\{S_i(\mathbf{e}) \mid p_i(\mathbf{e}) \in I\}$ if op_i is $\uplus$,
- $\{\neg S_i(\mathbf{e}) \mid p_i(\mathbf{e}) \in I\}$ if op_i is $\cup\!\!\!-$,
- $\{\neg S_i(\mathbf{e}) \mid p_i(\mathbf{e}) \notin I\}$ if op_i is $\Cap$.

A ground dl-atom A is *monotonic* relative to $\mathcal{T}$ if, for any two Herbrand interpretations I and I' such that $I \subseteq I'$ and $I \models_\mathcal{T} A$, we have that $I' \models_\mathcal{T} A$. Similarly, A is *anti-monotonic* relative to $\mathcal{T}$ if, for any two Herbrand interpretations I and I' such that $I' \subseteq I$ and $I \models_\mathcal{T} A$, we have that $I' \models_\mathcal{T} A$.

Given a dl-program $(\mathcal{T}, \Pi)$ and an Herbrand interpretation I of $\langle C, P_\Pi\rangle$, the *weak dl-transform* of Π relative to $\mathcal{T}$, denoted by $w\Pi^I_\mathcal{T}$, is the set of rules

$$a \leftarrow B' \tag{12}$$

where $a \leftarrow B, N$ is in Π, $I \models_\mathcal{T} B \wedge N$, and B' is obtained from B by removing all dl-atoms in it. Similarly, the *strong dl-transform* of Π relative to $\mathcal{T}$, denoted by $s\Pi^I_\mathcal{T}$, is the set of rules (12), where $a \leftarrow B, N$ is in Π, $I \models_\mathcal{T} B \wedge N$, and B' is obtained from B by removing all nonmonotonic dl-atoms in it. The only difference between these two transforms is whether monotonic dl-atoms remain in the positive body or not. Both transforms do not retain nonmonotonic dl-atoms.

An Herbrand interpretation I is a *weak (strong, respectively) answer set* of $(\mathcal{T}, \Pi)$ if I is minimal among the sets of atoms that satisfy $w\Pi^I_\mathcal{T}(s\Pi^I_\mathcal{T}, respectively)$.

5.2 Nonmonotonic dl-program as GQ-Formulas

We can view dl-programs as a special case of GQ-formulas. Consider a dl-program $(\mathcal{T}, \Pi)$ such that Π is ground. Under the strong answer set semantics we identify every dl-atom (9) in Π with

$$Q_{(9)}[\mathbf{x}_1]\dots[\mathbf{x}_k](p_1(\mathbf{x}_1), \dots, p_k(\mathbf{x}_k)) \tag{13}$$

if it is monotonic relative to $\mathcal{T}$, and

$$\neg\neg Q_{(9)}[\mathbf{x}_1]\dots[\mathbf{x}_k](p_1(\mathbf{x}_1), \dots, p_k(\mathbf{x}_k)) \tag{14}$$

otherwise. Since $\neg$ is an anti-monotone GQ, prepending $\neg\neg$ in front of the quantified formula in (14) means that, under the strong answer set semantics, every nonmonotonic dl-atom is understood in terms of an anti-monotone GQ.

Given an interpretation I, $Q^U_{(9)}$ is a function that maps $\mathcal{P}(U^{|\mathbf{x}_1|}) \times \dots \times \mathcal{P}(U^{|\mathbf{x}_k|})$ to $\{\mathbf{t}, \mathbf{f}\}$ such that, $Q^U_{(9)}(R_1, \dots, R_k) = \mathbf{t}$ iff $\mathcal{T} \cup \bigcup_{i=1}^k A_i(R_i)$ entails $Query(\mathbf{t})$, where $A_i(R_i)$ is
- $\{S_i(\boldsymbol{\xi}_i) \mid \boldsymbol{\xi}_i \in R_i\}$ if op_i is $\uplus$,
- $\{\neg S_i(\boldsymbol{\xi}_i) \mid \boldsymbol{\xi}_i \in R_i\}$ if op_i is $\cup\!\!\!-$,
- $\{\neg S_i(\boldsymbol{\xi}_i) \mid \boldsymbol{\xi}_i \in U^{|\mathbf{x}_i|} \setminus R_i\}$ if op_i is $\Cap$.

We say that I is a *strong answer set* of $(\mathcal{T}, \Pi)$ if I satisfies $\mathrm{SM}[\Pi; P_\Pi]$.

Similarly, a weak answer set of $(\mathcal{T}, \Pi)$ is defined by identifying every dl-atom (9) in Π with (14) regardless of A being monotonic or not. This means that, under the weak answer set semantics, every dl-atom is understood in terms of an anti-monotone GQ.

Example 1 continued. *The dl-atom*

$$\#dl[Man \uplus mm, Married \uplus mm, Woman \uplus mw, Married \uplus mw; \exists Spouse.\top](alice) \tag{15}$$

is identified with the generalized quantified formula

$$Q_{(15)}[x_1][x_2][x_3][x_4](mm(x_1), mm(x_2), mw(x_3), mw(x_4)) \tag{16}$$

where, for any interpretation I, $Q^U_{(15)}$ *is a function that maps* $\mathcal{P}(U)\times\mathcal{P}(U)\times\mathcal{P}(U)\times\mathcal{P}(U)$ *to* $\{\mathbf{t},\mathbf{f}\}$ *such that* $Q^U_{(15)}(R_1, R_2, R_3, R_4) = \mathbf{t}$ *iff* $\mathcal{T} \cup \{Man(c) \mid c \in R_1\} \cup \{Woman(c) \mid c \in R_3\} \cup \{Married(c) \mid c \in R_2 \cup R_4\}$ *entails* $\exists x Spouse(alice, x)$*.*

Consider an Herbrand interpretation $I = \{mw(alice)\}$*, which satisfies* (15)*. I also satisfies* (16) *since* $(x.mw(x))^I = \{alice\}$ *and* $\mathcal{T} \cup \{Woman(alice), Married(alice)\}$ *entails* $\exists x Spouse(alice, x)$*.*

The following proposition tells us that the definitions of a strong answer set and a weak answer set given here are reformulations of the original definitions from [3].

▶ Proposition 3. For any dl-program $(\mathcal{T}, \Pi)$, an Herbrand interpretation is a strong (weak, respectively) answer set of $(\mathcal{T}, \Pi)$ in the sense of [3] iff it is a strong (weak, respectively) answer set of $(\mathcal{T}, \Pi)$ in our sense.

5.3 Another Semantics of Nonmonotonic dl-programs

Shen [14] notes that both strong and weak answer set semantics suffer from circular justifications.

▶ **Example 5.** [14] Consider $(\mathcal{T}, \Pi)$, where $\mathcal{T} = \emptyset$ and Π is the program

$$p(a) \leftarrow \#dl[c \uplus p, b \cap\!\!\!\!- q;\ c \sqcap \neg b](a)\ , \tag{17}$$

in which the dl-atom is neither monotonic nor anti-monotonic. This dl-program has two strong (weak, respectively) answer sets: $\emptyset$ and $\{p(a)\}$. According to [14], the second answer set is circularly justified:

$$p(a) \Leftarrow \#dl[c \uplus p, b \cap\!\!\!\!- q; c \sqcap \neg b](a) \Leftarrow p(a) \wedge \neg q(a).$$

Indeed, $s\Pi^{\{p(a)\}}_{\mathcal{T}}$ ($w\Pi^{\{p(a)\}}_{\mathcal{T}}$, respectively) is $p(a) \leftarrow$, and $\{p(a)\}$ is its minimal model.

As we hinted in the previous section, this kind of circular justifications is related to the treatment that understands every nonmonotonic dl-atom in terms of an anti-monotone GQ, regardless of the nonmonotonic dl-atom's being anti-monotonic or not. In this case, in view of Proposition 1, predicates in a nonmonotonic dl-atom are exempt from the minimality checking. This is different from how we treat nonmonotone aggregates, where we simply identify them with nonmonotone GQs. This observation suggests the following alternative semantics of dl-programs, in which we understand only anti-monotonic dl-atoms in terms of anti-monotone GQs, unlike in the strong and the weak answer set semantics. We say that an Herbrand interpretation I is an *answer set* of $(\mathcal{T}, \Pi)$ if I satisfies $\mathrm{SM}[\Pi; P_\Pi]$, where we simply identify every dl-atom (9) in Π with (13).

This definition of an answer set has a reduct-based characterization as well. Just like we form a strong dl-transform, we first remove the negative body, but instead of removing all nonmonotonic dl-atoms in the positive body, we remove only anti-monotonic dl-atoms from the positive body. In other words, the *reduct* of Π relative to $\mathcal{T}$ and an Herbrand interpretation I of $\langle C, P_\Pi\rangle$, denoted by $\Pi^I_{\mathcal{T}}$, is the set of rules (12), where $a \leftarrow B, N$ is in Π, $I \models_{\mathcal{T}} B \wedge N$, and B' is obtained from B by removing all anti-monotonic dl-atoms in it. The following proposition shows that this modified definition of a reduct can capture the new answer set semantics of dl-programs.

▶ Proposition 4. For any dl-program $(\mathcal{T}, \Pi)$ and any Herbrand interpretation I of $\langle C, P_\Pi \rangle$, I is an answer set of $(\mathcal{T}, \Pi)$ according to the new definition iff I is minimal among the sets of atoms that satisfy $\Pi^I_{\mathcal{T}}$.

The new semantics does not have the circular justification problem described in Example 5.

Example 5 continued. *$\{p(a)\}$ is not an answer set of $(\mathcal{T}, \Pi)$ according to the new definition. The reduct $\Pi^{\{p(a)\}}_{\mathcal{T}}$ is (17) itself retaining the dl-atom unlike under the strong and the weak answer set semantics. We check that $\emptyset$, a proper subset of $\{p(a)\}$, satisfies it, which means that $\{p(a)\}$ is not an answer set.*

6 Related Work

We refer the reader to [2] for the semantics of HEX programs. It is not difficult to see that an external atom in a HEX program can be represented in terms of a generalized quantifier. Eiter *et al.* show how dl-atoms can be simulated by external atoms $\#dl[](x)$. The treatment is similar to ours in terms of generalized quantifiers. For another example, rule

$$reached(x) \leftarrow \#reach[edge, a](x)$$

defines all the vertices that are reachable from the vertex a in the graph with $edge$. The external atom $\#reach[edge, a](x)$ can be represented by a generalized quantified formula

$$Q_{reach}[x_1 x_2][x_3][x_4](edge(x_1, x_2), x_3 = a, x_4 = x),$$

where Q_{reach} is as defined in Example 2.

In fact, incorporation of generalized quantifiers in logic programming was considered earlier in [1], but the treatment there was not satisfactory because they understood generalized quantifiers simply as anti-monotone GQs in our sense. Without going into detail, this amounts to modifying our definition of F^* as

$$(Q[\mathbf{x}_1] \dots [\mathbf{x}_k](F_1(\mathbf{x}_1), \dots, F_k(\mathbf{x}_k)))^* = Q[\mathbf{x}_1] \dots [\mathbf{x}_k](F_1(\mathbf{x}_1), \dots, F_k(\mathbf{x}_k)) \ .$$

This approach does not allow recursion through generalized quantified formulas, and often yields an unintuitive result. According to [1], program $p(a) \leftarrow \forall x\, p(x)$ has two answer sets, $\emptyset$ and $\{p(a)\}$. The latter is "unfounded." This is not the case with the semantics that we introduced in this note. According to our semantics, which properly extends the semantics from [7], $\{p(a)\}$ is not an answer set.

7 Conclusion

We presented the stable model semantics for formulas containing generalized quantifiers, and showed that some recent extensions of the stable model semantics with "complex atoms" can be viewed as special cases of this formalism. We expect that the generality of the formalism will be useful in providing a principled way to study and compare the different extensions of the stable model semantics. As we observed, distinguishing among monotone, anti-monotone, and neither monotone nor anti-monotone GQs is essential in defining the semantics of such extensions, whereas the last group of GQs was not considered in the traditional stable model semantics.

Acknowledgements. We are grateful to Michael Bartholomew, Vladimir Lifschitz, and the anonymous referees for their useful comments. This work was partially supported by the National Science Foundation under Grant IIS-0916116.

References

1. Thomas Eiter, Georg Gottlob, and Helmut Veith. Modular logic programming and generalized quantifiers. In *Proceedings of International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, pages 290–309, 1997.
2. Thomas Eiter, Giovambattista Ianni, Roman Schindlauer, and Hans Tompits. A uniform integration of higher-order reasoning and external evaluations in answer-set programming. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*, pages 90–96, 2005.
3. Thomas Eiter, Giovambattista Ianni, Thomas Lukasiewicz, Roman Schindlauer, and Hans Tompits. Combining answer set programming with description logics for the semantic web. *Artificial Intelligence*, 172(12-13):1495–1539, 2008.
4. Wolfgang Faber, Gerald Pfeifer, and Nicola Leone. Semantics and complexity of recursive aggregates in answer set programming. *Artificial Intelligence*, 175(1):278–298, 2011.
5. Paolo Ferraris and Vladimir Lifschitz. On the stable model semantics of firsr-order formulas with aggregates. In *Proceedings of International Workshop on Nonmonotonic Reasoning (NMR)*, 2010.
6. Paolo Ferraris, Joohyung Lee, and Vladimir Lifschitz. A new perspective on stable models. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*, pages 372–379, 2007.
7. Paolo Ferraris, Joohyung Lee, and Vladimir Lifschitz. Stable models and circumscription. *Artificial Intelligence*, 175:236–263, 2011.
8. Paolo Ferraris. Answer sets for propositional theories. In *Proceedings of International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, pages 119–131, 2005.
9. Joohyung Lee and Yunsong Meng. On reductive semantics of aggregates in answer set programming. In *Proceedings of International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, pages 182–195, 2009.
10. Joohyung Lee, Vladimir Lifschitz, and Ravi Palla. A reductive semantics for counting and choice in answer set programming. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, pages 472–479, 2008.
11. Fangzhen Lin and Yi Zhou. From answer set logic programming to circumscription via logic of GK. *Artificial Intelligence*, 175:264–277, 2011.
12. Per Lindström. First-order predicate logic with generalized quantifiers. *Theoria*, 32:186–195, 1966.
13. A. Mostowski. On a Generalization of Quantifiers. *Fundamenta Mathematicae*, 44:12–35, 1957.
14. Yi-Dong Shen. Well-supported semantics for description logic programs. In *Proceedings of International Joint Conference on Artificial Intelligence*, pages 1081–1086, 2011.
15. Tran Cao Son and Enrico Pontelli. A constructive semantic characterization of aggregates in answer set programming. *Theory and Practice of Logic Programming*, 7(3):355–375, 2007.
16. Dag Westerståhl. Generalized quantifiers. In *The Stanford Encyclopedia of Philosophy (Winter 2008 Edition)*. 2008. `http://plato.stanford.edu/archives/win2008/entries/generalized-quantifiers/`.

Using Answer Set Programming in the Development of Verified Software

Florian Schanda[1] **and Martin Brain**[2]

1 **Altran Praxis Limited**
20 Manvers Street, Bath, BA1 1PX, UK
`florian.schanda@altran-praxis.com`

2 **Department of Computer Science***
University of Oxford, Oxford, OX1 3QD, UK
`martin.brain@cs.ox.ac.uk`

Abstract

Software forms a key component of many modern safety and security critical systems. One approach to achieving the required levels of assurance is to prove that the software is free from bugs and meets its specification. If a proof cannot be constructed it is important to identify the root cause as it may be a flaw in the specification or a bug. Novice users often find this process frustrating and discouraging, and it can be time-consuming for experienced users. The paper describes a commercial application based on Answer Set Programming called Riposte. It generates simple counter-examples for false and unprovable verification conditions (VCs). These help users to understand why problematic VC are false and makes the development of verified software easier and faster.

1998 ACM Subject Classification F.3.1 Specifying and Verifying and Reasoning about Programs

Keywords and phrases Answer Set Programming, verification, SPARK, Ada, contract based verification, safety critical

Digital Object Identifier 10.4230/LIPIcs.ICLP.2012.72

1 Introduction

A critical system is one whose failure would cause serious injury, one or more fatalities, major environmental damage or significant damage to other assets. Software is a component of many critical systems and is playing an ever increasing role in their monitoring and control. For example in modern aircraft, both civil and military, there are complex flight control systems which must never 'go wrong'. Errors in algorithms may cause wrong behaviour; software crashes may result in catastrophic failures. Part of the argument for the safety of a system is verification – showing that the system meets its specification. For software the specification may include functional properties (things the system must do) and erroneous behaviour (things that the system must not do). Testing may be sufficient to show functional properties (i.e. the system can track flights) but is not able to guarantee the absence of errors – for example testing alone cannot show that a system will never crash. Critical systems require a higher level of assurance, formal verification systems, such as SPARK[1] can provide this kind of certainty.

* Work conducted while at University of Bath and Altran Praxis.
[1] The SPARK programming language is not sponsored by or affiliated with SPARC International Inc. and is not based on the SPARC™ architecture.

© Florian Schanda and Martin Brain;
licensed under Creative Commons License ND
Technical Communications of the 28th International Conference on Logic Programming (ICLP'12).
Editors: A. Dovier and V. Santos Costa; pp. 72–85

Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

One of the major barriers to increasing the commercial adoption of formal verification is the perception that it is too expensive. Although formal development practices and verification have been shown to reduce total project costs [13, 21] as well as to increase levels of assurance, many companies focus only on the initial development time. "Programmers find verification hard, so it takes them longer and thus costs us more" is a common objection. This misses the wider context, but addressing this misapprehension is crucial to improving adoption. One route to doing so is to improve support tools for developing verified software. Given that developer time is at least 100 to 1000 times more expensive than CPU time, significant computation resources can be justified if they save developers' time.

The current proof tools for SPARK focus on the primary goal of quickly and easily discharging verification conditions (VCs). The proof of all verification conditions shows a number of properties about the system: for example that certain kinds of error cannot occur (for example buffer overruns), or that some security property holds (for example: only one door of an airlock must be open at any time).

There is only limited support for distinguishing between VCs that are unprovable due to incompleteness in the proof tools and those that are unprovable because they are false. When verifying finished and correct software, this is of little importance. However during development a significant minority of VCs may be unprovable. Distinguishing bugs (in specification or implementation) from areas of incompleteness is vital as the resolutions for each are very different and incorrectly classifying a verification failure can waste time and potentially introduce unsoundness (depending on the processes around the usage of the tool). Riposte is a tool based on Answer Set Programming (ASP) that supports developers in classifying and resolving verification failures by generating concrete counter-examples to false verification conditions. This paper:

- Overviews the architecture of Riposte and its usability features which are intended to produce more insightful counter-examples (Section 3).
- Describes the methodology used and experience gained in developing a commercial tool using ASP and the more unusual features of the problem encodings used by Riposte (Section 4).
- Gives statistics for the typical problem instances generated by Riposte and compares Riposte's performance to that of SMT solvers for analysing erroneous programs (Section 5).

2 SPARK

SPARK is a language and supporting toolset[2]; the primary design goal is the provision of an unambiguous language semantics and a sound verification system based on Hoare logic and theorem proving. The executable part of the language is a subset of Ada (83, 95 and 2005) and data flow and correctness contracts are given in Ada comments. Figure 2 shows a very simple SPARK program which will serve as an example throughout this paper, and Figure 1 illustrates the current architecture of the tool chain, with the phases of computation, flow of information and outputs. There are three key phases, referred to as examine, prove and summarise.

The 'front end' of the SPARK toolset is the Examiner. It checks the program for compliance with the SPARK subset, performs data flow analysis and generates VCs for each path between cut points (subprogram start and end, loops and assertions). VCs check contracts specified

[2] Available under the GPL from `http://libre.adacore.com/`

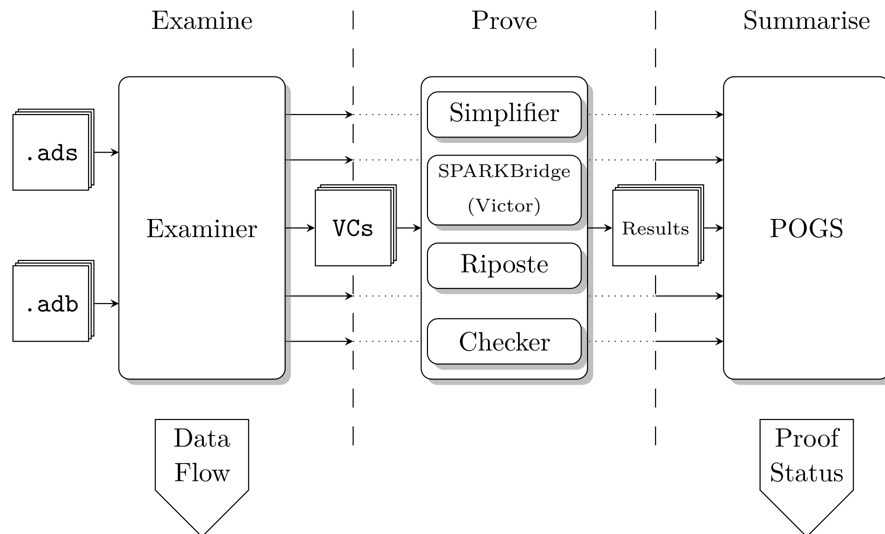


Figure 1 The SPARK tool chain.

by the user and freedom from run time exceptions such as integer overflow, array bounds checks and division by zero. Figure 3 shows a VC generated from the previous example program. VCs are expressed in functional description language (FDL), a simple intermediate language, and a variety of proof tools are available to discharge them. These include the Simplifier, a rewrite based automatic theorem prover; Victor, an SMT translator and prover driver [22] and the Checker, an interactive theorem prover. An Isabelle plug-in to read SPARK VCs [3] is also available. Finally the POGS tool is used to summarise the state of the proof of the entire system, giving statistics such as how many VCs there are in the system and how many of them are discharged.

SPARK is a mature system with the first version released in March 1987, and the SPARK tools have been used on a variety of industrial projects including applications such as flight control, rail signalling, and high-grade cryptographic systems.

SPARK places particular emphasis on modularity; this means it is common to verify software as it is being written, well before it is completed. Thus subprograms first analysed by the Examiner often contain errors and give undischarged VCs. Proof tools in earlier versions of SPARK did not distinguish between those VCs that are undischarged due to incompleteness and those undischarged because they are false. The resolution for these two kinds of failure are very different and misclassification can waste time and potentially introduce unsoundness. So there is a need for a counter-example generation tool to support users in locating the causes of verification failures.

3 Riposte architecture

Riposte consists of a front-end (implementing the parsing of FDL, interval reasoning, simple rewrite and the user interface) and a back-end (that is used to perform the actual search for counter-examples). The front-end generates an *AnsProlog* program for each conclusion analysed. The back-end is a further set of rules included by each program which encode the semantics of FDL. This program is then passed to an answer-set solver and any model returned by the answer-set solver represents a counter-example, which is then interpreted by

```
type Unsigned_Byte is new Integer range 0 .. 255;

function Add_UB (A, B: Unsigned_Byte)
                 return Unsigned_Byte
--# return Value => (Value > A);
is
begin
   return A + B;
end Add_UB;
```

Figure 2 Example SPARK subprogram with several bugs. The line starting with `-#` is a SPARK contract specifying a postcondition for the function.

```
function_add_ub_2.
H1:    true .
H2:    a >= unsigned_byte__first .
H3:    a <= unsigned_byte__last .
H4:    b >= unsigned_byte__first .
H5:    b <= unsigned_byte__last .
H6:    a + b >= unsigned_byte__base__first .
H7:    a + b <= unsigned_byte__base__last .
        ->
C1:    a + b > a .
C2:    a + b >= unsigned_byte__first .
C3:    a + b <= unsigned_byte__last .
```

Figure 3 An interesting VC for the code from Figure 2. H2 - H5 are the hypotheses that give the bounds for a and b. H6 and H7 state that $a+b$ will not overflow the base type for Unsigned_Byte, in our case this is a 32-bit signed integer. C1 is the proof obligation required to show the postcondition of the function (as specified by the user); C2 and C3 are required to show absence of run-time errors as the addition may overflow the range allowed for Unsigned_Byte (this proof obligation is automatically generated by the Examiner).

the front-end and expressed in a user-friendly way. Figure 4 shows the overall architecture of Riposte, where `gringo` is the grounder and `clasp` the answer-set solver of the Potassco [18] tool-chain.

Riposte is designed to be *sound* but *not complete*, thus an absence of a model guarantees that a given conclusion is necessarily true. However, there may be spurious counter-examples generated by Riposte (in particular in the presence of complex quantified expressions). To mitigate this Riposte also attempts to check each counter-example to determine if it is a valid counter-example.

The back-end of Riposte contains approximately 4,700 lines of *AnsProlog*describing 1,000 rules; the front-end is around 12,000 lines of Python. To our knowledge it is one of the largest commercial deployment of an answer set program to date.

3.1 Rewrite

After parsing, Riposte performs a number of rewrites and simplifications. These include putting the hypotheses and conclusions in prenex normal form, Skolemisation to remove existential quantifiers and elimination of redundant quantifiers. Normalising expressions makes the subsequent processing simpler and can make the problems easier.

3.2 Interval reasoning

The integers in FDL are mathematical and thus ‘infinite precision’. However in the VCs generated from SPARK programs every program variable is bounded. These bounds are

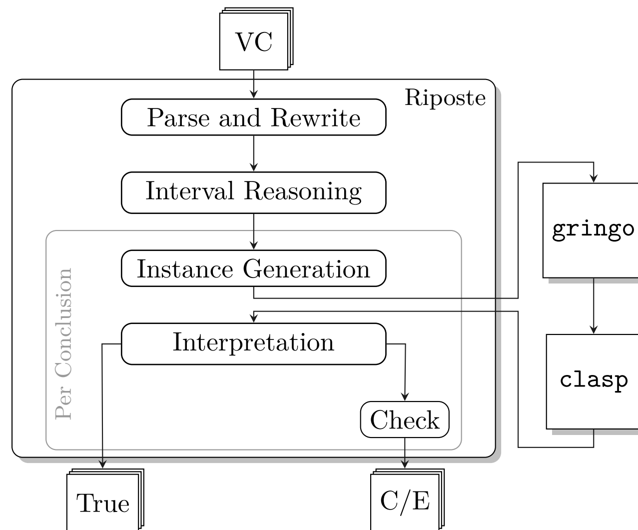


Figure 4 Architecture of Riposte.

found and interval reasoning techniques similar to bounds propagation algorithms used in CSP (Constraint Satisfaction Problem) solvers [16] are used to soundly refine the bounds. Sometimes this reasoning is sufficient to show a conclusion must always be true, in which case it can be discharged.

For example in Figure 3, Riposte can determine that the range of both a and b is $[0; 255]$, and the range for $a + b$ is $[0; 510]$. Riposte can now immediately rewrite conclusion 2 to 'true' as unsigned_byte___first is 0.

3.3 Program generation

Once bounds are established for all program variables and the formulae simplified, Riposte handles each conclusion individually. Although this requires more calls to the solver, the program variables and hypotheses used in each search can be reduced to only those that are necessary for a given conclusion. This not only makes the search faster, it significantly simplifies the counter-examples generated, as assignments for irrelevant program variables are not generated.

We now present a few interesting lines from the program generated for conclusion 3. Note that more of the encoding is described in Section 4, but most names should be fairly obvious.

A few important background literals required by the rest of the encoding are defined, the only one relevant for our example is wordLength; from interval arithmetic we know that the largest values (the base types for a and b) can fit into a signed 32-bit integer in our example.

```
%%% Background
wordLength(32).
literalIntegerLow(0).
literalIntegerHigh(1).
optimisationLength(8).
```

For each program variable present in the VC, Riposte generates an 'input variable', this defines the search space for the program.

```
variable(a,bitInteger,input).
variable(b,bitInteger,input).
```

Each VC may also make use of some numeric constants (0 and 255 in our example); the bit-patterns for those are also defined in the program, a snippet for the first 3 bits of constant 255 is shown below:

```
variable(bi_const_255,bitInteger,constant).
bitValue(bi_const_255,0,1).
bitValue(bi_const_255,1,1).
bitValue(bi_const_255,2,1).
bitValue(bi_const_255,3,1).
```

Riposte then encodes each hypothesis and the currently analysed conclusion. The encoding of a hypotheses (H2) is as follows:

```
%%% H2: a >= 0
variable(bi_leq_s(bi_const_0,a),boolean,expression).
computation(bi_leq_s(bi_const_0,a),
            bi_leq_s,
            bi_const_0,
            a).

hypothesis(bi_leq_s(bi_const_0,a)).
```

The `variable` literal declares the Boolean expression `bi_leq_s(bi_const_0, a)` with the `computation` literal generates the rules that compute its value and the `hypothesis` literal denotes that it is a hypothesis and thus must be true in all models.

Finally, the encoding of the conclusion analysed, C3, is shown below. Note the naming of the variables for expressions, such as `bi_plus_s(a, b)`; this avoids generating two calculations for the same expression twice and it also allows easy identification of what a variable represents from the name only.

```
%%% C3: a + b <= 255
variable(bi_plus_s(a,b),bitInteger,expression).
computation(bi_plus_s(a,b),
            bi_plus_s,
            a,
            b).

variable(bi_leq_s(bi_plus_s(a,b),bi_const_255),boolean,expression).
computation(bi_leq_s(bi_plus_s(a,b),bi_const_255),
            bi_leq_s,
            bi_plus_s(a,b),
            bi_const_255).

conclusion(bi_leq_s(bi_plus_s(a,b),bi_const_255)).
```

3.4 Interpretation

After the program has been generated and passed to `gringo` and `clasp`, a model may be returned. Each model contains a valuation for each input variable (`bitValue` for each bit of a bitIntegers, `boolenValue` for booleans, etc.).

```
*** Found a counter-example to function_add_ub_2, conclusion C3:
    (For path(s) from start to finish:)
H2: a >= 0
H3: a <= unsigned_byte__last
H4: b >= 0
H5: b <= unsigned_byte__last
   ->
C3: a + b <= unsigned_byte__last

This conclusion is false if:
   a = unsigned_byte__last
   b = 1
```

A number of basic, but effective, usability features have been implemented in order to assist the user with understanding the counter-example. In the output reproduced above for

conclusion 3 of our example, large numbers are translated back to the original constants or an easier expression[3]; in this case unsigned_byte___last is really 255, but it is shown using the original name used in the VC. Furthermore, in order to reduce visual clutter only the hypotheses which are relevant to our conclusion are printed.

Riposte also checks that the counter-example given does actually make all hypotheses true and the conclusion false. This currently functions as an integrity check but will be used to refine the modelling if spurious counter-examples are generated.

Caching of previous results using `Memcached` is also performed to allow incremental and distributed computation [8].

4 Methodology and Modelling

The experience of developing a commercial scale application using ASP has yielded some insights into the development process and some useful encoding techniques.

4.1 Methodology

Riposte was developed using the methodology described in [6]. The map from informal concepts (such as “the B’th bit of variable N has value V”) to literals was the first thing developed. Using this a number of simple programs were encoded manually and an interpretation script was developed. This allowed models to be understood in terms of the informal concepts rather than as a set of literals, and was invaluable in locating faults. The search space (each possible assignment to the variables in the FDL) was modelled and manually checked. Then the program was developed incrementally, one instruction at a time, with the behaviour of each section checked before proceeding. This greatly simplified debugging as the faults were almost always in the most recently added rules and their effects, in terms of the concepts they were supposed to represent, were easily visible. Three additional techniques were used to locate and prevent faults: random testing of individual instructions, system level regression testing and test driven development, and explicit modelling of assumptions about the model.

To test the individual instructions, a simple application was written that picked input values (covering all of the combinations of common ‘edge’ and extreme values and some random values), emulated the instruction and then produced a program that checked that the *AnsProlog* model gave the correct result. This proved useful while modelling the instruction as it allowed the partially completed model to be checked. In at least one case a discrepancy between the declarative *AnsProlog* and the procedural emulation in the test system was found to be a fault in the emulation!

At a higher level, system level test cases (VCs with annotations of which conclusions were supposed to have counter examples) were extensively used. Often suites of tests for a feature were written before they were implemented; in a fashion similar to test driven development. Once features were implemented, these test suites were used as whole system regression tests. This approach proved very effective and when the system was used on commercial code bases, very few faults were found.

The third technique for fault minimisation is more specific to *AnsProlog*. When developing a model there are normally a number of undocumented assumptions about the programs

[3] For example instead of printing 4503599627370495, Riposte will print $2 * *52 - 1$, which we contend is much more helpful.

and encodings. For example that each FDL program variable modelled is given only one type or that any bit is either 1 or 0. In the case of the program, these are normally regarded to be obvious from the informal meaning of the predicates and it is left to the programmer to generate valid programs. Implicit properties of the encoding can be given as auxiliary constraints if that helps inference. In Riposte assumptions about programs and the encoding are explicitly stated using rules that derive a "model error" literal. For example, separate literals are used to state when a bit is 1 or when it is 0 and the following rules are used to express the link between them.

```
%% Each bit of bitIntegers must be 1 (x)or 0
modelError(bit_is_both_1_and_0(N,B)) :-
   bitValue(N,B,1), bitValue(N,B,0), variable(N,bitInteger,R).
modelError(bit_is_neither_1_nor_0(N,B)) :-
   not bitValue(N,B,1), not bitValue(N,B,0), variable(N,bitInteger,R), bit(B).
```

There are two uses for these rules. During development it is possible to search for answer sets with model errors. This gives meaningful explanations of which implicit properties of the model have been broken, rather than yielding models. When Riposte is run in production mode, a constraint is added stating there are no modelling errors. Thus all of the rules describing modelling errors effectively become constraints, allowing equivalence preprocessing [19] to collapse the separate literals to one. This is an evolution of the techniques for error diagnosis used in *Anton* [5].

4.2 Encoding

A number of encoding techniques were developed to improve the performance and capacity of Riposte.

Variables are a central part of the model used in Riposte. They model FDL variables, constants and the values of expressions. For example, if the expression $a + b > 0$ appears in a hypothesis or conclusion, there will be five variables modelled; two FDL, or input integers, a and b, one integer constant, 0, and two expression variables, an integer for $a+b$ and a Boolean for $a + b > 0$. Choice rules are used so that FDL variables are assigned non-deterministic values. Constants are assigned direct values and the values of expression variables are given by the rules modelling their instruction. One key innovation was to name expression variables by the expression they compute. For example the variable corresponding to the expression $a + b$ would be named `bi_plus_s(a,b)`. This meant that all of the hypotheses and conclusions that referred to $a + b$ would automatically use the same variable and thus the same literals. Not only did this reduce the size of the programs generated, it also helped eliminate symmetries introduced by having multiple variables record the value of the same expression, and thus improved propagation.

One of the key challenges in modelling was how to deal with quantified expressions. As soon as the target application contains arrays, quantified hypotheses are unavoidable as even the simplest statement of type safety about arrays requires quantifiers. To illustrate Riposte's handling of quantifiers, consider the following (contrived) example:

```
function Contrived (A : Unsigned_Byte)
                  return Boolean
--# pre for all I in Unsigned_Byte range 50 .. 100 => (A /= I);
--# return A > 60 -> A > 150;
is
begin
   return True;
end Contrived;
```

The hypotheses which represents the precondition (effectively $a \notin [50, 100]$) is expressed in FDL translated as follows (note that the identifier I has been renamed by Riposte).

```
%%% H1: for_all(riposte____qid_1: unsigned_byte,
%%%            riposte____qid_1 >= 50 and riposte____qid_1 <= 100 ->
%%%               not a = riposte____qid_1)
```

Riposte handles quantifiers using the sound but not complete technique of instantiation. Every quantified hypothesis is replaced by a number of copies representing a subset of the possible bindings for the quantifier. Omitting particular bindings can fail to remove models (giving incompleteness) but cannot add models to a problem with no models (thus giving soundness). Due to space constraints we show this only for part of the statement, *not* $a = riposte____qid_1$. Note that RIPOSTE____QID_1 is variable whose instantiation is determined by the literal `hypothesisInstantiation`. `qi_h1` is an arbitrary constant identifying the relevant expression.

```
variable(bi_equal_l(a,RIPOSTE____QID_1),boolean,expression)
   :- hypothesisInstantiation(qi_h1,RIPOSTE____QID_1).
computation(bi_equal_l(a,RIPOSTE____QID_1),
            bi_equal_l,
            a,
            RIPOSTE____QID_1)
   :- hypothesisInstantiation(qi_h1,RIPOSTE____QID_1).

variable(b_not_l(bi_equal_l(a,RIPOSTE____QID_1)),boolean,expression)
   :- hypothesisInstantiation(qi_h1,RIPOSTE____QID_1).
computation(b_not_l(bi_equal_l(a,RIPOSTE____QID_1)),
            b_not_l,
            bi_equal_l(a,RIPOSTE____QID_1))
   :- hypothesisInstantiation(qi_h1,RIPOSTE____QID_1).
```

And finally the rule which encodes our *simple* but *surprisingly effective* instantiation heuristic. We essentially instantiate the quantified expression for all variables which are not expressions (i.e. for constants and input variables only).

```
hypothesisInstantiation(qi_h1,RIPOSTE____QID_1) :-
   variable(RIPOSTE____QID_1,bitInteger,R1), R1 != expression.
```

For our example this means that the quantified hypothesis is instantiated for $\{a, 0, 1, 50, 60, 100, 150, 255\}$ and Riposte gives $i = 101$ as a counter-example.

The last two encoding techniques improved the performance and completeness of Riposte, the next technique is focused on improving usability. Considering the program given in Figure 3, $a = 91$ and $b = 214$ is a counter example to conclusion 3. While this is an entirely correct counter-example it is perhaps not the most informative. To produce more helpful counter examples, Riposte makes use of the optimisation features of the answer set solver. This is used to produce counter examples in which the FDL or input variables are as close to zero as possible. By using an arbitrary order across the input variables and individual optimise statements for each variable, counter examples will often end up minimising some program variables and maximising other. For example in the case above, $a = 255$ and $b = 1$ is given. One of the advantages of using an answer set solver is being able to perform this optimisation.

5 Evaluation

This section gives two evaluations of Riposte. The first focuses on false VCs and compares Riposte with Victor and a variety of different SMT solvers. This evaluates Riposte in its intended usage scenario – finding counter examples to individual false VCs. The second experiment uses a large set of true VCs for a number of commercial applications and shows the distribution of problem size and run-time across real applications.

5.1 Comparison

As Riposte is a developer support tool, a key requirement is that it produces responses quickly and consistently across a range of real world programs. To test this a set of programs with undischarged VCs was created from publicly available SPARK applications: libsparkcrypto [26], Tokeneer [1] and SPARKSkein [9]. A number of subprograms whose proofs require non-formalised background information (for example, the number of certificates that can fit on the removable storage) were taken from Tokeneer. Subprograms taken from libsparkcrypto and SPARKSkein were modified to contain common bugs such as off by one errors, missing preconditions, indexing errors and insufficient loop invariants. The Examiner was used to generate VCs for these subprograms and the Simplifier used to remove simple true VCs, leaving a test set of 45 VCs. All experiments were run on an Intel i7 860 (2.8 GHz, 4 cores) desktop computer running Debian GNU/Linux, using a 20 minutes time limit per VC.

Figure 5 gives a graph of the cumulative time taken for Riposte to produce answers for the benchmark VCs. Results are also given for Victor, the SMT based prover for SPARK, using a variety of back end SMT solvers: Alt-Ergo [12], CVC3 [14] and Z3 [15]. These are included to give an idea of what constituted reasonable amounts of time and completeness, rather than for direct comparison.

Although the SMT solvers outperform Riposte for the easy VCs, the more complex VCs containing bugs are resolved much more quickly by Riposte; resulting in the overall fastest time to process all VCs. Riposte is the only tool that renders a verdict on all benchmark VCs within the time limit. The division between grounder and solver causes slightly higher overheads for Riposte, giving the lower results on the left hand side of the graph. However the total time taken by Riposte on all resolved VCs is significantly lower (Riposte 1600s, CVC3 9000s, Alt-Ergo 11100s, Z3 20800s; to the nearest 100s) even though coverage is higher.

Riposte's performance on these benchmarks is fairly typical. During development it has been used on over 22,500 VCs (including four industrial applications, one unknown to the tools authors) resolving 95% or more. When counter examples are found, they are typically found rapidly and within the time developers are willing to wait.

5.2 Program statistics

We have also used Riposte to analyse the three code-bases mentioned above in their original form to generate some statistics on the size and run-times of programs generated. Figure 6 shows the distribution of sizes the ground programs in terms of atoms and rules. It can be seen that most of the programs are small (≤ 25000 atoms/rules), but a few are very large ($\geq 1,1$ million atoms and $\geq 1,2$ million rules).

Figure 7 plots the time taken to ground each program against the time taken to solve. It can be seen that most programs take longer to ground than to actually solve and even then the combined time is usually significantly less than around 10 seconds.

6 Related work

A key precedent for using ASP to reason about programs is the TOAST superoptimiser [7]. Its model of instructions was somewhat simpler as it was modelling hardware and thus only had the register 'type' to consider. In comparison, Riposte's models include a much richer type system (as it is modelling a typed language) and supports both quantifiers and axioms for reasoning about more complex types.

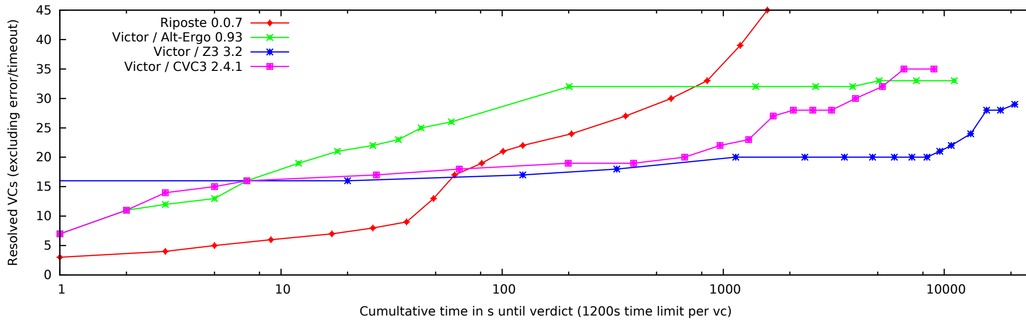
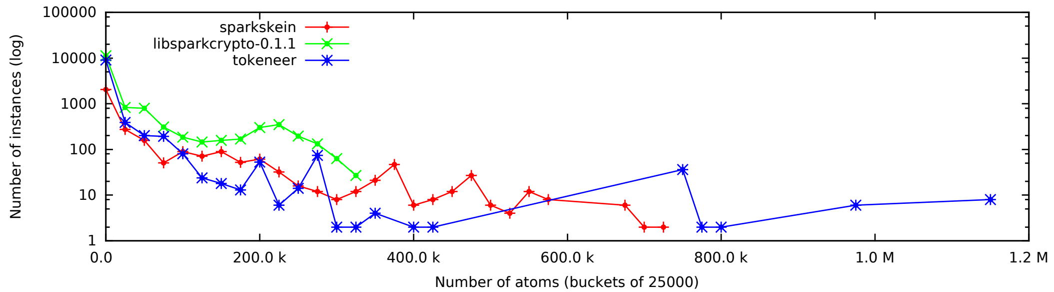


Figure 5 Cumulative time for returning a verdict on VCs for erroneous code.



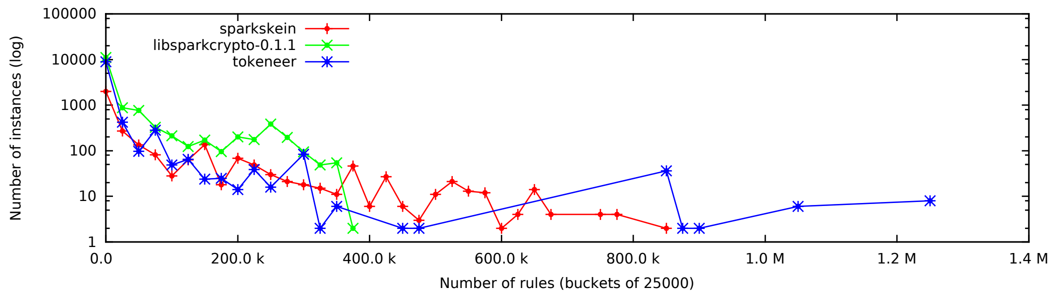


Figure 6 Program sizes in terms of atoms (above) and rules (below).

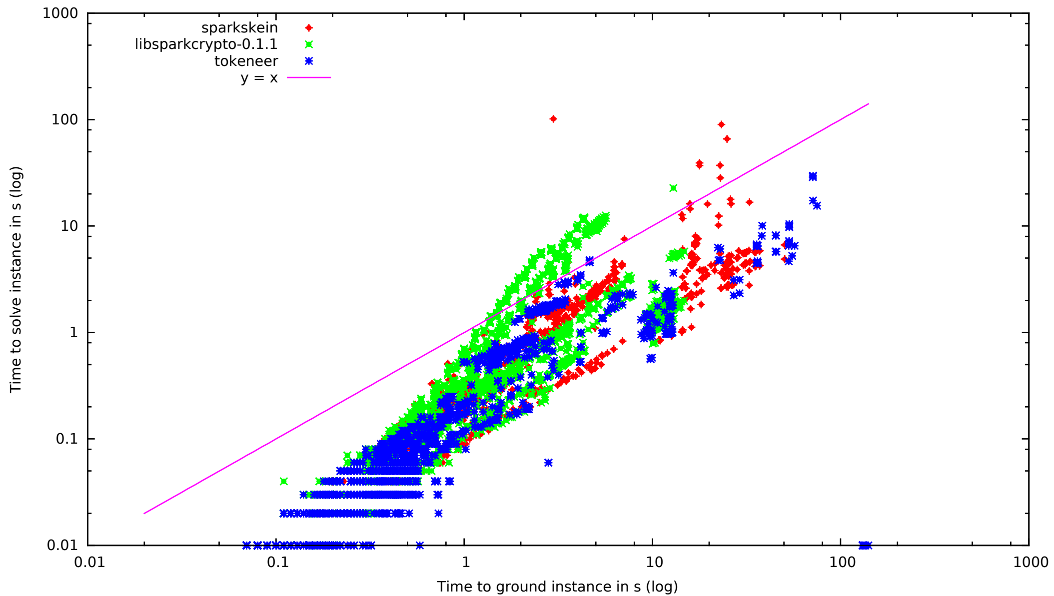


Figure 7 Time taken to ground each program v.s. time taken to solve.

The closest system to Riposte, both in terms of architecture and approach, is Nitpick [4], Isabelle's counter-example generator. It uses KodKod to generate counter-examples to HOL theorems. Although the theoretical foundations of answer set semantics and KodKod's FORL are very different, there are many parallels between the two systems, making KodKod effectively an equivalent approach. The one key difference is that Nitpick has to deal with infinite objects, making the encoding significantly more complex.

Systems that use SAT, SMT or other model generation solvers to discharge VCs (for example [11]) can potentially generate counter-examples directly from failed proof attempts. However there are a number of practical problems involving the size and complexity of the counter-examples generated [25].

A key problem is that compact VC generation algorithms [2] make it difficult to identify the root cause of a counter-example (as well as potentially significantly increasing the cost of verification [24]). One option is to 'tag' the VC with explanations. Tags can be additional propositions [23] or meta-information annotations [17]. As SPARK uses a more verbose but significantly simpler VC generation system (see Section 2), these are not needed in Riposte, since the failing condition (and why it is generated) is already available to the user.

Another area of research concerns the development of user interfaces to view and explore counter-examples once they have been generated. The VCC Model Viewer [11] and its successor, the Boogie Verification Debugger [20], show the power of integrating counter-example display into an IDE. An innovative approach to doing this is generating a program that triggers a bug corresponding to the counter-example [25] and then using a conventional debugger interface.

Finally, counter-examples play a key role in checking and refining abstraction in model checking systems, although this tends to be automatic (for example systems based on CEGAR [10]) rather than aimed at supporting end-users.

7 Conclusion and Future Work

This paper presents Riposte, a successful commercial application of answer set programming. Its performance is state of the art, as shown in Section 5. Furthermore it validates previous work on development methodologies [6] by showing it is possible to develop large application using them.

The next step for Riposte is integration into the next commercially supported release of the SPARK tools. This will definitely yield challenging examples generated from VCs for real world systems. It is hoped that these will be useful in improving the performance and capacity of answer set programming tools. One area of particular interest is improvement in the performance of grounders. As shown in Figure 7, grounding time is often the dominant factor in Riposte's performance. This is unusual as when the grounding is a bottleneck it is normally a space issue rather than run-time.

Another challenging area is moving counter examples beyond assignments of values to program variables. In some cases it is possible to produce expressions that describe a set of counter examples and are more informative than a single counter example. It may be possible to use the skeptical query mode of answer set solvers to find expressions that hold for every counter example. More generally, techniques for summarising the answer sets of a program would be of use.

References

1. Janet Barnes, Roderick Chapman, Randy Johnson, James Widmaier, Bill Everett, and David Cooper. Engineering the Tokeneer enclave protection software. In *ISSSE '06*. IEEE, 2006.
2. Mike Barnett and K. Rustan M. Leino. Weakest-precondition of unstructured programs. In *Proceedings of the 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 82–87. ACM, September 2005.
3. Stefan Berghofer. Verification of Dependable Software using SPARK and Isabelle. In Jörg Brauer, Marco Roveri, and Hendrik Tews, editors, *Proceedings of the 6th International Workshop on Systems Software Verification (SSV 2011)*, pages 48–65. TU Dresden, August 2011. Technical report TUD–FI11.
4. Jasmin Christian Blanchette and Tobias Nipkow. Nitpick: A counterexample generator for higher-order logic based on a relational model finder. In *Interactive Theorem Proving*, volume 6172, pages 131–146. Springer, July 2010.
5. Georg Boenn, Martin Brain, Marina De Vos, and John ffitch. Automatic music composition using answer set programming. *The Theory and Practise of Logic Programming*, 11(2-3):397–427, February 2011.
6. Martin Brain, Owen Cliffe, and Marina De Vos. A pragmatic programmer's guide to answer set programming. In Marina De Vos and Torsten Schaub, editors, *Proceedings of SEA09*, pages 49–63. Electronic proceedings at http://sea09.cs.bath.ac.uk, September 2009.
7. Martin Brain, Tom Crick, Marina De Vos, and John Fitch. Toast: Applying answer set programming to superoptimisation. In Sandro Etalle and Mirosław Truszczyński, editors, *Proceedings of ICLP06*, volume 4079, pages 270–284. Springer, 2006.
8. Martin Brain and Florian Schanda. A low cost technique for distributed and incremental verification. In *Verified Software: Theories, Tools and Experiments*, pages 114–129. Springer, 2012.
9. Roderick Chapman, Eric Botcazou, and Angela Wallenburg. SPARKSkein: A Formal and Fast Reference Implementation of Skein. In *SBMF 2011*, volume 7021 of *LNCS*, pages 16–27. Springer, 2011.
10. Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification*, volume 1855 of *LNCS*, pages 154–169, 2000.
11. Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, MichałMoskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A practical system for verifying concurrent c. In *Theorem Proving in Higher Order Logics*, volume 5674 of *LNCS*, pages 23–42. Springer, August 2009.
12. Sylvain Conchon, Evelyne Contejean, and Johannes Kanig. Ergo: A theorem prover for polymorphic first-order logic modulo theories, 2006.
13. M. Croxford and J. Sutton. Breaking through the V and V Bottleneck. In *Ada in Europe 1995*, volume 1031 of *LNCS*. Springer, 1996.
14. CVC3: An automatic theorem prover for satisfiability modulo theories (SMT). `http://www.cs.nyu.edu/acsys/cvc3`, 2006.
15. L. de Moura and N. Bjørner. Z3: An efficient SMT solver. *Tools and Algorithms for the Construction and Analysis of Systems*, 4963:337–340, 2008.
16. Rina Dechter. *Constraint Processing*. Morgan Kaufmann Publishers, 2003.
17. Ewen Denney and Bernd Fischer. Explaining verification conditions. In *Algebraic Methodology and Software Technology*, volume 5140 of *LNCS*, pages 145–159. Springer, 2008.
18. M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and M. Schneider. Potassco: The Potsdam answer set solving collection. *AI Communications*, 24(2):105–124, 2011.

19 M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. Advanced preprocessing for answer set solving. In M. Ghallab, C. Spyropoulos, N. Fakotakis, and N. Avouris, editors, *Proceedings of the Eighteenth European Conference on Artificial Intelligence (ECAI'08)*, pages 15–19. IOS Press, 2008.

20 Claire Le Goues, K. Rustan M. Leino, and MichałMoskal. The Boogie verification debugger. In *Software Engineering and Formal Methods*, volume 7041 of *LNCS*, pages 407–414. Springer, November 2011.

21 Anthony Hall and Roderick Chapman. Correctness By Construction: Developing a Commercial Secure System. *IEEE Software*, pages 18–25, Jan/Feb 2002.

22 Paul B. Jackson and Grant Olney Passmore. Proving SPARK Verification Conditions with SMT solvers. `http://homepages.inf.ed.ac.uk/pbj/papers/vct-dec09-draft.pdf`, December 2009.

23 K. Rustan M. Leino, Todd Millstein, and James B. Saxe. Generating error traces from verification-condition counterexamples. *Science of Computer Programming*, 55(1-3):209–226, March 2005.

24 K. Rustan M. Leino, MichałMoskal, and Wolfram Schulte. Verification condition splitting. Unpublished report., 2008.

25 Peter Müller and Joseph N. Ruskiewicz. Using debuggers to understand failed verification attempts. In *FM 2011: Formal Methods*, volume 6664 of *LNCS*, pages 73–87. Springer, 2011.

26 Alexander Senier. libsparkcrypto - a cryptographic library implemented in SPARK. `http://senier.net/libsparkcrypto`, 2010.

Generating Event-Sequence Test Cases by Answer Set Programming with the Incidence Matrix

Mutsunori Banbara[1], Naoyuki Tamura[1], and Katsumi Inoue[2]

1 **Information Science and Technology Center, Kobe University**
1-1 Rokko-dai, Nada-ku, Kobe, Hyogo 657-8501, Japan
`{banbara,tamura}@kobe-u.ac.jp`

2 **National Institute of Informatics**
2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo 101-8430, Japan
`inoue@nii.ac.jp`

Abstract

The effective use of ASP solvers is essential for enhancing efficiency and scalability. The incidence matrix is a simple representation used in Constraint Programming (CP) and Integer Linear Programming for modeling combinatorial problems. Generating test cases for event-sequence testing is to find a sequence covering array (SCA). In this paper, we consider the problem of finding optimal sequence covering arrays by ASP and CP. Our approach is based on an effective combination of ASP solvers and the incidence matrix. We first present three CP models from different viewpoints of sequence covering arrays: the naïve matrix model, the event-position matrix model, and the incidence matrix model. Particularly, in the incidence matrix model, an SCA can be represented by a $(0,1)$-matrix called the incidence matrix of the array in which the coverage constraints of the given SCA can be concisely expressed. We then present an ASP program of the incidence matrix model. It is compact and faithfully reflects the original constraints of the incidence matrix model. In our experiments, we were able to significantly improve the previously known bounds for many arrays of strength three. Moreover, we succeeded either in finding optimal solutions or in improving known bounds for some arrays of strength four.

1998 ACM Subject Classification D.1.6 Logic Programming; D.2.5 Testing and Debugging

Keywords and phrases Event-Sequence Testing, Answer Set Programming, Matrix Model, Constraint Programming, Propositional Satisfiability (SAT)

Digital Object Identifier 10.4230/LIPIcs.ICLP.2012.86

1 Introduction

Recent development of Answer Set Programming (ASP) [3, 15, 21] suggests a successful direction to extend logic programming to be more expressive and more efficient. ASP provides a rich modeling language and can be well suited for modeling combinatorial problems in Computer Science and Artificial Intelligence: multi-agent systems, systems biology, planning, scheduling, semantic web, and Constraint Satisfaction Problems (CSPs). Remarkable improvements in the efficiency of ASP solvers have been made over the last decade, through the adoption of advanced techniques of Constraint Programming (CP) and Propositional Satisfiability (SAT). Such improvements encourage researchers to solve hard combinatorial problems by using ASP.

Combinatorial testing is an effective black-box testing method to detect elusive failures of hardware/software. The basic idea is based on the observations that most failures are caused by interactions of multiple components. The number of test cases is therefore much smaller

© Mutsunori Banbara, Naoyuki Tamura, and Katsumi Inoue;
licensed under Creative Commons License ND
Technical Communications of the 28th International Conference on Logic Programming (ICLP'12).
Editors: A. Dovier and V. Santos Costa; pp. 86–97

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

than exhaustive testing. Generating test cases for combinatorial testing is to find a *Covering Array* (CA) in Combinatorial Designs [2, 4, 5, 6, 7, 8, 9, 17, 18, 20, 22, 23, 27, 28]. However, these CA-based combinatorial testing methods can not be directly applied to detect failures that are caused by a particular *event sequence*, an ordering of multiple events to be processed.

Event-sequence testing is a combinatorial testing method focusing on event-driven hardware/software. Suppose we want to test a system with 10 events. We have $10! = 3,628,800$ test cases for exhaustive testing. Instead, we might be satisfied with test cases that exercise all possible 3-sequences of 10 events (strength three event-sequence testing). Naively, we need ${}_{10}P_3 = 8 \times 9 \times 10 = 720$ test cases. We can reduce to less than 240 since one test case covers at least three 3-sequences. The question is "what is the smallest number of test cases that we need now?". It comes down to an instance of the problem of finding optimal *Sequence Covering Array* (SCA) proposed by Kuhn et al [19]. A sequence covering array provides a set of test cases, where each row of the array can be regarded as an event sequence for an individual test case. Fig.1 shows an optimal sequence covering array of 11 rows, an answer of the question above.

ASP solvers have an important role in the latest ASP technology. The effective use of them is essential for enhancing efficiency and scalability. The *incidence matrix* is a simple representation used in CP and integer linear programming for modeling combinatorial problems such as balanced incomplete block designs [9]. Our approach is based on an effective combination of ASP solvers and the incidence matrix.

In this paper, we consider the problem of finding optimal sequence covering arrays by ASP and CP. We first present three CP models from different viewpoints of sequence covering arrays: *naïve matrix model*, *event-position matrix model*, and *incidence matrix model*. Particularly, in the incidence matrix model, an SCA can be represented by a $(0,1)$-matrix called the incidence matrix of the array in which the coverage constraints of the given SCA can be concisely expressed. We then present an ASP program of the incidence matrix model. It is compact and faithfully reflects the original constraints of the incidence matrix model. For example, it requires only 8 rules for the arrays of strength three. From the perspective of ASP, Erdem et al. recently proposed an ASP-based approach for event-sequence testing [11], and have shown that it enables a tester to rapidly specify problems and to experiment with different formulations at a purely declarative level.

In our experiments, we were able to significantly improve the previously known bounds obtained by a greedy algorithm [19] and an ASP-based approach [11] for many arrays of strength three with small to large sizes of events. Moreover, we succeeded either in finding optimal solutions or in improving known bounds for some arrays of strength four.

2 Sequence Covering Arrays and Related Work

▶ **Definition 1.** A *sequence covering array* $SCA(n; S, t)$ is an $n \times |S|$ (n rows and $|S|$ columns) array $A = (a_{ij})$ over a finite set S of symbols with the property that

- each row of A is a permutation of S, and
- for each t-sequence $\sigma = (e_1, e_2, \ldots, e_t)$ over S, there exists at least one row r with column indices $1 \le c_1 < c_2 < \cdots < c_t \le |S|$ such that $e_i = a_{rc_i}$ for all $1 \le i \le t$.

The parameter n is the size of the array, S is the set of events, and t is the *strength* of the array. Then trivial case when $t = 2$ is excluded from further consideration.

▶ **Definition 2.** The *sequence covering array number* $SCAN(S, t)$ is the smallest n for which an $SCA(n; S, t)$ exists.

▶ **Definition 3.** A sequence covering array $SCA(n; S, t)$ is *optimal* if $SCAN(S, t) = n$.

a	**b**	**c**	d	e	f	g	h	i	j
f	**a**	i	g	j	h	e	**c**	d	**b**
h	d	i	**b**	e	**a**	j	g	f	**c**
i	**c**	j	d	**b**	**a**	h	f	e	g
g	d	f	e	b	a	h	j	c	i
d	j	h	**c**	g	e	**a**	i	f	**b**
g	c	b	j	i	e	h	a	f	d
h	j	e	b	f	i	g	a	d	c
i	h	f	c	b	g	d	a	e	j
e	f	j	d	g	i	**b**	**c**	**a**	h
e	d	c	j	i	f	h	g	a	b

- Each event is represented as an alphabet instead of an integer.
- Each row represents an event sequence.
- We highlight the different 3-sequences over $\{a, b, c\}$ to show all possible 3-sequences (six permutations) occur at least once.
- This property holds for all 3-sequences over $\{a, b, c, d, e, f, g, h, i, j\}$.

Figure 1 An optimal sequence covering array $SCA(11; 10, 3)$.

▶ **Notation 4.** Let s be an integer. $SCA(n; s, t)$ and $SCAN(s, t)$ are intended to denote, respectively, $SCA(n; \{1, \dots, s\}, t)$ and $SCAN(\{1, \dots, s\}, t)$.

Fig. 1 shows an example of $SCA(11; 10, 3)$, a sequence covering array of strength $t = 3$ with $s = 10$ events. It is an optimal sequence covering array which has $n = 11$ rows.

In this paper, we define two kinds of problems to make our approach more understandable. For a given tuple $\langle n, s, t \rangle$, *SCA decision problem* is the problem to decide whether an $SCA(n; s, t)$ exists or not, and find it if exists. For a given pair $\langle s, t \rangle$, *SCA optimization problem* is the problem to find an optimal covering array $SCA(n; s, t)$. Oetsch et al. have recently proved that the *Generalised Event Sequence Testing* (GEST) problem is NP-complete [1]. Most *SCA* decision problems studied in this paper are special cases of GEST.

Kuhn et al. proposed a greedy algorithm for solving the *SCA* optimization problems [19]. The practical effectiveness, especially scalability, of their algorithm has been shown by the fact that they succeeded in finding upper bounds for the arrays of strength $3 \leq t \leq 4$ with $s \leq 80$ events. We refer to their algorithm [19] as Kuhn's encoding.

Erdem et al. proposed ASP encodings and an ASP-based greedy algorithm for solving the *SCA* optimization problems [11]. They have found and proved optimal solutions for the arrays of strength $t = 3$ with $5 \leq s \leq 8$ events through their exact ASP encodings. Moreover, their ASP-based greedy algorithm that synergistically integrates ASP with a greedy method is designed to improve the scalability issue of the ASP encodings. We refer to their encodings [11] as Erdem's encoding. When we need to distinguish between their exact ASP encodings and greedy algorithm, we refer to the former as Erdem's exact encoding and the latter as Erdem's greedy encoding.

3 Constraint Programming Models

We propose three different CP models for solving the *SCA* decision problems: the naïve matrix model, the event-position matrix model, and the incidence matrix model. We assume throughout that we have an $SCA(n; s, 3)$, a sequence covering array of strength $t = 3$, for the sake of clarity. Note that our models can be extended in a straightforward way to the case of any strength $t \geq 3$. We also use a sequence covering array $SCA(6; \{a, b, c, d\}, 3)$ of Fig. 2 as a running example.

[1] Oetsch et al. personal communication

1	2	3	4
a	d	b	c
d	c	b	a
c	d	a	b
a	c	b	d
b	d	a	c
b	c	a	d

Figure 2 A sequence covering array $SCA(6; \{a, b, c, d\}, 3)$.

a	b	c	d
1	3	4	2
4	3	2	1
3	4	1	2
1	3	2	4
3	1	4	2
3	1	2	4

Figure 3 The event-position matrix of $SCA(6; \{a, b, c, d\}, 3)$ shown in Fig. 2.

3.1 Naïve Matrix Model

For a given SCA decision problem for $SCA(n; s, 3)$, the most direct model would be using an $n \times s$ (n rows and s columns) matrix of integer variables $m_{r,i}$ $(1 \le r \le n, 1 \le i \le s)$. The domain of each variable is $\{1, 2, \ldots, s\}$. This matrix identifies a sequence covering array itself. We also use the auxiliary binary variables $a_{r,(i,j,k),(p,q,u)}$ with $1 \le r \le n$, $1 \le i < j < k \le s$, $1 \le p, q, u \le s$, $p \ne q$, $p \ne u$, and $q \ne u$. The variable $a_{r,(i,j,k),(p,q,u)}$ is intended to denote $m_{r,i} = p$, $m_{r,j} = q$, and $m_{r,k} = u$ in the matrix.

A *global constraint* is a constraint that can specify a relation between an arbitrary number of variables [26]. In the naïve matrix model, we use the alldifferent constraint that is one of the best known and most studied global constraint in CP. The constraint alldifferent$(X_1, X_2, \ldots, X_\ell)$ ensures that the values assigned to the variables $X_1, X_2, \ldots, X_\ell$ must be pairwise distinct.

The constraints for $SCA(n; s, 3)$ are defined as follows.

- Permutation constraints:

$$\text{alldifferent}(m_{r,1}, m_{r,2}, \ldots, m_{r,s}) \qquad (1)$$

- Channeling constraints:

$$a_{r,(i,j,k),(p,q,u)} = 1 \Leftrightarrow (m_{r,i} = p) \wedge (m_{r,j} = q) \wedge (m_{r,k} = u) \qquad (2)$$

- Coverage constraints:

$$\sum_{\substack{1 \le r \le n \\ 1 \le i < j < k \le s}} a_{r,(i,j,k),(p,q,u)} \ge 1 \qquad (3)$$

where $1 \le r \le n$, $1 \le i < j < k \le s$, $1 \le p, q, u \le s$, $p \ne q$, $p \ne u$, and $q \ne u$.

The permutation constraints can be easily expressed by using alldifferent constraints of (1). That is, for every row, one alldifferent is enforced to ensure that every event in the range 1 to s occurs exactly once. The constraints (2) express the channeling constraints. The constraints (3) express the coverage constraints such that every 3-sequence of the events $\{1, \ldots, s\}$ occurs at least once in the matrix.

Note that the constraints of leftward arrows in (2) can be omitted. Even if they may be omitted, we can still get a solution. For any solution, the constraints (3) ensure that every 3-sequence of the events occurs at least once. For each such an occurrence, the corresponding entries (i.e. a 3-tuple of variables) of the matrix are derived from the constraints (2). The condition that each row is a permutation of the events is ensured by the constraints (1).

The drawback of this model is not only the number of instances required for the coverage constraints (3), but also the number of variables contained within each cardinality constraint in (3). We need in total ${}_sP_3$ cardinality constraints, and each of them contains $n\binom{s}{3}$ variables. To avoid this problem, we propose another matrix model, called the event-position matrix model.

	a	a	b	b	c	c	a	a	b	b	d	d	a	a	c	c	d	d	b	b	c	c	d	d
	b	c	a	c	a	b	b	d	a	d	a	b	c	d	a	d	a	c	c	d	b	d	b	c
	c	b	c	a	b	a	d	b	d	a	b	a	d	c	d	a	c	a	d	c	d	b	c	b
a d b c	1	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0
d c b a	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	1
c d a b	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0	0	1	0	0
a c b d	0	1	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0
b d a c	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	1	0	0	0	0
b c a d	0	0	0	1	0	0	0	0	1	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0

Figure 4 The incidence matrix of $SCA(6; \{a, b, c, d\}, 3)$ shown in Fig. 2.

3.2 Event-Position Matrix Model

We give another view of sequence covering arrays. For a given sequence covering array $A = (a_{ij})$, we can represent it by the *event-position matrix* of the array. The event-position matrix $B = (b_{ie})$ of A is defined so that $b_{ie} = j$ if $a_{ij} = e$. That is, the rows are the same as before but the columns are labeled with the distinct events, and each entry represents the position of its corresponding event. Fig. 3 shows the event-position matrix of $SCA(6; \{a, b, c, d\}, 3)$ shown in Fig. 2.

For a given SCA decision problem for $SCA(n; s, 3)$, in the event-position matrix model, we use again an $n \times s$ matrix of integer variables $x_{r,i}$ $(1 \leq r \leq n, 1 \leq i \leq s)$. It identifies an event-position matrix instead of a sequence covering array. The domain of each variable is $\{1, 2, \ldots, s\}$. We also use the auxiliary binary variables $y_{r,(i,j,k)}$ with $1 \leq r \leq n$, $1 \leq i, j, k \leq s$, $i \neq j$, $i \neq k$, and $j \neq k$. The variable $y_{r,(i,j,k)}$ is intended to denote $x_{r,i} < x_{r,j} < x_{r,k}$ in the event-position matrix.

The constraints for $SCA(n; s, 3)$ are defined as follows.

- Permutation constraints:

$$\mathsf{alldifferent}(x_{r,1}, x_{r,2}, \ldots, x_{r,s}) \tag{4}$$

- Channeling constraints:

$$y_{r,(i,j,k)} = 1 \Leftrightarrow (x_{r,i} < x_{r,j}) \wedge (x_{r,i} < x_{r,k}) \wedge (x_{r,j} < x_{r,k}) \tag{5}$$

- Coverage constraints:

$$\sum_{r} y_{r,(i,j,k)} \geq 1 \tag{6}$$

where $1 \leq r \leq n$, $1 \leq i, j, k \leq s$, $i \neq j$, $i \neq k$, and $j \neq k$.

The constraints (4) is the same as (1) of the previous model except that each argument represents the position of the event. The constraints (5) express the channeling constraints. The coverage constraints can be concisely expressed by the constraints (6). That is, for every 3-sequence (i, j, k) of the events, one cardinality constraint is enforced to ensure that there is at least one row r that satisfies the condition $x_{r,i} < x_{r,j} < x_{r,k}$. This means that we cover all possible 3-sequence.

The comparisons $x_{r,i} < x_{r,k}$ in (5) are clearly redundant and can be omitted, but we leave them because of efficiency improvements. The constraints of leftward arrows in (5) can be also omitted for the same reason as before.

3.3 Incidence Matrix Model

We now give yet another view of sequence covering arrays. For a given sequence covering array, we can represent it by the *incidence matrix* of the array. Each row is labeled with one row (i.e. an event sequence) of the array. Each column is labeled with one of all possible t-sequences of the events. The incidence matrix $C = (c_{ij})$ of $SCA(n; s, t)$ is a $(0, 1)$-matrix with n rows and ${}_sP_t$ columns such that $c_{ij} = 1$ if the t-sequence j is a sub-sequence of the event sequence i, and $c_{ij} = 0$ otherwise.

Fig. 4 shows the incidence matrix of $SCA(6; \{a, b, c, d\}, 3)$ shown in Fig. 2. Each row is labeled with one row of the $SCA(6; \{a, b, c, d\}, 3)$. Each of ${}_4P_3 = 24$ columns is labeled with one of all possible 3-sequences of the events $\{a, b, c, d\}$. The labels of the columns are written vertically. For example, the entry in the first row and first column is a 1 since "$a\ b\ c$" is a sub-sequence of "$a\ d\ b\ c$".

In contrast, on the incidence matrix, let us consider the constraints that must be satisfied for $SCA(6; \{a, b, c, d\}, 3)$. Each column has at least one 1 (coverage constraints). From a viewpoint of 3-combinations of the events $\{a, b, c, d\}$, there are $6 \times \binom{4}{3} = 24$ sub-matrices with one row and six columns. Each sub-matrix sharing the same three events in the columns has exactly one 1. Furthermore, for each row, such occurrences of 1's are consistent with each other in terms of the ordering of the events.

For a given SCA decision problem for $SCA(n; s, 3)$, in the incidence matrix model, we use an $n \times {}_sP_3$ matrix of binary variables $y_{r,(i,j,k)}$ with $1 \le r \le n$, $1 \le i, j, k \le s$, $i \ne j$, $i \ne k$, and $j \ne k$. We can express the permutation constraints by using only the $y_{r,(i,j,k)}$ variables, but it requires a large number of constraints that are very costly to deal with. To avoid this problem, we introduce the auxiliary binary variables $pr_{r,(i,j)}$ with $1 \le r \le n$, $1 \le i, j \le s$, and $i \ne j$. The variable $pr_{r,(i,j)}$ is intended to denote the event i precedes the event j in the row r.

The constraints for $SCA(n; s, 3)$ are defined as follows.

- Permutation constraints:

$$\big((pr_{r,(i,j)} = 1) \wedge (pr_{r,(j,k)} = 1)\big) \Rightarrow pr_{r,(i,k)} = 1 \tag{7}$$

$$\neg(pr_{r,(i,j)} = 1) \vee \neg(pr_{r,(j,i)} = 1) \tag{8}$$

$$(pr_{r,(i,j)} = 1) \vee (pr_{r,(j,i)} = 1) \tag{9}$$

- Channeling constraints:

$$y_{r,(i,j,k)} = 1 \Leftrightarrow (pr_{r,(i,j)} = 1) \wedge (pr_{r,(i,k)} = 1) \wedge (pr_{r,(j,k)} = 1) \tag{10}$$

- Coverage constraints:

$$\sum_r y_{r,(i,j,k)} \ge 1 \tag{11}$$

where $1 \le r \le n$, $1 \le i, j, k \le s$, $i \ne j$, $i \ne k$, and $j \ne k$.

The permutation constraints can be expressed by enforcing total ordering on the events: (7) for transitivity, (8) for asymmetry, and (9) for comparability (totality). Note that the constraints (7) can be replaced with the following arithmetic constraints (12), and the constraints (8) and (9) with (13).

$$pr_{r,(i,j)} + pr_{r,(j,k)} - pr_{r,(i,k)} \le 1 \tag{12}$$

$$pr_{r,(i,j)} + pr_{r,(j,i)} = 1 \tag{13}$$

The channeling constraints are expressed by the constraints (10) that are slightly modified to adjust the pr variables compared with (5). The coverage constraints (11) are the same as (6). The equations $pr_{r,(i,k)} = 1$ in (10) and the constraints of leftward arrows in (10) can be omitted for the same reason as before.

Table 1 Benchmark results of different CP models for $SCA(n; s, t)$.

n	s	t	Result	Incidence	Incidence (lex-row)	E-Position	E-Position (snake lex)	E-Position (double lex)
6	5	3	UNSAT	1.320	< 0.000	5.457	0.008	0.011
7*	5	3	SAT	< 0.000	< 0.000	0.005	0.010	0.010
7	6	3	UNSAT	1327.350	0.110	*T.O*	0.383	0.588
8*	6	3	SAT	0.005	0.006	0.012	0.016	0.022
7	7	3	UNSAT	1442.410	0.180	*T.O*	1.921	5.509
8*	7	3	SAT	0.008	0.015	0.032	0.077	0.027
7	8	3	UNSAT	*T.O*	0.390	*T.O*	8.280	28.870
8*	8	3	SAT	0.070	0.094	7.870	2.815	18.160
9	9	3	SAT	0.075	0.242	6.139	6.070	64.570
9	10	3	SAT	11.896	5.890	982.580	1188.240	*T.O*
10	11	3	SAT	0.047	0.052	59.670	30.216	67.031
10	12	3	SAT	0.046	0.456	774.338	117.258	*T.O*
10	13	3	SAT	0.980	0.371	*T.O*	*T.O*	*T.O*
10	14	3	SAT	5.546	25.880	*T.O*	*T.O*	*T.O*
10	15	3	SAT	541.480	443.012	*T.O*	*T.O*	*T.O*
11	16	3	SAT	89.580	107.334	*T.O*	*T.O*	*T.O*
11	17	3	SAT	62.560	*T.O*	*T.O*	*T.O*	*T.O*
12	18	3	SAT	3.603	3.830	*T.O*	*T.O*	*T.O*
12	19	3	SAT	2.851	18.840	*T.O*	*T.O*	*T.O*
12	20	3	SAT	22.500	180.256	*T.O*	*T.O*	*T.O*
12	21	3	SAT	1353.810	824.680	*T.O*	*T.O*	*T.O*
13	22	3	SAT	29.660	9.783	*T.O*	*T.O*	*T.O*
13	23	3	SAT	*T.O*	898.820	*T.O*	*T.O*	*T.O*
14	24	3	SAT	4.838	13.962	*T.O*	*T.O*	*T.O*
14	25	3	SAT	25.600	7.763	*T.O*	*T.O*	*T.O*
14	26	3	SAT	67.850	8.864	*T.O*	*T.O*	*T.O*
14	27	3	SAT	1126.390	251.660	*T.O*	*T.O*	*T.O*
14	28	3	SAT	*T.O*	641.320	*T.O*	*T.O*	*T.O*
15	29	3	SAT	127.470	18.955	*T.O*	*T.O*	*T.O*
15	30	3	SAT	673.210	190.200	*T.O*	*T.O*	*T.O*
17	40	3	SAT	771.990	*T.O*	*M.O*	*M.O*	*M.O*
23	5	4	UNSAT	*T.O*	0.046	*T.O*	3.554	4.980
24*	5	4	SAT	0.100	0.081	94.488	1.150	0.690
23	6	4	UNSAT	*T.O*	0.260	*T.O*	*T.O*	*T.O*
24*	6	4	SAT	0.230	0.460	376.184	*T.O*	*T.O*
38	7	4	SAT	*T.O*	40.390	*T.O*	*T.O*	*T.O*
47	8	4	SAT	*T.O*	688.400	*T.O*	*T.O*	*T.O*
52	9	4	SAT	*T.O*	51.950	*T.O*	*T.O*	*T.O*
58	10	4	SAT	341.420	659.830	*T.O*	*T.O*	*T.O*
65	11	4	SAT	*T.O*	159.330	*T.O*	*T.O*	*T.O*
69	12	4	SAT	*T.O*	243.590	*T.O*	*T.O*	*T.O*

4 Experiments

To evaluate the effectiveness of our CP models, we solve SCA optimization problems (35 problems in total) of strength $3 \leq t \leq 4$ with small to moderate sizes of events. For each problem, we solve multiple SCA decision problems of $SCA(n; s, t)$ with varying the value of n. Such decision problems contain both satisfiable and unsatisfiable problems and their optimal solutions exist on the boundaries.

For each SCA decision problem, we represent it by using our models with and without breaking the symmetries. More precisely, we apply the *lexicographic ordering constraints* for breaking the row symmetry in the naïve matrix model and the incidence matrix model. In the event-position matrix model, for breaking the row and column symmetry, we apply the *double lex* [12] and the *snake lex* [16] separately. In addition, we constrain every entry in the first row to be "1 2 ... s" for the naïve matrix model and the event-position matrix model with double lex. We note that applying these constraints for breaking the symmetries does not lose any solutions. For every model, we omit the constraints of leftward arrows in the channeling constraints.

For solving every model of each decision problem, we use a SAT-based CSP solver Sugar [2], an award-winning system in GLOBAL category (including global constraints such as alldifferent) of the 2008 and 2009 International CSP Solver Competitions. Sugar solves a finite linear CSP by encoding it into SAT and then solving the SAT-encoded problem by using an external SAT solver at the back-end. The SAT encoding that Sugar adopted is called the *order encoding* [24, 25]. It is efficient in the sense that unit propagation keeps the *bounds consistency* in original CSPs. We use MiniSat 2.2.0 (core) [10], Glucose 2 [1], and clasp 2.0.2 [13, 14] as back-end SAT solvers. The first two are efficient CDCL SAT solvers. The last one clasp is not only a state-of-the-art ASP solver, but also an efficient SAT solver. In particular, Glucose and clasp are award-winning solvers in the 2011 SAT Competition.

Table 1 shows the best CPU time in seconds of three SAT solvers for solving $SCA(n; s, t)$. We only shows our best lower and/or upper bounds of n for each SCA optimization problem. We use the symbol "$*$" to indicate that the value of n is optimal. The column "Result" indicates whether it is satisfiable (`SAT`) or unsatisfiable (`UNSAT`). The columns "Incidence" and "E-Position" indicate the incidence matrix model and the event-position matrix model respectively. Note that we exclude the results of the naïve matrix model from Table 1 since it was quite inefficient. All times were collected on Mac OS X with Intel Xeon 3.2GHz and 16GB memory. We set a timeout ("$T.O$") including the encoding time of Sugar to 1800 seconds for each SCA decision problem. The "$M.O$" indicates a memory error of SAT solvers.

We observe in Table 1 that the incidence matrix based models ("Incidence" and "Incidence with lex-row") are faster and much more scalable to the number of events than the event-position matrix based models ("E-Position", "E-Position with snake lex" and "E-Position with double lex"). "Incidence with lex-row" solved 39 SCA decision problems out of 41, rather than 31 of "Incidence", 14 of "E-Position with snake lex", 12 of "E-Position with double lex", and 11 of "E-Position". The main difference between two incidence matrix based models is that "Incidence with lex-row" were able to give solutions for 7 arrays of strength $t = 4$ not solved in timeout by "Incidence".

Our models reproduced and re-proved 4 previously known optimal solutions. Moreover, we found optimal solutions for $SCAN(5, 4)$ and $SCAN(6, 4)$. We also improved on previously known upper bounds [11, 19] for the arrays of strength $t = 3$ with $18 \leq s \leq 40$ events and strength $t = 4$ with $5 \leq s \leq 12$ events except $s = 10$.

5 An ASP Program of the Incidence Matrix Model

We present an ASP program of our best incidence matrix model. It is compact and faithfully reflects the original constraints of the incidence matrix model. Our program has $\binom{t}{2} + 5$ rules for the SCA decision problem of $SCA(n; s, t)$. For example, Fig. 5 shows the ASP program `sca3.lp` for $SCA(n; s, 3)$, which has only $\binom{3}{2} + 5 = 8$ rules. Note that this program can be extended in a straightforward way to the case of any strength $t \geq 3$. We use the syntax supported by the solver clasp and the grounder gringo [13, 14].

In Fig. 5, the first two rules `row(1..n)` and `col(1..s)` express that the row indices are integers in the range `1` to `n`, and the events are integers in the range `1` to `s`. The constants `n` and `s` are replaced with given values by a grounder. The third rule corresponds to the coverage constraints (11) where the predicate `y(R,I,J,K)` expresses the binary variable $y_{r,(i,j,k)}$. To express the coverage constraints, it uses special constructs called *cardinality expressions* of the form $\ell\{a_1, \ldots, a_k\}u$ where each a_i is an atom and ℓ and u are non-negative

[2] `http://bach.istc.kobe-u.ac.jp/sugar/`

```
% SCA(n;s,3)
row(1..n). col(1..s).

% coverage constraints
1{ y(R,I,J,K) : row(R) } :- col(I;J;K), I!=J, I!=K, J!=K.

% channeling constraints
pr(R,I,J) :- y(R,I,J,K).
pr(R,I,K) :- y(R,I,J,K).
pr(R,J,K) :- y(R,I,J,K).

% asymmetry & comparability constraints
1{ pr(R,I,J), pr(R,J,I) }1 :- row(R), col(I;J), I<J.

% transitivity constraints
pr(R,I,K) :- pr(R,I,J), pr(R,J,K), row(R), col(I;J;K), I!=J, I!=K, J!=K.
```

Figure 5 `sca3.lp`: An ASP program for $SCA(n; s, 3)$.

integers denoting the lower bound and the upper bound of the cardinality expression. The third rule first generates a candidate for the incidence matrix, and then constrains a lower bound on the number of atoms is 1 for each column (i.e. each 3-sequence of the events). From the fourth to the sixth rule, the predicate `pr(R,I,J)` expresses the auxiliary binary variable $pr_{r,(i,j)}$. These three rules correspond to the constraints of rightward arrows in the channeling constraints (10). The seventh rule again uses cardinality expressions to express the asymmetry and comparability constraints (13). The transitivity constraints (7) are expressed by the last rule. The command "`gringo sca3.lp -c n=`n `-c s=`s `| clasp`" gives an answer set of an $SCA(n; s, 3)$ decision problem. We can get a solution of the original problem by decoding the resulting answer set.

6 Comparison

We compare our ASP program with different approaches. We use Kuhn's benchmark set that consists of 62 SCA optimization problems for $SCAN(s, t)$ of strength $3 \leq t \leq 4$ with $5 \leq s \leq 80$ events. We execute our ASP program by using clasp 2.0.4 and gringo 2.0.5 to solve multiple SCA decision problems of $SCA(n; s, t)$ with varying the value of n for each optimization problem. All times were measured on Mac OS X with Intel Xeon 2.66GHz and 24GB memory. We set a timeout for clasp to 3600 seconds for each $SCA(n; s, t)$.

Table 2 shows the comparison results of different approaches on the best known upper bounds of $SCAN(s, t)$. Our comparison includes our ASP program with clasp, our CP models with Sugar, Erdem encoding [11], and Kuhn encoding [19]. We note that Erdem encoding is closely related to the event-position matrix model of our CP models. We highlight the best value of different approaches for each $SCAN(s, t)$. The symbol "-" is used to indicate that the result is not available in either our experiments or published literature.

In the case of strength $t = 3$, our ASP program with clasp were able to produce significantly improved bounds compared with those in Erdem greedy encoding and Kuhn encoding. The more events are considered, the more significant are the improvements. For example, when $s = 80$ events, it produced an array of $n = 24$ rows compared with 38 of Erdem and 42 of Kuhn. On average, it improved every bound of Erdem greedy encoding and Kuhn encoding by 10 and 9 rows respectively. Compared with Erdem exact encoding, our ASP program can be more scalable. In the case of strength $t = 4$, although not able to match Erdem greedy encoding for $SCAN(10, 4)$ and $SCAN(20, 4)$, our ASP program were able to improve every bound of Kuhn encoding for the arrays with $s \leq 23$ events by 19 rows on average.

Table 2 Comparison of different approaches on the best known upper bounds of $SCAN(s,t)$.

s	Our ASP with clasp		Our CP with Sugar		Erdem exact encoding [11]		Erdem greedy encoding [11]		Kuhn encoding [19]	
	$t=3$	$t=4$	$t=3$	$t=4$	$t=3$	$t=4$	$t=3$	$t=4$	$t=3$	$t=4$
5	**7**	**24**	**7**	**24**	**7**	–	–	–	8	29
6	**8**	**24**	**8**	**24**	**8**	–	–	–	10	38
7	**8**	40	**8**	**38**	**8**	–	–	–	12	50
8	**8**	**44**	**8**	47	**8**	–	–	–	12	56
9	**9**	53	**9**	**52**	**9**	–	–	–	14	68
10	**9**	59	**9**	58	**9**	–	11	**55**	14	72
11	**10**	**65**	**10**	**65**	**10**	–	–	–	14	78
12	**10**	73	**10**	**69**	**10**	–	–	–	16	86
13	**10**	**77**	**10**	–	**10**	–	–	–	16	92
14	**10**	**81**	**10**	–	**10**	–	–	–	16	100
15	**10**	**84**	**10**	–	**10**	–	–	–	18	108
16	**11**	**89**	**11**	–	**11**	–	–	–	18	112
17	**11**	**91**	**11**	–	**11**	–	–	–	20	118
18	**12**	**97**	**12**	–	–	–	–	–	20	122
19	**12**	**100**	**12**	–	–	–	–	–	22	128
20	**12**	105	**12**	–	–	–	19	**104**	22	134
21	**12**	**104**	**12**	–	–	–	–	–	22	134
22	**13**	**111**	**13**	–	–	–	–	–	22	140
23	14	**112**	**13**	–	–	–	–	–	24	146
24	**14**	–	**14**	–	–	–	–	–	24	**146**
25	**14**	–	**14**	–	–	–	–	–	24	**152**
26	**14**	–	**14**	–	–	–	–	–	24	**158**
27	**14**	–	**14**	–	–	–	–	–	26	**160**
28	**14**	–	**14**	–	–	–	–	–	26	**162**
29	**15**	–	**15**	–	–	–	–	–	26	**166**
30	**15**	–	**15**	–	–	–	23	**149**	26	166
40	**17**	–	**17**	–	–	–	27	**181**	32	198
50	**19**	–	–	–	–	–	31	–	34	**214**
60	**21**	–	–	–	–	–	34	–	38	**238**
70	**22**	–	–	–	–	–	36	–	40	**250**
80	**24**	–	–	–	–	–	38	–	42	**264**

7 Conclusion

In this paper, we considered the problem of finding optimal sequence covering arrays by ASP and CP. We presented three CP models from different viewpoints of sequence covering arrays. Among them, the incidence matrix model is efficient in the sense that an SCA can be represented by the incidence matrix of the array in which the coverage constraints of the given SCA can be concisely expressed. We presented a new ASP program that is compact and faithfully reflects the incidence matrix model. To evaluate the effectiveness of our ASP program, we solved Kuhn's benchmark set that consists of 62 SCA optimization problems for $SCAN(s,t)$ of strength $3 \leq t \leq 4$ with $5 \leq s \leq 80$ events. We were able to significantly improve the previously known bounds for many arrays, as shown in Table 2. Moreover, we found optimal solutions for $SCAN(5,4)$ and $SCAN(6,4)$. However, we were still not able to find any solutions for $SCAN(s,4)$ with $24 \leq s \leq 80$ events because of expensive grounding, which shows a limitation of our approach at present. To overcome this problem, hybrid approaches to SCA, like Erdem greedy encoding, can be promising.

Our approach is based on an effective combination of ASP solvers and the incidence matrix. It can be applied to a wide range of combinatorial search problems such as balanced incomplete block designs [9] and SAT-based standard combinatorial testing [2].

References

1. Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern SAT solvers. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI 2009)*, pages 399–404, 2009.
2. Mutsunori Banbara, Haruki Matsunaka, Naoyuki Tamura, and Katsumi Inoue. Generating combinatorial test cases by efficient SAT encodings suitable for CDCL SAT solvers. In *Proceedings of the 17th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR-17), LNCS 6397*, pages 112–126, 2010.
3. Chitta Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2003.
4. D.A. Bulutoglu and F. Margot. Classification of orthogonal arrays by integer programming. *Journal of Statistical Planning and Inference*, 138:654–666, 2008.
5. M. A. Chateauneuf and Donald L. Kreher. On the state of strength-three covering arrays. *Journal of Combinatorial Designs*, 10(4):217–238, 2002.
6. David M. Cohen, Siddhartha R. Dalal, Michael L. Fredman, and Gardner C. Patton. The AETG system: An approach to testing based on combinatiorial design. *IEEE Transactions on Software Engineering*, 23(7):437–444, 1997.
7. Myra B. Cohen, Matthew B. Dwyer, and Jiangfan Shi. Constructing interaction test suites for highly-configurable systems in the presence of constraints: A greedy approach. *IEEE Transactions on Software Engineering*, 34(5):633–650, 2008.
8. Myra B. Cohen, Peter B. Gibbons, Warwick B. Mugridge, and Charles J. Colbourn. Constructing test suites for interaction testing. In *Proceedings of the 25th International Conference on Software Engineering (ICSE 2003)*, pages 38–48, 2003.
9. Charles J. Colbourn and Jeffrey H. Dinitz. *Handbook of Combinatorial Designs*. Chapman & Hall/CRC, 2007.
10. Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In *Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing (SAT 2003), LNCS 2919*, pages 502–518, 2003.
11. Esra Erdem, Katsumi Inoue, Johannes Oetsch, Jorg Puhrer, Hans Tompits, and Cemal Yilmaz. Answer-set programming as a new approach to event-sequence testing. In Teemu Kanstrén, editor, *Proceedings of the 3rd International Conference on Advances in System Testing and Validation Lifecycle (VALID 2011)*, pages 25–34. Xpert Publishing Services, 2011.
12. Pierre Flener, Alan M. Frisch, Brahim Hnich, Zeynep Kiziltan, Ian Miguel, Justin Pearson, and Toby Walsh. Breaking row and column symmetries in matrix models. In *Proceedings of the 8th International Joint Conference on Principles and Practice of Constraint Programming (CP 2002), LNCS 2470*, pages 462–476, 2002.
13. Martin Gebser, Benjamin Kaufmann, André Neumann, and Torsten Schaub. Conflict-driven answer set solving. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI 2007)*, pages 386–392. MIT Press, 2007.
14. Martin Gebser, Benjamin Kaufmann, and Torsten Schaub. The conflict-driven answer set solver clasp: Progress report. In Esra Erdem, Fangzhen Lin, and Torsten Schaub, editors, *Proceedings of the Tenth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2009), LNCS 5753*, pages 509–514. Springer, 2009.
15. Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 1070–1080. MIT Press, 1988.
16. Andrew Grayland, Ian Miguel, and Colva M. Roney-Dougal. Snake lex: An alternative to double lex. In *Proceedings of the 15th International Joint Conference on Principles and Practice of Constraint Programming (CP 2009), LNCS 5732*, pages 391–399, 2009.

17 Alan Hartman and Leonid Raskin. Problems and algorithms for covering arrays. *Discrete Mathematics*, 284(1–3):149–156, 2004.

18 Brahim Hnich, Steven David Prestwich, Evgeny Selensky, and Barbara M. Smith. Constraint models for the covering test problem. *Constraints*, 11(2-3):199–219, 2006.

19 D. Richard Kuhn, James M. Higdon, James F. Lawrence, Raghu N. Kacker, and Yu Lei. Combinatorial methods for event sequence testing. *submitted for publication*, 2010. Available at `http://csrc.nist.gov/groups/SNS/acts/documents/event-seq101008.pdf`.

20 Yu Lei and Kuo-Chung Tai. In-parameter-order: A test generation strategy for pairwise testing. In *Proceedings of 3rd IEEE International Symposium on High-Assurance Systems Engineering (HASE 1998)*, pages 254–261, 1998.

21 Ilkka Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Ann. Mathematics and Artificial Intelligence*, 25(3–4):241–273, 1999.

22 Kari J. Nurmela. Upper bounds for covering arrays by tabu search. *Discrete Applied Mathematics*, 138(1-2):143–152, 2004.

23 Toshiaki Shiba, Tatsuhiro Tsuchiya, and Tohru Kikuno. Using artificial life techniques to generate test cases for combinatorial testing. In *Proceedings of 28th International Computer Software and Applications Conference (COMPSAC 2004)*, pages 72–77, 2004.

24 Naoyuki Tamura, Akiko Taga, Satoshi Kitagawa, and Mutsunori Banbara. Compiling finite linear CSP into SAT. In *Proceedings of the 12th International Joint Conference on Principles and Practice of Constraint Programming (CP 2006), LNCS 4204*, pages 590–603, 2006.

25 Naoyuki Tamura, Akiko Taga, Satoshi Kitagawa, and Mutsunori Banbara. Compiling finite linear CSP into SAT. *Constraints*, 14(2):254–272, 2009.

26 Willem Jan van Hoeve and Irit Katriel. Global constraints. In Francesca Rossi, Peter van Beek, and Toby Walsh, editors, *Handbook of Constraint Programming*, pages 169–208. Elsevier, 2006.

27 Alan W. Williams. Determination of test configurations for pair-wise interaction coverage. In *Proceedings of 13th International Conference on Testing Communicating Systems (TestCom 2000)*, pages 59–74, 2000.

28 Hantao Zhang. Combinatorial designs by SAT solvers. In *Handbook of Satisfiability*, pages 533–568. IOS Press, 2009.

Towards Testing Concurrent Objects in CLP*

Elvira Albert, Puri Arenas, and Miguel Gómez-Zamalloa

DSIC, Complutense University of Madrid (UCM), E-28040 Madrid, Spain
`{elvira,puri}@sip.ucm.es, mzamalloa@fdi.ucm.es`

Abstract

Testing is a vital part of the software development process. It is even more so in the context of concurrent languages, since due to undesired task interleavings and to unexpected behaviours of the underlying task scheduler, errors can go easily undetected. This paper studies the extension of the CLP-based framework for glass-box test data generation of sequential programs to the context of *concurrent objects*, a concurrency model which constitutes a promising solution to concurrency in OO languages. Our framework combines standard termination and coverage criteria used for testing sequential programs with specific criteria which control termination and coverage from the concurrency point of view, e.g., we can limit the number of task interleavings allowed and the number of loop unrollings performed in each parallel component, etc.

1998 ACM Subject Classification D.2.5 Testing and Debugging, D.1.3 Concurrent Programming, D.1.6 Logic Programming, D.1.5 Object-oriented Programming

Keywords and phrases Testing, Glass-box Test Data Generation, Active Objects, Symbolic Execution

Digital Object Identifier 10.4230/LIPIcs.ICLP.2012.98

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 (called *heap* in what follows) to which the different tasks can access. The focus of this paper is on the development of automated techniques for testing *concurrent objects*. The actor-based paradigm [1] on which concurrent objects are based has lately regained attention as a promising solution to concurrency in OO languages. For many application areas, standard mechanisms like threads and locks are too low-level and have shown to be error-prone and, more importantly, not *modular* enough. The concurrent objects model is based on considering objects as the concurrency units, i.e., each object conceptually has a dedicated processor (and can run in parallel with other objects). Communication is based on asynchronous method calls with standard objects as targets. An essential difference with thread-based concurrency is that task scheduling is *cooperative*, i.e., switching between tasks of the same object happens only at specific scheduling points during the execution, which are explicit in the source code and can be syntactically identified.

* This work was funded in part by the Information & Communication Technologies program of the European Commission, Future and Emerging Technologies (FET), under the ICT-231620 *HATS* project, by the Spanish Ministry of Science and Innovation (MICINN) under the TIN-2008-05624 and PRI-AIBDE-2011-0900 projects, by UCM-BSCH-GR35/10-A-910502 grant and by the Madrid Regional Government under the S2009TIC-1465 *PROMETIDOS-CM* project.

© Elvira Albert, Puri Arenas, and Miguel Gómez-Zamalloa;
licensed under Creative Commons License ND

Technical Communications of the 28th International Conference on Logic Programming (ICLP'12).
Editors: A. Dovier and V. Santos Costa; pp. 98–108

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Test data generation (TDG) is the process of automatically generating *test inputs* for interesting *coverage criteria*. The standard approach to (glass-box) TDG is to perform a *symbolic execution* of the program [12, 4, 9, 15, 16, 6, 17], where the contents of variables are expressions rather than concrete values. Symbolic execution produces a system of constraints over the input variables consisting of the conditions to execute the different paths. The conjunction of these constraints represents the equivalence class of inputs that would take this path. In what follows, we use the term *test case* to refer to such constraints. The CLP-based approach to glass-box TDG [8] is based on the idea of translating the program to be tested (written in some imperative language) into an equivalent CLP program. The key idea is that test cases can be obtained by executing the CLP-translated program using the standard symbolic execution mechanism of CLP. When the original language includes features which are not supported, or do not have the same behavior, as in CLP, e.g., the use of a heap or primitives for concurrency different from those of CLP, specific *built-in* operations must be implemented entirely in CLP in order to handle them, see [8, 2]. Then, symbolic execution simply consists in executing the translated CLP program together with the predefined built-ins. In particular we leverage typical termination and coverage criteria for sequential programs (e.g., loop-k) to the concurrent setting and, besides, combine them with novel criteria to ensure interesting coverage of the concurrent behaviors (e.g., we can limit and control the number of task switches). We ensure *fairness* in the selection of objects whose tasks are being tested by applying a coverage criterion that limits task switches at the object level.

2 Symbolic Execution of Concurrent Objects

In this section, we summarize symbolic execution of concurrent objects, as presented in [2]. Essentially, the process is formalized in two steps: first the program is translated into a CLP program which contains some built-in predicates to handle the heap and concurrency primitives and, second, an implementation entirely in CLP of the built-ins is provided such that symbolic execution can be then performed by just relying on the standard symbolic execution engine of CLP.

2.1 CLP Translated Programs

The imperative language with concurrent objects we consider is basically the subset of the ABS language [11] which is relevant to define the TDG framework. A *program* consists of a set of classes C, where C is defined as $\mathtt{class}\ \mathsf{C}[(\overline{\mathsf{T}\ \mathsf{x}})]\{\overline{\mathsf{T}\ \mathsf{x}};\overline{\mathsf{M}}\}$. Each "$\mathsf{T_i}\ \mathsf{x_i}$" declares a field $\mathsf{x_i}$ of type $\mathsf{T_i}$, and each $\mathsf{M_i}$ is a method definition which takes the form $\mathsf{T}\ \mathsf{m}(\mathsf{T_1}\ \mathsf{x_1},\ldots,\mathsf{T_n}\ \mathsf{x_n})\{\overline{\mathsf{T}\ \mathsf{z}};\ \bar{\mathsf{s}}\}$, where T is the type of the return value; $\mathsf{x_1},\ldots,\mathsf{x_n}$ are the formal parameters and $\overline{\mathsf{T}\ \mathsf{z}}$ are local variables. Finally, $\bar{\mathsf{s}}$ is a sequence of instructions which adhere to the following grammar:

$$
\begin{array}{lcl}
\mathsf{s} & ::= & \mathsf{x} = \mathsf{rhs} \mid \mathbf{await}\ \mathsf{g} \mid \mathbf{return}\ \mathsf{e} \mid \mathbf{if}\ (\mathsf{b})\,\{\,\mathsf{s}\,\}\,[\,\mathbf{else}\,\{\,\mathsf{s}\,\}] \mid \mathbf{while}\ (\mathsf{b})\,\{\,\mathsf{s}\,\} \mid \mathbf{skip} \\
\mathsf{rhs} & ::= & \mathsf{e} \mid \mathbf{new}\ \mathsf{C}\,[(\bar{\mathsf{e}})] \mid \mathsf{e}\ !\ \mathsf{m}(\bar{\mathsf{e}}) \mid \mathsf{x}.\mathbf{get} \\
e & ::= & \mathbf{null} \mid \mathsf{this.f} \mid \mathsf{x} \mid \mathsf{n} \mid \mathsf{e} + \mathsf{e} \mid \mathsf{e} * \mathsf{e} \mid \mathsf{e} - \mathsf{e} \\
\mathsf{g} & ::= & \mathsf{b} \mid \mathsf{e}? \mid \mathsf{g} \wedge \mathsf{g}
\end{array}
$$

The central concept of the concurrency model is that of *concurrent object*. Conceptually, each object has a dedicated processor and encapsulates a *local heap* which is not accessible from outside this object, i.e., fields are always accessed using the *this* object, and any other object can only access such fields through method calls. Concurrent objects live in a *distributed*

environment with asynchronous and unordered communication by means of asynchronous method calls, denoted `o ! m(ē)`. Method calls may be seen as triggers of concurrent activity, spawning new tasks (so-called *processes*) in the called object. After asynchronously calling `x=o ! m(ē)`, the caller may proceed with its execution without blocking on the call. Here `x` is a *future variable* which allows synchronizing with the completion of task `m`. In particular, the instruction `await x?` allows checking whether `m` has finished. In this case, execution of the current task proceeds and `x` can be used for accessing the return value of `m` via the instruction `x.get`. Otherwise, the current task releases the processor to allow another available task to take it.

The translation of an ABS program into an equivalent CLP program has been subject of previous work [2]. An important feature of the translation is that the imperative program works on a global state which contains the set of created objects. This is simulated by representing the state using additional arguments of all predicates. Each object of the state includes the set of fields (which is not accessible outside the object) and its queue of pending tasks. Tasks can be of three types: *call* are asynchronous calls, *await* are tasks suspended due to an await condition and *get* are tasks suspended due to a blocking get instruction. Future variables become *ready(_)* when the corresponding task is completed. The syntax of the state is:

$$
\begin{array}{rcl}
\mathit{State} & ::= & [\,]\mid[(\mathit{Num},\mathit{Object})|\mathit{State}]\\
\mathit{Fut} & ::= & \mathit{ready}(\mathit{Data})|\mathit{Var}\\
Q & ::= & [\,]\mid[\mathit{Task}|Q]\\
\mathit{Fields} & ::= & [\,]\mid[\mathit{field}(f,\mathit{Data})|\mathit{Fields}]\\
\mathit{Object} & ::= & \mathit{object}(C,\mathit{Fields},Q)\\
\mathit{Task} & ::= & \mathit{call}(\mathit{Call})\mid\mathit{await}(\mathit{Call},\mathit{Call})\mid\mathit{get}(\mathit{Fut},\mathit{Var},\mathit{Call})
\end{array}
$$

Intuitively, for each class, the CLP translation represents all its methods (as well as the intermediate blocks within the methods for loops, conditionals, etc.) by means of predicates in the CLP program which adhere to the following grammar:

$$
\begin{array}{rcl}
\mathit{Clause} & ::= & \mathit{Pred}(\mathit{Args},\mathit{Args},S,S):-[\bar{G},]\bar{B}.\\
\mathit{Args} & ::= & [\,]\mid[\mathit{Data}^*|\mathit{Args}]\\
S & ::= & \mathit{Var}\\
G & ::= & \mathit{Num}^*\ \mathit{Op}_R\ \mathit{Num}^*\mid\mathit{Ref}_1^*\backslash{==}\mathit{Ref}_2^*\mid\mathit{Var}=\mathit{Data}\\
\mathit{Ref} & ::= & \mathit{null}\mid\mathit{Var}\\
B & ::= & \mathit{Var}\ \#{=}\ \mathit{Num}^*\ \mathit{Op}_A\ \mathit{Num}^*\mid\mathit{Pred}(\mathit{Args},\mathit{Args},S,S)\mid\mathit{Var}{=}\mathit{Data}\mid\\
 & & \mathsf{newObj}(C,\mathit{Ref}^*,S,S)\mid\mathsf{getField}(\mathit{Ref}^*,\mathit{FSig},\mathit{Var},S)\mid\mathsf{async}(\mathit{Ref}^*,\mathit{Call},S,S)\mid\\
 & & \mathsf{setField}(\mathit{Ref}^*,\mathit{FSig},\mathit{Var}^*,S,S)\mid\mathsf{await}(\mathit{Call},\mathit{Call},S,S)\mid\\
 & & \mathsf{get}(\mathit{Var},\mathit{Var},\mathit{Call},S,S)\mid\mathsf{return}(\mathit{Var}^*,\mathit{Var},S,S)\mid\mathsf{futAvail}(\mathit{Var},\mathit{Var})\\
\mathit{Call} & ::= & \mathit{Pred}(\mathit{Args},\mathit{Args})\\
\mathit{Pred} & ::= & \mathit{BlockN}\mid\mathit{MethodN}\\
\mathit{Data} & ::= & \mathit{Num}\mid\mathit{Ref}\mid\mathit{Bool}\\
\mathit{Op}_R & ::= & \#{>}\mid\#{<}\mid\#{>=}\mid\#{=<}\mid\#{=}\mid\#\backslash{=}\\
\mathit{Op}_A & ::= & +\mid-\mid*\mid/\mid\mathit{mod}
\end{array}
$$

Num is a number, *Var* is a Prolog variable and *Bool* can be either *true* or *false*. An asterisk on any element denotes that it can be either as defined by the grammar or a variable. Each clause receives as input a possibly empty list of parameters (1st argument) and a global

```
class A {
Int n; Int ft;  // fields
Int sumFacts(A ob) {
  Fut<Int> f; Int res=0;
  Int m = this.n;
  await this.ft >= 0;
  while (m > 0) {
    f =ob ! fact(this.ft, this);
    await f ?;
    Int a = f.get;
    res = res + a;
    this.ft = this.ft + 1;
    m = m - 1;
  }
  return res;
}
Int fact(Int k, A ob){
  Fut <Int> f; Int res = 1;
  if (k <= 0) res = 1;
  else {
      f = ob ! fact(k - 1,this);
      await f ?; res = f.get;
      res = k * res;
  }
  return res;
}
void setN(Int a) { this.n=a; }
void setFt(Int b) { this.ft=b; }
Unit set(Int a, Int b){
    this.setN(a); this.setFt(b);
}
```

```
'A.sumFacts'([This, Ob], [R], S1, S2) :-
  getField(This, fSig('A', n), M, S1),
  await(awguard0([This, Ob], _),
        cont0([This, Ob, M], [R], S1, S2).
awguard0([This, Ob], [R], S, S) :-
  getField(This, fSig('A', ft), Ft, S),
  geq([Ft, 0], [R]).
cont0([This, Ob, M], [R], S1, S2) :-
  while([This, Ob, M, 0], [R], S1, S2).
while([Args], [R], S1, S2) :-
  M #=< 0,
  return([Res], [R], S1, S2).
while([Args], [R], S1, S2) :-
  M #> 0,
  getField(This, fSig('A', ft), Ft, S1),
  async(Ob, 'A.fact'([Ob, Ft, This], [F]), S1, S3),
  await(awguard1([F], _),
        cont1([Args, F], [R]), S3, S2).
awguard1([F], [V]) :- futAvail(F, V).
cont1([Args, F], [R], S1, S2) :-
  get(F, A, cont2([Args, A], [R]), S1, S2).
cont2([Args, A], [R], S1, S2) :-
  Res1 #= Res + A,
  getField(This, fSig('A', ft), Ft, S1),
  Ft1 #= Ft + 1,
  setField(This, fSig('A', ft), Ft1, S1, S3),
  M1 #= M - 1, while([Args1], [R], S3, S2).
geq([Ft, Z], [R]) :-
  Ft #< Z, R = false.
geq([Ft, Z], [R]) :-
  Ft #>= Z, R = true.
```

Figure 1 ABS running example (left). CLP translation of sumFacts (right).

state (3rd argument), and returns an output (2nd argument) and a final global state (4th argument). The body of a clause may include a sequence of guards followed by a sequence of instructions, including: arithmetic operations, calls to other predicates, builtins to handle the concurrency (namely await, get, futAvail and return) and builtins to operate on the heap [8]. The latter includes the builtin newObj(C, R, S_1, S_2) which creates a new object of class C in state S_1 and returns its assigned reference R and the updated state S_2; getField($R, FSig, V, S$) which retrieves in variable V the value of field $FSig$ of the object referenced by R in the state S and setField($R, FSig, V, S_1, S_2$) which sets the field $FSig$ of the object referenced by R in S_1 to V and returns S_2.

▶ **Example 1.** Fig. 1 (left) shows the implementation of a class A, which contains two integer fields and five methods. Method sumFacts computes $\sum_{k=ft}^{ft+(m-1)} k!$ by asynchronously invoking fact on object ob. The `await` instruction before entering the loop allows releasing the processor if ft is negative. Once it takes a non-negative value, the task can resume its

```
(a)async(Ref,Call,S1,S2) :- addTask(S1,Ref,call(Call),S2).
(b)await(Cond,Cont,S1,S3) :-
     Cond =..[_,[This|_],[Ret]], buildCall(Cond,S1,S2,CondCall), CondCall,
     (Ret = false -> addTask(S1,This,await(Cond,Cont),S2),
                     switchContext(S2,S3)
                     ; buildCall(Cont,S1,S3,ContCall), ContCall).
(c)get (FV,V,Cont,S1,S3) :- Cont =..[_,[This|_],_],
     (var(FV) -> addTask(S1,This,get(FV,V,Cont),S2),
                 switchContext(S2,S3)
                 ; FV = ready(V), buildCall(Cont,S1,S3,ContCall), ContCall).
(d)return([Ret],[ready(Ret)],S1,S2) :- switchContext(S1,S2).
(e)futAvail(FV,false) :- var(FV), !.          futAvail(ready(_),true).
-----------------------------------------------------------------------
(f)addTask(S1,Ref,T,S2) :- getCell(S1,Ref,object(C,Fs,Q1)),
     insert(Q1,T,Q2), setCell(S1,Ref,object(C,Fs,Q2),S2).
(g)switchContext(S1,S3) :- S1 = [(Ref,_)|_], firstToLast(S1,S2), switchContext_(S2,S3,Ref).
(h)switchContext_(S,S,Ref1) :- S = [(Ref2,object(_,_,[ ]))|_],Ref1 == Ref2.
(i)switchContext_(S1,S3,Ref1) :- \+ (S1 = [(Ref2,object(_,_,[ ])|_], Ref1 == Ref2),
     extractFirst(S1,Task,S2,Answer),
     runTaskOrSwitch(Answer,Task,Ref1,S3,S2).
(j)runTaskOrSwitch(true,Task,_Ref,S1,S3) :- runTask(Task,S1,S3).
(k)runTaskOrSwitch(false,_Task,Ref,S1,S3) :- firstToLast(S1,S2), switchContext_(S2,S3,Ref).
(l)runTask(call(ShortCall),S1,S2) :- buildCall(ShortCall,S1,S2,Call), Call.
(m)runTask(await(Cond,Cont),S1,S2) :- await(Cond,Cont,S1,S2).
(n)runTask(get(FV,V,Cont),S1,S2) :- get (FV,V,Cont,S1,S2).
(o)buildCall(ShortCall,S1,S2,Call) :- ShortCall =..[RN,In,Out], Call =..[RN,In,Out,S1,S2].
```

Figure 2 Implementation of Concurrency Builtins.

execution and enter the loop. Observe that an asynchronous call from sumFacts as follows f = ob ! fact(3, this); will add the task fact(3, this) to the queue of ob. When this task starts to execute it will add the task fact(2, ob) on the object this, which in turn will add the call fact(1, this) on ob and so on, in such a way that the factorial is computed in a distributed way between the two objects. Note that the calls are synchronized on future variables. This means that until the recursive call fact(1, this) is not completed the other tasks are suspended on their **await** conditions. Fig. 1 (right) shows the CLP translation of method sumFacts. We use $\overline{Args}$ and $\overline{Args_1}$ to abbreviate, resp., $This, Ob, M, Res$ and $This, Ob, M_1, Res_1$. Methods and intermediate blocks (like $cont_0$) are uniformly represented by means of predicates and are not distinguishable in the translated program. The list of input arguments of all rules includes: the *this* reference, the list of input parameters of the corresponding ABS method, and for intermediate blocks, their local variables. The list of output argument is always a unitary list with the return value. Loops in the source program are transformed into guarded rules (e.g., rule *while*). An important point to note is that, for all **await** and **get** statements, we introduce a *continuation* predicate (like $cont_i$, $0{\le}i{\le}2$) which allows us to suspend the current task (if needed) and then resume its execution at this precise point.

2.2 Implementation of Concurrency Builtins

Fig. 2 shows the CLP implementation of the builtins to handle concurrency of [2]. Boxes are used to indicate code that needs to be changed in order to define the TDG framework. *Asynchronous calls* are handled by predicate ⓐ which adds the asynchronous call Call to the queue of tasks of the receiver object Ref producing the updated state S_2. The call to addTask/4 searches the state for the object pointed to by reference Ref by means of getCell/3 [8], adds the task to its queue (using insert/3) and updates the state with the updated object (using setCell/3 [8]).

The fact that objects do not share memory ensures that their execution states are not affected by how distribution (or parallelism) is realized. Namely, *distribution* is implemented as follows: each object executes its scheduled task as far as possible and, when a task finishes or gets blocked, simulation proceeds circularly with the *next* object in the state. In contrast, *concurrency* occurs at the level of objects in the sense that tasks in the object queue are executed concurrently. The concurrency model of our language only specifies that the execution of the current task must proceed until a call to ⓑ, ⓒ, or ⓓ is found. The scheduling policy which decides which task executes next (among those ready for execution) is left unspecified.

Rule ⓖ is used when the execution of the current task can no longer proceed (hence it *releases* the processor). The implementation gives the turn of execution to the first task (according to the scheduling policy) of the following object (the next one in the state). This is implemented by always keeping the current object in the head of the state, and moving it to the last position when its current task finishes or gets blocked. If the current object has some pending task in its queue ⓙ, predicate extractFirst/4 bounds Answer to true. Otherwise, it is bound to false and the following object is tried ⓚ. The execution of the whole application finishes when there is no pending task in any object ⓗ.

Await ⓑ first checks its condition Cond by means of the meta-call CondCall. If the condition holds (Ret gets instantiated to true), a meta-call to the continuation Cont is made (meta-call ContCall). Otherwise (Ret is false), an await task is added to the queue of the current object and we switch context. Predicate ⓞ builds a *full* call from a call without states and two states. The evaluation of await conditions can involve return tests on future variables. This is represented in our CLP programs by a call to ⓔ. We use the special term ready(V) to know whether the execution has finished. Rule ⓔ checks whether the future variable is a CLP variable or is instantiated to ready(_) and returns, resp., false or true. When a method finishes its execution, we reach a return statement ⓓ which instantiates the future variable V associated to the current task to ready(V). This allows that, if the task that requested the execution of this one was blocked awaiting on this future variable, it can proceed its execution when it is re-scheduled. Namely, ⓒ first checks if the task can resume execution because its future variable has become instantiated. In such case, the continuation of this get is executed (meta-call ContCall). Otherwise, the current task is added to the queue and context is switched.

3 From Symbolic Execution to TDG

Having a CLP symbolic execution engine for concurrent objects is an important piece when defining the CLP-based TDG framework, but there are still many other missing pieces. Firstly, we need to define a TDG engine which incorporates relevant *coverage criteria* (CC). An important problem in symbolic execution is that, since the input data is unknown, the execution tree to be traversed is in general infinite. Hence it is required to integrate a

termination criterion which guarantees that the length of the paths traversed remains finite while at the same time an interesting set of test cases is generated, i.e., certain code *coverage* is achieved. The challenge when developing the TDG framework is integrating CC on the CLP-translated programs which achieve the desired degree of coverage on the original ABS.

3.1 Task-Level Coverage and Termination Criteria

Given a task executing on an object, we aim at ensuring its local termination by leveraging existing CC developed in the sequential setting to the context of concurrent objects. We focus on the loop-count criteria [10] which limits the number of times we iterate on loops to a threshold $\mathsf{K_l}$ (other existing criteria would pose similar problems and solutions). If we focus on a single task, this task-level CC can be integrated, as in the sequential CLP-based approach [8], by keeping track of the *ancestor sequences* for every call unfolded in the task. The main idea is that loop iterations are detected because recursive calls are performed. However, in order to distinguish a recursive call from an independent call to the same (recursive) predicate, we need to track the ancestors of each call. This can be implemented by using a global ancestor stack for the task such that each time an atom A is unfolded using a rule $H{:-}B_1, \ldots, B_n$, the predicate name of A (F/N where N is the arity) is pushed on the ancestor stack. Additionally, a $'\$\mathsf{pop}\$'$ mark is added to the new goal $B_1, \ldots, B_n, '\$\mathsf{pop}\$'$ to delimit the scope of the predecessors of F/N such that, once those atoms are evaluated, we find the mark $'\$\mathsf{pop}\$'$ and can remove F/N from the ancestor stack. This way, the ancestor stack, at each stage of the computation, contains the ancestors of the next atom to be selected for resolution.

Due to the coexistence of multiple tasks in the concurrent setting, the problem is more complicated and we need to construct the list of ancestor predicates for each available task and besides, as tasks can suspend their execution, be able to recover this information when they resume. Thus, the new syntax for tasks is:

$$Task ::= call(Call) \mid await(Call, Call, AncSt) \mid get(Fut, Var, Call, AncSt)$$

where *AncSt* is a list of elements of the form F/N. Additionally, we introduce atoms of the form taskSuspendMark to indicate to the TDG engine that a task is going to suspend and hence its ancestor stack needs to be stored. This is achieved by replacing the framed code in ⓑ and ⓒ in Fig. 2, resp., by:

```
(await)  taskSuspendMark(AncSt), addTask(S1,This,await(Cond,Cont,AncSt),S2),
(get)    taskSuspendMark(AncSt), addTask(S1,This,get(FV,V,Cont,AncSt),S2),
```

3.2 Task-Switching Coverage and Termination Criteria

Applying the task-level CC to all tasks does not guarantee termination. This is because we can switch from one task to another an infinite number of times. For example, consider the symbolic execution of $\mathsf{ob_1}$! $\mathsf{fact(n, ob_2)}$. We circularly switch from object $\mathsf{ob_1}$ to object $\mathsf{ob_2}$ an infinite number of times because each asynchronous call in one object adds another call on the other object (see Ex. 1). This is not detected by the task-level CC because each method invocation is a new task that has no ancestors. The same problem can happen even with a single object, e.g., in method sumFacts when executing `await` (ft >= 0), there is an infinite branch in the evaluation tree, corresponding to the case ft < 0 which is re-tried forever.

The number of task switches can be limited by simply allowing $\mathsf{K_s}$ executions of predicate (g) (Fig. 2). However, it might happen that, due to excessive task switching in certain objects, others are not properly tested (i.e., their tasks exercised) because the *global* number of allowed task switches has been exceeded. For example, suppose that we add the instructions $\mathsf{B}\ \mathsf{ob_2} = \mathsf{new}\ \mathsf{B}();\ \mathsf{ob_2}\ !\ \mathsf{q}();$ before the return in method sumFacts, where B is a class that implements method q but whose code is not relevant. Then, as the evaluation tree for the *while* loop generates an infinite number of task switches, the evaluation of the call $\mathsf{ob_2}\ !\ \mathsf{m}();$ is not reached. In order to have fairness in the process and guarantee proper coverage from the concurrency point of view, we propose to limit the number of task switches *per* object (i.e., per concurrency unit). For this purpose, objects are now of the form:

$$Object ::= object(C, Fields, Q, NT)$$

where NT is the number of tasks which have been extracted from its queue. Besides, similarly to the treatment of the task-level CC, we introduce special markers by replacing the framed code of rule (j) by:

$\mathsf{taskStartMark(S_1,Task),\ incNumTasks(S_1,S_2),\ runTask(Task,S_2,S_3),}$

which allows the TDG engine to realize that there has been a task switching and hence the limit needs to be checked. Predicate incNumTasks adds 1 to the number of task switches NT of the first object in $\mathsf{S_1}$, i.e., the object selected by extractFirst.

3.3 A CLP-based TDG Engine for Concurrent Objects

Fig. 3 presents a TDG engine, named unfold, which receives as input parameters the method call to be tested Root (last parameter), a list of atoms to be evaluated (initially Root), two constants $\mathsf{K_l}$ and $\mathsf{K_s}$ to limit, resp., the number of loop iterations and the number of task switches per object and the ancestor stack AncSt of the current task (initially empty). Rule (1) corresponds to the end of a successful derivation, it stores (using storeTestCase/1) the computed test case, namely the initial call Root instantiated with the bindings for the input/output parameters and the states, and the constraint store. The task-level CC is handled in rules (2), (4) and (7). Essentially, rule (7) checks if the number of iterations has not been exceeded (checkIter traverses the list of ancestors AncSt) and, if not, it adds the '\$pop\$' mark as explained in Sec. 3.1. Later, when this mark is reached in rule (2), the top of the stack is popped. In rule (4), when a task is suspending, the argument of taskSuspendMark gets unified with the current stack (fourth argument of unfold) to be later recovered, and execution proceeds. The treatment of the task switching criteria is captured by rule (3) which detects the mark introduced in Sec. 3.2 and invokes checkNtasks to check if the number of task switches in the current object exceeds $\mathsf{K_s}$. If the task to be started now is an await or get, predicate $\mathsf{recoverAncStack(Task, AncSt_p)}$ recovers its ancestor stack; if it is a call, initializes $\mathsf{AncSt_p}$ to empty. As we have seen in Sec. 3.2, after finding such mark, there is a call to incNumTasks.

The two remaining rules treat the builtins and the constraints. In particular, rule (6) handles the ABS builtin predicates in Fig. 2 which make callbacks to the program. They are treated differently from rule (7) because the loop-count criteria does not have to be applied on them. Rule (5) covers external predicates, i.e., constraints and the auxiliary predicates in Fig. 2. The difference w.r.t. rule (6) is that here we execute them (making call(A)) since the

```
① unfold([ ], _Kl, _Ks, _AncSt, Root) :- storeTestCase(Root).
② unfold(['$pop$'|R], Kl, Ks, [_|AncSt], Root) :- !, unfold(R, Kl, Ks, AncSt, Root).
③ unfold([taskStartMark(S, Task)|R], Kl, Ks, AncSt, Root) :- !,
      checkNtasks(S, Ks),
      recoverAncStack(Task, AncStp),
      unfold(R, Kl, Ks, AncStp, Root).
④ unfold([taskSuspendMark(AncSt)|R], Kl, Ks, AncSt, Rt) :- !,
      unfold(R, Kl, Ks, AncSt, Rt).
⑤ unfold([A|R], Kl, Ks, AncSt, Rt) :- isExternal(A), !,
      call(A), unfold(R, Kl, Ks, AncSt, Rt).
⑥ unfold([A|R], Kl, Ks, AncSt, Root) :- functor(A, F, Ar), isAbsBuiltin(F/Ar), !,
      clause(A, B), append(B, R, NG),
      unfold(NG, Kl, Ks, AncSt, Root).
⑦ unfold([A|R], Kl, Ks, AncSt, Root) :- functor(A, F, Ar), checkIter(AncSt, F, Ar, Kl),
      clause(A, B), append(B, ['$pop$'|R], NG),
      unfold(NG, Kl, Ks, [F/Ar|AncSt], Root).
checkIter([ ], _, _, K) :- K > 0.
checkIter([F/Ar|As], F, Ar, K) :- !, K > 1,K1 is K−1,checkIter(As, F, Ar, K1).
checkIter([_|As], F, Ar, K) : −checkIter(As, F, Ar, K).
checkNtasks([(_, object(_, _, _, NTs))|_], K) :- NTs < K.
incNumTasks(H, Hp) :- H = [(Ref, object(CN, Fs, Q, K))|RH],
      Kp is K + 1,
      Hp = [(Ref, object(CN, Fs, Q, Kp))|RH].
recoverAncStack(await(_, _, AncSt), AncSt).
recoverAncStack(get(_, _, _, AncSt), AncSt).
recoverAncStack(call(_), [ ]).
```

Figure 3 Implementation of TDG engine.

predicate is not part of the CLP program. The execution of unfold($[Root], K_l, K_s, [\,], Root$), where $Root{=}'C.m'(In, Out, S_1, S_2)$, computes an *incomplete* derivation tree for method m of class C, the different branches of the tree are obtained by backtracking. Successful branches are obtained in ① and incomplete branches by ③ and ⑦ when the termination tests stop the derivation. $\langle G \diamond \theta \rangle$ denotes a state with goal G and computed *constraint store* θ. Then, given the set of branches (derivations) for the derivation tree $\mathcal{T}$ associated to $\langle$unfold$([Root], K_l, K_s, [\,], Root) \diamond \{\}\rangle$, where $Root{=}'C.m'(In, Out, S_1, S_2)$, the *test cases* for m are the set of constraint stores θ associated to each output state $\langle \epsilon \diamond \theta \rangle$ of a successful branch in $\mathcal{T}$ (computed in ①), where ϵ is an empty goal.

▶ **Example 2.** Let us obtain the test cases for method sumFacts with K_l=1, K_s=2. The execution of unfold($[Root], 1, 2, [\,], Root$) with $Root{=}'A.sumFacts'(In, Out, S_1, S_2)$, first applies rule ⑦ which bounds In to the list $[This, Ob]$ and Out to $[R]$. Predicate checkIter succeeds (the ancestor stack is empty). The first instruction in $'A.sumFacts'$ (see Fig. 1 right) is a getField which bounds S_1 to $[(Id_1, object(A, [field(n, _)|_], [\,], 0))|_]$. Afterward, we find a call to await which is handled by rule ⑥, which in turns executes the rule ⓑ of await in Fig. 2. Here, the condition $awguard_0$ in the await adds to the list of fields of S_1 the literal $field(ft, _)$. The execution of the guard returns *false* and addTask inserts the task in the

queue of object Id_1 with the annotation taskSuspendMark. Next, switchContext will take the task and annotate it with taskStartMark. Now, rule (3) is applied and checkNtasks fails since the number of task switches (incremented by switchContex) for Id_1 is greater than 2. By backtracking, we generate the branch in which $Ft \geq 0$ which leads to executing $cont_0$ and, after unfolding the first clause of predicate *while*, the first test case is computed. In this test case, $S_1/(Id_1, object('A', [field(n, N_1), field(ft, Ft_1)|_], [\,], 0))|_]$, $R/0$, the constraint store contains $N_1{\leq}0$, $Ft_1{\geq}0$, and $S_1 \equiv S_2$. Again, by backtracking the second clause for *while* is tried. At this point, the ancestor stack is $[cont_0/4, sumFacts/4]$. The async call introduces a new object $(Id_2, object(A, _F, [call('A.fact')], 0))$ in the queue of Id_1. The execution of the await spawns the task $'A.fact'$ which returns 1. Thus, the second test case is computed $S_1/[(Id_1, object('A', [field(n, 1), field(ft,\ 0)|_],\ [\,], 0)), (Id_2, object('A', _, [\,], 0))|_]$, $S_2/[(Id_1, object('A', [field(n, 1), field(ft, 1)|_], [\ ], 1)),\quad (Id_2, object('A', _, [\ ], 1))|_]$ and $R/1$. Note that the number of task switches for both objects Id_1 and Id_2 changes from 0 in the initial state S_1 to 1 in the final state S_2. No more solutions are computed since the execution of fact is stopped after two task switches coming from the await in its body and checkIter fails when evaluating again predicate *while* as the stack of ancestors contains already $[\ldots, while, \ldots]$. Therefore, the two criteria are needed to ensure termination: $\mathsf{K_s}$ to limit the number of task switches between the two objects and $\mathsf{K_l}$ to limit the number of loop iterations in the *while* loop.

4 Conclusions, Related and Future Work

We have presented a novel approach to automate test case generation for concurrent objects, entirely implemented in CLP, which ensures *completeness* of the test cases w.r.t. several interesting criteria. The coverage criteria prune the tree in several dimensions: (1) limiting the number of iterations of loops at the level of tasks, (2) limiting the length of the queue of tasks of the objects such that the number of task interleavings that are tried remains finite, (3) limiting the number of task switches allowed in each concurrency unit. The technique is complete on the orderings in which tasks can be selected for execution, even allowing that different policies are applied on different objects. We argue that our CLP-based framework is at the same time practical and highly flexible and constitutes thus a promising approach to TDG of concurrent languages.

In future work, we plan to study the application of our framework to a thread-based concurrency model like Java [13, 5, 18]. The main conceptual difference with the actor-based model is that task scheduling is preemptive. Therefore, at any point, the current task can be suspended and interleaved with another one. Specific coverage criteria should be defined to control such interleavings in a way that the size of the symbolic execution tree remains reasonable and at the same time interesting test cases can be obtained. It seems that the combination with dynamic analysis is useful for this purpose [3]. We also want to investigate the application of further coverage criteria [14, 18, 7] to detect bugs related, for instance, to happen-before relations.

References

1 G.A. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, MA, 1986.

2 E. Albert, P. Arenas, and M. Gómez-Zamalloa. Symbolic Execution of Concurrent Objects in CLP. In *Practical Aspects of Declarative Languages (PADL'12)*, volume 7149 of *LNCS*, pages 123–137. Springer, January 2012.

3 Jun Chen and Steve MacDonald. Towards a better Collaboration of Static and Dynamic Analyses for Testing Concurrent Programs. In *Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging (PADTAD'08)*, page 8. ACM, 2008.
4 L. A. Clarke. A System to Generate Test Data and Symbolically Execute Programs. *IEEE Transactions on Software Engineering*, 2(3):215–222, 1976.
5 O. Edelstein, E. Farchi, E. Goldin, Y. Nir, Ratsaby G, and S. Ur. Framework for Testing Multi-Threaded Java Programs. *Concurrency and Computation: Practice and Experience*, 15(3-5):485–499, 2003.
6 Christian Engel and Reiner Hähnle. Generating Unit Tests from Formal Proofs. In *Tests and Proofs, First International Conference (TAP'07)*, volume 4454 of *LNCS*, pages 169–188. Springer, 2007.
7 M. Factor, E. Farchi, and Y. Malka Y. Lichtenstein. Testing Concurrent Programs: A Formal Evaluation of Coverage Criteria. In *Seventh Israeli Conference on Computer-Based Systems and Software Engineering (ICCSSE '96)*, pages 119–126, 1996.
8 M. Gómez-Zamalloa, E. Albert, and G. Puebla. Test Case Generation for Object-Oriented Imperative Languages in CLP. *Theory and Practice of Logic Programming, ICLP'10 Special Issue*, 10 (4–6), 2010.
9 A. Gotlieb, B. Botella, and M. Rueher. A CLP Framework for Computing Structural Test Data. In *Computational Logic*, pages 399–413, 2000.
10 W.E. Howden. Symbolic Testing and the DISSECT Symbolic Evaluation System. *IEEE Transactions on Software Engineering*, 3(4):266–278, 1977.
11 E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A Core Language for Abstract Behavioral Specification. In *Formal Methods for Components and Objects (FMCO 2010, Revised Papers)*, volume 6957 of *LNCS*, pages 142–164. Springer, 2012.
12 J. C. King. Symbolic Execution and Program Testing. *Communications of the ACM*, 19(7):385–394, 1976.
13 B. Long, D. Hoffman, and P. A. Strooper. Tool Support for Testing Concurrent Java Components. *IEEE Trans. Software Eng.*, 29(6):555–566, 2003.
14 Shan Lu, Weihang Jiang, and Yuanyuan Zhou. A Study of Interleaving Coverage Criteria. In *ESEC/SIGSOFT FSE*, pages 533–536. ACM, 2007.
15 C. Meudec. ATGen: Automatic Test Data Generation using Constraint Logic Programming and Symbolic Execution. *Softw. Test., Verif. Reliab.*, 11(2):81–96, 2001.
16 R. A. Müller, C. Lembeck, and H. Kuchen. A Symbolic Java Virtual Machine for Test Case Generation. In *IASTED Conf. on Software Engineering*. IASTED/ACTA Press, 2004.
17 T. Schrijvers, F. Degrave, and W. Vanhoof. Towards a Framework for Constraint-Based Test Case Generation. In *International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'09)*, volume 6037 of *LNCS*, pages 128–142. Springer, 2010.
18 Juichi Takahashi, Hideharu Kojima, and Zengo Furukawa. Coverage Based Testing for Concurrent Software. In *IEEE International Conference on Distributed Computing Systems Workshops (ICDCS 2008)*, pages 533–538. IEEE Computer Society, 2008.

Visualization of CHR through Source-to-Source Transformation

Slim Abdennadher and Nada Sharaf

Computer Science and Engineering Department
The German University in Cairo
slim.abdennadher,nada.hamed@guc.edu.eg

Abstract

In this paper, we propose an extension of Constraint Handling Rules (CHR) with different visualization features. One feature is to visualize the execution of rules applied on a list of constraints. The second feature is to represent some of the CHR constraints as objects and visualize the effect of CHR rules on them. To avoid changing the compiler, our implementation is based on source-to-source transformation.

1998 ACM Subject Classification I.2.2 Automatic Programming, D.3.2 Language Classifications

Keywords and phrases Source-to-Source Transformation, Constraint Handling Rules, Visualization

Digital Object Identifier 10.4230/LIPIcs.ICLP.2012.109

1 Introduction

Constraint Handling Rules (CHR) [4] is a high-level language especially designed for writing constraint solvers. CHR is essentially a committed-choice language consisting of multi-headed rules that transform constraints into simpler ones until they are solved. Over the last decade, CHR has matured to a powerful general purpose language by adding several features to it. There are quite a number of implementations of CHR. The most prominent ones are in Prolog.

So far, debugging the CHR code and tracing its run in Prolog was not visualized, thus less understandable and necessitating more concentration from the CHR programmer. Additionally, it offers only a small degree of freedom for the programmer to move backwards in the trace thread. In previous work [1], a tool called VisualCHR was developed to support the development of constraint solvers written in JCHR; an implementation of CHR in Java. To implement VisualCHR, the compiler of JCHR [9] has been modified to add the feature of visualization. However, such visualization feature was not available for Prolog versions.

The aim of the paper is to introduce an approach to add visualization features for CHR (implemented in SWI-Prolog) without changing the CHR compiler. Our approach uses source-to-source transformation by providing an implementation based on the core CHR language.

The paper will discuss features to visualize the execution of CHR rules as well as the graphical representation of objects and the effect of applying rules on them without changing the compiler.

The first transformer manipulates the input programs in order to be able to visualize the execution of their rules. The provided visual tracer shows the used constraints at each step and their effect on the existing constraints. The second transformer provides the user with the possibility to choose the type of objects that represent some CHR constraints and to visualize the objects after executing the rules on the constraints.

© Slim Abdennadher and Nada Sharaf;
licensed under Creative Commons License ND
Technical Communications of the 28th International Conference on Logic Programming (ICLP'12).
Editors: A. Dovier and V. Santos Costa; pp. 109–118

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

The paper is organized as follows. In Section 2, we introduce briefly the CHR language. In Section 3, some comparisons to related work will be discussed. In Section 4, the source-to-source transformation used is presented. Section 5 illustrates the visualization of the execution of the different rules and the visualization of some of the CHR constraints. Finally, we conclude with a summary and directions for future work.

2 Constraint Handling Rules by Example

A CHR rule consists of a head and a body and may contain a guard. CHR allows multiple heads and a conjunction of zero or more atoms in the guard as well as in the body of the rule. In the following, the syntax and the semantics of CHR are introduced by example.

Following is an example of a typical CHR program defining the partial order relation $\leq$ (`leq/2`). `leq(A,B)` holds if variable `A` is less than or equal to variable `B`.

```
:- use_module(library(chr)).
:- chr_constraint leq/2.

reflexivity  @ leq(X,X) <=> true.
antisymmetry @ leq(X,Y) , leq(Y,X) <=> X=Y.
idempotence  @ leq(X,Y)\ leq(X,Y) <=> true.
transitivity @ leq(X,Y) , leq(Y,Z) ==> leq(X,Z).
```

The first rule, which is called `reflexivity` (rule names are optional), is a single-headed simplification rule. It removes constraints of the form `leq(A,A)` from the constraint store. The second rule, `antisymmetry`, is a simplification rule with two heads. It replaces two symmetric `leq` constraints by an equality constraint. The equality constraint is usually handled by the host language; in this case Prolog does the unification.

Simplification rules correspond to logical equivalence, as the syntax suggests. The third rule is a simpagation rule which removes redundant copies of the same constraint. Such rules are often needed because of the multi-set semantics of CHR. Finally, the last rule (`transitivity`) is a propagation rule that adds redundant constraints. Propagation rules correspond to logical implication.

Execution proceeds by exhaustively applying the rules to a given input query. For example, given the query `leq(A,B), leq(B,C), leq(C,A)` the transitivity rule adds `leq(A,C)`. Then, by applying the `antisymmetry` rule, `leq(A,C)` and `leq(C,A)` are removed and replaced by `A=C`. Now the `antisymmetry` rule becomes applicable on the first two constraints of the original query. Now all CHR constraints are eliminated so no further rules can be applied, thus the answer `A=C, A=B` is returned.

3 Related Work

Due to the importance of source-to-source transformation and the advantages that it could bring around, various attempts have been made to incorporate and use such techniques with CHR. This section briefly mentions some of the work related to using source-to-source transformation with CHR.

In [6], a description of a source-to-source transformation technique for CHR was presented. Through the used technique, it is argued that it is rather easy to add source-to-source transformation to CHR. As discussed in [6], the input CHR program is represented in a "relational normal form" using some special CHR constraints that encode the different parts

of a CHR rule such as `head/4, guard/2, body/2, pragma/2` and `constraint/1`. The transformation is then done to this form. However, a new built-in predicate should be introduced to the CHR runtime system in order to register handlers as transformers, the intended order for application and the options that could provide additional control over the expansion [6]. In [6], some applications are shown such as bootstrapping the CHR compiler. Another example extends CHR by having probabilistic choice of rules. [7] provides more details about probabilistic constraint handling rules. The whole source-to-source transformation program for probabilistic CHR has a few rules and could fit into only one page [7]. Nevertheless, the runtime system had to be extended with rules for conflict resolution [6].

Another approach to program transformation was presented in [12], namely, unfolding. In order to do this, the syntax of CHR programs has to be modified. The rules have to be annotated to be in a specific format and to have a local token store. In other words, the operational semantics ω_t [3] are replaced with some modified semantics ω_t' [12]. In general, unfolding replaces a procedure call by its definition. According to [12], the unfolding process that is performed replaces the conjunction of constraints, S (considered to be the procedure call), in the body of a rule, `r`, with the body of another rule, `v`, given that the head of the rule `v` matches S.

In [5], the specialization of rules with respect to the goal is considered which is very interesting as it optimizes the program for input values of the goal. Since rules are specialized (by modifying some of their parts), and these new rules are added, this was also considered as a step towards transformation of CHR programs.

In [13], an implementation for aggregate functions in CHR was introduced. The implementation also used source-to-source transformation. However, in order to extend the current CHR systems with aggregates, a number of low-level compiler directives had to be added to the CHR system. In [11], more details about the implementation are offered. Meta CHR rules were used. Such rules rewrite the CHR rules of a specific program. A meta rule could be applied, if a single rule's head (in the original program) matched the meta occurrences of the (meta) rule. When the meta rule fires, the conjuncts of the program rule's head that caused the rule to fire are removed. The body of the meta rule could add new conjuncts to the program rule's guard or head. It could also add new rules to the program [11].

Finally, in [8], CHR^{rp}, which adds user-definable priorities to the different rules, was introduced. However, for CHR^{rp}, the priority semantics ω_p is introduced. Source-to-source transformation is used to translate CHR^{rp} into CHR.

The rest of this section goes through the differences and the advantages of the proposed system. First of all, our approach does not require any changes to the compiler or the CHR runtime system. The transformer and the output program are normal CHR programs that do not require any additions or changes to work. At the same time, the proposed system saves the user from having to translate the program into or from the "relational normal form". This translation is done automatically at the beginning through the Java application that parses the input file and finally through the transformer itself as it writes the result into a new file. Moreover, the order of the rules in the new program is the same as their order in the original program thus eliminating the rule ordering problem faced in [6]. The user is also able to control where new rules are added to the output program.

4 Source-to-Source Transformation for Visualization

In order to be able to have the required visual tracers, the original programs need to be modified to be able to interact with the tracers and produce the needed output. The

```
:- use_module(library(chr)).
handler leq.
:-chr_constraint leq/2.
reflexivity @ leq(X,X) <=> true.
idempotence @ leq(X,X)\leq(X,X) <=> true.
antisymmetry @ leq(X,Y), leq(Y,X) <=>  X = Y.
transitivity @ leq(X,Y), leq(Y,Z) ==>  leq(X,Z).
```

(a) A Sample input file.

```
head(reflexivity ,'leq(X,X)',remove)
body(reflexivity ,'true')
head(idempotence ,' leq(X,X)',keep)
head(idempotence ,'leq(X,X)',remove)
body(idempotence ,'true')
head(antisymmetry ,'leq(X,Y)',remove)
head(antisymmetry ,'leq(Y,X)',remove)
body(antisymmetry ,'X = Y')
```

(b) Some of the extracted information.

Figure 1 Sample of the information extracted through the Java application.

advantage of source-to-source transformation, in this context, is that it is able to manipulate input programs to change or add to their behavior the required functionalities without the need to do this modification manually. The proposed transformers use some of the central ideas introduced in [6]. In [6], some CHR constraints, that encode the different constituents of CHR rules, were introduced. The difference is that in the proposed transformers, instead of `head/4`, `head/3` is used since the information about the constraint's identifier is not needed. Using such CHR constraints, any CHR rule was transformed into "relational normal form".

According to [6], the modified behavior is also represented in terms of the introduced five CHR constraints. The proposed transformers, which are CHR handlers, operate on some specific CHR constraints including some of the constraints introduced in [6]. Consequently, the information regarding the different parts of the CHR rules needs to be extracted and represented in the needed format. The different parts of the rules are represented using CHR constraints. Some of them were introduced in [6] such as `guard/2 and body/2`. As mentioned before, `head/3` is used as well. For example, `head(reflexivity,'leq(X,X)',remove)`, represents the fact that the head of the rule named `reflexivity` contains `leq(X,X)` and that when executing the rule, this constraint is removed from the constraint store since `reflexivity` is a simplification rule.

Since some CHR solvers contain CHR rules as well as Prolog facts, the transformers use a new constraint named `facts/1`. This constraint is used to copy such Prolog facts into the output program. The proposed transformers write the modified program into a new file so that there is no need to alter the compiler or the CHR runtime system in anyway.

In addition to the introduced CHR constraints, there are other ones that are mainly used to write the modified rules and the facts to the output file. The proposed transformers operate on these constraints. Each of the transformers has two main functionalities. Firstly, it adds to the rules the required information and functionalities required to produce the needed visual tracer and interact with it. Secondly, it writes the new program into a file. This comes in handy since through using this arrangement, the original program does not have to be transformed every time. Instead it is transformed at the beginning and the new produced program could be used afterwards.

The question now may be how to represent the input program in the needed format so that the transformer could act on it. One possibility is to do this manually. In this case, the user would have to translate every part of the CHR program to produce the needed CHR constraints and then run the transformer on these extracted constraints. As simple as doing the translation might seem to be, this job could be very long and tedious. Therefore, a Java application is provided to parse the input file and do this translation automatically.

As shown in Figure 2, the Java application performs two tasks. First of all, it parses the input file in order to extract the needed information. In addition, it represents this information in the needed format using the new CHR constraints. Figure 1 shows some of

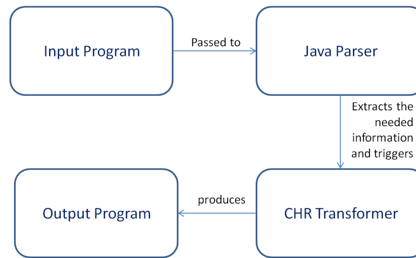


Figure 2 Overview of the system's architecture.

the extracted information from the `leq` handler presented in Section 2. Secondly, it runs the transformer using the extracted information to produce the new output program. The SWI-Prolog JPL interface [10] is used in order to run the transformer (which is a CHR program) from within the Java application. The new program has the required functionality of producing and interacting with the needed visual tracer.

The rest of the section provides an example of the output file after applying the transformation. It is concerned with the `leq` handler introduced in Section 2. The two proposed transformers produce similar output. Note that for simplicity reasons, the rules presented below are just an abstraction of the actual rules obtained by the transformers.

```
main:-initialize_visualizer_with_initial_store,proceed_tracer.
reflexivity @ leq(X,X) <=> send_visualizer_removed_head(leq(X,X)),
                           true,
                           send_visualizer_body(true),
                           rule_name(reflexivity),
                           proceed_tracer.
antisymmetry @ leq(X,Y), leq(Y,X) <=>
                            send_visualizer_removed_head(leq(X,Y)),
                            send_visualizer_removed_head(leq(Y,X)),
                            send_visualizer_body(X=Y),
                            rule_name(antisymmetry),
                            proceed_tracer,
                            X = Y.
idempotence  @ leq(X,Y)\ leq(X,Y) <=>
                            send_visualizer_removed_head(leq(X,Y)),
                            send_visualizer_kept_head(leq(X,Y)),
                            true,
                            send_visualizer_body(true),
                            rule_name(idempotence),
                            proceed_tracer.
transitivity @ leq(X,Y), leq(Y,Z) ==>
                            send_visualizer_kept_head(leq(X,Y)),
                            send_visualizer_kept_head(leq(Y,Z)),
                            send_visualizer_body(leq(X,Z)),
                            rule_name(transitivity),
                            proceed_tracer,leq(X,Z).
```

The `main` predicate rule is added to the program to be able to to initialize the visual tracer. `initialize_visualizer_with_initial_store` is an abstraction for the actions performed

to initialize the tracer. The corresponding Java class is initialized (according to the applied transformer) and the initial constraints in the constraint store are sent. `proceed_tracer` is used to add a row to the tracer's tree using the initial constraints. As for the CHR rules, the corresponding data is sent to the visual tracer. Afterwards, the tracer is advanced. Similarly, advancing the tracer here means that a new row is added to the tree data structure of the tracer. The tree is not visualized unless the user presses one of the buttons of the tracer as shown in Section 5.

For example, after the `idempotence` rule is transformed, the interactions needed with the visual tracer are added to the body. `send_visualizer_removed_head(leq(X,Y))` is used to refer to the Java method call that informs the tracer that the constraint `leq(X,Y)` that appears in the head of the rule should be removed on executing the rule. On the other hand, `send_visualizer_kept_head(leq(X,Y))` is used to refer to the Java method call informing the tracer that another constraint in the rule's head is `leq(X,Y)` and that it should be kept on executing the rule. Finally, `rule_name(idempotence)` and `proceed_tracer` are also abstractions for the Java method calls that update the tracer with the rule's name and add a row to the tracer's tree data structure using all the previously sent data respectively.

5 The Visualization

As introduced before, transformation is done in order to be able to visualize the execution of the rules or to be able to visualize some of the CHR constraints. In other words, two transformers are provided. The first one enables the user to visualize the execution in a step-by-step manner. The second transformer allows the user to visualize some of the CHR constraints themselves in order to be able to visually see the result of applying the rules of a specific program on such constraints.

In order to be able to use any of the visual tracers, the initial program has to be transformed first. Afterwards, the transformed program, which is automatically saved in a new file, is consulted in the usual way. The only thing that needs to be added to the query is `main`. This is used to initialize the visualization tool (which differs according to the transformer that was used in the first step). The two visual tracers were built using Java. The interface between SWI-Prolog and Java was also done using JPL [10]. Once the goal is entered, the execution of the CHR rules proceeds in the normal way. In addition, the visual tracer window opens. This section uses the `gcd` handler:

```
:- use_module(library(chr)).
:-chr_constraint gcd/1.

r1 @ gcd(N) \ gcd(M) <=> 0<N, N=<M | L is M mod N,
                         print('added'),nl,gcd(L).
```

5.1 Visualizing The Execution

The first transformer is used in order to be able to visualize the execution of the rules. The visual tree was built using the JGraph framework [2]. In order to visualize the execution of the different rules in a step-by-step manner, the play and back buttons are used. The fast forward button allows the user to view the full tree by one click.

The program used in this section computes the greatest common divisor of two numbers. The program contains one rule named r1. There are three properties for the visualized tree.

First of all, the name of the rule that was executed appears on the edge. Secondly, the active CHR constraints are represented with green boxes, whereas the ones that were removed from the constraint store (due to a simplification or a simpagation rule) appear as red boxes. Figure 3 shows the visualization tree after visualizing only one step. As seen through the figure, the two constraints `gcd(24)` and `gcd(40)` were used by the rule named r1. As a result of applying this rule, the constraint `gcd(16)` was added to the constraint store. Now the box containing `gcd(40)` is red while the one containing `gcd(24)` is green. This means that as a result of applying the rule named r1, the constraint `gcd(40)` was removed from the constraint store while the constraint `gcd(24)` was kept. Consequently, the rule r1 is a simpagation rule. Third of all, in addition to the constraints, the tree shows all the built-in constraints and the computations using blue boxes.

5.1.1 Removing Unneeded Nodes

The visual tracer allows users to remove any unneeded nodes in order to be able to customize the tree according to their needs. For example in Figure 3 all the print statements and computations are shown. If the user does not want to see any computation, then he/she could right click on the node so that a menu with the option "Remove Similar Constraints" appear.

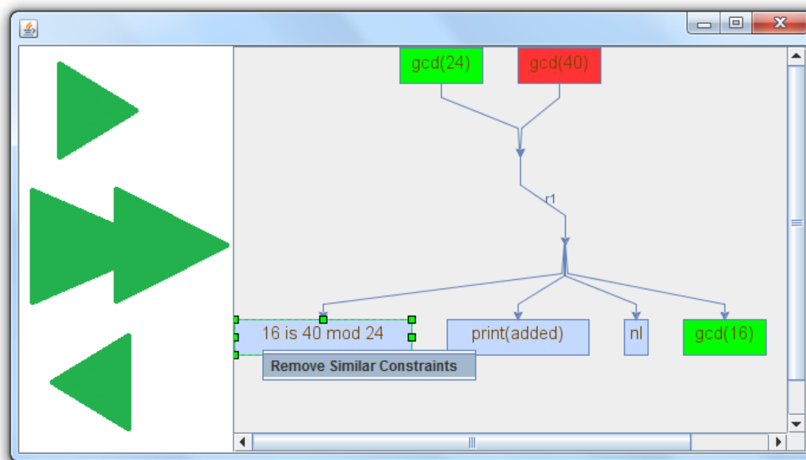


Figure 3 After the first step, right clicking a node to remove it from the tracer.

Once the user clicks on this option, the tree is redrawn without such constraints. Moreover, if the user does this in the middle of the visualization then this history is kept. In other words, if the user has chosen to remove computations then at every new step if a computation is done, it is not added to the tracer as shown in Figure 4.

5.2 Visualizing Constraints

The second transformer is used in order to be able to visualize some of the CHR constraints themselves. More specifically, if the input program manipulates a specific object, then using this transformer, the user is able to visualize the object (or the constraint itself).
For example the following program translates rectangles from one position to another.

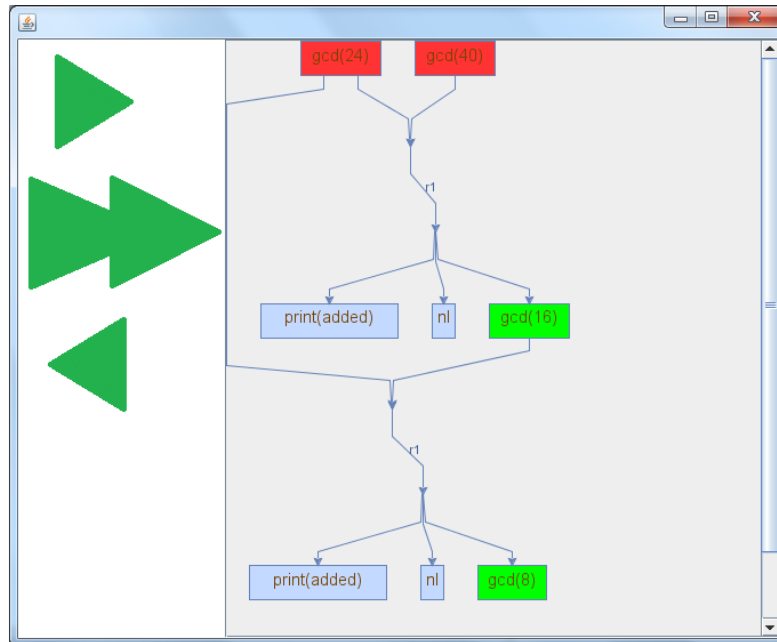


Figure 4 Node removed and history kept throughout the next step.

```
:-use_module(library(chr)).
:-chr_constraint rectangle/4,translateX/1,translateY/1.

translateX @ rectangle(X1,Y1,W,H),translateX(X) <=> NewX1 is X1+X,
                                                    rectangle(NewX1,Y1,W,H).
translateY @ rectangle(X1,Y1,W,H),translateY(Y) <=> NewY1 is Y1+Y,
                                                    rectangle(X1,NewY1,W,H).
```

In order to be able to visualize the constraints, the user has to inform the system of the CHR constraints that should be visualized and how they should be visualized. In other words, users have to specify that they want to visualize the constraint named `rectangle` and that this constraint should be visualized as a "Rectangle" object. Another example could involve the constraint `circle` that has to be visualized as an "Ellipse" object. Consequently, through the provided application, the user gets to choose the type of visualization objects that he/she would like to associate with CHR constraints. Users also decide about the constraints to visualize. They could choose to visualize all constraints, some of them or no constraints at all. Through the provided application, the user could specify all such details.

The CHR constraints in the input file appear to the user as shown in Figure 5. The default visualization object is "Line". The user could remove the constraints that are not to be visualized such as `translateX` and `translateY` in the previous example. Users could also select a constraint and choose to edit it in order to specify the type of object that should be associated with the constraint as shown in Figure 5. Once the user is done with transforming the input file, the produced output file could be used for visualization. The result of applying each rule could be seen through clicking as shown in Figure 6. Figure 6 is the result of applying the query `rectangle(10,10,50,50), translateX(100), translateY(100)` to the resulting output file (the file produced after applying the transformation). Since the object chosen for visualizing the `rectangle` constraint is "Rectangle", the arguments `10, 10, 50`

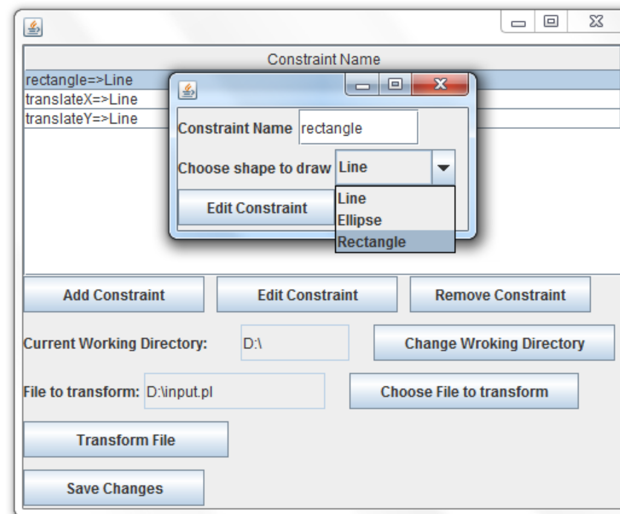
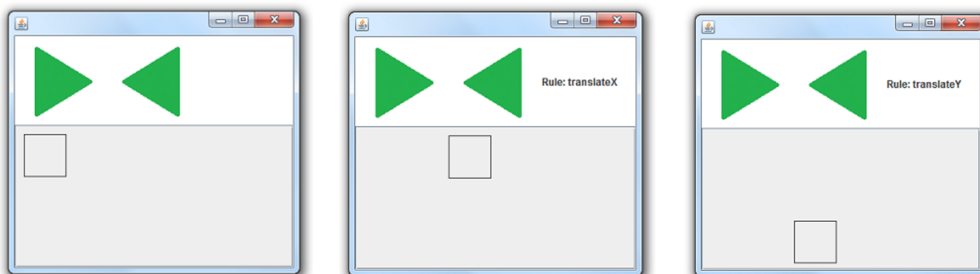


Figure 5 The user chooses to edit details about one of the constraints used in the input file.

and `50` correspond to the starting coordinates, the width and the height of the drawn rectangle respectively. Therefore, when translating the rectangle, only one of the starting coordinates needs to be changed.
Whenever a query is entered, the tracer shows the initial objects at the beginning. Through mouse clicks, the user is able to view actions performed in a step-by-step approach. At each step the new objects and the executed rule's name are shown. Figure 6b shows the rectangle after applying the `translateX` rule. The initial rectangle was removed since `translateX` is a simplification rule. Figure 6c visualizes the final rectangle after applying the rule `translateY` on the resulting rectangle. Similarly, the rectangle resulting from applying the `translateX` was removed since `translateY` is a simplification rule.



(a) The initial constraint. **(b)** After applying the rule translateX. **(c)** After applying the rule translateY.

Figure 6 Visualizing the constraints in a step-by-step approach.

6 Conclusions and Future Work

The paper introduced two source-to-source transformers for CHR. Although based on some central ideas introduced before, the new transformers introduce some new techniques of how the transformation could be done. It also overcomes many of the issues faced before. More

specifically, the transformer is incorporated within a Java application that parses the input file so that users do not have to worry about translating the program from or into any needed format. Since the transformer and the produced CHR programs were built using the current compiler and CHR runtime system, no modification is needed to be able to use them.

The transformers help visualize the execution of the different enclosed CHR rules in addition to visualizing some of the CHR constraints. Such visual tracers could be useful for many purposes including educational needs and debugging.

It was noticed that both transformers are similar. The technique used is the same in both of them. The only difference is the type of the visual tracer and its corresponding functionalities that are added to the output program. Consequently, for future work, the ultimate goal is to develop a general workbench/engine that facilitates prototyping of source-to-source transformation.

References

1 Slim Abdennadher and Matthias Saft. A Visualization Tool for Constraint Handling Rules. In *In Proceedings of 11th Workshop on Logic Programming Environments, 1th*, 2001.

2 Gaudenz Alder. *Design and Implementation of the JGraph Swing Component*, 1.0.6 edition, February 2003. Available at: http://jgraph.sourceforge.net/doc/paper/.

3 Gregory J. Duck, Peter J. Stuckey, Maria J. García de la Banda, and Christian Holzbaur. The refined operational semantics of constraint handling rules. In Bart Demoen and Vladimir Lifschitz, editors, *ICLP*, volume 3132 of *Lecture Notes in Computer Science*, pages 90–104. Springer, 2004.

4 Thom W. Frühwirth. Theory and practice of constraint handling rules, special issue on constraint logic programming. *Journal of Logic Programming*, 37(1-3):95–138, October 1998.

5 Thom W. Frühwirth. Specialization of concurrent guarded multi-set transformation rules. In Sandro Etalle, editor, *LOPSTR*, volume 3573 of *Lecture Notes in Computer Science*, pages 133–148. Springer, 2004.

6 Thom W. Frühwirth and Christian Holzbaur. Source-to-source transformation for a class of expressive rules. In Francesco Buccafurri, editor, *APPIA-GULP-PRODE*, pages 386–397, 2003.

7 Thom W. Frühwirth, Alessandra Di Pierro, and Herbert Wiklicky. Probabilistic constraint handling rules. *Electr. Notes Theor. Comput. Sci.*, 76:115–130, 2002.

8 Leslie De Koninck, Tom Schrijvers, and Bart Demoen. User-definable rule priorities for chr. In Michael Leuschel and Andreas Podelski, editors, *PPDP*, pages 25–36. ACM, 2007.

9 Matthias Schmauss. An implementation of CHR in Java. An implementation of CHR in Java, Master Thesis, Institute of Computer Science, LMU, Munich, Germany, November 1999.

10 Paul Singleton, Fred Dushin, and Jan Wielemaker. JPL: A bidirectional Prolog/Java interface. `http://www.swi-prolog.org/packages/jpl/`.

11 Jon Sneyers, Peter Van Weert, Tom Schrijvers, and Bart Demoen. Aggregates in CHR. Technical Report CW 481, Leuven, Belgium, March 2007.

12 Paolo Tacchella, Maurizio Gabbrielli, and Maria Chiara Meo. Unfolding in chr. In *Proceedings of the 9th ACM SIGPLAN international conference on Principles and practice of declarative programming*, PPDP '07, pages 179–186, New York, NY, USA, 2007. ACM.

13 Peter Van Weert, Jon Sneyers, and Bart Demoen. Aggregates for chr through program transformation. In Andy King, editor, *LOPSTR*, volume 4915 of *Lecture Notes in Computer Science*, pages 59–73. Springer, 2007.

Static Type Inference for the Q language using Constraint Logic Programming

Zsolt Zombori, János Csorba, and Péter Szeredi

Department of Computer Science and Information Theory
Budapest University of Technology and Economics
Budapest, Magyar tudósok körútja 2. H-1117, Hungary
`{zombori,csorba,szeredi}@cs.bme.hu`

Abstract

We describe an application of Prolog: a type inference tool for the Q functional language. Q is a terse vector processing language, a descendant of APL, which is getting more and more popular, especially in financial applications. Q is a dynamically typed language, much like Prolog. Extending Q with static typing improves both the readability of programs and programmer productivity, as type errors are discovered by the tool at compile time, rather than through debugging the program execution.

We map the task of type inference onto a constraint satisfaction problem and use constraint logic programming, in particular the Constraint Handling Rules extension of Prolog. We determine the possible type values for each program expression and detect inconsistencies. As most built-in function names of Q are overloaded, i.e. their meaning depends on the argument types, a quite complex system of constraints had to be implemented.

1998 ACM Subject Classification I.2.3 Deduction and Theorem Proving

Keywords and phrases logic programming, types, static type checking, CSP, CHR, Q language

Digital Object Identifier 10.4230/LIPIcs.ICLP.2012.119

1 Introduction

Our paper presents most recent developments of the `qtchk` type analysis tool, for the Q vector processing language. The tool has been designed in a collaborative project between Budapest University of Technology and Economics and Morgan Stanley Business and Technology Centre, Budapest. We described our first results in [19]. That version provided *type checking*: the programmer had to provide type annotations (in the form of appropriate Q comments) and our task was to verify the correctness of the annotations. Since then, we moved from type checking towards *type inference*: we devised an algorithm for inferring the possible types of all program expressions, without relying on user provided type information. Our preliminary results with the type inferencer were presented in [4]. Now we report on the more mature `qtchk` system that is nearly complete. The main goal of the type inference tool is to detect type errors and provide detailed error messages. Our tool can help detect program errors that would otherwise stay unnoticed, thanks to which it has the potential to greatly enhance program development. We perform type inference using constraint logic programming: the initial task is mapped onto a constraint satisfaction problem (CSP), which is solved using the Constraint Handling Rules extension of Prolog [7], [15].

In Section 2 we give some background information. Section 3 briefly discusses approaches to type inference that are related to our work. Section 4 contains our main contribution: we present static type inference as a constraint satisfaction problem. Section 5 presents the `qtchk` program, a static type inferencer for Q that implements the algorithm outlined

© Zsolt Zombori, János Csorba, and Péter Szeredi;
licensed under Creative Commons License ND
Technical Communications of the 28th International Conference on Logic Programming (ICLP'12).
Editors: A. Dovier and V. Santos Costa; pp. 119–129

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

in Section 4. Due to lack of space, we only address type inference proper. More details about parsing Q programs and the system architecture can be found in [19]. In Section 6 we evaluate our tool.

2 Background

In this section we present the Q programming language. Due to lack of space we do not describe the necessary background related to constraint logic programming: we expect the readers to be familiar with the constraint satisfaction problem, the Prolog language and the Constraint Handling Rules (CHR) extension of Prolog.

2.1 The Q Programming Language

Q is a highly efficient vector processing functional language, which is well suited to performing complex calculations quickly on large volumes of data. Consequently, numerous investment banks (Morgan Stanley, Goldman Sachs, Deutsche Bank, Zurich Financial Group, etc.) use this language for storing and analysing financial time series[1]. The Q language [1] first appeared in 2003 and is now (February 2012) so popular, that it is ranked among the top 50 programming languages by the TIOBE Programming Community [18].

Types Q is a strongly typed, dynamically checked language. This means that while each variable is associated with a well-defined type, the type of a variable is not declared explicitly, but stored along its value during execution. The most important types are as follows:

- **Atomic types** in Q correspond to those in SQL with some additional date and time related types that facilitate time series calculations. Q has the following 16 atomic types: `boolean`, `byte`, `short`, `int`, `long`, `real`, `float`, `char`, `symbol`, `date`, `datetime`, `minute`, `second`, `time`, `timespan`, `timestamp`.
- **Lists** are built from Q expressions of arbitrary types, e.g. `(1;2.2;‘abc)` is a list comprising two numbers and a symbol.
- **Dictionaries** are a generalisation of lists and provide the foundation for tables. A dictionary is a mapping given by exhaustively enumerating all domain-range pairs. E.g., `((‘a;‘b) ! (1;2))` is a dictionary that maps symbols `a,b` to integers `1,2`, respectively.
- **Tables** are lists of special dictionaries that correspond to SQL records.
- **Functions** correspond to mathematical mappings specified by an algorithm.

Main Language Constructs As Q is a functional language, functions form the basis of the language. A function is composed of an optional parameter list and a body comprising a sequence of expressions to be evaluated. Function application is the process of evaluating the sequence of expressions obtained after substituting actual arguments for formal parameters. For example, the expression `f: {[x] $[x>0;sqrt x;0]}` defines a function of a single argument x, returning $\sqrt{x}$, if $x > 0$, and 0 otherwise. Input and return values of functions can also be functions. While being a functional language, Q also has imperative features, such as multiple assignment[2] of variables and loops.

[1] Kx-Systems: `http://kx.com/Customers/end-user-customers.php`
[2] Assignment is denoted by a colon, e.g. `x:x*2` doubles the value of the variable `x`.

Type restrictions in Q The program code environment can impose various kinds of restrictions on types of expressions. In certain contexts, only one type is allowed. For example, in the do-loop `do[n;x:x*2]`, the first argument is required to be an integer. In other cases we expect a polymorphic type, such as a list (`list(A)`, where `A` is an arbitrary type). In the most general case, there is a restriction involving the types of several expressions. For instance, in the expression `x = y + z`, the type of `x` depends on those of `y` and `z`. A type analyser for Q has to use a framework that allows for formulating all type restrictions that can appear in the program.

2.2 Restriction of the Q language for type reasoning

Q is a very permissive language. In consultation with experts at Morgan Stanley we decided to impose some restrictions on the language supported by the inference tool, in order to promote good coding practice and make the type analyser more efficient.

With multiple assignment variables and dynamic typing, Q allows for setting a variable to a value of type different from that of the current value. However, this is not the usual practice and it defies the very goal of type checking. Hence we agreed that each variable should have a single type in a program, otherwise the type analyser gives an error message.

Other restrictions concern the types of the built-in functions. Most built-in functions in Q are highly overloaded, thanks to which some functions do not raise errors for certain "strange" arguments. For example, the built-in function `last` takes a list as argument and returns the last element of the list. However, this function works on atomic arguments as well: it simply returns the input argument. To increase the efficiency of the type reasoner we decided to ignore some special meanings of some built-in functions. For example, we neglected this special meaning of the `last` function. Consequently, we infer that the argument of the `last` function is a list, which is not necessarily true in general.

3 Related Work

One of the first algorithms for type inference is the Hindley-Milner type system [8]. It associates the program with a set of equations which can be solved by unification. It supports parametric polymorphism, i.e., allows for using type variables. Most type systems for statically typed functional languages are extensions of the Hindley-Milner system, for example the ML family [14] and Haskell [9]. We also find several examples of dynamically typed languages extended with a type system allowing for type checking and type inference. These attempts aim to combine the safeness of static typing with the flexibility of dynamic typing. [12] describe a polymorphic type system for Prolog.

A major limitation of the Hindley-Milner system is that it requires disjoint types. This limitation is lifed in *subtyping* [2], which is a generalisation of Hindley-Milner. Here, the input program is mapped into type constraints of the form $U \subseteq V$ where U and V are types. [11] and [10] present type checking tools for Erlang, a dynamically typed functional language, based on subtyping. They introduce the notion of success typing: in case of potential type errors, they assume that the programmer knows what he wants and only reject programs where the type error is certain. Their tool aims to automatically discover hidden type information, without requiring any alteration of the code. Q is a dynamically typed functional language, just like Erlang. While the language naturally yields many constraints of the form $U \subseteq V$, subtyping is not sufficient to capture all constraints related to types. Built-in functions are highly overloaded (ad-hoc polymorphism), and we need more sophisticated tools, like constraint logic programming, to formulate and handle complex constraints. [5] report

on using constraints in type checking and inference for Prolog. They transform the input logic program with type annotations into another logic program over types, whose execution performs the type checking. [17] describe a generic type inference system for a generalisation of the Hindley-Milner approach using constraints, and also report on an implementation using Constraint Handling Rules. The CLP($\mathcal{SET}$) [6] framework provides constraint logic reasoning over sets. Our solution has many similarities to CLP($\mathcal{SET}$) as types can be easily seens as sets of expressions. The main difference is that we have to handle infinite sets.

4 Type Inference as a Constraint Satisfaction Problem

In this section we give an overview of our approach of transforming the problem of type reasoning into a CSP. Type reasoning starts from a program code that can be seen as a complex expression built from simpler expressions. Our aim is to assign a type to each expression appearing in the program in a coherent manner. The types of some expressions are known immediately (atomic expressions, certain built-in functions), besides, the program syntax imposes restrictions between the types of certain expressions. The aim of the reasoner is to assign a type to each expression that satisfies all the restrictions.

We associate a CSP variable with each subexpression of the program. Each variable has a domain, which initially is the set of all possible types. Different type restrictions can be interpreted as constraints that restrict the domains of some variables. In this terminology, the task of the reasoner is to assign a value to each variable from the associated domain that satisfies all the constraints. However, our task is more difficult than a classical CSP, because there are infinitely many types, which cannot be represented explicitly in a list.

4.1 Type Language for Q

We describe the type language developed for Q. We allow polymorphic type expressions, i.e., any part of a complex type expression can be replaced with a variable. Expressions are built from atomic types and variables using type constructors. The abstract syntax of the type language – which is also the Prolog representation of types – is as follows:

```
TypeExpr =
    AtomicTypes     | TypeVar     | symbol(Name)     | any
  | list(TypeExpr) | tuple([TypeExpr,...,TypeExpr])
  | dict(TypeExpr, TypeExpr)     | func(TypeExpr, TypeExpr)
```

`AtomicTypes` This is shorthand for the 16 atomic types of Q. Furthermore, the `numeric` keyword can be used to denote a type consisting of all numeric values.

`TypeVar` represents an arbitrary type expression, with the restriction that the same variables stand for the same type expression. Type variables allow for defining polymorphic type expressions, such as `list(A) -> A` and `tuple([A,A,B])`.

`symbol(Name)` This is a degenerate type, as it has a single instance only, namely the provided symbol. Nevertheless, it is important because in order to support certain table operations, the type reasoner needs to know what exactly the involved symbols are.

`any` This is a generic type description, which denotes all data structures allowed by Q.

`list(TE)` The set of all lists whose elements are from the set represented by *TE*.

`tuple([TE`$_1$`, ..., TE`$_k$`])` The set of all lists of length k, such that the ith element is from the set represented by TE_i.

`dict(`TE_1`, `TE_2`)` The set of all dictionaries, defined by an explicit association between a domain list (TE_1) and a range list(TE_2) via positional correspondence. For example, the dictionary `(‘name;‘date) !  (‘Joe; 1962)` has type `dict(tuple([symbol(name),symbol(date)]),tuple([symbol(Joe),int]))`[3].

`func(`TE_1`, `TE_2`)` The set of all functions, such that the domain and range are from the sets represented by TE_1 and TE_2, respectively.

4.2 Domains

Type expressions can be embedded into each other (e.g. `list(int)`, `list(list(int))`, etc.), and tuples can be of arbitrary length, consequently we have infinitely many types, which makes representing domains more difficult. Furthermore, the types determined by the type language are not disjoint. For example `1.1f` might have type `float` or `numeric` as well. It is evident that every expression which satisfies type `float` also satisfies type `numeric`, i.e., `float` is a *subtype* of `numeric`. We will use the subtype relation to represent infinite domains finitely as intervals: a domain will be represented with an upper and a lower bound.

Partial Ordering We say that type expression T_1 is a subtype of type expression T_2 ($T_1 \leq T_2$) if and only if, all expressions that satisfy T_1 also satisfy T_2. The subtype relation determines a partial ordering over type expressions. For example, consider the `tuple([int,int])` type which represents lists of length two, both elements being integers. Every expression that satisfies `tuple([int,int])` also satisfies `list(int)`, i.e., `tuple([int,int])` is a subtype of `list(int)`. For atomic expressions it is trivial to check if one type is the subtype of another. Complex type expressions can be checked using some simple recursive rules. For example, `list(A)` is a subtype of `list(B)` exactly if `A` is subtype of `B`.

Finite Representation of the Domain The domain of a variable is initially the set of all types, which can be constrained with different upper and lower bounds.

An upper bound restriction for variable X_i is a list $L_i = [T_{i1}, \ldots, T_{in_i}]$, meaning that the upper bound of X_i is $\bigcup_{j=1}^{n_i} T_{ij}$, i.e., the type of X_i is a subtype of some element of L_i. Disjunctive upper bounds are very common and natural in Q, for example, the type of an expression might have to be either `list` or `dict`. The conjunction of upper bounds is easily described by having multiple upper bounds. If we have two upper bounds L_1 and L_2 on the same variable X_i, this means the value of X_i expression has to be in $\bigcup(T_{1j} \bigcap T_{2k})$, for all $1 \leq j \leq n_1$ and $1 \leq k \leq n_2$.

A lower bound restriction for variable X_i is a single type expression T_i, meaning that T_i is a subtype of the type of X_i. For lower bounds, it is their union which is naturally represented by having multiple constraints: if X has two lower bounds T_1 and T_2, then $T_1 \cup T_2$ has to be subtype of the type of X. We do not use lists for lower bounds and hence cannot represent the intersection of lower bounds. We chose this representation because no language construct in Q yields a conjunctive lower bound.

With the following example we demonstrate that lower and upper bounds are natural restrictions in Q: In the code `a:  f[b]` function `f` is applied to `b` and the result is assigned to `a`. Suppose the type of `f` turns out to be a map from `numeric` to `tuple([int, int])`. We can infer that the type of `b` must be at most `numeric`, which can be expressed with an

[3] To facilitate type inference for tables, we include detailed information on the domain/range of a dictionary in its type. (A record is a dictionary with the domain being a list of column names.)

upper bound. The result of `f` of `b` has the type `tuple([int,int])`, which means, that the type of `a` must be at least `tuple([int,int])`, which can be expressed with a lower bound. If later the type of `a` turns out to be `list(int)` (a list of integers) and the type of `b` to be e.g. `float`, then the above expression is type correct.

4.3 Constraints

After parsing – where we build an abstract syntax tree representation of the input program – the type analyser traverses the abstract syntax tree and imposes constraints on the types of the subexpressions. The constraints describing the domain of a variable are particularly important, we call them *primary constraints*. These are the upper and lower bound constraints. We will refer to the rest of the constraints as *secondary constraints*. Secondary constraints eventually restrict domains by generating primary constraints, when their arguments are sufficiently instantiated (i.e., domains are sufficiently narrow). Constraints that can be used for type inference can originate from the following sources in a Q program:

Built-in functions For every built-in function, there is a well-defined relationship between the types of its arguments and the type of the result. These relations are expressed by adequate – sometimes quite complicated – constraints.

Atomic expressions The types of atomic expressions are revealed already by the parser, so for example, `2.2f` is immediately known to be a `float`.

Variables Local variables are made globally unique by the parser, so variables with the same name must have the same type. We ensure this by equating their corresponding domains.

Program syntax Most syntactic constructs impose constraints on the types of their constituent constructs. For example, the first argument of an `if-then-else` construct must be `int` or `boolean`. Another example is the assignment construct. The type of the left side has to be at least as "broad" as the type of the right side. It means the type of the right side is subtype of the type of the left side.

4.4 Constraint Reasoning

In this subsection we describe how the constraints are used to infer possible types. Constraint reasoning is based on a *production system* [13], i.e., a set of IF-THEN rules. We maintain a *constraint store* which holds the constraints to be satisfied for the program to be type correct. We start out with an initial set of constraints. A production rule fires when certain constraints appear in the store and results in adding or removing some constraints. With CHR terminology, we say that each rule has a head part that holds the constraits necessary for firing and a body containing the constraints to be added. The constraints to be removed are a subset of the head constraints. One can also provide a guard part to specify more refined firing conditions.

The semantics of the constraints is given by describing their consequences and their interactions with other constraints. At each step we systematically check for rules that can fire. The more rules we provide the more reasoning can be performed.

Primary constraints represent variable domains. If a domain turns out to be empty, this indicates a type error and we expect the reasoner to detect this. Hence, it is very important for the constraint system to handle primary constraints as "cleverly" as possible. For this, we formulated rules to describe the following interactions on primary constraints[4]:

---

[4] Concrete examples of rules will be given in Section 5.

- Two upper bounds on a variable should be replaced with their intersection.
- Two lower bounds on a variable should be replaced with their union.
- If a variable has an upper and a lower bound such that no type satisfies both, then the clash should be made explicit by setting the upper bound to the empty set.
- Upper and lower bounds can be polymorphic, i.e., they might contain other variables. From the fact that the lower bound must be a subtype of the upper bound, we can propagate constraints to the variables appearing in the bounds.

Secondary constraints connect different variables and restrict several domains. Unfortunately, it is not realistic to capture all interactions of secondary constraints as that would require exponentially many rules in the number of constraints. Hence, we only describe (fully) the interaction of secondary constraints with primary constraints, i.e., we formulate rules of the form: if certain arguments of the constraints are within a certain domain, then some other argument can be restricted. E.g., if there is a summation in Q and we know that the arguments are numeric values, then the result must be either integer or float. If the second argument later turns out to be float, then the result must be float. At this point, there is nothing more to be inferred and the constraint can be eliminated from the store.

Our aim is to eventually eliminate all secondary constraints. If we manage to do this, the domains described by the primary constraints constitute the set of possible type assignments to each expression. In case some domain is the empty set, we have a type error. Otherwise, we consider the program type correct.

If the upper and lower bounds on a variable determine a singleton set, then we say that it is *instantiated*. If all arguments of a secondary constraint are instantiated, then there are two possibilities. If the instantiation satisfies the constraint, then the latter can be removed from the store. Otherwise, the constraint fails.

Error Handling As we parse the input program, we generate constraints and add them to the constraint store. The production rules automatically fire whenever they can. If some domain gets restricted to the empty set, this means that the corresponding expression cannot be assigned any type, i.e., we have a type error. At this point we mark the erroneous expression, as well as the primary constraints whose interaction resulted in the empty domain. This information – along with the position of the expression – is used to generate an error message. The primary constraints are meant to justify the error. Once the error has been detected and noted, we roll back to the addition of the last constraint and simply proceed by skipping the constraint. This way, the type analyser can detect more than one error during a single run.

Labeling Eventually, after all constraints have been added, we obtain a constraint store such that none of the rules can fire any more. There are three possibilities:

- There were some discovered errors. Then we display the collected error messages and terminate the type inference algorithm.
- There were no type errors found and only primary constraints remain. In this case the domains described by the primary constraints all contain at least one element. Any type assignment from the respective domains satisfies all constraints, so the type analyser stops with success.
- No type errors were found, however, some secondary constraints remain. In order to decide if the constraints are consistent, we do *labeling*.

Labeling is the process of systematically assigning values to variables from within their domains. The assignments wake up production rules. We might obtain a failure, in which case we roll back until the last assignment and try the next value. Eventually, either we find a type assignment to all variables that satisfies all constraints or we find that there is no consistent assignment. In the first case we indicate that there is no type error. In the second case, however, we showed that the type constraints are inconsistent, so an error message to this effect is displayed. Due to the potentially large size of the search space traversed in labeling, it looks very difficult to provide the user with a concise description of the error.

5 Implementation – the `qtchk` program

We built a Prolog program called `qtchk` that implements the type analysis described in Section 4. It runs both in SICStus Prolog 4.1 [16] and SWI Prolog 5.10.5. It consists of over 8000 lines of code[5]. Constraint reasoning is performed using Constraint Handling Rules. Q has many irregularities and lots of built-in functions (over 160), due to which a complex system of constraints had to be implemented using over 60 constraints. The detailed user manual for `qtchk` can be found in [3] that contains lots of examples along with the concrete syntax of the Q language.

5.1 Representing variables

All subexpressions of the program are associated with CSP variables. In case some constraint fails, we need to know which expression is erroneous in order to generate a useful error message. If the arguments of the constraints are variables, we do not have this information at hand. Hence, instead of variables we use identifiers `ID = id(N,Type,Error)`[6], which consist of three parts: an integer `N` which uniquely identifies the corresponding expression, the type proper `Type` (which is a Prolog variable before the type is known) and an error flag `Error` which is used for error propagation. We use the same representation for type variables in polymorphic types, e.g. the type `list(X)` may be represented by `list(id(2))`.

5.2 Constraint Reasoning

Constraint reasoning is performed using the Constraint Handling Rules library of Prolog. CHR has proved to be a good choice as it is a very flexible tool for describing the behaviour of constraints. Any constraint involving arbitrary Prolog structures could be formulated. We illustrate our use of CHR by presenting some rules that describe the interaction of primary constraints. Our two primary constraints are

- `subTypeOf(ID,L)`: The type of identifier `ID` is a subtype of some type in `L`, where `L` is a list of polymorphic type expressions.
- `superTypeOf(ID,T)`: The type of `ID` is a supertype of `T`, a polymorphic type expression.

With polymorphic types we can restrict the domain by a type expression containing the type of another identifier. If the type of such an identifier becomes known, the latter is replaced with the type in the constraint. For example, if we have constraints
`subTypeOf(id(1),[float,list(id(2))])`, `superTypeOf(id(1),tuple([id(3),int])`
and the types of `id(2)` and `id(3)` later both turn out to be `int`, then the constraints are automatically replaced with

[5] We are happy to share the code over e-mail with anyone interested in it.
[6] In order to make the following examples easier to read, we will write `id(N)` instead of `id(N,Type,Error)`

`subTypeOf(id(1),[float,list(int)])`, `superTypeOf(id(1),tuple([int,int])`.
Due to the lower bound, `float` can be eliminated from the upper bound. This is performed by the following CHR rule:

```
superTypeOf(X,A) \ subTypeOf(X,B0) <=> eliminate_sub(A, B0, B) |
      create_log_entry(eliminate_sub(X,A,B0,B)), subTypeOf(X, B).
```

We make use of the Prolog predicate `eliminate_sub(A,B0,B)`, which expresses that the list of upper bounds `B0` can be reduced to a proper subset `B` based on lower bound `A`. We obtain: `subTypeOf(id(1),[list(int)])`, `superTypeOf(id(1),tuple([int,int])`.

5.3 Error Handling

During constraint reasoning, a Prolog failure indicates some type conflict. Before we roll back to the last choice point, we remember the details of the error. We maintain a log that contains entries on how various domains change during the reasoning and what constraints were added to the store. Furthermore, to make error handling more uniform, whenever secondary constraints are found violated, they do not lead to failure, but they set some domain empty. Hence, we only need to handle errors for primary constraints. Whenever a domain gets empty, we mark the expression associated with the domain and we look up the log to find the domain restrictions that contributed to the clash. We create and assert an error message and let Prolog fail. For example, the following message

```
Expected to be broader than (int -> numeric) and
               narrower than (int -> int)
  file:samples/s1.q  line:13  character:4
   {[x] f[x]}
   ^^^^^^^^^^
```

indicates that the underlined function definition is erroneous: the return value is numeric or broader, although it is supposed to be narrower than integer.

6 Evaluation

The best way to evaluate our tool would be on Q programs developed by Morgan Stanley. However, we could not obtain such programs due to the security policy of the company. Instead, we used user contributed Q examples, publicly available at the homepage of Kx-System [1]. This test set contains several (extended) examples from the Q tutorial and other more complex programs. Table 1 summarizes our findings.

Table 1 Test results.

All	*Type correct*	*Restrictions*	*Labeling timeout*	*Type error*	*Analyser error*
128	43 (33.6%)	43 (33.6%)	32 (25%)	5 (3.9%)	5 (3.9%)

We used 128 publicly available Q programs. Of this 43 were found type correct. As explained in Subsection 2.2, we made some restrictions on the Q language, following the requirements of Morgan Stanley. 43 programs were found erroneous due to not fulfilling these restrictions. Most of the error messages arise from the same variable used with different types and from some neglected special meaning of built-in functions. We often found the

case that a function is called but defined in another file that was not included among the examples. In such programs the lack of information often resulted in an extremely large search space to be traversed during labeling. In 32 programs labeling could not find any solution within the given time limit (1000 sec), partly for the former reason.

We were happy to find 5 genuine errors in the test set. These are in the following programs: `run.q`[7], `mserve.q`[8], `oop.q`[9], `quant.q`[10], and `dgauss.q`[11]. We have found 5 programs containing some language element that our tool cannot handle well. We are in the process of eliminating these problems.

7 Conclusions

We presented an algorithm and the tool `qtchk` that can be used for checking Q programs for type correctness. We described how to map this task onto a constraint satisfaction problem which we solve using constraint logic programming tools. We have found that our program is a useful tool for finding type errors, as long as the programmers adhere to some coding practices, negotiated with Morgan Stanley, our project partner. However, we believe that the restrictions that we impose on the use of the Q language are reasonable enough for other programmers as well, and our tool will find users in the broader Q community.

Acknowledgements

The results discussed above are supported by the grant TÁMOP – 4.2.2.B-10/1–2010-0009 .

References

1. Jeffry A. Borror. *Q For Mortals: A Tutorial In Q Programming.* CreateSpace, Paramount, CA, 2008.
2. Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM COMPUTING SURVEYS*, 17(4):471–522, 1985.
3. János Csorba, Péter Szeredi, and Zsolt Zombori. *Static Type Checker for Q Programs (Reference Manual)*, 2011. http://www.cs.bme.hu/~zombori/q/qtchk_reference.pdf.
4. János Csorba, Zsolt Zombori, and Péter Szeredi. Using constraint handling rules to provide static type analysis for the q functional language. *CoRR*, abs/1112.3784, 2011.
5. Bart Demoen, M. García de la Banda, and P. Stuckey. Type constraint solving for parametric and ad-hoc polymorphism. In *Proceedings of Australian Workshop on Constraints*, pages 1–12, 1998.
6. Agostino Dovier, Carla Piazza, Enrico Pontelli, and Gianfranco Rossi. Sets and constraint logic programming. *ACM Trans. Program. Lang. Syst.*, 22(5):861–931, September 2000.
7. Th. Fruehwirth. Theory and Practice of Constraint Handling Rules. In P. Stuckey and K. Marriot, editors, *Journal of Logic Programming*, volume 37(1–3), pages 95–138, October 1998.
8. R. Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:pp. 29–60, 1969.
9. Mark P. Jones. Typing Haskell in Haskell. In *Haskell Workshop*, 1999.

[7] `http://code.kx.com/wsvn/code/contrib/cburke/qreference/source/run.q`
[8] `http://code.kx.com/wsvn/code/kx/kdb+/e/mserve.q`
[9] `http://code.kx.com/wsvn/code/contrib/azholos/oop.q`
[10] `http://code.kx.com/wsvn/code/contrib/gbaker/common/quant.q`
[11] `http://code.kx.com/wsvn/code/contrib/gbaker/deprecated/dgauss.q`

10 Tobias Lindahl and Konstantinos F. Sagonas. Practical type inference based on success typings. In Annalisa Bossi and Michael J. Maher, editors, *PPDP*, pages 167–178. ACM, 2006.
11 Simon Marlow and Philip Wadler. A practical subtyping system for Erlang. *SIGPLAN Not.*, 32:136–149, August 1997.
12 Alan Mycroft and Richard A. O'Keefe. A polymorphic type system for Prolog. *Artificial Intelligence*, 23(3):295–307, 1984.
13 A. Newell and H.A. Simon. *Human Problem Solving*. Prentice Hall, Englewood Cliffs, 1972.
14 Francois Pottier and Didier Remy. The essence of ML type inference. *Advanced Topics in Types and Programming Languages*, pages 389–489, 2005.
15 Tom Schrijvers and Bart Demoen. The K.U.Leuven CHR system: implementation and application. In *First Workshop on Constraint Handling Rules: Selected Contributions*, pages 1–5, 2004.
16 SICS. *SICStus Prolog Manual version 4.1.3*. Swedish Institute of Computer Science, September 2010.
`http://www.sics.se/sicstus/docs/latest4/html/sicstus.html`.
17 Martin Sulzmann and Peter J. Stuckey. HM(X) type inference is CLP(X) solving. *Journal of Functional Programming*, 18:251–283, March 2008.
18 TIOBE. TIOBE programming-community, TIOBE index, 2010. `http://www.tiobe.com`.
19 Zsolt Zombori, János Csorba, and Péter Szeredi. Static type checking for the q functional language in prolog. In John P. Gallagher and Michael Gelfond, editors, *ICLP (Technical Communications)*, volume 11 of *LIPIcs*, pages 62–72. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2011.

Improving Lazy Non-Deterministic Computations by Demand Analysis

Michael Hanus

Institut für Informatik, CAU Kiel, D-24098 Kiel, Germany
mh@informatik.uni-kiel.de

Abstract

Functional logic languages combine lazy (demand-driven) evaluation strategies from functional programming with non-deterministic computations from logic programming. The lazy evaluation of non-deterministic subexpressions results in a demand-driven exploration of the search space: if the value of some subexpression is not required, the complete search space connected to it is not explored. On the other hand, this improvement could cause efficiency problems if unevaluated subexpressions are duplicated and later evaluated in different parts of a program. In order to improve the execution behavior in such situations, we propose a program analysis that guides a program transformation to avoid such inefficiencies. We demonstrate the positive effects of this program transformation with KiCS2, a recent highly efficient implementation of the functional logic programming language Curry.

1998 ACM Subject Classification D.1.6 Logic Programming

Keywords and phrases functional logic programming, implementation, program analysis

Digital Object Identifier 10.4230/LIPIcs.ICLP.2012.130

1 Motivation

Functional logic languages support the most important features of functional and logic programming in a single language (see [10, 32] for recent surveys). They provide higher-order functions and demand-driven evaluation from functional programming as well as logic programming features like non-deterministic search and computing with partial information (logic variables). This combination led to new design patterns [8, 11], better abstractions for application programming (e.g., programming with databases [18, 26], GUI programming [29], web programming [30, 31, 35], string parsing [22]), and new techniques to implement programming tools, like partial evaluators [3] or test case generators [27, 50].

The implementation of functional logic languages is challenging due to the combination of the various language features. For instance, one can

- design new abstract machines appropriately supporting these operational features and implementing them in some (typically, imperative) language, like C [43] or Java [12, 37],
- compile into logic languages like Prolog and reuse the existing backtracking implementation for non-deterministic search as well as logic variables and unification for computing with partial information [7, 41], or
- compile into non-strict functional languages like Haskell and reuse the implementation of lazy evaluation and higher-order functions [20, 21].

The latter approach requires the implementation of non-deterministic computations in a deterministic language but has the advantage that the explicit handling of non-determinism allows for various search strategies, like depth-first, breadth-first, parallel, or iterative deepening, instead of committing to a fixed (incomplete) strategy like backtracking [20].

© Michael Hanus;
licensed under Creative Commons License ND
Technical Communications of the 28th International Conference on Logic Programming (ICLP'12).
Editors: A. Dovier and V. Santos Costa; pp. 130–143

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

In this paper we consider KiCS2 [19], a new system that compiles functional logic programs of the source language Curry [38] into purely functional Haskell programs. However, the techniques presented in this paper can also be applied to similar implementations, like KiCS [21] or ViaLOIS [13]. KiCS2 can compete with or outperform other existing implementations of Curry [19]. In particular, deterministic parts of a program are much faster executed than in Prolog-based Curry implementations. Non-determinism is implemented in KiCS2 by representing all non-deterministic results of a computation as a data structure. This structure is traversed by operations implementing the search for solutions. Thus, different search strategies are supported by KiCS2. This flexibility might cause efficiency problems in some situations due to the duplication of unevaluated subexpressions (see below for a more detailed explanation). Therefore, we propose a new technique to improve such problematic situations based on the following steps:

1. The run-time behavior of the program is analyzed. In particular, information about demanded arguments and the non-determinism behavior is approximated.
2. The information obtained from this analysis is used to transform the source program. In particular, the computation of a non-deterministic subexpression is enforced earlier when its value is definitely demanded.

In this paper, we review Curry and its implementation with KiCS2, discuss the potential problems of this implementation and present a program transformation based on a demand analysis which avoids these problems in many practical cases. Due to lack of space, we have to omit some details which can be found in a corresponding technical report [34].

2 Functional Logic Programming and Curry

The declarative multi-paradigm language Curry [38] combines features from functional programming (demand-driven evaluation, parametric polymorphism, higher-order functions) and logic programming (computing with partial information, unification, constraints). The syntax of Curry is close to Haskell[1] [47]. In addition, Curry allows free (logic) variables in conditions and right-hand sides of defining rules. The operational semantics is based on an optimal evaluation strategy [6] which is a conservative extension of lazy functional programming and logic programming.

A Curry program consists of the definition of data types (introducing *constructors* for the data types) and *operations* on these types. For instance, the data types for Boolean values and polymorphic lists are as follows:

```
data Bool = False | True
data List a = [] | a : List a    -- [a] denotes "List a"
```

Note that, in a functional logic language like Curry, not all definable operations are functions in the classical mathematical sense. There are also operations, sometimes called "non-deterministic functions" [28], which might yield more than one result on the same input. For instance, Curry contains a *choice* operation defined by:

```
x ? _ = x
_ ? y = y
```

[1] Variables and function names usually start with lowercase letters and the names of type and data constructors start with an uppercase letter. The application of f to e is denoted by juxtaposition ("$f\ e$").

A *value* is an expression without defined operations. Thus, the expression "`True ? False`" has two values: `True` and `False`. If expressions have more than one value, one wants to select intended values according to some constraints, typically in conditions of program rules. A *rule* has the form "$f\ t_1 \ldots t_n$ `|` c `=` e" where the (optional) condition c is a *constraint*, like the trivial constraint `success` or an *equational constraint* e_1 `=:=` e_2 which is satisfied if both sides are reducible to unifiable values. For instance, the rule

```
last xs | (ys++[z]) =:= xs  =  z          where ys,z free
```

defines an operation to compute the last element `z` of a list `xs` based on the (infix) operation "`++`" which concatenates two lists (in contrast to Prolog, free variables like `ys` or `z` need to be declared explicitly to make their scopes clear). As mentioned above, operations can be non-deterministic:

```
aBool = True ? False
```

Using such non-deterministic operations as arguments might cause a semantical ambiguity which has to be fixed. Consider the operations

```
xor True  x = not x             not True  = False
xor False x = x                 not False = True
```

```
xorSelf x = xor x x
```

and the expression "`xorSelf aBool`". If we interpret this program as a term rewriting system, we could have the derivation

```
xorSelf aBool  →  xor aBool aBool   →  xor True aBool
               →  xor True False    →  not False      →  True
```

leading to the unintended result `True`. Note that this result cannot be obtained if we use a strict strategy where arguments are evaluated prior to the function calls. In order to avoid dependencies on the evaluation strategies and exclude such unintended results, González-Moreno et al. [28] proposed the rewriting logic CRWL as a logical (execution- and strategy-independent) foundation for declarative programming with non-strict and non-deterministic operations. CRWL specifies the *call-time choice* semantics [40], where values of the arguments of an operation are determined before the operation is evaluated. This can be enforced in a lazy strategy by sharing actual arguments. For instance, the expression above can be lazily evaluated provided that all occurrences of `aBool` are shared so that all of them reduce either to `True` or to `False` consistently

In order to provide a precise definition of the semantics of non-deterministic and non-strict operations, we assume a given program $\mathcal{P}$ and extend standard expressions so that they can also contain the special symbol $\bot$ to represent *undefined or unevaluated values*. A *partial value* is a value containing occurrences of $\bot$. A *partial constructor substitution* is a substitution that replaces variables by partial values. Then we denote by

$$[\mathcal{P}]_\bot = \{\sigma(l) = \sigma(r) \mid l = r \in \mathcal{P}, \sigma \text{ partial constructor substitution}\}$$

the set of all *partial constructor instances* of the program rules. A *context* $\mathcal{C}[\cdot]$ is an expression with some "hole". Then the reduction relation used in this paper is defined as follows:[2]

$$\begin{array}{ll} \mathcal{C}[f\ t_1 \ldots t_n] \twoheadrightarrow \mathcal{C}[r] & \text{if } f\ t_1 \ldots t_n = r \in [\mathcal{P}]_\bot \\ \mathcal{C}[f\ e_1 \ldots e_n] \twoheadrightarrow \mathcal{C}[\bot] & \text{if } f \text{ is a defined operation} \end{array}$$

[2] Conditional rules are not considered in the reduction relation since they can be eliminated [4] by transforming each conditional rule "l `|` c `=` e" into "l `= cond` c e" where `cond` is defined by "`cond success x = x`".

The first rule models the call-time choice: if a rule is applied, the actual arguments of the operation must have been evaluated to partial values. The second rule models non-strictness where unevaluated operations are replaced by an undefined value (which is intended if the value of this subexpression is not demanded). A partial value t is called a *normal form* of e if $e \overset{*}{\twoheadrightarrow} t$. Note that the derivation for "`xorSelf aBool`" shown above is not possible w.r.t. $\twoheadrightarrow$. The equivalence of this rewrite relation and CRWL is shown in [42, 33].

We do not discuss the implementation of free (logic) variables in the following. This is justified by the fact that logic variables, denoting arbitrary but unknown values, can be replaced by generators, i.e., operations that non-deterministically evaluate to all possible ground values of the type of the free variable. For instance, the operation `aBool` is a generator for Boolean values so that one can transform the expression "`not x`", where `x` is a free variable, into "`not aBool`". It has been shown [9, 25] that computing with logic variables by narrowing [48, 51] and computing with generators by rewriting are equivalent, i.e., compute the same values. Since such generators are standard non-deterministic operations, they are translated like any other operation.

3 Compiling Non-Deterministic Programs

In this section, we sketch the implementation of non-deterministic programs in a purely functional language. This translation scheme is used by KiCS2 to compile Curry programs into Haskell programs. More details can be found in [16, 17, 19].

As mentioned in the introduction, we are interested in an implementation supporting different, in particular, complete search strategies. Thus, implementations based on a particular search strategy, like backtracking, which can also be found in approaches to support non-deterministic computations in functional programs [23, 39], are too limited. To provide various, also user-definable, search strategies, we explicitly represent all non-deterministic results of a computation in a data structure. This is achieved by extending each data type of the source program by a constructor to represent a choice between two values. For instance, the data type for Boolean values as defined above is translated into the Haskell data type[3]

```
data Bool = False | True | Choice ID Bool Bool
```

In order to implement the call-time choice semantics discussed in Sect. 2, each `Choice` constructor has an additional argument. For instance, the evaluation of `xorSelf aBool` duplicates the argument operation `aBool`. Thus, we have to ensure that both duplicates, which later evaluate to a non-deterministic choice between two values, yield either `True` or `False`. This is obtained by assigning a unique identifier (of type `ID`) to each `Choice`. In order to get unique identifiers on demand, we pass a (conceptually infinite) set of identifiers, also called *identifier supply*, to each operation.[4] Hence, each `Choice` created during run time can pick its unique identifier from this set. For this purpose, we assume a type `IDSupply`, representing an infinite set of identifiers, with operations

```
thisID     :: IDSupply → ID
leftSupply :: IDSupply → IDSupply
```

[3] Actually, our compiler adds also information to handle failures and performs some renamings to avoid conflicts with predefined Haskell entities by introducing type classes to resolve overloaded symbols like `Choice`.

[4] Note that the target program should be free of side effects in order to enable various search strategies, including parallel ones.

```
rightSupply :: IDSupply → IDSupply
```

`thisID` takes some identifier from this set, and `leftSupply` and `rightSupply` split this set into two disjoint subsets without the identifier obtained by `thisID`. There are different implementations available [14] so that KiCS2 is parametric over concrete implementations of `IDSupply`. A simple one can be based on unbounded integers, see [19].

Now, the correct handling of the call-time choice semantics can be obtained by adding an additional argument of type `IDSupply` to each operation. For instance, the operation `aBool` defined above is translated into:

```
aBool :: IDSupply → Bool
aBool s = Choice (thisID s) True False
```

Similarly, the operation

```
main :: Bool
main = xorSelf aBool
```

is translated into

```
main :: IDSupply → Bool
main s = xorSelf (aBool (leftSupply s)) (rightSupply s)
```

so that the set `s` is split into a set `(leftSupply s)` containing identifiers for the evaluation of `aBool` and a set `(rightSupply s)` containing identifiers for the evaluation of `xorSelf`.

Since all data types are extended by additional constructors, we must also extend the definition of operations performing pattern matching.[5] For instance, the operation `xor` is extended by an identifier supply and further matching rules:

```
xor :: Bool → Bool → IDSupply → Bool
xor True             x s = not x s
xor False            x s = x
xor (Choice i x1 x2) x s = Choice i (xor x1 x s) (xor x2 x s)
```

The third rule transforms a non-deterministic argument into a non-deterministic result, i.e., a non-deterministic choice is moved one level up. This is also called a “pull-tab” step [5].

In our concrete example, we assume that choice identifiers are implemented as integers [19]. Thus, if we evaluate the expression `(main 1)` w.r.t. the transformed rules defining `xor`, we obtain the result

```
Choice 2 (Choice 2 False True) (Choice 2 True False)
```

Hence, the result is non-deterministic and contains three choices with identical identifiers. To extract all values from such a `Choice` structure, we have to traverse it and compute all possible choices but consider the choice identifiers to make consistent (left/right) decisions. Thus, if we select the left branch as the value of the outermost `Choice`, we also have to select the left branch in the selected argument `(Choice 2 False True)` so that `False` is the only value possible for this branch. Similarly, if we select the right branch as the value of the outermost `Choice`, we also have to select the right branch in its selected argument `(Choice 2 True False)`, which again yields `False` as the only possible value. In consequence, the unintended value `True` cannot be extracted.

As one can see, the implementation is modularized in two phases that are interleaved by the lazy evaluation strategy of the target language: any expression is evaluated to a

[5] To obtain a simple compilation scheme, KiCS2 transforms source programs into uniform programs [19] where pattern matching is restricted to a single argument. This is always possible by introducing auxiliary operations.

tree representation of all its values and the main user interface (responsible for printing all results) extracts the correct values from this tree structure. As a consequence, one can easily implement various search strategies to extract these values as different tree traversal strategies. Due to the overall lazy evaluation strategy, infinite search spaces does not cause a complication. For instance, if one is interested only in a single solution, one can extract some value even if the computed choice structure is conceptually infinite.

4 Demand Analysis

The translation scheme presented in the previous section leads to an implementation with a good efficiency (e.g., the benchmarks presented in [19] show that it outperforms all other Curry implementations for deterministic operations, and, for non-deterministic operations, outperforms Prolog-based implementations of Curry and can compete with MCC [43], a Curry implementation that compiles to C). It is also used in a slightly modified form in another recent compact compiler for functional logic languages [13]. However, there are situations where this scheme cause efficiency problems. For instance, consider the evaluation of the expression `(main 1)` (for simplicity, we do not show the sharing of subexpressions done by the lazy evaluation strategy):

```
main 1 →* xorSelf (aBool 2) 3
       →* xor (aBool 2) (aBool 2) 3
       →* xor (Choice 2 True False) (Choice 2 True False) 3
       →* Choice 2 (Choice 2 False True) (Choice 2 True False)
```

As one can see, the (initially) single occurrence of the non-deterministic operation `aBool`, whose evaluation introduces a `Choice` constructor, is duplicated so that it results (in combination with the pull-tab step) in three `Choice` constructors. Since the overall strategy to extract values from choice structures has to traverse this choice structure, this might lead to an explosion of the search space in some cases (see benchmarks in Section 6).

A careful analysis shows that this problem stems from the lazy evaluation strategy. Hence, an improvement might be possible by changing the evaluation strategy. The operation `xorSelf` always demands the value of its argument in order to apply some reduction rule. Thus, one can also try to evaluate the argument *before* an attempt to evaluate `xorSelf`. Such a kind of call-by-value or strict evaluation can be achieved by introducing a *strict application* operation "`sApply`" implemented in the target code as follows:

```
sApply f (Choice i x1 x2) s = Choice i (sApply f x1 s) (sApply f x2 s)
sApply f x                s = f x s
```

Hence, `sApply` enforces the evaluation of the argument (to an expression without a defined operation at the top, also called *head normal form*) before the operation is applied. In particular, if the argument is a non-deterministic choice, it is moved outside the application. This operation is available as a predefined infix operation "`$!`" in Curry. Now consider what happens if we redefine `main` by

```
main = xorSelf $! aBool
```

and evaluate the translated `main` expression:

```
main 1 →* sApply xorSelf (aBool 2) 3
       →* sApply xorSelf (Choice 2 True False) 3
       →* Choice 2 (xorSelf True 3) (xorSelf False 3)
       →* Choice 2 (xor True True 3) (xor False False 3)
       →* Choice 2 False False
```

Hence, the computed choice structure does not contain duplicated `Choice` constructors, as desired. However, an unrestricted use of "`$!`" might destroy the completeness of the evaluation strategy. For instance, consider the definition

```
ok x = True
loop = loop
```

Then "`ok loop`" has the value `True` but the evaluation of "`ok $! loop`" does not terminate.

As a consequence, we need some information about the demand of operations in order to insert strict applications only for demanded arguments. This seems quite similar to strictness information in purely functional programming [46]. However, the techniques developed there cannot be applied to functional logic programs. For instance, consider the operation `f` defined by

```
f 0 = 0
f x = 1
```

As a functional program, `f` is strict since the first rule demands its argument. As a functional logic program, `f` does not strictly demand its argument: due to the non-deterministic semantics, all rules can be used to compute a result so that we can apply the second rule to evaluate `(f loop)` to the value `1`.

These considerations show that we need a notion of demand specific for functional logic programs. Using the rewrite relation $\twoheadrightarrow$ introduced above, we say that a unary operation[6] f *demands* its argument if $\bot$ is the only normal form of $(f\ \bot)$. Thus, if a demanded argument is not reducible to some expression with a constructor at the root, the application is always undefined. This justifies the use of the strict application operation "`$!`" to demanded arguments.

Hence, we are left with the problem of detecting demanded arguments in a program. Since this property is undecidable in general, we can try to approximate it by some program analysis. Early work on analyzing the behavior of functional logic programs [36, 45, 53] tried to approximate narrowing derivations for confluent term rewriting systems so that it is not applicable in our more general framework of non-deterministic operations. A more appropriate analysis can be based on a fixpoint characterization of CRWL rewriting [1, 44]. An analysis to approximate call patterns w.r.t. CRWL rewriting has been presented in [33]. Since the undefined value $\bot$ is a specific pattern, we can use a variant of this analysis to approximate demanded arguments. Thus, we summarize the main techniques and results of this analysis in the following.

Since we want to approximate the input/output relation of operations, an *interpretation* I is some set of equations

$$I = \{f\ t_1 \dots t_n) \doteq t \mid f\ n\text{-ary operation}, t_1, \dots, t_n, t \text{ are partial values}\}$$

The *evaluation of an expression e w.r.t. I* is a mapping $eval_I$ from expressions into sets of partial values defined by (where C and f denotes a constructor and an operation symbol, respectively):

$$\begin{aligned} eval_I(x) &= \{x\} \\ eval_I(C\ e_1 \dots e_n) &= \{C\ t_1 \dots t_n \mid t_i \in eval_I(e_i), i = 1, \dots, n\} \\ eval_I(f\ e_1 \dots e_n) &= \{\bot\} \cup \{t \mid t_i \in eval_I(e_i), i = 1, \dots, n, f\ t_1 \dots t_n \doteq t \in I\} \end{aligned}$$

[6] The extension to operations with more than one argument is straightforward.

Hence, an operation is approximated as undefined or evaluated with the information provided by the interpretation.

For the demand analysis, we are interested in the behavior of operations when they are called with undefined arguments. Thus, it is not necessary to compute the complete semantics of a program but it is sufficient to compute the behavior w.r.t. a given set of *initial calls* $\mathcal{M}$ containing elements of the form $f\ t_1 \ldots t_n$ where f is an operation and $t_1, \ldots, t_n$ are partial values. Then we define the transformation $T_\mathcal{M}$ on interpretations I by

$$\begin{aligned} T_\mathcal{M}(I) = \{s \doteq \bot \mid s \in \mathcal{M}\} \cup \{s \doteq r' \mid s \doteq t \in I,\ s = r \in [\mathcal{P}]_\bot,\ r' \in eval_I(r)\} \\ \cup \{f\ t_1 \ldots t_n \doteq \bot \mid s \doteq t \in I,\ s = r \in [\mathcal{P}]_\bot, f\ e_1 \ldots e_n \text{ is a subterm of } r, \\ t_i \in eval_I(e_i),\ i = 1, \ldots, n\} \end{aligned}$$

Intuitively, the transformation $T_\mathcal{M}$ adds to the set of initial calls in each iteration

1. better approximations of the rules' right-hand sides ($s \doteq r'$) and
2. new function calls occurring in right-hand sides ($f\ t_1 \ldots t_n \doteq \bot$).

Here, "better" should be interpreted w.r.t. the usual approximation ordering $\sqsubseteq$ where $\bot$ is the minimal element. As usual, we define

$$\begin{aligned} T_\mathcal{M} \uparrow 0 &= \emptyset \\ T_\mathcal{M} \uparrow k &= T_\mathcal{M}(T_\mathcal{M} \uparrow (k-1)) \quad (\text{for } k > 0) \end{aligned}$$

Since the mapping $T_\mathcal{M}$ is continuous on the set of all interpretations, the least fixpoint $C_\mathcal{M} = T_\mathcal{M} \uparrow \omega$ exists. The following theorem states the correctness of this fixpoint semantics w.r.t. CRWL rewriting.

▶ **Theorem 1** ([33]). *If $s \doteq t \in C_\mathcal{M}$, then $s \stackrel{*}{\twoheadrightarrow} t$. If $s \in \mathcal{M}$ and t is a partial value with $s \stackrel{*}{\twoheadrightarrow} t$, then $s \doteq t \in C_\mathcal{M}$.*

We call an equation $s \doteq t \in I$ *maximal* in I if there is no $s \doteq t' \in I$ with $t' \neq t$ and $t \sqsubseteq t'$. The set of all maximal elements of an interpretation I is denoted by $max(I)$. Maximal elements can be used to characterize a demanded argument, as the following result shows.

▶ Proposition 2. Let f be a unary operation and $f\ \bot \in \mathcal{M}$. If $f\ \bot \doteq \bot \in max(C_\mathcal{M})$, then f demands its argument.

The proposition suggests that one should analyze the least fixpoint w.r.t. a set of initial calls having $\bot$ at argument positions. In order to obtain a computable approximation of the least fixpoint, we use the theory of abstract interpretation [24] and define appropriate abstract domains and abstract operations (like abstract constructor application and abstract matching) to compute an abstract fixpoint.

Interesting finite abstractions of partial values are sets of terms up to a particular depth k, e.g., as already used in the abstract diagnosis of functional programs [2], abstraction of term rewriting systems [15], or call pattern analysis of functional logic programs [33]. Due to its quickly growing size, this domain is mainly useful in practice for depth $k = 1$. In the domain of depth-bounded terms, subterms that exceed the given depth k are replaced by the specific constant $\top$ that represents any term, i.e., the abstract domain of *depth-k terms* consists of partial values up to a depth k extended by the constant $\top$. For instance, `False:`$\top$ is a depth-2 term. If one defines abstract constructor applications (by applying the constructor and cutting subterms deeper than k) and an abstract matching of linear constructor terms against depth-k terms (see [33] for details), one can compute an abstract least fixpoint which approximates the least fixpoint of concrete computations.

For instance, consider the operations “`?`”, `not`, `xor`, and `xorSelf` defined above. In order to approximate their demanded arguments, we define a set of initial calls where one argument is $\bot$ and all other arguments are $\top$:

$$\mathcal{M} = \{\bot\texttt{?}\top,\ \top\texttt{?}\bot,\ \texttt{not}\ \bot,\ \texttt{xor}\ \bot\ \top,\ \texttt{xor}\ \top\ \bot,\ \texttt{xorSelf}\ \bot\}$$

Then the abstract least fixpoint w.r.t. $\mathcal{M}$ (note that the depth k is not relevant in this example) contains the following abstract equations:

$$\bot\texttt{?}\top \doteq \top,\ \top\texttt{?}\bot \doteq \top,\ \texttt{not}\ \bot \doteq \bot,\ \texttt{xor}\ \bot\ \bot \doteq \bot,\ \texttt{xor}\ \bot\ \top \doteq \bot,\ \texttt{xor}\ \top\ \bot \doteq \bot,$$
$$\texttt{xorSelf}\ \bot \doteq \bot$$

Since all these elements are also maximal, we can deduce by Proposition 2 that all arguments of `not`, `xor`, and `xorSelf` are demanded whereas “`?`” has no demanded argument. Of course, the analysis becomes more interesting in the case of recursive functions. We omit further examples here but refer to Section 6 for some benchmarks.

Our demand analysis can be extended in various ways. For instance, higher-order features can be covered by transforming higher-order applications into calls to an “apply” operation that implements the application of an arbitrary function occurring in the program to an expression [52]. This technique is also known as “defunctionalization” [49]. Primitive operations, like arithmetic functions, usually demand all their arguments. Thus, their behavior can be approximated by returning the result $\bot$ if some argument is $\bot$, and otherwise $\top$ is returned.

5 Program Transformation

We want to improve the non-determinism behavior of functional logic programs by transforming them according to the ideas sketched in the previous section. As already discussed, this can be done by adding strict applications to demanded arguments that are non-deterministic. A method to approximate demanded arguments has already been shown. The approximation of non-deterministic expressions is much simpler. For this purpose, we define an operation as *non-deterministic* if it contains a call to “`?`” or a free variable in some of its defining rules, or if it depends directly or indirectly on some non-deterministic operation. Thus, this property can be computed using the defining rules and their program dependency graph.

Based on this information, we can classify expressions: an expression is non-deterministic if it contains some non-deterministic operation. Now we perform the following transformation of the source program: if there is some application (f e) in some rule, where e is non-deterministic and the argument of f is demanded, replace this application by (f `$!` e). For instance, the program rule

```
main = xorSelf aBool
```

will be transformed into

```
main = xorSelf $! aBool
```

since the argument of `xorSelf` is demanded (as approximated above) and the argument `aBool` is non-deterministic. The extension of this transformation to operations with more than one argument is straightforward.

The effect of this transformation will be shown in the next section by some benchmarks.

Table 1 Benchmarks comparing original and optimized programs.

Benchmark	ViaLOIS (original)	KiCS2 (original)	KiCS2 (optimized)
`last2`	n/a	1.34	0.94
`last6`	n/a	2.72	0.94
`addNum2`	1.25	1.54	0.01
`addNum5`	22.08	8.58	0.01
`addPair`	1.36	1.54	0.01
`addTriple`	4.45	3.65	0.01
`half2`	2.18	3.78	1.44
`half5`	4.97	6.37	1.44
`dupList2`	n/a	3.34	0.11
`dupList5`	n/a	52.49	0.11
`select`	22.51	6.37	0.01
`queens`	n/a	36.62	1.26
`psort`	4.08	4.98	4.78

6 Benchmarks

We have implemented (in Curry) the program transformation shown above in a first prototype in order to get some ideas about its effectiveness. The program analyzer uses the depth-k domain to approximate demanded arguments. In order to provide an efficient analysis, only maximal abstract elements are stored in the current interpretation and the fixpoint iteration is done by an iteration using working lists. The non-determinism information is approximated in a separate analysis. The analysis results are used to guide the program transformation sketched above which produces the optimized Curry program.

Since our prototype does not support all features of Curry (e.g., no I/O), we have tested it only on smaller benchmark programs. Since our transformation is intended to improve non-deterministic programs, we have selected programs where non-deterministic operations occur as arguments.

The benchmarks were executed on a Linux machine running Linux (Ubuntu 11.10) with an Intel Core i5 (2.53GHz) processor. We omit the analysis times since they are less than 10 milliseconds for all presented examples. We tested two recent Curry implementations that are based on the idea to present non-deterministic values in a data structure: KiCS2 [19] with the Glasgow Haskell Compiler (GHC 7.0.3, option -O2) as its back end, and ViaLOIS [13] with the OCaml native-code compiler (version 3.12.0) as its back end. Table 1 shows the run times (in seconds) of a compiled executable for different programs. The programs `last2` and `last6` compute the last element of a list (of 10,000 elements) and add it two and six times to itself, respectively. `addNum2` and `addNum5` non-deterministically choose a number (out of 2000) and add it two and five times, respectively. Similarly, `addPair` and `addTriple` non-deterministically create a pair and triple of the same elements and add the components. `half2` and `half5` compute the half of a number n (here: 2000) by solving the equation `x+x=:=`n and add the result two and five times, respectively. `dupList2` and `dupList5` check a list `xs` (of 2000 elements) whether it is a duplicated list by solving the equation `ys+ys=:=xs` and concatenating `ys` two and five times, respectively. `select` non-deterministically selects an element in a list and returns the element and a list computed by deleting the selected element. `queens` computes all safe placements of eight queens on a

chessboard by enumerating all placements and non-deterministically checking whether two queens can attack each other. In this example, the duplication of choices stems from lazy pattern matching, as pointed out in [16, Sect. 6.9]. Finally, `psort` is the naive permutation sort applied to a list of 14 elements.

Since ViaLOIS is in an experimental state, it does not support all features of Curry (in particular, free variables of type integer are not supported) so that some benchmarks are not executable with ViaLOIS (marked by "n/a"). For the same reason, ViaLOIS does not support the primitive operation "`$!`" necessary for the optimization presented in this paper. Thus, the optimized programs are only executed with KiCS2. As one can see, the improvements obtained by our optimization are quite relevant for the considered class of programs. Only the improvement for `psort` is small since we cannot strictly evaluate the complete permutation.

7 Conclusions

We have shown a program transformation to improve the efficiency of non-deterministic computations in implementations of functional logic languages with a demand-driven strategy. If such implementations support a variety of search strategies, in particular, complete strategies, they often present the computation space in some tree structure which is explored by the search strategy [13, 19, 20]. This has the risk that non-deterministic structures are duplicated which increases the complexity of traversing the resulting structures. In order to overcome this disadvantage, we presented an analysis to approximate demanded arguments and use this information to evaluate non-deterministic arguments in a strict manner. We have also shown results from a prototypical implementation of this approach.

Since this work is based on techniques from various domains ranging from implementations of declarative languages to program analysis frameworks for such languages, there is a lot of related work. Since we already discussed related approaches throughout this paper, we omit a further discussion here. For future work, our demand analysis should be extended to enable the analysis of complete applications. This requires the appropriate approximation of all primitive operations, including I/O operations, and a modular analysis to be applied to larger programs. Furthermore, the use of other abstract domains, like rational trees, that can also approximate the demand of arbitrary large structures (e.g., lists) is another interesting topic fur future work.

References

1 J.M. Almendros-Jiménez and A. Becerra-Terón. A framework for goal-directed bottom-up evaluation of functional logic programs. In *Proc. of the 5th International Symposium on Functional and Logic Programming (FLOPS 2001)*, pages 153–169. Springer LNCS 2024, 2001.

2 M. Alpuente, M. Comini, S. Escobar, M. Falaschi, and S. Lucas. Abstract diagnosis of functional programs. In *Proc. of the 12th Int'l Workshop on Logic-Based Program Synthesis and Transformation (LOPSTR 2002)*, pages 1–16. Springer LNCS 2664, 2002.

3 M. Alpuente, M. Falaschi, and G. Vidal. Partial evaluation of functional logic programs. *ACM Transactions on Programming Languages and Systems*, 20(4):768–844, 1998.

4 S. Antoy. Constructor-based conditional narrowing. In *Proc. of the 3rd International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP 2001)*, pages 199–206. ACM Press, 2001.

5 S. Antoy. On the correctness of pull-tabbing. *Theory and Practice of Logic Programming*, 11(4-5):713–730, 2011.

6 S. Antoy, R. Echahed, and M. Hanus. A needed narrowing strategy. *Journal of the ACM*, 47(4):776–822, 2000.

7 S. Antoy and M. Hanus. Compiling multi-paradigm declarative programs into Prolog. In *Proc. International Workshop on Frontiers of Combining Systems (FroCoS'2000)*, pages 171–185. Springer LNCS 1794, 2000.

8 S. Antoy and M. Hanus. Functional logic design patterns. In *Proc. of the 6th International Symposium on Functional and Logic Programming (FLOPS 2002)*, pages 67–87. Springer LNCS 2441, 2002.

9 S. Antoy and M. Hanus. Overlapping rules and logic variables in functional logic programs. In *Proceedings of the 22nd International Conference on Logic Programming (ICLP 2006)*, pages 87–101. Springer LNCS 4079, 2006.

10 S. Antoy and M. Hanus. Functional logic programming. *Communications of the ACM*, 53(4):74–85, 2010.

11 S. Antoy and M. Hanus. New functional logic design patterns. In *Proc. of the 20th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2011)*, pages 19–34. Springer LNCS 6816, 2011.

12 S. Antoy, M. Hanus, J. Liu, and A. Tolmach. A virtual machine for functional logic computations. In *Proc. of the 16th International Workshop on Implementation and Application of Functional Languages (IFL 2004)*, pages 108–125. Springer LNCS 3474, 2005.

13 S. Antoy and A. Peters. Compiling a functional logic language: The basic scheme. In *Proc. of the Eleventh International Symposium on Functional and Logic Programming*, pages 17–31. Springer LNCS 7294, 2012.

14 L. Augustsson, M. Rittri, and D. Synek. On generating unique names. *Journal of Functional Programming*, 4(1):117–123, 1994.

15 D. Bert and R. Echahed. Abstraction of conditional term rewriting systems. In *Proc. of the 1995 International Logic Programming Symposium*, pages 147–161. MIT Press, 1995.

16 B. Braßel. *Implementing Functional Logic Programs by Translation into Purely Functional Programs*. PhD thesis, Christian-Albrechts-Universität zu Kiel, 2011.

17 B. Braßel and S. Fischer. From functional logic programs to purely functional programs preserving laziness. In *Proceedings of the 20th International Symposium on Implementation and Application of Functional Languages (IFL 2008)*, pages 25–42. Springer LNCS 5836, 2008.

18 B. Braßel, M. Hanus, and M. Müller. High-level database programming in Curry. In *Proc. of the Tenth International Symposium on Practical Aspects of Declarative Languages (PADL'08)*, pages 316–332. Springer LNCS 4902, 2008.

19 B. Braßel, M. Hanus, B. Peemöller, and F. Reck. KiCS2: A new compiler from Curry to Haskell. In *Proc. of the 20th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2011)*, pages 1–18. Springer LNCS 6816, 2011.

20 B. Braßel and F. Huch. On a tighter integration of functional and logic programming. In *Proc. APLAS 2007*, pages 122–138. Springer LNCS 4807, 2007.

21 B. Braßel and F. Huch. The Kiel Curry System KiCS. In *Applications of Declarative Programming and Knowledge Management*, pages 195–205. Springer LNAI 5437, 2009.

22 R. Caballero and F.J. López-Fraguas. A functional-logic perspective of parsing. In *Proc. 4th Fuji International Symposium on Functional and Logic Programming (FLOPS'99)*, pages 85–99. Springer LNCS 1722, 1999.

23 K. Claessen and P. Ljunglöf. Typed logical variables in Haskell. In *Proc. ACM SIGPLAN Haskell Workshop*, Montreal, 2000.

24 P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixpoints. In *Proc. of the 4th ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

25 J. de Dios Castro and F.J. López-Fraguas. Extra variables can be eliminated from functional logic programs. *Electronic Notes in Theoretical Computer Science*, 188:3–19, 2007.

26 S. Fischer. A functional logic database library. In *Proc. of the ACM SIGPLAN 2005 Workshop on Curry and Functional Logic Programming (WCFLP 2005)*, pages 54–59. ACM Press, 2005.

27 S. Fischer and H. Kuchen. Systematic generation of glass-box test cases for functional logic programs. In *Proceedings of the 9th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'07)*, pages 63–74. ACM Press, 2007.

28 J.C. González-Moreno, M.T. Hortalá-González, F.J. López-Fraguas, and M. Rodríguez-Artalejo. An approach to declarative programming based on a rewriting logic. *Journal of Logic Programming*, 40:47–87, 1999.

29 M. Hanus. A functional logic programming approach to graphical user interfaces. In *International Workshop on Practical Aspects of Declarative Languages (PADL'00)*, pages 47–62. Springer LNCS 1753, 2000.

30 M. Hanus. High-level server side web scripting in Curry. In *Proc. of the Third International Symposium on Practical Aspects of Declarative Languages (PADL'01)*, pages 76–92. Springer LNCS 1990, 2001.

31 M. Hanus. Type-oriented construction of web user interfaces. In *Proceedings of the 8th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'06)*, pages 27–38. ACM Press, 2006.

32 M. Hanus. Multi-paradigm declarative languages. In *Proceedings of the International Conference on Logic Programming (ICLP 2007)*, pages 45–75. Springer LNCS 4670, 2007.

33 M. Hanus. Call pattern analysis for functional logic programs. In *Proceedings of the 10th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'08)*, pages 67–78. ACM Press, 2008.

34 M. Hanus. Improving lazy non-deterministic computations by demand analysis. Technical report 1209, Christian-Albrechts-Universität Kiel, 2012.

35 M. Hanus and S. Koschnicke. An ER-based framework for declarative web programming. In *Proc. of the 12th International Symposium on Practical Aspects of Declarative Languages (PADL 2010)*, pages 201–216. Springer LNCS 5937, 2010.

36 M. Hanus and S. Lucas. A semantics for program analysis in narrowing-based functional logic languages. In *Proc. 4th Fuji International Symposium on Functional and Logic Programming (FLOPS'99)*, pages 353–368. Springer LNCS 1722, 1999.

37 M. Hanus and R. Sadre. An abstract machine for Curry and its concurrent implementation in Java. *Journal of Functional and Logic Programming*, 1999(6), 1999.

38 M. Hanus (ed.). Curry: An integrated functional logic language (vers. 0.8.2). Available at `http://www.curry-language.org`, 2006.

39 R. Hinze. Prolog's control constructs in a functional setting - axioms and implementation. *International Journal of Foundations of Computer Science*, 12(2):125–170, 2001.

40 H. Hussmann. Nondeterministic algebraic specifications and nonconfluent term rewriting. *Journal of Logic Programming*, 12:237–255, 1992.

41 F. López-Fraguas and J. Sánchez-Hernández. TOY: A multiparadigm declarative system. In *Proc. of RTA'99*, pages 244–247. Springer LNCS 1631, 1999.

42 F.J. López-Fraguas, J. Rodríguez-Hortalá, and J. Sánchez-Hernández. A simple rewrite notion for call-time choice semantics. In *Proceedings of the 9th ACM SIGPLAN Inter-*

national Conference on Principles and Practice of Declarative Programming (PPDP'07), pages 197–208. ACM Press, 2007.

43 W. Lux. Implementing encapsulated search for a lazy functional logic language. In *Proc. 4th Fuji International Symposium on Functional and Logic Programming (FLOPS'99)*, pages 100–113. Springer LNCS 1722, 1999.

44 J.M. Molina-Bravo and E. Pimentel. Modularity in functional-logic programming. In *Proc. of the Fourteenth International Conference on Logic Programming (ICLP'97)*, pages 183–197. MIT Press, 1997.

45 J.J. Moreno-Navarro, H. Kuchen, J. Mariño-Carballo, S. Winkler, and W. Hans. Efficient lazy narrowing using demandedness analysis. In *Proc. of the 5th International Symposium on Programming Language Implementation and Logic Programming*, pages 167–183. Springer LNCS 714, 1993.

46 A. Mycroft. The theory and practice of transforming call-by-need into call-by-value. In *Proc. International Symposium on Programming*, pages 269–281. Springer LNCS 83, 1980.

47 S. Peyton Jones, editor. *Haskell 98 Language and Libraries—The Revised Report*. Cambridge University Press, 2003.

48 U.S. Reddy. Narrowing as the operational semantics of functional languages. In *Proc. IEEE Internat. Symposium on Logic Programming*, pages 138–151, Boston, 1985.

49 J.C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM Annual Conference*, pages 717–740. ACM Press, 1972.

50 C. Runciman, M. Naylor, and F. Lindblad. Smallcheck and lazy smallcheck: automatic exhaustive testing for small values. In *Proc. of the 1st ACM SIGPLAN Symposium on Haskell*, pages 37–48. ACM Press, 2008.

51 J.R. Slagle. Automated theorem-proving for theories with simplifiers, commutativity, and associativity. *Journal of the ACM*, 21(4):622–642, 1974.

52 D.H.D. Warren. Higher-order extensions to Prolog: are they needed? In *Machine Intelligence 10*, pages 441–454, 1982.

53 F. Zartmann. Denotational abstract interpretation of functional logic programs. In *Proc. of the 4th International Symposium on Static Analysis (SAS'97)*, pages 141–156. Springer LNCS 1302, 1997.

The additional difficulties for the automatic synthesis of specifications posed by logic features in functional-logic languages*

Giovanni Bacci[1], Marco Comini[1], Marco A. Feliú[2], and Alicia Villanueva[2]

1 **Dipartimento di Matematica e Informatica**
University of Udine (Italy)
{giovanni.bacci,marco.comini}@uniud.it

2 **DSIC, Universitat Politècnica de València (Spain)**
{mfeliu,villanue}@dsic.upv.es

Abstract

This paper discusses on the additional issues for the automatic inference of algebraic property-oriented specifications which arises because of interaction between laziness and logical variables in lazy functional logic languages.

We present an inference technique that overcomes these issues for the first-order fragment of the lazy functional logic language Curry. Our technique statically infers from the source code of a Curry program a specification which consists of a set of equations relating (nested) operation calls that have the same behavior. Our proposal is a (glass-box) semantics-based inference method which can guarantee, to some extent, the correctness of the inferred specification, differently from other (black-box) approaches based on testing techniques.

1998 ACM Subject Classification F.3.1 Specifying and Verifying and Reasoning about Programs

Keywords and phrases Curry, property-oriented specifications, semantics-based inference methods

Digital Object Identifier 10.4230/LIPIcs.ICLP.2012.144

1 Introduction

Specifications have been widely used for several purposes: they can be used to aid (formal) verification, validation or testing, to instrument software development, as summaries in program understanding, as documentation of programs, to discover components in libraries or services in a network context, etc. [2, 16, 6, 12, 8, 19, 14, 9]. We can find several proposals of (automatic) inference of high-level specifications from an executable or the source code of a system, like [2, 6, 12, 9], which have proven to be very helpful.

There are many classifications in the literature depending on the characteristics of specifications [13]. It is common to distinguish between *property-oriented* specifications and *model-oriented* or *functional* specifications. Property-oriented specifications are of higher description level than other kinds of specifications: they consist in an indirect definition of the system's behavior by means of stating a set of properties, usually in the form of axioms,

* M. A. Feliú and A. Villanueva have been partially supported by the EU (FEDER), the Spanish MICINN/MINECO under grant TIN2010-21062-C02-02, the Spanish MEC FPU grant AP2008-00608, and by the Generalitat Valenciana, ref. PROMETEO2011/052.

© Giovanni Bacci, Marco Comini, Marco A. Feliú, and Alicia Villanueva;
licensed under Creative Commons License ND

Technical Communications of the 28th International Conference on Logic Programming (ICLP'12).
Editors: A. Dovier and V. Santos Costa; pp. 144–153

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

that the system must satisfy [18, 17]. In other words, a specification does not represent the functionality of the program (the output of the system) but its properties in terms of relations among the operations that can be invoked in the program (i.e., identifies different calls that have the same behavior when executed). This kind of specifications is particularly well suited for program understanding: the user can realize non-evident information about the behavior of a given function by observing its relation with other functions.

Clearly, the task of automatically inferring program specifications is in general undecidable and, given the complexity of the problem, there exists a large number of different proposals which impose several restrictions.

We can identify two mainstream approaches to perform the inference of specifications: glass-box and black-box. The glass-box approach [2, 6] assumes that the source code of the program is available. In this context, the goal of inferring a specification is mainly applied to document the code, or to understand it [6]. Therefore, the specification must be more succinct and comprehensible than the source code itself. The inferred specification can also be used to automatize the testing process of the program [6] or to verify that a given property holds [2]. The black-box approach [12, 9] works only by running the executable. This means that the only information used during the inference process is the input-output behavior of the program. In this setting, the inferred specification is often used to discover the functionality of the system (or services in a network) [9]. Although black-box approaches work without any restriction on the considered language – which is rarely the case in a glass-box approach – in general, they cannot *guarantee* the correctness of the results (whereas indeed semantics-based glass-box approaches can).

QuickSpec [6] is an (almost) black-box approach for Haskell programs [15] based on testing. The tool automatically infers program specifications as sets of equations of the form $e_1 = e_2$, where e_1, e_2 are generic program expressions that (should) have the same computational behavior. This approach has two properties that we like:

it is completely automatic as it needs only the program to run, plus some indications on target functions and generic values to be employed in equations, and

the outcomes are very intuitive since they are expressed *only* in terms of the program components, so the user does not need any kind of extra knowledge to interpret the results.

We aim to develop a method with similar outcomes for the lazy functional logic language Curry [10, 11]. Curry is a multi-paradigm programming language that combines in a seamless way features from functional programming (nested expressions, lazy evaluation, higher-order functions) and logic programming (logical variables, partial data structures, built-in search).

However, due to its very high-level features (in particular lazy evaluation and logical variables), the problem of inferring specifications for this kind of languages immediately poses several additional problems w.r.t. the functional paradigm (as well as other paradigms).

In this paper we discuss these issues in detail and we motivate why any proposal that aims to infer property-oriented specifications, like those of the QuickSpec approach, for lazy functional logic languages need to be radically different from the method used by QuickSpec.

2 Analysis of the issues posed by the logical features of Curry

Curry is a *lazy* functional *logic* language which admits free (logical) variables in expressions and program rules are evaluated non-deterministically.[1] Differently from the functional case

[1] Variables and function names start by a character in lower case, whereas data constructors and type names start by a letter in upper case. For a complete description of the Curry language, the interested

(of QuickSpec), due to the logical features, an equation $e_1 = e_2$ can be interpreted in many different ways. We will discuss the key points of the problem by means of a (very simple) illustrative example.

▶ **Example 1** (Boolean logic example). Consider the definition for the boolean data type with values `True` and `False` and boolean operations `and`, `or`, `not` and `imp`:

```
and True x = x
and False _ = False
or True _ = True
or False x = x
```

```
not True = False
not False = True
imp False x = True
imp True x = x
```

This is a pretty standard "short-cut" definition of boolean connectives. For example, the definition of `and` states that whenever the first argument is equal to `False`, the function returns the value `False`, regardless of the value of the second argument. Since the language is lazy, in this case the second argument will not be evaluated.

For the program of Example 1, one could expect to have in its property-oriented specification equations like

$$\texttt{imp x y} = \texttt{or (not x) y} \qquad \texttt{not (and x y)} = \texttt{or (not x) (not y)} \tag{2.1}$$

$$\texttt{and x (and y z)} = \texttt{and (and x y) z} \qquad \texttt{not (or x y)} = \texttt{and (not x) (not y)} \tag{2.2}$$

$$\texttt{and x y} = \texttt{and y x} \tag{2.3}$$

$$\texttt{not (not x)} = \texttt{x} \tag{2.4}$$

which are well-known laws among the (theoretical) boolean operators. These equations, of the form $e_1 = e_2$, can be read as

all possible outcomes for e_1 *are also outcomes for* e_2, and vice versa. (2.5)

In the following, we call this equivalence *computed result equivalence* and we denote it by $=_{CR}$. Actually, Equations (2.1), (2.2) and (2.3) are *literally* valid in this sense since, in Curry, free variables are admitted in expressions, and the mentioned equations are valid *as they are*. This is quite different from the pure functional case where equations *have to be interpreted* as properties that hold for any *ground* instance of the variables occurring in the equation.

On the contrary, Equation (2.4) is not *literally* valid since the goal `not (not x)` is evaluated to $\{\texttt{x/True}\} \cdot \texttt{True}$[2] and $\{\texttt{x/False}\} \cdot \texttt{False}$, whereas `x` is evaluated just to $\{\} \cdot \texttt{x}$. Note however that any ground instance of the two goals evaluates to the same results, namely both `True` and `not (not True)` are evaluated to $\{\} \cdot \texttt{True}$, and both `False` and `not (not False)` are evaluated to $\{\} \cdot \texttt{False}$.

Decidedly, also this notion of *ground equivalence* is interesting for the user, and we denote it by $=_G$. This notion coincides with the (only possible) one used in the pure functional paradigm: two terms are ground equivalent if, for all ground instances, the outcomes of both terms coincide.

Because of the presence of logical variables, there is another very relevant difference w.r.t. the pure functional case concerned with *contextual equivalence*: given a valid equation

reader can consult [11].

[2] The expression $\{\texttt{x/True}\} \cdot \texttt{True}$ denotes that the normal form `True` has been reached with computed answer substitution $\{\texttt{x/True}\}$.

$e_1 = e_2$, is it true that, for any context C, the equation $C[e_1] = C[e_2]$ still holds? Curry is not referentially transparent[3] w.r.t. its operational behavior, i.e., an expression can produce different computed values when it is embedded in a context that binds its free variables (as shown by the following artificial example), which makes the answer to the question posed above not straightforward.

▶ **Example 2.** Given a program with the following rules

```
g x = C (h x)
h A = A
```

```
g’ A = C A
f (C x) B = B
```

the expressions `g x` and `g’ x` compute the same result, namely $\{$`x/A`$\} \cdot$ `C A`. However, the expression `f (g x) x` computes one result, namely $\{$`x/B`$\} \cdot$ `B`, while expression `f (g’ x) x` computes none.

Thus, in the Curry case, it becomes mandatory to *additionally* ask in the equivalence notion of (2.5) that the outcomes must be equal also when the two terms are embedded within any context. We call this equivalence *contextual equivalence* and we denote it by $=_C$. Actually, Equations (2.1) and (2.2) are valid w.r.t. this equivalence notion.

We can see that $=_C$ is (obviously) stronger than $=_{CR}$, which is in turn stronger than $=_G$. As a conclusion, for our example we would get the following (partial) specification.

`not (or x y)` $=_C$ `and (not x) (not y)`
`not (and x y)` $=_C$ `or (not x) (not y)`
`and x (and y z)` $=_C$ `and (and x y) z`
`imp x y` $=_C$ `or (not x) y`
`not (not x)` $=_G$ `x`
`and x y` $=_G$ `and y x`

The inference of $=_C$ equalities poses serious issues to testing-based methods like QuickSpec. First, expressions have to be nested within some outer context in order to establish their $=_C$ equivalence. Since the number of needed terms to be evaluated grows exponentially w.r.t. the depth of nestings, the addition of a further level of depth can dramatically alter the performance. Moreover, if we try to mitigate this problem by reducing the number of terms/tests to be checked, the quality of the produced equations will degrade sensibly. Second, since the typical real life case is that the program in consideration is just a module of a complex software system, it may happen that no function in the considered module can discriminate contexts; but in other modules there could be one. Clearly, we could imagine to run the tool on the entire system but, besides the obvious increment of cost, the “caller” module could have not been implemented yet. Thus, we would need to reconsider the outputs of synthesis whence some new module is added.

This kind of issues do not arise with languages, like Haskell, which are referentially transparent: essentially, languages where the semantics of all nested expressions can be obtained by instantiating the semantics of the outer expression with those of the arguments.

Contrary to testing-based approaches, a semantics-based method that computes the (compositional) semantics of a part of code does not suffer of these issues[4]. Obviously, in

[3] The concept of referential transparency of a language can be stated in terms of a formal semantics as: the semantics equivalence of two expressions e, e' implies the semantics equivalence of e and e' when used within any context $C[\cdot]$. Namely, $\forall e, e', C.\ [\![e]\!] = [\![e']\!] \implies [\![C[e]]\!] = [\![C[e']]\!]$.

[4] Evidently, the semantics to be employed needs to be fully abstract w.r.t. contextual embedding in order to compute correct $=_C$ equations.

this case the problem is the undecidability of the semantics' computation, thus suitable approximations must be used. This would lead to possibly erroneous equations, but this also happens with testing-based approaches.

Since Curry is *not* referentially transparent, we do not consider the semantics-based approach an option, but a must. In the following we present a first proposal of a semantics-based method that tackles the presented issues and discuss about its limitations.

3 Formalization of equivalence notions

In this section, we formally present all the kinds of term equivalence notions that are used to compute equations of the specification. We need first to introduce some basic formal notions that are used in the rest of the paper.

We say that a first order Curry program is a set of rules P built over a signature Σ partitioned in $\mathcal{C}$, the *constructor* symbols, and $\mathcal{D}$, the *defined* symbols. $\mathcal{V}$ denotes a (fixed) countably infinite set of variables and $\mathcal{T}(\Sigma, \mathcal{V})$ denotes the terms built over signature Σ and variables $\mathcal{V}$. A *fresh* variable is a variable that appears nowhere else.

In order to state formally the equivalences described before we need two semantic evaluation functions $\mathcal{E}^{C}[\![]\!]$ and $\mathcal{E}^{CR}[\![]\!]$ which enjoy some properties.

$\mathcal{E}^{CR}[\![t;\, P]\!]$ gives the *computed results* (CR) semantics of the term t with (definitions from) the program P. This semantics has to be fully abstract w.r.t. the behavior of computed results. Namely, the semantics of two terms t_1, t_2 are identical if and only if the evaluations of t_1 and t_2 compute the same results. It is theoretically possible to use just a correct semantics, but clearly in such case we will not have all equivalences which are valid w.r.t. a fully abstract semantics.

$\mathcal{E}^{C}[\![t;\, P]\!]$ gives the *contextual* (C) semantics of the term t with the program P. This semantics has to be fully abstract w.r.t. the behavior of computed results *under any context*. Namely, the semantics of two terms t_1, t_2 are identical if and only if, for any context C, the evaluations of $C[t_1]$ and $C[t_2]$ compute the same results. We say that such a semantics fulfills referential transparency.

The semantics which can be obtained by collecting all results of the official small-step operational semantics of Curry [11, App. D.5], as well as the I/O semantics of [1], can be used for $\mathcal{E}^{CR}[\![t;\, P]\!]$ but they are not referentially transparent. On the contrary, the full small-step operational semantics of Curry is referentially transparent but is far from being fully-abstract.

The $\mathbb{WERS}$-semantics of [3, 4] is an appropriate choice for $\mathcal{E}^{C}[\![t;\, P]\!]$ and moreover the set of its leaves is an appropriate choice for $\mathcal{E}^{CR}[\![t;\, P]\!]$. However, our proposal does not rely on these particular semantics and any semantics which fulfills the aforementioned requirements can be used.

Now we are ready to formally introduce our notion of specification. An (algebraic) specification $\mathcal{S}$ is a set of (sequences of) equations of the form $t_1 =_{K} t_2 =_{K} \ldots =_{K} t_n$, with $K \in \{C, CR, G\}$ and $t_1, t_2, \ldots, t_n \in \mathcal{T}(\Sigma, \mathcal{V})$. K distinguishes the kinds of computational equalities that we previously informally discussed, which we now present formally.

Contextual Equivalence $=_{C}$. States that two terms t_1 and t_2 are equivalent if $C[t_1]$ and $C[t_2]$ have the same behavior for any context $C[\cdot]$. Namely,

$$t_1 =_{C} t_2 \iff \mathcal{E}^{C}[\![t_1;\, P]\!] = \mathcal{E}^{C}[\![t_2;\, P]\!]$$

Computed-result equivalence $=_{CR}$. This equivalence states that two terms are equivalent when the outcomes of their evaluation are the same. Namely

$$t_1 =_{CR} t_2 \iff \mathcal{E}^{CR}[\![t_1;\, P]\!] = \mathcal{E}^{CR}[\![t_2;\, P]\!]$$

The $=_{CR}$ equivalence is coarser than $=_{C}$ ($=_{C} \subseteq =_{CR}$) as shown by Example 2.

Ground Equivalence $=_{G}$. This equivalence states that two terms are equivalent if all possible ground instances have the same outcomes. Namely

$$t_1 =_{G} t_2 \iff \forall\theta \,\text{grounding.}\; \mathcal{E}^{CR}[\![t_1\theta;\, P]\!] = \mathcal{E}^{CR}[\![t_2\theta;\, P]\!]$$

By definition, the $=_{G}$ equivalence is coarser than $=_{CR}$ ($=_{CR} \subseteq =_{G}$).

User Defined Equality Equations. The symbol $=_{Ueq}$ is used for *user-defined equality equations*. Equality equations depend upon a user-defined notion of equality. When dealing with a user-defined data type, the user may have defined a specific notion of equivalence by means of an "equality" function. Let us call $equal(t_1, t_2)$ such user-defined function. Then, we state that

$$t_1 =_{Ueq} t_2 \iff \mathcal{E}^{CR}[\![equal(t_1, t_2);\, P]\!] = \mathcal{E}^{CR}[\![\texttt{True}]\!] \iff equal(t_1, t_2) =_{CR} \texttt{True}$$

Clearly, we do not have necessarily any relation between $=_{Ueq}$ and the others, as the user function *equal* may have nothing to do with equality. However, in typical situations such a function is defined to preserve at least $=_{G}$, meaning that $t_1 =_{G} t_2$ implies $t_1 =_{Ueq} t_2$.
In any case, as clear from the definition, this is technically just a particular instance of $=_{CR}$, so it does not need to be considered by itself and in the following we will not consider it explicitly.
Nevertheless, these equations can provide the user significant information about the structure and behavior of the program and a pragmatical tool should reasonably present a sequence `True` $=_{CR} equal(t_1, t_2) =_{CR} \ldots =_{CR} equal(t_{n-1}, t_n)$ as $t_1 =_{Ueq} \ldots =_{Ueq} t_n$ for readability purposes.

Note that $=_{G}$ is the only possible notion in the pure functional paradigm. This fact allows one to have an intuition of the reason why the problem of specification synthesis is more complex in the functional logic paradigm.

To summarize, we have $=_{C} \subseteq =_{CR} \subseteq =_{G}$ and only $=_{C}$ is referentially transparent (i.e., a congruence w.r.t. contextual embedding).

4 Deriving Specifications from Programs

The idea underlying the process of inferring specifications is that of computing the semantics of various terms and then identify all terms which have the same semantics. However, not all equivalences are as important as others, given the fact that many equivalences are simply consequences of others. For example, if $t_i =_{C} s_i$ then, for all contexts C, $C[t_1, \ldots, t_n] =_{C} C[s_1, \ldots, s_n]$, thus the latter *derived* equivalences are uninteresting and should be omitted. Indeed, it would be desirable to synthesize the *minimal* set of equations from which, by deduction, all equalities can be derived. This is certainly a complex issue in testing approaches (it is certainly a big part in the QuickSpec method). With a semantics-based approach it is fairly natural to produce just the relevant equations. The idea is to proceed bottom-up, by starting from the evaluation of simpler terms and then newer terms are constructed (and evaluated) by using only semantically different arguments.

There is also another source of redundancy due to the inclusion of relations $=_{K}$. For example, since $=_{C}$ is the finer relation, if $t =_{C} s$ then $t =_{CR} s$ and $t =_{G} s$. To avoid

the generation of coarser redundant equations, a simple solution is that of starting with $=_C$ equivalences and, once these are all settled, to proceed with the evaluation of the CR semantics *only* of non $=_C$ equivalent terms. Thereafter, we can evaluate the ground semantics of non $=_{CR}$ equivalent terms.

Let us describe in more detail the specification inference process. The input of the process consists of the Curry program to be analyzed and two additional parameters: a *relevant* API, Σ^r, and a maximum term size, *max_size*. The *relevant* API allows the user to choose the operations in the program that will be present in the inferred specification, whereas the maximum term size limits the size of the terms in the specification. The inference process consists of three phases, one for each kind of equality: first $=_C$ and then $=_{CR}$ and $=_G$. Terms are classified by their semantics into a data structure, which we call *classification*, consisting of a set of *equivalence classes* (*ec*) formed by

- $sem(ec)$: the semantics of (all) the terms in that class;
- $terms(ec)$: the set of terms belonging to that equivalence class;
- $rep(ec)$: the *representative term* of the class ($rep(ec) \in terms(ec)$).

The *representative term* is the term which is used in the construction of nested expressions when the equivalence class is considered. To output smaller equations it is better to choose the smallest term in the class (w.r.t. the function *size*), but any element of $terms(ec)$ can be used.

For the sake of comprehension, we present an untyped version of the method.

Computation of the initial classification

We initially create a classification which contains:

- one class for a free (logical) variable $\langle \mathcal{E}[\![x]\!],\, x,\, \{x\}\rangle$ [5];
- the classes for any built-in or user-defined constructor.

Then, for all symbols f/n of the relevant API, Σ^r, and distinct variables $x_1, \ldots, x_n$, we *add to classification* the term $t = f(x_1, \ldots, x_n)$ with semantics $s = \mathcal{E}^C[\![t;\, P]\!]$. This operation looks for an equivalence class *ec* in the current classification whose semantics coincides with s. If it is found, then the term t is added to the set of terms in *ec*. Otherwise a new equivalence class $\langle s,\, t,\, \{t\}\rangle$ is created.

Generation of $=_C$ classification

We iteratively select all symbols f/n of the relevant API Σ^r and n equivalence classes $ec_1, \ldots, ec_n$ from the current classification. We build the term $t = f(t_1, \ldots, t_n)$, where each t_i is the representative term of ec_i, $t_i = rep(ec_i)$; then, we compute the semantics $s = \mathcal{E}^C[\![t;\, P]\!]$ and update the current classification by *adding to classification* t and s as described before.

If the classification changes, then we iterate by considering again all the symbols in the relevant API to build and evaluate new (maybe greater) terms. This phase is doomed to terminate because at each iteration we consider, by construction, terms which are different from those already existing in the classification and whose size is strictly greater than the size of its subterms (but the size is bounded by *max_size*).

Let us show an example:

---

[5] The typed version uses one variable for each type

▶ **Example 3.** Let us recall the program of Example 1 and choose as relevant API the functions `and`, `or` and `not`. In the first iteration, the terms $t_{1.1}$ = `not x`, $t_{1.2}$ = `and x y`, and $t_{1.3}$ = `or x y` are built. After computing the semantics, and since the semantics of none of them coincides with the semantics of an existing equivalence class, three new equivalence classes appear, one for each term.

During the second iteration, the following two terms (among others) are built: the term $t_{2.1}$ = `and (not x) (not x')` is built as the instantiation of $t_{1.2}$ with $t_{1.1}$, and the term $t_{2.2}$ = `not (or x y)` as the instantiation of $t_{1.1}$ with $t_{1.3}$. The semantics of these two terms are the same, but it is different from the semantics of the existing equivalence classes, thus during this iteration (at least) this new class is computed. From this point on, only the representative of the class will be used for constructing new terms. We choose the smaller term as the representative, which in the example is $t_{2.2}$ ($rep(ec_1) = t_{2.2}$), thus terms like `not (and (not x) (not x'))` will never be built.

Generation of the $=_C$ specification

Since, by construction, we have avoided much redundancy thanks to the strategy used to generate the equivalence classes, we now have only to take each equivalence class with more than one term and generate equations for these terms.

Generation of $=_{CR}$ equations

The second phase works on the former classification by first transforming each equivalence class ec by replacing the C-semantics $sem(ec)$ with $\mathcal{E}^{CR}[\![rep(ec); P]\!]$ and $terms(ec)$ with the (singleton) set $\{rep(ec)\}$.

After the transformation, some of the equivalence classes which had different semantic values may now have the same CR-semantics and then we merge them, making the union of the term sets $terms(ec)$.

Thanks to the fact that, before merging, all equivalence classes were made of just singleton term sets, we cannot generate (again) equations $t_1 =_{CR} t_2$ when an equation $t_1 =_C t_2$ had been already issued.

Let us clarify this phase by an example.

▶ **Example 4.** Assume we have a classification consisting of three equivalence classes with C-semantics s_1, s_2 and s_3 and representative terms t_{11}, t_{22} and t_{31}:

$$ec_1 = \langle s_1, t_{11}, \{t_{11}, t_{12}, t_{13}\}\rangle \qquad ec_2 = \langle s_2, t_{22}, \{t_{21}, t_{22}\}\rangle \qquad ec_3 = \langle s_3, t_{31}, \{t_{31}\}\rangle$$

We generate equations $t_{11} =_C t_{12} =_C t_{13}$ and $t_{21} =_C t_{22}$.

Now, assume that $\mathcal{E}^{CR}[\![t_{11}]\!] = x_0$ and $\mathcal{E}^{CR}[\![t_{22}]\!] = \mathcal{E}^{CR}[\![t_{31}]\!] = x_1$. Then (since t_{12}, t_{13} and t_{21} are removed) we obtain the new classification

$$ec_4 = \langle x_0,\ t_{11},\ \{t_{11}\}\rangle \qquad ec_5 = \langle x_1,\ t_{22},\ \{t_{22}, t_{31}\}\rangle$$

Hence, the only new equation is $t_{22} =_{CR} t_{31}$. Indeed, equation $t_{11} =_{CR} t_{12}$ is uninteresting, since we already know $t_{11} =_C t_{12}$ and equation $t_{21} =_{CR} t_{31}$ is redundant (because $t_{21} =_C t_{22}$ and $t_{22} =_{CR} t_{31}$).

Successive (sub-)phases

The resulting (coarser) classification is then used to produce the $=_{CR}$ equations, as done before, by generating equations for all non-singletons term sets. In the last phase, we

transform again the classification by replacing the *CR*-semantics with the ground semantics (and term sets with singleton term sets). Then we merge eventual equivalence classes with the same semantics and, finally, we generate $=_G$ equations for non singleton sets.

4.1 Feasibility considerations

In a semantics-based approach, one of the main problems to be tackled is effectiveness. The semantics of a program is in general infinite and thus some approximation has to be used in order to have a terminating method.

Several solutions can be adopted. To experiment on the validity of our proposal we have implemented the basic functionality of this methodology in a prototype, written in Haskell, available at `http://safe-tools.dsic.upv.es/absspec`. The computation of $\mathcal{E}^{C}[\![]\!]$ is based on an implementation of the immediate consequences operator $\mathcal{P}^{\nu}[\![P]\!]$ of the (fixpoint) $\mathbb{WERS}$-semantics of [3, 4]. To achieve termination, the prototype computes a fixed number of steps of $\mathcal{P}^{\nu}[\![P]\!]$. Then, it proceeds with the classification as described with a further enhancement which is possible due to properties of the $\mathbb{WERS}$-semantics. Namely, the semantics $\mathcal{E}^{CR}[\![]\!]$ can be obtained directly by transforming the $\mathcal{E}^{C}[\![]\!]$ semantics, concretely just by loosing internal structure. Therefore, no (costly) computation of $\mathcal{E}^{CR}[\![]\!]$ is needed, but just a quick filtering. The implementation of $=_G$ equalities is still ongoing work.

We are aware that many other attempts to guarantee termination could be used. Certainly, given our know-how, in the future we will experiment with abstractions obtained by abstract interpretation [7] (the $\mathbb{WERS}$-semantics itself has been obtained as an abstract interpretation).

5 Conclusions and Future Work

This paper discusses about the issues that arise for the automatic inference of high-level, property-oriented (algebraic) specifications because of the presence of logical features in functional-logic languages. Then, a first preliminary proposal which overcomes these issues is presented. To the best of our knowledge, in the functional logic setting there are currently no proposals for specification synthesis. There is a testing tool, EasyCheck [5], in which specifications are *used* as the input for the testing process. Given the properties, EasyCheck executes ground tests in order to check whether the property holds.

Our method computes a concise specification of program properties from the source code of the program. We hope to have convinced the reader that we reached our main goal, that is, to get a concise and clear specification that is useful for the programmer in order to detect possible errors, or to check program's correctness.

We have developed a prototype that implements the basic functionality of the approach. We are working on the inclusion of all the functionalities described in this paper.

It would be interesting in the future, once our proposal is mature, to investigate on the appropriateness also for referentially transparent languages like Haskell.

References

1 E. Albert, M. Hanus, F. Huch, J. Oliver, and G. Vidal. Operational Semantics for Declarative Multi-Paradigm Languages. *Journal of Symbolic Computation*, 40(1):795–829, 2005.

2 G. Ammons, R. Bodík, and J. R. Larus. Mining specifications. In *29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL'02)*, pages 4–16, New York, NY, USA, 2002. Acm.

3 G. Bacci. *An Abstract Interpretation Framework for Semantics and Diagnosis of Lazy Functional-Logic Languages.* PhD thesis, Dipartimento di matematica e Informatica, Università di Udine, 2011.

4 G. Bacci and M. Comini. A Compact Goal-Independent Bottom-Up Fixpoint Modeling of the Behaviour of First Order Curry. Technical Report DIMI-UD/06/2010/RR, Dipartimento di Matematica e Informatica, Università di Udine, 2010.

5 J. Christiansen and S. Fischer. Easycheck – test data for free. In *Proceedings of the 9th International Symposium on Functional and Logic Programming (FLOPS'08)*, volume 4989 of *Lecture Notes in Computer Science*, pages 322–336. Springer, 2008.

6 K. Claessen, N. Smallbone, and J. Hughes. QuickSpec: Guessing Formal Specifications using Testing. In *4th International Conference on Tests and Proofs (TAP 2010)*, volume 6143, pages 6–21, 2010.

7 P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, Los Angeles, California, January 17–19*, pages 238–252, New York, NY, USA, 1977. ACM Press.

8 C. Ghezzi and A. Mocci. Behavior model based component search: an initial assessment. In *Proceedings of 2010 ICSE Workshop on Search-driven Development: Users, Infrastructure, Tools and Evaluation (SUITE'10)*, pages 9–12, New York, NY, USA, 2010. Acm.

9 C. Ghezzi, A. Mocci, and M. Monga. Synthesizing intensional behavior models by graph transformation. In *31st International Conference on Software Engineering (ICSE'09)*, pages 430–440, 2009.

10 M. Hanus. A unified computation model for functional and logic programming. In *24th ACM Symposium on Principles of Programming Languages (POPL 97)*, pages 80–93, 1997.

11 M. Hanus. Curry: An integrated functional logic language (vers. 0.8.2), 2006. Available at URL: `http://www.informatik.uni-kiel.de/~curry`.

12 J. Henkel, C. Reichenbach, and A. Diwan. Discovering documentation for java container classes. *IEEE Transactions on Software Engineering*, 33(8):526–542, 2007.

13 A. A. Khwaja and J. E. Urban. A property based specification formalism classification. *The Journal of Systems and Software*, 83:2344–2362, 2010.

14 I. Nunes, A. Lopes, and V. Vasconcelos. Bridging the Gap between Algebraic Specification and Object-Oriented Generic Programming. In Saddek Bensalem and Doron Peled, editors, *9th International Workshop on Runtime Verification (RV 2009)*, volume 5779 of *Lecture Notes in Computer Science*, pages 115–131. Springer, 2009.

15 S. Peyton Jones. *Haskell 98 Language and Libraries - The Revised Report.* Cambridge University Press, Cambridge, UK, 2003. Available at `http://www.haskell.org/definition/`.

16 D. Rayside, A. Milicevic, K. Yessenov, G. Dennis, and D. Jackson. Agile specifications. In *Companion to the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2009)*, pages 999–1006. Acm, 2009.

17 H. van Vliet. *Software Engineering–Principles and Practice.* John Wiley, 1993.

18 J. M. Wing. A specifier's introduction to formal methods. *Computer*, 23(9):10–24, 1990.

19 B. Yu, L. Kong, Y. Zhang, and H. Zhu. Testing Java Components based on Algebraic Specifications. In *First International Conference on Software Testing, Verification, and Validation (ICST 2008)*, pages 190–199. IEEE Computer Society, 2008.

A Concurrent Operational Semantics for Constraint Functional Logic Programming *

Rafael del Vado Vírseda, Fernando Pérez Morente, and Marcos Miguel García Toledo

Departamento de Sistemas Informáticos y Computación, Universidad Complutense de Madrid
C. Profesor José García Santesmases, s/n. 28040 Madrid, Spain
rdelvado@sip.ucm.es fperezmo@fdi.ucm.es mmgarciat@fdi.ucm.es

Abstract

In this paper we describe a sound and complete concurrent operational semantics for constraint functional logic programming languages which allows to model declarative applications in which the interaction between demand-driven narrowing and constraint solving helps to prune the search space, leading to shorter goal derivations. We encode concurrency into the generic $CFLP(\mathcal{D})$ scheme, a uniform foundation for the operational semantics of constraint functional logic programming systems parameterized by a constraint solver over the given domain $\mathcal{D}$. In this concurrent version of the $CFLP(\mathcal{D})$ scheme, goal solving processes can be executed concurrently and cooperate together to perform their specific tasks via demand-driven narrowing and declarative residuation guided by constrained definitional trees, constraint solving, and communication by synchronization on logical variables.

1998 ACM Subject Classification D.1.6 Logic Programming; D.3.3 Language Constructs and Features: Constraints, functions and concurrent programming structures.

Keywords and phrases Constraint logic programming, concurrent logic programming, functional logic programming.

Digital Object Identifier 10.4230/LIPIcs.ICLP.2012.154

1 Introduction

Multiparadigm logic programming languages and systems [2, 6, 8, 11] aim to integrate the most important declarative programming paradigms, namely *functional programming* (demand-driven rewriting strategies, higher-order facilities, etc.) and *(constraint) logic programming* (goal solving, logical variables, computation with constraints, etc.). The endeavor to extend this declarative combined logic paradigm to a practical language suitable for concurrent executions has stimulated much research over the last two decades, resulting in a large variety of proposals [3, 6, 8]. The aim of this research area is the development of *concurrent functional and constraint logic programming* systems [2, 8] that maintain the balance between expressiveness and declarative reading: abstraction, computations as proofs, amenability to meta-programming, etc. However, the interactions between all these different features are complex, so the design and implementation of a sound and complete theoretical framework of concurrent and constrained multiparadigm logic programming systems is non-trivial.
A common feature of the various approaches is the attempt to define declarative operational models for concurrency within the *Constraint Logic Programming scheme* $CLP(\mathcal{D})$ [7], which

* This work has been partially supported by the Spanish projects STAMP (TIN2008-06622-C03-01), Prometidos-CM (S2009TIC-1465), and GPD (UCM-BSCH-GR35/10-A-910502).

© Rafael del Vado Vírseda, Fernando Pérez Morente, and Marcos Miguel García Toledo;
licensed under Creative Commons License ND
Technical Communications of the 28th International Conference on Logic Programming (ICLP'12).
Editors: A. Dovier and V. Santos Costa; pp. 154–163

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

replaces the basic computational model of logic programming (i.e., *SLD-resolution* with syntactic *unification*) by *constraint solving* over some *constraint domain* $\mathcal{D}$ (e.g., the integer or the real numbers). The *CLP*($\mathcal{D}$) scheme can be generalized into the framework of *concurrent constraint programming* [10] to accommodate a simple and powerful model of declarative concurrent computation based on a *global store*, represented by a constraint on the values that variables can assume. All goal solving processes of the system share this common store, and instead of "reading" and "writing" the values of variables, processes may *ask* (check if a constraint is entailed by the store) and *tell* (augment the store with a new constraint).

The *CFLP*($\mathcal{D}$) scheme [9] elegantly captures the fundamental ideas behind the multiparadigm logic systems, generalizing the *CLP*($\mathcal{D}$) scheme to provide uniform foundations for the semantics of functional and constraint logic programming languages. The efficient operational semantics relies on *demand-driven narrowing with definitional trees* [12], a combination of syntactic unification and demand-driven rewriting, parameterized by a constraint solver over the given domain $\mathcal{D}$, which is sound and complete with respect to a declarative semantics formalized by a *constraint rewriting logic* [9], and uses a hierarchical structure called *definitional tree* to efficiently control the computation. The current version of the constraint functional logic system $\mathcal{TOY}$ [11] has been designed to efficiently implement the *CFLP*($\mathcal{D}$) scheme. However, concurrency is not supported in the *CFLP*($\mathcal{D}$) execution model and its efficient implementation in the $\mathcal{TOY}$ system, although declarative forms of concurrency do exist for similar (but less expressive) approaches [3, 6, 8].

Despite those concurrent extensions, we are not aware of any implementation backed by theoretical results concerning the combination of sound and complete demand-driven narrowing with definitional trees and constraint solving to provide a more powerful declarative integration of concurrent programming techniques. The development of these practical techniques is essential for the implementation of concurrent multiparadigm logic programming systems, since they allow further optimizations related to the synchronization and communication of constraint solving mechanisms that can considerably reduce the search space generated by narrowing.

The aim of this paper is to provide a well-founded concurrent operational semantics that has the potential to be at the basis of more efficient implementations of constraint functional logic programming languages than the current ones [4]. The main contribution of this work is the hybrid operational combination between constraint solving and demand-driven narrowing guided by definitional trees, to show that this concurrent operational model allows constraint solving to efficiently reduce the search space generated by narrowing.

The rest of this paper is organized as follows. Section 2 introduces our approach by presenting an example of declarative concurrency in *CFLP*($\mathcal{FD}$), a concrete instance of constraint functional logic programming over the finite domain $\mathcal{FD}$ of integer numbers. In Sections 3 and 4 we introduce and enrich the presentation of the generic *CFLP*($\mathcal{D}$) scheme [9] underlying the implementation of the $\mathcal{TOY}$ system [11], now with concurrent features. Finally, Section 5 summarizes some conclusions and plans for future work.

2 An Example of Concurrent Execution in *CFLP*($\mathcal{FD}$)

For a first impression of our proposal of a concurrent operational model in constraint functional logic programming, we consider the following *conditional* ($\Leftarrow$) *rewriting rules* ($\rightarrow$) with constraints over the *constraint finite domain* $\mathcal{FD}$ of integers defining a simple *CFLP*($\mathcal{FD}$)-program to compute *Fibonacci numbers*:

$$\begin{array}{lll} fib\,(0) & \to & 1 \\ fib\,(1) & \to & 1 \\ fib\,(N) & \to & fib\,(N-1) + fib\,(N-2) \Leftarrow \quad N \geq 2 \end{array}$$

From this program, we want to compute all the values for the variable X from the *user-defined constraint* $fib\,(X) \leq 2$ (i.e., the values 0, 1 and 2 for X). We propose a concurrent operational semantics to improve the efficiency of the sequential $\mathcal{TOY}(\mathcal{FD})$ system [11] implementing $CFLP(\mathcal{FD})$. This enhanced operational semantics begins separating (we use the symbol '$\Box$' for this purpose) the initial goal $fib\,(X) \leq 2$ into the evaluation of a *function call* $fib\,(X) \to R$ and a solved *constraint store* with a *primitive constraint* $R \leq 2$, introducing a logical variable R for communication and synchronization between both parts:

$$fib\,(X) \to R \;\Box\; R \leq 2$$

The new system tries to concurrently evaluate both parts: the function call $fib\,(X) \to R$ by *demand-driven narrowing* [12] (i.e., a combination of *lazy rewriting* '$\to$' and *unification* '$\mapsto$' by substitutions) for each of the three (variable-renamed) program rules, and the primitive constraint $R \leq 2$ by the $\mathcal{FD}$*-constraint solver* of *SICStus Prolog* underlying $\mathcal{TOY}(\mathcal{FD})$ [11]:

1. **First program rule:** $fib\,(0) \to 1$

   In order to evaluate the function call $fib\,(X) \to R$, the first program rule can be applied to instantiate the argument X to 0 (indicated in the goal by the separation symbol '$\Box$' and the unification substitution $\{X \mapsto 0\}$) and to store the corresponding rewriting result 1 in the logical variable R:

   $$1 \to R \;\Box\; R \leq 2 \;\Box\; \{X \mapsto 0\}$$

   Now, we can reduce R to 1 and apply the accumulated substitution $\{R \mapsto 1, X \mapsto 0\}$ to instantiate the constraint $R \leq 2$. Then, the $\mathcal{FD}$-constraint solver checks the satisfiability of the instantiated store $1 \leq 2$. Thus, the first answer computed by constrained demand-driven narrowing is $\{X \mapsto 0\}$:

   $\Box\; 1 \leq 2 \;\Box\; \{R \mapsto 1, X \mapsto 0\} \Rightarrow \boxed{\sigma_1 = \{X \mapsto 0\}}$ (*First computed answer*)

2. **Second program rule:** $fib\,(1) \to 1$

   Concurrently, X can be also instantiated to 1 in our computational model, and then the second program rule can be applied to compute the second answer:

   $1 \to R \;\Box\; R \leq 2 \;\Box\; \{X \mapsto 1\}$
   $\Box\; 1 \leq 2 \;\Box\; \{R \mapsto 1, X \mapsto 1\} \Rightarrow \boxed{\sigma_2 = \{X \mapsto 1\}}$ (*Second computed answer*)

3. **Third program rule:** $fib\,(X) \to fib\,(X-1) + fib\,(X-2) \Leftarrow X \geq 2$

   Also concurrently, a variable-renamed variant of the third program rule can be applied to the goal, resulting in the evaluation of two new *fib* function calls:

   $fib\,(X-1) + fib\,(X-2) \to R \;\Box\; X \geq 2,\, R \leq 2$
   $fib\,(X-1) \to R_1,\, fib\,(X-2) \to R_2 \;\Box\; X \geq 2,\, R_1 + R_2 = R,\, R \leq 2$

   In this third case (3), our enhanced version of the $\mathcal{TOY}(\mathcal{FD})$ system explores concurrently two possible ways to efficiently compute more answers, according to the two possible flows of communication and synchronization (i.e., instantiation of the common logical variables X, R_1, R_2 and R) between the mechanisms of demand-driven narrowing and constraint solving, differing in their length (and therefore in efficiency) due to the different concurrent interleavings of both computational mechanisms.

3.1 From narrowing to constraint solving: In this first case, closer to the sequential execution of the $\mathcal{TOY}(\mathcal{FD})$ system [11], our operational model evaluates concurrently the function calls *fib* $(X - 1)$ and *fib* $(X - 2)$ (or equivalently, the flattened and standardized forms *fib* (N_1) and *fib* (N_2) with new constraints $N_1 = X - 1$ and $N_2 = X - 2$, respectively, in the common constraint store) by applying again a combination of demand-driven narrowing and constraint solving. For example, the system can compute the value $\{X \mapsto 2\}$ for X applying concurrently the second and the first program rules, respectively, to compute $\{N_1 \mapsto 1\}$ and $\{N_2 \mapsto 0\}$, and then applying the constraint solver to $1 = X - 1$ and $0 = X - 2$. Then, the corresponding result 1 will be stored in the logical variables R_1 and R_2:

$1 \rightarrow R_1, 1 \rightarrow R_2 \,\Box\, X \geq 2, R_1 + R_2 = R,\ R \leq 2, 1 = X-1, 0 = X-2 \,\Box\, \{N_1 \mapsto 1, N_2 \mapsto 0\}$
$1 \rightarrow R_1, 1 \rightarrow R_2 \,\Box\, R_1 + R_2 = R,\ R \leq 2 \,\Box\, \{N_1 \mapsto 1, N_2 \mapsto 0\ X \mapsto 2\}$

To ensure the consistency of this evaluation process by the demand-driven narrowing computation, our concurrent operational model has to protect (or *suspend*) the evaluation of variables R_1 and R_2 from the action of the constraint solver in favour of an evaluation only by narrowing to compute $\{R_1 \mapsto 1, R_2 \mapsto 1\}$ from $1 \rightarrow R_1$ and $1 \rightarrow R_2$. Analogously, since we want to compute values $\{N_1 \mapsto 1, N_2 \mapsto 0\}$ for the variables N_1 and N_2 by narrowing, we also need to protect both variables from the action of the constraint solver (this is the so-called *flex* narrowing option in this work). Finally, since both processes are synchronized by sharing the common constraint store that contains $R_1 + R_2 = R$, $R \leq 2$, and we have computed by narrowing the values $\{R_1 \mapsto 1, R_2 \mapsto 1\}$, the constraint solver can compute now the substitution $\{R \mapsto 2\}$ and offer to the user the third computed answer $\{X \mapsto 2\}$:

$\Box\, 1 + 1 = R,\ R \leq 2 \,\Box\, \{N_1 \mapsto 1, N_2 \mapsto 0, X \mapsto 2, R_1 \mapsto 1, R_2 \mapsto 1\}$
$\Box\, 2 \leq 2 \,\Box\, \{N_1 \mapsto 1, N_2 \mapsto 0, X \mapsto 2, R_1 \mapsto 1, R_2 \mapsto 1, R \mapsto 2\} \Rightarrow \boxed{\sigma_3 = \{X \mapsto 2\}}$

At this point, the narrowing computation in $\mathcal{TOY}(\mathcal{FD})$ performs an infinite and useless "trial and error" generation of other possible values for X to find new possible answers. For example, alternatively applying the first and second program rules it is possible to compute other values for N_1 and N_2 due to the concurrent evaluation of *fib* (N_1) and *fib* (N_2): $\{N_1 \mapsto 0, N_2 \mapsto 0\}$, $\{N_1 \mapsto 0, N_2 \mapsto 1\}$ or $\{N_1 \mapsto 1, N_2 \mapsto 1\}$. All of these concurrent processes only obtain inconsistent values for X from $N_1 = X-1$ and $N_2 = X - 2$ and must be discarded. Moreover, for each application of the third program rule, we have to evaluate two new function calls *fib* in order to infinitely compute concrete values for R_1 and R_2, and to check that each concrete instance of the constraint store $R_1 + R_2 = R$, $R \leq 2$ fails. How can our concurrent operational model efficiently help to prevent this infinite and useless search space generated by narrowing? This is the main idea of our paper:

3.2 From constraint solving to narrowing: In this case, our concurrent operational model needs to protect (or *suspend*) variables N_1 and N_2, now from the narrowing action (this is the so-called *rigid* narrowing or *residuation* option in this work). Then, as an important difference with respect to (3.1), the solver allows to solve the constraint $X \geq 2$ to generate and assign directly to the variables X, $N_1 = X - 1$ and $N_2 = X - 2$ only correct integer values: $\{X \mapsto 2, N_1 \mapsto 1, N_2 \mapsto 0\}$, $\{X \mapsto 3, N_1 \mapsto 2, N_2 \mapsto 1\}$, $\{X \mapsto 4, N_1 \mapsto 3, N_2 \mapsto 2\}$, etc. For each of these possible values, the system creates a process and awakes simple concurrent applications of *rewriting* (instead of expensive "trial and error" applications of narrowing as we have seen in (3.1)). For example, for the values $\{X \mapsto 2, N_1 \mapsto 1, N_2 \mapsto 0\}$ the concurrent system computes the same third

answer $\{X \mapsto 2\}$ in less time. For any other value $X \geq 3$, this process is free to use, concurrently, efficient $\mathcal{FD}$-constraint solving techniques [1, 4] to add directly to the constraint store $R_1 + R_2 > 2$. Then, the solver fails checking the extended common constraint store $R_1 + R_2 > 2$, $R_1 + R_2 = R$, $R \leq 2$ and stops the generation of more values for X. Moreover, since the goal solving processes share the same constraint store, the (3.2) option kills automatically all the remaining active processes in the (3.1) option, avoiding the generation of an infinite and useless narrowing computation. In conclusion, in this case constraint solving has helped to efficiently compute the last answer, and at the same time has reduced the search space generated by narrowing.

3 Concurrent Constraint Functional Logic Programming

In this section we give a revised summary of the generic *CFLP*($\mathcal{D}$) scheme [9] underlying our proposal of a concurrent system for multiparadigm logic programming.

3.1 Expressions, Patterns, and Constraints

A *signature* is a tuple $\Sigma = \langle DC, FS \rangle$ where $DC = \bigcup_{n \in \mathbb{N}} DC^n$ and $FS = \bigcup_{n \in \mathbb{N}} FS^n$ are families of countably infinite and mutually disjoint sets of *data constructors* and *evaluable function symbols*. Evaluable functions can be further classified into domain dependent *primitive functions* $PF^n \subseteq FS^n$ (e.g., $+, \leq \in PF^2$) and user *defined functions* $DF^n = FS^n \setminus PF^n$ for each arity $n \in \mathbb{N}$ (e.g., $fib \in DF^1$). We also assume a countably infinite set *Var* of *variables* $X, Y, \ldots$ and a set $\mathcal{U}$ of *primitive elements* $u, v, \ldots$ (as e.g., the set $\mathbb{Z}$ of integer numbers).

Expressions $e, e' \in Exp(\mathcal{U})$ have the syntax $e ::= \bot \mid u \mid X \mid h \mid (e\, e')$, where $\bot$ is a special symbol in DC^0 to denote an undefined data value, $u \in \mathcal{U}$, $X \in Var$, and $h \in DC \cup FS$. The following classification of expressions is useful: $X\, \overline{e}_m$ with $X \in Var$ and $m \geq 0$ is called a *flexible expression*, while $u \in \mathcal{U}$ and $h\, \overline{e}_m$ with $h \in DC \cup FS$ are called *rigid expressions*. Moreover, a rigid expression $h\, \overline{e}_m$ is called *active* if and only if $h \in FS^n$ and $m \geq n$, and *passive* otherwise. The occurrence of a symbol is *passive* if and only if is a primite element $u \in \mathcal{U}$ or is the root symbol h of a passive expression (a symbol used in this sense is called a *passive symbol*). Another class of expressions are *Patterns* $s, t \in Pat(\mathcal{U})$, built as $t ::= \bot \mid u \mid X \mid c\, \overline{t}_m \mid f\, \overline{t}_m$, where $c \in DC^n$ $(m \leq n)$ and $f \in FS^n$ $(m < n)$.

For every expression e, the set of *positions* in e is inductively defined as follows: the empty sequence identifies e itself, and for every expression of the form $h\overline{e}_m$, the sequence $i \cdot q$, where i is a positive integer not greater than m and q is a position, identifies the subexpression of e_i at q. The subexpression of e at p is denoted by $e|_p$ and the result of *replacing* $e|_p$ with e' in e is denoted by $e[e']_p$. If e is a *linear* expression (without repeated variable occurrences), $pos(X, e)$ will be used for the position of the variable X occurring in e. *Substitutions* $\sigma \in Sub(\mathcal{U})$ are mappings $\sigma : \mathcal{V} \rightarrow Pat(\mathcal{U})$ extended homomorphically to $\sigma : Exp(\mathcal{U}) \rightarrow Exp(\mathcal{U})$. We define the *domain* $Dom(\sigma)$ of a substitution σ as the collection of variables that are not mapped to themselves.

A *constraint domain* $\mathcal{D}$ provides a set of specific data elements $u \in \mathcal{U}$ along with certain primitive functions $p \in PF$ operating on them. For example, the *constraint finite domain* $\mathcal{FD}$ [4, 9] can be formalized as a structure with carrier set consisting of patterns built from the symbols in a signature Σ and the set of primitive elements $\mathbb{Z}$. Symbols in Σ are intended to represent data constructors (e.g., the list constructors), domain specific primitive functions (e.g., addition and multiplication over $\mathbb{Z}$), and user defined functions. *Constraints* have the syntactic form $p\, \overline{e}_n$, with $p \in PF^n$ a primitive relational symbol and $\overline{e}_n \in Exp(\mathcal{U})$ (e.g., $fib\,(X) \leq 2$, $X \geq 2$ or $R_1 + R_2 = R$ in infix notation).

3.2 Programs and Constrained Definitional Trees

In the sequel, we assume an arbitrarily fixed constraint domain $\mathcal{D}$ built over a set of primitive elements $\mathcal{U}$. In this setting, a *program* is a set of constrained rewrite rules that defines the behavior of possibly higher-order and/or non-deterministic lazy functions over $\mathcal{D}$, called *program rules*. More precisely, a program rule R for $f \in DF^n$ has the form $f\,\bar{t}_n \to r \Leftarrow P \,\square\, C$ (abbreviated as $f\,\bar{t}_n \to r$ if P and C are both empty; see Section 2) and is required to satisfy:

- The left-hand side $f\,\bar{t}_n$ is a linear expression with $\bar{t}_n \in Pat(\mathcal{U})$, and the right-hand side $r \in Exp(\mathcal{U})$.
- P is a finite sequence of so-called *productions* of the form $e_i \to R_i$ $(1 \le i \le k)$, intended to be interpreted as a conjunction of local definitions with no cycles [9]. For all $1 \le i \le k$, $e_i \in Exp(\mathcal{U})$, and $R_i \notin Var(f\,\bar{t}_n)$ are different variables.
- C is a finite set of constraints, also intended to be interpreted as a conjunction, and possibly including occurrences of user-defined function symbols.

$\mathcal{T}_\tau$ is a *constrained Definitional Tree* over $\mathcal{D}$ ($cDT(\mathcal{D})$ for short) with *call pattern* τ (a linear pattern of the form $f\bar{t}_n$, where $f \in DF^n$ and $\bar{t}_n \in Pat(\mathcal{U})$) if its depth is finite and one of the following cases holds for the rules of a program $\mathcal{P}$:

- $\mathcal{T}_\tau \equiv \underline{rule}(\tau \to r_1 \Leftarrow P_1 \,\square\, C_1 \,\|\, \ldots \,\|\, r_m \Leftarrow P_m \,\square\, C_m)$, where $\tau \to r_i \Leftarrow P_i \,\square\, C_i$ for all $1 \le i \le m$ are variants of overlapping program rules in $\mathcal{P}$.
- $\mathcal{T}_\tau \equiv \underline{case}(\tau, X, op, [\mathcal{T}_1, \ldots, \mathcal{T}_k])$, where X is a variable in τ, $op \in \{\mathit{flex}, \mathit{rigid}, \mathit{flex}/\mathit{rigid}\}$, $h_1, \ldots, h_k$ $(k > 0)$ are pairwise different passive symbols of $\mathcal{P}$, and for all $1 \le i \le k$, $\mathcal{T}_i$ is a $cDT(\mathcal{D})$ with call pattern $\tau\sigma_i$, where $\sigma_i = \{X \mapsto h_i\overline{Y}_{m_i}\}$ with $\overline{Y}_{m_i}$ new distinct variables such that $h_i\overline{Y}_{m_i} \in Pat(\mathcal{U})$.

A $\mathcal{T}_f$ *of a function symbol* $f \in DF^n$ defined by a program $\mathcal{P}$ is a $cDT(\mathcal{D})$ with call pattern $f\overline{X}_n$, where $\overline{X}_n$ are new variables, and the collection of all the program rules obtained from the different $\underline{rule}$ nodes equals, up to variants, the collection of all the program rules defining f in $\mathcal{P}$.

3.3 Goals and Answers

A *goal* G for a program has the general form $P \,\square\, C \,\square\, S \,\square\, \sigma$, where the separation symbol '$\square$' must be interpreted as a conjunction, and:

- $P \equiv e_1 \to R_1, \ldots, e_n \to R_n$ is a finite conjunction of so-called *productions*, where each R_i is a distinct variable and e_i is an expression (we call these productions *suspensions*), or a pair of the form $< \tau, \mathcal{T} >$ with τ an instance of the call pattern in the root of a $cDT(\mathcal{D})$ $\mathcal{T}$ (we call these productions *demanded productions*). The set of *produced variables* is $PVar(P) =_{def} \{R_1, \ldots, R_n\}$ (e.g., R, R_1 and R_2 in Section 2).
- $C \equiv \delta_1, \ldots, \delta_k$ is a finite conjunction of constraints (possibly including user-defined function symbols; e.g., $\mathit{fib}\,(X) \le 2$ in the initial goal of Section 2).
- $S \equiv \pi_1, \ldots, \pi_l$ is a finite conjunction of *primitive* constraints (i.e., constraints with only pattern arguments; e.g., $R \le 2$), called *constraint store*.
- $\sigma \in Sub(\mathcal{U})$ is an idempotent substitution called *answer substitution* such that $Dom(\sigma) \cap Var(P \,\square\, C \,\square\, S) = \emptyset$.

A *solved goal* is a goal $\square \,\square\, S \,\square\, \sigma$ in which P and C are empty, and identifies an *answer* $S \,\square\, \sigma$ (or simply σ, as we have seen in Section 2). We say that $X \in Var(G)$ is a *demanded variable* in G if and only if one of the following cases holds:

1. Any substitution that is a solution of S cannot bind X to the undefined value $\perp$ (shortly, $X \in DVar_{\mathcal{D}}(S)$). For example, $R \in DVar_{\mathcal{FD}}(R \leq 3)$.
2. There exists a suspension $(X\overline{a}_k \to R) \in P$ such that $k > 0$ and R is a demanded variable in G (this case is only necessary to deal with higher-order [9]).
3. There exists a demanded production $(< e, \underline{case}\,(\tau, Y,\ op,\ [\mathcal{T}_1, \ldots, \mathcal{T}_k]) > \to R) \in P$ such that $X = e|_{pos(Y,\tau)}$ and R is a demanded variable in G (see e.g., N_1 and N_2 in (3.1) and (3.2)). If op in the branch node is of type *flex*, the variable X is called a *flex variable* (e.g., N_1 and N_2 in (3.1)). Otherwise, the variable X is called a *rigid variable* (e.g., N_1 and N_2 in (3.2)).

4 A Concurrent Operational Semantics for $CFLP(\mathcal{D})$

In this section we present a set of *concurrent goal transformation rules* of the form $G \Vdash_{\mathbf{R}} \|_{i=1}^{k} G_i$, specifying all the possible *concurrent evaluations* ($\|_{i=1}^{k}$) of subgoals G_i obtained by applying a rule $\mathbf{R}$ of goal solving ($\Vdash_{\mathbf{R}}$) to a goal G in our concurrent operational semantics for the $CFLP(\mathcal{D})$ scheme. All these rules (formally presented in Figures 1 and 2) have been implemented in the $\mathcal{TOY}$ system [11] and are implicitly applied in our running example of Section 2. We refer the reader to that section for detailed examples illustrating the application of all these rules. We write $G \Vdash^* \|_{i=1}^{k} G_i$ to represent *concurrent derivations*, given by the successive application ($\Vdash^*$) of concurrent goal transformation rules from G. For example, the concurrent derivation $G \Vdash^* G'_1 \,\|\, G_{21} \,\|\, G_{22}$ represents the concurrent goal transformation steps $G \Vdash G_1 \,\|\, G_2$ with $G_1 \Vdash G'_1$ and $G_2 \Vdash G_{21} \,\|\, G_{22}$.

Each time a goal G contains the conjunction of two or more atomic statements that could be concurrently evaluated (e.g., two or more productions), our operational model creates concurrent goal solving processes, each of one consisting of an atomic statement from G, together with the necessary information for an adequate and consistent demand-driven evaluation applying a concurrent goal transformation rule (i.e., the sets of produced, demanded, rigid and flex variables). Moreover, for synchronization and in order to properly combine the possible computed answers from subgoal processes, as well as the cases in which processes remain *suspended* (indicated by the symbol $\circlearrowleft$) or *fail* (indicated by the symbol $\blacksquare$), new subgoals must share the constraint store of the main goal G.

4.1 Concurrent Demand-Driven Narrowing and Residuation

We start with a suspension $e \to R$ representing the computation of a function call, for example $fib\,(X) \to R$, where e has a user-defined function symbol f in the root (e.g., fib) and R is a demanded variable (e.g., by the constraint store $R \leq 2$). Then, the rule **DT** (see Figure 1) is applicable, awakening the suspension $e \to R$, decorating e with an appropriate $cDT(\mathcal{D})$ $\mathcal{T}_f$ (e.g., $\mathcal{T}_{fib}$ given in Section 3), and introducing a new demanded production $< e, \mathcal{T}_f > \to R$ into the goal. If the function call is not demanded (i.e., R is not a demanded variable), this computation remains suspended ($\circlearrowleft$) until the variable R disappears from the goal (and then the suspension can be eliminated) or R becomes demanded. The goal transformation rules for demanded productions $< e, \mathcal{T}_f > \to R$ encode the *demand-driven narrowing strategy* [12] guided by the constrained definitional tree $\mathcal{T}_f$, now in a concurrent setting:

- If $\mathcal{T}_f$ is a $\underline{rule}$ tree, then the transformation **RRA** can be concurrently applied ($\|_{i=1}^{k}$) for each of the k available overlapping program rules for rewriting e, introducing appropriate suspensions and constraints into the new subgoals so that a demand-driven evaluation can be ensured.

DT Definitional Tree

$$f\overline{e}_n \to R, P \,\square\, C \,\square\, S \,\square\, \sigma \Vdash_{\mathbf{DT}} \left\{ \begin{array}{ll} < f\overline{e}_n, \mathcal{T}_{f\overline{X}_n} > \to R, P \,\square\, C \,\square\, S \,\square\, \sigma & \text{if } R \in DVar_{\mathcal{D}}(P \,\square\, S) \\ \circlearrowleft & \text{if } R \notin DVar_{\mathcal{D}}(P \,\square\, S) \end{array} \right\}$$

if $f \in DF^n$, and all variables in $\mathcal{T}_{f\overline{X}_n}$ are new variables.

RRA Rewrite Rule Application

$$< f\overline{e}_n, \underline{rule}\,(f\overline{t}_n \to r_1 \Leftarrow P_1 \,\square\, C_1 \,\|\, \ldots \,\|\, r_k \Leftarrow P_k \,\square\, C_k) > \to R, P \,\square\, C \,\square\, S \,\square\, \sigma \Vdash_{\mathbf{RRA}}$$
$$\|_{i=1}^{k} \; \overline{e_n \to t_n}, r_i \to R, P_i, P \,\square\, C_i, C \,\square\, S \,\square\, \sigma$$

CSS Case Selection

$$< e, \underline{case}\,(\tau, X, op, [\mathcal{T}_1, \ldots, \mathcal{T}_k]) > \to R, P \,\square\, C \,\square\, S \,\square\, \sigma \Vdash_{\mathbf{CSS}} < e, \mathcal{T}_i > \to R, P \,\square\, C \,\square\, S \,\square\, \sigma$$

if $e|_{pos(X,\tau)} = h_i \ldots$ with $1 \le i \le k$ given by e, and h_i is the passive symbol associated to $\mathcal{T}_i$.

CC Case non-Cover

$$< e, \underline{case}\,(\tau, X, op, [\mathcal{T}_1, \ldots, \mathcal{T}_k]) > \to R, P \,\square\, C \,\square\, S \,\square\, \sigma \Vdash_{\mathbf{CC}} \blacksquare$$

if $e|_{pos(X,\tau)} = h \ldots$ is a passive symbol $h \notin \{h_1, \ldots, h_k\}$, being h_i the passive symbol associated to $\mathcal{T}_i$.

DN Demand Narrowing

$$< e, \underline{case}\,(\tau, X, op, [\mathcal{T}_1, \ldots, \mathcal{T}_k]) > \to R, P \,\square\, C \,\square\, S \,\square\, \sigma \Vdash_{\mathbf{DN}}$$
$$e|_{pos(X,\tau)} \to R', < e[R']_{pos(X,\tau)}, \underline{case}\,(\tau, X, op, [\mathcal{T}_1, \ldots, \mathcal{T}_k]) > \to R, P \,\square\, C \,\square\, S \,\square\, \sigma$$

if $e|_{pos(X,\tau)} = g \ldots$ with $g \in FS$ active (primitive or defined function), and R' a new variable.

DP Demand Produced Variable

$$< e, \underline{case}\,(\tau, X, op, [\mathcal{T}_1, \ldots, \mathcal{T}_k]) > \to R, P \,\square\, C \,\square\, S \,\square\, \sigma \Vdash_{\mathbf{DP}} \circlearrowleft$$

if $e|_{pos(X,\tau)} = Y$ with $Y \in PVar(P)$.

DR Demand Residuation

$$< e, \underline{case}\,(\tau, X, rigid, [\mathcal{T}_1, \ldots, \mathcal{T}_k]) > \to R, P \,\square\, C \,\square\, S \,\square\, \sigma \Vdash_{\mathbf{DR}} \circlearrowleft$$

if $e|_{pos(X,\tau)} = Y$ with $Y \notin PVar(P)$.

DI Demand Instantiation

$$< e, \underline{case}\,(\tau, X, flex, [\mathcal{T}_1, \ldots, \mathcal{T}_k]) > \to R, P \,\square\, C \,\square\, S \,\square\, \sigma \Vdash_{\mathbf{DI}}$$
$$\|_{i=1}^{k} \; (< e, \mathcal{T}_i > \to R, P \,\square\, C \,\square\, S)\sigma_i \,\square\, \sigma\sigma_i$$

if $e|_{pos(X,\tau)} = Y$ with $Y \notin PVar(P)$, and $\sigma_i = \{Y \mapsto h_i\overline{Y}_{m_i}\}$ with h_i $(1 \le i \le k)$ the passive symbol associated to $\mathcal{T}_i$, and $\overline{Y}_{m_i}$ are new variables.

Figure 1 Rules for concurrency in constrained demand-driven narrowing and residuation.

- If $\mathcal{T}_f$ is a *case* tree, one of the transformations **CSS**, **CC**, **DN**, **DP**, **DR** or **DI** must be applied, according to the kind of symbol h occurring in e at the case-distinction position $pos(X, \tau)$:
  - If h is a passive symbol h_i, then **CSS** selects the appropriate subtree $\mathcal{T}_i$ (otherwise **CC** fails $\blacksquare$).
  - If h is an active primitive or defined function symbol g, then **DN** introduces a new demanded suspension in the goal to evaluate $e|_{pos(X,\tau)}$.
  - If h is a produced variable Y, the goal must remain suspended ($\circlearrowleft$) using **DP** until a concurrent process of the computation evaluates Y.
  - If Y is a non-produced variable, there are two possibilities:
    * If the branch node has the option *rigid* (or *flex/rigid*), we must suspend the evaluation ($\circlearrowleft$) using **DR** until the variable has been bound, for example, by the action of the constraint solver (as we have seen in (3.2) for N_1 and N_2, and we

AC Atomic Constraint

$$P \,\square\, p\overline{e}_n, C \,\square\, S \,\square\, \sigma \Vdash_{\mathbf{AC}} \|_{i=1}^{n}\; e_i \to X_i, P \,\square\, C \,\square\, p\overline{X}_n, S \,\square\, \sigma$$

if $p \in PF^n$, $p\overline{e}_n$ is a constraint, and $\overline{X}_n$ are new variables.

CS Constraint Solving

$$P \,\square\, C \,\square\, S \,\square\, \sigma \Vdash_{\mathbf{CS}\{\chi\}} \|_{i=1}^{k} \left\{ \begin{array}{ll} (P \,\square\, C)\sigma_i \,\square\, S_i \,\square\, \sigma\sigma_i & \text{if (i) or (ii) or (iii) in Section 4.2} \\ \circlearrowleft & \text{otherwise} \end{array} \right\}$$

if $\mathit{Solver}^{\mathcal{D}}(S, \chi) = \bigvee_{i=1}^{k}(S_i \,\square\, \sigma_i)$ with $\chi =_{def} PVar(P) \cup FVar(P)$.

SF Solving Failure

$$P \,\square\, C \,\square\, S \,\square\, \sigma \Vdash_{\mathbf{SF}\{\chi\}} \blacksquare \qquad \text{if } \mathit{Solver}^{\mathcal{D}}(S, \chi) = \mathit{fail}.$$

Figure 2 Rules for concurrent constraint solving.

will formalize in the next subsection). This case corresponds to the computational principle of declarative *residuation* [6].

* If the branch node has the option *flex* (or *flex/rigid*), then **DI** selects concurrently ($\|_{i=1}^{k}$) each subtree $\mathcal{T}_i$, generating an appropriate binding σ_i for Y (as e.g., for N_1 and N_2 in (3.1)).

4.2 Concurrent Constraint Solving

The goal transformation rules concerning *concurrent constraint solving* (see Figure 2) are designed to concurrently combine the evaluation of (primitive or user-defined) constraints with the action of a constraint solver over the given domain. The first rule **AC** evaluates non-primitive constraints $p\overline{e}_n$ (e.g., $\mathit{fib}\,(X) \leq 2$) by performing a concurrent evaluation ($\|_{i=1}^{n}$) of their arguments e_i in suspensions $e_i \to X_i$, and introducing a flattened primitive constraint $p\overline{X}_n$ into the common constraint store, with new logical variables $\overline{X}_n$ for the communication and synchronization among all these concurrent goal solving processes.

For the evaluation of primitive constraints in a constraint domain $\mathcal{D}$ we postulate a *constraint solver* of the form $\mathit{Solver}^{\mathcal{D}}(S, \chi)$, which can reduce any given finite conjunction of primitive constraints S representing the constraint store of the goal into an equivalent simpler solved form. The constraint solver needs to take proper care of a selected set of so-called *critical* (or *protected*) *variables* $\chi =_{def} PVar(P) \cup FVar(P)$ occurring in S to ensure a correct demand-driven evaluation (see variables R_1, R_2 and N_1, N_2 in (3.1) and (3.2) of Section 2). We require that any solver invocation returns a finite disjunction of k simpler solved form alternatives $S_i \,\square\, \sigma_i$. Then, the rule **CS** describes the possible concurrent ($\|_{i=1}^{k}$) evaluations of a single goal by a solver's invocation for each possible alternative solved form computed by the constraint solver. To avoid deadlock situations, we require solvers to have the ability to compute and discriminate a distinction of the following cases and situations for each concurrent solved form alternative (illustrated by (3.1) and (3.2) in Section 2):

(i) A suspended production ($\circlearrowleft$) (e.g., suspended by the **DT** rule) with a non-demanded critical variable at the right-hand side may be now demanded (and then activated) by the new constraint store S_i of some alternative $S_i \,\square\, \sigma_i$ (formally, $DVar_{\mathcal{D}}(S_i) \cap \chi \neq \emptyset$), or

(ii) A suspended demanded production ($\circlearrowleft$) (for example, suspended by the **DR** rule) could be activated by applying σ_i to instantiate a *rigid* and not produced variable in this production (i.e., $Dom(\sigma_i) \cap RVar(P) \neq \emptyset$), or

(iii) A suspended production ($\circlearrowleft$) could be irrelevant for the new constraint store S_i (i.e., $Var(S_i) \cap \chi = \emptyset$) and then has to be eliminated.

For any other situation, the corresponding goal solving process must be suspended (↻) by the action of the constraint solver. Additionally, the failure rule **SF** is used for failure detection (■) in the constraint solving process.

4.3 Soundness and Completeness

We conclude this section with the main theoretical result of the paper ensuring *soundness* and *completeness* for concurrent *CFLP*($\mathcal{D}$)-derivations w.r.t. the declarative semantics of the *CFLP*($\mathcal{D}$) scheme formalized in [5, 9] by means of a *Constraint Rewriting Logic CRWL*($\mathcal{D}$).

▶ **Theorem 1** (Soundness and Completeness). *Let $S \square \sigma$ be an answer of G.*

(a) **Soundness:** *If $G \Vdash^* \|_{i=1}^{k} G_i$ is a concurrent derivation from G of a finite number k of goals G_i, for each $G_i \equiv \square \square S_i \square \sigma_i$ a solved goal, $S_i \square \sigma_i$ is an answer of the initial goal G. Formally, $Sol_{\mathcal{D}}(G_i) \subseteq Sol_{\mathcal{P}}(G)$.*

(b) **Completeness:** *There exists a concurrent derivation $G \Vdash^* \|_{i=1}^{k} G_i$, ending with a finite number k of solved goals $G_i \equiv \square \square S_i \square \sigma_i$, that covers all the solutions of the initial answer $S \square \sigma$. Formally, $Sol_{\mathcal{P}}(G) \subseteq \bigcup_{i=1}^{k} Sol_{\mathcal{D}}(G_i)$.*

5 Conclusions and Future Work

The set of transformation rules presented in Section 4 provides a *sound* and *complete* operational model to describe a concurrent *CFLP*($\mathcal{D}$) scheme as a novel generalization of the classical *CLP*($\mathcal{D}$) scheme useful for concurrent functional and constraint logic programming.

We are currently investigating other practical instances of constraint domains (e.g., linear and non-linear arithmetic constraints over real numbers) and the cooperative integration of more efficient constraint solving methods into our concurrent system (e.g., based on the *ILOG CP* technology [1] or using declarative modeling languages such as *OPL*).

References

1. I. Castiñeiras and F. Sáenz Pérez. *Integrating ILOG CP Technology into $\mathcal{TOY}$*. In Proc. WFLP'09, pages 27-43, 2009.
2. *Curry*. Available at `http://www-ps.informatik.uni-kiel.de/currywiki/`.
3. R. Echahed and W. Serwe. *Defining Actions in Concurrent Declarative Programming*. In Electr. Notes Theor. Comput. Sci. 64, pages 176-194, 2002.
4. A. J. Fernández et. al. *Constraint functional logic programming over finite domains*. Journal of TPLP 7(5), pages 537–582, 2007.
5. F.J. López Fraguas, M. Rodríguez Artalejo, and R. del Vado. *A Lazy Narrowing Calculus for Declarative Constraint Programming*. In PPDP'04, pages. 43–54, 2004.
6. M. Hanus. *Multiparadigm Declarative Languages*. ICLP'07, pages 45-75, 2007.
7. J. Jaffar and J.L. Lassez. *Constraint logic programming*. POPL'87, pages 111-119, 1987.
8. M. Marin, T. Ida, and W. Schreiner. *CFLP: A Mathematica Implementation of a Distributed Constraint Solving System*. The Math. Journal 8(2), pages 287-300, 2001.
9. F.J. López, M. Rodríguez, and R. del Vado. *A new generic scheme for functional logic programming with constraints*. Journal of HOSC 20 (1-2), pages 73-122, 2007.
10. V. Saraswat and M. Rinard. *Concurrent constraint programming*. POPL'90, pages 232-245.
11. *$\mathcal{TOY}$: A Constraint Functional Logic System*. Available at `toy.sourceforge.net`.
12. R. del Vado. *A demand-driven narrowing calculus with overlapping definitional Trees*. In PPDP 2003, ACM, pages 253-263, 2003.

Surviving Solver Sensitivity: An ASP Practitioner's Guide

Bryan Silverthorn[1], Yuliya Lierler[2], and Marius Schneider[3]

1 **Department of Computer Science**
The University of Texas at Austin, Austin, TX, USA
`bsilvert@cs.utexas.edu`

2 **Department of Computer Science**
University of Kentucky, Lexington, KY, USA
`yuliya@cs.uky.edu`

3 **Institute of Computer Science**
University of Potsdam, Potsdam, Germany
`manju@cs.uni-potsdam.de`

Abstract

Answer set programming (ASP) is a declarative programming formalism that allows a practitioner to specify a problem without describing an algorithm for solving it. In ASP, the tools for processing problem specifications are called answer set solvers. Because specified problems are often NP complete, these systems often require significant computational effort to succeed. Furthermore, they offer different heuristics, expose numerous parameters, and their running time is sensitive to the configuration used. Portfolio solvers and automatic algorithm configuration systems are recent attempts to automate the problem of manual parameter tuning, and to mitigate the burden of identifying the right solver configuration. The approaches taken in portfolio solvers and automatic algorithm configuration systems are orthogonal. This paper evaluates these approaches, separately and jointly, in the context of real-world ASP application development. It outlines strategies for their use in such settings, identifies their respective strengths and weaknesses, and advocates for a methodology that would make them an integral part of developing ASP applications.

1998 ACM Subject Classification I.2.2 Automatic analysis of algorithms

Keywords and phrases algorithm configuration, algorithm selection, portfolio solving, answer set programming, algorithm portfolios

Digital Object Identifier 10.4230/LIPIcs.ICLP.2012.164

1 Introduction

Answer set programming (ASP) [19, 20] is a declarative programming formalism based on the answer set semantics of logic programs [9]. Its origins go back to the observation that the language of logic programs can be used to model difficult combinatorial search problems so that *answer sets* correspond to the solutions of a problem. In the declarative programming paradigm, a software engineer expresses the logic of a computation without describing its control flow or algorithm. Thus a declarative program is a description of what should be accomplished, rather than a description of how to go about accomplishing it. As a result, declarative programming requires tools that process given problem specifications and find their solutions. In ASP such systems are called answer set *solvers*. They implement a difficult computational task, since the problem of deciding whether a logic program has an answer set is NP-complete. Despite the complexity of the task, ASP and its tools have proved to

© Bryan Silverthorn, Yuliya Lierler, and Marius Schneider;
licensed under Creative Commons License ND
Technical Communications of the 28th International Conference on Logic Programming (ICLP'12).
Editors: A. Dovier and V. Santos Costa; pp. 164–175

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

be useful; ASP has been used to model and solve problems in many real-world domains, including computational biology and linguistics.

The computational methods of top-performing answer set solvers such as CMODELS [10] and CLASP [8, 5] are strongly related to those of satisfiability (SAT) solvers—software systems designed to find satisfying assignments for propositional formulae [11]. It is well known that modern SAT solvers are sensitive to parameter configuration. The same holds for answer set solvers. These systems typically implement numerous heuristics and expose a variety of parameters to specify the chosen configuration. For example, the command line

```
clasp --number=1 --trans-ext=no --eq=5 --sat-prepro=0 --lookahead=no
      --heuristic=Berkmin --rand-freq=0.0 --rand-prob=no
      --rand-watches=true --restarts=100,1.5 --shuffle=0,0
      --deletion=3.0,1.1,3.0 --strengthen=yes
      --loops=common --contraction=250 --verbose=1
```

represents the default configuration of answer set solver CLASP (version 2.0.2). On one hand, a rich set of heuristics implemented in CLASP makes this solver successfully applicable to a variety of problem domains. On the other hand, given an application at hand, it is unclear how to go about picking the *best* configuration of the system. Gebser et al. [7] write:

> In fact, we are unaware of any true application on which CLASP is run in its default settings. Rather, in applications, "black magic" is used to find suitable search parameters.

We believe that *black magic* refers to manual tuning that relies on "rules of thumb", solver familiarity, and the user's domain expertise. Unfortunately, it is unreasonable to expect that a regular ASP application developer possesses enough knowledge about the internals of answer set solvers and their heuristics to understand the full implications of picking a particular configuration. Furthermore, requiring such extraordinary expertise would diminish the idea of declarative programming itself.

In this paper we evaluate some of the tools available for addressing the black magic problem—including portfolio solvers such as CLASPFOLIO [7] and BORG [23], and automatic algorithm configuration systems such as PARAMILS [14]—in the context of specific problem domains in ASP, highlighting real-world applications. Our goal is not to develop a novel system, but to aid ASP application developers, especially those hoping to leverage existing tools for portfolio solving and configuration tuning. We provide a case study that illustrates and evaluates alternative methodologies in three practical domains: weighted sequence [18], natural language parsing [17], and Riposte [2]. The struggle to deal with solver sensitivity in the former two domains, in fact, triggered the research described in this paper. Our hope is to identify a systematic way to remove haphazard manual tuning and performance evaluation from the process of applying ASP tools to a new domain. We focus our attention on tuning the ASP solver CLASP (version 2.0.2). The configuration space of CLASP consists of 8 binary, 7 categorical, and 25 continuous parameters, which makes us believe that this system alone is a good choice for evaluation. Nevertheless, all of the methods investigated here may accommodate any other solver of interest, as well as multiple solvers.

Six candidate strategies are studied:

1. selecting the best single configuration from among the 25 representative CLASP configurations used by CLASPFOLIO (a CLASP-based portfolio solver) for each domain;
2. constructing a "solver execution schedule" over those 25 configurations;
3. applying CLASPFOLIO, trained on its large set of ASP instances, to each domain without further modification;

4. training a portfolio specifically on each application domain using the 25 CLASP configurations of CLASPFOLIO;
5. tuning a single CLASP configuration specifically for each domain using PARAMILS; and
6. training a portfolio specifically on each application domain using multiple configurations produced by tuning CLASP, using PARAMILS, for individual instances of the domain.

We believe that these options are representative of modern approaches for dealing with solvers' configuration sensitivity, able to illustrate the strengths and weaknesses of each approach. Also, to the best of our knowledge, the evaluation of strategies 4 and 6 on individual problem domains is unique to this paper.

We start by describing the domains used in the proposed case study. Section 3 gives an overview of portfolio methods in general together with the details of the strategies 1, 2, 3, and 4. Section 4 outlines the general principles behind the algorithm configuration system PARAMILS and describes the details of strategy 5. Strategy 6 is specified in Section 5. We conclude with a thorough analysis of the methods' performance in practice.

2 Review of Application Domains

In this work we compare and contrast several methodologies on three domains that stem from different subareas of computer science. This section provides a brief overview of these applications. We believe that these domains represent a broad spectrum of ASP applications, and are thus well-suited for the case study proposed. The number of instances available for each application ranged from several hundred to several thousand. The complexity of the instances also varied. The diversity of the instances and their structure played an important role in our choice of domains.

The **weighted-sequence** (WSEQ) domain is a handcrafted benchmark problem that was used in the Third Answer Set Programming Competition[1] (ASPCOMP) [3]. Its key features are inspired by the important industrial problem of finding an optimal join order by cost-based query optimizers in database systems. In our analysis we used 480 instances of the problem, which were generated according to the metrics described by Lierler et al. [18].

The **natural language parsing** (NLP) domain formulates the task of parsing natural language, i.e., recovering the internal structure of sentences, as a planning problem in ASP. In particular, it considers the combinatory categorical grammar formalism for realizing parsing. Lierler and Schüller [17] describe the procedure of acquiring instances of the problem using CCGbank[2], a corpus of parsed sentences from real world sources. In this work we study 1,861 instances produced from the CCGbank data.

Riposte (RIP)[3] is a project in computer aided verification, where ASP is used to generate counterexamples for the FDL intermediate language of the SPARK program verification system. These counterexamples point at the problematic areas of the analyzed code. We evaluate instances created from the application of Riposte to a SPARK implementation of the Skein hash function; these 3,133 instances were shared with us by Martin Brain in February 2012.

---

[1] `https://www.mat.unical.it/aspcomp2011/OfficialProblemSuite`. WSEQ was referred to as a benchmark number 28, *Weight-Assignment Tree.*

[2] `http://groups.inf.ed.ac.uk/ccg/ccgbank.html`.

[3] `https://forge.open-do.org/projects/riposte`

3 Algorithm Portfolio Methods

In *portfolio solving*, an "algorithm portfolio method" or "portfolio solver" automatically divides computation time among a suite of solvers. SAT competitions[4] have provided a rich source of diverse solvers and benchmark instances, and have spurred the development of portfolio solving. Several different types of portfolio solvers exist. These range from simple methods that divide computational resources equally among a hand-selected suite of solvers, to more complex systems that make informed decisions by analyzing the appearance of instances. The development of the ASP portfolio solver CLASPFOLIO [7] was largely inspired by the advances of this approach in SAT, and especially by the ideas championed by the portfolio SAT solver SATZILLA [26].

Gebser et al. [6] suggest that portfolio solving in general and CLASPFOLIO in particular is a step toward overcoming the sensitivity of modern answer set solvers to parameter settings. Nevertheless, the extent to which existing portfolio solvers achieve this goal on individual application domains is an open issue. In this paper, we shed light on two questions related to it. First, how well does a general-purpose portfolio, trained on many different instance types, perform when compared against the default CLASP configuration on a particular domain? Second, what benefits are gained from moving from general-purpose to application-driven portfolios, by training a portfolio solver specifically for the application in question?

Two different portfolio systems are employed in considering these questions: the CLASPFOLIO algorithm-selection system is used as a general-purpose portfolio, and the BORG algorithm-portfolio toolkit is used to construct domain-specific portfolios based on the MAPP architecture [23]. Both of these approaches are described below.

3.1 Algorithm Selection and CLASPFOLIO

Systems for automatic algorithm selection, such as SATZILLA for SAT and CLASPFOLIO for ASP, leverage the appearance of an instance to make decisions about which solver or configuration to apply. An algorithm selection system typically involves two components:

- a suite or *portfolio* of different solvers or solver configurations, and
- a solver (or configuration) *selector*.

The selector is responsible for picking the best-performing solver for a particular instance. The definition of "best-performing" is arbitrary, but expected run time is often used. The efficiently computable properties of an instance on which these methods base their decisions are called numerical *features* of that instance.

Techniques from supervised machine learning are used to build the selector component. Thousands of runs are observed during a *training* phase, and each run is labeled with its performance score and the features of its associated instance. These examples are then used to learn a function that maps an instance, using its features, directly to a solver selection decision. Using this architecture, algorithm-selection portfolios have been top performers at the SAT and ASP competitions. For example, the portfolio answer set solver CLASPFOLIO was the winner of the NP category in the system track of ASPCOMP.

The CLASPFOLIO (version 1.0.1) solver employs 25 representative configurations of CLASP (version 2.0.2), and a feature set that includes properties of an ASP instance ranging from the number of constraints to the length of clauses learned from short initial runs. The choice of configurations of CLASP that were used in building CLASPFOLIO relied on the expertise

[4] http://www.satcompetition.org/.

of Benjamin Kaufmann, the main designer of CLASP, and on black magic. CLASPFOLIO[5] was trained on 1,901 instances from 60 different domains. It will be used to evaluate the performance of general-purpose portfolio, one designed to operate on a wide variety of instance types. A different system, but one that exhibits comparable performance, is used to evaluate the performance of domain-specific portfolio solvers. It is described next.

3.2 Solver Scheduling, MAPP, and BORG

We utilize the BORG toolkit[6] as our experimental infrastructure. Like tools such as RUNSOLVER [21], BORG executes solvers while measuring and limiting their run time. It is also designed to collect and analyze solver performance data over large collections of instances, to compute instance feature information, and to construct different portfolio solvers.

To build domain-specific portfolios, BORG instantiates the "modular architecture for probabilistic portfolios" (MAPP) [23]. Unlike an algorithm selection method, MAPP computes the complete *solver execution schedule* that approximately maximizes the probability of solving the instance within the specified run time constraint. A solver execution schedule consists of one or more sequential calls to possibly different solvers, where the last call is allocated all remaining runtime.

Unlike an algorithm selection portfolio, then, MAPP may run more than one solver on an instance. This strategy has proved to be effective. Earlier versions of MAPP [24], built for a portfolio of pseudo-Boolean (PB) solvers, took first place in the main category of the 2010 and 2011 PB competitions.

Two different types of MAPP portfolios are evaluated:

- MAPP^-, which does not use instance features, and thus consistently executes a single solver execution schedule computed over the run times of all training instances, and
- MAPP^+, which uses instance features to tailor each execution schedule to a given instance.

We used the BORG framework to create MAPP portfolios using the same 25 configurations of CLASP employed by CLASPFOLIO. This will allow us to more fairly compare the effectiveness of the application-driven portfolio-solving approach studied here to that of the general-purpose CLASPFOLIO system. Furthermore, we use CLASPFOLIO itself to compute instance-specific features for MAPP^+. As a result, MAPP^+ tailors a solver execution schedule to each instance using the same features available to CLASPFOLIO.

We also use BORG to select the "best single" (BESTSINGLE) configuration of CLASP, from among those 25, that maximizes the probability of successfully solving an instance of the training set.

Whether they employ pure algorithm selection or solver execution scheduling, portfolio methods have repeatedly proved successful on collections of competition instances. Such collections include instances of many different problems. Section 6 evaluates their behavior instead on collections of instances drawn entirely from each of our representative domains.

The next section discusses an orthogonal methodology for handling solver sensitivity. Instead of marshalling multiple fixed configurations, it follows a local search strategy through the configuration space of a solver, attempting to identify the best-performing single configuration on a domain.

---

[5] `http://potassco.sourceforge.net/#claspfolio`
[6] `http://nn.cs.utexas.edu/pages/research/borg/`.

4 Automatic Algorithm Configuration

The success of portfolio solving in competition demonstrates that selecting a solver's configuration is important. This success, however, leads to an obvious question: instead of focusing on the selection of an existing configuration, can we obtain a new configuration that performs better (or best) on a particular domain? This paper examines this possibility by applying a tool for *automatic algorithm configuration* to CLASP on our three application domains.

We take PARAMILS[7] [14] (version 2.3.5) as a representative of automatic algorithm configuration tools. Other systems of this kind include SMAC [12, 13] and GGA [1]. PARAMILS is based on iterative local search in the configuration space, and evaluates the investigated configurations on a given training set of instances. Its *focusedILS* approach allows it to focus the evaluation on a subset of the given instances, and thus to assess the quality of a configuration more quickly. This subset is adaptively extended after each update of the current suboptimal solution. The idea behind this approach is that a configuration that performs well on a small subset is also a good choice for the entire instance set. We designed our algorithm configuration experiments based on this observation.

In the experiments we tuned CLASP, with the help of PARAMILS, on a randomly sampled subset of 50 instances for each of the domains. The maximal cutoff time of each CLASP call was 1,200 seconds, the tuning time was 120,000 seconds, and the minimization of the average runtime was the optimization objective. Since PARAMILS uses a local search approach, it (i) is non-deterministic and (ii) can become trapped in a local optimum. Therefore, we ran the PARAMILS experiment ten times, independently, and afterwards chose the configuration with the best performance.

For all experiments, we used a discretized configuration space of CLASP selected by Benjamin Kaufmann. It is similar to the parameter file used in the experiments of Gebser et al. [7], and is available online at `http://www.cs.uni-potsdam.de/wv/claspfolio/`.

5 Domain-Specific Portfolio Synthesis

An automatic algorithm configuration system such as PARAMILS generates a *single* configuration tuned on a set of many instances. On the other hand, the assumption made by portfolio methods is that *multiple* configurations exhibit complementary strengths on a distribution of instances. If this assumption does not hold on some domain for a standard suite of solvers, is it possible to use automatic algorithm configuration to generate a new suite of complementary solvers? Systems such as Hydra have explored this possibility in SAT [25]. Here, we evaluate a simple strategy for doing so in ASP, leveraging PARAMILS. In this protocol, we

1. randomly sample a set of N instances from the domain,
2. use PARAMILS to tune a configuration of CLASP specifically for each instance,
3. collect training data for each configuration across the entire domain, and
4. construct a portfolio using those training data.

This methodology follows from the assumption that multiple distinct instance subtypes exist in the domain, and that instances belonging to these subtypes will thus be present in the random sample. By tuning a configuration to each instance, and therefore to each subtype, a portfolio of complementary solvers may emerge.

In our evaluations, we use $N = 20$ instances sampled from each domain to test this possibility in the set of experiments described next. We applied the same PARAMILS settings

[7] `http://www.cs.ubc.ca/labs/beta/Projects/ParamILS/`

as described in previous section, with the exception of running PARAMILS only once, instead of ten times, to tune CLASP on each of the 20 instances. Afterwards, we selected 16 configurations for WSEQ, 16 for NLP and 6 for RIP, which were found on instances with runtimes longer than 0.2 seconds on average. Typically, all inspected CLASP configurations performed comparably for these easy instances. Once again we utilized the BORG toolkit to build the kind of portfolio solvers described in Section 3.2 for each of the studied domains, in this case using the configurations found by PARAMILS.

6 Experimental Results

The experiments in this section compare and contrast the approaches discussed for handling solver sensitivity in ASP. To recap, we will compare strategies 1–6, summarized in the introduction, by measuring the performance of the BESTSINGLE, MAPP, and PARAMILS-based solvers trained with various CLASP configurations and on different training sets. In addition, we will present the performance of:

- CLASPFOLIO,
- the DEFAULT configuration of CLASP, and
- the ORACLE portfolio, also called the *virtual best solver*, which corresponds to the minimal run time on each instance given a portfolio approach with perfect knowledge.

The configurations found by PARAMILS are not included in the portfolios of BESTSINGLE, CLASPFOLIO, MAPP and ORACLE. All solver runs were collected on a local cluster (Xeon X5355 @ 2.66GHz) with a timeout set to 1,200 CPU seconds.

We use the standard technique of five-fold cross validation to get an unbiased evaluation. Each collection of instances is split into pairs of training and test sets. In five-fold cross validation, these pairs are generated by dividing the collection into five disjoint test sets of equal size, with the instances left out of each test set used to form each training set.

First, to illustrate the potential effectiveness of portfolio methods, Table 1 presents the performance of CLASPFOLIO and MAPP (trained with the CLASPFOLIO configurations on the CLASPFOLIO training set) on the ASPCOMP instances from the NP category in the system track. Note that the performance of CLASPFOLIO and of the MAPP$^+$ solver appear quite similar in this situation. This performance similarity allows us to take the MAPP$^+$ approach as representative of portfolio methods in general in our evaluations. These results show that portfolios are clearly effective on *heterogeneous* collections of instances that include multiple qualitatively different problem domains. It is less clear, however, that multiple complementary solver configurations exist across instances within a single problem domain.

To answer the question of whether portfolio solving provides any benefit on individual problem domains, Figure 1 presents performance curves for experiments run on each domain separately, under five-fold cross validation. BESTSINGLE, the MAPP portfolios, and ORACLE were all trained on each specific domain (using the CLASPFOLIO configurations). PARAMILS denotes the CLASP configuration found for each domain, as described in Section 4. On a domain included in its training set (WSEQ, 5 instances), CLASPFOLIO performs well, only slightly worse than a portfolio trained specifically on that domain. On domains not included in its training set—NLP and RIP—CLASPFOLIO is less effective, beating the DEFAULT configuration on NLP but losing to it on RIP. Lacking domain-specific training, then, CLASPFOLIO can struggle to identify good configurations. Portfolios trained for each domain (MAPP$^-$ and MAPP$^+$) consistently perform much better than DEFAULT and CLASPFOLIO. This improvement seems to be due to identifying a single good configuration: note that the MAPP$^-$

Table 1 The number of instances solved and the mean run time (MRT) on those solved instances for single-solver and portfolio strategies on the 125 ASPCOMP instances (with all portfolios trained on the CLASPFOLIO training set.)

	ASPCOMP	
Solver	**Solved**	**MRT (s)**
DEFAULT	75	82.99
BESTSINGLE	82	63.75
MAPP^{-}	82	63.75
MAPP^{+}	84	75.41
CLASPFOLIO	85	97.47
ORACLE	91	48.84

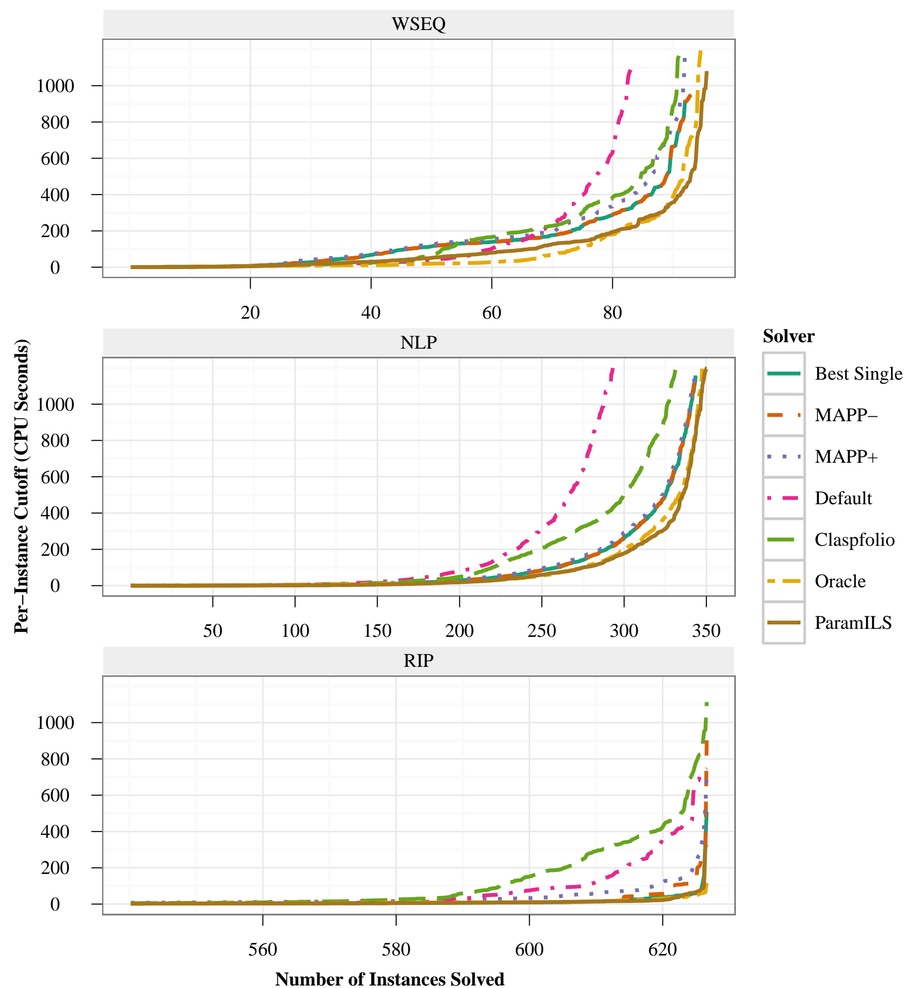


Figure 1 Cactus plots presenting the performance, under five-fold cross validation, of strategies 1-5 on the three application domains considered in this paper.

Table 2 Results summarizing the performance of "PARAMILS-based" portfolios, as described in Section 5, according to the mean number of instances solved and the mean run time on those instances. These scores were averaged over five-fold cross validation.

	WSEQ		NLP		RIP	
Solver	**Solved**	**MRT (s)**	**Solved**	**MRT (s)**	**Solved**	**MRT (s)**
BESTSINGLE (ILS)	93.40	163.95	343.80	95.27	626.20	1.39
MAPP$^-$ (ILS)	93.40	163.95	343.80	95.26	626.00	1.95
MAPP$^+$ (ILS)	93.80	170.52	342.40	100.61	626.00	5.48
ORACLE (ILS)	94.80	72.27	349.40	80.66	626.20	1.13

and BESTSINGLE solvers are almost identical in their performance. This hypothesis was confirmed by analyzing the solver execution schedule of MAPP$^-$: it turns out that MAPP$^-$, on these domains, may practically be identified with the BESTSINGLE solver approach. Comparing MAPP$^+$ and MAPP$^-$ performance, then, shows that feature-based prediction provides no benefit in these experiments. The feature computation overhead incurred by MAPP$^+$ and CLASPFOLIO on "easy" domains, such as RIP, is also evident. In these single-problem domains, in other words, the portfolio approach is useful for systematically identifying a good solver configuration, but struggles to make useful performance predictions from feature information.

In contrast, configurations tuned via PARAMILS perform very well. The performance of PARAMILS tracks that of the ORACLE portfolio. Both portfolios and algorithm configuration improve on the performance of DEFAULT by large margins.

Note also, however, that run time can be misleading. For example, DEFAULT is faster on some instances of the WSEQ domain, but solves fewer overall. These deceptive aspects of solver performance strongly suggest that an ASP application developer should employ a tool, such as a portfolio framework, to systematically collect and analyze solver performance. Cursory approaches, such as manually experimenting with only a few instances, can lead to the suboptimal selection of a configuration.

Since collecting training data from the entire domain incurs substantial cost, our recommendation would be to collect such data on a modest randomly sampled subset of instances. If configurations exhibit substantial differences in performance on that subset, and especially if the performance gap between the BESTSINGLE solver and the ORACLE portfolio is large, then additional training data may enable a portfolio method to make up some of that difference. Such decisions might also be made based on recently proposed formal definitions of instance set homogeneity [22].

Configurations found by PARAMILS provide substantial gains in performance on every domain. It is interesting to see that they perform nearly the same as the ORACLE portfolio of CLASPFOLIO configurations: if perfect algorithm selection were somehow available, we would not need to tune the configuration. Conversely, it is impressive that the range of configurations spanned by the CLASPFOLIO suite of solvers can be equaled by a single tuned configuration on these domains.

Table 2 presents details of the performance of portfolios obtained under the methodology described in Section 5. No further improvement in comparison to the PARAMILS configuration could be obtained under this methodology. Either a single configuration is sufficient to achieve maximum CLASP-derived performance on these domains, or a more sophisticated approach to portfolio construction must be used—the ISAC approach [15], for example, which attempts to explicitly identify subgroups of instances within the domain, or the Hydra

system, which accounts for overall portfolio performance in making tuning decisions [25]. This question is left to future investigation.

7 Conclusions

The results of this experimental study strongly recommend two courses of action for ASP application developers, one general and one specific. As a general recommendation, it is clear that significant care must be paid to solver parameterization in order to accurately characterize performance on a domain. Employing a portfolio toolkit to systematically collect run time data and select the best CLASPFOLIO configuration is a reasonable and straightforward first step. As a specific recommendation, however, the use of automatic algorithm configuration can wring more performance from a domain. Preparing such a tool, however, itself requires intimate knowledge of a specific solver. Solver authors could empower the solver's users by providing configuration files for PARAMILS or a related tool.

One final observation made clear by this work is the importance of understanding the desired solver performance objective. An ASP developer must carefully select an appropriate run time budget for their task, and must carefully weigh their desires for efficiency and consistent success. These desires may be in conflict, and the effectiveness of algorithm portfolio and configuration methods both depend on a user understanding and accurately specifying their own preferences.

The need for studies such as that conducted in this paper has also been expressed by Karp [16]. By looking at worst-case asymptotic performance over the space of all possible inputs, theoretical computer science typically predicts the intractability in general of the computational tasks exemplified by ASP or SAT. In practice, however, these challenging tasks can often be solved, thus driving the need for an experimental approach to the task of finding and evaluating algorithms for difficult search problems on specific domains. Karp writes:

> A tuning strategy [searches] the space of concrete algorithms consistent with the algorithmic strategy to find the one that performs best on the training set. Finally, an evaluation method compares the chosen algorithm with its competitors on a verification set of instances drawn from the same distribution as the training set.

The case study presented in this work, as well as the methodologies it explored, are steps toward refining such an experimental approach—an approach that appears essential to enabling a practitioner to evaluate and apply increasingly powerful, increasingly sensitive parameterized solvers.

Acknowledgments

We are grateful to Vladimir Lifschitz, Peter Schüller, and Miroslaw Truszczynski for useful discussions related to the topic of this work. Martin Brain, Peter Schüller, and Shaden Smith assisted us with the instances used in this case study. Yuliya Lierler was supported by a CRA/NSF 2010 Computing Innovation Fellowship.

References

1 C. Ansótegui, M. Sellmann, and K. Tierney. A Gender-Based Genetic Algorithm for the Automatic Configuration of Algorithms. In I. Gent, editor, *Proceedings of the CP'09*, volume 5732 of *Lecture Notes in Computer Science*, pages 142–157. Springer-Verlag, 2009.

2 M. Brain and F. Schanda. Riposte: Supporting development in spark using counter-examples. Unpublished manuscript, 2012.

3 F. Calimeri, G. Ianni, F. Ricca, M. Alviano, A. Bria, G. Catalano, S. Cozza, W. Faber, O. Febbraro, N. Leone, M. Manna, A. Martello, C. Panetta, S. Perri, K. Reale, M. Carmela Santoro, M. Sirianni, G. Terracina, and P. Veltri. The third answer set programming competition: Preliminary report of the system competition track. In Delgrande and Faber [4], pages 388–403.

4 J. Delgrande and W. Faber, editors. *Proceedings of the LPNMR'11*, volume 6645 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 2011.

5 M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and M. Schneider. Potassco: The Potsdam answer set solving collection. *AI Communications*, 24(2):105–124, 2011.

6 M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub. Challenges in answer set solving. In M. Balduccini and T. Son, editors, *Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning: Essays in Honor of Michael Gelfond*, volume 6565, pages 74–90. Springer-Verlag, 2011.

7 M. Gebser, R. Kaminski, B. Kaufmann, T. Schaub, M. Schneider, and S. Ziller. A portfolio solver for answer set programming: Preliminary report. In Delgrande and Faber [4], pages 352–357.

8 M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. Conflict-driven answer set solving. In *Proceedings of the IJCAI'07*, pages 386–392. MIT Press, 2007.

9 M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In Robert Kowalski and Kenneth Bowen, editors, *Proceedings of the ICLP'88*, pages 1070–1080. MIT Press, 1988.

10 E. Giunchiglia, Y. Lierler, and M. Maratea. Answer set programming based on propositional satisfiability. *Journal of Automated Reasoning*, 36:345–377, 2006.

11 C. Gomes, H. Kautz, A. Sabharwal, and B. Selman. Satisfiability solvers. In Frank van Harmelen, Vladimir Lifschitz, and Bruce Porter, editors, *Handbook of Knowledge Representation*, pages 89–134. Elsevier, 2008.

12 F. Hutter, H. Hoos, and K. Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *Proceedings of LION'11*, pages 507–523, 2011.

13 F. Hutter, H. Hoos, and K. Leyton-Brown. Parallel algorithm configuration. In *Proceedings of the LION'12*, 2012. To appear.

14 F. Hutter, H. Hoos, K. Leyton-Brown, and T. Stützle. ParamILS: An automatic algorithm configuration framework. *Journal of Artificial Intelligence Research*, 36:267–306, 2009.

15 S. Kadioglu, Y. Malitsky, M. Sellmann, and K. Tierney. ISAC—instance-specific algorithm configuration. In *Proceedings of the ECAI'10*, 2010.

16 R. Karp. Heuristic algorithms in computational molecular biology. *Journal of Computer and System Sciences*, 77(1):122–128, 2011.

17 Y. Lierler and P. Schüller. Parsing combinatory categorial grammar with answer set programming: Preliminary report. In *Workshop on Logic programming (WLP)*, 2011.

18 Y. Lierler, S. Smith, M. Truszczynski, and A. Westlund. Weighted-sequence problem: Asp vs casp and declarative vs problem oriented solving. In *Proceedings of the PADL'12)*, 2012.

19 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. Springer Verlag, 1999.

20 I. Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25:241–273, 1999.

21 O. Roussel. Controlling a Solver Execution with the runsolver Tool. *Journal on Satisfiability, Boolean Modeling and Computation*, 7(4):139–144, 2011.

22 M. Schneider and H. Hoos. Quantifying homogeneity of instance sets for algorithm configuration. In Y. Hamadi and M. Schoenauer, editors, *Proceedings of the LION'12*, 2012. Submitted for Post-Proceedings.

23 B. Silverthorn. *A Probabilistic Architecture for Algorithm Portfolios*. PhD thesis, The University of Texas at Austin, 2012.

24 B. Silverthorn and R. Miikkulainen. Latent class models for algorithm portfolio methods. In *Proceedings of the AAAI'10*, 2010.

25 L. Xu, H. Hoos, and K. Leyton-Brown. Hydra: Automatically configuring algorithms for portfolio-based selection. In *Proceedings of the AAAI'10*, 2010.

26 L. Xu, F. Hutter, H. Hoos, and K. Leyton-Brown. SATzilla: Portfolio-based algorithm selection for SAT. *Journal of Artificial Intelligence Research*, 32:565–606, 2008.

aspeed: ASP-based Solver Scheduling

Holger Hoos[1], Roland Kaminski[2], Torsten Schaub[3], and Marius Schneider[4]

1 **University of British Columbia, Canada**
`hoos@cs.ubc.ca`
2 **University of Potsdam, Germany**
`kaminski@cs.uni-potsdam.de`
3 **University of Potsdam, Germany**
`torsten@cs.uni-potsdam.de`
4 **University of Potsdam, Germany**
`manju@cs.uni-potsdam.de`

Abstract

Although Boolean Constraint Technology has made tremendous progress over the last decade, it suffers from a great sensitivity to search configuration. This problem was impressively counterbalanced at the 2011 SAT Competition by the rather simple approach of *ppfolio* relying on a handmade, uniform and unordered solver schedule. Inspired by this, we take advantage of the modeling and solving capacities of ASP to automatically determine more refined, that is, non-uniform and ordered solver schedules from existing benchmarking data. We begin by formulating the determination of such schedules as multi-criteria optimization problems and provide corresponding ASP encodings. The resulting encodings are easily customizable for different settings and the computation of optimum schedules can mostly be done in the blink of an eye, even when dealing with large runtime data sets stemming from many solvers on hundreds to thousands of instances. Also, its high customizability made it easy to generate even parallel schedules for multi-core machines.

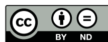
1998 ACM Subject Classification I.2.8 Problem Solving, Control Methods, and Search

Keywords and phrases Algorithm Schedule, Portfolio-based Solving, Answer Set Programming

Digital Object Identifier 10.4230/LIPIcs.ICLP.2012.176

1 Introduction

Boolean Constraint Technology has made tremendous progress over the last decade, leading to industrial-strength solvers. Although this advance in technology was mainly conducted in the area of Satisfiability Testing (SAT; [3]), it meanwhile also led to significant boosts in neighboring areas, like Answer Set Programming (ASP; [2]), Pseudo-Boolean Solving [3, Chapter 22], and even (multi-valued) Constraint Solving [21]. However, there is yet a prize to pay. Modern Boolean constraint solvers are rather sensitive to the way their search parameters are configured. Depending on the choice of the respective configuration, the solver's performance may vary by several orders of magnitude. Although this is a well-known issue, it was impressively laid bare once more at the 2011 SAT competition, where 16 prizes were won by the portfolio-based solver *ppfolio* [17]. The idea underlying *ppfolio* is very simple: it independently runs several solvers in parallel. If only one processing unit is available, three solvers are started. By relying on the operating system, each solver gets nearly the same time to solve a given instance. We refer to this as a uniform, unordered solver schedule. If more


© Holger Hoos, Roland Kaminski, Torsten Schaub, and Marius Schneider;
licensed under Creative Commons License ND
Technical Communications of the 28th International Conference on Logic Programming (ICLP'12).
Editors: A. Dovier and V. Santos Costa; pp. 176–187

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Table 1 Table of solver runtimes on problem instances with $\kappa = 10$.

	s_1	s_2	s_3	*oracle*
i_1	**1**	(11)	3	1
i_2	5	(11)	**2**	2
i_3	8	**1**	(11)	1
i_4	(11)	(11)	**2**	2
i_5	(11)	**6**	(11)	6
i_6	(11)	8	(11)	8
timeouts	3	3	3	0

processing units are available, one solver is in turn started on each unit; though multiple ones may end up on the last unit.

Inspired by this plain, yet successful system, we provide a more elaborate, yet still simple approach that takes advantage of the modeling and solving capacities of ASP to automatically determine more refined, that is, non-uniform and ordered solver schedules from existing benchmarking data. The resulting encodings are easily customizable for different settings. For instance, our approach is directly extensible to the generation of parallel schedules for multi-core machines. Also, the computation of optimum schedules can mostly be done in the blink of an eye, even when dealing with large runtime data sets stemming from many solvers on hundreds to thousands of instances. Unlike both, our approach does not rely on any domain-specific features, which makes it easily adaptable to other problems.

The remainder of this article is structured as follows. In Section 2, we formulate the determination of optimum schedules as multi-criteria optimization problems. In doing so, our primary emphasis lies in producing robust schedules that aim at the fewest number of timeouts by non-uniformly attributing each solver (or solver configuration) a different time slice. Once such a robust schedule is found, we optimize its runtime by selecting the best solver alignment. We next extend this approach to parallel settings in which multiple processing units are available. With these specifications at hand, we proceed in two steps. First, we provide an ASP encoding for computing (parallel) timeout-minimal schedules (Section 3). Once such a schedule is identified, we use the encoding to find a time-minimal alignment of its solvers (Section 4). Both ASP encodings reflect interesting features needed for dealing with large sets of runtime data. Finally, in Section 5, we provide a empirical evaluation of the resulting system *aspeed*, and we contrast it with related approaches (Section 6). In what follows, we presuppose a basic acquaintance with ASP (see [2] for a comprehensive introduction).

2 Solver Scheduling

Sequential Scheduling. Given a set I of problem instances and a set S of solver configurations, we use function $t : I \times S \mapsto \mathbb{R}$ to represent a table of solver runtimes on instances. Also, we use an integer κ to represent a given cutoff time.

For illustration, consider the runtime function in Table 1; it deals with 6 problem instances, i_1 to i_6, and 3 solvers, s_1, s_2, and s_3. Each solver can solve three out of six instances within the cutoff time, $\kappa = 10$. A timeout is represented in Table 1 by 11, that is, an increased cutoff time. The oracle solver, also called virtually best solver, is obtained by assuming the best performance of each individual solver. As we see in the rightmost column, the oracle

would allow for solving all instances in our example within the cutoff time. Thus, if we knew beforehand which solver to choose for each instance, we could solve all of them. Unlike this, we can already obtain an improvement by successively running each solver within a limited time slice rather than running one solver until cutoff. For instance, running s_1 for 1, s_2 for 6, and s_3 for 2 seconds allows us to solve 5 out of 6 instances, as indicated in bold in Table 1. In what follows, we show how such a schedule can be obtained beforehand from given runtime data.

Given I, S, t, and κ as specified above, a timeout-optimal solver *schedule* can be expressed as an unordered tuple σ, represented as a function $\sigma : S \to [0, \kappa]$, satisfying the following condition:

$$\begin{aligned} \sigma \in \arg\max\nolimits_{\sigma:S\to[0,\kappa]} & \; |\{i \mid t(i,s) \leq \sigma(s), (i,s) \in I \times S\}| \\ \text{such that} & \quad \textstyle\sum_{s\in S}\sigma(s) \leq \kappa \end{aligned} \tag{1}$$

An optimal schedule σ consists of slices $\sigma(s)$ indicating the (possibly zero) time allotted to each solver $s \in S$. Such a schedule maximizes the number of solved instances, or conversely, minimizes the number of obtained timeouts.

The above example corresponds to the schedule $\sigma = \{s_1 \mapsto 1, s_2 \mapsto 6, s_3 \mapsto 2\}$; in fact, σ constitutes one among nine timeout-optimal solver schedules in our example. Note that the sum of all time slices is even smaller than the cutoff time. Hence, all schedules obtained by adding 1 to either of the three solvers are also timeout-optimal. A timeout-optimal schedule consuming the entire allotted time is $\{s_1 \mapsto 0, s_2 \mapsto 8, s_3 \mapsto 2\}$.

In practice, however, the criterion in (1) turns out to be too coarse, that is, it yields many heterogeneous solutions among which we would like to make an educated choice. To this end, we take advantage of L-norms for regulating the selection. In our case, an L^n-norm on schedules is defined as[1] $\Sigma_{s\in S,\sigma(s)\neq 0}\, \sigma(s)^n$. Depending upon the choice of n as well as whether we minimize or maximize the norm, we obtain different selection criteria. For instance, L^0-norms suggest using as few/many solvers as possible and L^1-norms aim at minimizing/maximizing the sum of time slices. Minimizing the L^2-norm amounts to allotting each solver a similar time slice, while maximizing it prefers schedules with large runtimes for few solvers. In more formal terms, an L^n-norm gives rise to objective functions of the following form. For a set of schedules of a set S of solvers, we define:

$$\sigma \in \arg\min\nolimits_{\sigma:S\to[0,\kappa]} \Sigma_{s\in S,\sigma(s)\neq 0}\, \sigma(s)^n \tag{2}$$

An analogous function is obtained for maximization with $\arg\max$.

For instance, our exemplary schedule $\sigma = \{s_1 \mapsto 1, s_2 \mapsto 6, s_3 \mapsto 2\}$ has the L^i-norms 3, 9, and 41 for $i = 0..2$. For a complement, we get norms 3, 9, and 27 for the (suboptimal) uniform schedule $\{s_1 \mapsto 3, s_2 \mapsto 3, s_3 \mapsto 3\}$ and 1, 9, and 81 for a singular schedule $\{s_3 \mapsto 9\}$, respectively. Although we empirically discovered no clear edge of the latter, we favor a schedule with a minimal L^2-norm. First, it leads to a significant reduction of candidate schedules and, second, it results in schedules with a most homogeneous distribution of time slices, similar to *ppfolio*. In fact, our exemplary schedule has the smallest L^2-norm among all nine timeout-optimal solver schedules.

Once we have identified a most robust schedule wrt criteria (1) and (2), it is interesting to know which solver alignment yields the best performance as regards time. More formally,

[1] The common L^n-norm is defined as $\sqrt[n]{\Sigma_{x\in X}x^n}$. We take the simpler definition in view of using it merely for optimization.

we define an *alignment* of a set S of solvers as the bijective function $\pi : \{1, \dots, |S|\} \to S$. Consider the above schedule $\sigma = \{s_1 \mapsto 1, s_2 \mapsto 6, s_3 \mapsto 2\}$. The alignment $\pi = \{1 \mapsto s_1, 2 \mapsto s_3, 3 \mapsto s_2\}$ induces the execution sequence (s_1, s_3, s_2) of σ. This sequence solves all six benchmarks in Table 1 in 29 seconds; in detail, it takes $1, 1{+}2, 1{+}2{+}1, 1{+}2, 1{+}2{+}6, 1{+}2{+}6$ seconds for benchmark i_k for $k = 1..6$. Note that benchmark i_3 is successfully solved by the third solver in the alignment, viz. s_2. Hence the total time amounts to the allotted time by σ to s_1 and s_3, viz. $\sigma(s_1)$ and $\sigma(s_3)$, plus the effective time of s_2, viz. $t(i_3, s_2)$. Because the timeout-minimal time slices are given, we do not distinguish whether an alignment solves a benchmark after the total time of the schedule or not. For instance, our exemplary alignment π takes 9 seconds on both i_5 and i_6, although it only solves the former but not the latter.

This can be made precise as follows. Given a schedule σ and an alignment π of a set S of solvers, and an instance $i \in I$, we define:

$$\tau_{\sigma,\pi}(i) = \begin{cases} \left(\sum_{j=1}^{\min(P)-1} \sigma(\pi(j))\right) + t(i, \pi(\min(P))) & \text{if } P \neq \emptyset, \\ \kappa & \text{otherwise} \end{cases} \tag{3}$$

where $P = \{l \in \{1, \dots, |S|\} \mid t(i, \pi(l)) \leq \sigma(\pi(l))\}$. While $\min P$ gives the position of the first solver solving instance i in a schedule σ aligned by π, $\tau_{\sigma,\pi}(i)$ gives the total time to solve instance i by schedule σ aligned by π. If an instance i cannot be solved at all by a schedule, $\tau_{\sigma,\pi}(i)$ is set to the cutoff κ. For our exemplary schedule σ and its alignment π, we get for i_3: $\min P = 3$ and $\tau_{\sigma,\pi}(i_3) = 1 + 2 + 1 = 4$.

For a schedule σ of solvers in S, we then define:

$$\pi \in \arg\min_{\pi:\{1,\dots,|S|\} \to S} \sum_{i \in I} \tau_{\sigma,\pi}(i) \tag{4}$$

For our timeout-optimal schedule $\sigma = \{s_1 \mapsto 1, s_2 \mapsto 6, s_3 \mapsto 2\}$ wrt criteria (1) and (2), we obtain two optimal alignments, yielding execution alignments (s_3, s_1, s_2) and (s_1, s_3, s_2), both of which result in a solving time of 29 seconds.

Parallel Scheduling. The increasing availability of multi-core processors makes it interesting to extend our approach for distributing a schedule's solvers on different processing units. For simplicity, we take a coarse approach in binding solvers to units, thus precluding re-allocations during runtime.

To begin with, let us provide a formal specification of the extended problem. To this end, we augment our ensemble of concepts with a set U of (processing) units and associate each unit with subsets of solvers from S. More formally, we define a *distribution* of a set S of solvers as the function $\eta : U \to 2^S$ such that $\bigcap_{u \in U} \eta(u) = \emptyset$. With it, we can determine timeout-optimal solver schedules for several cores simply by strengthening the condition in (1) to the effect that all solvers associated with the same unit must respect the cutoff time. This leads us to the following extension of (1):

$$\begin{aligned} &\sigma \in \arg\max_{\sigma:S \to [0,\kappa]} |\{i \mid t(i,s) \leq \sigma(s), (i,s) \in I \times S\}| \\ &\quad \text{such that} \qquad \sum_{s \in \eta(u)} \sigma(s) \leq \kappa \ \text{ for each } u \in U \end{aligned} \tag{5}$$

For illustration, let us reconsider Table 1 along with schedule $\sigma = \{s_1 \mapsto 1, s_2 \mapsto 8, s_3 \mapsto 2\}$. Assume that we have two cores, 1 and 2, along with the distribution $\eta = \{1 \mapsto \{s_2\}, 2 \mapsto \{s_1, s_3\}\}$. This distributed schedule solves all benchmarks in Table 1 with a cutoff of $\kappa = 8$. Hence, it is an optimal solution to the optimization problem in (5).

We keep the definitions of a schedule's L^n-norm as a global constraint.

For determining our secondary criterion, enforcing time-optimal schedules, we relativize the auxiliary definitions in (3) to account for each unit separately. Given a schedule σ and a

set U of units, we define for each unit $u \in U$ a *local alignment* of the solvers in $\eta(u)$ as the bijective function $\pi_u : \{1, \dots, |\eta(u)|\} \to \eta(u)$. Given this and an instance $i \in I$, we extend the definitions in (3) as follows:

$$\tau_{\sigma,\pi_u}(i) = \begin{cases} \left(\sum_{j=1}^{\min(P)-1} \sigma(\pi_u(j))\right) + t(i, \pi_u(\min(P))) & \text{if } P \neq \emptyset, \\ \kappa & \text{otherwise} \end{cases} \tag{6}$$

where $P = \{l \in \{1, \dots, |\eta(u)|\} \mid t(i, \pi_u(l)) \leq \sigma(\pi_u(l))\}$.

The collection $(\pi_u)_{u\in U}$ regroups all local alignments into a *global alignment*. For a schedule σ of solvers in S and a set U of (processing) units, we then define:

$$(\pi_u)_{u\in U} \in \arg\min_{(\pi_u:\{1,\dots,|\eta(u)|\}\to\eta(u))_{u\in U}} \sum_{i\in I} \min_{u\in U} \tau_{\sigma,\pi_u}(i) \tag{7}$$

For illustration, reconsider the above distribution and suppose we chose the local aligments $\pi_1 = \{s_2 \mapsto 1\}$ and $\pi_2 = \{s_1 \mapsto 1, s_3 \mapsto 2\}$. This global alignment solves all six benchmark instances in 22 seconds. In more detail, it takes $1_2, 1 + 2_2, 1_1, 1 + 2_2, 6_1, 8_1$ seconds for benchmark i_k for $k = 1..6$, where the solving unit is indicated by the subscript.

Note that the definitions in (5), (6), and (7) correspond to their sequential counterparts in (1), (3), and (4) whenever we are faced with a single processing unit.

3 Solving Timeout-Optimal Scheduling with ASP

To begin with, we detail the basic encoding for identifying robust (parallel) schedules. In view of the remark at the end of the last section, however, we directly provide an encoding for parallel scheduling, which collapses to one for sequential scheduling whenever a single processing unit is used.

Following good practice in ASP, a problem instance is expressed as a set of facts. That is, Function $t : I \times S \mapsto \mathbb{R}$ is represented as facts of form `time(i,s,t)`, where $\mathtt{i} \in I$, $\mathtt{s} \in S$, and `t` is the runtime $t(\mathtt{i}, \mathtt{s})$ converted to a natural number with a limited precision. The cutoff is expressed via Predicate `kappa/1`. And the number of available processing units is captured via Predicate `units/1`, here instantiated with 2 cores. Given this, we can represent the contents of Table 1 as follows.

```
kappa(10).
units(2).

time(i1, s1,  1).  time(i1, s2, 11).  time(i1, s3,  3).
time(i2, s1,  5).  time(i2, s2, 11).  time(i2, s3,  2).
time(i3, s1,  8).  time(i3, s2,  1).  time(i3, s3, 11).
time(i4, s1, 11).  time(i4, s2, 11).  time(i4, s3,  2).
time(i5, s1, 11).  time(i5, s2,  6).  time(i5, s3, 11).
time(i6, s1, 11).  time(i6, s2,  8).  time(i6, s3, 11).
```

The encoding in Listing 1 along with all following ones are given in the input language of *gringo*, documented in [7]. The first three lines of Listing 1 provide auxiliary data. The set S of solvers is given by Predicate `solver/1`. Similarly, the runtimes for each solver are expressed by `time/2`. In addition, the ordering `order/3` of instances by time per solver is precomputed.

```
order(I,K,S) :-
  time(I,S,T), time(K,S,V), (T,I) < (V,K),
  not time(J,S,U) : time(J,S,U) : (T,I) < (U,J) : (U,J) < (V,K).
```

The above results in facts `order(I,K,S)` capturing that instance `I` is solved immediately before instance `K` by solver `S`. Although this information could be computed via ASP (as shown above), we make use of external means for sorting (the above rule needs cubic time for instantiation, which is infeasible for a few thousand instances).[2]

The idea is now to guess for each solver a time slice and a processing unit. With the resulting schedule, all solvable instances can be identified. And finally all schedules solving most instances are selected.

Listing 1 ASP encoding for Timeout-Minimal (Parallel) Scheduling.

```
1  solver(S)  :- time(_,S,_).
2  time(S,T)  :- time(_,S,T).
3  unit(1..N) :- units(N).

5  {slice(U,S,T): time(S,T): T <= K: unit(U)} 1 :- solver(S),kappa(K).
6  slice(S,T) :- slice(_,S,T).

8   :- not [ slice(U,S,T) = T ] K, kappa(K), unit(U).

10 solved(I,S) :- slice(S,T), time(I,S,T).
11 solved(I,S) :- solved(J,S), order(I,J,S).
12 solved(I)   :- solved(I,_).

14 #maximize { solved(I) @ 2 }.
15 #minimize [ slice(S,T) = T*T @ 1 ].
```

A schedule is represented by atoms `slice(U,S,T)` allotting a time slice `T` to solver `S` on unit `U`. In Line 5, at most one time slice is chosen for each solver subject to the trivial condition that it is equal or less the cutoff time. At the same time, a processing unit is uniquely assigned to the selected solver. The following line projects out the processing unit because it is irrelevant when determining solved instances (in Line 10). The integrity constraint in Line 8 ensures that the sum over all selected time slices on each core is not greater than the cutoff time. This implements the side condition in (5); and it reduces to the one in (1) whenever a single unit is considered. In lines 10 to 12, all instances solved by the selected time slices are gathered via predicate `solved/1`. Given that we collect in Line 6 all time slices among actual runtimes, each time slice allows for solving at least one instance. This property is used in Line 10 to identify the instance `I` solvable by solver `S`. Given this and the sorting of instances by solver performance in `order/3`, we collect in Line 11 all instances that can be solved even faster than the instance in Line 10. Note that at first sight it might be tempting to encode this differently:

```
solved(I) :- slice(S,T), time(I,S,TS), T <= TS.
```

The problem with the above rule is that it has a quadratic number of instantiations in the number of benchmark instances in the worst case. Unlike this, our ordering-based encoding is linear because only successive instances are considered. Finally, the number of solved instances is maximized in Line 14, following the recipe in (5) (or (1), respectively). This major objective gets a higher priority, viz. `2`, than the L^2-norm from (2) having priority `1`.

[2] To be precise, we use *gringo*'s embedded scripting language *lua* for sorting.

4 Solving (Timeout and) Time-Minimal Parallel Scheduling with ASP

In the previous section, we have determined a timeout-minimal schedule. Here, we present an encoding that takes such a schedule and calculates a solver alignment per processing unit while minimizing the overall runtime according to Criterion (7). This two-phase approach is motivated by the fact that an optimal alignment must be determined among all permutations of a schedule. While a one shot approach had to account for all permutations of all potential timeout-minimal schedules, our two-phase approach reduces the second phase to searching among all permutations of a single timeout-minimal schedule.

We begin by extending the problem instance of the last section (in terms of `kappa/1`, `units/1`, and `time/3`) by facts over `slice/3` providing the time slices of a timeout-minimal schedule (per solver and processing unit). To take on our example from Section 2, we use the obtained timeout-minimal schedule to create the following problem instance:

```
kappa(10). units(2).
time(i1, s1,  1).  time(i1, s2, 11).  time(i1, s3,  3).
...
slice(1,s2,8). slice(2,s1,1). slice(2,s3,2).
```

The idea of the encoding in Listing 2 is to guess a permutation of solvers and then to use ASP's optimization capacities for calculating a time-minimal alignment. The challenging part is to keep the encoding compact. That is, we have to keep the size of the instantiation of the encoding small because otherwise we fail to solve common problems with thousands of benchmark instances. To do this, we make use of `#sum` aggregates with negative weights to find the fastest processing unit without representing any sum of times explicitly.

Listing 2 ASP encoding for Time-Minimal (Parallel) Scheduling.

```
1  solver(U,S)      :- slice(U,S,_).
2  instance(I)      :- time(I,_,_).
3  unit(1..N)       :- units(N).
4  solvers(U,N)     :- unit(U), N := {solver(U,_)}.
5  solved(U,S,I)    :- time(I,S,T), slice(U,S,TS), T <= TS.
6  solved(U,I)      :- solved(U,_,I).
7  capped(U,I,S,T) :- time(I,S,T), solved(U,S,I).
8  capped(U,I,S,T) :- slice(U,S,T), solved(U,I), not solved(U,S,I).
9  capped(U,I,d,K) :- unit(U), kappa(K), instance(I), not solved(U,I).
10 capped(I,S,T)    :- capped(_,I,S,T).

12 1 { order(U,S,X) : solver(U,S) } 1 :- solvers(U,N), X = 1..N.
13 1 { order(U,S,X) : solvers(U,N) : X = 1..N } 1 :- solver(U,S).

15 solvedAt(U,I,X+1) :- solved(U,S,I), order(U,S,X).
16 solvedAt(U,I,X+1) :- solvedAt(U,I,X), solvers(U,N), X <= N.

18 mark(U,I,d,K) :- capped(U,I,d,K).
19 mark(U,I,S,T) :- capped(U,I,S,T), order(U,S,X), not solvedAt(U,I,X).
20 min(1,I,S,T)   :- mark(1,I,S,T).

22 less(U,I) :- unit(U), unit(U+1), instance(I),
23   [min(U,I,S1,T1): capped(I,S1,T1) = T1, mark(U+1,I,S2,T2) = -T2] 0.

25 min(U+1,I,S,T) :- min(U,I,S,T), less(U,I).
26 min(U,I,S,T)    :- mark(U,I,S,T), not less(U-1,I).

28 #minimize [min(U,_,_,T): not unit(U+1) = T].
```

The block in Line 1 to 10 gathers static knowledge about the problem instance, that is, solvers per processing unit (`solver/2`), instances appearing in the problem description (`instance/1`), available processing units (`unit/1`), number of solvers per unit (`solvers/2`), instances solved by a solver within its allotted slice (`solved/3`), and instances that could be solved on a unit given the schedule (`solved/2`). In view of Equation (6), we precompute the times that contribute to the values of τ_{σ,π_u} and capture them in `capped/4` (and `capped/3`). A fact `capped(U,I,S,T)` assigns to instance `I` run by solver `S` on unit `U` a time `T`. In Line 7, we assign the time needed to solve the instance if it is within the solver's time slice. In Line 8, we assign the solver's time slice if the instance could not be solved but at least one other solver could solve it on the processing unit. In Line 9, we assign the whole cutoff to dummy solver `d` (we assume that there is no other solver called `d`) if the instance could not be solved on the processing unit at all; this is to implement the else case in (6) and (3).

The actual encoding starts in Line 12 and 13 by guessing a permutation of solvers. Here the two head aggregates ensure that for every solver (per unit) there is exactly one index and vice versa. In Line 15 and 16, we mark indexes (per unit) as solved if the solver with the preceding index could solve the instance or if the previous index was marked as solved. Note that this is a similar "chain construction" as done in the previous section in order to avoid a combinatorial blow-up.

In the block from Line 18 to 26, we determine the time for the fastest processing unit depending on the guessed permutation. The rules in Line 18 and 19 mark the necessary times that have to be added up on each processing unit. The sums of the marked times correspond to $\tau_{\sigma,\pi_u}(i)$ in Equation (6) and (3). Next, we determine the smallest sum of times. Therefore, we iteratively determine the minimum. An atom `min(U,I,S,T)` marks the times of the fastest unit in the range from unit `1` to `U` to solve an instance (or the cutoff via dummy solver `d` if the schedule does not solve the instance for the unit). To begin with, we initialize `min/4` with the times for the first unit in Line 20. Then, we add a rule in Line 22 and 23 that, given minimal times for units in the range of `1` to `U` and times for unit `U+1`, determines the faster one. The current minimum contributes positive times to the sum, while unit `U+1` contributes negative times. Hence, if the sum is negative or zero, the sum of times captured in `min`/4 is smaller or equal to the sum of times of unit `U+1` and the unit thus slower than some preceding unit, which makes the aggregate true and derives the corresponding atom over `less/2`. Depending on `less/2`, we propagate the smaller sum, which is either contributed by the preceding units (Line 25) or the unit `U+1` (Line 26). Finally, in Line 28 the times of the fastest processing unit are minimized in the optimization statement, which implements Equation (7) and (4).

5 Experiments

After describing the theoretical foundations and ASP encodings underlying our approach, we now present some short results from an empirical evaluation. The python implementation of our solver, dubbed *aspeed*, uses the ASP systems [4] of the potassco group [6], namely grounder the *gringo* (3.0.4) and the ASP solver *clasp* (2.0.5). The sets of runtime data (including a list of the solvers and instances) used in this work are freely available online [1].

To provide a thorough empirical evaluation of our approach, we selected five large data sets of runtimes for two prominent and widely studied problems, SAT and ASP. The sets *Random*, *Crafted* and *Application* contain the runtimes taken from the 2011 SAT Competition [18]; the *3s-Set* is the training set of the portfolio SAT solver *3s* [13]; and the ASP instance set (*ASP-Set*) contains runtimes based on different configurations of the highly parametric ASP solver *clasp* [8].

Table 2 Comparison of different approaches w.r.t. #timeouts for a cutoff time of 5000 CPU seconds for *Random* ($|I| = 600$, $|S| = 9$), *Crafted* ($|I| = 300$, $|S| = 15$), *Application* ($|I| = 300$, $|S| = 18$) and *3s-Set* ($|I| = 5467$, $|S| = 37$) and 600 seconds for *ASP-Set* ($|I| = 313$, $|S| = 8$).

	Random		*Crafted*		*Application*		*3s-Set*		*ASP-Set*	
Single Best	254	(42.3%)	155	(51.6%)	85	(28.3%)	1881	(34.4%)	28	(8.9%)
Uniform	155	(25.8%)	123	(41.5%)	116	(38.6%)	1001	(18.3%)	29	(9.2%)
ppfolio-like	127	(21.1%)	126	(42.0%)	82	(27.3%)	645	(11.8%)	17	(5.4%)
satzilla	115	(19.2%)	101	(34.0%)	74	(24.7%)	−−	(−%)	−−	(−%)
aspeed (seq)	131	(21.8%)	98	(32.6%)	83	(27.6%)	536	(9.8%)	18	(5.7%)
aspeed (par 8)	109	(18.2%)	85	(28.3%)	51	(17.0%)	140	(2.5%)	8	(2.6%)
Oracle	108	(18%)	77	(26%)	45	(15%)	0	(0%)	4	(1.3%)

Based on these data sets, we compare sequential *aspeed* and parallel *aspeed* with eight cores (*par 8*) against the best solver in the portfolio (*Single Best*), a uniform distribution of the time slices over all solvers in the portfolio (*Uniform*), the *Oracle* performance (also called *virtual best solver*) and two SAT solvers: a *ppfolio-like* approach inspired by the single-threaded version of *ppfolio*, where the best three complementary solvers are selected with a uniform distribution of time slices, and *satzilla* [23] based on the results of [24] as a representative of a sequential portfolio-based algorithm selector. To obtain an unbiased evaluation of performance, we used 10-fold cross validation. Table 2 shows the number of timeouts and, in brackets, the corresponding fraction of the instance set; hence, small numbers indicate better performance. In all cases, *aspeed* showed better performance than the *Single Best* solver. *aspeed* performed better than *ppfolio-like* in three out of five settings, namely on *Crafted*, *3s-Set* and *ASP-Set*, and better than *satzilla* in one out of three settings, namely, *Crafted*.

6 Related Work

Our work forms part of a long line of research that can be traced back to John Rice's seminal work on algorithm selection [16] on one side, and to work by Huberman, Lukos, and Hogg [12] on parallel algorithm portfolios on the other side.

Most recent work on algorithm selection is focused on mapping problem instances to a given set of algorithms, where the algorithm to be run on a given problem instance i is typically determined based on a set of (cheaply computed) features of i. This is the setting considered prominently by Rice [16], as well as by the work on SATzilla, which makes use of regression-based models of running time [22, 23]; work on the use of decision trees and case-base reasoning for selecting bid evaluation algorithms in combinatorial auctions [10, 5]; and work on various machine learning techniques for selecting algorithms for finding maximum probable explanations in Bayes nets in real time [11]. All these approaches are similar to ours in that they exploit complementary strengths of a set of solvers for a given problem; however, unlike these per-instance algorithm selection methods, *aspeed* selects and schedules solvers to optimize performance on a set of problem instances, and therefore does not require instance features.

cphydra is a portfolio-based procedure for solving constraint programming problems that is based on case-based reasoning for solver selection and a simple complete search procedure for sequential solver scheduling [15]. Like the previously mentioned approaches, and unlike *aspeed*, it requires instance features for solver selection, and, according to its authors, is limited to a low number of solvers (in their work, five). Like the simplest variant of *aspeed*,

the solver scheduling in *cphydra* aims to maximize the number of given problem instances solved within a given time budget.

Early work on parallel algorithm portfolios highlights the potential for performance improvements, but does not provide automated procedures for selecting the solvers to be run in parallel from a larger base set [12, 9]. *ppfolio*, which demonstrated impressive performance at the 2011 SAT Competition, is a simple procedure that runs between 3 and 5 SAT solver concurrently (and, depending on the number of processors or cores available, potentially in parallel) on a given SAT instance. The component solvers have been chosen manually based on performance on past competition instances, and they are all run for the same amount of time. Unlike *ppfolio*, our approach automatically selects solvers to minimize the number of timeouts or total running time on given training instances using a powerful ASP solver and can, at least in principle, work with much larger numbers of solvers. Furthermore, unlike *ppfolio*, *aspeed* can allot variable amounts of time to each solver to be run as part of a sequential schedule.

Concurrently with our work presented here, Yun and Epstein [25] developed an approach that builds sequential and parallel solver schedules using case-based reasoning in combination with a greedy construction procedure. Their RSR-WG procedure combines fundamental aspects of *cphydra* [15] and GASS [20]; unlike *aspeed*, it relies on instance features. RSR-WG uses a relatively simple greedy heuristic to optimize the number of problem instances solved within a given time budget by the parallel solver schedule to be constructed; our use of an ASP encoding, on the other hand, offers considerably more flexibility in formulating the optimization problem to be solved, and our use of powerful, general-purpose ASP solvers can at least in principle find better schedules. Our approach also goes beyond RSR-WG in that it permits the optimization of parallel schedules for runtime.

Perhaps most closely related to our approach is the recent work of Kadioglu et al. on algorithm selection and scheduling [13]. They study pure algorithm selection and various scheduling procedures based on mixed integer programming techniques. Unlike *aspeed*, their more sophisticated procedures rely on instance features for nearest-neighbour-based solver selection, based on the (unproven) assumption that any given solver shows similar performance on instances with similar features [14]. (We note that in the literature on artificially created, 'uniform random' SAT and CSP instances there is some evidence suggesting that at least with the cheaply computable features that can be practically exploited by per-instance algorithm selection approaches this assumption may not hold.) We focussed deliberately on a simpler setting than their best-performing semi-static scheduling approach in that we do not use per-instance algorithm selection, yet still obtain excellent performance; furthermore, we consider the more general case of parallel solver schedules, while their work is limited to sequential execution of solvers.

7 Conclusion

In this work, we demonstrated how ASP formulations and a powerful ASP solver (*clasp*) can be used to compute sequential and parallel solver schedules. Compared with earlier model-free and model-based approaches (*ppfolio* and *satzilla*, respectively), our new procedure, *aspeed*, performs very well on SAT and ASP – two widely studied problems for which substantial and sustained effort is being expended in the design and implementation of high-performance solvers.

aspeed is open-source and available online [1]. We expect *aspeed* to work particularly well in situations where various different kinds of problem instances have to be solved (e.g.,

competitions) or where single good (or even dominant) solvers or solver configurations are unknown (e.g., new applications). Our approach leverages the power of multi-core and multi-processor computing environments and, because of its use of easily modifiable and extensible ASP encodings, can in principle be readily modified to accommodate different constraints on and optimization criteria for the schedules to be constructed. Unlike most other portfolio-based approaches, *aspeed* does not require instance features and can therefore be applied more easily to new problems.

Because, like various other approaches, *aspeed* is based on minimisation of timeouts, it is currently only applicable in situations where some instances cannot be solved within the time budget under consideration (this setting prominently arises in many solver competitions). In future work, we intend to investigate strategies that automatically reduce the time budget if too few timeouts are observed on training data; we are also interested in the development of better techniques for directly minimizing runtime.

In situations where there is a solver or configuration that dominates all others across the instance set under consideration, portfolio-based approaches are generally not effective (with the exception of performing multiple independent run of a randomized solver). The degree to which performance advantages can be obtained through the use of portfolio-based approaches, and in particular *aspeed*, depends on the degree to which there is complementarity between different solvers or configurations, and it would be interesting to investigate this dependence quantitatively, possibly based on recently proposed formal definitions of instance set homogeneity [19].

Acknowledgments

This work was partially funded by the German Science Foundation (DFG) under grant SCHA 550/8-2.

References

1 aspeed. Available at `http://www.cs.uni-potsdam.de/aspeed/`.

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

3 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.

4 F. Calimeri, G. Ianni, F. Ricca, M. Alviano, A. Bria, G. Catalano, S. Cozza, W. Faber, O. Febbraro, N. Leone, M. Manna, A. Martello, C. Panetta, S. Perri, K. Reale, M. Santoro, M. Sirianni, G. Terracina, and P. Veltri. The third answer set programming competition: Preliminary report of the system competition track. In *Proceedings of the Eleventh International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'11)*, volume 6645 of *Lecture Notes in Artificial Intelligence*, pages 388–403. Springer-Verlag, 2011.

5 C. Gebruers, A. Guerri, B. Hnich, and M. Milano. Making choices using structure at the instance level within a case based reasoning framework. In *Proceedings of the First Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, volume 3011 of *Lecture Notes in Computer Science*, pages 380–386. Springer, 2004.

6 M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and M. Schneider. Potassco: The Potsdam answer set solving collection. *AI Communications*, 24(2):105–124, 2011.

7. M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and S. Thiele. A user's guide to `gringo`, `clasp`, `clingo`, and `iclingo`.
8. M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. Conflict-driven answer set solving. In *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI'07)*, pages 386–392. AAAI Press/The MIT Press, 2007.
9. C. Gomes and B. Selman. Algorithm portfolios. *Artificial Intelligence*, 126(1-2):43–62, 2001.
10. A. Guerri and M. Milano. Learning techniques for automatic algorithm portfolio selection. In *Proceedings of the Sixteenth European Conference on Artificial Intelligence (ECAI'04)*, pages 475–479, 2004.
11. H. Guo and W. Hsu. A learning-based algorithm selection meta-reasoner for the real-time MPE problem. In *Proceedings of the Seventeenth Australian Joint Conference on Artificial Intelligence*, pages 307–318. Springer, 2004.
12. B. Huberman, R. Lukose, and T. Hogg. An economic approach to hard computational problems. *Science*, 27:51–53, 1997.
13. S. Kadioglu, Y. Malitsky, A. Sabharwal, H. Samulowitz, and M. Sellmann. Algorithm Selection and Scheduling. In *Proceedings of the Seventeenth International Conference on Principles and Practice of Constraint Programming (CP'11)*, volume 6876 of *Lecture Notes in Computer Science*, pages 454–469. Springer-Verlag, 2011.
14. S. Kadioglu, Y. Malitsky, M. Sellmann, and K. Tierney. ISAC – instance-specific algorithm configuration. In *Proceedings of the Nineteenth European Conference on Artificial Intelligence (ECAI'10)*, pages 751–756. IOS Press, 2010.
15. E. O'Mahony, E. Hebrard, A. Holland, C. Nugent, and B. O'Sullivan. Using case-based reasoning in an algorithm portfolio for constraint solving. In *Proceedings of the Nineteenth Irish Conference on Artificial Intelligence and Cognitive Science (AICS'08)*, 2008.
16. J. Rice. The algorithm selection problem. *Advances in Computers*, 15:65–118, 1976.
17. O. Roussel. Description of ppfolio, 2011.
18. SATComp11. Available at `http://www.cril.univ-artois.fr/SAT11/`.
19. M. Schneider and H. Hoos. Quantifying homogeneity of instance sets for algorithm configuration. In *Proceedings of the Sixth International Conference Learning and Intelligent Optimization (LION'12)*, *Lecture Notes in Computer Science*. Springer-Verlag, 2012.
20. M. Streeter, D. Golovin, and S. Smith. Combining multiple heuristics online. In *Proceedings of the Twenty-second National Conference on Artificial Intelligence (AAAI'07)*, pages 1197–1203. AAAI Press, 2007.
21. N. Tamura, A. Taga, S. Kitagawa, and M. Banbara. Compiling finite linear CSP into SAT. *Constraints*, 14(2):254–272, 2009.
22. L. Xu, H. Hoos, and K. Leyton-Brown. Hierarchical Hardness Models for SAT. In *Proceedings of the Thirteenth International Conference on Principles and Practice of Constraint Programming (CP'07)*, volume 4741 of *Lecture Notes in Computer Science*, pages 696–711. Springer-Verlag, 2007.
23. L. Xu, F. Hutter, H. Hoos, and K. Leyton-Brown. SATzilla: Portfolio-based algorithm selection for SAT. *Journal of Artificial Intelligence Research*, 32:565–606, 2008.
24. L. Xu, F. Hutter, H. Hoos, and K. Leyton-Brown. Detailed SATzilla Results from the Data Analysis Track of the 2011 SAT Competition. Technical report, University of British Columbia, 2011.
25. X. Yun and S. Epstein. Learning algorithm portfolios for parallel execution. In *Proceedings of the Sixth International Conference Learning and Intelligent Optimization (LION'12)*, *Lecture Notes in Computer Science*, Springer-Verlag, 2012.

Answer Set Solving with Lazy Nogood Generation

Christian Drescher and Toby Walsh

NICTA* and the University of New South Wales

Abstract

Although Answer Set Programming (ASP) systems are highly optimised, their performance is sensitive to the size of the input and the inference it encodes. We address this deficiency by introducing a new extension to ASP solving. The idea is to integrate external propagators to represent parts of the encoding implicitly, rather than generating it a-priori. To match the state-of-the-art in conflict-driven solving, however, external propagators can make their inference explicit on demand. We demonstrate applicability in a novel Constraint Answer Set Programming system that can seamlessly integrate constraint propagation without sacrificing the advantages of conflict-driven techniques. Experiments provide evidence for computational impact.

1998 ACM Subject Classification I.2.3 Deduction and Theorem Proving

Keywords and phrases Conflict-Driven Nogood Learning, Constraint Answer Set Programming, Constraint Propagation, Lazy Nogood Generation

Digital Object Identifier 10.4230/LIPIcs.ICLP.2012.188

1 Introduction

Developing a powerful paradigm for declarative problem solving is one of the key challenges in the area of knowledge representation and reasoning. A promising candidate is Answer Set Programming (ASP; [24, 16, 33, 43, 37, 2]) which builds on Logic Programming and Nonmonotonic Reasoning. Its success depends on two factors: efficiency of the solving capacities, and modelling convenience. Efficient ASP solvers [26, 22, 36, 32] match the state-of-the-art in conflict-driven solving [41], including conflict-driven learning, lookback-based heuristics, and backjumping. However, their performance is sensitive to the size of problem encodings which can quickly become infeasible, for instance, through the worst-case exponential number of loops in a logic program [34], or constructs that are naturally non-propositional, like constraints over finite domains. A variety of extensions to ASP have been proposed that deal with some of these issues via integration of other declarative problem solving paradigms. Recently, for example, we have witnessed the development of Constraint Answer Set Programming (CASP) that integrates Constraint Programming (CP) with ASP, supporting constraints over finite domains, and most importantly, global constraints. While this approach certainly increases modelling convenience and can drastically decrease the size of an encoding, it does not fully carry over to conflict-driven solving technology [12].

We address this problem and present a new computational extension to ASP solving, called Lazy Nogood Generation. Motivated by the success of Lazy Clause Generation [46] in Constraint Satisfaction Problem (CSP) solving, the key idea is to generate (parts of) the problem encoding on demand, only when new information can be propagated. We make several contributions to the study of Lazy Nogood Generation in ASP. First, we lay the foundations of external propagation based on a uniform characterisation of answer

* NICTA is funded by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

© Christian Drescher and Toby Walsh;
licensed under Creative Commons License ND

sets in terms of nogoods. This provides the underpinnings to represent conditions on the answer sets of a logic program without encoding the entire problem a-priori. However, external propagators can make parts of the encoding explicit, in particular, when they can trigger inference. As we shall see, our techniques generalise existing ones, e.g., loop formula propagation [22], and weight constraint rule propagation [21]. Second, we specify a decision procedure for ASP solving with Lazy Nogood Generation. It is centred around conflict-driven solving and integrates external propagation. Third, we demonstrate applicability. We show how to seamlessly integrate constraint propagation with our framework, resulting in a novel approach to CASP solving. Finally, we empirically evaluate a prototypical implementation and compare to the state-of-the-art in ASP and CASP solving.

2 Background

Many tasks from the declarative problem solving domain can be defined as CSP, that is a tuple (V, D, C) where V is a finite set of *constraint variables*, each $v \in V$ has an associated finite *domain* $dom(v) \in D$, and C is a set of constraints. A *constraint* c is a k-ary relation, denoted $R(c)$, on the domains of the variables in $S(c) \in V^k$. A *(constraint variable) assignment* is a function A that assigns to each variable $v \in V$ a value from $dom(v)$. For a constraint c with $S(c) = (v_1, \ldots, v_k)$ define $A(S(c)) = (A(v_1), \ldots, A(v_k))$. The constraint c *satisfied* if $A(S(c)) \in R(c)$. Otherwise, we say that c is *violated*. Let $C^A = \{c \in C \mid A(S(c)) \in R(c)\}$. An assignment A is a *solution* iff $C = C^A$. CP systems are oriented towards solving CSP and typically interleave backtracking search to explore assignments with *constraint propagation* to prune the set of values a variable can take. The effect of constraint propagation is studied in terms of *local consistency*. E.g., a binary constraint c is called *arc consistent* iff a variable in $S(c)$ is assigned any value, there exists a value in the domain for the other variable in $S(c) \setminus \{v\}$ such that c is not violated. An n-ary constraint c is called *domain consistent* iff $v \in S(c)$ is assigned any value, there exist values in the domains of all other variables in $S(c) \setminus \{v\}$ such that c is not violated. Observe that, in general, a constraint propagator that enforces domain consistency prunes more values than one that enforces arc consistency on a binary decomposition of the original constraint. CSPs can be encoded with ASP [43], which is founded on Logic Programming.

A *(normal) logic program* P over an alphabet $\mathcal{A}$ is a finite set of *rules* r of the form $a_0 \leftarrow a_1, \ldots, a_m, {\sim}a_{m+1}, \ldots, {\sim}a_n$ where $a_i \in \mathcal{A}$ are *atoms* for $0 \leq i \leq n$. A *default literal* is an atom a or its *default negation* ${\sim}a$. The atom $H(r) = a_0$ is called the *head* of r and the set of default literals $B(r) = \{a_1, \ldots, a_m, {\sim}a_{m+1}, \ldots, {\sim}a_n\}$ is called the *body* of r. For a set of default literals S, define $S^+ = \{a \mid a \in S\}$ and $S^- = \{a \mid {\sim}a \in S\}$. For restricting S to atoms $\mathcal{E}$, define $S|_{\mathcal{E}} = \{a \mid a \in S^+ \cap \mathcal{E}\} \cup \{{\sim}a \mid a \in S^- \cap \mathcal{E}\}$. For $X \subseteq \mathcal{A}$ define *external support* for X as $ES_P(X) = \{B(r) \mid r \in P,\ H(r) \in X,\ B(r)^+ \cap X = \emptyset\}$. The set of atoms occurring in P is denoted by $At(P)$, and the set of bodies in P is $B(P) = \{B(r) \mid r \in P\}$. For regrouping rules sharing the heads in $X \subseteq \mathcal{A}$, define $P_X = \{r \in P \mid H(r) \in X\}$, and for bodies sharing the same head a, define $B(a) = \{B(r) \mid r \in P,\ H(r) = a\}$. A *logic program with externals over* $\mathcal{E}$ is a logic program P over an alphabet distinguishing regular atoms $\mathcal{A}$ and external atoms $\mathcal{E}$, such that $H(r) \in \mathcal{A}$ for each $r \in P$. Let $Y \subseteq \mathcal{E}$. For a logic program P over externals from $\mathcal{E}$ define the *pre-reduct* $P(Y) = \{H(r) \leftarrow B(r)|_{\mathcal{A} \setminus \mathcal{E}} \mid r \in P,\ B(r)^+|_{\mathcal{E}} \subseteq Y,\ B(r)^-|_{\mathcal{E}} \cap Y = \emptyset\}$. A *splitting set* for a logic program P [35] is a set $\mathcal{E} \subseteq \mathcal{A}$ if $H(r) \in \mathcal{E}$ then $B(r)^+ \cup B(r)^- \subseteq \mathcal{E}$ for each $r \in P$. Observe that, if $\mathcal{E}$ is a splitting set of P, it *splits* P into a logic program $P_{\mathcal{E}}$ over $\mathcal{E}$ and a logic program $P_{\mathcal{A} \setminus \mathcal{E}}$ with externals over $\mathcal{E}$. The semantics of a logic program P is given by its answer sets. A set $X \subseteq \mathcal{A}$ is an *answer set* of P, if X is a minimal model

of the *reduct* $P^X = \{H(r) \leftarrow B(r)^+ \mid r \in P,\ B(r)^- \cap X = \emptyset\}$ [24]. Let $\mathcal{E}$ be a splitting set of P. The set $Z \subseteq \mathcal{A}$ is an answer set of P iff $Z = X \cup Y$ such that X is an answer set of $P_{\mathcal{E}}$ and Y is an answer set of $P_{\mathcal{A}\setminus\mathcal{E}}(Y)$ (Splitting Set Theorem, [35]). Although our semantics is propositional, modern ASP systems support *non-ground* logic programs and construct atoms in $\mathcal{A}$ from a first-order signature via a *grounding* process, systematically substituting all occurrences of first-order variables by terms, resulting in a *(ground) instantiation*.

Following [22], the answer sets of a logic program P can be characterised as Boolean assignments over $At(P) \cup B(P)$ that do not conflict with the conditions induced by the *completion* [9] and all *loop formulas* of P [30], expressed in terms of nogoods [11]. Formally, a *(Boolean) assignment* $\mathbf{A}$ is a sequence $(\sigma_1, \dots, \sigma_n)$ of *(signed) literals* σ_i of the form $\mathbf{T}a$ or $\mathbf{F}a$ where a is in the scope of $\mathbf{A}$, e.g., $S(\mathbf{A}) = At(P) \cup B(P)$. The complement of a literal σ is denoted $\overline{\sigma}$. True and false variables in $\mathbf{A}$ are accessed via $\mathbf{A}^\mathbf{T}$ and $\mathbf{A}^\mathbf{F}$, respectively. A *nogood* represents a set $\delta = \{\sigma_1, \dots, \sigma_n\}$ of signed literals, expressing a condition *conflicting* with any assignment $\mathbf{A}$ if $\delta \subseteq \mathbf{A}$. If $\delta \setminus \mathbf{A} = \{\sigma\}$ and $\overline{\sigma} \notin \mathbf{A}$, we say that δ is *unit* and *asserts* the *unit-resulting* literal $\overline{\sigma}$. A *total* assignment, that is $\mathbf{A}^\mathbf{T} \cup \mathbf{A}^\mathbf{F} = S(\mathbf{A})$ and $\mathbf{A}^\mathbf{T} \cap \mathbf{A}^\mathbf{F} = \emptyset$, is a *solution* for a set of nogoods Γ if $\delta \not\subseteq \mathbf{A}$ for each $\delta \in \Gamma$.

3 Nogoods of Logic Programs with Externals

We generalise [22] and describe nogoods capturing completion and loop formulas for a logic program P with externals over $\mathcal{E}$. For $\beta = \{a_1, \dots, a_m, \sim a_{m+1}, \dots, \sim a_n\} \in B(P)$, define

$$\Delta_\beta = \left\{ \begin{array}{l} \{\mathbf{T}a_1, \dots, \mathbf{T}a_m, \mathbf{F}a_{m+1}, \dots \mathbf{F}a_n, \mathbf{F}\beta\}, \\ \{\mathbf{F}a_1, \mathbf{T}\beta\}, \dots, \{\mathbf{F}a_m, \mathbf{T}\beta\}, \{\mathbf{T}a_{m+1}, \mathbf{T}\beta\}, \dots, \{\mathbf{T}a_n, \mathbf{T}\beta\} \end{array} \right\}.$$

Intuitively, the nogoods in Δ_β enforce the truth of body β iff all its elements are satisfied. For an atom $a \in At(P)$ with $B(a) = \{\beta_1, \dots, \beta_k\}$, define

$$\Delta_a = \{\ \{\mathbf{T}\beta_1, \mathbf{F}a\}, \dots, \{\mathbf{T}\beta_k, \mathbf{F}a\}, \{\mathbf{F}\beta_1, \dots, \mathbf{F}\beta_k, \mathbf{T}a\}\ \}.$$

Let $\Delta_P^{\mathcal{E}} = \bigcup_{\beta \in B(P)} \Delta_\beta \cup \bigcup_{a \in At(P)\setminus\mathcal{E}} \Delta_a$. The solutions for $\Delta_P^\emptyset$ correspond to the models of the completion of P [22]. To capture the effect of loop formulas induced by a set $L \subseteq At(P) \setminus \mathcal{E}$, for $a \in L$ define $\lambda(a, L) = \{\{\mathbf{T}a\} \cup \{\mathbf{F}\beta \mid \beta \in ES_P(L)\}\}$. The set of loop nogoods is $\Lambda_P^{\mathcal{E}} = \bigcup_{L \subseteq At(P)\setminus\mathcal{E}, L \neq \emptyset} \{\lambda(a, L) \mid a \in L\}$. Let P be a logic program and $X \subseteq \mathcal{A}$. Then, X is an answer set of P iff there is a (unique) solution for $\Delta_P^\emptyset \cup \Lambda_P^\emptyset$ such that $\mathbf{A}^\mathbf{T} \cap At(P) = X$ [22]. We combine this result with the Splitting Set Theorem [35].

▶ **Proposition 1.** *Let P be a logic program, $\mathcal{E}$ a splitting set for P, and $X \subseteq \mathcal{A}$. Then, X is an answer set of P iff there is a (unique) solution $\mathbf{A}$ for $\Delta_{P_\mathcal{E}}^\emptyset \cup \Lambda_{P_\mathcal{E}}^\emptyset \cup \Delta_{P_{\mathcal{A}\setminus\mathcal{E}}}^{\mathcal{E}} \cup \Lambda_{P_{\mathcal{A}\setminus\mathcal{E}}}^{\mathcal{E}}$ such that $\mathbf{A}^\mathbf{T} \cap (At(P_\mathcal{E}) \cup At(P_{\mathcal{A}\setminus\mathcal{E}})) = X$.*

An efficient algorithm for computing solutions to $\Delta_P^\emptyset \cup \Lambda_P^\emptyset$ is Conflict-Driven Nogood Learning (CDNL, [22]). It combines search and propagation by recursively assigning the value of a proposition and performing unit-propagation to determine its consequences [41].

4 Lazy Nogood Generation

Instead of generating all nogoods $\Delta_P^\emptyset \cup \Lambda_P^\emptyset$ a-priori, referred to as *eager* encoding, we introduce external propagators to generate nogoods on demand, i.e., only when they are able to propagate new information. We call this technique *Lazy Nogood Generation*, generalising an approach to encoding constraints over finite domains into sets of clauses by executing constraint propagation during SAT search and recording the propagation in terms of clauses (Lazy Clause Generation; [46]). Formally, an *external propagator* for a set of

nogoods Γ is a function π that maps a Boolean assignment to a subset of Γ such that for each total assignment $\mathbf{A}$ if $\delta \subseteq \mathbf{A}$ for some $\delta \in \Gamma$ then $\delta' \subseteq \mathbf{A}$ for some $\delta' \in \pi(\mathbf{A})$. In other words, an external propagator generates a conflicting nogood from Γ iff some nogood in Γ is conflicting with the total assignment. We call an external propagator *conflict-optimal*, if this condition holds for each (partial) assignment. Notice that, even for a conflict-optimal external propagator, unit-propagation on Γ can infer more unit-resulting literals than unit-propagation on lazily generated nogoods. To close this gap, we define inference-optimal external propagators. An external propagator π for a set of nogoods Γ is *inference-optimal* if π is conflict-optimal and for each non-conflicting assignment $\mathbf{A}$ if $\delta \setminus \mathbf{A} = \{\sigma\}$ such that $\overline{\sigma} \notin \mathbf{A}$ for some $\delta \in \Gamma$ then $\delta' \setminus \mathbf{A} = \{\sigma\}$ for some $\delta' \in \pi(\mathbf{A})$. The correspondence between external propagation and the set of nogoods it represents can be formalised as follows.

▶ **Proposition 2.** *Let Δ be a set of nogoods, and π be an external propagator for $\Gamma \subseteq \Delta$. Then, the assignment $\mathbf{A}$ is a solution of Δ iff $\mathbf{A}$ is a solution of $(\Delta \setminus \Gamma) \cup \pi(\mathbf{A})$.*

One of the advantages of Lazy Nogood Generation over eager encodings is space efficiency. For instance, the worst-case exponential number of loops in a logic program P makes an eager encoding of the conditions induced by $\Lambda_P^{\emptyset}$ infeasible [34]. Non-optimal external propagation, however, can check whether an *unfounded set* [50] has to be falsified in linear time [7], and determines nogoods in $\Lambda_P^{\emptyset}$ on demand via directed unfounded set inference [22]. To reflect Lazy Nogood Generation also on the language level of ASP, we make use of *splitting* [35] for outsourcing conditions over $\mathcal{E} \subseteq \mathcal{A}$ into $P_{\mathcal{E}}$. Instead of making $P_{\mathcal{E}}$ explicit, however, a set of external propagators Π can be provided that precisely represent the conditions induced by $P_{\mathcal{E}}$. We will write $At(\Pi)$ to access $\mathcal{E}$. The previous propositions yield the following result.

▶ **Theorem 3.** *Let P be a logic program, $\mathcal{E}$ a splitting set for P, Π a set of external propagators for $\Delta_{P_{\mathcal{E}}}^{\emptyset} \cup \Lambda_{P_{\mathcal{E}}}^{\emptyset}$, and $X \subseteq \mathcal{A}$. Then, X is an answer set of P iff there is a (unique) solution $\mathbf{A}$ for $\Delta_{P_{\mathcal{A}\setminus\mathcal{E}}}^{\mathcal{E}} \cup \Lambda_{P_{\mathcal{A}\setminus\mathcal{E}}}^{\mathcal{E}} \cup \bigcup_{\pi\in\Pi} \pi(\mathbf{A})$ s.t. $\mathbf{A}^{\mathbf{T}} \cap (At(P_{\mathcal{E}}) \cup At(\Pi)) = X$.*

External propagation provides a form of modularity that allows programmers to select encodings which propagate better, but were previously avoided for space-related reasons. E.g., in [12] we describe eager encodings that simulate constraint propagators for the ALL-DIFFERENT constraint which achieve arc, bound, or range consistency. A constraint propagator that can achieve domain consistency exists [48] but it cannot be simulated efficiently [6]. Because of the fact that external propagators generate nogoods only on demand, however, we can implicitly represent encodings via Lazy Nogood Generation that are otherwise infeasible.

5 Conflict-Driven Nogood Learning with Lazy Nogood Generation

We develop a decision procedure for answer set solving with Lazy Nogood Generation based on CDNL [22]. It is centred around *conflict analysis* according to the *First-UIP* scheme [41]. That is, a conflicting nogood is iteratively resolved against other nogoods until a conflicting nogood that contains a *unique implication point* is obtained. This guides backjumping. Recording the resolved nogood enables conflict-driven learning, which can further prune the search space. For controlling the set of recorded nogoods, deletion strategies can be applied (cf. [42]). In contrast to CDNL we will integrate external propagators that perform Lazy Nogood Generation in order to represent conditions on the answer sets of a logic program that are not encoded eagerly. Much like their eager counterpart, lazily generated nogoods can contribute to conflict analysis and lookback-based search heuristics. This can improve propagation. Different to eagerly encoded nogoods, however, the amount of lazily generated nogoods can be controlled via deletion.

Input : A logic program P with external propagators Π.
Output : An answer set of P if one exists.

```
1  A ← ∅                                              // Boolean assignment
2  ∇ ← ∅                                       // set of recorded nogoods
3  dl ← 0                                                // decision level
4  loop
5      (A, ∇) ← Propagation(P, Π, ∇, A)
6      if δ ⊆ A for some δ ∈ Δ_P^{At(Π)} ∪ ∇ then
7          if dl = 0 then return no answer set
8          (ε, k) ← ConflictAnalysis(δ, P, ∇, A)
9          ∇ ← ∇ ∪ {ε}
10         A ← A\{σ ∈ A | k < dl(σ)}
11         dl ← k
12     else if A^T ∪ A^F = At(P) ∪ B(P) ∪ At(Π) then
13         return A^T ∩ (At(P) ∪ At(Π))
14     else
15         σ_d ← Select(P, Π, ∇, A)
16         A ← A ∘ (σ_d)
17         dl ← dl + 1
```

Algorithm 1 CDNL-LNG.

5.1 Main Algorithm

Algorithm 1 specifies our main procedure, CDNL-LNG. It takes a logic program P with external propagators Π, and starts with an empty assignment $\mathbf{A}$ and an empty set ∇ that will store recorded nogoods, including lazily generated nogoods. The *decision level* dl is initialised with 0. Its purpose is counting *decision literals* in the assignment. We use $dl(\sigma)$ to access the decision level of literal σ. The following loop is very similar to CDNL. First, Propagation (Line 5) extends $\mathbf{A}$ and ∇, as described in the next section. If this encounters a conflict (Line 6), the ConflictAnalysis procedure generates a conflicting nogood ε by exploiting interdependencies between nogoods in $\Delta_P^{At(\Pi)} \cup \nabla$ through conflict resolution, and determines a decision level k to continue search at. Then, ε is added to the set of recorded nogoods ∇ in Line 9. This can prune the search space and lead to faster propagation. Lines 10–11 account for backjumping to level k. Thereafter ε is unit and triggers inference in the next round of propagation. If ConflictAnalysis, however, yields a conflict at level 0, no answer set exists (Line 7). Furthermore, we distinguish the cases of a complete assignment (Lines 12–13) and a partial one (Lines 14–17). In case of a complete assignment, the atoms in $\mathbf{A^T}$ correspond to an answer set of P. In the other case, $\mathbf{A}$ is partial and no nogood is conflicting. Then, a decision literal σ_d is selected by some heuristic, added to $\mathbf{A}$, and the decision level is incremented. While ConflictAnalysis and Select are similar to the ones in CDNL, we extend Propagation to accommodate Lazy Nogood Generation.

5.2 Propagation

A specification of our Propagation procedure is shown in Algorithm 2. It works on a logic program P with external propagators Π, a set of recorded nogoods ∇, and an assignment $\mathbf{A}$. Propagation interleaves unit-propagation on nogoods $\Delta_P^{At(\Pi)}$ and recorded nogoods ∇ including lazily generated nogoods from external propagators. We start with

Input : A logic program P with external propagators Π, recorded nogoods ∇, Boolean assignment $\mathbf{A}$.
Output : An extended assignment and set of recorded nogoods.

```
1  loop
2    repeat                                                         // unit-propagation
3      if δ ⊆ A for some δ ∈ Δ_P^{At(Π)} ∪ ∇ then return (A, ∇)
4      Σ ← {δ ∈ Δ_P^{At(Π)} ∪ ∇ | δ\A = {σ}, σ̄ ∉ A}
5      if Σ ≠ ∅ then let σ ∈ δ\A for some δ ∈ Σ in
6        A ← A ∘ (σ̄)
7    until Σ = ∅
8    foreach π ∈ Π do
9      Σ ← π(A)                                                      // external propagation
10     if Σ ≠ ∅ then break
11   if Σ = ∅ then
12     Σ ← LoopFormulaPropagation(P, A)                      // loop formula propagation
13   if Σ = ∅ then return (A, ∇)
14   ∇ ← ∇ ∪ Σ
```

Algorithm 2 Propagation.

unit-propagation (Lines 2–7), resulting either in a conflict, i.e., some nogood is conflicting (Line 3), or in a fixpoint possibly extending $\mathbf{A}$ with unit-resulting literals. If there is no conflict, Propagation performs external propagation following some priority (Lines 8–10). Based on $\mathbf{A}$, each propagator may encode inference in a set of lazily generated nogoods Σ which is added to the set of recorded nogoods ∇ at the end of the loop in Line 14. The LoopFormulaPropagation procedure (Line 12; [22]) works similarly to ensure that no loop formula is violated, i.e., no loop nogood in $\Lambda_P^{At(\Pi)}$ is conflicting. This only has an effect if the logic program is non-tight [18]. Note that external propagation is interleaved by unit-propagation in order to assign unit-resulting literals immediately and detect conflicts early. Our algorithm also favours external propagation over loop formula propagation, motivated by the fact that external propagators can affect the assignment to atoms in $At(\Pi)$, possibly falsifying external support for a loop in P.

6 Constraint Answer Set Solving via Lazy Nogood Generation

One difficult task for ASP solving with Lazy Nogood Generation remains, i.e., to design efficient external propagators. A research area that is largely concerned with efficient propagation is CP. We here follow the idea from [46] and apply CP techniques to generate lazy nogoods representing constraints over finite domains. To reflect this on the language level, we make use of CASP, a paradigm that naturally merges CP and ASP.

CASP abstracts from non-propositional constraints by incorporating *constraint atoms* into logic programs. We access the constraint atom associated to a constraint c via the function $At(c)$. A *constraint logic program* is a tuple $\mathbb{P} = (V, D, C, P)$, where V, D, C are the same as in the definition of a CSP, and P is a logic program with externals over *constraint atoms* $\mathcal{C} = \{At(c) \mid c \in C\}$. A fundamental difference to traditional CP is that, in CASP, each constraint c is reified via $At(c)$. Its truth value is determined by the conditions induced by P and an assignment A to the variables in $S(c)$. The set of constraint atoms $\mathcal{C}^A = \{At(c) \mid$

Input : A Boolean assignment $\mathbf{A}$.
Output : A set of lazily generated nogoods.

1 $\nabla \leftarrow \emptyset$ // *set of lazily generated nogoods*
2 **if** $\mathbf{T}val(v,i) \in \mathbf{A}$ for some $i \in dom(v)$ **then**
3 $\quad \nabla \leftarrow \{\{\mathbf{T}val(v,i),\ \mathbf{T}val(v,j)\} \mid j \in dom(v)\backslash\{i\},\ \mathbf{F}val(v,j) \notin \mathbf{A}\}$
4 **if** $\mathbf{T}val(v,i) \notin \mathbf{A}$ for some $i \in dom(v) \wedge \forall j \in dom(v)\backslash\{i\}\ \mathbf{F}val(v,j) \in \mathbf{A}$ **then**
5 $\quad \nabla \leftarrow \{\{\mathbf{F}val(v,i) \mid i \in dom(v)\}\}$
6 **return** ∇

Algorithm 3 An external propagator for the value encoding Γ_v.

$c \in C^A\}$ correspond to the constraints satisfied by A. Let $\mathbb{P}$ be a constraint logic program and A an assignment. The pair (X, A) is a *constraint answer set* of $\mathbb{P}$ iff X is an answer set of $P(\mathcal{C}^A)$ (cf. [23]). Given that assignments A and their effect on each constraint can be represented in a logic program [43], the task of computing constraint answer sets can be reduced to the one of computing answer sets of P with external propagators for generating assignments A and capturing the inference of constraint propagation.

To begin with, CASP solving via Lazy Nogood Generation requires a propositional representation of assignments to constraint variables. In the *value encoding*, an atom $val(v,i)$, representing $v = i$, is introduced for each variable $v \in V$ and value $i \in dom(v)$. Intuitively, the atom $val(v,i)$ is true if v takes the value i, and false if v takes a value different from i (cf. [51]). To insure that an assignment $\mathbf{A}$ represents a consistent set of possible values for v, we encode the conditions that v must not take two values, i.e., $\{\mathbf{T}val(v,i), \mathbf{T}val(v,j)\} \not\subseteq \mathbf{A}$ for all $i,j \in dom(v),\ i \neq j$, and that v must take at least one value, i.e., $\mathbf{F}val(v,i) \notin \mathbf{A}$ for some $i \in dom(v)$, in the set of nogoods $\Gamma_v = \{\{\mathbf{T}val(v,i), \mathbf{T}val(v,j)\} \mid i,j \in dom(v),\ i \neq j\} \cup \{\{\mathbf{F}val(v,i) \mid i \in dom(v)\}\}$ [12]. We employ external propagators to represent the nogoods in Γ_v. Algorithm 3 provides a specification of an inference-optimal external propagator for this task. It takes a Boolean assignment $\mathbf{A}$ and returns a set of lazily generated nogoods, initialised in Line 1, that are unit or conflicting. Lines 2–3 insure that if v is assigned a value i then all other values are removed from its domain, while Lines 4–5 deal with the condition that there is at least one value that can be assigned to v. This procedure can be made very efficient, e.g., by using *watched literals* [42]. Another representation for constraint variables is the *bound encoding*, where an atom is introduced for each variable $v \in V$ and value $i \in dom(v)$ to represent that v is bounded by i, i.e., $v \leq i$ (cf. [49]). Similar to the value encoding, we can define nogoods that insure a consistent Boolean assignment [12]. A combination of value and bound encoding is also possible.

We see atoms from the value and bound encoding as *primitive constraints*, as all constraints can be decomposed into nogoods over them, e.g., by describing changes in the variables' domains inferred by constraint propagation. This way, constraint propagators can be encoded eagerly or lazily. Transforming a constraint propagator into an external propagator is straightforward: Rather than applying domain changes directly, the constraint propagator has to be made encoding its inferences in form of nogoods over primitive constraints [46].

▶ **Example 4.** An external propagator for encoding the reified ALL-DIFFERENT constraint c is specified in Algorithm 4. Provided with a Boolean assignment $\mathbf{A}$, it starts with an empty set of lazily generated nogoods, followed by a distinction into two cases. First, if the constraint is to be satisfied, i.e., $\mathbf{T}At(c) \in \mathbf{A}$, then for each variable in the scope of the constraint that has a value assigned, a nogood is generated that asserts the removal of this value from the domain of all other variables in the scope of the constraint (Lines 2–3). On the other hand,

Input : A Boolean assignment $\mathbf{A}$.
Output : A set of lazily generated nogoods.

```
1  ∇ ← ∅                                                    // set of lazily generated nogoods
2  if TAt(c) ∈ A then foreach v ∈ S(c) s.t. Tval(v,i) ∈ A for some i ∈ dom(v) do
3      ∇ ← ∇ ∪ {{TAt(c), Tval(v,i), Tval(w,i)} | w ∈ S(c)\{v},
                                                  i ∈ dom(w), Fval(w,i) ∉ A}
4  else
5      foreach v ∈ S(c) s.t. Tval(v,i) ∈ A for some i ∈ dom(v) do
6          if w ∈ S(c)\{v} s.t. Tval(w,i) ∈ A then
7              if FAt(c) ∉ A then
8                  ∇ ← {{TAt(c), Tval(v,i), Tval(w,i)}}
9              return ∇
10     if ∀v ∈ S(c) ∃i ∈ dom(v) s.t. Tval(v,i) ∈ A then
11         ∇ ← {{FAt(c)} ∪ {Tval(v,i) | v ∈ S(c), i ∈ dom(v), Tval(v,i) ∈ A}}
12 return ∇
```

Algorithm 4 An external propagator for encoding the reified ALL-DIFFERENT constraint c.

if the constraint is not set to be satisfied, the algorithm checks whether two variables in the scope of the constraint have the same value assigned (Lines 5–9). If so, the ALL-DIFFERENT constraint is violated and a nogood asserting that the constraint atom is set to false will be returned (unless $\mathbf{F}At(c) \in \mathbf{A}$, in which case the constraint atom is already false). If, however, no such two variables can be found and all variables in the scope of the constraint have a value assigned, then the ALL-DIFFERENT condition is satisfied and a nogood is generated that asserts the truth of the constraint atom (Lines 10–11). Observe that this propagator enforces arc consistency on the binary decomposition of the reified ALL-DIFFERENT constraint if $At(c)$ is true, but propagates weakly if $At(c)$ is false. However, propagators that achieve higher levels of local consistency are also possible [48].

While constraint propagators encode their inference into unit or conflicting nogoods, unit-propagation processes this information within the next iteration. Unit-propagation, constraint propagation, and loop formula propagation are repeated until a fixpoint is reached or a conflict is encountered. By generating a conflicting nogood, e.g., a constraint propagator can yield that the underlying constraint is violated.

7 Experiments

We have implemented our approach with Lazy Nogood Generation for constraint variables, the ALL-DIFFERENT and integer LINEAR constraints within a new version of our prototypical CASP system *inca* [54] which is based on the latest development version of *clingo* (3.0.92; [53]). The default setting uses an ALL-DIFFERENT propagator that enforces arc consistency, while $inca^{DC}$ enforces domain consistency, representing an infeasible encoding. To compare with the state-of-the-art, we include *clingcon* (2.0.0-beta; [53]) in our analysis. It also extends *clingo*, but integrates the CP solver *gecode* (3.7.1;[52]). Similar to our approach, *clingcon* is based on CDNL and abstracts from the constraints via constraint atoms, but it employs *gecode* to check the existence of a constraint variable assignment that does not violate any constraint (according to the assignment to constraint atoms). In turn, the CP solver can yield a conflict or propagate constraint atoms by generating nogoods over constraint atoms that

Table 1 Average time in seconds over completed runs on Quasigroup, Graceful Graph, Packing, and Numbrix benchmarks. Number of completed runs are given in parenthesis.

benchmark class	*clingo*	*clingcon*	*clingcon*^{DC}	*inca*	*inca*^{DC}
Quasigroup Completion (200)	106.6 (93)	34.4 (9)	4.6 (200)	86.2 (171)	24.7 (200)
Quasigroup Existence (21)	25.7 (18)	61.4 (10)	88.2 (11)	60.3 (20)	26.6 (20)
Graceful Graphs (10)	3.0 (9)	15.7 (4)	31.3 (7)	5.2 (6)	12.6 (10)
Packing (50)	104.1 (1)	33.1 (50)	33.1 (50)	24.6 (50)	24.6 (50)
Numbrix (12)	10.4 (12)	17.4 (12)	51.3 (12)	1.3 (12)	5.2 (12)
weighted, penalised time	228.3	267.0	124.6	103.1	24.2

occur in the constraint logic program. This constitutes a very limited form of Lazy Nogood Generation. We have set *clingcon* to generate nogoods by looking at dependency between constraints according to the *irreducibly inconsistent set construction* method in "forward" mode, when we noticed that this option significantly improves the performance of *clingcon*. Furthermore, the setting *clingcon*^{DC} uses domain consistency propagation. Our experiments also consider eager encodings for a comparison with the state-of-the-art in ASP solving, given through *clingo*. We conducted experiments on Quasigroup Completion ($n = 40$), Quasigroup Existence (QG1-4: $n = 7 \ldots 9$; QG5: $n = 12 \ldots 14$; QG6-7: $n = 10 \ldots 12$) and Graceful Graphs benchmarks that stem from [12], Packing benchmarks from [8] and Numbrix [45] puzzles. Experiments were run on a Linux PC, where each run was limited to 600 sec CPU time on a 2.00 GHz core and 2 GB RAM. A summary of our results is provided in Table 1.

Although more benchmark classes are needed for a meaningful comparison, we can draw a few interesting conclusions. First, execution time can improve when CP constructs are treated by external propagation rather than encoding them eagerly. The latter can lead to huge encodings, in particular, when large domains are involved. In fact, the encoding of the Packing problem that was given in the system track of the competition quickly reaches the memory limit of 2 GB in 49 over 50 instances, while the CASP systems *clingcon* and *inca* solve every instance within a reasonable amount of space and time. Second, the advantage of generating nogoods to describe the inferences of constraint propagators is that CDNL can exploit constraint interdependencies for directing search, and most importantly conflict analysis. The fact that *clingcon* does not encode CP constructs into nogoods, by design, is likely to be the reason for its limited success in our experiments, where *clingcon* is particularly ineffective on Quasigroup problems. Third, experiments show that our approach, represented through *inca*, combines the best of both worlds: It can avoid huge encodings via abstraction to external propagation while retaining the ability to make the encoding explicit. It outperforms the state-of-the-art in CASP solving on individual benchmark classes, and is more robust over all benchmark instances. On most benchmarks, a dedicated treatment of infeasible ALL-DIFFERENT encodings via external propagation has further improved performance.

8 Related Work

Related work on the integration of ASP with other declarative problem solving paradigms is plentiful, and roughly falls into one of three categories: translation-based approaches, modular approaches, and integrated approaches. In translation-based approaches, all parts of an (extended) ASP model are eagerly encoded into a single language for which highly efficient off-the-shelf solvers are available. Niemelä [43] provides a simple mapping of constraints into ASP given by allowed or forbidden combinations of values. We have demonstrated efficiency in [12], describing what type of local consistency the unit-propagation of an ASP

solver achieves on value, bounds, and range encodings. Specialised encodings for GRAMMAR and related constraints are presented in [14]. There is also a substantial body of work on encoding constraints into SAT [51, 25, 4, 15, 49, 5] which can be translated into ASP [43]. Similarly, (extended) ASP models can be translated, e.g., into SAT [27], SAT with inductive definitions [38], and difference logic [28]. In a modular approach, theory-specific solvers interact in order to compute solutions. Baselice et al. [3] and Mellarkod and Gelfond [39] combine systems for solving ASP and CP that do not ground constraint variables. Instead, constraint variables are handled in a CP solver. Dal Palú et al. [10] employ a CP system for intermediate grounding and the computation of answer sets. The approach taken by Balduccini [1] consists of writing logic programs whose answer sets encode a desired CSP, which is, in turn, solved by a CP system. Järvisalo et al. [29] obtain the overall semantics from the ones of individual modules, including CP modules. While above modular approaches see ASP and CP solvers as blackboxes, Mellarkod et al. [40] integrate a CP solver into the decision engine of a backtracking-based ASP solver. Gebser et al. [23] integrate constraint atoms with conflict-driven techniques by extending the conflict analysis of an ASP solver. An implementation of their approach is given through the CASP system *clingcon*. The abstraction from the inference performed by constraint propagation, however, limits the exploitation of constraint interdependencies. ASP solving via Lazy Nogood Generation was first outlined in [13], and falls into the category of integrated approaches. The related work closest to this paper is Lazy Clause Generation [46], a SAT-based approach to CSP solving where lazy clause generators encode the inference of *propagation rules* into clauses. However, our approach is fundamentally more general then Lazy Clause Generation, where the truth value of each constraint atom is known a-priori and every nogood is represented by a clause. Nogoods can also be represented by other ASP constructs, such as cardinality rules, weight constraint rules [44], and aggregation [47, 19]. Gebser et al. [20] show that constraint variables can be conveniently expressed by means of cardinality rules. Elkabani et al. [17] provide a generic framework which provides an elegant treatment of such extensions to ASP, employing constraint propagators for their handling, though, without support for conflict-driven techniques. A thorough approach to integrating propagators for weight constraint rules within a conflict-driven framework is presented in [21].

Alternative computation models that aim at limiting the need for preliminary grounding but do not integrate ASP with other declarative paradigms have also been proposed (e.g., [31]).

9 Conclusion

We presented a comprehensive extension for ASP solving to address the scalability and efficiency of ASP, called Lazy Nogood Generation. Founded on a nogood-based characterisation of external propagation, our techniques allow for representing encodings that are otherwise infeasible. However, external propagators can make parts of the encoding explicit whenever it triggers inference. We presented key algorithms that are centred around conflict-driven learning, and seamlessly applied our techniques to CASP solving by employing constraint propagation. Experiments show that our prototypical implementation is competitive with the state-of-the-art. We expect further significant computational impact given the empirical evidence provided by Lazy Clause Generation [46]. Moreover, Lazy Nogood Generation generalises Lazy Clause Generation, as every nogood can be syntactically represented by a clause, but other ASP constructs are also possible. Future work considers the exploitation of ASP constructs like aggregation and loops. Many questions on modelling and solving CASP also remain open, concerning encoding optimisations and further language extensions.

References

1. M. Balduccini. Representing constraint satisfaction problems in answer set programming. In *25th International Conference on Logic Programming (ICLP'09) Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP'09)*, 2009.
2. C. Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving.* Cambridge University Press, 2003.
3. S. Baselice, P. Bonatti, and M. Gelfond. Towards an integration of answer set and constraint solving. In *21st International Conference on Logic Programming (ICLP'05)*, pages 52–66. Springer, 2005.
4. C. Bessière, E. Hebrard, and T. Walsh. Local consistencies in SAT. In *6th International Conference on Theory and Applications of Satisfiability Testing (SAT'03)*, pages 299–314. Springer, 2003.
5. C. Bessière, G. Katsirelos, N. Narodytska, C.-G. Quimper, and T. Walsh. Decompositions of all different, global cardinality and related constraints. In *21st International Joint Conference on Artificial Intelligence (IJCAI'09)*, pages 419–424, 2009.
6. C. Bessière, G. Katsirelos, N. Narodytska, and T. Walsh. Circuit complexity and decompositions of global constraints. In *21st International Joint Conference on Artificial Intelligence (IJCAI'09)*, pages 412–418, 2009.
7. F. Calimeri, W. Faber, N. Leone, and G. Pfeifer. Pruning operators for answer set programming systems. In *9th International Workshop on Non-Monotonic Reasoning (NMR'02)*, pages 200–209, 2002.
8. F. Calimeri, G. Ianni, F. Ricca, M. Alviano, A. Bria, G. Catalano, S. Cozza, W. Faber, O. Febbraro, N. Leone, M. Manna, A. Martello, C. Panetta, S. Perri, K. Reale, M. C. Santoro, M. Sirianni, G. Terracina, and P. Veltri. The third answer set programming competition: Preliminary report of the system competition track. In *11th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'11)*, pages 388–403. Springer, 2011.
9. K. Clark. Negation as failure. In *Logic and Data Bases*, pages 293–322. Plenum Press, 1978.
10. A. Dal Palù, A. Dovier, E. Pontelli, and G. Rossi. Answer set programming with constraints using lazy grounding. In *25th International Conference on Logic Programming (ICLP'09)*, pages 115–129. Springer, 2009.
11. R. Dechter. *Constraint Processing.* Morgan Kaufmann, 2003.
12. C. Drescher and T. Walsh. A translational approach to constraint answer set solving. *Theory and Practice of Logic Programming, 26th International Conference on Logic Programming (ICLP'10) Special Issue*, 10(4-6):465–480, 2010.
13. C. Drescher and T. Walsh. Conflict-driven constraint answer set solving with lazy nogood generation. In *25th AAAI Conference on Artificial Intelligence (AAAI'11)*, pages 1772–1773. AAAI Press, 2011.
14. C. Drescher and T. Walsh. Modelling grammar constraints with answer set programming. In *Technical Communications of the 27th International Conference on Logic Programming (ICLP'11)*, volume 11 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 28–39. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2011.
15. N. Eén and N. Sörensson. Translating pseudo-Boolean constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation*, 2:1–26, 2006.
16. T. Eiter, G. Gottlob, and H. Mannila. Disjunctive datalog. *ACM Transactions on Database Systems*, 22(3):364–418, 1997.
17. I. Elkabani, E. Pontelli, and T. Son. Smodels with CLP and its applications: A simple and effective approach to aggregates in ASP. In *20th International Conference on Logic Programming (ICLP'04)*, pages 73–89. Springer, 2004.

18 E. Erdem and V. Lifschitz. Tight logic programs. *Theory and Practice of Logic Programming*, 3(4-5):499–518, 2003.

19 W. Faber, G. Pfeifer, and N. Leone. Semantics and complexity of recursive aggregates in answer set programming. *Artificial Intelligence*, 175(1):278–298, 2011.

20 M. Gebser, H. Hinrichs, T. Schaub, and S. Thiele. xpanda: A (simple) preprocessor for adding multi-valued propositions to ASP. In *23rd Workshop on (Constraint) Logic Programming (WLP'09)*, 2009.

21 M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub. On the implementation of weight constraint rules in conflict-driven ASP solvers. In *25th International Conference on Logic Programming (ICLP'09)*, pages 250–264. Springer, 2009.

22 M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. Conflict-driven answer set solving. In *20th International Joint Conference on Artificial Intelligence (IJCAI'07)*, pages 386–392. AAAI Press/MIT Press, 2007.

23 M. Gebser, M. Ostrowski, and T. Schaub. Constraint answer set solving. In *25th International Conference on Logic Programming (ICLP'09)*, pages 235–249. Springer, 2009.

24 M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *5th International Conference and Symposium on Logic Programming (ICLP/SLP'88)*, pages 1070–1080. MIT Press, 1988.

25 I. P. Gent. Arc consistency in SAT. In *15th Eureopean Conference on Artificial Intelligence (ECAI'02)*, pages 121–125. IOS Press, 2002.

26 E. Giunchiglia, Y. Lierler, and M. Maratea. Answer set programming based on propositional satisfiability. *Journal of Automated Reasoning*, 36(4):345–377, 2006.

27 T. Janhunen and I. Niemelä. Compact translations of non-disjunctive answer set programs to propositional clauses. In *11th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'11)*, pages 111–130. Springer, 2011.

28 T. Janhunen, I. Niemelä, and M. Sevalnev. Computing stable models via reductions to difference logic. In *10th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'09)*, pages 142–154. Springer, 2009.

29 M. Järvisalo, E. Oikarinen, T. Janhunen, and I. Niemelä. A module-based framework for multi-language constraint modeling. In *10th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'09)*, pages 155–169. Springer, 2009.

30 J. Lee. A model-theoretic counterpart of loop formulas. In *19th International Joint Conference on Artificial Intelligence (IJCAI'05)*, pages 503–508. Professional Book Center, 2005.

31 C. Lefèvre and P. Nicolas. The first version of a new ASP solver : ASPeRiX. In *10th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'09)*, pages 522–527. Springer, 2009.

32 Y. Lierler. Abstract answer set solvers with backjumping and learning. *Theory and Practice of Logic Programming*, 11(2-3):135–169, 2011.

33 V. Lifschitz. Answer set planning. In *16th International Conference on Logic Programming (ICLP'99)*, pages 23–37. MIT Press, 1999.

34 V. Lifschitz and A. Razborov. Why are there so many loop formulas? *ACM Transactions on Computational Logic*, 7(2):261–268, 2006.

35 V. Lifschitz and H. Turner. Splitting a logic program. In *11th International Conference on Logic Programming (ICLP'94)*, pages 23–37. MIT Press, 1994.

36 M. Maratea, F. Ricca, W. Faber, and N. Leone. Look-back techniques and heuristics in DLV: Implementation, evaluation, and comparison to QBF solvers. *Algorithms*, 63(1-3):70–89, 2008.

37 V. M. Marek and M. Truszczynski. Stable models and an alternative logic programming paradigm. *The Logic Programming Paradigm: a 25-year perspective*, pages 375–398. Springer, 1999.

38 M. Mariën, J. Wittocx, M. Denecker, and M. Bruynooghe. SAT(ID): Satisfiability of propositional logic extended with inductive definitions. In *11th International Conference on Theory and Applications of Satisfiability Testing (SAT'08)*, pages 211–224. Springer, 2008.

39 V. Mellarkod and M. Gelfond. Integrating answer set reasoning with constraint solving techniques. In *9th International Symposium Proceedings on Functional and Logic Programming (FLOPS'08)*, pages 15–31. Springer, 2008.

40 V. Mellarkod, M. Gelfond, and Y. Zhang. Integrating answer set programming and constraint logic programming. *Annals of Mathematics and Artificial Intelligence*, 53(1-4):251–287, 2008.

41 D. Mitchell. A SAT solver primer. *Bulletin of the European Association for Theoretical Computer Science*, 85:112–133, 2005.

42 M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *38th Design Automation Conference (DAC'01)*, pages 530–535. ACM, 2001.

43 I. Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25(3-4):241–273, 1999.

44 I. Niemelä, P. Simons, and T. Soininen. Stable model semantics of weight constraint rules. In *5th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'99)*, pages 317–331. Springer, 1999.

45 `http://www.parade.com/askmarilyn/numbrix/`.

46 O. Ohrimenko, P. J. Stuckey, and M. Codish. Propagation via lazy clause generation. *Constraints*, 14(3):357–391, 2009.

47 N. Pelov. *Semantics of logic programs with aggregates*. PhD thesis, Department of Computer Science, K.U. Leuven, Belgium, 2004.

48 J.-C. Régin. A filtering algorithm for constraints of difference in CSPs. In *12th AAAI Conference on Artificial Intelligence (AAAI'94)*, pages 362–367. AAAI Press, 1994.

49 N. Tamura, A. Taga, S. Kitagawa, and M. Banbara. Compiling finite linear CSP into SAT. *Constraints*, 14(2):254–272, 2009.

50 A. Van Gelder, K. A. Ross, and J. S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, 1991.

51 T. Walsh. SAT v CSP. In *6th International Conference on Principles and Practice of Constraint Programming (CP'00)*, pages 441–456. Springer, 2000.

52 `http://www.gecode.org`.

53 `http://potassco.sourceforge.net`.

54 `http://potassco.sourceforge.net/labs.html`.

Lazy Model Expansion by Incremental Grounding

Broes De Cat[1], Marc Denecker[2], and Peter Stuckey[3]

1 **Department of Computer Science, K.U.Leuven, Belgium**
`broes.decat@cs.kuleuven.be`
2 **Department of Computer Science, K.U.Leuven, Belgium**
`marc.denecker@cs.kuleuven.be`
3 **National ICT Australia, Victoria Laboratory,**
Department of Computing and Information Systems,
University of Melbourne, Australia
`peter.stuckey@nicta.com.au`

Abstract

Ground-and-solve methods used in state-of-the-art Answer Set Programming and model expansion systems proceed by rewriting the problem specification into a ground format and afterwards applying search. A disadvantage of such approaches is that the rewriting step blows up the original specification for large input domains and is unfeasible in case of infinite domains. In this paper we describe a *lazy* approach to model expansion in the context of first-order logic that can cope with large and infinite problem domains. The method interleaves grounding and search, incrementally extending the current partial grounding only when necessary. It often allows to solve the original problem without creating the full grounding and is hence more widely applicable than ground-and-solve. We report on an existing implementation within the IDP system and on experiments that show the promise of the method.

1998 ACM Subject Classification F.4.1 Mathematical Logic, I.2.4 Knowledge Representation Formalisms and Methods, I.2.8 Problem Solving, Control Methods, and Search

Keywords and phrases Knowledge representation and reasoning, model generation, grounding, IDP framework, first-order logic

Digital Object Identifier 10.4230/LIPIcs.ICLP.2012.201

1 Introduction

Model expansion [8] is the task of generating models of a logical theory for a given universe of domain elements. It is a widely accepted way to solve a range of problems, by encoding the problem in a declarative (logic) language such that structures which satisfy the specification are solutions to the problem. Model expansion is related to answer set generation in Answer Set Programming [10] and to finding variable assignments satisfying sets of constraints in Constraint Programming [1]. One approach used to solve such inference tasks is the *ground-and-solve* paradigm. The problem specification is formulated in a high-level (user-friendly) language, which is then rewritten into a lower-level representation on which a search algorithm can be applied. This process is called *grounding* (also known as *unrolling*). Examples are the high-level language FO($\cdot$) [5], which is grounded to its propositional fragment PC($\cdot$); ASP is grounded to propositional ASP and MiniZinc [9] is unrolled into Flatzinc.

An important bottleneck to applying ground-and-solve is the size of the grounding. Grounding an FO theory results in a blow-up of the size of the theory, related to the nesting

© Broes De Cat, Marc Denecker, and Peter Stuckey;
licensed under Creative Commons License ND
Technical Communications of the 28th International Conference on Logic Programming (ICLP'12).
Editors: A. Dovier and V. Santos Costa; pp. 201–211

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

depth of quantifiers and the size of the domains of the original theory. There are lots of practical problems in which the propositional theory is too large to generate.

In this paper, we present a novel approach to remedy this bottleneck, based on rewriting the theory *lazily* instead of up-front. The two main ideas are *placeholder introduction* and $\forall$-*instantiation*. Placeholders which represent non-ground formulas are introduced in the grounding. During search, they are "grounded further" depending on the interpretation. For sentences of (universally quantified) disjunctions, conditions are derived under which those sentences can certainly be satisfied by some extension of the current interpretation. As long as the interpretation satisfies those conditions, the sentence is not added to the grounding. Consider for example a disjunctive sentence: as long as one disjunct is true in the model, the value of the others is irrelevant.

It is also shown that the approach becomes even stronger when we are only interested in a subset of the full solution, as long as it can certainly be extended to a model. For example for planning problems, we are often only interested in the actions necessary to achieve the goal, independent of the full (possibly infinite) time frame and any additional relationships.

The approach is presented for theories in function-free first-order logic. Without loss of generality, any FO theory can be transformed into one not containing functions [18]. In section 3, it is shown how to represent partly ground theories and their properties. Section 4 shows which formulas to delay and the lazy model expansion algorithm is presented in section 4.4. Experimental results are provided in section 5, related work in 6.

2 Preliminaries

In this section, we first present syntax and semantics of FO, used throughout the paper, followed an introduction to the inference tasks *modelexpansion* and *grounding*.

2.1 FO

We assume familiarity with classical logic. A vocabulary Σ consists of a set of predicate and function symbols. Propositional symbols and constants are 0-ary predicate symbols, respectively function symbols. FO terms and formulae are defined as usual, and are built inductively from variables, constant and function symbols, logical connectives ($\neg$, $\wedge$, $\vee$) and quantifiers ($\forall$, $\exists$).

Each variable x is assumed to have an associated set of domain elements t over which the variable ranges, denoted as $x[\![t]\!]$. We sometimes refer to such a variable as a *typed* variable and formulae containing only typed variables as typed formulae. Given a formula φ with a free variable x, substitution of x with domain element d is denoted as $\varphi[x/d]$.

Throughout the paper, A and L are used to refer to an atom, respectively a literal. A *ground sentence* is a sentence without variables (hence also without quantifiers). A *ground theory* is a theory consisting of ground sentences.

In this paper we deal with three-valued interpretations I which allow us to adequately represent the partial structure within a search algorithm. We will sometimes write an interpretation I as the set of all domain literals which are true in it. For example given a set of domain atoms $\{P, Q, R\}$, then $I = \{P, \neg Q\}$, denotes the interpretation in which P is true, Q is false and R is unknown.

The interpretation of a sentence under an interpretation I, denoted $I(\varphi)$, is defined as usual except for quantified formulae, as we assume each variable is typed. For existential quantification, $I(\exists x[\![t]\!](\varphi))$ is true iff there is a $d \in t$ such that $I(\varphi[x/d]) = \mathbf{t}$; and false iff for all $d \in t$ we have $I(\varphi[x/d]) = \mathbf{f}$. (Typed) universal quantification is defined similarly.

The interpretation of formulae containing the shorthands $\Rightarrow, \Leftarrow$ and $\equiv$ is taken to be the interpretation of the formulae they represent.

A (three-valued) interpretation I is a *model* of an FO sentence if and only if the sentence is true under the interpretation. It is a model of an FO theory $\mathcal{T}$ if and only if it is a model of each of the sentences in $\mathcal{T}$.

An interpretation I is *more precise* than an interpretation I' if and only if I is identical to I' except on symbols which are unknown in I' ($I' \subseteq I$). Two interpretations I and J *agree on shared symbols* if there is no proposition P where $\{P, \neg P\} \subseteq I \cup J$.

An occurrence of a subformula φ in $\mathcal{T}$ is called *monotone* if it is not in the scope of a negation. It is *anti-monotone* if it is in the scope of a negation. If it occurs as a subformula of an equivalence, it is called *non-monotone*. This reflects the well-known property that increasing the truth value of an atom with only monotone occurrences, increases the truth value of formulas. If the atom has only anti-monotone occurrences, then increasing its truth value decreases the value of formulae.

With a slight abuse of notation, given a theory $\mathcal{T}$, $\mathcal{T}$ is used both to refer to the set and the conjunction of the sentences it contains.

2.2 Model expansion

Model generation is the inference task of, given a vocabulary Σ, a theory $\mathcal{T}$ over Σ and a (partial) interpretation $\mathcal{S}_{in}$ of Σ, finding models M which satisfy $\mathcal{T}$ and are more precise than $\mathcal{S}_{in}$. If the universe of domain elements is part of the input structure, the inference task is called *model expansion*, denoted as $\text{MX}\langle\mathcal{T}, \mathcal{S}_{in}\rangle$. Model expansion can be used to solve problems by modelling them as a logical theory and structure such that solutions to the problem are models of the theory extending the structure[8].

In this paper we consider an instance of model expansion where also an "output" vocabulary σ_{out} is given, a subset of Σ. The idea is then to generate interpretations I which are two-valued on σ_{out} and for which an extension exists which is a model of the theory $\mathcal{T}$ and is more precise than $\mathcal{S}_{in}$. Conceptually, this comes down to problems where we are only interested in some part of the solution, as long as we are guaranteed that a complete solution exists. This task generalizes both satisfiability checking (where σ_{out} is emtpy) and model expansion (where $\sigma_{out} = \Sigma$). It is denoted as $\text{MX}\langle\mathcal{T}, \mathcal{S}_{in}, \sigma_{out}\rangle$.

In the next sections, we present an approach for model expansion over an empty output vocabulary. In section 4.4, the approach is extended (in a straightforward way) to non-empty output vocabularies.

2.3 Grounding

Basically, *grounding* is the process of instantiating all variables with domain elements to obtain a propositional theory. The *full grounding* of a typed, free-variable free FO formula ψ, $G_{full}(\psi)$, is defined by Table 1. The size of the full grounding of a formula is exponential in the nesting depth of quantifiers and polynomial in the size of the domains.

More intelligent grounding techniques exist which reduce the size of the grounding, such as grounding with bounds [16].

3 Delayed theories

Lazy grounding (lazy mx) is an approach to interleave grounding and search. The key idea of our approach is to partly ground the input theory and to delay grounding of the

Table 1 The definition of the full grounding $G_{full}(\psi)$ of an FO formula ψ not containing free variables.

Original formula ψ	Full grounding $G_{full}(\psi)$
$P(\bar{d})$	$P(\bar{d})$
$\neg P(\bar{d})$	$\neg P(\bar{d})$
$\bigwedge_{i\in[1,n]} \varphi_i$	$\bigwedge_{i\in[1,n]} G_{full}(\varphi_i)$
$\bigvee_{i\in[1,n]} \varphi_i$	$\bigvee_{i\in[1,n]} G_{full}(\varphi_i)$
$\forall y[\![t]\!] : \varphi$	$\bigwedge_{d\in t} G_{full}(\varphi[y/d])$
$\exists y[\![t]\!] : \varphi$	$\bigvee_{d\in t} G_{full}(\varphi[y/d])$
$\varphi \equiv \varphi'$	$G_{full}(\varphi) \equiv G_{full}(\varphi')$

remainder. Conditions on the partial interpretation are derived which govern whether additional grounding is necessary. We call such conditions *delays*.

▶ **Example 1.** Consider the sentence $(\exists x[\![t]\!] : P(t))$ with t ranging from 1 to n. As long as $P(1)$ is not false, it can still be assigned a value (true) such that the formula becomes satisfied. Therefore, we can delay the remaining instantiations by replacing them with a new Tseitin symbol T, resulting in the ground clause $P(1) \vee T$ and the non-ground "delayed" formula $T \equiv \exists x[\![t\backslash 1]\!]$. The latter formula is only grounded further if T becomes true.

Such a condition on the partial structure I, $I(T) \neq \mathbf{t}$, is called a *delay*. As long as it is satisfied, no additional grounding is performed on the associated sentence. The result is a theory which is equivalent to the original one (as Tseitin transformation was used) and if a partial interpretation can be found which satisfies the ground portion and the delay, it can certainly be extended to a full model (setting $I(T)$ to $I(\exists x[\![t\backslash 1]\!])$). ☐

▶ **Example 2** (Continued)**.** Consider the theory consisting of the previous sentence and the additional sentence $(\forall x[\![t]\!] : P(x) \equiv \varphi(x))$, with φ a general formula not containing P. As long as $P(d)$ has not been assigned a value, $P(d)$ can still be assigned a value such that $P(d) \equiv \varphi[x/d]$ is consistent (namely $I(P(d)) = I(\varphi[x/d])$). Consequently, grounding only has to be applied for instantations of x with domain elements d for which $I(P(d))$ is not unknown. The delay is then the condition $I(\exists x[\![t]\!] : \neg P(d)) \neq \mathbf{u}$. ☐

Delayed grounding can lead to a significant reduction in the size of the grounding. In the case of $(\exists x[\![t]\!] : P(t))$, quantifier instantiation is only done partially. In the case of $(\forall x[\![t]\!] : P(x) \equiv \varphi(x))$, it is even completely avoided as long as the delay is satisfied.

3.1 Delays on formulae

A *delayed sentence* has the form $(\varphi)^{\delta}$ where φ is a sentence and δ is a delay condition (in short, a *delay*). A *delayed theory* $\mathcal{T}_d$ is a theory consisting of a ground theory $\mathcal{G}$ and a residual (non-ground) theory $\mathcal{D}$ consisting of *delayed sentences*. We will often denote it by $\langle \mathcal{G}, \mathcal{D} \rangle$. Such a delayed theory will be obtained by partially grounding the theory, resulting in $\mathcal{G}$, and partially delaying the grounding, resulting in $\mathcal{D}$.

Two types of delays are considered:

- A *true-delay*, denoted $\varphi \neq \mathbf{t}$, is satisfied in an interpretation I iff $I(\varphi) \neq \mathbf{t}$.
- A *known-delay*, denoted $\varphi = \mathbf{u}$, is satisfied in an interpretation I iff $I(\varphi) = \mathbf{u}$.

We say that a (partial) interpretation I satisfies (is a model of) a delayed sentence $(\varphi)^{\delta}$ if I satisfies φ. We say that I *weakly* satisfies $(\varphi)^{\delta}$ if I satisfies the delay or I satisfies φ. By extension, an interpretation weakly satisfies a (delayed) theory iff it satisfies all its sentences.

We will say that a delayed sentence $(\phi)^\delta$ is *active* in I if its condition δ is not satisfied, otherwise it is *inactive*. Conceptually, grounding will be triggered when a sentence becomes active, transforming it into ground sentences and inactive delayed sentences.

The lazy grounding algorithm will iteratively reduce delayed theories into "more ground" delayed theories. The main invariant of the algorithm is that any such delayed theory is a *partial grounding* of $\mathcal{T}$: A delayed theory $\mathcal{T}_d$ is a *partial grounding* of a theory $\mathcal{T}$ iff

- $\mathcal{T}$ and $\mathcal{T}_d$ are "logically equivalent" in the sense that each 2-valued model M of $\mathcal{T}$ can be extended to a model of $\mathcal{T}_d$ and vice versa, each 2-valued model of $\mathcal{T}_d$ satisfies $\mathcal{T}$.
- Each interpretation that weakly satisfies $\mathcal{T}_d$ has a two-valued extension that satisfies $\mathcal{T}$.

▶ **Example 3** (Continued)**.** The delayed theory introduced in example 2 is a partial grounding of its original theory. It consists of a ground theory $P(1) \lor T$ and of the delayed sentences

$$(T \equiv \exists x[\![t\backslash 1]\!])^{T\neq \mathbf{t}} \qquad (\forall x[\![t]\!] : P(x) \equiv \varphi(x))^{\exists x[\![t]\!]:\neg P(d)=\mathbf{u}}$$

The next section shows which delays can be safely introduced to guarantee this invariant.

4 Introducing delayed sentences

The lazy grounding component of the lazy mx algorithm is responsible for the grounding of an active delayed theory into a more ground, inactive one.[1] To this end, delayed sentences are replaced by a combination of ground and delayed sentences. This is either achieved with Tseitin introduction (section 4.1) or $\forall$-instantiation (section 4.2). The lazy mx algorithm itself is then presented in section 4.4.

4.1 Tseitin introduction

Recall from example 1 that $\exists x[\![t]\!] : P(x)$ was partially grounded to the ground formula $P(1) \lor T$ and a delayed sentence $((T \equiv (\exists x[\![t\backslash 1]\!] : P(x))))^{T\neq \mathbf{t}}$. We here describe this operation.

▶ **Definition 4** (Tseitin introduction)**.** Given a delayed theory $\mathcal{T}_d = \langle \mathcal{G}, \mathcal{D}\rangle$ and a set of occurrences of a formula φ in sentences ψ with $(\psi)^\delta \in \mathcal{T}_d$, the Tseitin introduction for φ in $\mathcal{T}_d$ is the delayed theory $\mathcal{T}_d$'$=\langle \mathcal{G}, \mathcal{D}'\rangle$ where $\mathcal{D}$' is obtained from $\mathcal{D}$ by

- substituting each selected occurrence of φ in $\mathcal{D}$ with the new propositional symbol T_φ
- adding a new delayed sentence $(T_\varphi \equiv \varphi)^{\delta'}$ where δ' is determined as follows:
  - If all selected occurrences of φ are monotone in $\mathcal{D}$, then $\delta' = (T_\varphi \neq \mathbf{t})$.
  - If all are anti-monotone, then $\delta' = (\neg T_\varphi \neq \mathbf{t})$.
  - Otherwise, $\delta' = (T_\varphi = \mathbf{u})$. □

Applying Tseitin introduction to any partial grounding of a theory $\mathcal{T}$ results in a partial grounding of $\mathcal{T}$.

▶ **Example 5.** Consider the theory $\mathcal{T} = P \equiv \forall x[\![t]\!] : Q(x)$. Applying Tseitin introduction to $\forall x[\![t\backslash 1]\!] : Q(x)$ results in the ground theory $P \equiv Q(1) \land T$ and the delayed sentence $(T \equiv \forall x[\![t\backslash 1]\!] : Q(x))^{T = \mathbf{u}}$. This delayed theory is a partial grounding of $\mathcal{T}$.

[1] Note that an initial theory $\mathcal{T}$ trivially corresponds to the delayed theory $\langle \emptyset, \{(\varphi)^{\mathbf{t}\neq\mathbf{t}} \mid \varphi \in \mathcal{T}\}\rangle$ with an empty ground theory and all its formulae active.

4.2 $\forall$-instantiation

Another approach to introducing delays applies to sentences of which a condition on their satisfiability can be derived. For some classes of formulae such conditions are well-known:

▶ **Example 6.** Consider the definite clause $\forall \overline{x}[\![\overline{t}]\!] : P_1(\overline{x}) \wedge \ldots \wedge P_n(\overline{x}) \Rightarrow Q(\overline{x})$. Any interpretation I in which none of the (ground) *heads* $Q(\overline{d})$ are false can be extended to an interpretation which satisfies all clauses, namely the extension in which all heads are true. Consequently, only instantiations with domain elements $\overline{d}$ for which $I(Q(\overline{d}))$ is false need to be grounded. The remaining instantiations can then be delayed on the falsity of their heads. Assume for example that only $I(Q(\overline{d}_1)) = \mathbf{f}$ for some $\overline{d}_1 \in \overline{t}$. The delayed theory $\mathcal{T}_d$

$$\mathcal{T}_d = \left\langle \phi[\overline{x}/\overline{d}_1] \Rightarrow Q(\overline{d}_1),\ \{(\forall \overline{x}[\![\overline{t} \backslash \overline{d}_1]\!] : \phi(\overline{x}) \Rightarrow Q(\overline{x}))^{\exists \overline{x}[\![\overline{t} \backslash \overline{d}_1]\!] : \neg Q(\overline{x})\ \neq\ \mathbf{t}}\} \right\rangle$$

is then a partial grounding of the definite clause under I. ☐

Below, it is shown formally how to delay instantiation for universally quantified disjunctive sentences based on their satisfiability. Delaying those is not captured by Tseitin introduction and they represent a class of formulae which occur often in practice. At the end of the section, it is shown how the approach can be extended to other classes of formulae.

▶ **Definition 7** ($\forall$-instantiation)**.** Consider a delayed theory $\mathcal{T}_d = \langle \mathcal{G}, \mathcal{D} \rangle$, a partial grounding of $\mathcal{T}$, and an interpretation I. Assume a sentence $\psi = \forall \overline{x}[\![\overline{t}]\!] : \bigvee_{i \in [1,m]} \varphi_i$ with $(\psi)^\delta \in \mathcal{D}$.[2] Applying $\forall$-instantiation to ψ for $\mathcal{T}_d$ under I consists of selecting a subset S_d of $\bigcup_{i \in [1,m]} \varphi_i$ such that each formula in S_d is a literal of which the symbol does not occur with opposite sign in any delay in $\mathcal{T}_d$. Assume nd denotes the set of tuples of domain elements which falsify all formulae in S_d under I (so they cannot be delayed). Then $\langle \mathcal{G}_{rem}, \mathcal{D}_{rem} \rangle$ is the grounding of the sentence $\forall \overline{x}[\![nd]\!] \bigvee_{i \in [1,m]} \varphi_i$. The remaining instantiations are delayed by the delay condition $\chi = \exists \overline{x}[\![\overline{t} \backslash nd]\!] : \bigwedge_{\varphi_i \in S_d} \neg \varphi_i \neq \mathbf{t}$. The result is the delayed theory $\mathcal{T}'_d = \left\langle \mathcal{G} \wedge \mathcal{G}_{rem}, \mathcal{D} - \{\psi\} \cup \mathcal{D}_{rem} \cup (\forall \overline{x}[\![\overline{t} \backslash nd]\!] \bigvee_{i \in [1,m]} \varphi_i)^\chi \right\rangle$. ☐

As for Tseitin introduction, it can be shown that applying $\forall$-instantiation to any partial grounding of a theory $\mathcal{T}$ results in a partial grounding of $\mathcal{T}$.

It should be noted that whether such a delay can contain a literal over a symbol P depends on occurrences of P in existing delays. If multiple delays are watching different truth assignments to the same symbol, inconsistencies might not be detected.

$\forall$-instantiation can be extended to other classes of formulae such as equivalences and non-monotone occurrences of quantifiers. For sentences of the form $\forall \overline{x}[\![\overline{t}]\!] : L(\overline{x}) \equiv \varphi(\overline{x})$ for example, the strategy is as outlined above except that χ becomes a known-delay. In the same fashion, the approach can be extended to *inductive definitions*[5], sets of rules of the form $(\forall \overline{x} : L(\overline{x}) \leftarrow \phi)$ evaluated by the well-founded semantics[13]. Furthermore, the exact delays used by $\forall$-instantiation allow us to trade-off propagation versus grounding size and towards solving query tasks. Details are out of the scope of this paper, but results of these ideas are included in the prototype implementation used in the experiments.

4.3 Delayed grounding algorithm

With these techniques, we can now give an informal (due to lack of space) presentation of the delayed grounding algorithm for_del_gnd. The algorithm takes as arguments an

[2] A delayed sentence $(\varphi \wedge \varphi')^\delta$ can be seen as the union of delayed sentences $(\varphi)^\delta$ and $(\varphi')^\delta$.

interpretation I, a theory $\mathcal{T}$ and the set of delays of all currently delayed sentences. It returns a delayed theory $\mathcal{T}_d$ inactive in I which is a partial grounding of $\mathcal{T}$.

We assume a standard top-down reduced grounding algorithm, such as in e.g. [15], which recursively visits the theory top-down to ground it. The algorithm keeps track of the set of ground and delayed sentences and of the context ((anti/non)-monotone).

Lazy grounding is applied in two different scenarios. Firstly, if a universally quantified disjunctive sentence is encountered for which a set of subformulas can be selected according to the above conditions, the sentence is delayed using ∀-instantiation (recursively grounding non-delayable instantiations). Secondly, for an existential quantification or disjunction, a non-false subformula is selected randomly[3] and grounded recursively; the remainder of the formula is delayed by Tseitin introduction. The second approach is also applied for non-monotone occurrences of universal quantifiers and conjunctions.

4.3.1 Incremental delayed grounding algorithm

Initially, for_del_gnd is applied to I and $\mathcal{T}$ (no delayed sentences yet), resulting in an initial delayed theory $\mathcal{T}_d$. In order to construct a weak model for a delayed theory $\mathcal{T}_d$, search and grounding are interleaved. When, during search, an interpretation I is constructed where some delays in $\mathcal{T}_d$ are active, further grounding is applied to the associated delayed sentences. This is achieved by iterating over all delayed sentences in $\mathcal{T}_d$ and *incrementally* applying for_del_gnd to each active delayed sentence. Each new ground sentence is added to the ground theory and the original delayed sentence is replaced by new delayed sentences (if any). This algorithm is denoted as inc_del_gnd. It takes as input an interpretation I and a delayed theory $\mathcal{T}$ and the result is a partial grounding of $\mathcal{T}$ which is inactive under I.

▶ **Example 8.** Consider $T = \exists x[\![t]\!] : (P(x) \land R(x)) \lor (\forall y[\![t']\!] : Q(x, y))$. Delayed grounding of this sentence is achieved by selecting a domain element $d \in t$ and applying for_del_gnd to $(P(d) \land R(d)) \lor (\forall y[\![t']\!] : Q(d, y))$ while applying Tseitin introduction to the residual subformula $\phi = \exists x[\![t \backslash d]\!] : (P(x) \land R(x)) \lor (\forall y[\![t']\!] : Q(x, y))$.

Applying for_del_gnd to $(P(d) \land R(d)) \lor (\forall y[\![t']\!] : Q(d, y))$ recursively calls for_del_gnd on $P(d) \land R(d)$ and Tseitin introduction on the other disjunct $\psi = (\forall y[\![t']\!] : Q(d, y))$. The resulting delayed theory consists of the ground sentence $(P(d) \land R(d)) \lor T_\psi \lor T_\phi$ and the true-delayed sentences:

$$(T_\psi \equiv (\forall y[\![t']\!] : Q(d, y)))^{T_\psi \neq \mathbf{t}}$$
$$(T_\phi \equiv \exists x[\![t \backslash d]\!] : (P(x) \land R(x)) \lor (\forall y[\![t']\!] : Q(x, y)))^{T_\phi \neq \mathbf{t}}$$
□

4.4 Lazy model expansion

The complete lazy model expansion algorithm, denoted lazy_mx and shown below, interleaves grounding and search based on a standard (incremental) CDCL search algorithm. The algorithm (shown below) gets as input a theory $\mathcal{T}$ and a pre-interpretation S_{in} and maintains the current delayed theory $\langle \mathcal{G}, \mathcal{D} \rangle$. The (current) ground theory provides the constraints used during search. If a conflict at root level is encountered, then $\mathcal{G}$ has no model and hence neither does $\mathcal{T}$ since $\langle \mathcal{G}, \mathcal{D} \rangle$ is a partial grounding of $\mathcal{T}$. If a delay is active, grounding is performed to construct a new delayed theory which is a partial grounding of $\mathcal{T}$. If all delays are inactive and the search algorithm detects that I satisfies all constraints in $\mathcal{G}$, I is a weak model of $\langle \mathcal{G}, \mathcal{D} \rangle$. As $\langle \mathcal{G}, \mathcal{D} \rangle$ is a partial grounding of $\mathcal{T}$, $\mathcal{T}$ is satisfiable.

[3] Better heuristics are part of future work.

```
lazy_mx (𝒯, I)
  ⟨𝒢, 𝒟⟩ := for_del_gnd(𝒯, I, ∅)
  while true do
    I := unit_propagation(𝒢,I)
    if (conflict detected)
      if (at root level) return false
      𝒢 := 𝒢 ∧ conflict clause
      I := I at state of backjump point
    else if (some delay in 𝒟 is active in I)
      ⟨𝒢, 𝒟⟩ := inc_del_gnd (⟨𝒢, 𝒟⟩,I)
    else if (satisfaction of 𝒢 in I is detected) return true
    else I := I ∪ {l} with l a search choice
```

If the lazy_mx algorithm returns *true*, $\mathcal{T}$ has a model that is more precise than I. If the algorithm returns *false*, no interpretation exists which is more precise than I and satisfies $\mathcal{T}$. The algorithm terminates if $\mathcal{T}$ and I are finite. If $\mathcal{T}$ has a finite number of sentences, termination is possible but not guaranteed (not even when a finite model exists).

Deciding atoms occurring in known-delays when I is already a weak model of $\mathcal{T}$ will obviously cause unnecessary grounding. As standard search algorithms decide all literals in the ground theory, we use a modified algorithm which tracks satisfaction of constraints in an efficient way without deciding all literals in the ground theory.

To handle non-empty output vocabularies σ_{out}, we modified the algorithm to always return models which are two-valued on σ_{out}. This is achieved by forcing the search algorithm to decide all atoms in the set of domain atoms of σ_{out} under $\mathcal{S}_{in}$.

5 Experiments

A prototype implementation was created within the IDP-3 system, a knowledge base system based on extensions of first-order logic. The IDP system is a state-of-the-art model expansion system, based on the ground-and-solve paradigm [14], [3].

Experiments were conducted with three setups: basic model expansion (denoted IDP), lazy model expansion by Tseitin introduction (IDP_T) and by Tseitin introduction and by ∀-instantiation ($\text{IDP}_{T,S}$).

The considered benchmarks represent a diverse set of problems, both existing benchmarks (e.g. from previous ASP competitions) and newly constructed ones. As most existing benchmarks are problems with a feasible grounding and difficult search part, we also created new instances which are computationally easier but have a very large grounding. This combination will allow to assess the strengths and weaknesses of the approach.

For each benchmark instance, runtime and grounding size are measured for each setup. Grounding size is measured as the number of literals over the input vocabulary. The grounding size of all setups is compared to the (theoretical) grounding size of the full grounding (see table 1). [4] The results are shown in table 2.

The experiments show that, for a range of benchmarks and instances, lazy mx by incremental grounding can be very beneficial. For most benchmarks, the grounding size is reduced orders of magnitude over both the full grounding and the reduced grounding as

[4] For the full grounding, the input structure is not taken into account. Consequently, even the grounding of the IDP setup can be smaller than the full grounding as IDP constructs a reduced grounding.

Table 2 Experimental results of applying lazy model expansion (IDP_T and $\text{IDP}_{T,S}$) compared to default model expansion (IDP). Grounding time (in seconds) is denoted as t, grounding size as G, $**$ denotes ASP competition instances. A timeout of 1000 seconds was used and a memory limit of 3 Gb, — indicates timeout or memory overflow. All experiments were run on an Intel Core 2 Machine (dual 2.40Ghz) running Ubuntu 10.4. The version of the IDP system used in the experiments and all data files are available from `http://dtai.cs.kuleuven.be/krr/research/experiments`.

Benchmark	G_{full}	G_{IDP}	G_{IDP_T}	$G_{\text{IDP}_{T,S}}$	t_{IDP}	t_{IDP_T}	$t_{\text{IDP}_{T,S}}$
func-1	$8.0 * 10^7$	$8.0 * 10^7$	$1.6 * 10^5$	540	99.03	4.07	0.1
func-2	∞	—	—	1370	—	—	0.1
bnq**	$1.4 * 10^8$	$1.1 * 10^5$	$1.1 * 10^5$	$6.8 * 10^4$	2.56	2.56	1.96
packing-1	$1.0 * 10^{10}$	$1.2 * 10^8$	$1.1 * 10^8$	$1.0 * 10^6$	171	172	5.0
packing-2	$3.1 * 10^{12}$	—	—	$2.2 * 10^7$	—	—	27.0
agentK	$5.0 * 10^6$	—	—	626	—	—	0.02
planning1	∞	—	—	385	—	—	0.29
planning2-1	$3.0 * 10^8$	$2.0 * 10^8$	$.05 * 10^6$	$4.3 * 10^4$	139.02	5.96	0.46
planning2-2	$3.0 * 10^{10}$	—	$5.1 * 10^8$	$2.5 * 10^6$	—	455.02	31.05
soko-18**	$1.6 * 10^8$	$8.3 * 10^7$	—	—	247.5	—	—
soko-L	$3.7 * 10^8$	—	$1.5 * 10^6$	$4.0 * 10^5$	—	16.0	6.0
reach-08**	$2.3 * 10^{18}$	—	—	60	—	—	26.05
reach-14**	$6.2 * 10^{14}$	—	—	$1.7 * 10^5$	—	—	3.36

done by IDP. Running times on many benchmarks go from untractable to solvable within seconds. Runtime is only worse for the sokoban 18 problem.

Tseitin introduction proves to be an advantage in benchmarks such as encoding functional dependencies and planning: problems which are generally solved by selecting an appropriate (small) subset of literals to assign, even if this choice is unguided. Indeed, it is enough to select one domain element for each function range or only actions for a small timeframe in many planning problems. On the other hand sokoban 18 shows that in hard planning problems, the incremental approach has an adverse effect on the search (introduction of large number of Tseitin literals). As expected, it has few effects on problems with a universal quantifications over large domains, such as packing 2, reachability and func 2.

Delaying using $\forall$-instantiation was expected to have a positive effect on most benchmarks, unless the loss in propagation is too significant (such as for sokoban 18). On all other benchmarks, it has an outspoken positive effect:

- Even for bounded N-Queens, a hard search problem, the grounding size is reduced and the performance increased, because non-propagating implications are not grounded.
- Problems with an infinite full grounding can be solved, such as planning problems over infinite times. The conjunction with Tseitin introduction is crucial to delay both existential and universal quantifiers.
- It acts as a kind of *dynamic dependency analysis*, selecting the parts of the theory which (hopefully) contribute to finding a model. This can be observed in particular in reachability (reach-*), in fact a query task generally solved by (static) dependency analysis. Additionally, the dynamic character of our approach is both at least as powerful and more general, during search only grounding what becomes relevant.

6 Related work

Within logic programming and ASP, static dependency analysis is applied as a means to reduce the size of the grounding up-front by detecting non-relevant parts of the theory. It has been implemented for example in [7] and [17]. Furthermore, lazy grounding techniques have been researched within ASP [6], [12] and CP [11]. Such techniques usually work on delaying grounding of specific constraints as long as they do not cause propagation, for example for all-different constraints, aggregates and equality reasoning. Our lazy mx approach is more general, as it performs dependency analysis dynamically and is able to delay grounding even when propagation is possible, but might be less powerful for constraints for which specific algorithms exist. Comparing those techniques in-depth is part of future work.

The model generation theorem prover Paradox [4] uses a grounding technique based on lazily extending the domain of the quantifiers. It first chooses a domain (all domain elements are symmetrical) and constructs the full grounding. If no model is found, it increases the domain size, until a model is found or a bound on the size is hit (if one could be derived). Such an approach stands orthogonal to the work presented in this paper and it is part of future work to combine the advantages of both approaches.

As part of future work, we will investigate the relation with techniques used to delay the grounding of quantifiers such as *skolemisation*, used e.g. in theorem proving, and congruence closure algorithms which reason on equality of terms, from the domain of SAT-Modulo-Theories. Another promising topic is that of undoing grounding on backtracking. In effect, it might be possible to track whether delays have become inactive again and to remove the associated constraints again. Such a strategy would allow to reduce the size of the grounding again when a different part of the search space comes under investigation.

7 Conclusion

Lazy model expansion is an approach to model expansion that interleaves solving and search. It can be highly beneficial when the original theory has a large (or infinite) grounding, because it tries to introduce just enough grounding to solve the problem. The disadvantages of lazy mx are that it provides less propagation than full grounding, and the order of grounding can effect search detrimentally. There remains much future work to improve lazy mx by incorporating ideas such as lifted unit propagation and devising better heuristics for controlling delay, but there are already examples where lazy mx is highly beneficial.

Acknowledgements

Broes De Cat is funded by the Institute for Science and Technology Flanders (IWT).

Research supported by Research Foundation-Flanders (FWO-Vlaanderen).

NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

References

1. Krzysztof R. Apt. *Principles of Constraint Programming*. Cambridge University Press, 2003.
2. Chitta Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2003.
3. Bart Bogaerts, Broes De Cat, Stef De Pooter, and Marc Denecker. The IDP framework reference manual. `http://dtai.cs.kuleuven.be/krr/software/idp3/documentation`, 2012.
4. Koen Claessen and Niklas Sörensson. New techniques that improve MACE-style model finding. In *MODEL*, 2003.
5. Marc Denecker and Eugenia Ternovska. A logic of nonmonotone inductive definitions. volume 9, 2008.
6. Claire Lefèvre and Pascal Nicolas. The first version of a new ASP solver : ASPeRiX. In *LPNMR*, pages 522–527, 2009.
7. Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. The DLV system for knowledge representation and reasoning. volume 7, pages 499–562, 2006.
8. David G. Mitchell and Eugenia Ternovska. A framework for representing and solving NP search problems. In Manuela M. Veloso and Subbarao Kambhampati, editors, *AAAI*, pages 430–435. AAAI Press / The MIT Press, 2005.
9. N. Nethercote, P.J. Stuckey, R. Becket, S. Brand, G.J. Duck, and G. Tack. Minizinc: Towards a standard CP modelling language. In C. Bessiere, editor, *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming*, volume 4741 of *LNCS*, pages 529–543. Springer-Verlag, 2007.
10. Ilkka Niemelä. Answer set programming: A declarative approach to solving search problems. In *JELIA*, pages 15–18, 2006. Invited talk.
11. O. Ohrimenko, P.J. Stuckey, and M. Codish. Propagation via lazy clause generation. *Constraints*, 14(3):357–391, 2009.
12. Alessandro Dal Palù, Agostino Dovier, Enrico Pontelli, and Gianfranco Rossi. Answer set programming with constraints using lazy grounding. In Patricia M. Hill and David Scott Warren, editors, *ICLP*, volume 5649 of *LNCS*, pages 115–129. Springer, 2009.
13. Allen Van Gelder. The alternating fixpoint of logic programs with negation. *Journal of Computer and System Sciences*, 47(1):185–221, 1993.
14. Johan Wittocx, Maarten Mariën, and Marc Denecker. The IDP system: a model expansion system for an extension of classical logic. In Marc Denecker, editor, *LaSh*, pages 153–165, 2008.
15. Johan Wittocx. *Finite Domain and Symbolic Inference Methods for Extensions of First-Order Logic*. PhD thesis, Department of Computer Science, K.U.Leuven, Leuven, Belgium, May 2010.
16. Johan Wittocx, Maarten Mariën, and Marc Denecker. Grounding FO and FO(ID) with bounds. *Journal of Artificial Intelligence Research*, 38:223–269, 2010.
17. M. Gebser and R. Kaminski and A. König and T. Schaub. Advances in *gringo* Series 3. In James P. Delgrande and Wolfgang Faber, editors, *LPNMR*, volume 6645 of *LNCS*, pages 345-351. Springer, 2011.
18. Heinz-Dieter Ebbinghaus and Jörg Flum and Wolfgang Thomas. Mathematical logic (2. ed.). Springer, 1994.

Unsatisfiability-based optimization in clasp*

Benjamin Andres, Benjamin Kaufmann, Oliver Matheis, and Torsten Schaub

University of Potsdam,
August-Bebel-Str. 89,
D-14482 Potsdam, Germany
{bandres,matheis,torsten}@cs.uni-potsdam.de

Abstract

Answer Set Programming (ASP) features effective optimization capacities based on branch-and-bound algorithms. Unlike this, in the area of Satisfiability Testing (SAT) the finding of minimum unsatisfiable cores was put forward as an alternative approach to solving Maximum Satisfiability (MaxSAT) problems. We explore this alternative approach to optimization in the context of ASP. To this end, we extended the ASP solver *clasp* with optimization components based upon the computation of minimal unsatisfiable cores. The resulting system, *unclasp*, is based on an extension of the well-known algorithms *msu1* and *msu3* and tailored to the characteristics of ASP. We evaluate our system on multi-criteria optimization problems stemming from realistic Linux package configuration problems. In fact, the ASP-based Linux configuration system *aspuncud* relies on *unclasp* and won four out of seven tracks at the 2011 MISC competition.

1998 ACM Subject Classification I.2.8 Problem Solving, Control Methods, and Search

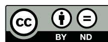
Keywords and phrases answer-set-programming, solvers

Digital Object Identifier 10.4230/LIPIcs.ICLP.2012.212

1 Introduction

Answer Set Programming (ASP,[2]) utilizes effective and elaborate optimization techniques based on branch-an-bound algorithms [11]. While these technique have shown to be able solve many problems efficiently, an alternative to this approach emerged in the field of Satisfiability Testing (SAT) for solving Maximum Satisfiability (MaxSAT,[7]) problems. The approach of using unsatisfiable cores has already shown to be successful by the *Maximum Satisfiability with UNsatisfiable COREs* (*MSUnCore*,[10]) solver, being ranged as the best solver in the industrial category in the 2008 MaxSAT evaluation. A closer survey shows that the *MSUnCore* is able to solve some instances that are difficult for the ASP solver *clasp* [5] efficiently. To this end, we propose a new algorithm for solving optimization problems in ASP, combining the *MSUnCore* solving techniques of *msu1* and *msu3* with regard of the characteristics of ASP. The algorithm is extended with techniques for multi-criteria optimization and utilizes the algorithm from [9] for solving weighted optimization problems. The implemented algorithm forms a branch of the *clasp* solver, *unclasp*, specialized for solving unweighted multi-criteria optimization ASP problems. In fact, *aspuncud*, an ASP-based Linux configuration system based on *unclasp*, won four out of seven tracks at the 2011 Mancoosi International Solver Competition (MISC) competition [8]. The *unclasp* solver is available in the lab section of the Potsdam Answer Set Solving Collection [4].

* This work was partially funded by the German Science Foundation (DFG) under grant SCHA 550/8-2.


© Benjamin Andres, Benjamin Kaufmann, Oliver Matheis, and Torsten Schaub;
licensed under Creative Commons License ND
Technical Communications of the 28th International Conference on Logic Programming (ICLP'12).
Editors: A. Dovier and V. Santos Costa; pp. 212–221

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

The remainder of the paper is structured as follows. The fundamentals of the *MSUnCore* algorithms are introduced in section 2. Section 3 presents our adaptation of the unsatisfiable based *MSUnCore* algorithms for solving ASP optimization, and its implementation into *unclasp* Our experimental results, including instances from MISC and a number of (un)weighted problems, are discussed in section 4. Section 5 concludes the paper.

2 The *MSUnCore* Algorithm

The *MSUnCore* solver features several strategies for solving unweighted, partial MaxSAT problems. Partial MaxSAT is an extension of the SAT problem, in which the number of satisfied clauses of a given subset of the SAT problem is to be maximized, while all other clauses of the problem must still hold. The clauses of the subset are called *soft*, while all other are *hard* clauses. A partial MaxSAT is unweighted if no clause of the subset is favored, and weighted otherwise. To this end, all *MSUnCore* strategies utilize unsatisfiability based approaches. The basic idea behind unsatisfiability based optimization is trying to solve the given problem and to extract an unsatisfiable core if the problem is not satisfiable. An unsatisfiable core is a subset of clauses of the original problem whose conjunction is still unsatisfiable. All *soft* clauses of the extracted core are relaxed, allowing the solver to arbitrarily satisfy one of the clauses. Afterwards, the problem is tried to be solved again with the relaxed clauses. This procedure is iterated until the problem is satisfied or an unsatisfiable core with no *soft* clauses is identified, meaning that the problem is unsatisfiable. Each of the four strategies of MSUnCore, *msu1-4*, implements a different approach in utilizing the identified unsatisfiable core. In the following the *msu1* algorithm is explained as an introduction to unsatisfiability based optimization. Subsequently, the features of the other *MSUnCore* algorithms are presented.

Algorithm 1 presents the pseudo code of the *msu1* strategy. The algorithm consists of a main loop, identifying a solution with the maximum number of satisfied clauses. To this end, the loop iterates through the following steps. First, the problem is passed to a solver. If the solver returns satisfiability, an optimal solution for the problem is found and the loop ends. Otherwise, the reason for unsatisfiability, i.e. an unsatisfiable core, is taken from the solver. Next, all soft clauses of the identified core are relaxed, that is, a unique variable v is added to the clause. Since the newly added relaxation variable is unique, the solver is able to arbitrarily set the variable to true and thus to satisfy the clause. If the identified core does not contain any soft clauses, the core can not be relaxed and the loop ends returning the problem as unsatisfiable, since every solution to the problem must injure at least one clause within the core by definition. Finally, an *at-most-1* constraint is added to the problem to ensure that at most one of the newly added relaxation variables is set to true. If V is the set of newly added relaxation variables, then the *at-most-1* constraint for *msu1* consists of the following clauses:

the clause $\bigvee_{v \in V} v$, and one clause in the form of $\neg v_i \vee \neg v_j$ for any $v_i, v_j \in V$.

The first clause ensures that at least one of the relaxation variables is true, while the others prevent that more than one is true. Thus, $|V|$ variables and $1 + \binom{|V|}{2}$ clauses are added to the problem. Since in every iteration only one additional *soft* clause is satisfied through relaxation, the first valid solution found by the SAT solver is optimal. The number of additional variables and clauses needed is a major disadvantage for the *msu1* algorithm, especially for bigger problems. The *msu2* and the *msu3* algorithms offer two approaches for reducing the number of needed clauses and relaxation variables, respectively.

By introducing additional relaxation variables *msu2* is able to reduce the number of additional clauses to encode the *at-most-1* constraint from $\Theta(|V|^2)$ to $\Theta(|V|)$. Since ASP offers its own efficient encoding of the *at-most-1* constraint, this approach is not further examined. As stated above, *msu1* adds a new relaxation variable for every *soft* clause in an indentified unsatisfiable core. This leads to clauses with many relaxation variables in the case of intersection between cores. Thus, creating several possible combinations for satisfying one set of clauses, obfuscating the solving process. The *msu3* algorithm trades the ability to distinct between identified unsatisfiable cores for a reduced number of relaxation variables. At first, all identified unsatisfiable cores are removed from the problem without substitution, until the problem is satisfiable. While this is not compulsory, it allows to identify disjoint unsatisfiable cores efficiently. Afterwards, the identified cores are relaxed, but instead of adding a relaxation variable each time a clause occurs in an identified core as in *msu1*, *msu3* relaxes each clause only once. Thereby, potentially reducing the number of used relaxation variables. While *msu1-3* use true top-down approaches for finding optimal solutions, *msu4* combines the *msu3* approach with bottom-up search. Instead of allowing only one additional *soft* clause to be relaxed in each iteration, *msu4* states that each subsequent solution must satisfy at least one additional *soft* clause without being relaxed. Thus, approaching the optimal solution from the lower end. All identified unrelaxable cores are treated as in *msu4*.

Algorithm 1 Iterative UNSAT Core Elimination of *msu1*.

```
T := ∅ {set of all relaxation variables}
while SAT solver returns UNSATISFIABLE do
  LET UC be the UNSAT core provided by the SAT solver
  S := ∅ {set of new relaxation variables for UC}
  for all Clause c ∈ UC do
    if c is relaxable then
      Allocate a new relaxation variable v
      c := c ∪ {v}
      S := S ∪ {v}
    end if
  end for
  if S = ∅ then
    return CNF UNSATISFIABLE
  else
    Add clauses enforcing the one-hot constraint for S to the SAT solver
    T := T ∪ S
  end if
end while
R := {v|v ∈ T, v = 1}; k := |R|
return Satisfying Assignment, k, R.
```

3 Implementation of *unclasp*

The problem of ASP optimization is strongly related to partial MaxSAT. Both problems consist of an unrelaxable rule set and a linear optimization function. While the literals of an ASP optimization rule can be interpreted as distinct soft clauses of a partial MaxSAT problem, the relaxation variables used to solve a partial MaxSAT can be used to form an

optimization rule in ASP. We utilize this correlation in our approach for developing an unsatisfiability based algorithm for ASP optimization. This approach tries to combine the advantages of *msu1* with the practical improvements of *msu3* in regard of the characteristics of ASP. To this end, we create a branch of the ASP solver *clasp*, utilizing its sophisticated ASP solving technique. The resulting system, *unclasp*, is specialized for unweighted, hierarchic optimization problems. Although, able to solve weighted problems as well. The advantages gained by porting the *MSUnCore's* unsatisfiability based approach of solving MaxSAT, to an ASP solving strategy are presented in the following. Afterwards, an extension for solving weighted problems, is explained. Finally, the implementation of our approach into *unclasp* is presented.

When solving ASP optimization problems with the *msu3* approach, the one-literal clauses of an ASP optimization rule offer a distinct advantage over the longer clauses of MaxSAT. Since, in *msu3* each clause is limited to only one relaxation variable, one can interpret the negated literal as its own relaxation variable.

Take, for example, the one-literal clause $\{\neg l\}$. If extended by the relaxation variable v we get $\{\neg l \vee v\}$, which is equivalent $v \leftarrow l$. Since, an optimal solution minimizes the number of true relaxation variables, and v is unique in the problem description, $v \longleftrightarrow l$ follows. Thus, $\{l\}$ is the relaxed clause of $\{\neg l\}$.

With this, *msu3* can be processed for ASP problems without additional variables. The next strong point of ASP is its management of cardinality constraints. Cardinality constraints are satisfied if the number of satisfied literals of the constraint are within a specified range. The ASP solver *clasp* is able to handle the constraint as a single rule simply by counting the number of satisfied literals. *Clasp* only generates the specific clauses for the cardinality constraint when needed for resolution. This is done on demand. This does not only allow to encode *at-most-1* constraints efficiently, but also to formulate *at-most-n* constraints. Being constraints, that enforces the usage of at most n relaxation variables.

In respect to these two characteristics of ASP, our approach works as follows: The problem is given to the solver, and in case of unsatisfiability the core is extracted. All clauses from the optimization rule in the core are relaxed as described above, and an *at-most-1* constraint from the new relaxation variables is formulated as in *msu1*. In difference to *msu1*, the relaxed clauses are marked as *hard* for any successive solving pass. This prevents them to be relaxed a second time. Instead, the new *at-most-1* constraint becomes a soft rule. When a *at-most-n* constraint is encountered in an unsatisfiable core, it is relaxed by an *at-most-(n+1)* rule for the same variables. This algorithm iterates until the problem is solved or an unsatisfiable core without *soft* clauses is encountered.

For dealing with weighted problems the idea from [9] is added to the management of unsatisfiable cores. After the core is identified, each containing clause is split into two. This is done in such a way, that a maximum equal weighted core and a remainder is obtained. Now, the equal weighted core is interpreted as unweighted, while the clauses in the remainder are added to the problem. For example, the weighted core $(a = 3, b = 5, c = 4)$ is split into an equal weighted core $(a = 3, b = 3, c = 3)$ and its remainder $(b = 2, c = 1)$. The equal weighted core is now interpreted as unweighted (a, b, c) and the weighted clauses $b = 2$ and $c = 1$ are added to the problem.

ASP allows to formulate multiple optimization rules and to order them in a hierarchy. Meaning that an optimal solution to a hierarchic optimization problem, has an optimal value in the highest hierarchy level and in all lower levels in an optimal value in respect to the solutions possible for its predecessor levels. This is handled as a sequence of distinct optimizations with the identified optimal values from the higher levels as additional criterion.

The implementation of *unclasp* is based on the *clasp* solver by utilizing a modification of the internal *clasp* function *ClaspFacade::solveIncremental*. This function runs one loop, divided into four phases as shown in figure 1. The phases are surrounded by the control functions *initStep* and *nextStep*, initializing the next pass through the loop and deciding whether another pass is necessary respectively. Figure 2 shows the added interface for implementing our approach.

The loop differentiates in our implementation between the first and all successive run-throughs. In the first pass, the *initStep* function is skipped and the problem is read in the *read* step. During *preprocess* the rules of the problem are translated into an internal representation and the constant *minimizeconstraint_*, holding all literals to be optimized, is formulated by the build in *ProgramBuilder* function. Next, in the *getAssumptions* phase, *assumelevel* is called, extracting the literals of the current optimization level from *minimize-constraint_* and copying them into the *assumptions_* set. This is done to account for literals that influence more than one optimization level. Also, the weights of the minimization literals are copied into the *weightmap_*. In the *solve* step, a solution to the problem with respect to the *assumptions_* is searched for. This leads to three different outcomes determining the behavior of the *nextStep* function. If the problem is unsatisfiable under assumptions, that is, the extracted unsatisfiable core consists at least one literal from *assumptions_*, the loop starts another pass, beginning with *nextStep*. If no literal from *assumptions_* is in the identified core, the problem is unsatisfiable and the incremental solving process is terminated. Finally, in the case of satisfiability, *assumelevel* is called, checking whether another optimization level exists. If no further level in the optimization hierarchy exists, the solution found is optimal and the loop terminates. Otherwise, the *assumptions_* are transformed into facts by the *factifyassumptions* function and *assumelevel* extracts the new *assumptions_* from *minimizeconstraint_*. Then, the optimization continues with another pass through the loop.

In all successive passes the *initStep* function calls the *internal2program* function, which translates the unsatisfiable core from the previous pass back into program variables. Afterwards, *analysecore* is called for managing the translated core. First, the minimal weight of the core's literals is identified and the weight of these literals in the *weightmap* are reduced by this amount. Then an *at-most-1* constraint for the core's literals is formulated by the *addconstraint* function. A guard variable with the previous identified weight allows the relaxation of the variable, i.e. marking it as *soft* with the appropriate weight. The guard variable is saved in *addtoassumptions_* for later usage in the *getAssumptions* phase, when an internal representation for the guard is generated. If a literal of the core is already a guard variable of an *at-most-n* constraint, an appropriate *at-most-(n+1)* variable is added to *addconstraints*. All weights in the *weightmap* that are reduced to 0 are removed from *assumptions_*. The *read* phase is skipped after the first pass and the new constraints from *initStep* are added into the internal representation. In the *getAssumptions* phase the variables of *addtoassumptions_* are mapped to the internal literals with help of the buildin *SolverStrategies::SymbolTable*. The internal literals are then added to *assumptions_*. The *solve* phase and the *nextStep* function behave as described above.

4 Experiments

We evaluate our implementation on optimization instances taken from the MISC and the ASP problem collection asparagus [1]. In order to be able to compare our approach from Section 3 with the *MSUnCore* algorithms, we implemented an ASP optimizing variant of *msu1*, *msu3* and *msu4* in *unclasp*. Thus, *unclasp* is in no need to reduce the number of

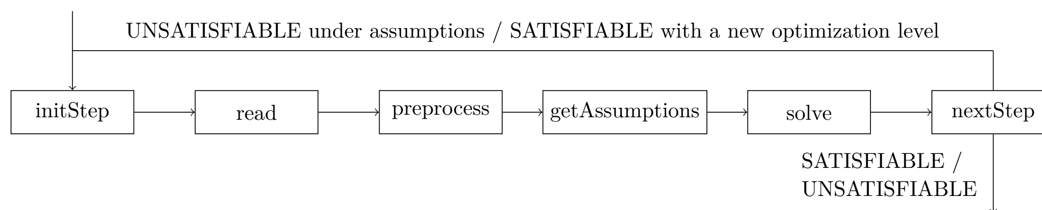


Figure 1 The incremental solving procedure.

Table 1 Runtime parameters of the optimizing strategies used.

new	`unclasp --opt-uncore=oll`
msu1	`unclasp --opt-uncore=msu1`
msu3	`unclasp --opt-uncore=msu3`
msu4	`unclasp --opt-uncore=msu4`
clasp	`clasp-2.0.0 --sat-prepro --restarts=128 --local-restarts --heuristic=VSIDS --solution-recording --opt-hierarch=1 --opt-heu=1`

clauses introduced by the *one-hot* condition followed by *msu2*. To compare the unsatisfiability based approach with the branch-and-bound approach we included the *clasp 2.0* solver into the benchmark. Table 1 presents the runtime parameters used for each strategy. The parameters for *clasp* are specialized for the MISC benchmark set. Below, we report the sequential runtimes on a Linux machine equipped with 3.4 GHz Intel Xeon CPUs and 32 GB RAM. Finally, a timeout was set after 300 seconds.

The MISC benchmark set consists of instances where, given a set of installed and available packages, a solution has to satisfy requests of package addition and removal, while minimizing the effect on the current installation. Packages may depend on or conflict each other, creating a combinational rich unweighted hierarchic optimization problem with a large number of widely independent optimization variables. The huge number of suboptimal solutions necessitate a sophisticated search heuristic and restart policy.

Figure 3 presents a solution cost distribution plot [6] of the runtime measured. The x-axis shows the runtime for solving one instance, while the y-axis labels the percentage of instances solved within the time. The plot shows that for smaller runtimes, up to 10 seconds, all approaches are able to solve a comparable number of instances. On larger runtimes the approaches start to differentiate from each other and a gap emerges. The *msu1* and our new approach clearly dominate the benchmark, able to solve all but two instances in less than 70 seconds, each. Please note, that these two instances were not solved by any approach within the time frame of 300 seconds, demonstrating their complexity. The next two best performing solvers are *msu3* and *msu4* with 10 and 12 unsolved instances, respectively. While being able to solve some of the instances faster than the unsatisfiability based approaches, *clasp* did not solve a higher percentage of MISC instances on any given runtime. In addition *clasp* could not solve 43 instances before reaching time-out. The performance of *msu4* and *clasp* indicates, that the bottom-up strategy used by them is not ideal. The benchmark shows further, that the reduction of relaxation variables pursued by *msu3* is not advantageous for the MISC problem class, overall.

To evaluate the performance of the unsatisfiability based strategies in general and our proposed algorithm in particular, we selected nine general optimization problems from the

```
class UncoreControl: public IncrementalControl {
public:
    virtual void initStep(ClaspFacade& f);
    void getAssumptions(ClaspFacade& f, LitVec& a);
    virtual bool nextStep(ClaspFacade& f);
protected:
    void initassumptions(ClaspFacade& f);
    void assumelevel(ClaspFacade& f);
    void removefromassumptions(const LitVec&);
    void factifyassumptions(ClaspFacade& f);
    virtual void analysecore(ClaspFacade& f, const LitVec&);
    Var addconstraint(ClaspFacade& f, const LitVec&,
        unsigned int bound);
    Literal internal2program(ClaspFacade& f, Literal v);

    const MinimizeConstraint* minimizeconstraint_;
    std::set<Literal> assumptions_;
    VarVec addtoassumptions_;
    Guardtable guardtable_;
    unsigned int level_;
    bool nextlevel_;
    std::vector<unsigned int>min_;
    SolveStats totalstats_;
    std::map<Literal,int> weightmap_;
};
```

Figure 2 Interface for implementing *unclasp* into *clasp*.

asparagus benchmark collection. Most of them were used in the ASP'09 competition [3]. Table 2 shows the selected optimization problems and the corresponding runtime of the optimizing technique. The upper six problems are unweighted, while the lower three are weighted. In the **15-puzzle** and **sokoban** instances the solver has to minimize the number of steps needed to solve the problem. Since all steps build upon each other, there are no combinatorics in the minimization function. Thus, the size of the identified core in every iteration of *unclasp* is one (i.e. the next step). Because of that, the unsatisfiability based algorithms behave the same. The branch-and-bound based algorithm of *clasp* also achieves similar runtime results. The **clique** problem describes the problem of finding the maximal clique in an undirected graph, given. Here, the bottom-up algorithms of *msu4* and *clasp* are clearly superior to the top-down algorithms, as the instances become larger. Of the top-down algorithms, our new approach is able to achieve the best runtime results, while *msu1* could not solve three of the five instances. The other three unweighted problems, **labyrinthpath**, **minimum postage stamp problem** (**mpsp**) and **weight bounded dominating set** (**wbds**) use a large number of optimization variables and also return large unsatisfiable cores. This highlights the different strategies of the unsatisfiability based approaches. In **labyrinthpath** *msu1's* ability to distinguish identified cores is advantageous over *msu3* reduced number of additional variables, as with the MISC benchmark set. In **mpsp** the opposite is the case, and *msu3* outperforms *msu1*, able to solve two more instances. Interestingly, *msu4* performs similar to *msu3*. Our approach achieves the best results on

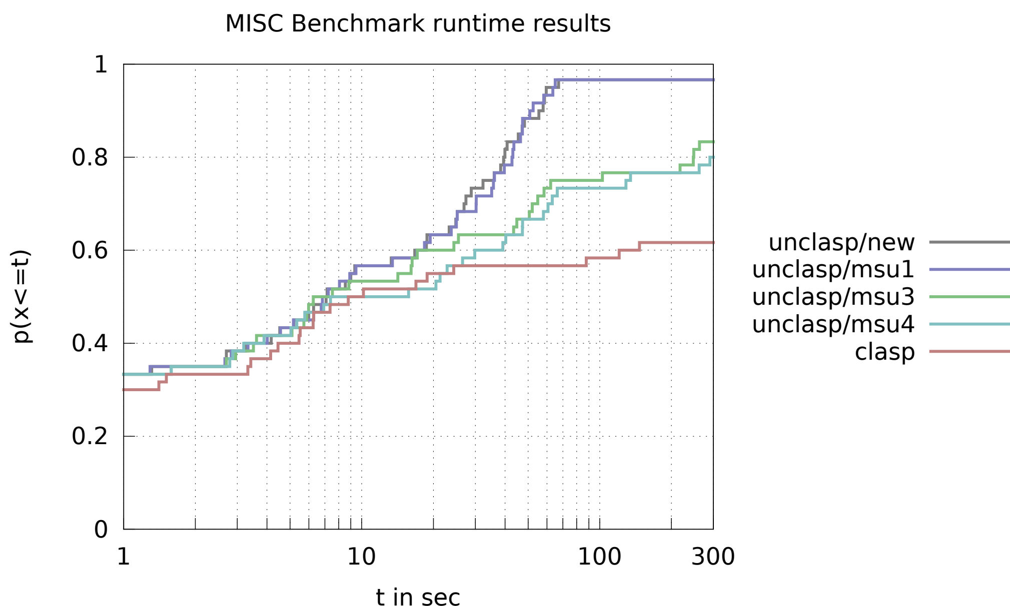


Figure 3 Solution cost distribution plot of the MISC instance runtime.

these three problems. Especially on the **mpsp** and **wbds** instances, where it is able to solve instances every other approach could not. *clasp* is not able to compete with the unsatisfiability based approach with the exception of *msu1* in the **mpsp** problem.

The bottom-up algorithms *msu4* and *clasp* perform well in the weighted problems, **companyctrl**, **opendoors** and **fastfood**, while *clasp* has a better runtime on **opendoors** and *msu4* on **fastfood**. The **companyctrl** problem demonstrates the benefit of the splitting algorithm. While the other approaches are able to solve the problem without much effort, the *msu3* algorithm, which is incompatible with the algorithm from [9], has much difficulty. The **fastfood** problem shows the limit of the top-down unsatisfiability based algorithms. While the problem is easy in general, the cores get too big after a few loop iterations. On this problem class the variable reduction of the *msu3* algorithm shows to be useful, allowing *msu3* to solve three of the five instances tested.

Overall, unsatisfiability based optimization has shown to be efficient in solving unweighted optimization problems. The benchmark demonstrates the effectiveness of our implementation, but also shows its limits on weighted optimization problems.

5 Discussion

We presented an approach to bring unsatisfiabilty based optimization to ASP. Our approach combines the *msu1* and *msu3* strategies of the *MSUnCore* MaxSAT solver with regard to the special characteristics of ASP. The resulting algorithm is specialized for solving unweighted hierarchical optimization problems. In fact, our implementation of the proposed algorithm into *unclasp* was able to to solve a number of problems faster than the traditional branch-and-bound approach utilized by *clasp*. This shows that the *unclasp* approach is a useful addition to the algorithms currently used by *clasp* for solving unweighted problems, expanding *clasp's* portfolio of optimization strategies.

Table 2 Runtime of selected optimization instances from asparagus.

	oll	msu1	msu3	msu4	clasp
15-puzzle					
init1	9,58	9,42	9,53	9,45	6,74
init1simple	0,93	0,93	0,93	0,94	2,02
init1simple2	0,42	0,42	0,42	0,42	1,48
init2	81,84	82,74	82,27	81,81	73,85
init3	13,38	13,28	13,44	13,43	16,81
sokoban					
dimitr__yo51s10	0,24	0,24	0,25	0,25	0,21
dimitr__yo51s14	1,65	1,65	1,66	1,66	1,06
dimitr__yo51s17	2,54	2,55	2,55	2,58	1,59
dimitr__yo52s10	1,67	1,67	1,67	1,67	0,65
dimitr__yo55s10	0,81	0,81	0,81	0,83	0,49
clique					
gen10__25	0,02	0,02	0,02	0,02	0,02
gen200__8000	4,77	300,00	21,35	0,73	0,77
gen300__20000	28,87	300,00	143,45	4,17	3,94
gen75__1000	0,06	8,19	0,12	0,07	0,05
gen100__2000	0,14	300,00	0,51	0,16	0,08
labyrinthpath					
l10__10__01	1,88	1,89	1,88	1,86	18,33
l11__11__01	2,41	1,88	2,39	2,37	300,00
l12__12__01	3,44	4,82	212,90	236,66	300,00
l13__13__01	300,00	300,00	300,00	300,00	300,00
l14__14__01	300,00	300,00	300,00	300,00	300,00
mpsp					
mpsp30-2	0,04	0,13	0,08	0,09	0,05
mpsp36-2	0,12	3,35	0,44	1,02	0,30
mpsp48-2	2,99	300,00	50,90	53,01	116,59
mpsp54-2	1,96	300,00	64,83	36,83	80,59
mpsp60-2	17,14	300,00	300,00	300,00	300,00
wbds					
r100__400__11__1	72,01	300,00	300,00	300,00	300,00
r100__400__11__13	3,40	300,00	300,00	300,00	300,00
r100__400__11__9	4,16	300,00	300,00	300,00	300,00
r150__600__11__17	300,00	300,00	300,00	300,00	300,00
r150__600__11__3	300,00	300,00	300,00	300,00	300,00
companyctrl					
02-company	0,85	0,88	285,46	0,93	0,74
12-company	3,96	4,02	4,52	4,08	3,91
22-company	1,46	1,52	300,00	1,52	1,13
32-company	0,96	0,94	300,00	0,92	0,85
42-company	1,54	1,51	1,76	1,71	1,52
opendoors					
level__00	0,09	0,10	0,09	0,11	0,11
level__05	0,19	0,26	0,19	0,22	0,20
level__10	0,37	0,37	0,37	0,62	0,28
level__17	24,73	6,76	24,17	71,98	4,92
level__28	300,00	300,00	300,00	300,00	18,64
fastfood					
a5.16.dl	300,00	300,00	32,36	12,15	13,02
a5.4.dl	300,00	300,00	300,00	3,84	1,15
fa8.17.dl	300,00	300,00	24,42	8,72	5,92
a8.8.dl	300,00	300,00	300,00	276,82	300,00
a9.11.dl	300,00	300,00	28,94	7,29	6,15

References

1. http://asparagus.cs.uni-potsdam.de.
2. C. Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2003.
3. M. Denecker, J. Vennekens, S. Bond, M. Gebser, and M. Truszczyński. The second answer set programming competition. pages 637–654.
4. M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and M. Schneider. Potassco: The Potsdam answer set solving collection. *AI Communications*, 24(2):105–124, 2011.
5. M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. clasp: A conflict-driven answer set solver. pages 260–265.
6. H. Hoos and T. Stützle. *Stochastic Local Search: Foundations & Applications*. Elsevier / Morgan Kaufmann, 2004.
7. D. Johnson. *Approximation algorithms for combinatorial problems*. Journal of Computer and System Sciences, Academic, 9, 1974.
8. mancoosi. http://www.mancoosi.org.
9. V. Manquinho, J. Marques-Silva, and J. Planes. Algorithms for weighted Boolean optimization. pages 495–508.
10. msuncore. http://www.csi.ucd.ie/staff/jpms/soft/soft.php.
11. P. Simons, I. Niemelä, and T. Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1-2):181–234, 2002.

An FLP-Style Answer-Set Semantics for Abstract-Constraint Programs with Disjunctions*

Johannes Oetsch, Jörg Pührer, and Hans Tompits

Technische Universität Wien,
Institut für Informationssysteme 184/3,
Favoritenstraße 9–11, A–1040 Vienna, Austria,
`{oetsch,puehrer,tompits}@kr.tuwien.ac.at`

Abstract

We introduce an answer-set semantics for abstract-constraint programs with disjunction in rule heads in the style of Faber, Leone, and Pfeifer (FLP). To this end, we extend the definition of an answer set for logic programs with aggregates in rule bodies using the usual FLP-reduct. Additionally, we also provide a characterisation of our semantics in terms of unfounded sets, likewise generalising the standard concept of an unfounded set. Our work is motivated by the desire to have simple and rule-based definitions of the semantics of an answer-set programming (ASP) language that is close to those implemented by the most prominent ASP solvers. The new definitions are intended as a theoretical device to allow for development methods and methodologies for ASP, e.g., debugging or testing techniques, that are general enough to work for different types of solvers. We use abstract constraints as an abstraction of literals whose truth values depend on subsets of an interpretation. This includes weight constraints, aggregates, and external atoms, which are frequently used in real-world answer-set programs. We compare the new semantics to previous semantics for abstract-constraint programs and show that they are equivalent to recent extensions of the FLP semantics to propositional and first-order theories when abstract-constraint programs are viewed as theories.

1998 ACM Subject Classification D.1.6 Logic Programming, D.3.1 Formal Definitions and Theory, I.2.4 Knowledge Representation Formalisms and Methods

Keywords and phrases answer-set programming, abstract constraints, aggregates, disjunction

Digital Object Identifier 10.4230/LIPIcs.ICLP.2012.222

1 Introduction

In order to reflect various programming needs, the basic answer-set programming (ASP) language, as originally defined by Gelfond and Lifschitz [12], has been extended in several ways to accommodate constructs like aggregates, weight constraints, and external atoms. *Abstract-constraint programs* [21, 23] are generalised logic programs providing abstractions of such commonly-used constructs and thus are perfectly suited to study different language extensions in a uniform manner. Hereby, abstract constraints are dedicated literals whose truth value depends on a set of propositional atoms.

In this paper, we consider abstract-constraint programs with disjunction in the heads and define an answer-set semantics for this kind of programs in the style of Faber, Leone, and Pfeifer ("FLP" for short), based on a simple reduct-based definition extending the original one defined for disjunctive logic program for aggregates in rule bodies [6]. The FLP semantics

* This work was partially supported by the Austrian Science Fund (FWF) under grant P21698.

© Johannes Oetsch, Jörg Pührer, and Hans Tompits;
licensed under Creative Commons License ND
Technical Communications of the 28th International Conference on Logic Programming (ICLP'12).
Editors: A. Dovier and V. Santos Costa; pp. 222–234

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

has been introduced to provide an intuitive handling of aggregates and is implemented in the solvers `DLV` [4, 16] and `DLVHEX` [3]. Recently, the FLP semantics has also been extended to propositional theories [30] and to first-order theories with aggregates [1]. However, in contrast to these extensions, the language we consider can be viewed as the smallest superset of the languages supported by current state-of-the-art ASP solvers.

Besides the basic reduct-based definition of answer sets, we also introduce a characterisation of our semantics in terms of *unfounded sets*, generalising the standard concept of an unfounded set [5].

Concerning the semantics for abstract-constraint programs in general, among the different proposals in the literature [23, 29, 27, 22, 20, 19, 26], to the best of our knowledge, only the work of Shen, You, and Yuan [27] deals with disjunctions in the head, i.e., they consider the same language as we do. Their semantics coincides with ours for the case of convex abstract-constraint programs, which is also the fragment that is currently implemented in common ASP solvers, but their approach depends on an involved program transformation that introduces fresh atoms—a potential advantage of our definition lies in its simplicity. Moreover, while we treat abstract-constraint atoms in the spirit of the FLP semantics, Shen, You, and Yuan [27] handle them the same way as Son, Pontelli, and Tu [29]. Relations to semantics for more restricted classes of abstract-constraint programs follow from known results.

Our main motivation for developing the characterisations discussed in this paper is of a rather practical nature. We want to have clear, declarative, and rule-based definitions that capture the languages of a majority of modern ASP solvers to a large extent. The new characterisations are intended as a theoretical device to facilitate uniform development methods and methodologies for ASP, like debugging or testing techniques [25, 13], that are general enough to work for different types of solvers. Indeed, since sufficiently efficient ASP solvers became available in the late 1990s, there has never been a standard for implemented ASP languages. Different ASP solvers support different language features, some of which are syntactic sugar, while others add expressiveness to the formalism. In particular, the languages of `DLV`, `Clasp` [10, 11], and of other solvers based on the grounder `lparse` and its de-facto successor `Gringo`, like `smodels` [24, 28], `cmodels` [18], and `pbmodels` [20], support different features. For instance, `DLV` allows for disjunction in rule heads which are not supported in `Clasp` and many related solvers. These, on the other hand, allow for weight constraints [28] in rule heads, whereas aggregates in `DLV` are restricted to appear in rule bodies only.[1]

The semantics characterised in this paper conservatively extends that of `DLV`, providing a theoretical basis for adding, e.g., choice rules to the language of `DLV`. In particular, our characterisation in terms of unfounded sets can be seen as a practical step towards an implementation in `DLV` as unfounded sets are central elements of the evaluation strategy of this solver. The introduced semantics also coincides with that of Simons, Niemelä, and Soininen [28] implemented in `Clasp` whenever no negative weights appear in weight constraints. Negative weights are rarely used and their semantics have been considered unintuitive by some authors [9, 8]. Thus, our characterisations lay a solid foundation for programming support methods operating on both solver dialects of `Clasp` and `DLV`. Besides that, they are of theoretical interest as they clarify the role of aggregate domains in rule heads in extensions of the FLP semantics. Moreover, the reduct-based definition identifies a single condition on the spoiling interpretation that is necessary for extending the original

[1] Note that the ASP solver `ClaspD` [2] supports both disjunctions and aggregates (more precisely, weight constraints) in rule heads but not within the same rule.

definition of the FLP semantics to programs with aggregates in rule heads.

Besides the relation of our semantics with the one by Shen, You, and Yuan [27], as pointed out above, we also discuss relations to other proposals of semantics for abstract-constraint programs. As mentioned, the FLP semantics has been extended to propositional theories by Truszczyński [30] for comparison with the semantics by Ferraris [7, 8]. Like the definition of Ferraris, the FLP semantics for propositional theories depends on a recursively defined reduct. Our results show that the semantics introduced in this paper is equivalent to that proposed by Truszczyński when abstract-constraint programs are translated into theories. In this sense, our definitions reflect the semantics by Truszczyński for programs with disjunctive rules.

This paper is organised as follows. In the next section, we give some background on abstract-constraint programs and discuss how special literals often used in real-world answer-set programs can be expressed as abstract-constraint atoms. In Section 3, we first recapitulate the FLP semantics for the fragment of abstract-constraint programs corresponding to the logic-programming language it was originally designed for and discuss shortcomings of a straightforward extension of the FLP semantics to full abstract-constraint programs. We then continue with our reduct-based semantics and the characterisation in terms of unfounded sets. Section 4 presents relations to other semantics of abstract-constraint programs. Moreover, we discuss the relation to recent extensions of the FLP semantics to theories. We conclude the paper in Section 5. For space reasons, most proofs are omitted.

2 Preliminaries

We assume a fixed propositional language based on a countable set $\mathcal{A}$ of (propositional) atoms. We use "*not*" as the symbol for *default negation*. An *abstract-constraint atom*, or *c-atom*, is a pair $A = \langle D, C\rangle$, where $D \subseteq \mathcal{A}$ is the *domain* of A, denoted by D_A, and $C \subseteq 2^D$ is a collection of sets of atoms, called the *satisfiers* of A, denoted by C_A. The domain of a default negated c-atom *not* A is given by $D_{not\ A} = D_A$. For an atom a, we identify the c-atom $\langle\{a\}, \{\{a\}\}\rangle$ with a. We call such c-atoms *elementary*.

An *abstract-constraint program*, or simply *program*, is a finite set of rules of the form

$$A_1 \vee \cdots \vee A_k \leftarrow A_{k+1}, \ldots, A_m, not\ A_{m+1}, \ldots, not\ A_n, \tag{1}$$

where $0 \leq k \leq m \leq n$ and any A_i for $1 \leq i \leq n$ is a c-atom. For a rule r of form (1), $\mathrm{B}(r) = \{A_{k+1}, \ldots, A_m,\ not\ A_{m+1}, \ldots, not\ A_n\}$ is the *body* of r, $\mathrm{B}^+(r) = \{A_{k+1}, \ldots, A_m\}$ is the *positive body* of r, $\mathrm{B}^-(r) = \{A_{m+1}, \ldots, A_n\}$ is the *negative body* of r, and $\mathrm{H}(r) = \{A_1, \ldots, A_k\}$ is the *head* of r. If $\mathrm{B}(r) = \emptyset$ and $\mathrm{H}(r) \neq \emptyset$, then r is a *fact*. For facts, we usually omit the symbol "$\leftarrow$". The domain of a rule r is $D_r = \bigcup_{X \in \mathrm{H}(r) \cup \mathrm{B}(r)} D_X$. A rule r of form (1) is *normal* if $k = 1$. A program is *normal* if it contains only normal rules. A program is a *logic program* if it contains only elementary c-atoms. Furthermore, a program is an *elementary-head program* if only elementary c-atoms appear in rule heads.

An *interpretation* is a set of atoms. For two sets I and X of atoms, $I|_X = I \cap X$ is the *projection* of I to X. An interpretation I *satisfies* a c-atom $\langle D, C\rangle$, symbolically $I \models \langle D, C\rangle$, if $I|_D \in C$. Moreover, $I \models not\ \langle D, C\rangle$ iff $I \not\models \langle D, C\rangle$.

A c-atom A is *monotone* if, for all interpretations I, I', if $I \subset I'$ and $I \models A$, then also $I' \models A$. A c-atom A is *convex* if, for all interpretations I, I', I'', if $I \subset I' \subset I''$, $I \models A$, and $I'' \models A$, then also $I' \models A$. Moreover, a program is monotone (resp., convex) if all contained c-atoms are monotone (resp., convex). An interpretation I satisfies a set S of c-atoms, symbolically $I \models S$, if $I \models A$ for all $A \in S$. Moreover, I satisfies a rule r, symbolically

$I \models r$, if $I \models \mathrm{B}(r)$ implies $I \models A$ for some $A \in \mathrm{H}(r)$. As well, I satisfies a set Π of rules, symbolically $I \models \Pi$, if $I \models r$ for every $r \in \Pi$. If $I \models \Pi$, we say that I is a *model* of Π.

A rule r such that $I \models \mathrm{B}(r)$ is called *active under* I. The set $\Pi^I = \{r \in \Pi \mid I \models \mathrm{B}(r)\}$ of all active rules of a program Π under an interpretation I is the *FLP-reduct of* Π [6].

As mentioned in the introduction, c-atoms are used to represent special literals used in logic programming, like aggregates and weight constraints, for formal study. Such special literals have in common that their truth values are determined by sets of atoms in an interpretation. Throughout this paper we will identify such special literals with c-atoms. As examples, since our motivation is to obtain characterisations of a semantics close to that of the popular answer-set solvers `Clasp` and `DLV`, we next illustrate how frequently-used language constructs, viz. *weight constraints* as used in `Clasp` and *aggregates* as used in `DLV`, can be represented as c-atoms. We consider variable-free variants only since variables are not needed in the remainder of the paper. Note that both `Clasp` and `DLV` rely on a grounding step before solving.

Simons, Niemelä, and Soininen [28] introduced *weight constraints* for normal logic programs. A weight constraint is an expression of form

$$l\ [a_1 = w_1, \ldots, a_k = w_k, \mathit{not}\ a_{k+1} = w_{k+1}, \ldots, \mathit{not}\ a_n = w_n]\ u\,,$$

where each a_i is an atom and each weight w_i is a real number, for $1 \leq i \leq n$. The lower bound l and the upper bound u are either a real number, ∞, or $-\infty$. However, the authors effectively require weights to be non-negative, as in their semantics negative weights are eliminated in a pre-processing step that has been claimed to lead to unintuitive results in several works [9, 8]. If all weights are non-negative, weight constraints are convex. Intuitively, the sum of weights w_i of those atoms a_i, $1 \leq i \leq k$, that are true and the weights of the atoms a_i, $k < i \leq n$, that are false must lie within the lower and the upper bound. More formally, an interpretation I satisfies a weight constraint if

$$l \leq (\sum_{1 \leq i \leq k, a_i \in I} w_i + \sum_{k < i \leq n, a_i \notin I} w_i) \leq u\,.$$

A special form of a weight constraint is a *cardinality constraint* where all weights are 1. The intuition is that lower and upper bounds define how many of the contained atoms may be true in an answer set. A further specialised form of a cardinality constraint is a *choice atom* that is of the form

$$0\ [a_1 = 1, \ldots, a_k = 1]\ k\,.$$

Choice atoms are often used in the head of a rule for non-deterministically guessing a subset of its domain $\{a_1, \ldots, a_k\}$. They are often abbreviated as $\{a_1, \ldots, a_k\}$.

A weight constraint

$$l\ [a_1 = w_1, \ldots, a_k = w_k, \mathit{not}\ a_{k+1} = w_{k+1}, \ldots, \mathit{not}\ a_n = w_n]\ u$$

corresponds to the c-atom $\langle D, C\rangle$, where $D = \{a_1, \ldots, a_n\}$ and

$$C = \{X \subseteq D \mid l \leq (\sum_{1 \leq i \leq k, a_i \in X} w_i + \sum_{k < i \leq n, a_i \notin X} w_i) \leq u\}\,.$$

We next define aggregates following Faber [5]. A *ground set* is a set of pairs of the form $\langle \vec{c} : I\rangle$, where $\vec{c}$ is a list of constants and I is a set of atoms. An *aggregate function* is of the form $f[S]$, where S is a ground set and f is an *aggregate function symbol*. Intuitively,

f stands for a mapping from multisets of constants to constants. An aggregate atom is of the form $f[S] \prec c$, where $f[S]$ is an aggregate function, c is a constant called *guard*, and $\prec \in \{=, <, \leq, \geq, >\}$ is a predefined comparison operator. Given an interpretation I and a ground set S, $I(S)$ is the multiset

$$[c_1 \mid \langle c_1, \ldots, c_n : I' \rangle \in S, I' \subseteq I] .$$

Then, an aggregate atom $f[S] \prec c$ is satisfied by I if $f(I(S)) \prec c$. Moreover, a default negated aggregate atom *not* $f[S] \prec c$ is satisfied by I if $f[S] \prec c$ is not satisfied by I. An aggregate atom $f[S] \prec c$ can be expressed as a c-atom

$$\langle D, \{X \subseteq D \mid f(X(S)) \prec c\} \rangle ,$$

where $D = \bigcup_{\langle \vec{c}:I' \rangle \in S} I'$.

As an example, consider the aggregate atom $\#count[S] = 1$, where

$$S = \{\langle 2 : queen_2_1 \rangle, \langle 2 : queen_2_2 \rangle, \langle 2 : queen_2_3 \rangle, \langle 2 : queen_2_4 \rangle\},$$

stemming from an instantiation of an encoding of the n-queens problem with $n = 4$. Intuitively, the aggregate atom is true when only one queen is located on row 2 of a chessboard. The aggregate function symbol $\#count$ maps a multiset of constants to its cardinality. Hence, under interpretation $I_1 = \{queen_2_3\}$, we have that $I_1(S) = [2]$, therefore $\#count(I_1(S)) = 1$, and hence $\#count[S] = 1$ is satisfied by I_1. For $I_2 = \{queen_2_3, queen_2_4\}$, we have $I_1(S) = [2, 2]$, therefore $\#count(I_2(S)) = 2$, and $\#count[S] = 1$ is not satisfied by I_2.

3 Reduct-Based Answer-Set Semantics

Before presenting our actual definition of an FLP-style semantics for abstract-constraint programs, we first recapitulate the FLP semantics by Faber, Pfeifer, and Leone [6] for disjunctive logic programs with aggregates appearing in rule bodies only and afterwards discuss the shortcomings of a straightforward extension of their definition to full abstract-constraint programs.

3.1 Prelude: FLP-Semantics for Elementary-Head Programs and a Straightforward Extension

As stated above, Faber, Pfeifer, and Leone [6] defined a semantics for disjunctive logic programs with aggregates appearing in rule bodies only. This class of programs, viewed as abstract-constraint programs, corresponds to the fragment of elementary-head programs. We refer to their semantics as the *FLP semantics*, defined as follows.

▶ **Definition 1** ([6]). Let Π be an elementary-head program. Then, an interpretation I is an *FLP answer set of* Π if $I \models \Pi^I$ and there is no $I' \subset I$ such that $I' \models \Pi^I$. The set of all FLP answer sets of Π is denoted by $AS_{FLP}(\Pi)$.

For the same class of programs, Faber [5] provided a definition of unfounded sets that we generalise to full abstract-constraint programs later on. Note that Faber considers strong negation and partial interpretations which we do not cover in this paper.

▶ **Definition 2** ([5]). Let Π be an elementary-head program and I an interpretation. Then, a set X of atoms is *unfounded in* Π *with respect to* I if, for each rule $r \in \Pi$ with $\mathrm{H}(r) \cap X \neq \emptyset$,

- $I \not\models \mathrm{B}(r)$,
- $I \setminus X \not\models \mathrm{B}(r)$, or
- $I \models l$, for some $l \in \mathrm{H}(r) \setminus X$.

As shown by Faber [5], a model I of a program Π is an FLP answer set of Π iff $I \cap X = \emptyset$, for each unfounded set X for Π with respect to I.

Now, let us call the *extended FLP semantics* the one obtained from Definition 1 by keeping the conditions of the definition but allowing Π to be a general abstract-constraint program. This straightforward extensions leads to undesired results, however, as we illustrate next.

As stated earlier, a popular form of aggregates used in the head of rules in ASP are choice atoms. Consider the program consisting of the fact

$$\langle \{a, b\}, \{\emptyset, \{a\}, \{b\}, \{a, b\}\} \rangle$$

which corresponds to the choice atom $\{a, b\}$. Here, the intended behaviour of a choice atom, viz. expressing a non-deterministic choice between sets $\emptyset$, $\{a\}$, $\{b\}$, and $\{a, b\}$, can only be achieved if non-minimal answer sets are permitted. The extended FLP semantics, however, allows only the empty set as an answer set of this program.

We are interested in a notion of answer set that prevents minimisation between the different satisfiers of an abstract-constraint atom and thus allows for using choice atoms with their usual meaning. This is introduced in the following.

3.2 Basic Definition and Unfounded Sets

▶ **Definition 3.** Let Π be an abstract-constraint program and I an interpretation. Then, I is an *answer set* of Π if $I \models \Pi^I$, and there is no $I' \subset I$ such that

(i) $I' \models \Pi^I$, and
(ii) for every $r \in \Pi^I$ with $I' \models \mathrm{B}(r)$, there is some $A \in \mathrm{H}(r)$ with $I' \models A$ and $I'|_{D_A} = I|_{D_A}$.

The set of answer sets of Π is denoted by $AS(\Pi)$.

This definition differs from the one of Faber, Pfeifer, and Leone [6] by the additional Condition (ii) on the spoiling interpretation I'. Intuitively, the purpose of this condition is to prevent minimisation within c-atoms.

▶ **Example 4.** Consider program Π_1 consisting of the fact

$$\langle \{a, b\}, \{\{a\}, \{b\}, \{a, b\}\} \rangle$$

that realises a choice of at least one atom from $\{a, b\}$. The answer sets of Π_1 are given by $\{a\}$, $\{b\}$, and $\{a, b\}$. Without Condition (ii), however, we would lose the answer set $\{a, b\}$ as, e.g., $\{a\} \subseteq \{a, b\}$ and $\{a\} \models \Pi^{\{a,b\}}$.

Opposed to the extended FLP semantics for programs where such a choice cannot be expressed without introducing auxiliary atoms, we do not enforce subset-minimal answer sets.

The next example illustrates that there are however minimisation effects between different c-atoms in a disjunction.

▶ **Example 5.** Consider the program

$$\Pi_2 = \langle\{a,b\},\{\{a\},\{b\},\{a,b\}\}\rangle \vee \langle\{a,c\},\{\{a,c\}\}\rangle$$

that also consist of a single (disjunctive) fact. Interpretations $\{a\}$, $\{b\}$, and $\{a,b\}$ are answer sets of Π_2. However, the satisfier $\{a,c\}$ of the second disjunct is not an answer set. Here, $\{a\}$ is the spoiling interpretation, since for

$$A = \langle\{a,b\},\{\{a\},\{b\},\{a,b\}\}\rangle$$

we have $\{a\} \models A$ and $\{a\}|_{D_A} = \{a,c\}|_{D_A}$.

Often, answer sets are computed following a two-step strategy: First a model of the program is built and in the second step it is checked whether this model obeys a foundedness condition ensuring that it is an answer set. Intuitively, every set of atoms in an answer set must be "supported" by some active rule that derives one of the atoms. Here, it is important that the reason for this rule to be active does not depend on the atom it derives. Such rules are referred to as *external support* [14]. In what follows, we extend this notion to our setting.

▶ **Definition 6.** Let r be a rule, X a set of atoms, and I an interpretation. Then, r is an *external support for X with respect to I* if

- $I \models \mathrm{B}(r)$,
- $I \setminus X \models \mathrm{B}(r)$,
- there is some $A \in \mathrm{H}(r)$ with $X|_{D_A} \neq \emptyset$ and $I|_{D_A} \subseteq S$ for some $S \in C_A$, and
- for all $A \in \mathrm{H}(r)$ with $I \models A$ we have $(X \cap I)|_{D_A} \neq \emptyset$.

We next show how answer sets can be characterised in terms of external supports.

▶ **Theorem 7.** *Let Π be a program and I an interpretation. Then, I is an answer set of Π iff I is a model of Π and every X with $\emptyset \subset X \subseteq I$ has an external support $r \in \Pi$ with respect to I.*

To complete the picture, we express the absence of an external support in an interpretation by extending the concept of an *unfounded set* [17, 5] to abstract-constraint programs (for the case of total interpretations). Defining unfounded sets in terms of external supports is motivated by the duality of these notions as discussed by Lee [14].

▶ **Definition 8.** Let Π be a program, X a set of atoms, and I an interpretation. Then, X is *unfounded in Π with respect to I* if there is no rule $r \in \Pi$ that is an external support for X with respect to I.

Note that this is a conservative extension of Definition 2 for elementary-head programs.

Theorem 7 now immediately yields the following result:

▶ **Theorem 9.** *Let Π be a program and I an interpretation. Then, I is an answer set of Π iff I is a model of Π, and there is no set X with $\emptyset \subset X \subseteq I$ that is unfounded in Π with respect to I.*

Faber [5] also provides a characterisation of answer sets based on the *unfounded-freeness* property for the class of programs he considered. This concept can be lifted to the case of abstract-constraint programs under our semantics.

▶ **Definition 10.** Let Π be a program and I an interpretation. Then, I is *unfounded-free* in Π if $I \cap X = \emptyset$ for each unfounded set X in Π with respect to I.

Opposed to Theorems 7 and 9, the definition of unfounded-freeness does not restrict the considered unfounded sets to subsets of the interpretation. Therefore, it is important to note that due to the definition of external support, the part of an unfounded set contained in the interpretation is itself an unfounded set.

▶ **Proposition 11.** Let X be a set of atoms, and I an interpretation. If a rule r is an external support for $I \cap X$ with respect to I then r is an external support for X with respect to I.

▶ **Lemma 12.** *Let X be a set of atoms, Π a program, and I an interpretation. If X is unfounded in Π with respect to I then $I \cap X$ is unfounded in Π with respect to I.*

We conclude the section with the result that characterises answer sets in terms of unfounded-free models, generalising Corollary 3 of Faber [5].

▶ **Theorem 13.** *Let Π be a program and I an interpretation. Then, I is an answer set of Π iff I is a model of Π and unfounded-free in Π.*

Proof. ($\Rightarrow$) Suppose that I is an answer set of Π. By Theorem 9, I is a model of Π and it holds that ($*$) there is no set X with $\emptyset \subset X \subseteq I$ that is unfounded in Π with respect to I. Assume that I is not unfounded-free in Π. Then, there is some unfounded set X for Π with respect to I such that $I \cap X \neq \emptyset$. Hence, by Lemma 12, $I \cap X$ is an unfounded set in Π with respect to I, contradicting ($*$).
($\Leftarrow$) Towards a contradiction, assume that I is not an answer set of Π. By Theorem 9, there must be some set X with $\emptyset \subset X \subseteq I$ that is unfounded in Π with respect to I. Hence, as thus $I \cap X \neq \emptyset$, I is not unfounded-free in Π. ◀

4 Relation to other Semantics

In this section, we shed some light on commonalities and differences of our semantics with related proposals. First, we discuss relations to semantics that follow the tradition of Simons, Niemelä, and Soininen [28] and then to other FLP-style semantics. A characteristic difference of the two categories of semantics is how non-convex body literals may give support to atoms in an interpretation.

As an example, consider the program consisting of the following rules:

$$\begin{aligned} & a \leftarrow \langle \{a, b\}, \{\emptyset, \{a, b\}\} \rangle, \\ & a \leftarrow b, \quad \text{and} \\ & b \leftarrow a. \end{aligned}$$

While $\{a, b\}$ is an answer set under FLP-style semantics, it is not considered stable in, e.g., the semantics discussed in the following subsection.

4.1 Semantics in the Tradition of Simons, Niemelä, and Soininen

Shen, You, and Yuan [27] defined a stable model semantics for abstract-constraint programs involving disjunction, i.e., the language fragment they consider is the same as in our setting. Let us call a stable model following Shen, You, and Yuan [27] an *SYY stable model*.[2]

The following result can be shown:

[2] Their construction is quite involved and is omitted here for space reasons.

▶ **Theorem 14.** *Let* Π *be a program such that all c-atoms appearing in a body of* Π *are convex. If* I *is an answer set of* Π*, then* I *is an SYY stable model of* Π*.*

Regarding the converse direction, an even stronger result holds:

▶ **Theorem 15.** *Let* Π *be a program. If* I *is an SYY stable model of* Π*, then* I *is an answer set of* Π*.*

Due to known results from the literature [27, 19, 29], Theorems 14 and 15 imply that our semantics is equivalent to a range of semantics proposed for more restricted classes of abstract-constraint programs including ones for *normal monotone abstract-constraint programs* [23, 22] and *normal convex abstract-constraint programs* [20] that are based on a non-deterministic one-step provability operator.

Furthermore, there are semantics defined for normal abstract-constraint programs where every answer set in the respective approach is an answer set as defined in our paper and where, if the considered programs are convex, also the converse holds, i.e., an answer set as defined in this paper is also an answer set in the respective approach. In particular, these include

- the approach by Liu et al. [19] based on computations,
- the work of Son, Pontelli, and Tu [29] that use the concept of conditional satisfaction of c-atoms for defining their semantics, and
- the reduct-based semantics by Shen and You [26].

Liu and Truszczyński [20] showed that their semantics for normal convex abstract-constraint programs resembles that of normal programs with weight constraints [28] with non-negative integer weights. As stated earlier, this type of weight constraints can be represented by convex abstract-constraint atoms. Due to the relation of the semantics by Liu and Truszczyński and ours, answer sets as defined in this paper coincide with stable models as defined by Simons, Niemelä, and Soininen for this class of programs. This semantics has been implemented in `smodels` and the state-of-the-art ASP solver `Clasp`.

4.2 Semantics in the Style of Faber, Pfeifer, and Leone

The straightforwardly extended FLP semantics for abstract-constraint programs, as discussed in Section 3, and our proposed semantics are interrelated as follows.

▶ **Theorem 16.** *For any program* Π*, each extended FLP answer set of* Π *is an answer set of* Π*.*

As intended, for the restricted setting of elementary-head programs that was considered by Faber, Pfeifer, and Leone [6], our semantics coincides with theirs.

▶ **Theorem 17.** *For an elementary-head program* Π*, it holds that* $AS(\Pi) = AS_{FLP}(\Pi)$*.*

Proof. $AS_{FLP}(\Pi) \subseteq AS(\Pi)$ holds by Theorem 16. Assume now that $I \in AS(\Pi)$ but $I \notin AS_{FLP}(\Pi)$. From $I \in AS(\Pi)$ it follows that $I \models \Pi^I$. Hence, by Definition 1, there must be some $I' \subset I$ such that $I' \models \Pi^I$. Furthermore, by Definition 3, there must be some $r \in \Pi^I$ such that $I' \models \mathrm{B}(r)$ and $(*)$ for all $l \in \mathrm{H}(r)$ with $I' \models l$, $I'|_{D_l} \neq I|_{D_l}$ holds. From $r \in \Pi^I$, $I' \models \mathrm{B}(r)$, and $I' \models \Pi^I$, we get that $I' \models \mathrm{H}(r)$. Thus, there is some $l' \in \mathrm{H}(r)$ with $I' \models l$. From the definition of the satisfaction relation follows $I'|_{D_l} = \{l\}$. As $I' \subset I$ and $D_l = \{l\}$, we get $I|_{D_l} = \{l\}$, and hence $I'|_{D_l} = I|_{D_l}$. As this contradicts $(*)$, $AS(\Pi) = AS_{FLP}(\Pi)$ must hold. ◀

Truszczyński [30] introduced an FLP-style semantics for propositional theories. A main goal of his paper is to study the differences between the semantics by Faber, Pfeifer, and Leone and that of Ferraris [8]. It is worth mentioning that the same differences to the latter apply to the semantics defined in this paper. In particular, they differ in the treatment of default negated literals that are non-convex. For further information on the relation between these families of semantics, we refer to the work of Truszczyński [30] and of Lee and Meng [15] who reduce elementary-head programs under the FLP semantics to propositional formulas under the semantics of Ferraris.

For comparison with the work of Truszczyński, we consider propositional theories over the language determined by $\mathcal{A}$ and the Boolean connectives $\bot$, $\wedge$,$\vee$, and $\supset$. Moreover, we use the shorthands $\top = \bot \supset \bot$ and $\neg f = f \supset \bot$. Given an interpretation I and a formula f, the classical satisfaction relation $I \models f$ is defined as usual. Also, following custom, we identify empty disjunctions with $\bot$ and empty conjunctions with $\top$.

▶ **Definition 18** ([30]). Let f be a propositional formula and I an interpretation. The *T-reduct*, f^I, of f is defined inductively as follows, where a is an atom, $\circ \in \{\wedge, \vee\}$, and g and h are propositional formulas:

$$\bot^I = \bot.$$

$$a^I = \begin{cases} a & \text{if } I \models a, \\ \bot & \text{otherwise.} \end{cases}$$

$$(g \circ h)^I = \begin{cases} g^I \circ h^I & \text{if } I \models g \circ h, \\ \bot & \text{otherwise.} \end{cases}$$

$$(g \supset h)^I = \begin{cases} g \supset h^I & \text{if } I \models g \text{ and } I \models h, \\ \top & \text{if } I \not\models g, \\ \bot & \text{otherwise.} \end{cases}$$

For a propositional theory F, F^I is defined as $\{f^I \mid f \in F\}$.

▶ **Definition 19** ([30]). Let F be a propositional theory and I an interpretation. Then, I is a *T-answer set* of F iff I is a subset-minimal model of F^I.

Note that any T-answer set of F is also a model of F. In order to compare our semantics and the semantics by Truszczyński, we use a standard translation of abstract-constraint programs to propositional theories. To this end, we use the following representation of abstract-constraint atoms in terms of DNF formulas.

▶ **Definition 20** ([27]). Let $A = \langle D, C\rangle$ be an abstract constraint atom where D consists of atoms only. Then,

$$\varphi(A) = \bigvee_{X \in C} ((\bigwedge_{l \in X} l) \wedge (\bigwedge_{l \in D \setminus X} \neg l)).$$

We extend the translation $\varphi(\cdot)$ to rules and abstract-constraint programs as follows.

▶ **Definition 21.** Let r be a rule of the form (1) where every A_i, $1 \leq i \leq n$, is an abstract-constraint atom whose domain is restricted to atoms. Then, $\varphi(r) = \varphi_B(r) \rightarrow \varphi_H(r)$, where

$$\varphi_H(r) = \varphi(A_1) \vee \cdots \vee \varphi(A_k) \text{ and}$$
$$\varphi_B(r) = \varphi(A_{k+1}) \wedge \cdots \wedge \varphi(A_m) \wedge \neg\varphi(A_{m+1}) \wedge \cdots \wedge \neg\varphi(A_n).$$

Finally, for a program Π, we define the propositional theory $\varphi(\Pi) = \{\varphi(r) \mid r \in \Pi\}$.

Obviously, for a rule r and an interpretation I, $I \models \mathrm{H}(r)$ iff $I \models \varphi_H(r)$, and $I \models \mathrm{B}(r)$ iff $I \models \varphi_B(r)$.

We next present the relation of our semantics to the approach of Truszczyński.

▶ **Theorem 22.** *Let Π be a program and I an interpretation. Then, I is an answer set of Π iff I is a T-answer set of $\varphi(\Pi)$.*

As Bartholomew, Lee, and Meng [1] have shown that their semantics for first-order theories with aggregates extends that of Truszczyński, the same relation applies to our approach.

5 Conclusion

In this work, we presented a new definition of answer sets for disjunctive abstract-constraint programs and a respective characterisation in terms of unfounded sets. The underlying semantics is a conservative extension of that by Faber, Pfeifer, and Leone [6] for disjunctive logic programs with aggregates in rule bodies only to the case where aggregates are also allowed in rule heads. Moreover, we showed that our semantics is also equivalent to a range of semantics that follow the understanding of Simons, Niemelä, and Soininen [28] for convex programs. Thereby, we reached our goal of providing simple definitions of an answer set that captures the essence of the semantics as implemented in popular ASP solvers like `Clasp` and `DLV`.

As regards future work, we are currently working on novel debugging techniques supporting software developers in writing answer-set programs that exploit the characterisations presented in this paper. Moreover, it would be interesting to explore how our notion of external support relates to loop formulas for abstract constraint programs as defined by You and Liu [31].

Acknowledgements

We would like to thank the reviewers for their constructive comments which helped to improve this paper.

References

1 Michael Bartholomew, Joohyung Lee, and Yunsong Meng. First-order extension of the FLP stable model semantics via modified circumscription. In *Proceedings of the 22nd International Joint Conference on Artificial Intelligence* (*IJCAI 2011*), pages 724–730. AAAI Press, 2011.

2 Christian Drescher, Martin Gebser, Torsten Grote, Benjamin Kaufmann, Arne König, Max Ostrowski, and Torsten Schaub. Conflict-driven disjunctive answer set solving. In *Proceedings of the 11th International Conference on Principles of Knowledge Representation and Reasoning* (*KR 2008*), pages 422–432. AAAI Press, 2008.

3 Thomas Eiter, Giovambattista Ianni, Roman Schindlauer, and Hans Tompits. Effective integration of declarative rules with external evaluations for semantic-web reasoning. In *Proceedings of the 3rd European Semantic Web Conference* (*ESWC 2006*), volume 4011 of *Lecture Notes in Computer Science*, pages 273–287. Springer, 2006.

4 Thomas Eiter, Nicola Leone, Cristinel Mateis, Gerald Pfeifer, and Francesco Scarcello. The KR system DLV: Progress report, comparisons and benchmarks. In *Proceedings of the 6th International Conference on Principles of Knowledge Representation and Reasoning* (*KR 1998*), pages 406–417. Morgan Kaufmann Publishers, 1998.

5 Wolfgang Faber. Unfounded sets for disjunctive logic programs with arbitrary aggregates. In *Proceedings of the 8th International Conference on Logic Programming and Nonmonotonic Reasoning* (*LPNMR 2005*), volume 3662 of *Lecture Notes in Computer Science*, pages 40–52. Springer, 2005.

6 Wolfgang Faber, Gerald Pfeifer, and Nicola Leone. Semantics and complexity of recursive aggregates in answer set programming. *Artificial Intelligence*, 175(1):278–298, 2011.

7 Paolo Ferraris. Answer sets for propositional theories. In *Proceedings of the 8th International Conference on Logic Programming and Nonmonotonic Reasoning* (*LPNMR 2005*), volume 3662 of *Lecture Notes in Computer Science*, pages 119–131. Springer, 2005.

8 Paolo Ferraris. Logic programs with propositional connectives and aggregates. *ACM Transactions on Computational Logic*, 12(4):25, 2011.

9 Paolo Ferraris and Vladimir Lifschitz. Weight constraints as nested expressions. *Theory and Practice of Logic Programming*, 5(1-2):45–74, 2005.

10 Martin Gebser, Benjamin Kaufmann, André Neumann, and Torsten Schaub. clasp: A conflict-driven answer set solver. In *Proceedings of the 9th International Conference on Logic Programming and Nonmonotonic Reasoning* (*LPNMR 2007*), volume 4483 of *Lecture Notes in Computer Science*, pages 260–265. Springer, 2007.

11 Martin Gebser, Benjamin Kaufmann, and Torsten Schaub. The conflict-driven answer set solver clasp: Progress report. In *Proceedings of the 10th International Conference on Logic Programming and Nonmonotonic Reasoning* (*LPNMR 2009*), volume 5753 of *Lecture Notes in Computer Science*, pages 509–514. Springer, 2009.

12 Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9(3/4):365–386, 1991.

13 Tomi Janhunen, Ilkka Niemelä, Johannes Oetsch, Jörg Pührer, and Hans Tompits. On testing answer-set programs. In *Proceedings of the 19th European Conference on Artificial Intelligence* (*ECAI 2010*), volume 215 of *Frontiers in Artificial Intelligence and Applications*, pages 951–956. IOS Press, 2010.

14 Joohyung Lee. A model-theoretic counterpart of loop formulas. In Leslie Pack Kaelbling and Alessandro Saffiotti, editors, *Proceedings of the 19th International Joint Conference on Artificial Intelligence* (*IJCAI 2005*), pages 503–508, Denver, CO, USA, 2005. Professional Book Center.

15 Joohyung Lee and Yunsong Meng. On reductive semantics of aggregates in answer set programming. In *Proceedings of the 10th International Conference on Logic Programming and Nonmonotonic Reasoning* (*LPNMR 2009*), volume 5753 of *Lecture Notes in Computer Science*, pages 182–195. Springer, 2009.

16 Nicola Leone, Gerald Pfeifer, Wofgang Faber, Thomas Eiter, Georg Gottlob, Somina Perri, and Francesco Scarcello. The DLV system for knowledge representation and reasoning. *ACM Transactions on Computational Logic*, 7(3):499–562, 2006.

17 Nicola Leone, Pasquale Rullo, and Francesco Scarcello. Disjunctive stable models: Unfounded sets, fixpoint semantics, and computation. *Information and Computation*, 135(2):69–112, 1997.

18 Yuliya Lierler. CMODELS – SAT-based disjunctive answer-set solver. In *Proceedings of the 8th International Conference on Logic Programming and Nonmonotonic Reasoning* (*LPNMR 2005*), volume 3662 of *Lecture Notes in Computer Science*, pages 447–451. Springer, 2005.

19 Lengning Liu, Enrico Pontelli, Tran Cao Son, and Mirosław Truszczyński. Logic programs with abstract constraint atoms: The role of computations. *Artificial Intelligence*, 174(3-4):295–315, 2010.

20 Lengning Liu and Mirosław Truszczyński. Properties and applications of programs with monotone and convex constraints. *Journal of Artificial Intelligence Research*, 27:299–334, 2006.

21 V. Wiktor Marek and Jeffrey B. Remmel. Set constraints in logic programming. In *Proceedings of the 7th International Conference on Logic Programming and Nonmonotonic Reasoning* (*LPNMR 2004*), volume 2923 of *Lecture Notes in Computer Science*, pages 167–179. Springer, 2004.

22 Victor W. Marek, Ilkka Niemelä, and Mirosław Truszczyński. Logic programs with monotone abstract constraint atoms. *Theory and Practice of Logic Programming*, 8(2):167–199, 2008.

23 Victor W. Marek and Mirosław Truszczyński. Logic programs with abstract constraint atoms. In *Proceedings of the 19th National Conference on Artificial Intelligence* (*AAAI 2004*), pages 86–91. AAAI Press, 2004.

24 Ilkka Niemelä and Patrik Simons. Smodels - An implementation of the stable model and well-founded semantics for normal logic programs. In *Proceedings of the 4th International Conference on Logic Programming and Nonmonotonic Reasoning* (*LPNMR 1997*), volume 1265 of *Lecture Notes in Computer Science*, pages 420–429. Springer, 1997.

25 Johannes Oetsch, Jörg Pührer, and Hans Tompits. Stepping through an answer-set program. In *Proceedings of the 11th International Conference on Logic Programming and Nonmonotonic Reasoning* (*LPNMR 2011*), volume 6645 of *Lecture Notes in Computer Science*, pages 134–147. Springer, 2011.

26 Yi-Dong Shen and Jia-Huai You. A generalized Gelfond-Lifschitz transformation for logic programs with abstract constraints. In *Proceedings of the 22nd Conference on Artificial Intelligence* (*AAAI 2007*), pages 483–488. AAAI Press, 2007.

27 Yi-Dong Shen, Jia-Huai You, and Li-Yan Yuan. Characterizations of stable model semantics for logic programs with arbitrary constraint atoms. *Theory and Practice of Logic Programming*, 9(4):529–564, 2009.

28 Patrik Simons, Ilkka Niemelä, and Timo Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1–2):181–234, 2002.

29 Tran Cao Son, Enrico Pontelli, and Phan Huy Tu. Answer sets for logic programs with arbitrary abstract constraint atoms. *Journal of Artificial Intelligence Research*, 29:353–389, 2007.

30 Mirosław Truszczyński. Reducts of propositional theories, satisfiability relations, and generalizations of semantics of logic programs. *Artificial Intelligence*, 174(16-17):1285–1306, 2010.

31 Jia-Huai You and Guohua Liu. Loop formulas for logic programs with arbitrary constraint atoms. In *Proceedings of the 23rd AAAI Conference on Artificial Intelligence* (*AAAI 2008*), pages 584–589. AAAI Press, 2008.

Reconciling Well-Founded Semantics of DL-Programs and Aggregate Programs*

Jia-Huai You[1], John Morris[1], and Yi Bi[2]

1 **Department of Computing Science**
University of Alberta, Canada
2 **School of Computer Science and Technology**
Tianjin University, China

Abstract

Logic programs with aggregates and description logic programs (dl-programs) are two recent extensions to logic programming. In this paper, we study the relationships between these two classes of logic programs, under the well-founded semantics. The main result is that, under a satisfaction-preserving mapping from dl-atoms to aggregates, the well-founded semantics of dl-programs by Eiter et al., coincides with the well-founded semantics of aggregate programs, defined by Pelov et al. as the least fixpoint of a 3-valued immediate consequence operator under the ultimate approximating aggregate. This result enables an alternative definition of the same well-founded semantics for aggregate programs, in terms of the first principle of unfounded sets. Furthermore, the result can be applied, in a uniform manner, to define the well-founded semantics for dl-programs with aggregates, which agrees with the existing semantics when either dl-atoms or aggregates are absent.

1998 ACM Subject Classification I.2.4 Knowledge Representation Formalisms and Methods

Keywords and phrases Well-founded semantics, description logic programs, aggregate logic programs, three-valued logic.

Digital Object Identifier 10.4230/LIPIcs.ICLP.2012.235

1 Introduction

In logic programming beyond positive logic programs, almost all semantics of the current interest can be traced back to the origin of two semantics, the stable model semantics [10] and the well-founded semantics [17]. While the former is based on guess-and-verify to sort contradictory information into different stable models/answer sets, the latter is defined by a built-in mechanism to circumvent contradictory conclusions, thus making safe inferences in the presence of data that require conflicting interpretations.

More recently, well-founded semantics have been studied for two extensions of logic programming: *logic programs with aggregates* (or, *aggregate programs*) [5, 13, 14] and *description logic programs* (*dl-programs*) [8]. The former brings into logic programming reasoning with constraints, while the latter is an example of logic programming with *external atoms* [7]. In a dl-program an atom can be a *dl-atom*, which is a well-designed interface to an underlying description logic knowledge base. In this way, some decidable fragments of first-order logic can be integrated into rule-based non-monotonic reasoning. These extensions substantially widen the application range of logic programming.

* The work was partially supported by Natural Sciences and Engineering Research Council of Canada.

© Jia-Huai You, John Morris, and Yi Bi;
licensed under Creative Commons License ND
Technical Communications of the 28th International Conference on Logic Programming (ICLP'12).
Editors: A. Dovier and V. Santos Costa; pp. 235–246

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

An aggregate is a constraint, which is a relation on a domain where the tuples in the relation are called *admissible solutions*. A dl-atom can also be viewed as a constraint, in terms of the sets of (ordinary) atoms under which it is satisfied. Despite this close connection, the semantics for these two kinds of programs have been studied independently. In [8], the semantics is defined under the first principle of unfounded sets, while in the work of Pelov et al. [14], a purely algebraic framework is developed under the theory of approximating operators on bilattices [4], parameterized by approximating operators and aggregate relations. In particular, given an aggregate program Π, the well-founded semantics, based on the least fixpoint of a 3-valued immediate consequence operator Φ_{Π}^{aggr}, is defined along with the *ultimate aggregate relation*. Let us call this semantics the (ultimate) well-founded semantics of Π. It extends the well-founded semantics for normal logic programs.

In this paper, we study the relationships between dl-programs and aggregate programs under the well-founded approach. The main result is that the well-founded semantics of dl-programs can be obtained from the ultimate well-founded semantics of aggregate programs, under a mapping from dl-atoms to aggregates. This leads to the following conclusions: on the one hand, the well-founded semantics for dl-programs can be viewed as a special case of the ultimate well-founded semantics for aggregate programs, and on the other hand, the latter semantics can be defined, alternatively, employing the notion of unfounded sets.[1] As a result, the well-founded semantics can be defined, in a uniform manner using the first principle of unfoundedness, for logic programs that may contain both dl-atoms and aggregates.

The paper is organized as follows. The next section provides some definitions. Section 3 introduces the well-founded semantics for dl-programs. Section 4 shows that under a mapping from dl-atoms to aggregates, the well-founded semantics for dl-programs is precisely that of the corresponding aggregate programs. Then in Section 5 we extend the well-founded semantics to logic programs that may contain dl-atoms as well as aggregates. Section 6 is about related work followed by comments on future work.

2 Preliminaries

We introduce dl-programs. Although technically this paper does not intimately depend on description logics (DLs) [1], some familiarity would be convenient.

A *DL knowledge base* L consists of a finite set of axioms built over a vocabulary $\Sigma_L = (\mathbf{A} \cup \mathbf{R}, \mathbf{I})$, where $\mathbf{A}$, $\mathbf{R}$ and $\mathbf{I}$ are pairwise disjoint (denumerable) sets of *atomic concepts, atomic roles* and *individuals*, respectively. As usual, concepts can be built from atomic concepts and other constructs, such as $\sqcap$ (conjunction), $\sqcup$ (disjunction), $\neg$ (negation), and various restrictions (see [1] for more details).

Let $\mathbf{P}$ be a finite set of *predicate* symbols and $\mathbf{C}$ a nonempty finite set of *constants* such that $\mathbf{P} \cap (\mathbf{A} \cup \mathbf{R}) = \emptyset$ and $\mathbf{C} \subseteq \mathbf{I}$. A *term* is either a constant from $\mathbf{C}$ or a *variable*. An atom is of the form $p(t_1, ..., t_m)$, where p is a predicate symbol from $\mathbf{P}$, and t_i is a term. An equality (resp. inequality) is of the form $t_1 = t_2$ (resp. $t_1 \neq t_2$), where t_1 and t_2 are terms. A *dl-query* is of the form $Q(\mathbf{t})$, where $\mathbf{t}$ is a list of terms, and Q is an equality/inequality symbol, or a concept, a role, or a concept inclusion axiom, built from $\mathbf{A} \cup \mathbf{R}$.

A *dl-atom* is of the form $DL[S_1 op_1 p_1, \cdots, S_m op_m p_m; Q](\mathbf{t})$, where S_i is a concept or role built from $\mathbf{A} \cup \mathbf{R}$, or an equality/inequality symbol; $op_i \in \{\uplus, \cupdot, \cap\!\!\!\!-\}$; $p_i \in \mathbf{P}$ is a unary

[1] In fact, such a definition of unfounded sets was already proposed in [9].

predicate symbol if S_i is a concept, and a binary predicate symbol otherwise; and $Q(\mathbf{t})$ is a dl-query.

A *dl-rule* (or *rule*) is of the form $h \leftarrow A_1, ..., A_m, not\ B_1, ..., not\ B_n$, where h is an atom, and A_i and B_i are atoms or equalities/inequalities or dl-atoms. An atom or a dl-atom A, and its negated form $not\ A$, is called a *literal*. For any rule r, we denote the head of the rule by $H(r)$, and the body by $B(r)$. In addition, $B^+ = \{A_1, ..., A_m\}$ and $B^- = \{B_1, ..., B_n\}$. A *rule base* P is a finite set of rules.

A *dl-program* is a combined knowledge base $KB = (L, P)$, where L is a DL knowledge base and P is a rule base.

A *ground instance* of a rule r is obtained by first replacing every variable in r with a constant from $\mathbf{C}$, then replacing with $\top$ (resp. $\bot$) every equality/inequality if it is valid (resp. invalid) under the unique name assumption (UNA). $\top$ and $\bot$ are two special predicates such that $\top$ (resp. $\bot$) is true (resp. false) in every interpretation.

In this paper, we assume a rule base P is already grounded using the constants appearing in the given non-ground program. Likewise, when we refer to an atom/dl-atom/literal, by default we mean it is one without variables.

The *Herbrand base* of a rule base P, denoted by HB_P, is the set of all ground atoms $p(t_1, ..., t_m)$, where p is from $\mathbf{P}$ and t_i is a constant from $\mathbf{C}$, both occurring in P. Any subset of HB_P is an *interpretation* of P.

▶ **Definition 1.** *Let $KB = (L, P)$ be a dl-program and $I \subseteq HB_P$ an interpretation. Define the* satisfaction relation under L*, denoted $\models_L$, as follows:*

1. $I \models_L \top$ *and* $I \not\models_L \bot$.
2. *For any atom $a \in HB_P$, $I \models_L a$ if $a \in I$.*
3. *For any (ground) dl-atom $A = DL[S_1 op_1 p_1, \cdots, S_m op_m p_m; Q](\mathbf{c})$ occurring in P, $I \models_L A$ if $L \cup \bigcup_{i=1}^m A_i \models Q(\mathbf{c})$, where*
$$A_i = \begin{cases} \{S_i(\mathbf{e}) \mid p_i(\mathbf{e}) \in I\}, & \text{if } op_i = \uplus\,; \\ \{\neg S_i(\mathbf{e}) \mid p_i(\mathbf{e}) \in I\}, & \text{if } op_i = \,\cup\!\!\!\!-\,; \\ \{\neg S_i(\mathbf{e}) \mid p_i(\mathbf{e}) \notin I\}, & \text{if } op_i = \,\cap\!\!\!\!-\,. \end{cases}$$
4. *For any ground atom or dl-atom A, $I \models_L not\ A$ if $I \not\models_L A$.*

The above satisfaction relation naturally extends to conjunctions of literals. For a rule $r \in P$, $I \models_L r$ if $I \not\models_L B(r)$ or $I \models_L H(r)$. I is a model of a dl-program $KB = (L, P)$ if $I \models_L r$ for all $r \in P$.

A ground dl-atom A is *monotonic* relative to KB if for any $I \subseteq J \subseteq HB_P$, $I \models_L A$ implies $J \models_L A$. Otherwise, A is *nonmonotonic*.

Additional notations: Given a set S of atoms, $\neg.S = \{\neg a \mid a \in S\}$; given a set P of rules, $Lit_P = HB_P \cup \neg.HB_P$; if I is a set of literals, $I^+ = \{a \mid a \text{ is an atom in } I\}$ and $I^- = \{a \mid \neg a \in I\}$; a set of literals $I \subseteq Lit_P$ is consistent if there is no atom a such that $a \in I$ and $\neg a \in I$. In this paper, by an *interval* $[S_1, S_2]$, where S_1 and S_2 are sets and $S_1 \subseteq S_2$, we mean the set $\{S \mid S_1 \subseteq S \subseteq S_2\}$.

3 Well-Founded Semantics for Arbitrary DL-Programs

The well-founded semantics is first defined for dl-programs with dl-atoms that may only contain operators $\uplus$ and $\cup\!\!\!\!-$ [8]. These dl-atoms are monotonic. It is then commented (see Section 9.2 of [8]) that the definition can be generalized to the class of all dl-programs. For contrast, here we introduce the well-founded semantics for arbitrary dl-programs directly.

▶ **Definition 2. (Unfounded set)** Let $KB = (L, P)$ be a dl-program and $I \subseteq Lit_P$ be consistent. A set $U \subseteq HB_P$ is an *unfounded set of* KB *relative to* I iff the following holds:

For every $a \in U$ and every rule $r \in P$ with $H(r) = a$, either (i) $\neg b \in I \cup \neg.U$ for some ordinary atom $b \in B^+(r)$, or (ii) $b \in I$ for some ordinary atom $b \in B^-(r)$, or (iii) for some $b \in B^+(r)$, it holds that $S^+ \not\models_L b$ for each consistent $S \subseteq Lit_P$ with $I \cup \neg.U \subseteq S$, or (iv) for some $b \in B^-(r)$, it holds that $S^+ \models_L b$ for each consistent $S \subseteq Lit_P$ with $I \cup \neg.U \subseteq S$.

Intuitively, the definition says that an atom a is in an unfounded set U, relative to I, because, for every rule with a in the head, at least one body literal is not satisfied by I under L, and this fact remains to hold for any consistent extension of $I \cup \neg.U$.

▶ **Definition 3.** Let $KB = (L, P)$ be a dl-program. We define the operators T_{KB}, U_{KB}, and W_{KB} on all consistent $I \subseteq Lit_P$ as follows:

(i) $a \in T_{KB}(I)$ iff $a \in HB_P$ and some $r \in P$ exists such that (a) $H(r) = a$, (b) for all $b \in B^+(r)$, $S^+ \models_L b$ for each consistent S with $I \subseteq S \subseteq Lit_P$, (c) $\neg b \in I$ for all ordinary atoms $b \in B^-(r)$, and (d) for all $b \in B^-(r)$, $S^+ \not\models_L b$ for each consistent S with $I \subseteq S \subseteq Lit_P$.
(ii) $U_{KB}(I)$ is the greatest unfounded set of KB relative to I; and
(iii) $W_{KB}(I) = T_{KB}(I) \cup \neg.U_{KB}(I)$.

With the standard definition of monotonicity of operators over complete lattices, one can verify easily that the operators T_{KB}, U_{KB}, and W_{KB} are all monotonic.

As a notation, we define $W^0_{KB} = \emptyset$, and $W^{i+1}_{KB} = W_{KB}(W^i_{KB})$, for all $i \geq 0$. In the sequel, we will use a similar notion for other monotonic operators, but sometimes we may start applying such an operator from a nonempty set (this will be made clear when such a situation arises).

▶ **Definition 4. (Well-founded Semantics)** Let $KB = (L, P)$ be a dl-program. The *well-founded semantics* of KB, denoted by $WFS(KB)$, is defined as the least fixpoint of the operator W_{KB}, denoted $lfp(W_{KB})$. An atom $a \in HB_P$ is *well-founded* (resp. *unfounded*) relative to KB iff a (resp. $\neg a$) is in $lfp(W_{KB})$.

▶ **Example 5.** Consider a dl-program $KB = (\emptyset, P)$, where P consists of

$$
\begin{array}{ll}
r_1: & p(a) \leftarrow not\ DL[S_1 \cap\!\!\!\!- \, q, S_2 \uplus r; \neg S_1 \sqcap S_2](a). \\
r_2: & q(a) \leftarrow DL[S \uplus q; S](a). \\
r_3: & r(a) \leftarrow DL[S \cap\!\!\!\!- \, q; \neg S](a).
\end{array}
$$

Starting with $W^0_{KB} = \emptyset$, for example, we do not derive $p(a)$ since there is a consistent extension that satisfies the dl-atom in rule r_1, but $\{q(a)\}$ is an unfounded set relative to $\emptyset$. The reader can verify that $W^1_{KB} = \{\neg q(a)\}$, $W^2_{KB} = \{\neg q(a), r(a)\}$, and $W^3_{KB} = \{\neg q(a), \neg p(a), r(a)\}$, which is the least fixpoint of W_{KB}.

We now discuss an alternative way to construct the least fixpoint of W_{KB}. The technical result given here will be used later when relating to the ultimate well-founded semantics for aggregate programs.

Since the operator T_{KB} only generates positive atoms, given a consistent $I \subseteq Lit_P$, we can apply T_{KB} iteratively, with I^- fixed. That is,

$$T^0_{KB} = I^+, T^1_{KB} = T_{KB}(T^0_{KB} \cup \neg.I^-), ..., T^{k+1}_{KB} = T_{KB}(T^k_{KB} \cup \neg.I^-), ... \quad (1)$$

Since this sequence is $\subseteq$-increasing, a fixpoint exists. Let us denote it by $FP_{T_{KB}}(I)$. Note that the operator $FP_{T_{KB}} : Lit_P \to HB_P$ is monotonic relative to a fixed I^-. Namely, for any consistent sets of literals I_1 and I_2 such that $I_1^- = I_2^-$ and $I_1 \subseteq I_2$, we have $FP_{T_{KB}}(I_1) \subseteq FP_{T_{KB}}(I_2)$.

Now, following Definition 3, we define an operator V_{KB}, which is similar to W_{KB}, as follows: Given a consistent set of literals $I \subseteq Lit_P$,

$$V_{KB}(I) = FP_{T_{KB}}(I) \cup \neg.U_{KB}(I) \tag{2}$$

As the operator V_{KB} is monotonic, its least fixpoint exists, which we denote by $lfp(V_{KB})$. We can show that (the proof is omitted for lack of space)

▶ **Lemma 6.** $lfp(V_{KB}) = lfp(W_{KB})$.

4 Representing DL-Programs by Aggregate Programs

In general, an aggregate in a logic program is a constraint atom. Since in this paper our interest is in the semantics, we assume that an aggregate is a constraint whose semantics is pre-defined in terms of its domain and admissible solutions. An explicit representation of such constraints has been called *abstract constraint atoms* (or just *c-atoms*) [12].

We assume a propositional language, $\mathcal{L}_\Sigma$, determined by a fixed countable set Σ of propositional atoms. A *c-atom* A is a pair (D, C), where D is a nonempty finite set of atoms in Σ and $C \subseteq 2^D$. We use A_d and A_c to refer to the components D and C of A, respectively. As an abstraction, a c-atom A can be used to represent the semantics of any constraint with a set A_c of admissible solutions over a finite domain A_d [11, 12]. Therefore, in the sequel we will use the aggregate notation and c-atoms exchangeably.

The *complement* of a c-atom A is the c-atom A' with $A'_d = A_d$ and $A'_c = 2^{A_d} \setminus A_c$.

An interpretation $I \subset \Sigma$ *satisfies* an atom a if $a \in I$; $\neg a$ if $a \notin I$. I satisfies a c-atom A, written as $I \models A$, if $A_d \cap I \in A_c$; *not* A, written $I \models not\ A$, if $A_d \cap I \notin A_c$. Therefore, it follows that I satisfies *not* A iff I satisfies the complement of A. I satisfies a conjunction E of atoms or c-atoms, written $I \models E$, if I satisfies every conjunct in it.

A c-atom A is *monotone* if for any $J \supseteq I$, that I satisfies A implies J satisfies A. Otherwise, A is nonmonotone.

An *aggregate program* (or exchangeably, a *logic program with c-atoms*) is a finite set of rules of the form $h \leftarrow B_1, ..., B_n, not\ C_1, ..., not\ C_k$, where h, B_i, and C_i are ordinary atoms or c-atoms. Given a rule r, we use $H(r)$ to denote the head and $B(r)$ to denote the body.

Note that in [14] negative aggregates $\neg C$ are allowed, but here we write them as *not* C.

The notations Lit_Π, S^+, and S^- (given a set of literals S) are defined similarly as for those used for dl-programs.

▶ **Definition 7. (From dl-programs to aggregate programs)** Given a dl-program $KB = (L, P)$, we obtain an aggregate program, denoted $\beta(KB)$, by a mapping β_{KB} from atoms, dl-atoms, and their default negation occurring in P to aggregates as follows:

- If A is an ordinary atom a then $\beta_{KB}(A) = a$, and
- If A is a dl-atom then $\beta_{KB}(A) = (HB_P, C)$, where $C = \{I \subseteq HB_P \mid I \models_L A\}$.
- For any default negation of the form *not* A, $\beta_{KB}(not\ A) = \beta_{KB}(A)'$.

In the sequel, as the underlying KB is always clear, we will drop the subscript in β_{KB}. Also, by abuse of notation, given a rule $r \in P$, we denote by $\beta(B(r))$ the translated conjunction in the body of r, and by $\beta(r)$ the translated rule. Since this mapping does not

introduce new symbols, given a dl-program $KB = (L, P)$, we can identify Σ for the translated aggregate programs with HB_P.

As an example, consider a dl-program $KB = (\emptyset, \{p(a) \leftarrow DL[S \cap\!\!\!\!- p; \neg S](a)\})$. The translated aggregate program consists of a single rule, $p(a) \leftarrow (\{p(a)\}, \{\emptyset\})$. The c-atom in the rule represents the semantics of, e.g., the cardinality constraint, $card_{=}(\{x | p(x)\}, 0)$.

▶ **Lemma 8.** *Let $KB = (L, P)$ be a dl-program and $I \subseteq HB_P$.*
(i) *For any dl-atom A, $I \models_L A$ iff $I \models \beta(A)$, and $I \not\models_L A$ iff $I \models \beta(A)'$.*
(ii) *For any rule $r \in P$, I satisfies r iff I satisfies $\beta(r)$.*
(iii) *I is a model of KB iff I is a model of $\beta(KB)$.*

4.1 Relationship

Following [13, 14], given a complete lattice $\langle L, \preceq \rangle$, the *bilattice* induced from it is the structure $\langle L^2, \preceq, \preceq_p \rangle$, where for all $x, y, x', y' \in L$,

$$(x, y) \preceq (x', y') \quad \text{if and only if} \quad x \preceq x' \text{ and } y \preceq y'$$
$$(x, y) \preceq_p (x', y') \quad \text{if and only if} \quad x \preceq x' \text{ and } y' \preceq y$$

The order $\preceq$ on L^2 is called the *produce order*, while $\preceq_p$ is called the *precision order*. Both orders are complete lattice orders on L^2. We are interested only in the subset of pairs (x, y) that are *consistent*, i.e., $x \preceq y$, and when $x = y$ it is said to be *exact*. We denote the set of consistent pairs by L^c.

Given a bilattice $\langle L^2, \preceq, \preceq_p \rangle$, the $\preceq$-least element is $(\bot, \bot)$ and the $\preceq_p$-least element is $(\bot, \top)$. E.g., consider the complete lattice $\langle 2^\Sigma, \subseteq \rangle$ where Σ is a set of atoms. For the bilattice induced from it, the $\preceq$-least element is $(\emptyset, \emptyset)$ and the $\preceq_p$-least element is $(\emptyset, \Sigma)$.

The idea of the well-founded semantics for an aggregate program is to start with the $\preceq_p$-least element $\langle \emptyset, \Sigma \rangle$, and apply an *approximating operator*, denoted A, in a way that not only are the true atoms computed, but also the false atoms that are not reachable by derivations. It approximates an operator O on L, whose fixpoints are exact pairs on L^2.

▶ **Definition 9.** Let $O : L \to L$ be an operator on a complete lattice $\langle L, \preceq \rangle$. We say that $A : L^c \to L^c$ is an *approximating operator* of O iff the following conditions are satisfied:
- A extends O, i.e., $A(x, x) = (O(x), O(x))$, for every $x \in L$.
- A is $\preceq_p$-monotone.

The condition on A is a mild one: it only requires to extend O on exact pairs, in addition to monotonicity.

For aggregate programs, given a language $\mathcal{L}_\Sigma$, a program Π, and a monotonic approximating operator A of some operator O, we will compute a sequence

$$(\emptyset, \Sigma) = (u_0, v_0), (u_1, v_1), ..., (u_k, v_k), ..., (u_\infty, v_\infty) \tag{3}$$

such that the internal $[u_i, v_i]$ is decreasing, i.e., $[u_{i+1}, v_{i+1}] \subset [u_i, v_i]$, for all i, eventually reaching a fixpoint, which is denoted by (u_∞, v_∞).

Intuitively, one can think of this sequence as representing the process that, initially nothing is *known to be true* and every atom in Σ is *potentially true* (as such, nothing is *known to be false*); and given (u_i, v_i), after the current iteration, $u_{i+1} \setminus u_i$ is the set of atoms that become known to be true and $v_i \setminus v_{i+1}$ is the set of atoms that become known to be false. In this way, u_i represents a *lower estimate* and v_i an *upper estimate* of the eventual fixpoint. At the end, u_∞ is the set of true atoms and $\Sigma \setminus v_\infty$ is the set of false atoms, and the truth value of the remaining atoms is *undefined*. This gives a 3-valued interpretation of the least fixpoint.

A fixpoint operator that constructs sequence (3) can be defined in different ways. For example, we can simply take the approximating operator A, i.e., $A(u_i, v_i) = (u_{i+1}, v_{i+1})$, for all i. If A is $\preceq_p$-monotone, then the least fixpoint of A exists, which is called the *Kripke-Kleene fixpoint* of A.

As alluded to earlier, our interest is the well-founded semantics, which is determined by the so-called *well-founded fixpoint* of A. It is computed by a *stable revision operator*, denoted by $\mathcal{St}_\Pi$, for a given aggregate program Π. Namely, $\mathcal{St}_\Pi(u_i, v_i) = (u_{i+1}, v_{i+1})$, where (u_{i+1}, v_{i+1}) is computed from (u_i, v_i) using two component operators of A. The first one, denoted by $A^1(\cdot, v_i)$, is A with v_i fixed, and similarly, the second, denoted by $A^2(u_i, \cdot)$, is A with u_i fixed. Given an upper estimate b, we compute a new lower estimate by an iterative process:

$$x_0 = \bot, x_1 = A^1(x_0, b), ..., x_{i+1} = A^1(x_i, b), ... \tag{4}$$

until a fixpoint is reached. That is, if $b = v_i$, then $u_{i+1} = x_\infty$ where $x_\infty = A^1(x_\infty, b)$. The operator that generates x_∞ is called the *lower revision operator*.

On the other hand, given a lower estimate a, we compute a new upper estimate

$$y_0 = a, y_1 = A^2(a, y_0), ..., y_{i+1} = A^2(a, y_i), ... \tag{5}$$

until a fixpoint is reached. That is, if $a = u_i$, then $v_{i+1} = y_\infty$ where $y_\infty = A^2(a, y_\infty)$. The operator that generates y_∞ is called the *upper revision operator*.

It can be seen that if A is $\preceq_p$-monotone, so is $\mathcal{St}_\Pi$, thus the least fixpoint of $\mathcal{St}_\Pi$ can be constructed by a sequence (3), where (u_∞, v_∞) is called the *well-founded fixpoint* of A, which is the least fixpoint of the stable revision operator $\mathcal{St}_\Pi$.

By this parameterized algebraic approach one can define possibly different well-founded semantics by employing different $\preceq_p$-monotone approximating operators. In the context of aggregate programs, the operator we are approximating is the standard immediate consequence operator extended to aggregate programs Π, i.e., $\mathcal{T}_\Pi : \Sigma \to \Sigma$, where

$$\mathcal{T}_\Pi(I) = \{H(r) \mid r \in \Pi \text{ and } I \models B(r)\}. \tag{6}$$

To approximate $\mathcal{T}_\Pi$ while preserving the well-founded and stable model semantics for normal logic programs, in [13], a three-valued immediate consequence operator Φ_Π^{aggr} is defined for aggregate programs, which maps 3-valued interpretations to 3-valued interpretations. Recall that a 3-valued interpretation can be represented by a pair (I_1, I_2) of 2-valued interpretations with $I_1 \subseteq I_2$, where I_1 is the set of atoms assigned to *true*, $\Sigma \backslash I_2$ is the set of atoms assigned to *false*, and all the other atoms are assigned to *undefined*. Thus, Φ_Π^{aggr} maps a pair of 2-valued interpretations to a pair of 2-valued interpretations, i.e., $\Phi_\Pi^{aggr}(I_1, I_2) = (I'_1, I'_2)$. The definition of Φ_Π^{aggr} guarantees that it approximates the operator $\mathcal{T}_\Pi$, in that for any fixpoint (I, J) of Φ_Π^{aggr}, and for any x such that $\mathcal{T}_\Pi(x) = x$, $(I, J) \preceq_p (x, x)$.

From the definition of Φ_Π^{aggr} above, two component operators are induced. They are

$$\Phi_\Pi^{aggr,1}(I_1, I_2) = I'_1 \quad \text{and} \quad \Phi_\Pi^{aggr,2}(I_1, I_2) = I'_2 \tag{7}$$

The original definition of Φ_Π^{aggr} is given in 3-valued logic, parameterized by the choice of *approximating aggregates* [13]. In [16], the authors showed an equivalent definition of $\Phi_\Pi^{aggr,1}$ in terms of *conditional satisfaction*, when the approximating aggregate used is the *ultimate approximating aggregate*. We state this result below (see Appendix of [16]). Here, we replace aggregates with c-atoms. In a similar way, an equivalent definition of $\Phi_\Pi^{aggr,2}$ can be obtained.

▶ **Theorem 10.** *Let* Π *be an aggregate program, and* I *and* M *interpretations with* $I \subseteq M \subseteq \Sigma$*. Then,*

$$\Phi_{\Pi}^{aggr,1}(I,M) = \{H(r) \mid r \in \Pi, \forall J \in [I,M], J \models B(r)\} \quad (8)$$
$$\Phi_{\Pi}^{aggr,2}(I,M) = \{H(r) \mid r \in \Pi, \exists J \in [I,M], J \models B(r)\} \quad (9)$$

▶ **Lemma 11.** *The component operators* $\Phi_{\Pi}^{aggr,1}(\cdot,b)$ *and* $\Phi_{\Pi}^{aggr,2}(a,\cdot)$ *are* $\subseteq$*-monotone, and* Φ_{Π}^{aggr} *is* $\subseteq_p$*-monotone.*

Therefore, the stable revision operator $\mathcal{S}t_{\Pi}$ induced from Φ_{Π}^{aggr} is also $\subseteq_p$-monotone, and we take the least fixpoint of this operator for the semantics. Recall that this fixpoint has been referred to as the well-founded fixpoint of Φ_{Π}^{aggr}.

▶ **Definition 12.** Let Π be an aggregate program and (u_∞, v_∞) the well-founded fixpoint of Φ_{Π}^{aggr}. The *ultimate well-founded semantics* of Π based on Φ_{Π}^{aggr}, denoted by $UWFS(\Pi)$, is defined as $u_\infty \cup \neg.(\Sigma \setminus v_\infty)$.

In the sequel, we will drop the phrase "based on Φ_{Π}^{aggr}", with the understanding that the underlying approximating operator is Φ_{Π}^{aggr} as identified in Theorem 10.

▶ **Example 13.** Consider the following aggregate program Π:

$$p(-1). \qquad\qquad p(-2) \leftarrow sum_{\leq}(\{x \mid p(x)\}, 2).$$
$$p(3) \leftarrow sum_{>}(\{x \mid p(x)\}, -4). \qquad\qquad p(-4) \leftarrow sum_{\leq}(\{x \mid p(x)\}, 0).$$

The aggregates under *sum* are self-explaining, e.g., $sum_{\leq}(\{x \mid p(x)\}, 2)$ means that the sum of x for satisfied atoms $p(x)$ is less than or equal to 2. For the construction of the well-founded fixpoint, we start with the pair $(\emptyset, \Sigma)$. The reader can apply equations in (7) to verify: by applying the operator $\Phi_{\Pi}^{aggr,1}(\cdot,\Sigma)$ iteratively, we get a new lower estimate $Q = \{p(-1), p(-2), p(-4)\}$; and by applying $\Phi_{\Pi}^{aggr,2}(\emptyset,\cdot)$ iteratively, we get an upper estimate Σ, which is the same as before. Thus the new pair is (Q,Σ). Continuing in the next iteration, Q remains the same but $p(3)$ is no longer derivable. We thus have $(Q, \Sigma - \{p(3)\})$, which is a fixpoint. So the ultimate well-founded semantics is that all atoms in Q are true, $p(3)$ is false, and nothing is undefined.

▶ **Theorem 14.** *Let* $KB = (L,P)$ *be a dl-program. The well-founded semantics of* KB *coincides with the ultimate well-founded semantics of the aggregate program* $\beta(KB)$*. That is,* $WFS(KB) = UWFS(\beta(KB))$*.*

Proof. (Sketch) Let $\Pi = \beta(KB)$ and (u_∞, v_∞) in sequence (3) be the ultimate well-founded fixpoint of Φ_{Π}^{aggr}. Recall that $WFS(KB) = lfp(W_{KB}) = lfp(V_{KB})$ (the latter is by Lemma 6) and $UWFS(\Pi) = u_\infty \cup \neg.(\Sigma \setminus v_\infty)$. We prove the coincidence by induction on the sequences of constructing $lfp(V_{KB})$ and (u_∞, v_∞). In the proof, we assume rules in P are of the form $a \leftarrow \phi$ or $a \leftarrow not\ \phi$, where ϕ is a dl-atom. The proof can be generalized to arbitrary dl-rules. Below, we identify Σ for the corresponding aggregate program with HB_P for the given dl-program, i.e., we let $\Sigma = HB_P$.

Clearly, $V_{KB}^0 = u_0 \cup \neg.(\Sigma \setminus v_0) = \emptyset$. Assume $(V_{KB}^i)^+ = u_i$ and $(V_{KB}^i)^- = \Sigma \setminus v_i$ and we show $(V_{KB}^{i+1})^+ = u_{i+1}$ and $(V_{KB}^{i+1})^- = \Sigma \setminus v_{i+1}$, for all $i \geq 0$. Note that from (1), and by induction hypothesis, we have

$$(V_{KB}^{i+1})^+ = FP_{T_{KB}}(V_{KB}^i) = FP_{T_{KB}}(u_i \cup \neg.(\Sigma \setminus v_i)) \quad (10)$$

(a) $(V_{KB}^{i+1})^+ = u_{i+1}$. First, observe that for the approximating operator Φ_Π^{aggr}, x_∞ in (4) can be computed equivalently by starting with u_i, i.e.,

$$x_0 = u_i, x_1 = \Phi_\Pi^{aggr,1}(x_0, v_i), ..., x_{i+1} = \Phi_\Pi^{aggr,1}(x_i, v_i), ..., x_\infty = \Phi_\Pi^{aggr,1}(x_\infty, v_i) \quad (11)$$

and $u_{i+1} = x_\infty$. According to (10), we need to show $FP_{T_{KB}}(u_i \cup \neg.(\Sigma \setminus v_i)) = u_{i+1}$. We prove this by showing a one-one correspondence between the steps in (1) and those in (11). That is, $x_k = T_{KB}^k$ for all $k \geq 0$. The base case is due to the induction hypothesis, namely $x_0 = u_i = (V_{KB}^i)^+ = T_{KB}^0$ (note that T_{KB}^0 here refers to the one in (1)). Assume $x_k = T_{KB}^k$ and we show $x_{k+1} = T_{KB}^{k+1}$, for all $k \geq 0$. For any atom $a \in \Sigma$, $a \in x_{k+1}$ iff for some rule $r \in P$ with $H(r) = a$ such that for every $J \in [x_k, v_i]$, $J \models \beta(B(r))$). Let us label the last statement as (C1).

Suppose $r = a \leftarrow \phi$. By Lemma 8, $J \models \beta(\phi)$ iff $J \models_L \phi$. From the induction hypothesis, we have $(V_{KB}^i)^- = \Sigma \setminus v_i$, and it follows

$$[x_k, v_i] = \{S^+ \mid S \text{ is consistent and } x_k \cup \neg.\Sigma \setminus v_i \subseteq S \subseteq Lit_P\}$$

From $x_k = T_{KB}^k$, it follows that (C1) iff $a \in T_{KB}^{k+1}$, as the condition (b) of Definition 3 is satisfed: for all $b \in B^+(r)$, $S^+ \models_L b$ for each consistent S with $T_{KB}^k \subseteq S \subseteq Lit_P$. The case where $r = a \leftarrow not\ \phi$ can be proved similarly, based on condition (d) of Definition 3.

(b) $(V_{KB}^{i+1})^- = \Sigma \setminus v_{i+1}$. Namely, $U_{KB}(V_{KB}^i) = \Sigma \setminus v_{i+1}$, i.e., the greatest unfounded set of KB relative to V_{KB}^i is precisely the fixpoint y_∞ ($= v_{i+1}$) below:

$$y_0 = u_i, y_1 = \Phi_\Pi^{aggr,2}(u_i, y_0), ..., y_{i+1} = \Phi_\Pi^{aggr,2}(u_i, y_i), ..., y_\infty = \Phi_\Pi^{aggr,2}(u_i, y_\infty) \quad (12)$$

(b-1) Prove that for any $a \in \Sigma$, if $a \in v_{i+1}$ then $a \notin U$, for any unfounded set U of KB relative to V_{KB}^i. By definition, $a \in v_{i+1}$ iff $a \in y_k$, for some $k \geq 0$, iff there is a rule $r \in P$ with $H(r) = a$ such that $\exists J \in [u_i, y_k]$, $J \models \beta(B(r))$. By Lemma 8, $J \models \beta(B(r))$ iff $J \models_L \phi$, if r is of form $a \leftarrow \phi$. This violates condition (iii) in Definition 2, as by induction hypothesis there is a consistent extension S of V_{KB}^i such that $S^+ = J$. The proof is similar if r is of form $a \leftarrow not\ \phi$, in which case condition (iv) is violated.

(b-2) Show that $a \notin v_{i+1} \Rightarrow a \in U_{KB}(V_{KB}^i)$, for all $a \in \Sigma$. That $a \notin v_{i+1}$ (i.e., $a \notin y_\infty$) means, for every rule $r \in P$ with $H(r) = a$, and for all $I \in [u_i, y_\infty]$, $I \not\models \beta(B(r))$, hence by Lemma 8, $I \not\models_L \phi$ if $r = a \leftarrow \phi$ and $I \models_L \phi$ if $r = a \leftarrow not\ \phi$. Note that

$$[u_i, y_\infty] = \{S^+ \mid S \text{ is consistent and } u_i \cup \neg.\Sigma \setminus y_\infty \subseteq S \subseteq Lit_P\}$$

From the induction hypothesis we know $u_i = (V_{KB}^i)^+$ and $(V_{KB}^i)^- = \Sigma \setminus v_i$, and notationally $v_{i+1} = y_\infty$. It follows from Definition 2 that $\Sigma \setminus v_{i+1}$ is an unfounded set of KB relative to V_{KB}^i, and $a \in \Sigma \setminus v_{i+1}$. Obviously, it is the greatest unfounded set of KB relative to V_{KB}^i, since for any atom $\varphi \in y_\infty$, there is a derivation of φ based on V_{KB}^i. Therefore, $a \in U_{KB}(V_{KB}^i)$. The proof is completed. ◀

5 Well-Founded Semantics of DL-Programs with Aggregates

A dl-program with aggregates is a combined knowledge base $KB = (L, P)$, where L is a DL knowledge base and P a finite set of rules of the form $a \leftarrow b_1, ..., b_k, \texttt{not}\ c_1, ..., \texttt{not}\ c_n$, where a is an atom, and each b_i or c_j is either an ordinary atom, a dl-atom, or an aggregate atom. In the following, we continue to denote by HB_P the set of atoms composed from the constants and predicate symbols of the underlying language.

Now we extend the satisfaction relation $\models_L$ to cover aggregates. Let $KB = (L, P)$ be a dl-program with aggregates and $I \subseteq HB_P$ an interpretation. For any aggregate ϕ, we define $I \models_L \phi$ iff $I \models \phi$, and extend $\models_L$ naturally to conjunctions of atoms, dl-atoms, aggregates, and their negations. Then, Definitions 2 and 3 can be adopted directly, by replacing "dl-program" with "dl-program with aggregates". To distinguish, let us denote the fixpoint operator W_{KB} in Definition 3 by W'_{KB}.

▶ **Definition 15. (Well-founded semantics for dl-programs with aggregates)** Let $KB = (L, P)$ be a dl-program with aggregates. The *well-founded semantics* of KB is defined as the least fixpoint of the operator W'_{KB}, denoted $lfp(W'_{KB})$.

▶ **Theorem 16.** *Let $KB = (L, P)$ be a dl-program with aggregates. (i) If P contains no aggregates, then $lfp(W'_{KB}) = lfp(W_{KB})$; and (ii) If P contains no dl-atoms, then $lfp(W'_{KB}) = lfp(W'_P) = UWFS(P)$.*

For illustration, we close this section by presenting a dl-program with aggregates.

▶ **Example 17.** Consider $KB = (L, P)$ with $L = \{Vip \sqsubseteq CR\}$, possibly plus some assertions of individuals in the concepts Vip and/or CR, where CR stands for Customer-Record, and P containing

1. $purchase(X) \leftarrow purchase(X, Obj), item(Obj)$.
2. $client(X) \leftarrow DL[CR \uplus purchase; CR](X)$.
3. $imp_client(X) \leftarrow DL[Vip](X)$.
4. $imp_client(X) \leftarrow client(X), sum_{\geq}(\{Y \mid item(Obj), cost(Obj,Y), purchase(X,Obj)\},100)$.
5. $discount(X) \leftarrow imp_client(X)$.
6. $promo_offer(X) \leftarrow DL[CR \uplus imp_client; CR](X), card_{=}(\{Y \mid purchase(Y)\}, 0)$.

Rule 1 is self-explaining. Rule 2 queries the DL knowledge base in order to enhance the *client* predicate. In rules 3 and 4 we establish that important clients are those who have spent at least \$100 or are VIPs. Rules 5 and 6 provide benefits to certain customers. In rule 5, a discount is offered to important clients - VIPs and those whose purchases sum to \$100 or more. Rule 6 describes a promotional offer for non-VIP customers who have not made any purchases - they are potential clients. For applications, P may contain some facts about *items*, *cost*, and *purchase*.

6 Related Work and Further Direction

The close relationships between well-founded model, partial stable models, and stable models are well-understood (see, e.g., [6, 15, 19]). That the well-founded model of a normal logic program is contained in all its stable models makes it possible in a stable model solver to compute the well-founded model as the first approximation. The well-founded semantics has been defined for disjunctive programs [18] and default logic [2]. The close relationship between dl-programs and aggregate programs is noticed in [8], but left as an interesting future direction.

In [9], the notion of unfounded sets for arbitrary aggregate programs is defined (which generalizes that of [3] for logic programs with monotone and anti-monotone aggregates): A set of atoms U is an unfounded set for an aggregate program P and a partial interpretation I, if for every $a \in U$, and for every rule $r \in P$ with a as the head, a literal ξ in $B(r)$ is false w.r.t. I or w.r.t. $(I - U) \cup \neg.U$. The latter expression equals $I \cup \neg.U$ if $I \cap U = \emptyset$. Here, falsity in I amounts to falsify in all of its totalization. It thus gives the same effect as requiring

that ξ is not satisfied by any consistent extension of $I \cup \neg.U$ in our definition. Thus, it follows from our result that, if we define the well-founded semantics for arbitrary aggregate programs using the notion of unfounded sets in [9], the resulting semantics is equivalent to the ultimate well-founded semantics defined by Pelov et al. [13, 14].

The complexity issues for various classes of dl-programs and aggregate programs under the well-founded semantics will be addressed in future work.

References

1. F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation and Applications.* Cambridge University Press, 2003.
2. Gerhard Brewka and Georg Gottlob. Well-founded semantics for default logic. *Fundamenta Informaticae*, 31(3/4):221–236, 1997.
3. Francesco Calimeri, Wolfgang Faber, Nicola Leone, and Simona Perri. Declarative and computational properties of logic programs with aggregates. In *Proc. IJCAI-05*, pages 406–411, 2005.
4. M. Denecker, V. W. Marek, and M. Truszczynski. Ultimate approximation and its application in nonmonotonic knowledge representation systems. *Information and Computation*, 192(1):84–121, 2004.
5. M. Denecker, N. Pelov, and M. Bruynooghe. Ultimate well-founded and stable semantics for logic programs with aggregates. In *Proc. ICLP'01*, pages 212–226, 2001.
6. Phan Minh Dung. On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games. *Artif. Intell.*, 77(2):321–358, 1995.
7. Thomas Eiter, Giovambattista Ianni, Roman Schindlauer, and Hans Tompits. A uniform integration of higher-order reasoning and external evaluations in answer-set programming. In *Proc. IJCAI-05*, pages 90–96, 2005.
8. Thomas Eiter, Thomas Lukasiewicz, Giovambattista Ianni, and Roman Schindlauer. Well-founded semantics for description logic programs in the semantic web. *ACM Transactions on Computational Logic*, 12(2), 2011. Article 3.
9. W. Faber. Unfounded sets for disjunctive logic programs with arbitrary aggregates. In *proc. LPNMR-05*, pages 40–52, 2005.
10. Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *Proc. ICLP'88*, pages 1070–1080, 1988.
11. V. W. Marek and J. B. Remmel. Set constraints in logic programming. In *Proc. LPNMR-04*, pages 167–179, 2004.
12. V. W. Marek and M. Truszczynski. Logic programs with abstract constraint atoms. In *Proceedings of AAAI-04*, pages 86–91, 2004.
13. N. Pelov, M. Denecker, and M. Bruynooghe. Partial stable models for logic programs with aggregates. In *Proc. LPNMR-04*, pages 207–219, 2004.
14. N. Pelov, M. Denecker, and M. Bruynooghe. Well-founded and stable semantics of logic programs with aggregates. *Theory and Practice of Logic Programming*, 7:301–353, 2007.
15. Teodor C. Przymusinski. The well-founded semantics coincides with the three-valued stable semantics. *Fundam. Inform.*, 13(4):445–463, 1990.
16. Tran Cao Son and Enrico Pontelli. A constructive semantic characterization of aggregates in answer set programming. *TPLP*, 7(3), 2007.
17. Allen Van Gelder, Kenneth A. Ross, and John S. Schlipf. The well-founded semantics for general logic programs. *J. ACM*, 38(3):620–650, 1991.

18 Kewen Wang and Lizhu Zhou. Comparisons and computation of well-founded semantics for disjunctive logic programs. *ACM Trans. Comput. Log.*, 6(2):295–327, 2005.

19 Jia-Huai You and Li Yan Yuan. On the equivalence of semantics for normal logic programs. *J. Log. Program.*, 22(3):211–222, 1995.

Preprocessing of Complex Non-Ground Rules in Answer Set Programming*

Michael Morak and Stefan Woltran

Institute of Information Systems 184/2
Vienna University of Technology
Favoritenstrasse 9–11, 1040 Vienna, Austria
E-mail: [surname]@dbai.tuwien.ac.at

Abstract

In this paper we present a novel method for preprocessing complex non-ground rules in answer set programming (ASP). Using a well-known result from the area of conjunctive query evaluation, we apply hypertree decomposition to ASP rules in order to make the structure of rules more explicit to grounders. In particular, the decomposition of rules reduces the number of variables per rule, while on the other hand, additional predicates are required to link the decomposed rules together. As we show in this paper, this technique can reduce the size of the grounding significantly and thus improves the performance of ASP systems in certain cases. Using a prototype implementation and the benchmark suites of the Answer Set Programming Competition 2011, we perform extensive tests of our decomposition approach that clearly show the improvements in grounding time and size.

1998 ACM Subject Classification D.1.6 Logic Programming

Keywords and phrases answer set programming, hypertree decomposition, preprocessing

Digital Object Identifier 10.4230/LIPIcs.ICLP.2012.247

1 Introduction

Starting from the pioneering work of Gelfond and Lifschitz [16, 17], the declarative problem solving paradigm of answer set programming (short: ASP, see e.g. [2]) has become a central formalism in artificial intelligence and knowledge representation. This is due to its simple, yet expressive modelling language, which is implemented by systems showing a steadily increasing performance. Such systems follow a two-step approach for evaluating a program: The so-called *grounder* instantiates rules by replacing the various variables with applicable constants. This yields a propositional logic program (consisting of propositional or "ground" rules) that is equivalent for the given domain. This program is then finally fed into the actual solver. In systems like *lparse/smodels* [23] or *gringo/clasp* [12] this separation is quite strict whereas *DLV* [20] followed a more integrated approach.

Although today's ASP systems have reached an impressive state of sophistication, we believe that there is still room for improvement, in particular on the level of grounding. In fact, since checking whether a non-ground rule fires is already NP-complete [9] in general (as easily shown by analogy to the conjunctive query evaluation problem, which is also NP-complete, cf. [1]), grounders have to list all possibly applicable instantiations of non-ground rules which are, by the NP-completeness of the aforementioned problem, exponentially many in the worst case. However, often the rules exhibit a particular structure which, in theory, could be used to avoid or at least reduce this blow-up. Several preprocessing and optimization techniques

* This work was supported by special fund "Innovative Projekte 9006.09/008" of TU Vienna.

© Michael Morak and Stefan Woltran;
licensed under Creative Commons License ND
Technical Communications of the 28th International Conference on Logic Programming (ICLP'12).
Editors: A. Dovier and V. Santos Costa; pp. 247–258

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

that work well in practice have been developed in the past, see, e.g., [10, 13, 11], but to the best of our knowledge, in the area of ASP, decomposition of rules via hypergraphs has not been implemented or systematically investigated yet.

In this paper we present such a novel preprocessing strategy. It is based on ideas of Gottlob et. al. in [19], who employed a similar mechanism to efficiently solve the boolean conjunctive query evaluation problem. In our approach, each rule is represented as a hypergraph, where each variable in the rule is represented by a vertex and each predicate in the rule is represented by a hyperedge in the hypergraph. Using a hypertree decomposition of this hypergraph representation, the rule can then be split up into an equivalent set of smaller rules, whose grounding is only exponential in the size of the nodes in the hypertree decomposition (i.e., the number of variables in each node). In cases where the size of the nodes is considered to be bound by a fixed constant, the grounding thus remains linear in the size of the non-ground program when using current generation grounders. First experiments with a prototype implementation and the benchmarks from the well-known Third ASP Competition 2011 [7] show a significant decrease both in grounding time and grounding size for certain problems.

2 Preliminaries

In this section we give a brief introduction to Answer Set Programming (ASP) as well as the to the concepts of hypergraphs and hypertree decompositions.

Logic Programs and Answer Set Semantics

We focus here only on the basic definitions; for a comprehensive and recent introduction to answer set programming, see [6].

Disjunctive logic programs are programs that consist of rules of the form

$$H_1 \vee \cdots \vee H_k \leftarrow P_1, \ldots, P_n, \neg N_1, \ldots, \neg N_m$$

where H_i, P_i and N_i are atoms. An *atom* A is a predicate with an arity and accordingly many variables or constant symbols (also called domain elements). If the arity is 0, we simply write A instead of $A()$. Variables are denoted by capital letters, constants by lower-case words. If an atom does not contain variables it is said to be *ground*. For a rule r of above form, we denote by $H(r)$ the set of head atoms of r (i.e. $H(r) = \{H_1, \ldots, H_k\}$); the positive body we denote by $B^+(r) = \{P_1, \ldots, P_n\}$ and the negative body by $B^-(r) = \{N_1, \ldots, N_n\}$. $H_1, \ldots, H_k$ are called the head atoms, and $P_1, \ldots, P_n$ (resp. $N_1, \ldots, N_m$) are called the positive body (resp. negative body) atoms of the rule. Moreover, we use $B(r) = \{P_1, \ldots, P_n, \neg N_1, \ldots, \neg N_n\}$ to denote the set of all *literals* in the body or r. The $\neg$ operator is a unary logical connective, called the *negation as failure* operator or, alternatively, *default negation*. Given a logic program Π, we denote by B_Π its *Herbrand Base*, i.e., the set of all ground atoms which can be constructed from the constants and predicates in Π.

A rule is said to be *safe* if every variable occurring in the head or negative body of the rule also occurs in the positive body of the rule. From this point onward, we only consider logic programs whose rules are safe.

▶ **Example 1.** An example logic program is given below:

$$q \leftarrow E(X, Y), \neg E(X, a)$$

It has the intended meaning that the boolean predicate q is true, if there exists an edge from a vertex X to a vertex Y in a graph, but not from the vertex X to a constant vertex a. ◀

A logic program is said to be *ground*, if it does not contain any rules with variables. A non-ground rule (i.e. one that contains variables) can be seen as an abbreviation for all possible instantiations of the variables with domain elements. In answer set programming, this step is usually explicitly performed by a grounder. Note that such a ground program can be exponential in the size of the non-ground program. In what follows, we denote by $Gr(\Pi)$ the grounding of a program Π. Moreover, we denote by $Gr(r, \Pi)$ the grounding of a single rule r with respect to the domain elements occurring in Π. Clearly, $Gr(\Pi) = \bigcup_{r \in \Pi} Gr(r, \Pi)$.

A set S of ground atoms is a *model* of a disjunctive logic program Π if S satisfies each rule in $Gr(\Pi)$. A ground rule r is satisfied by S if $H(r) \cap S \neq \emptyset$ holds, whenever $B(r)$ is satisfied by S (i.e., whenever $B^+(r) \subseteq S$ and $B^-(r) \cap S = \emptyset$). The *reduct* Π^S of a ground disjunctive logic program Π with respect to a set S of ground atoms is defined as:

$$\Pi^S = \{H(r) \leftarrow B^+(r) \mid r \in \Pi, B^-(r) \cap S = \emptyset\}$$

A set S of ground atoms is an *answer set* of a logic program Π if S is a minimal model of $(Gr(\Pi))^S$, the reduct of the grounding of Π with respect to S.

Hypergraphs and Hypertree Decompositions

Tree decompositions and treewidth, originally defined in [24], are a well known tool to tackle computationally hard problems (see, e.g., [3, 4] for an overview). Treewidth is a measure for the cyclicity of a graph and many NP-complete problems become tractable in cases where the treewidth is bounded. However, many problems are even better represented by hypergraphs. In [18] the concepts of hypertree decompositions and hypertree width were introduced that extend the measurement of cyclicity to hypergraphs.

A *hypergraph* is a pair $H = (V, E)$ with a set V of vertices and a set E of hyperedges. A hyperedge $e \in E$ is itself a set of vertices, with $e \subseteq V$. A *hypergraph of a non-ground logic program rule* r is a pair $HG(r) = (V, E)$ such that V consists of all the variables occurring in r and E is a set of hyperedges, such that for each atom $A \in B(r)$ there exists exactly one hyperedge $e \in E$, which consists of all the variables occurring in A. Furthermore there exists exactly one hyperedge $e \in E$ that contains all the variables occurring in $H(r)$.

The following definition is central for our purposes:

A *(generalized) hypertree decomposition* of a hypergraph $H = (V, E)$ is a triplet $HD = \langle T, \chi, \lambda \rangle$, where $T = (N, F)$ is a (rooted) tree and χ and λ are labelling functions such that for each node $n \in N$, $\chi(n) \subseteq V$ and $\lambda(n) \subseteq E$ and the following conditions hold:

1. for every $e \in E$ there exists a node $n \in N$ such that $e \subseteq \chi(n)$,
2. for every $v \in V$ the set $\{n \in N \mid v \in \chi(n)\}$ induces a connected subtree of T,
3. for every node $n \in N$, $\chi(n) \subseteq \bigcup_{e \in \lambda(n)} e$.

A *hypertree decomposition of a logic program rule* r is therefore a hypertree decomposition of the hypergraph of r. The *width* of a hypertree decomposition is the maximum λ-set size over all its nodes. The minimum width over all possible hypertree decompositions is called the *(generalized) hypertree width*. Similarly, the *treewidth* of a hypertree decomposition is defined by the maximum χ-set size, minus one, of a hypretree decomposition of minimal width.

Unfortunately, for a given hypergraph, it is NP-hard to compute a hypertree decomposition of minimum width. However, efficient heuristics have been developed that offer good approximations (cf. [8, 5]). In practice it turns out that these approximations are often sufficient to obtain good results with decomposition-based algorithms (i.e., algorithms that take the problem and its hypertree decomposition as input).

3 Preprocessing of Non-ground Rules

In this section we describe our main contribution, a novel method for preprocessing complex logic program rules in order to decrease the size of the grounding.

Current grounders for answer set programming do not consider the structure of a rule and thus, when grounding, the number of ground rules produced can in the worst case be exponential in the number of variables occurring in the rule. However, given a hypertree decomposition of such a rule, the exponentiality of the grounding can be restricted to the maximum χ-set size of the decomposition.

In order to describe our algorithm, we introduce the following notational aids: For a node n in a hypertree decomposition, we represent by $\text{parent}(n)$ and $\text{desc}(n)$ the parent node of n and the set of descendants (or child nodes) of n respectively. For a set (or sequence) B of literals and a set $\mathbf{X}$ of variables, we denote with $B \cap \mathbf{X}$ (with some abuse of notation) the literals in B that have at least one of the variables in $\mathbf{X}$ occurring in them. E.g., if $B(r) = E(X_1, X_2), E(X_2, X_3), \neg E(X_3, X_4, c)$, then the intersection $B(r) \cap \{X_1, X_4\} = E(X_1, X_2), \neg E(X_3, X_4, c)$.

Given these shorthands, the rewriting of logic program rules according to our method works by running the following algorithm Preprocess:

1. We compute a (generalized) hypertree decomposition $HD(r) = HD(HG(r)) = \langle T = (N, F), \chi, \lambda \rangle$ of a given logic program rule r, trying to minimize the maximal χ-set size. W.l.o.g. we assume that the edge representing $H(r)$ occurs only in the root node of T.
2. We do a bottom-up traversal of the hypertree decomposition of r. For each node $n \in N$ (except the root) in the decomposition, let $\mathbf{Y_n} = \chi(n) \cap \chi(\text{parent}(n))$ and T_n be a fresh predicate to store the current result. At each node $n \in N$ we generate a rule r_n of the form:

$$
\begin{aligned}
T_n(\mathbf{Y_n}) \quad \leftarrow \quad & (B(r) \cap \chi(n)) \\
& \cup \{\Sigma_X(X) \mid X \in B^-(r) \cap \chi(n)\} \\
& \cup \{T_m(\mathbf{Y_m}) \mid m \in \text{desc}(n)\}
\end{aligned}
$$

   The additional temporary predicates $\Sigma_X(X)$ are necessary to guarantee safety of the generated rule. To this end, for each variable X occurring in $B^-(r) \cap \chi(n)$, we generate a rule

$$\Sigma_X(X) \leftarrow b$$

   where $b \in B^+(r)$ with X as one if its arguments[1].
   For the root node n, we generate a rule similar to r_n but replace $T_n(\mathbf{Y_n})$ by $H(r)$ and we furthermore add all ground atoms of $B(r)$ to this generated rule (since those atoms are not represented in the tree decomposition). We refer to this generated rule as the *head rule*. Generated rules stemming from a leaf node $n \in N$ are referred to as *leaf rules*. Atoms of the from $T_n(\mathbf{Y})$ and $\Sigma_X(X)$ are subsequently called temporary atoms.

▶ **Definition 2.** Given a rule r we denote by r^* the set of rules obtained by running Preprocess on r. Moreover, for a logic program Π and $r \in \Pi$, we define $\Pi_{r^*} = (\Pi \setminus \{r\}) \cup r^*$.

The intuition underlying the Preprocess algorithm is the following: Grounders have to compute all the groundings for every rule in a given logic program. When these rules involve multiple joins, this can be inefficient, because the grounder has to compute all possible tuples

---

[1] We select here such a b from $B^+(r)$ with minimal arity. Note that such a predicate exists since r is safe.

satisfying the first join, and then, for *each* of those, compute all possible tuples satisfying the next join, and so forth.

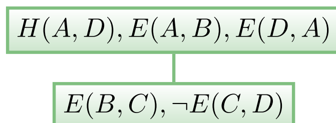
However, the grounder actually only needs to store the values that are involved in the next join, and perform the join operation on them, instead of the complete set of tuples. The Preprocess algorithm makes this explicit: The hypertree decomposition takes care of splitting the rules into multiple parts (i.e., the nodes in the decomposition). By construction of the decomposition, the join operations performed inside a node cannot be split up any further, thus, for each of the nodes we generate a rule performing these joins. However, in the temporary head predicate we then only store the variables that are actually involved in a join in the next node, thereby allowing the grounder to ignore the other variables for any subsequent joins.

The following brief example shows this behaviour:

▶ **Example 3.** Given the rule

$$r = H(A, D) \leftarrow E(A, B), E(B, C), \neg E(C, D), E(D, A)$$

we compute a (simple) decomposition $HD(r)$, for instance the following:



This decomposition then yields the following set of rules r^*, when applying the steps discussed above:

$$\Sigma_D(D) \leftarrow E(D, A)$$

$$T_1(B, D) \leftarrow E(B, C), \neg E(C, D), \Sigma_D(D)$$

$$H(A, D) \leftarrow E(A, B), E(D, A), T_1(B, D)$$

The resulting set of rules is equivalent to the rule r in the sense of Theorem 4 below, however the number of possible ground rules is now only in $O(2^{\max_{n \in N} |\chi(n)|})$ instead of $O(2^{|\mathbf{X}|})$, with $\mathbf{X}$ the variables in r. ◀

Once we have preprocessed a rule (or, every rule in a logic program), it is easy to recreate the answer sets of the original program, as the following theorem states:

▶ **Theorem 4.** *Let Π be a logic program. Then for every answer set A of Π there exists exactly one answer set $A_{r^*} \supseteq A$ of Π_{r^*} and for every answer set A_{r^*} of Π_{r^*} there exists exactly one answer set $A \subseteq A_{r^*}$ of Π, such that in both cases it holds that $B_\Pi \cap A_{r^*} = A$.*

Due to space constraints, we refer the reader to the full version of this paper [22] for the proof of this and the next theorem.

Note that Theorem 4 also shows that we can replace in a program Π step-by-step each rule r by the corresponding replacement r^* and obtain a program equivalent to Π in the sense of Theorem 4 where each rule has been decomposed.

This leads to a decrease in grounding size, depending on the treewidth of the rules in the program. We define the size of a rule to be the size of its hypergraph representation. Then we can state the following theorem:

▶ **Theorem 5.** *Let Π be a logic program and $r \in \Pi$ a rule of size n. If r has bounded treewidth, then the size of $Gr(r^*, \Pi_{r^*})$ is linear in the size of the rule; and, in fact, is bounded by the function $O(2^k \cdot n)$, where k is the treewidth of r.*

▶ **Corollary 6.** *Let Π be a logic program. If every rule in Π has bounded treewidth, then the size of $Gr(\Pi)$ is linear in the size of Π.*

The implications of the above theorem, as we will show in Section 4, can lead to substantial speedups in the time it takes current-generation grounders to ground a logic program.

4 Experimental Evaluation

In order to empirically test our projected runtime behaviour, we have implemented a prototypical rule-preprocessing system available at

`http://www.dbai.tuwien.ac.at/research/project/dynasp/dynasp/#additional`

This tool makes use of the **SHARP** framework for hypertree decomposition-based algorithms[2]. Our system handles all basic ASP rules, including inequality as well as comparisons. However, arithmetical operations are currently not implemented.

Using our prototype, we performed a series of tests on a set of benchmarks from the third ASP competition[3] (see also [7]). We selected the following four problems from the competition

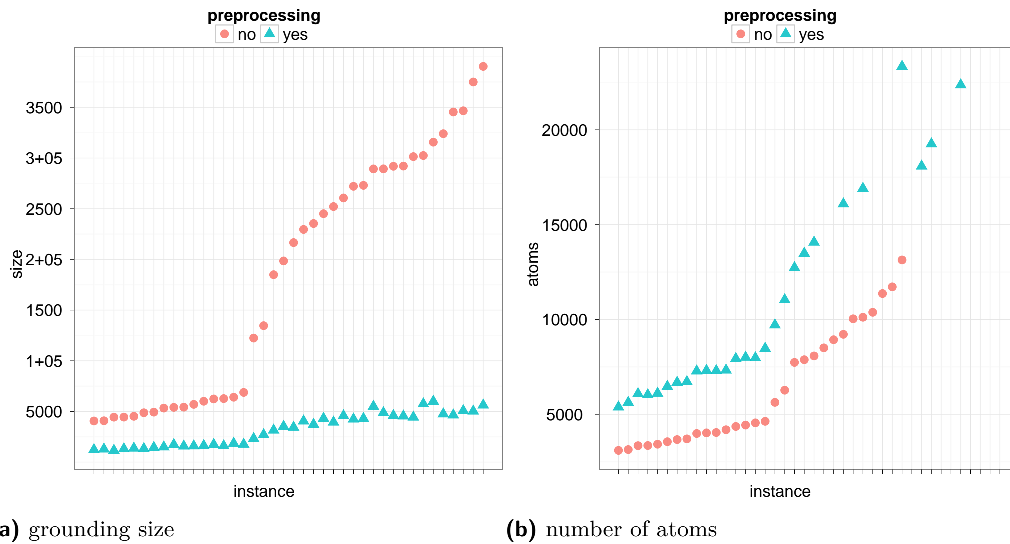
- Sokoban Decision
- Stable Marriage
- Minimal Diagnosis
- Partner Units Polynomial

This particular selection is motivated by the fact that these encodings do not use any arithmetical operations, choice rules or other ASP extensions, thus our first prototype is able to process them.

We chose problems from the ASP competition to show that, even though the encodings have been extensively hand-tuned, by intelligently splitting rules according to our algorithms, it is still possible to obtain improved grounding results. This also signifies the usefulness of our algorithm, because employing it would eliminate the need for extensive, time-consuming and notoriously imperfect hand-tuning.

In the following plots, red dots represent the value measured for the original benchmark instance and blue triangles represent the value measured for the preprocessed benchmark instance. Only the non-ground encoding was preprocessed, afterwards it was passed to gringo [15], together with the actual problem instance from the third ASP competition website, and the output was fed into claspd[4] [14]. For each problem a sample of 50 problem instances was selected. The time for preprocessing was not recorded in our plots, as for our benchmark instances it was not measurable (i.e. always below 0.1 seconds). The time limit for both gringo and claspd was 600 seconds each. If a timeout occurred, then no point was plotted for the respective instance. The "size" of the grounded program was measured by recording the number of variables, as determined by running claspd. As claspd introduces variables not only for atoms but also for rule bodies, this gives a useful impression of the actual problem size.

[2] `http://www.dbai.tuwien.ac.at/research/project/sharp`
[3] `http://aspcomp2011.mat.unical.it`
[4] In short test-runs we obtained similar results for the well-known DLV solver [20].



(a) grounding size **(b)** number of atoms

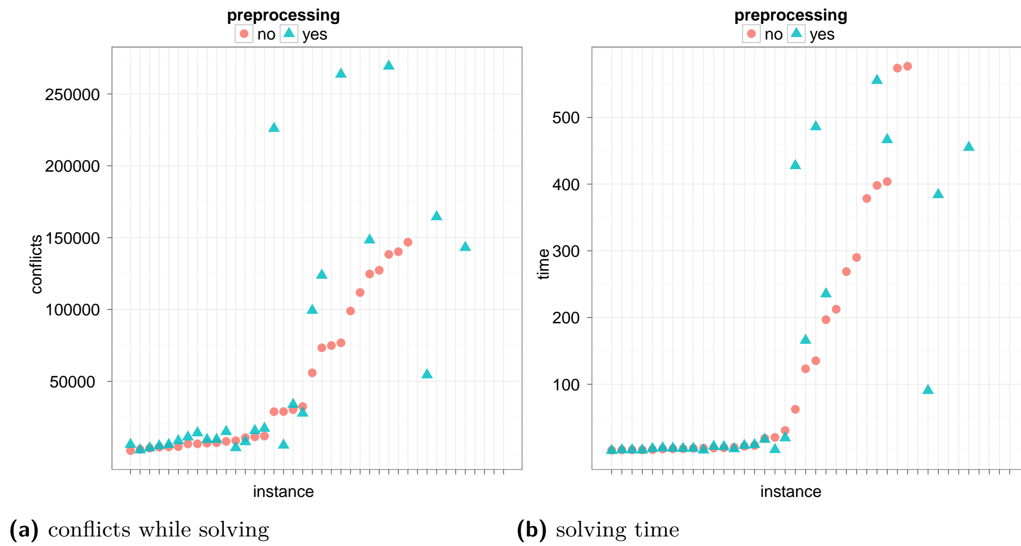
Figure 1 Grounding size and number of ground atoms for the Sokoban Decision problem.

Figure 1a shows the size of the preprocessed grounded Sokoban Decision program that was output by gringo in relation to the size of the grounding of the original. As can be seen the grounding size can be reduced dramatically. On average, the size of the ground program was reduced by 78%.

Figure 1b shows the number of atoms in the grounded Sokoban Decision problem. Given that our preprocessing strategy introduces a number of temporary predicates in the non-ground encoding, the number of actual atoms in the ground program increases by a linear factor, as each hypertree decomposition itself is linear in the size of the respective rule, and at each node, a single new temporary predicate is introduced. However, because of the nature of our preprocessing method, the number of rules decreases, and the decrease in the number of rules corresponds well with the decrease in size of the grounding.

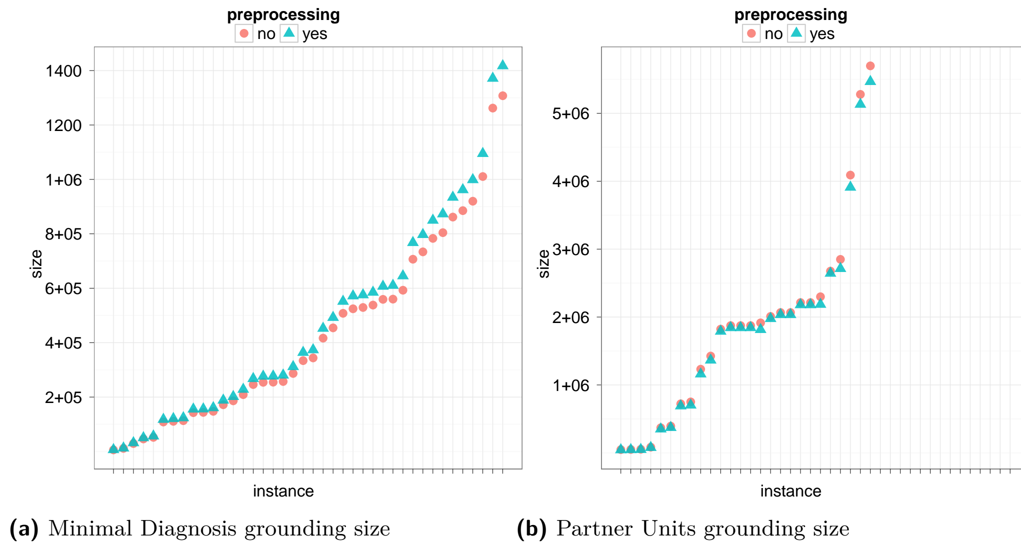
Figure **??** shows the time in seconds needed by claspd for solving the whole grounded problem, as well as the number of conflicts it encountered while doing so. Except for a few cases, the solving time of claspd, when combined with our preprocessing algorithm, is slightly increased, despite the much smaller size of the ground program. In rare cases however, there is a substantial slowdown of claspd. However we also noticed that for a number of instances, the smaller size of the ground program enabled claspd to solve the problem without hitting the time limit (see the topmost few instances in Figure 2b). The number of conflicts, shown in Figure 2a exhibit a similar behaviour. In most cases, an increased number of conflicts also entails an increased number of restarts of claspd.

Note that this increase in solving time could be easily eliminated if the solver (clasdp or otherwise) would be aware of the nature our the temporary atoms. The increase is mainly due to the solver making lots of unnecessary guesses about which temporary atoms should be in the answer set and which ones should not. However, by Lemma 3.4 in [22], given a set of non-temporary atoms, the temporary atoms for this set can always be deterministically calculated with minimal overhead. Therefore the solver could (a) ignore all rules with temporary head atoms, as by the Lemma 3.4 in [22] those are always satisfied, (b) for a guessed (partial) answer set, compute the corresponding temporary atoms as per the proof of Lemma 3.4 in [22] and (c) check, whether the head rule is satisfied.



(a) conflicts while solving **(b)** solving time

Figure 2 The number of conflicts encountered and the time in seconds needed by claspd for solving the grounded Sokoban Decision problem.

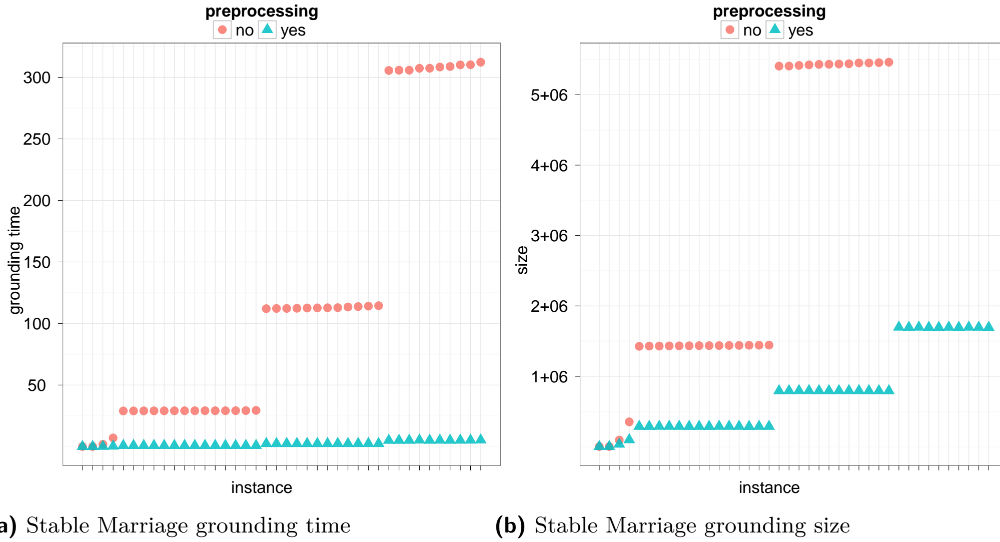


(a) Minimal Diagnosis grounding size **(b)** Partner Units grounding size

Figure 3 Grounding size of the Minimal Diagnosis and the Partner Units Polynomial problems.

The Sokoban Decision problem is the only problem in our benchmark selection that involves a solving phase. The other three problems that we discuss in the following are in fact solved by the grounder itself, therefore only the grounding size and grounding time plots are relevant for these problems.

Figure 3 shows the size of the grounding of the Minimal Diagnosis and Partner Units Polynomial problems. In the latter, only a single rule is split up, which is a rule with an all-positive body (i.e. no default negation). In this case our approach works best, because no



(a) Stable Marriage grounding time **(b)** Stable Marriage grounding size

Figure 4 Grounding time and grounding sizes for the Stable Marriage problem.

domain closure predicates (Σ) are needed. This simple split-up rule already decreases the grounding size by an average of 4%, as seen in Figure 3b.

On the other hand, for the Minimal Diagnosis problem in Figure 3a, all the rules that are split up are of the form

$$a(U, V) \leftarrow b(U, S), b(V, T), S \mathrel{!=} T$$

and therefore get split up into the following three rules:

$$T_1(V, S) \leftarrow b(V, T), S \mathrel{!=} T, \Sigma_S(S)$$
$$a(U, V) \leftarrow b(U, S), T_1(V, S)$$
$$\Sigma_S(S) \leftarrow b(U, S)$$

In this case, with our approach there is a chance that the actual grounding size increases, especially if many valid groundings for the fact $b(U, S)$ exist. Note that the grounding size with our preprocessing algorithm is always upper-bounded by $O(2^{\max_{n \in N} |\chi(n)|})$, as opposed to exponential in the number of variables of the whole rule. However these worst-case bounds are seldom exhausted. Whether a rule that gets split up as described above is actually beneficial to the overall grounding size, heavily depends on the configuration of the ground facts that are supplied to the grounder.

Note also that if our preprocessing approach would be integrated directly into the grounder, it would eliminate the need for domain closure predicates as the grounder already knows about the domain anyway. In this case it would be impossible for the grounding size to increase when employing our preprocessing approach and thus the only potential disadvantage could be eliminated.

Lastly, the Stable Marriage problem in Figure 4 shows the strength of our preprocessing algorithm. Here the non-ground rules contain many free variables and many predicates are joined together which forms the ideal basis for our algorithm. The non-ground rules force gringo to output almost exponentially many groundings for each rule. Figure 4a shows that

a significant speedup in all cases can here be gained, for the worst-case instances, cutting the grounding time from over 300 seconds to about 5 seconds. Also the grounding size decreases dramatically. In Figure 4b it can also be seen, that for the topmost 15 instances, clasp could not even finish parsing the gringo output within the timeout limit of 600 seconds. In case of our significantly reduced grounding size, this was however easily possible.

5 Conclusion

In this paper, we have presented a novel preprocessing strategy for non-ground rules in answer set programming. The preprocessing intelligently splits up non-ground rules into smaller ones by means of a hypertree decomposition in order to decrease the maximum number of variables per rule (and thus to reduce the size of the entire grounding). This technique follows the rule of thumb experienced ASP users will apply when encoding their problems. However, for complex rules, manual splitting becomes increasingly difficult and the readability of the encoding may suffer considerably. Also, programs may be automatically generated or specified for the purpose of presentation rather than for optimization (for instance, specifications in general game playing, see, e.g., [21]).

Benchmarks performed on problems used in the well-established answer set programming competition show significant potential of our strategy and thus warrant inclusion of such a method into existing grounders. The speedup of the grounding process is due to two factors:

Firstly, if the number of rule instantiations is reduced significantly, also the time it takes to compute and output each of these instantiations is reduced by the same amount. This effect can clearly be seen for the Stable Marriage problem in the previous section.

Secondly, by splitting up rules into smaller, equivalent ones, the number of joins between non-ground predicates is reduced. Therefore the grounder does not have to perform as many join operations as before, which also leads to a speedup of the grounding process.

Future Work

In order to use the demonstrated positive effects of our algorithm in state-of-the-art ASP grounders and solvers, there are two approaches worth investigating.

Firstly, if this preprocessing approach is directly incorporated to a grounder, the grounder may use the information about temporary predicates in order to speed up the grounding process further. Also, the domain closure predicates (Σ) are currently only a workaround, as currently our preprocessing algorithm has no information about the domain of specific variables in a non-ground rule. However, if included directly into the grounder, the domain closure predicates would become obsolete, as the grounder can immediately fill the respective variables with their now known domain, as the grounder has full information about the ground facts and domains of the various predicates and variables. This would not only lead to a speedup, but also would further decrease the size of the grounding, as the domain predicates do no longer exist, eliminating also the increase in size of the Minimal Diagnosis grounding.

Secondly, even though the size of the ground program decreases in all our benchmark cases except the Minimal Diagnosis problem, the solving time actually increases. This means that claspd is currently not aware of the tree-like structure of the split-up rules in the preprocessed and grounded instance. If the grounder could pass information about the temporary predicates to the solver, this could significantly speed up the solving process, as the temporary predicates could be automatically dismissed from the computation and the answer sets.

References

1. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases.* Addison-Wesley, 1995.
2. C. Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving.* Cambridge University Press, 2003.
3. H. L. Bodlaender. A tourist guide through treewidth. *Acta Cybern.*, 11(1-2):1–22, 1993.
4. H. L. Bodlaender. Discovering treewidth. In P. Vojtás, M. Bieliková, B. Charron-Bost, and O. Sýkora, editors, *SOFSEM 2005: 31st Conference on Current Trends in Theory and Practice of Computer Science. Proceedings*, volume 3381 of *LNCS*, pages 1–16. Springer, 2005.
5. H. L. Bodlaender and A. M. C. A. Koster. Treewidth computations I. Upper bounds. *Inf. Comput.*, 208(3):259–275, 2010.
6. G. Brewka, T. Eiter, and M. Truszczynski. Answer set programming at a glance. *Commun. ACM*, 54(12):92–103, 2011.
7. F. Calimeri, G. Ianni, F. Ricca, M. Alviano, A. Bria, G. Catalano, S. Cozza, W. Faber, O. Febbraro, N. Leone, M. Manna, A. Martello, C. Panetta, S. Perri, K. Reale, M. C. Santoro, M. Sirianni, G. Terracina, and P. Veltri. The third answer set programming competition: Preliminary report of the system competition track. In J. P. Delgrande and W. Faber, editors, *11th Conference on Logic Programming and Nonmonotonic Reasoning, LPNMR 2011. Proceedings*, volume 6645 of *LNCS*, pages 388–403. Springer, 2011.
8. A. Dermaku, T. Ganzow, G. Gottlob, B. J. McMahan, N. Musliu, and M. Samer. Heuristic methods for hypertree decomposition. In A. F. Gelbukh and E. F. Morales, editors, *MICAI 2008: 7th Mexican International Conference on Artificial Intelligence, Proceedings*, volume 5317 of *LNCS*, pages 1–11. Springer, 2008.
9. T. Eiter, W. Faber, M. Fink, and S. Woltran. Complexity results for answer set programming with bounded predicate arities and implications. *Ann. Math. Artif. Intell.*, 51(2-4):123–165, 2007.
10. W. Faber, N. Leone, C. Mateis, and G. Pfeifer. Using database optimization techniques for nonmonotonic reasoning. In *Proc. 7th International Workshop on Deductive Databases and Logic Programming (DDLP'99)*, pages 135–139, 1999.
11. M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub. Challenges in answer set solving. In M. Balduccini and T. Son, editors, *Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning: Essays in Honor of Michael Gelfond*, volume 6565, pages 74–90. Springer, 2011.
12. M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. *clasp* : A conflict-driven answer set solver. In C. Baral, G. Brewka, and J. S. Schlipf, editors, *Logic Programming and Nonmonotonic Reasoning, 9th International Conference, LPNMR 2007. Proceedings*, volume 4483 of *LNCS*, pages 260–265. Springer, 2007.
13. M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. Advanced preprocessing for answer set solving. In M. Ghallab, C. D. Spyropoulos, N. Fakotakis, and N. M. Avouris, editors, *ECAI 2008 - 18th European Conference on Artificial Intelligence, Proceedings*, volume 178 of *Frontiers in Artificial Intelligence and Applications*, pages 15–19. IOS Press, 2008.
14. M. Gebser, B. Kaufmann, and T. Schaub. The conflict-driven answer set solver clasp: Progress report. In E. Erdem, F. Lin, and T. Schaub, editors, *Logic Programming and Nonmonotonic Reasoning, 10th International Conference, LPNMR 2009, Potsdam, Germany, September 14-18, 2009. Proceedings*, volume 5753 of *LNCS*, pages 509–514. Springer, 2009.
15. M. Gebser, T. Schaub, and S. Thiele. Gringo : A new grounder for answer set programming. In C. Baral, G. Brewka, and J. S. Schlipf, editors, *Logic Programming and Nonmonotonic Reasoning, 9th International Conference, LPNMR 2007, Tempe, AZ, USA, May 15-17, 2007, Proceedings*, volume 4483 of *LNCS*, pages 266–271. Springer, 2007.
16. M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proc. ICLP/SLP*, pages 1070–1080, 1988.

17 M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Comput.*, 9(3/4):365–386, 1991.

18 G. Gottlob, N. Leone, and F. Scarcello. Hypertree decompositions and tractable queries. In *Proceedings of the Eighteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, 1999*, pages 21–32. ACM Press, 1999.

19 G. Gottlob, N. Leone, and F. Scarcello. The complexity of acyclic conjunctive queries. *J. ACM*, 48(3):431–498, 2001.

20 N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. The dlv system for knowledge representation and reasoning. *ACM Trans. Comput. Log.*, 7(3):499–562, 2006.

21 M. Möller, M. T. Schneider, M. Wegner, and T. Schaub. Centurio, a general game player: Parallel, Java- and ASP-based. *Künstliche Intelligenz*, 25(1):17–24, 2011.

22 M. Morak and S. Woltran. Preprocessing of complex non-ground rules in answer set programming. Technical Report DBAI-TR-2011-72 (revised version), Institute of Information Systems 184/2, Vienna University of Technology, Austria, 2012.

23 I. Niemelä and P. Simons. Smodels - an implementation of the stable model and well-founded semantics for normal logic programs. In J. Dix, U. Furbach, and A. Nerode, editors, *Logic Programming and Nonmonotonic Reasoning, 4th International Conference, LPNMR'97, Dagstuhl Castle, Germany. Proceedings*, volume 1265 of *Lecture Notes in Computer Science*, pages 421–430. Springer, 1997.

24 N. Robertson and P. D. Seymour. Graph minors. III. Planar tree-width. *J. Comb. Theory, Ser. B*, 36(1):49–64, 1984.

Two-Valued Logic Programs

Vladimir Lifschitz

University of Texas at Austin, USA

Abstract

We define a nonmonotonic formalism that shares some features with three other systems of nonmonotonic reasoning—default logic, logic programming with strong negation, and nonmonotonic causal logic—and study its possibilities as a language for describing actions.

1998 ACM Subject Classification D.1.6 Logic Programming

Keywords and phrases Answer set programming, Non monotonic reasoning, Foundations

Digital Object Identifier 10.4230/LIPIcs.ICLP.2012.259

1 Introduction

A stable model of a logic program, according to the original definition of the stable model semantics [4], is a set of ground atoms. Intuitively, the atoms that belong to the model are true, and all other atoms are false. Thus a stable model gives complete information about the truth values of all atoms; the incompleteness of information encoded in a logic program can be only expressed by the existence of several stable models.

When classical (strong) negation was added to the language of logic programs in [5], and the term "answer set" was introduced, the situation changed. An answer set is a set of ground literals that is consistent but possibly incomplete. Thus an answer set can be thought of as a function that assigns to each ground atom A one of three values: *true* (A belongs to the set), *false* ($\neg A$ belongs to the set), or *unknown* (the set contains neither A nor $\neg A$). An answer set can represent incomplete information.

On the other hand, in the original stable model semantics truth and falsity were not symmetric: if an atom does not occur in the heads of rules of a logic program then it is treated as false. In the answer set semantics, the truth value of such an atom is *unknown*.

To sum up, 1988-style stable models are asymmetric and represent complete information; 1991-style answer sets are symmetric and can represent incomplete information.

The nonmonotonic formalism described in this note is motivated by the fact that some important uses of answer set programming (ASP) call for both symmetry and completeness. We often encounter this situation when ASP is applied to reasoning about truth-valued fluents. To describe a state, we need to provide complete information about the values of all fluents. The modification of the stable model semantics defined below treats truth and falsity symmetrically, like the 1991 version, and at the same time guarantees the completeness of information, as the 1988 version.

Two-valued logic programs share some features with default logic [13] and with nonmonotonic causal logic in the sense of [11]. As in the case of default logic, the nonmonotonicity of two-valued logic programs is determined by the use of "justifications." Literals play a special role in their syntax, as they do in the definition of an answer set in [5], and this fact allows us to make their semantics relatively simple: it does not refer to deductive closure in the sense of classical logic. As in nonmonotonic causal logic, their semantics is defined in terms of two-valued truth assignments—or, in other words, consistent and complete sets of literals—rather than (possibly incomplete) extensions or (possibly incomplete) answer sets.

© Vladimir Lifschitz;
licensed under Creative Commons License ND
Technical Communications of the 28th International Conference on Logic Programming (ICLP'12).
Editors: A. Dovier and V. Santos Costa; pp. 259–266

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

2 Definitions

2.1 Syntax

In this note, *formulas* are propositional formulas formed from a fixed set σ of atoms. A *(two-valued) rule* is an expression of the form

$$L_0 \leftarrow L_1, \ldots, L_n \,:\, F, \tag{1}$$

where the *head* L_0 and the *premises* $L_1, \ldots, L_n$ ($n \geq 0$) are literals, and the *justification* F is a formula. Rule (1) reads: derive L_0 from $L_1, \ldots, L_n$ if F is a consistent assumption.

A pair of rules of the form

$$\begin{array}{rl} A & \leftarrow L_1, \ldots, L_n \,:\, F \wedge A, \\ \neg A & \leftarrow L_1, \ldots, L_n \,:\, F \wedge \neg A, \end{array}$$

where A is an atom, can be abbreviated as

$$\{A\} \leftarrow L_1, \ldots, L_n \,:\, F \tag{2}$$

("derive any of the literals A, $\neg A$ from $L_1, \ldots, L_n$ if that literal is consistent with assumption F"). This abbreviation is similar to choice rules in the sense of [12]. Both in (1) and in (2), if F is $\top$ (truth) then we will drop the colon and F at the end of the rule. If, in addition, $n = 0$ then $\leftarrow$ can be dropped too.

A *(two-valued) program* is a set of rules.

2.2 Semantics

As in classical propositional logic, an *interpretation* is a function from σ to $\{\mathit{false}, \mathit{true}\}$. We will identify an interpretation I with the set of literals that are satisfied by I.

The *reduct* of a program Π relative to an interpretation I is the set of rules

$$L_0 \leftarrow L_1, \ldots, L_n \tag{3}$$

corresponding to the rules (1) of Π for which $I \models F$. We say that I is a *model* of Π if the smallest set of literals closed under the rules (3) equals I. In other words, models of Π are fixpoints of the operator α_Π from interpretations to sets of literals defined as follows: $\alpha_\Pi(I)$ is the smallest set of literals closed under the reduct of Π relative to I.

It is clear that the set of models of a program is not affected by replacing the justification of a rule with an equivalent formula. It is clear also that every literal that belongs to a model of Π is the head of a rule of Π. It follows that if some atom from σ does not occur in the heads of rules then the program is inconsistent (that is, has no models). This is a property that two-valued programs share with causal theories in the sense of [11].

2.3 Example

Let Π be the program

$$\begin{array}{l} \{a\}, \\ b \leftarrow a, \end{array} \tag{4}$$

or, written in full,

$$\begin{array}{rl} a & \leftarrow \,:\, a, \\ \neg a & \leftarrow \,:\, \neg a, \\ b & \leftarrow a \,:\, \top, \end{array}$$

with $\sigma = \{a, b\}$. Since Π has no rules with the head $\neg b$, the only possible models are $I_1 = \{a, b\}$ and $I_2 = \{\neg a, b\}$. The reduct of Π relative to I_1 consists of the rules a and $b \leftarrow a$, so that $\alpha_\Pi(I_1) = \{a, b\} = I_1$; I_1 is a model. The reduct relative to I_2 consists of the rules $\neg a$ and $b \leftarrow a$, so that $\alpha_\Pi(I_2) = \{\neg a\} \neq I_2$; I_2 is not a model.

2.4 Constraints

Adding a pair of rules of the form

$$\begin{array}{rl} A & \leftarrow : F, \\ \neg A & \leftarrow : F \end{array} \tag{5}$$

to a program Π eliminates the models of Π that satisfy F. (Proof: adding these rules makes the reduct of the program relative to I inconsistent if I satisfies F, and does not affect the reduct otherwise.) We will call (5) a *constraint* and write it as $\leftarrow F$.

2.5 Clausal Form

We say that a program Π is in *clausal form* if each of its justifications is a conjunction of literals (possibly the empty conjunction $\top$). For instance, program (4) is in clausal form.

Replacing a rule of the form

$$L_0 \leftarrow L_1, \ldots, L_n \, : \, F \vee G$$

in any program with the pair of rules

$$\begin{array}{c} L_0 \leftarrow L_1, \ldots, L_n \, : \, F, \\ L_0 \leftarrow L_1, \ldots, L_n \, : \, G \end{array}$$

does not affect the set of models. (Proof: for any interpretation I, the reduct relative to I remains the same.) It follows that any program can be converted to clausal form by rewriting the justifications in disjunctive normal form and then breaking every rule into several rules corresponding to the disjunctive terms of its justification.

3 Relation to Traditional ASP Programs

3.1 Reduction to Programs with Strong Negation

As mentioned in the introduction, two-valued programs are essentially a special case of nondisjunctive programs with strong negation. To make that claim precise, we will define a simple translation that turns any two-valued program Π in clausal form into a program with strong negation. That program, $tv2sn(\Pi)$, is the set of rules

$$L_0 \leftarrow L_1, \ldots, L_n, not\ \overline{L_{n+1}}, \ldots, not\ \overline{L_p}$$

for all rules

$$L_0 \leftarrow L_1, \ldots, L_n \, : \, L_{n+1} \wedge \cdots \wedge L_p$$

of Π. (By $\overline{L}$ we denote the literal complementary to L.) For instance, $tv2sn$ turns program (4) into

$$\begin{array}{rl} a & \leftarrow not\ \neg a, \\ \neg a & \leftarrow not\ a, \\ b & \leftarrow a. \end{array} \tag{6}$$

An interpretation I (that is to say, a sound and complete set of literals) is a model of Π iff I is an answer set of $tv2sn(\Pi)$. (Proof: the reduct of $tv2sn(\Pi)$ relative to I in the sense of [5] is identical to the reduct of Π relative to I.) In other words, models of Π are identical to complete answer sets of $tv2sn(\Pi)$. For instance, program (6) has two answer sets, $\{a, b\}$ and $\{\neg a\}$. The first of them is the only model of (4); the second is incomplete.

Incomplete answer sets of a program with strong negation can be eliminated by adding the rules

$$\leftarrow not\ A, not\ \neg A \tag{7}$$

for all atoms A. Consequently models of a program Π in clausal form are identical to the answer sets of the program obtained from $tv2sn(\Pi)$ by adding rules (7) for all A from σ.

3.2 Complete Answer Sets in Disguise

In many ASP programs, strong negation is not used at all. Answer sets of such a program are sets of positive literals; the intuition is that the falsity of an atom is indicated by its absence in the answer set, rather than the presence of its negation. In this situation, we can think of an answer set consisting of positive literals as a "complete answer set in disguise"—as a complete answer set X with all negative literals removed (symbolically, $X \cap \sigma$).

Similarly, a program without strong negation can be viewed as a "two-valued program in disguise." Let Π be a set of rules of the form

$$A_0 \leftarrow A_1, \ldots, A_n, not\ A_{n+1}, \ldots, not\ A_p, \tag{8}$$

where each A_i is an atom. By $lp2tv(\Pi)$ we denote the two-valued program consisting of the rules

$$A_0 \leftarrow A_1, \ldots, A_n \,:\, \neg A_{n+1} \wedge \cdots \wedge \neg A_p$$

for all rules (8) of Π, and the rules

$$\neg A \leftarrow \,:\, \neg A \tag{9}$$

for all atoms A. Rule (9) makes the closed world assumption for A explicit.

Answer sets of Π can be characterized as sets of the form $X \cap \sigma$, where X is a model of $lp2tv(\Pi)$. (Proof: $tv2sn(lp2tv(\Pi))$ is the closed world interpretation of Π in the sense of [5, Section 6].) Thus the map $X \mapsto X \cap \sigma$ is a 1–1 correspondence between the models of $lp2tv(\Pi)$ and the models of Π.

Consider, for instance, the program Π consisting of one rule $a \leftarrow not\ b$. The corresponding two-valued program is

$$\begin{aligned} a &\leftarrow : \neg b, \\ \neg a &\leftarrow : \neg a, \\ \neg b &\leftarrow : \neg b. \end{aligned}$$

Its only model is $\{a, \neg b\}$. By removing the negative literal $\neg b$ from it, we get $\{a\}$, the only answer set of Π.

4 Relation to Causal Logic

Recall that a causal theory in the sense of [11] is a set of rules of the form $F \leftarrow G$, where F and G are propositional formulas. The *reduct* of a causal theory T relative to an interpretation I is the set of the heads F of all rules $F \leftarrow G$ of T for which I satisfies G. An

interpretation I is a *model* of a causal theory T if the reduct of T relative to I is satisfied by I and is not satisfied by any other interpretation. This semantics formalizes the philosophical principle that McCain and Turner call the law of universal causation.

A causal theory is *definite* if the head of each of its rules is a literal. For any definite causal theory T, we define the corresponding two-valued program $ct2tv(T)$ as the set of rules $F \leftarrow : G$ for all rules $F \leftarrow G$ of T. Models of any definite causal theory T are identical to models of program $ct2tv(T)$. (Proof: consider the reduct X of a definite causal theory T relative to an interpetation I; I is the only interpretation satisfying X iff $X = I$.) In other words, definite causal theories are essentially two-valued programs whose rules have no premises. We can say also that two-valued programs generalize definite causal theories by allowing "logic programming style premises" in the bodies of rules.

If the bodies of rules of a definite causal theory T are conjunctions of literals then $ct2tv(T)$ is a program in clausal form, and the transformation *tv2sn* defined above can be used to turn that program into a program with strong negation. By composing *ct2tv* with *tv2sn* we get the translation from the language of causal theories into logic programming with strong negation familiar from [10, Section 6.3.3].

5 Representing Action Descriptions by Two-Valued Programs

Consider a finite set σ of propositional atoms divided into two groups, *fluents* and *elementary actions*. An *action* is a function from elementary actions to truth values. A *transition system* T is determined by a set of functions from fluents to truth values, called the *states* of T, and a set of triples $\langle s_0, a, s_1\rangle$, where s_0 and s_1 are states of T, and a is an action. These triples are called the *transitions* of T. A transition system can be visualized as a directed graph that has states as its vertices, with an edge from s_0 to s_1 labeled a for every transition $\langle s_0, a, s_1\rangle$. Informally speaking, a transition $\langle s_0, a, s_1\rangle$ expresses the possibility of the system changing its state from s_0 to s_1 when the elementary actions to which a assigns the value *true* are concurrently executed.

Action description languages $\mathcal{B}$ and $\mathcal{C}$, defined in [6, Section 5, 6] and [8], and reviewed in [7, Section 2], serve for describing action domains by specifying transition systems. They are closely related to logic programs under the answer set semantics [1, 9]. In this section we show how the semantics of $\mathcal{B}$ and of a large ("definite") fragment of $\mathcal{C}$ can be characterized in terms of two-valued programs.

5.1 Translating $\mathcal{B}$-Descriptions

This review of the syntax of $\mathcal{B}$ follows [7, Section 2.1.1]. A *fluent literal* is a literal containing a fluent. A *condition* is a set of fluent literals. An *action description in the language* $\mathcal{B}$, or a $\mathcal{B}$*-description*, is a set of expressions of two forms: *static laws*

$$L \textbf{ if } C,$$

where L is a fluent literal, and C is a condition, and *dynamic laws*

$$e \textbf{ causes } L \textbf{ if } C,$$

where e is an elementary action, L is a fluent literal, and C is a condition. The semantics of the language (see, for instance, [7, Section 2.1.2]) defines, for every $\mathcal{B}$-description D, which transition system it represents.

The set of transitions of that system can be described by the program *b2tv*(D), defined as follows. Its signature σ_1 consists of the symbols of the forms

$$f(0),\ e(0),\ f(1), \tag{10}$$

where f is a fluent and e is an elementary action. Its rules are

(i) $L(t) \leftarrow L_1(t), \ldots, L_n(t)$, where $t = 0, 1$, for each static law

$$L \textbf{ if } L_1, \ldots, L_n$$

from D;

(ii) $L(1) \leftarrow e(0), L_1(0), \ldots, L_n(0)$ for each dynamic law

$$e \textbf{ causes } L \textbf{ if } L_1, \ldots, L_n$$

from D;

(iii) $L(1) \leftarrow L(0) : L(1)$ for every fluent literal L,

(iv) $\{A(0)\}$ for every atom A of σ.

Rules (iii) solve the frame problem by formalizing the commonsense law of inertia [14]; they are similar to the "frame default" from [13]. Rules (iv) express that both the initial values of fluents and the elementary actions to be executed can be chosen arbitrarily.

Recall that we agreed to identify truth-valued functions with sets of literals (Section 2.2). Using this convention, we can characterize the set of transitions of an arbitrary $\mathcal{B}$-description D in terms of models of *b2tv*(D) as follows:

Proposition. *For any sets s_0, s_1 of fluent literals, and any action a, $\langle s_0, a, s_1\rangle$ is a transition of $T(D)$ iff the set*

$$\{L(0) : L \in s_0 \cup a\} \cup \{L(1) : L \in s_1\}$$

is a model of b2tv(D).

This fact is a reformulation of Lemma 2 from [7], in view of the property of the transformation *tv2sn* noted in Section 3.1. It establishes a 1–1 correspondence between the transitions of D and the models of *b2tv*(D).

5.2 Translating Definite $\mathcal{C}$-Descriptions

This review of the syntax of $\mathcal{C}$ follows [7, Section 2.2.1]. An *action description in the language* $\mathcal{C}$, or *$\mathcal{C}$-description*, is a set of expressions of the two forms: *static laws*

$$\textbf{caused } F \textbf{ if } G, \tag{11}$$

where F and G are formulas that do not contain elementary actions, and *dynamic laws*

$$\textbf{caused } F \textbf{ if } G \textbf{ after } H, \tag{12}$$

where F and G are formulas that do not contain elementary actions, and H is a formula. The semantics of the language (see, for instance, [7, Section 2.2.2]) defines, for every $\mathcal{C}$-description D, which transition system it represents.

A $\mathcal{C}$-description is *definite* if, in each of its laws (11), (12), the head F is a literal. For any definite $\mathcal{C}$-description D, the set of transitions of the corresponding system can be described by the program *c2tv*(D), defined as follows. Its signature σ_1 consists of the same

symbols (10) as in the case of $\mathcal{B}$-descriptions. For any formula F of the signature σ, by $F(0)$ we will denote the formula of the signature σ_1 obtained from F by appending the string $'(0)'$ to each atom. For any formula F of the signature σ that does not contain elementary actions, by $F(1)$ we will denote the formula of the signature σ_1 obtained from F by appending the string $'(1)'$ to each atom. The rules of *c2tv*(D) are

(i) $F(t) \leftarrow : G(t)$, where $t = 0, 1$, for each static law (11) from D;
(ii) $F(1) \leftarrow : G(1) \wedge H(0)$ for each dynamic law (12) from D;
(iii) $\{A(0)\}$ for every atom A of σ.

The characterization of transitions given by the proposition from Section 5.1, with *b2tv* replaced by *c2tv*, holds for any definite $\mathcal{C}$-description D. This fact is a corollary to Proposition 2 from [8], in view of the property of the transformation *ct2tv* noted in Section 4 above. It establishes a 1–1 correspondence between the transitions of D and the models of *c2tv*(D).

If H in a dynamic law (12) is a conjunction of literals $L_1 \wedge \cdots \wedge L_n$ then the rule in clause (ii) of the definition of *c2tv* can be rewritten as

$$F(1) \leftarrow L_1(0), \ldots, L_n(0) \,:\, G(1),$$

and the models of the theory will remain the same.

6 Conclusion

We have seen that the language of two-valued programs is sufficiently rich for expressing the ASP solution to the frame problem that exploits the distinction between strong negation and negation as failure (Section 5.1), and that it can model the uses of ASP that avoid strong negation altogether (Section 3.2). There are also "mixed" representations, which express the falsity of some atoms explicitly, in terms of strong negation, and treat the falsity of other atoms in the spirit of an implicit closed world assumption. Such representations can be often expressed by two-valued programs as well.

Uses of ASP for which the language of two-valued programs is inadequate are relatively rare, but they do exist. Incomplete answer sets are essential for representing "weak exceptions" to defaults, as in [2, Example 3.2]: birds normally fly; wounded birds *may or may not* fly. Another example is given by the approach to conformant planning presented in [15]. The planner described in that paper operates with "partial states"—incomplete sets of literals that approximate states in the sense of Section 5. The difference between the applications of ASP that can be naturally represented by two-valued programs and the applications for which it is not the case is an important distinction between two kinds of answer set programs.

Two-valued programs can be viewed as a special case of multi-valued propositional formulas under the stable model semantics introduced in [3].[1] A preprocessor converting such formulas (perhaps from a subset that includes two-valued programs) into input accepted by answer set solvers would be a useful knowledge representation tool.

Acknowledgements

Thanks to Marc Denecker, Michael Gelfond, Joohyung Lee, Yuliya Lierler, and Fangkai Yang for useful discussions related to the topic of this note, and to the anonymous referees for their comments.

[1] Joohyung Lee, personal communication, April 4, 2012.

References

1. Marcello Balduccini and Michael Gelfond. Diagnostic reasoning with A-Prolog. *Theory and Practice of Logic Programming*, 3(4-5):425–461, 2003.
2. Chitta Baral and Michael Gelfond. Logic programming and knowledge representation. *Journal of Logic Programming*, 19,20:73–148, 1994.
3. Michael Bartholomew and Joohyung Lee. Stable models of formulas with intensional functions. In *Proceedings of International Conference on Principles of Knowledge Representation and Reasoning (KR)*, 2012. To appear.
4. Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In Robert Kowalski and Kenneth Bowen, editors, *Proceedings of International Logic Programming Conference and Symposium*, pages 1070–1080. MIT Press, 1988.
5. Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9:365–385, 1991.
6. Michael Gelfond and Vladimir Lifschitz. Action languages[2]. *Electronic Transactions on Artificial Intelligence*, 3:195–210, 1998.
7. Michael Gelfond and Vladimir Lifschitz. The common core of action languages $\mathcal{B}$ and $\mathcal{C}$. In these proceedings, 2012.
8. Enrico Giunchiglia and Vladimir Lifschitz. An action language based on causal explanation: Preliminary report. In *Proceedings of National Conference on Artificial Intelligence (AAAI)*, pages 623–630. AAAI Press, 1998.
9. Vladimir Lifschitz and Hudson Turner. Representing transition systems by logic programs. In *Proceedings of International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, pages 92–106, 1999.
10. Norman McCain. *Causality in Commonsense Reasoning about Actions*[3]. PhD thesis, University of Texas at Austin, 1997.
11. Norman McCain and Hudson Turner. Causal theories of action and change. In *Proceedings of National Conference on Artificial Intelligence (AAAI)*, pages 460–465, 1997.
12. 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.
13. Raymond Reiter. A logic for default reasoning. *Artificial Intelligence*, 13:81–132, 1980.
14. Murray Shanahan. *Solving the Frame Problem: A Mathematical Investigation of the Common Sense Law of Inertia.* MIT Press, 1997.
15. Phan Huy Tu, Tran Cao Son, Michael Gelfond, and Ricardo Morales. Approximation of action theories and its application to conformant planning. *Artificial Intelligence*, 175:79–119, 2011.

[2] `http://www.ep.liu.se/ea/cis/1998/016/`
[3] `ftp://ftp.cs.utexas.edu/pub/techreports/tr97-25.ps.gz`

Possibilistic Nested Logic Programs

Juan Carlos Nieves and Helena Lindgren

Department of Computing Science
Umeå University
SE-901 87, Umeå, Sweden
jcnieves,helena@cs.umu.se

Abstract

We introduce the class of possibilistic nested logic programs. These possibilistic logic programs allow us to use nested expressions in the bodies and the heads of their rules. By considering a possibilistic nested logic program as a possibilistic theory, a construction of a possibilistic logic programing semantics based on answer sets for nested logic programs and the proof theory of possibilistic logic is defined. We show that this new semantics for possibilistic logic programs is computable by means of transforming possibilistic nested logic programs into possibilistic disjunctive logic programs. The expressiveness of the possibilistic nested logic programs is illustrated by scenarios from the medical domain. In particular, we exemplify how possibilistic nested logic programs are expressive enough for capturing medical guidelines which are pervaded of vagueness and qualitative information.

1998 ACM Subject Classification I.2.3 Deduction and Theorem Proving

Keywords and phrases Answer Set Programming, Uncertain Information, Possibilistic Reasoning

Digital Object Identifier 10.4230/LIPIcs.ICLP.2012.267

1 Introduction

In the literature, one can find different approaches for encoding qualitative information [12, 18, 20]. A common strategy for capturing qualitative information is by using non-numerical values. Possibilistic reasoning has shown to be a suitable approach for dealing with qualitative reasoning [18]. In particular, this feature is based on the fact that the possibilistic values of a possibilistic knowledge base can be non-numerical values which capture the uncertainty of a knowledge base.

In the context of possibilistic logic programming, there are few proposals which deal with non-numerical values which are not totally ordered [14]. However, the expressiveness of the approach presented in [14] is restricted to disjunctive logic programs. Indeed most of the logic programming approaches which deal with uncertain information make syntactic restrictions to their specification languages. By not having syntactic restriction in a symbolic specification, one can provide a transparent method to capture real data domains. For instance, there are different ways to interpret a medical guideline for diagnosis (we will illustrate this in the body of the paper). The presence of more than one disease in an individual (comorbidity) is common in older people, and some guidelines have expressions supporting both potential comorbidity and differential diagnosis. For example, the most frequently used guideline for mental diseases uses a multiple axis system between certain guidelines for expressing comorbidity [2]. Still, additional diagnostic criteria are needed to assess a potential dementia disease, which use a different way to express the ambiguity built into diagnosis of neurological and mental diseases. The uncertainty is reflected in the vocabulary

© Juan Carlos Nieves and Helena Lindgren;
licensed under Creative Commons License ND
Technical Communications of the 28th International Conference on Logic Programming (ICLP'12).
Editors: A. Dovier and V. Santos Costa; pp. 267–276

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

used in these guidelines (e.g., possible, probable, unlikely, supportive, etc.). The meaning of such expressions represents different and sometimes overlapping ranges of possibilities, which, consequently, cannot be *totally ordered* when are formalized. However, there are practical reasons for reusing the vocabulary in the guidelines for expressing the knowledge in formal reasoning. Firstly, to provide to a clinician explanations of the reasoning and to mirror the uncertainty of an assessment in the available evidence-based medical knowledge, and secondly, to allow an expert physician to validate a knowledge base which handles comorbidity. An example of a possibilistic rule, which captures both uncertainty, ambiguity and a potential multi-diagnosis, is the following: ***possible***: $DLB \wedge AD \leftarrow visHall \wedge slow \wedge prog \wedge epiMem$. (It is possible that both Alzheimer's disease and Lewy Body dementia are present based on the observed symptoms). Another example illuminates how negation as failure can be utilized: ***probable***: $VaD \leftarrow fn \wedge radVasc \wedge\ not\ (AD \vee DLB)$ (vascular dementia is probable present considering the observations and since we do not have reasons to believe that Alzheimer's disease or Lewy Body dementia are present).

Against this background, we extended the results presented in [14] and [10] by introducing the class of *possibilistic nested logic programs*. These possibilistic logic programs allow us to use nested expressions in the bodies and the heads of their rules. Given that possibilistic logic is *axiomatizable* in the necessity-value case [6], we define the semantics of the possibilistic nested logic programs by considering the *proof theory* of possibilistic logic. In particular, by considering a possibilistic nested logic program as a possibilistic theory, a construction of a possibilistic logic programing semantics based on answer sets for nested logic programs [10] and the proof theory of possibilistic logic [6] is defined. It is worth mentioning that the answer set semantics inference can also be characterized as a *logic inference* in terms of the proof theory of intuitionistic logic and intermediate logics [17, 16].

We also show that the new possibilistic semantics generalizes the previous possibilistic semantics introduced in [13, 14]. In order to define a general method for computing the possibilistic answer sets of a possibilistic nested program, the idea of equivalence between possibilistic programs is explored.

The rest of the paper is divided as follows: In the following section, some basic concepts of nested logic programs and possibilistic logic are introduced. After this, the syntaxis and semantics of the possibilistic nested logic programs are introduced. In this section, some properties of the possibilistic nested logic semantics are identified (by lack of space, the formal proofs are omitted). In the last section, an outline of our conclusions and future work is presented.

2 Background

In this section, we introduce some basic concepts of Nested Logic Programs [10] and Possibilistic Logic [6]. We assume that the reader is familiarized with basic concepts in classical logic and logic programming semantics, *e.g.* interpretations, models, *etc.* A good introductory treatment of these concepts can be found in [3].

2.1 Nested Logic Programs

The considered language consists of: (i) an enumerable set $\mathcal{A}$ of elements called *atoms* (denoted by a, b, c, ...), (ii) *logic connectives* $\wedge$, $\vee$, $\neg$, *not*, $\bot$, $\top$ in which $\{\wedge, \vee\}$, $\{not, \neg\}$, $\{\top, \bot\}$ are 2-place, 1-place and 0-place connectives respectively and (iii) *auxiliary symbols* "(", ")", ".". We refer to a *literal* as an atom a or an extended atom $\neg a$. We denote by $\mathcal{L}$ the set of literals built using elements in $\mathcal{A}$.

Literals, $\bot$ and $\top$ are considered *elementary formulas*, while $\{\vee, \wedge, not\}$ *formulas* (denoted as A, B, C, ...) are constructed from elementary formulas using the connectives $\{\vee, \wedge, not\}$ arbitrarily nested (strong negation $\neg$ is allowed to appear only in front of atoms). As probably noted, we are considering two types of negations in this paper: strong negation $\neg$ (as it called by the Answer Set Programming community [3]) and negation as failure not.

Given a finite set of literals $\mathcal{L}$, a *nested rule* is an expression of the form $H \leftarrow B$, where H and B are either an elementary formula or a $\{\vee, \wedge, not\}$ formulas (known as the *head* and the *body* respectively). Some particular cases are *facts*, of the form $H \leftarrow \top$ (written as H), and *constraints*, $\bot \leftarrow B$ (written as $\leftarrow B$). If no occurrences of not appear in a rule, then the rule is called a *definite nested rule*.

A *nested logic program* P is a finite set of nested rules. If the program does not contain not, then the program is called a *definite nested program*.

The semantics for nested programs was introduced in [10]. Like the classic answer set semantics [7], the semantics for nested logic programs is defined in two steps: first for definite nested logic programs and after for general nested logic programs (programs which contain negation as failure).

▶ **Definition 1.** [10] Let M be a set of literals. M satisfies a definite nested formula A (denoted by $M \models A$), recursively as follows:

- for elementary A, $M \models A$ if $A \in M$ or $A = \top$
- $M \models A \wedge B$ if $M \models A$ and $M \models B$
- $M \models A \vee B$ if $M \models A$ or $M \models B$

▶ **Definition 2.** [10] Let P be a definite nested logic program. A set of literals M is closed under P if, $\forall r \in P$ such that $r = H \leftarrow B$, $M \models H$ whenever $M \models B$.

▶ **Definition 3.** [10] Let M be a set of literals and P a definite nested logic program. M is called an answer set for P if M is minimal among the consistent sets of literals closed under P.

In order to manage the negation as failure in nested logic programs, a syntactic reduction for nested logic programs was defined.

▶ **Definition 4.** [10] The reduction of a nested formula with respect to a set of literals M is recursively defined as follows:

- for elementary A, $A^M = A$
- $(A \wedge B)^M = A^M \wedge B^M$
- $(A \vee B)^M = A^M \vee B^M$
- $(not\ A)^M = \begin{cases} \bot, & \text{if } M \models A^M \\ \top, & \text{otherwise} \end{cases}$
- $(H \leftarrow B)^M = H^M \leftarrow B^M$

▶ **Definition 5.** [10] The reduct of a nested logic program P^M with respect to a set of literals M is defined as follows:

- $P^M = \{(H \leftarrow B)^M \mid H \leftarrow B \in P\}$

Please observe that P^M is a definite nested logic program. Hence, the following definition follows from the answer set definition.

▶ **Definition 6.** [10] Let P be a nested logic program and M be a set of literals. M is an answer set of P if it is an answer set of P^M.

Table 1 Examples of possibilistic rules captured by the syntax of possibilistic nested programs.

Syntax	Rule Type
$\alpha : a \wedge not\ b \leftarrow p \wedge not\ (\neg q \vee r).$	*possibilistic nested rule*
$\alpha : a \vee b \leftarrow c \wedge not\ \neg e.$	*possibilistic disjunctive rule* [14]
$\alpha : a \leftarrow c \wedge not\ d.$	*possibilistic normal rule* [13]

3 Possibilistic Nested Logic Programs

In this section, the general syntax and semantics for possibilistic nested logic programs will be presented. We will show that the semantics of the possibilistic nested logic programs generalizes the logic programming semantics of both the nested logic programs and the possibilistic disjunctive logic programs (the particular case of possibilistic normal logic programs is also considered). In order to define a process for computing the possibilistic answer sets of a possibilistic nested logic program some transformations between possibilistic programs are formalized.

The syntax of the possibilistic nested logic programs is based on the standard syntax of nested logic programs.

3.1 Syntax

We start by defining some concepts for managing the possibilistic values of a possibilistic knowledge base. We want to point out that in the whole document only finite lattices are considered.

A *possibilistic atom* is a pair $p = (a, q) \in \mathcal{A} \times \mathcal{Q}$, in which $\mathcal{A}$ is a finite set of atoms and $(\mathcal{Q}, \leq)$ is a lattice. The projection $*$ to a possibilistic atom p is defined as follows: $p^* = a$. Also given a set of possibilistic atoms S, $*$ over S is defined as follows: $S^* = \{p^* | p \in S\}$.

Let $(\mathcal{Q}, \leq)$ be a lattice. A possibilistic nested rule r is of the form:

$$\alpha : A \leftarrow B$$

in which $\alpha \in \mathcal{Q}$ and $A \leftarrow B$ is a nested rule. The projection $*$ for a possibilistic nested rule is $r^* = A \leftarrow B$. On the other hand, the projection n for a possibilistic nested rule is $n(r) = \alpha$. This projection denotes the degree of necessity captured by the certainty level of the information described by r. A possibilistic nested constraint c is of the form:

$$\top_{\mathcal{Q}} : \ \leftarrow B$$

in which $\top_{\mathcal{Q}}$ is the top of the lattice $(\mathcal{Q}, \leq)$ and $\leftarrow B$ is a nested constraint as defined in the background section. The projection $*$ for a possibilistic nested constraint c is: $c^* = \ \leftarrow B$.

A possibilistic nested program P is a tuple of the form $\langle(\mathcal{Q}, \leq), N\rangle$, in which N is a finite set of possibilistic nested rules and possibilistic nested constraints. The generalization of $*$ over P is as follows: $P^* = \{r^* | r \in N\}$. If N^* is a set of nested definite rules, P is called a possibilistic nested definite logic program. Different formula combinations lead to different logic rules as shown in Table 1.

We illustrate a possibilistic nested program with an example from the dementia domain (simplified due to space reasons). A summary of the clinical guidelines which are used in

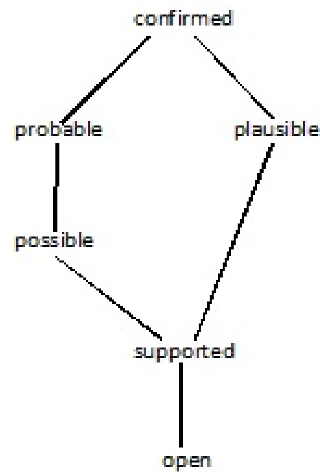


Figure 1 Graph representation of a lattice.

the dementia example given here can be found in [15] and includes [2]. We use the following abbreviations:

$$\begin{aligned} AD &= \textit{Alzheimer's disease} \\ DLB &= \textit{Lewy body type of dementia} \\ VaD &= \textit{Vascular dementia} \\ epiMem &= \textit{Episodic memory dysfunction} \\ fluctCog &= \textit{Fluctuating cognition} \\ fn &= \textit{Focal neurological signs} \\ prog &= \textit{Progressive course} \\ radVasc &= \textit{Radiology exam shows vascular signs} \\ slow &= \textit{Slow, gradual onset} \\ extraPyr &= \textit{Extrapyramidal symptoms} \\ visHall &= \textit{Visual hallucinations} \end{aligned}$$

We extract the following labels describing different levels of uncertainty of assessments from the clinical guidelines: $\mathcal{Q} := \{confirmed, probable, possible, plausible, supported, open\}$. To describe their relationships, let $<$ be a partial order such that the following set of relations holds: $\{confirmed > probable$, $probable > possible$, $confirmed > plausible$, $plausible > supported$, $possible > supported$, $supported > open\}$, see Figure 1. Given $x, y \in \mathcal{Q}$, the relation $x > y$ means that y is less certain than x.

▶ **Example 7.** The following clauses are included in our possibilistic nested logic program:

1. **probable**: $VaD \leftarrow fn \wedge radVasc \wedge$ not $(AD \vee DLB)$
2. **probable**: $DLB \leftarrow extraPyr \wedge visHall \wedge not\ fn$
3. **probable**: $DLB \leftarrow fluctCog \wedge visHall \wedge not\ fn$
4. **probable**: $DLB \leftarrow fluctCog \wedge extraPyr \wedge not\ fn$
5. **probable**: $VaD \wedge DLB \leftarrow fn \wedge radVasc \wedge extraPyr \wedge fluctCog$

6. **possible**: $VaD \wedge DLB \leftarrow fn \wedge fluctCog$
7. **possible**: $VaD \wedge AD \leftarrow fn \wedge slow \wedge prog \wedge epiMem$
8. **possible**: $VaD \wedge AD \leftarrow radVasc \wedge slow \wedge prog \wedge epiMem$
9. **possible**: $DLB \wedge AD \leftarrow fluctCog \wedge slow \wedge prog \wedge epiMem$
10. **possible**: $DLB \wedge AD \leftarrow extraPyr \wedge slow \wedge prog \wedge epiMem$
11. **possible**: $DLB \wedge AD \leftarrow visHall \wedge slow \wedge prog \wedge epiMem$
12. **possible**: $DLB \leftarrow fluctCog$
13. **possible**: $DLB \leftarrow visHall$
14. **possible**: $DLB \leftarrow extraPyr$
15. **possible**: $VaD \leftarrow fn$
16. **possible**: $VaD \leftarrow radVasc$
17. **supported**: $VaD \leftarrow fluctCog$
18. **plausible**: $VaD \leftarrow fn$
19. **probable**: $AD \leftarrow slow \wedge prog \wedge epiMem \wedge \ not\ (VaD \vee DLB)$

A problem in the dementia domain is that a large number of symptoms are overlapping between diseases. In addition, it is common to have more than one disease causing dementia in old age and in later stages of the disease progression (comorbidity). Typically, formal representations do not support this kind of complexity of a differential diagnostic process. The advantage of applying possibilistic nested rules is that it provides a transparent method to capture the different ways to interpret a set of findings, including potential comorbidity. Transparency is highly desirable in a knowledge modeling situation where medical domain experts are responsible for the content. Our example exemplify this, showing that one of two possible medical conditions may be present, or both.

3.2 Possibilistic Nested Logic Semantics

In order to define the semantics of the possibilistic nested logic programs, we introduce some basic concepts with respect to sets of possibilistic atoms.

Given a finite set of atoms $\mathcal{A}$, a lattice ($\mathcal{Q}$,$\leq$) and a the function *Cardinality* which returns the cardinality of a set:

$$PS = \{S | S \in 2^{\mathcal{A} \times \mathcal{Q}} \text{ and } \forall x \in \mathcal{A}, Cardinality(\{(x, \alpha) | (x, \alpha) \in S\}) \leq 1\}$$

Observe that every $S \in PS$ is a set of possibilistic atoms where every atom $x \in \mathcal{A}$ at most occurs one time in S.

▶ **Definition 8.** [14] Let $\mathcal{A}$ be a finite set of atoms and ($\mathcal{Q}$,$\leq$) be a lattice. $\forall A, B \in \mathcal{PS}$, we define

$$A \sqcap B = \{(x, \mathcal{GLB}(\{\alpha, \beta\})) | (x, \alpha) \in A \wedge (x, \beta) \in B\}.$$

$$A \sqcup B = \{(x, \alpha) | (x, \alpha) \in A \ and\ x \notin B^*\} \cup \{(x, \alpha) | x \notin A^* \ and\ (x, \alpha) \in B\} \cup \{(x, \mathcal{LUB}(\{\alpha, \beta\})) | (x, \alpha) \in A \text{ and } (x, \beta) \in B\}.$$

$$A \sqsubseteq B \iff A^* \subseteq B^*, \text{ and } \forall x, \alpha, \beta, (x, \alpha) \in A \wedge (x, \beta) \in B \text{ then } \alpha \leq \beta.$$

Before moving on, let us define the concept of *i-greatest set w.r.t.* $\mathcal{PS}$ as follows: Given $M \in \mathcal{PS}$, M is an *i-greatest set* in $\mathcal{PS}$ iff $\nexists M' \in \mathcal{PS}$ such that $M \sqsubseteq M'$. For instance, let $\mathcal{PS} = \{\{\{(a, 2), (b, 1)\}, \{(a, 2), (b, 2)\}\}$. One can see that $\mathcal{PS}$ has one i-greatest sets: $\{(a, 2), (b, 2)\}$.

Similar to the definition of answer set semantics for nested logic programs, the possibilistic answer set semantics for possibilistic nested logic programs is defined in terms of a syntactic reduction.

▶ **Definition 9** (Reduction P_M). Let $P = \langle(\mathcal{Q}, \leq), N\rangle$ be a possibilistic nested logic program, M be a set of atoms. P reduced by M is the following possibilistic definite nested logic program:

$$P_M := \{\alpha : (A \leftarrow B)^M | \alpha : A \leftarrow B \in N \text{ and } M \text{ is closed under } (A \leftarrow B)^M\}$$

Observe that the reduction $(A \leftarrow B)^M$ is according to Definition 4 and P_M is a possibilistic definite nested logic programs.

Now by considering the inference of possibilistic logic ($\vdash_{PL}$) and the reduction P_M, the inference relation $\Vdash_{PL}$ is defined as follows:

▶ **Definition 10.** Let $P = \langle(\mathcal{Q}, \leq), N\rangle$ be a possibilistic nested logic program and $M \in \mathcal{PS}$. We write $P \Vdash_{PL} M$ when M^* is an answer set of P^* and $P_{M^*} \vdash_{PL} M$.

Observe that the inference relation $\Vdash_{PL}$ is considering the standard definition of answer sets for nested logic programs (Definition 6). In particular, $\Vdash_{PL}$ is identifying sets of possibilistic atoms which satisfy P. However, not all these sets are optimal in the sense of necessity-values of a possibilistic theory. Hence, in order to define the possibilistic answer sets of a possibilistic nested logic programs we consider the idea of an *i-greatest set*.

▶ **Definition 11.** Let $P = \langle(\mathcal{Q}, \leq), N\rangle$ be a possibilistic nested logic program and M be a set of possibilistic atoms. M is a possibilistic answer set of P iff M is an i-greatest set in $\mathcal{PS}$ such that $P \Vdash_{PL} M$. $NSEM(P)$ denotes the set of possibilistic answer sets of P.

In order to illustrate the definition of answer sets for possibilistic nested logic programs, let us consider a subset of possibilistic nested rules which were introduced in Example 7.

▶ **Example 12.** Let $P = \langle(\mathcal{Q}, \leq), N\rangle$ be a possibilistic nested logic program in which $(\mathcal{Q}, \leq)$ is the lattice introduced in Example 7 and N is the following set of possibilistic nested rules:

$$\begin{array}{ll}
\textit{confirmed}: & \textit{fn} \leftarrow \top \\
\textit{confirmed}: & \textit{radVasc} \leftarrow \top \\
\textit{confirmed}: & \textit{extraPyr} \leftarrow \top \\
\textit{confirmed}: & \textit{fluctCog} \leftarrow \top \\
\textit{probable}: & \textit{VaD} \wedge \textit{DLB} \leftarrow \textit{fn} \wedge \textit{radVasc} \wedge \\
 & \textit{extraPyr} \wedge \textit{fluctCog} \\
\textit{possible}: & \textit{DLB} \leftarrow \textit{extraPyr} \\
\textit{probable}: & \textit{VaD} \leftarrow \textit{fn} \wedge \textit{radVasc} \wedge \\
 & \textit{not}\ (\textit{AD} \vee \textit{DLB})
\end{array}$$

In order to infer the answer sets of P, the first step is to find, the answer set of P^*. It is not hard to see that P^* has only one answer set which is $M = \{\textit{fn}, \textit{radVasc}, \textit{extraPyr}, \textit{fluctCog}, \textit{DLB}, \textit{VaD}\}$. Now, one can see that P_M is:

$$\begin{array}{ll}
\textit{confirmed}: & \textit{fn} \leftarrow \top \\
\textit{confirmed}: & \textit{radVasc} \leftarrow \top \\
\textit{confirmed}: & \textit{extraPyr} \leftarrow \top \\
\textit{confirmed}: & \textit{fluctCog} \leftarrow \top \\
\textit{probable}: & \textit{VaD} \wedge \textit{DLB} \leftarrow \textit{fn} \wedge \textit{radVasc} \wedge \\
 & \textit{extraPyr} \wedge \textit{fluctCog} \\
\textit{possible}: & \textit{DLB} \leftarrow \textit{extraPyr}
\end{array}$$

Observe that the possibilistic nested rule $r = probable : VaD \leftarrow fn \wedge radVasc \wedge not\ (AD \vee DLB)$ was removed because $(r^*)^M$ is not closed under M. Now let us consider $M_1 = \{(fn, confirmed), (radVasc, confirmed), (extraPyr, confirmed), (fluctCog, confirmed), (DLB, probable), (VaD, probable)\}$ and $M_2 = \{(fn, confirmed), (radVasc, confirmed), (extraPyr, confirmed), (fluctCog, confirmed), (DLB, possible), (VaD, probable)\}$.

One can see that $P_M \vdash_{PL} M_1$ and $P_M \vdash_{PL} M_2$. Since $M = M_1^* = M_2^*$, hence both M_1^* and M_2^* are answer sets of P^*. Therefore $P_M \Vdash_{PL} M_1$ and $P_M \Vdash_{PL} M_2$. This means that both M_1 and M_2 are two potential sets to be answer sets of P. Observe that $M_2 \sqsubseteq M_1$, therefore M_2 is not an i-greatest set. One can see that M_1 is an i-greatest set, therefore M_1 is the unique possibilistic answer set of P.

An obvious property of the logic programming semantics of the possibilistic nested logic programs is that it generalizes the logic programming semantics of nested logic programs

▶ Proposition 1. Let $P = \langle(\mathcal{Q}, \leq), N\rangle$ be a possibilistic nested logic program. If M is a possibilistic answer set of P then M^* is an answer set of P^*.

In the family of possibilistic logic programs, the approach presented in this paper generalizes the approaches presented in [13] and [14].

Let us formalize the relationship between the nested possibilistic semantics and the possibilistic stable semantics. The last one was introduced by [13].

▶ Proposition 2. Let $P = \langle(\mathcal{Q}, \leq), N\rangle$ be a possibilistic nested logic program such that for all $r \in N$, $r = \alpha : A_0 \leftarrow A_1 \wedge \cdots \wedge A_j \wedge\ not\ A_{j+1} \wedge \cdots \wedge not\ A_n$, $\mathcal{L}_{N^*}$ has no extended atoms and $(\mathcal{Q}, \leq)$ is a total ordered set. If M is a consistent possibilistic answer set of P then M is a possibilistic stable model according to the definition from [13].

Now, let us show that the possibilistic semantics for possibilistic nested logic programs generalizes the semantics of possibilistic disjunctive logic programs.

▶ Proposition 3. Let $P = \langle(\mathcal{Q}, \leq), N\rangle$ be a possibilistic nested logic program such that for all $r \in N$, $r = \alpha : A_0 \vee \cdots \vee A_m \leftarrow A_{m+1} \wedge \ldots A_j \wedge not\ A_{j+1} \wedge \cdots \wedge\ not\ A_n$ in which $A_i (0 \leq i \leq n)$ are literals. If M is a consistent possibilistic answer set of P then M is a possibilistic answer set according to the definition from [14].

It is known that the answer set semantics for nested logic programs is computable [10]. Indeed, one can find solvers of nested logic programs [19]. On the other hand, the possibilistic inference of possibilistic logic is complete and sound by a possibilistic extended version of the classical resolution rule [6]. Hence, it is not difficult to define an algorithm for computing the possibilistic answer sets of a possibilistic nested logic program.

A common strategy for computing the answer set of a nested logic program is to translate the nested logic programs into disjunctive ones. Hence, the answer sets of the nested logic programs are characterized by the answer sets of disjunctive logic programming systems. This strategy can be also applied for computing the answer sets of possibilistic nested logic programs via possibilistic disjunctive logic programs.

By lack of space, we omit the details of the transformation of any possibilistic nested logic program into a possibilistic logic program. The details of this transformation will be presented in the long version of this paper. In the following theorem, it is assumed that there is a transformation of any possibilistic nested logic program into a possibilistic disjunctive logic program.

▶ **Theorem 13.** *Let $P = \langle(\mathcal{Q}, \leq), N\rangle$ be a possibilistic nested logic program and P' a possibilistic disjunctive logic program obtained by transforming P. If M' is an answer set of P' then $M = \{(a, \alpha) | (a, \alpha) \in M'$ and $a \in M'^* \cap \mathcal{L}_{P^*}\}$ is answer set of P.*

4 Conclusions and Future Work

In the logic programming literature, one can find different approaches for expressing uncertain information [8, 13, 4, 1, 21, 14, 5]; however, most of them define syntactic restriction to their specification languages. Against this background, we introduce the class of possibilistic nested logic programs. The syntax and semantics of these programs generalize previous works in the paradigm of Answer Set Programming plus Possibilistic Logic (Proposition 2, Proposition 3). Moreover, our approach generalizes the frame of nested logic programs (Proposition 1). We show that the semantics of the possibilistic nested programs can be computed by transforming possibilistic nested logic programs into possibilistic disjunctive logic programs (Theorem 13).

In the long version of this paper, we will present a process for transforming a possibilistic nested logic program into a possibilistic disjunctive logic program. In this process, we will identify the class of *possibilistic generalized disjunctive logic programs* which is a subclass (syntactically speaking) of the possibilistic nested logic programs. Let us observe that the class of possibilistic generalized disjunctive logic programs is a class of possibilistic programs which is interesting by itself due to this class of logic programs is the possibilistic extension of the generalized disjunctive logic programs explored in [9].

To the best of our knowledge, the approach presented in this paper is the first work to attend to manage uncertain information with no-syntactic restrictions in its rules. It is worth mentioning that the possibilistic nested logic programs combine both non-monotonic reasoning and reasoning under uncertainty in a single framework.

Since the uncertain information in possibilistic nested logic programs can be captured by partially ordered sets, the possibilistic nested programs define a suitable approach for capturing qualitative information. In particular, we have illustrated that possibilistic nested logic programs are expressive enough for capturing ambiguous and uncertain knowledge content in medical guidelines. The approach has the potential to provide medical experts, who are usually not experts in knowledge representation, with a formal framework that is transparent and intuitive for knowledge modeling.

In our future, we will explore practical algorithms for implementing a solver for possibilistic nested logic programs. It is worth mentioning that there already exist solvers of nested logic programs [19]; hence, a solver for nested logic programs can be taken as a starting point for a solver for possibilistic nested logic programs. The approach described in this paper will be evaluated in practical knowledge modeling and diagnostic situations involving medical professionals as part of the ACKTUS project [11].

5 Acknowledgements

This research has been supported by VINNOVA (The Swedish Governmental Agency for Innovation Systems) and the Swedish Brain Power.

References

1 Teresa Alsinet, Carlos Iván Chesñevar, Lluis Godo, and Guillermo Ricardo Simari. A logic programming framework for possibilistic argumentation: Formalization and logical properties. *Fuzzy Sets and Systems*, 159(10):1208–1228, 2008.

2 American Psychiatric Association. *Diagnostic and Statistical Manual of Mental Disorders DSM-IV-TR Fourth Edition (Text Revision)*. Amer Psychiatric Pub, 4th edition, 2000.

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

4 Chitta Baral, Michael Gelfond, and J. Nelson Rushton. Probabilistic reasoning with answer sets. *TPLP*, 9(1):57–144, 2009.

5 Kim Bauters, Steven Schockaert, Martine De Cock, and Dirk Vermeir. Weak and strong disjunction in possibilistic asp. In *SUM*, volume 6929 of *Lecture Notes in Computer Science*, pages 475–488. Springer, 2011.

6 Didier Dubois, Jérôme Lang, and Henri Prade. Possibilistic logic. In Dov Gabbay, Christopher J. Hogger, and J. A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming, Volume 3: Nonmonotonic Reasoning and Uncertain Reasoning*, pages 439–513. Oxford University Press, Oxford, 1994.

7 Michael Gelfond and Vladimir Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, 9:365–385, 1991.

8 Michael Kifer and V. S. Subrahmanian. Theory of generalized annotated logic programming and its applications. *J. Log. Program.*, 12(3&4):335–367, 1992.

9 V. Lifschitz. *Principles of Knowledge Representation*, chapter Foundations of Logic Programming, pages 69–128. CSLI Publications, 1996.

10 Vladimir Lifschitz, Lappoon R. Tang, and Hudson Turner. Nested expressions in logic programs. *Annals of Mathematics and Artificial Intelligence*, 25(3-4):369–389, 1999.

11 Helena Lindgren and Peter Winnberg. Evaluation of a semantic web application for collaborative knowledge building in the dementia domain. In *eHealth*, pages 62–69, 2010.

12 Peter Lucas. Symbolic diagnosis and its formalisation. *The Knowledge Engineering Review*, 12:109–146, 1997.

13 Pascal Nicolas, Laurent Garcia, Igor Stéphan, and Claire Lefèvre. Possibilistic Uncertainty Handling for Answer Set Programming. *Annals of Mathematics and Artificial Intelligence*, 47(1-2):139–181, 2006.

14 Juan Carlos Nieves, Mauricio Osorio, and Ulises Cortés. Semantics for Possibilsitic Disjuntive Programs. *Theory and Practice of Logic Programming*, Available on doi: 10.1017/S1471068411000408, 2011.

15 J O'Brien, D Ames, and A Burns, editors. *Dementia*. Arnold, 2000.

16 Mauricio Osorio, Juan Antonio Navarro Pérez, and José Arrazola. Applications of intuitionistic logic in answer set programming. *TPLP*, 4(3):325–354, 2004.

17 David Pearce. Stable inference as intuitionistic validity. *J. Log. Program.*, 38(1):79–91, 1999.

18 Henri Prade. Advances in data management. In *Current Research Trends in Possibilistic Logic: Multiple Agent Reasoning, Preference Representation, and Uncertain Databases*. Springer Berlin / Heidelberg, 2009.

19 Vladimir Sarsakov, Torsten Schaub, Hans Tompits, and Stefan Woltran. nlp: A compiler for nested logic programming. In *LPNMR*, Lecture Notes in Computer Science, pages 361–364. Springer, 2004.

20 David Silverman. *Interpreting Qualitative Data*. SAGE Publications, 2006.

21 Davy Van-Nieuwenborgh, Martine De Cock, and Dirk Vermeir. An introduction to fuzzy answer set programming. *Ann. Math. Artif. Intell.*, 50(3-4):363–388, 2007.

A Tarskian Informal Semantics for Answer Set Programming*

Marc Denecker[1], Yuliya Lierler[2], Miroslaw Truszczynski[2], and Joost Vennekens[3]

1 **Department of Computer Science, K.U. Leuven**
3001 Heverlee, Belgium
`marc.denecker@cs.kuleuven.be`

2 **Department of Computer Science, University of Kentucky**
Lexington, KY 40506-0633, USA
`yuliya|mirek@cs.uky.edu`

3 **Campus De Nayer | Lessius Mechelen | K.U. Leuven**
2860 Sint-Katelijne-Waver, Belgium
`joost.vennekens@cs.kuleuven.be`

Abstract

In their seminal papers on stable model semantics, Gelfond and Lifschitz introduced ASP by casting programs as epistemic theories, in which rules represent statements about the knowledge of a rational agent. To the best of our knowledge, theirs is still the only published systematic account of the intuitive meaning of rules and programs under the stable semantics. In current ASP practice, however, we find numerous applications in which rational agents no longer seem to play any role. Therefore, we propose here an alternative explanation of the intuitive meaning of ASP programs, in which they are not viewed as statements about an agent's beliefs, but as objective statements about the world. We argue that this view is more natural for a large part of current ASP practice, in particular the so-called Generate-Define-Test programs.

1998 ACM Subject Classification D.1.6 Logic Programming, I.2.4 Knowledge Representation Formalisms and Methods

Keywords and phrases Answer set programming, informal semantics, generate-define-test

Digital Object Identifier 10.4230/LIPIcs.ICLP.2012.277

1 Introduction

The key postulate of declarative programing is that programs *reflect* the way information about a domain of discourse is described in natural language. The syntax must align with linguistic patterns we use and the formal semantics must capture the way we understand them. To put it differently, a declarative programming formalism (logic) must have an *informal semantics*, an intuitive and precise link between the formal syntax and semantics of the logic, and informal intended meanings of natural language expressions describing the "real world". This informal semantics explains what programs mean or, in other words, provides programs with an informal but precise natural language reading.

Having an informal semantics is important to a declarative programming formalism. It facilitates effective coding by offering intuitions to guide the programmer, and provides a

* The first and the fourth author are supported by Research Foundation-Flanders (FWO-Vlaanderen) and by GOA 2003/08 "Inductive Knowledge Bases". The second author was supported by a CRA/NSF 2010 Computing Innovation Fellowship. The third author was supported by the NSF grant IIS-0913459.

© Marc Denecker, Yuliya Lierler, Miroslaw Truszczynski, and Joost Vennekens;
licensed under Creative Commons License ND

basis for the programming methodology. It promotes understanding of code and helps in teaching how to program. It suggests extensions of the logic to facilitate expressing new types of knowledge, and it helps explain the relationship to other logics. The postulate is essential anywhere the emphasis on declarativeness is paramount, in particular, in addition to declarative programming, also in knowledge representation and database query languages and, more generally, in all contexts where we need to think how information about a domain of discourse has been or is to be encoded in a logic.

First-order (FO) logic has a clear informal semantics aligned with the classical *Tarskian* formal semantics of the logic. Indeed, the constructors of the FO language directly correspond to natural language connectives and expressions "for all" and "there is". The formal Tarskian semantics of these syntactic constructors is given by *interpretations* (*structures*), which are abstract mathematical representations of informally understood "possible objective state of affairs," and it literally reflects the informal understanding of the natural language connectives and quantifying expressions. It is that informal semantics that makes FO sentences legible and their intended meaning clear, and is largely responsible for the widespread use of FO logic in declarative programming, knowledge representation and database query languages.

Our main goal in this work is to analyze the role of informal semantics in the development of answer set programming (ASP) and its effective use. The key step is to clarify what informal semantics we have in mind. According to the intuitions Gelfond and Lifschitz exploited when introducing the stable-model (answer-set) semantics [10, 11], a program is a formal representation of a set of *epistemic* propositions believed by a rational introspective agent, and stable models of the program represent that agent's *belief sets*. This *epistemic* informal semantics linked ASP to autoepistemic logic by Moore [16] and default logic by Reiter [20], and supported applications in nonmonotonic reasoning. However, the epistemic perspective does not seem to be relevant to the way ASP is predominantly used now, as a formalism for modeling search problems [14, 17]. We argue that for such use of ASP a *Tarskian* informal semantics, not unlike the one for the FO logic, fits the bill better.

Interestingly, while the Tarskian informal semantics of ASP seems to have been implicitly followed by a vast majority of ASP users, it has never been explicitly identified or analyzed. We do so in this paper. We describe that informal semantics and show how it explains the way ASP developed and how it is intimately related to the currently dominating form of ASP, the *generate-define-test* (GDT) ASP [12]. We point out how the Tarskian informal semantics connects GDT ASP with the logic FO(ID). We argue that taking the Tarskian informal semantics seriously strongly suggests that the language of GDT ASP can be streamlined while in the same time generalizing the current one.

We present the Tarskian informal semantics for ASP in the context of a more general formalism, which we introduce first. We call it *first-order answer set programming* of ASP-FO for short. ASP-FO can be viewed as a modular first-order generalization of GDT ASP with unrestricted interpretations as models, with open and closed domains, non-Herbrand functions, and FO constraints and rule bodies. It is closely connected to the logic FO(ID) [3, 6] and has formal connections to the equilibrium logic [18]. ASP-FO generalizes GDT ASP and so the informal semantics we develop for ASP-FO applies to GDT ASP, too.

2 Generate-Define-Test methodology

GDT is an effective methodology to encode search problems in ASP. In GDT, a programmer conceives the problem as consisting of three parts: GENERATE, DEFINE and TEST [12]. The role of GENERATE is to *generate the search space*. Nowadays this is often encoded by a set of

choice rules:

$$\{A\} \leftarrow B_1, \ldots, B_n, \texttt{not } C_1, \ldots, \texttt{not } C_m, \tag{1}$$

where A, B_i and C_i are atoms. Such a rule states that atom A can be arbitrarily true or false, if the condition expressed by the rule's body holds. This condition may refer to other generated predicates, or to defined predicates. The DEFINE part is a set of definitions of some auxiliary predicates. Each definition is encoded by a group of rules

$$A \leftarrow B_1, \ldots, B_n, \texttt{not } C_1, \ldots, \texttt{not } C_m, \tag{2}$$

where A, B_i, C_j are atoms and A is the auxiliary predicate that is being defined. These rules describe how to derive the auxiliary predicates from the generated predicates or from other defined predicates, typically in a deterministic way. Finally, the TEST part eliminates generated answer sets that do not satisfy desired constraints. They are represented by constraint rules:

$$\leftarrow B_1, \ldots, B_n, \texttt{not } C_1, \ldots, \texttt{not } C_m, \tag{3}$$

A set of these three types of rules will be called a *GDT program*.

For instance, the GDT-program (4) below encodes the Hamiltonian cycle problem. The example illustrates that an ASP program conceived in the GDT way typically shows a rich internal structure.

GENERATE	$\{In(x,y)\} \leftarrow Edge(x,y).$	
DEFINE	$Node(V). \ \ldots \ Node(W).$	
	$Edge(V,V'). \ \ldots \ Edge(W,W').$	
	$T(x,y) \leftarrow In(x,y).$ $T(x,y) \leftarrow T(x,z), T(z,y).$	(4)
TEST	$\leftarrow In(x,y), In(x,z), y \neq z.$	
	$\leftarrow In(x,z), In(y,z), x \neq y.$	
	$\leftarrow Node(x), Node(y), \texttt{not } T(x,y).$	

Each of the three parts may again consist of independent components. For instance, TEST in the example above consists of three independent constraints; DEFINE contains separate definitions for three predicates *Node*, *Edge* and T. This internal structure exists in the mind of programmers, but is not explicit in ASP programs and often becomes apparent only when we investigate the dependencies between predicates. This motivates us to define a logic which does make the internal structure of a GDT-program explicit.

3 Concepts of Tarskian model semantics

A *vocabulary* Σ is a set of *predicate* and *function* symbols, each with a non-negative integer *arity*. Terms, formulas and sentences are defined as in FO.

An *interpretation* (or *structure*) $\mathfrak{A}$ of a vocabulary Σ is given by a non-empty set $dom(\mathfrak{A})$, the *domain* of $\mathfrak{A}$, and, for each symbol τ of Σ, a value $\tau^{\mathfrak{A}}$, the *interpretation* of τ. If τ is an n-ary function symbol, $\tau^{\mathfrak{A}}$ is an n-ary total function over $dom(\mathfrak{A})$. If τ is an n-ary predicate symbol, $\tau^{\mathfrak{A}}$ is an n-ary relation over $dom(\mathfrak{A})$. If $\mathfrak{A}$ is an interpretation of a vocabulary Σ, we call Σ the vocabulary of $\mathfrak{A}$ and write it as $\Sigma_{\mathfrak{A}}$. An interpretation of the empty vocabulary consists only of its domain.

If $\Sigma' \subseteq \Sigma_{\mathfrak{A}}$, we define the *projection* of $\mathfrak{A}$ on Σ', written $\mathfrak{A}|_{\Sigma'}$, to be the interpretation of Σ' with the same domain and the same interpretation of each symbol $\tau \in \Sigma'$ as $\mathfrak{A}$. We then also say that $\mathfrak{A}$ is an *extension* of its projection $\mathfrak{A}|_{\Sigma'}$.

Let $\mathfrak{A}$ and $\mathfrak{A}'$ be interpretations of the same vocabulary Σ, having the same domain, and assigning the same values to every function symbol in Σ. We say that $\mathfrak{A}$ is a *subinterpretation* of $\mathfrak{A}'$, written $\mathfrak{A} \subseteq \mathfrak{A}'$, if, for every predicate symbol P of Σ, the relation $P^{\mathfrak{A}}$ interpreting this predicate symbol in $\mathfrak{A}$ is a subset of the corresponding relation $P^{\mathfrak{A}'}$.

A variable assignment θ for an interpretation $\mathfrak{A}$ assigns to each variable v an element $\theta(v)$ in $dom(\mathfrak{A})$. When x is a variable and d an element of $dom(\mathfrak{A})$, we write $\theta[x:d]$ for a variable assignment that assigns d to x but is otherwise the same as θ. The interpretation $t^{\mathfrak{A},\theta}$ of a term in an interpretation $\mathfrak{A}$ under variable assignment θ is defined through the standard induction. As usual, we assume that $\wedge, \forall, \Rightarrow$ are defined in terms of $\neg, \vee$ and $\exists$.

▶ **Definition 1** (Satisfiability relation $\mathfrak{A}, \theta \models \varphi$)**.** Let φ be an FOL formula and $\mathfrak{A}$ a structure over a vocabulary containing all function and relation symbols in φ. We define $\mathfrak{A}, \theta \models \varphi$ by induction on the structure of φ:
- $\mathfrak{A}, \theta \models P(\bar{t})$ if $\bar{t}^{\,\mathfrak{A},\theta} \in P^{\mathfrak{A}}$;
- $\mathfrak{A}, \theta \models \psi \vee \phi$ if $\mathfrak{A}, \theta \models \psi$ or $\mathfrak{A}, \theta \models \phi$;
- $\mathfrak{A}, \theta \models \neg\psi$ if $\mathfrak{A}, \theta \not\models \psi$;
- $\mathfrak{A}, \theta \models \exists x\ \psi$ if for some $d \in dom(I)$, $\mathfrak{A}, \theta[x:d] \models \psi$.

When φ is a sentence (no free variables), then θ is irrelevant and we write $\mathfrak{A} \models \varphi$.

In a Tarskian model semantics, a structure represents a potential state of affairs. For a sentence φ and a structure $\mathfrak{A}$, $\mathfrak{A} \models \varphi$ formalizes that φ is **true** in the state of affairs as given by $\mathfrak{A}$. If all we know about the state of affairs is that φ is true in it, then a structure $\mathfrak{A}$ is a *possible* state of the world, or a *possible world*, if and only if $\mathfrak{A} \models \varphi$.

4 The logic ASP-FO

We introduce a modular form of ASP to represent the different kind of modules in GDT programs. We define a G-module, D-module and T-module.

▶ **Definition 2.** A *choice rule* is an expression of the form: $\forall \bar{x}\ (\{P(\bar{t})\} \leftarrow \varphi)$, where φ is an FO formula, $P(\bar{t})$ is an atom and $\bar{x}$ includes all free variables appearing in the rule. A G-*module* is a set of choice rules with the same predicate in their head.

▶ **Definition 3.** A D-*module* $\mathcal{D}$ is a pair $\langle \mathit{Ext}, \Pi \rangle$ where Ext is a set of predicates, called *defined* or *output predicates*, and Π is a set of rules of the form

$$\forall \bar{x}\ (P(\bar{t}) \leftarrow \varphi), \tag{5}$$

where $P(\bar{t})$ is an atom with $P \in \mathit{Ext}$, and φ is an FO formula with all its free variables amongst $\bar{x}$.

For a D-module $\mathcal{D}$, we denote the set of its defined predicate symbols by $\mathit{Ext}(\mathcal{D})$. We write $\mathit{Par}(\mathcal{D})$ for the set of all other symbols in Π. We call $\mathit{Par}(\mathcal{D})$ the set of *parameter* or *input symbols*. For a set of rules Π, we denote by $heads(\Pi)$, the set of all predicate symbols appearing in the head of a rule $r \in \Pi$. In the following we identify a D-module $\langle heads(\Pi), \Pi \rangle$ with Π.

▶ **Definition 4.** A T-*module* is an FO sentence.

▶ **Definition 5.** An *ASP-FO-theory* is a set of G-modules, D-modules and T-modules.

There is an obvious syntactical match between these language constructs and those used in ASP to express GENERATE, DEFINE and TEST modules. For instance, an ASP constraint (3) corresponds to the T-module– FO sentence: $\forall \bar{x}(\neg(B_1 \wedge \cdots \wedge B_n \wedge \neg C_1 \wedge \cdots \wedge \neg C_m))$, where $\bar{x}$ is the set of variables occurring in (3), and we identify normal rules (2) with the universal closure of $A \leftarrow B_1 \wedge \cdots \wedge B_n \wedge \neg C_1 \wedge \cdots \wedge \neg C_m$.

Note that an ASP-FO theory preserves the internal structure of the GENERATE, DEFINE and TEST parts. For instance, we may write the Hamiltonian cycle theory as:

$$\begin{array}{ll}
\text{GENERATE} & \{\forall x \forall y(\{In(x,y)\} \leftarrow Edge(x,y))\} \\
\hline
\text{DEFINE} & \{Vertex(V) \leftarrow \mathbf{t}, \ldots, Vertex(W) \leftarrow \mathbf{t}\} \\
& \{Edge(V,V') \leftarrow \mathbf{t}, \ldots, Edge(W,W') \leftarrow \mathbf{t}\} \\
& \left\{ \begin{array}{l} \forall x \forall y(T(x,y) \leftarrow In(x,y)) \\ \forall x \forall y(T(x,y) \leftarrow In(x,z) \wedge In(z,y)) \end{array} \right\} \\
\hline
\text{TEST} & \forall x \forall y \forall z \neg(In(x,y) \wedge In(x,z) \wedge y \neq z) \\
& \forall x \forall y \forall z \neg(In(x,z) \wedge In(y,z) \wedge x \neq y) \\
& \forall x \forall y(Vertex(x) \wedge Vertex(y) \Rightarrow T(x,y))
\end{array} \tag{6}$$

In defining the formal semantics of ASP-FO, we aim to ensure that three conditions are satisfied. First, the structures are to be viewed as possible worlds, i.e., they should represent possible states of affairs, not states of belief. We do not restrict to Herbrand interpretations. Second, to respect the modular structure of an ASP-FO theory, its semantics should be modular, that is, defined in terms of the semantics of its modules. We therefore simply define that a structure $\mathfrak{A}$ is a model of a ASP-FO theory T iff it is a model of each of its modules. In other words, an ASP-FO theory can be understood as a monotone conjunction of its modules. Third, as ASP-FO is to reflect the GDT methodology, ASP-FO theories resulting from GDT programs must have the same meaning.

The definition of satisfaction of a T-module, i.e., an FO sentence, is standard (Definition 1). It follows that ASP-FO is a conservative extension of FO.

The semantics for a D-module is a generalization of the stable semantics to arbitrary structures and to FO rule bodies. For reasons explained in the next section, we use the semantics that was introduced by Pelov et al. [19] and, in the way we follow here, by Vennekens et al. [22]. It uses a pair of interpretations to simulate the construction of the Gelfond-Lifschitz reduct.

▶ **Definition 6** (Satisfaction by pairs of interpretations)**.** Let φ be an FO formula, $\mathfrak{A}$ and $\mathfrak{B}$ interpretations of all symbols in φ having the same domain and assigning the same values to all function symbols, and let θ be a variable assignment. We define the relation $(\mathfrak{A}, \mathfrak{B}), \theta \models \varphi$ by induction on the structure of φ:

- $(\mathfrak{A}, \mathfrak{B}), \theta \models P(\bar{t})$ if $\mathfrak{A}, \theta \models P(\bar{t})$,
- $(\mathfrak{A}, \mathfrak{B}), \theta \models \neg\varphi$ if $(\mathfrak{B}, \mathfrak{A}), \theta \not\models \varphi$,
- $(\mathfrak{A}, \mathfrak{B}), \theta \models \varphi \vee \psi$ if $(\mathfrak{A}, \mathfrak{B}), \theta \models \varphi$ or $(\mathfrak{A}, \mathfrak{B}), \theta \models \psi$
- $(\mathfrak{A}, \mathfrak{B}), \theta \models \exists x \psi$ if for some $d \in dom(I)$, $(\mathfrak{A}, \mathfrak{B}), \theta[x : d] \models \psi$.

This truth assignment interprets positive occurrences of atoms in $\mathfrak{A}$, and negative occurrences in $\mathfrak{B}$. Indeed, (positive) atoms are interpreted in $\mathfrak{A}$, but every occurrence of $\neg$ switches the role of $\mathfrak{A}$ and $\mathfrak{B}$.

For two structures $\mathfrak{A}$ and $\mathfrak{B}$ that have the same domain and interpret disjoint vocabularies, $\mathfrak{A} \circ \mathfrak{B}$ denotes the structure that interprets the union of the vocabularies of $\mathfrak{A}$ and $\mathfrak{B}$, has the same domain as $\mathfrak{A}$ and $\mathfrak{B}$, and coincides with $\mathfrak{A}$ and $\mathfrak{B}$ on their respective vocabularies.

▶ **Definition 7** (Parameterized stable-model semantics)**.** For a D-module $\mathcal{D}$, an interpretation $\mathcal{M}$ of $Ext(\mathcal{D})$ is a *stable model* of $\mathcal{D}$ relative to an interpretation $\mathfrak{A}_p$ of $Par(\mathcal{D})$ if $\mathcal{M}$ is the least[1] of all interpretations $\mathfrak{A}$ of $Ext(\mathcal{D})$ that have the same domain as $\mathfrak{A}_p$, interpret function symbols in the same way as $\mathfrak{A}_p$ and for each rule $\forall \bar{x}\ (P(\bar{t}) \leftarrow \varphi)$ of $\mathcal{D}$ and each variable assignment θ, if $(\mathfrak{A}_p \circ \mathfrak{A}, \mathfrak{A}_p \circ \mathcal{M}), \theta \models \varphi$ then $\mathfrak{A}, \theta \models P(\bar{t})$.

This parameterized stable-model semantics generalizes the original one in three ways: it is parameterized, i.e., it builds stable models on top of a given interpretation of the parameter symbols; it handles FO bodies; and it works for arbitrary (not only Herbrand) interpretations.

▶ **Definition 8.** A structure $\mathfrak{A}$ is a *model* of a D-module $\mathcal{D}$ (notation $\mathfrak{A} \models \mathcal{D}$) if $\mathfrak{A}|_{Ext(\mathcal{D})}$ is a stable model of $\mathcal{D}$ relative to $\mathfrak{A}|_{Par(\mathcal{D})}$.

We now turn our attention to G-modules. We note that the point of a choice rule is to "open up" certain atoms $P(\bar{d})$ – to allow them to be true without forcing them to be true.

▶ **Definition 9.** A structure $\mathcal{M}$ is a model of a G-module $\mathcal{G}$ if for each variable assignment θ such that $\mathcal{M}, \theta \models P(\bar{x})$ there is a choice rule $\forall \bar{y}\ (\{P(\bar{t})\} \leftarrow \varphi)$ in $\mathcal{G}$ such that $\bar{t}^{\mathcal{M},\theta} = \bar{x}^{\theta}$ and $\mathcal{M}, \theta \models \varphi$.

A G-module can be translated to an equivalent singleton G-module, using a process similar to *predicate completion.* First, we note that any choice rule $\forall \bar{x}\ (\{P(\bar{t})\} \leftarrow \varphi)$ can be rewritten as $\forall \bar{y}\ (\{P(\bar{y})\} \leftarrow \exists \bar{x}(\bar{y} = \bar{t} \wedge \varphi))$. Next, any finite set of choice rules $\forall \bar{x}\ (\{P(\bar{x})\} \leftarrow \varphi_i)$ can be combined into a single choice rule $\forall \bar{x}\ (\{P(\bar{y})\} \leftarrow \varphi_1 \vee \cdots \vee \varphi_n))$. It is straightforward to show that these transformations are equivalence-preserving. Together with this result, the following theorem implies that each (finite) G-module is equivalent to an FO sentence. Thus, G-modules are redundant in ASP-FO, since they can be simulated by T-modules.

▶ **Theorem 10.** *An interpretation $\mathcal{M}$ satisfies a singleton* G*-module* $\{\forall \bar{x}\ (\{P(\bar{x})\} \leftarrow \varphi)\}$ *if and only if $\mathcal{M}$ satisfies* $\forall \bar{x}\ (P(\bar{x}) \Rightarrow \varphi)$*.*

For instance, the singleton G-module of the GENERATE part of (4) corresponds to the following FO sentence: $\forall x \forall y(In(x, y) \Rightarrow Edge(x, y))$.

ASP-FO is an open domain logic with uninterpreted function symbols. Logic programming and ASP often restrict the semantics to Herbrand interpretations only.

▶ **Definition 11.** The *Herbrand* module over a set σ of function symbols is the expression $\mathcal{H}(\sigma)$. We say that $\mathcal{M} \models \mathcal{H}(\sigma)$ if $dom(\mathcal{M})$ is the set of variable-free terms that can be built from σ and for each such term t, $t^{\mathcal{M}} = t$.

Herbrand modules are useful in applications with complete knowledge of the domain. By adding $\mathcal{H}(\sigma)$ for the set σ of all function symbols of Σ to an ASP-FO theory, we limit its semantics to Herbrand models of σ. By adding $\mathcal{H}(\sigma)$ for a strict subset σ of function symbols, the remaining function symbols behave as uninterpreted symbols and take arbitrary interpretation in the Herbrand universe consisting of the terms of σ. Herbrand modules can be expressed by means of D- and T-modules (as in the logic FO(ID) [3]). Thus, they are redundant.

Relationship with FO and ASP. ASP-FO is not only a conservative extension of FO but also of the basic ASP language of normal programs. Note that a set of normal rules can be seen as a D-module defining all predicates.

[1] The term "least" is understood with respect to the notion of subinterpretation defined earlier. One can show that such a least interpretation always exists.

▶ **Theorem 12.** *For a normal program* Π *over vocabulary* Σ*, a structure* $\mathfrak{A}$ *is a stable model of* Π *if and only if* $\mathfrak{A}$ *is a model of the ASP-FO theory* $\{(\Sigma_P, \Pi), \mathcal{H}(\Sigma_F)\}$*, where* Σ_P, Σ_F *is the set of all predicate and function symbols of* Σ*, respectively.*

This theorem allows us to represent an entire normal logic program as a single D-module (and an auxiliary Herbrand module). However, as stated before, what we would like to show is the equivalence of GDT-programs in ASP and the corresponding ASP-FO theories.

Let us now consider a GDT-program Π consisting of a set of choice rules of form (1), normal rules of form (2) and constraints of form (3). We define the *(positive) predicate dependency graph* of Π as the directed graph with all predicate symbols of Π as its vertices and with an edge from P to Q whenever P appears in the head of a rule and Q occurs positively in the body of that rule (i.e., in the scope of an even number of negations).

Without loss of generality we assume that each predicate of Π appears in the head of at least one of its rules. By $heads(\Pi)$ we denote the set of all predicate symbols appearing in the heads of the rules of the form (1) or (2) in Π. A partition $\Pi_0, \ldots, \Pi_n$ of Π is a *splitting*[2] of Π if:

- for each i, Π_i is either a singleton containing a constraint, the set of all choice rules for some predicate P, or a normal logic program;
- $heads(\Pi_i) \cap heads(\Pi_j) = \emptyset$ for $i \neq j$;
- for any strongly connected component S of the predicate dependency graph of Π, $S \subseteq heads(\Pi_i)$ for some i;
- for any predicate symbol P occurring in the head of some choice rule in Π there is no edge from P to P in the predicate dependency graph of Π.

We can identify each Π_i in a splitting with an ASP-FO module in the obvious way: a Π_i that consists of a constraint corresponds to a T-module, a Π_i consisting of choice rules corresponds to a G-module, and a Π_i consisting of normal rules corresponds to a D-module.

▶ **Theorem 13.** *For a GDT-program* Π*, if* $\Pi_0, \ldots, \Pi_n$ *is a splitting of* Π*, then an interpretation* $\mathcal{M}$ *is answer set of* Π *if and only if* $\mathcal{M}$ *is a model of* $\{\mathfrak{M}_0, \ldots, \mathfrak{M}_n, \mathcal{H}(\Sigma)\}$*, where each* $\mathfrak{M}_i$ *is the ASP-FO module corresponding to* Π_i*.*

For instance, the horizontal lines within GENERATE, DEFINE, and TEST parts of the Hamiltonian cycle program (4) identify a partition that satisfies the conditions of a splitting. Theorem 13 states that the answer sets of (4) coincide with models of the ASP-FO theory (6).

The practice of ASP demonstrates that the vast majority of GDT programs admit a splitting. Theorem 13 shows that ASP-FO (i) extends this fragment of ASP in a direct way, and (ii) interprets those ASP programs as the *monotone conjunction* of their components.

Theorem 13 fails to take into account three common extensions of the ASP language: aggregates (or weight expressions), disjunction in the head, and strong negation. Each of these limitations can be lifted (we do not discuss the details due to space restrictions).

Relation to FO(ID). A theory in FO(ID) is a set of FO sentences and *inductive definitions.*[3] These definitions are syntactically identical to D-modules of ASP-FO, but are interpreted under a two-valued parameterized variant of the well-founded semantics, rather than the parameterized stable-model semantics used in ASP-FO.

▶ **Definition 14.** A Σ-interpretation $\mathfrak{A}$ is a model of an FO(ID) definition Δ (notation $\mathfrak{A} \models \Delta$) if $\mathfrak{A}|_{Ext(\mathcal{D})}$ is the well-founded model of Δ relative to $\mathfrak{A}|_{Par(\mathcal{D})}$, as defined in [7].

[2] The conditions on splitting follow the requirements stated in the Symmetric Splitting Theorem in [9].
[3] Some versions of FO(ID) allow also boolean combinations of FO formulas and definitions [6].

Denecker and Ternovska [6] introduced the notion of a *total definition*. An FO(ID) definition is *total* if it has only two-valued well-founded models and hence expresses a total, deterministic function from $Par(\mathcal{D})$-interpretations to $Ext(\mathcal{D})$-interpretations. Syntactic conditions such as no negative occurrences of defined symbols in the bodies of rules (defining the class of *positive* definitions), predicate stratification and local stratification all guarantee that a definition is total. Thus, many definitions occurring in practice are total. For total definitions (D-modules), the well-founded and stable models coincide [19]. Consequently, the logics ASP-FO and FO(ID) restricted to total definitions (D-modules) coincide, too.

Looking back at the ASP-FO theory for the Hamiltonian circuit program, we see that all three of its D-modules are positive. Hence, it is equivalent to the FO(ID) theory of the same syntactic form.

Relation to the equilibrium logic first-order ASP. There is a formal connection between ASP-FO D-modules and extensions of ASP to the first-order setting based on equilibrium logic [18] and the operator SM [8]. We consider the latter two under the restriction to formulas representing rules of the form (5). In this case, the semantics coincide if the bodies of rules (5) have no nested occurrences of negation [21]. However, the two generalizations of ASP differ if nested occurrences are allowed. For instance, the D-module $\{P \leftarrow \neg\neg P\}$ has only $\emptyset$ as a model, while in these generalizations also $\{P\}$ is a model. More importantly though, they differ at the conceptual level. The logic ASP-FO directly extends FO. The equilibrium logic version of first-order ASP is based on the quantified logic HT that differs substantially from FO and, arguably, lacks its direct connection to everyday linguistic patterns.

5 Informal semantics of ASP-FO and the Generate-Define-Test methodology

A formal semantics is just a mathematical definition and therefore, by itself, it does not yet explain how expressions in a logic relate to the real world. For this, also an *informal semantics* is needed, i.e., an intuitive interpretation for the logic's syntactic and semantic objects. In this paper, we interpret an answer set as a Tarskian representation of a possible state of the world. The aim of this section is to investigate in detail how the connectives of ASP should be understood in this new perspective.

It is a common adage in knowledge representation that humans are only able to comprehend a large theory if its meaning is composed from the meanings of its components through a simple and natural semantic composition operator. The most basic composition operator is simple conjunction. It is the use of this operator that causes FO to be monotonic. The meaning of an ASP-FO theory is constructed from the meaning of its individual modules by precisely the same form of conjunction. This is in perfect agreement with our intuition of modules as imposing constraints on possible worlds, independently from each other. Whatever analysis remains to be done has then to be concerned with individual modules.

Informal semantics of T-modules/FO sentences. FO sentences express propositions about an objective world, not about beliefs, intentions, or other propositional attitudes. In Tarskian model semantics, a structure $\mathfrak{A}$ serves as a mathematical abstraction of an objective world. The recursive rules of the definition of truth of a sentence in $\mathfrak{A}$ (Definition 1) specify the formal semantics of FO simply by translating each formal connective into an informal one: $\wedge$ into the natural language “and”, $\vee$ into “or”, etc. Iterated application of these rules translates an FO sentence into a natural language sentence that accurately captures its meaning.

The existence of this informal semantics does not mean that each FO sentence has a self-evident meaning. Sentences with three or more nestings of quantifiers are hard to understand. The material implication $\psi \Rightarrow \varphi$ also may cause difficulties. Nevertheless, for a core fragment of FO, sentences have an accurate and reliable informal semantics. For example, given the informal meaning of the symbols *Node* and T in the Hamiltonian circuit example, the informal semantics of

$$\forall x \forall y (Node(x) \wedge Node(y) \Rightarrow T(x, y))$$

is the proposition that each node can be reached from every other one.

Informal semantics of choice rules. Choice rules in ASP are often explained in a computational way, as generators of the search space. Here we propose a declarative interpretation. The set of ASP choice rules for predicate P

$$\{P(\bar{t}_1)\} \leftarrow \varphi_1. \quad \dots \quad \{P(\bar{t}_n)\} \leftarrow \varphi_n.$$

constitutes a G-module in ASP-FO which can be further translated in

$$\forall x (P(\bar{x}) \;\Rightarrow\; (\bar{x} = \bar{t}_1 \wedge \varphi_1) \vee \dots \vee (\bar{x} = \bar{t}_n \wedge \varphi_n))$$

In the Tarskian possible-world perspective, this sentence says that P is universally false with exceptions explicitly listed in the consequent of the implication. In other words, a G-module expresses the *local closed world assumption* (LCWA) on P, together with an exception mechanism to relax this LCWA and reinstall the open world assumption (OWA) on certain parts of the domain. For instance, $\forall x \forall y (In(x, y) \Rightarrow Edge(x, y))$, the ASP-FO image of the ASP choice rule $\{In(x, y)\} \leftarrow Edge(x, y)$, states that $In(x, y)$ is false except when $Edge(x, y)$ is true, in which case $In(x, y)$ might be either true or false.

This analysis of ASP choice rule modules as FO sentences shows that logical connectives in choice rule bodies, *including* negation, have their standard FO meaning. However, the meaning of a choice module as a whole is not composed from the meaning of its individual rules by monotone conjunction. Instead, adding a rule to a module corresponds to adding a disjunct to its FO axiom. Hence, the underlying composition operator of this sort of module is actually anti-monotonic: the module becomes *weaker* with each rule added. This agrees with the role of a choice rule as expressing an *exception* to the LCWA imposed by the module. The more exceptions there are, the weaker this LCWA.

Informal semantics of D-modules. In the GDT methodology, D-modules serve to *define* a set of auxiliary predicates and do so using a *rule-based, potentially recursive* syntax [12]. Even though current ASP practice tacitly assumes that the stable-model semantics is a correct semantics for such modules, this is actually far from trivial. As far as we know, this issue has not yet been addressed in the literature. Our results allow us to present the following argument to fill this gap.

Informal rule-based definitions (such as Definition 1) abound in mathematics. They express a precise, objective form of informal knowledge. A formal rule-based definition construct should match with the informal one. The three most common forms of definitions in formal sciences are non-inductive definitions, monotone inductive definitions (e.g., transitive closure) and definitions by induction over a well-founded order (e.g., the definition of $\models$ in FO, cf. Definition 1). Denecker [2, 3] was first to argue that rules under the well-founded semantics provide a uniform and correct formalization of these. Later, Denecker et al. [5] and Denecker and Ternovska [6] extended the original arguments. A full discussion of the arguments is beyond the scope of this paper but the essence is that an informal inductive

definition describes how to construct the defined relation by iterated application of rules and that the well-founded semantics correctly "simulates" this construction for the three aforementioned forms of definitions.

Not every formal rule set can be understood as a "good" informal inductive definition (i.e., one that a formal scientist would accept). In particular, a "good" definition should define for each object whether it is an element of the defined set or not. In formal terms, this means that a "good" formal rule set should have a total, i.e., 2-valued, well-founded model. Accordingly, such definitions are called *total* [3, 6]. Since parameterized stable and well-founded semantics coincide for total definitions, the above arguments apply immediately also to total D-modules. Therefore, the work by Denecker and his coauthors also provides a detailed explanation of why total D-modules under the stable model semantics correctly formalize the natural language concept of an inductive definition. To the best of our knowledge, such an explanation has not yet appeared in print before.

It does not apply to all of ASP, though. First, the analysis by Denecker and co-authors consistently interprets structures as possible worlds; therefore, our argument does not apply to the epistemic interpretation of stable models. Second, when we go beyond total D-modules, the correspondence to FO(ID) breaks down. In FO(ID), such rule sets are unsatisfiable, whereas in ASP-FO, they may have 0, 1 or more models. How such rule sets can be interpreted is an open question, but in practice there seems little need for non-total D-modules. Indeed, D-modules are non-total only in case of cycles over negation. In early applications of ASP, such cycles over negation were used to encode the generate and test parts of the search problem. However, more recently, these roles have been taken over by choice rules and constraints. Consequently, cycles over negation in D-modules have become very rare. In fact, in the current practice of ASP, D-modules almost always seem to be either positive or to contain only locally stratified negation. (but see below for an exception).

Comparison with the epistemic view. It is most interesting that the same mathematical principle can play a very different role depending on whether we take an epistemic or a possible world view on ASP. Under the stable model semantics, no atom belongs to an answer set unless it is derived by some rule (in an appropriate cycle-free manner). Under the epistemic view of an answer set, the informal explanation is that a *rational* agent should only *believe* an atom (or literal) if he has a justification for doing so. In the Tarskian setting, this explanation does not work, simply because the presence of an atom in an answer set does not reflect that it is *believed* but rather that it is *true* in the possible world. Thus, what the stable semantics expresses in the Tarskian view is that atoms cannot be true unless there is a reason for them to be so, which is a form of Closed World Assumption (CWA). In particular, it is a *global* CWA on *all* predicates. Of course, this is a strong assumption that often needs to be relaxed and this is where choice rules naturally step in. In epistemic ASP, on the other hand, no implicit CWA is imposed; if CWA is desired it must be stated explicitly, e.g., by rules $\sim P(\bar{x}) \leftarrow \texttt{not}\ P(\bar{x})$ involving strong negation [11]. Since there is, therefore, no implicit global CWA to "open," the role of choice rules is difficult to explain in this context. A remarkable conclusion is that the mathematical principle to formalize rationality in the epistemic view of stable models actually expresses a form of CWA in the possible world view of stable models. A more detailed discussion on the importance of the informal semantics of the "models" of a logic program can be found in [4].

The form of CWA implemented by the parameterized stable-model semantics in ASP-FO differs from other instances of CWA. It is *local*, i.e., applied only to the defined predicates $Ext(\mathcal{D})$, and it is also *parameterized*, in the sense that it is applied *given* the parameter $\mathfrak{A}_P$. For instance, the D-module $(\{P\}, \{P \leftarrow Q\})$ imposes CWA on P but it does not entail

$\neg P$. This is due to the parameter Q, which causes the ASP-FO semantics to admit two models: if the parameter Q is true, then P can be derived, so $\{Q, P\}$ is a model; if the parameter Q is false, then P cannot be derived and, by the CWA, must be false, so $\emptyset$ is also a model. Strikingly, this particular form of CWA, which deviates from standard forms of CWA, coincides for the important fragment of total D-modules with the precise and well-known mathematical principle of definition by induction. Whether the form of CWA underlying D-modules has natural KR applications beyond total definitions is an intriguing question. Such applications might be found in ASP programs that utilize cycles over negation for purposes other than to express choices or constraints, e.g., to express causal rules as in [13].

On the nature of negation and rule operator. Taking a possible world view also forces us to modify our interpretation of negation as failure. The embedding of ASP constraints and choice rules in FO shows that ASP's unary rule operator $\leftarrow$ for constraints as well as negation as failure `not` in such rules are the same as classical negation. As for negation in D-modules, we started this section by noting that in a Tarskian view, negation cannot be epistemic. Indeed, let us look at what negation means in informal definitions, for instance, in the following (informal) rule from our (informal) Definition 1: $\mathfrak{A}, \theta \models \neg\psi$ if $\mathfrak{A}, \theta \not\models \psi$. The definition is by structural induction, hence this rule should not be applied before rules deriving subformulas of $\neg\psi$. Once this condition is met, the rule derives $I, \theta \models \neg\varphi$ when *it is not the case that* $\mathfrak{A}, \theta \models \psi$. This is standard objective negation as formalized by classical negation in FO.

The difference between a rule $\forall x\ (P(\bar{t}) \leftarrow \varphi)$ in an FO(ID) definition or a D-module and a material implication $\forall x\ (P(\bar{t}) \Leftarrow \varphi)$ therefore does not lie in the interpretation of the connectives of φ. Instead, it lies in the rule operator $\leftarrow$, which differs from material implication $\Leftarrow$. Previous studies of inductive definitions called this operator also the *production operator*, reflecting its role of producing new elements of the defined relation. As discussed in [7], part of its meaning is the restriction that such elements should be produced in accordance with the well-founded order over which the induction is happening. This makes a rule indeed quite different from a material implication.

6 Discussion

Interpreting the answer-set semantics as a Tarskian possible-world semantics is a major mental leap which affects our interpretation of the ASP formalism, its composition laws, and the meaning of its connectives. While many ASP researchers may have already made this leap in their day-to-day programming under the Generate-Define-Test methodology, this paper offers the first detailed discussion of its consequences. To conduct our analysis, we presented the formalism ASP-FO, whose modular structure is geared specifically towards the GDT paradigm. By studying our possible-world perspective on ASP-FO, we obtained an informal semantics for the GDT fragment of ASP, which combines modules by means of the standard conjunction, and captures the roles of different modules in GDT-based programs.

We proposed ASP-FO as a theoretical mechanism to study GDT and ASP from a Tarskian perspective. However, ASP-FO is also a viable ASP logic for which several efficient ASP tools exists. Similarly to FO(ID), ASP-FO is an open domain logic and its models can be infinite. In general, the satisfiability problem is undecidable (and not just co-semidecidable) — the result can be obtained by adapting the corresponding result concerning the logic FO(ID) [6]. In many search problems, however, a finite domain is given. That opens a way to practical problem solving. One can apply finite Herbrand model generation or *model expansion* [15] and the corresponding tools [1]. Also, the IDP system [23] implements both the FO(ID) and ASP-FO semantics.

Finally, let us put the goals of this paper in a broader historical perspective. First, both logic programming and nonmonotonic reasoning were anti-theses to classical logic (FO), motivated by respectively computational and representational issues with the latter. The work on ASP-FO and earlier on FO(ID) effectively presents a synthesis of these paradigms with FO. Second, the view of logic programs as definitions was already present in Clark's view, albeit implicitly, and his completion semantics is not fully adequate to formalize this idea. Later, Gelfond and Lifschitz proposed to interpret logic programs as epistemic theories. The view on D-modules presented in this paper is a proposal to "backtrack" to Clark's original view.

References

1. A. Aavani, X. Wu, S. Tasharrofi, E. Ternovska, and D. G. Mitchell. Enfragmo: A system for modelling and solving search problems with logic. In N. Bjørner and A. Voronkov, editors, *Proceedings of the 18th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning, LPAR 2012*, volume 7180 of *LNCS*, pages 15–22, Berlin, 2012. Springer.
2. M. Denecker. The well-founded semantics is the principle of inductive definition. In J. Dix, L. Fariñas del Cerro, and U. Furbach, editors, *Proceedings of the European Workshop on Logics in Artificial Intelligence, JELIA 1998*, volume 1489 of *LNCS*, pages 1–16, Berlin, 1998. Springer.
3. M. Denecker. Extending classical logic with inductive definitions. In J.W. Lloyd, V. Dahl, U. Furbach, M. Kerber, K.-K. Lau, C. Palamidessi, L.M. Pereira, Y. Sagiv, and P.J. Stuckey, editors, *Proceedings of First International Conference on Computational Logic, CL 2000*, volume 1861 of *LNCS*, pages 703–717, Berlin, 2000. Springer.
4. M. Denecker. What's in a model? Epistemological analysis of logic programming. In D. Dubois, C.A. Welty, and M.-A. Williams, editors, *Proceedings of the 9th International Conference on Principles of Knowledge Representation and Reasoning*, pages 106–113, Palo Alto, CA, 2004. AAAI Press.
5. M. Denecker, M. Bruynooghe, and V.W. Marek. Logic programming revisited: Logic programs as inductive definitions. *ACM Transactions on Computational Logic*, 2(4):623–654, 2001.
6. M. Denecker and E. Ternovska. A logic of nonmonotone inductive definitions. *ACM Transactions on Computational Logic*, 9(2):1–50, 2008.
7. M. Denecker and J. Vennekens. Well-founded semantics and the algebraic theory of non-monotone inductive definitions. In C. Baral, G. Brewka, and J.S. Schlipf, editors, *Proceedings of the 9th International Conference on Logic Programming and Nonmonotonic Reasoning, LPNMR 2007*, volume 4483 of *LNCS*, pages 84–96, Berlin, 2007. Springer.
8. P. Ferraris, J. Lee, and V. Lifschitz. Stable models and circumscription. *Artificial Intelligence*, 175:236–263, 2011.
9. P. Ferraris, J. Lee, V. Lifschitz, and R. Palla. Symmetric splitting in the general theory of stable models. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence, IJCAI-2009*, pages 797–803, Palo Alto, CA, 2009. AAAI Press.
10. M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In R. Kowalski and K. Bowen, editors, *Proceedings of International Logic Programming Conference and Symposium*, pages 1070–1080, Cambridge, MA, 1988. MIT Press.
11. M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9:365–385, 1991.
12. V. Lifschitz. Answer set programming and plan generation. *Artificial Intelligence*, 138:39–54, 2002.

13 V. Lifschitz and H. Turner. Representing transition systems by logic programs. In M. Gelfond, N. Leone, and G. Pfeifer, editors, *Proceedings of International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, volume 1730 of *LNCS*, pages 92–106, Berlin, 1999. Springer.

14 V.W. Marek and M. Truszczynski. Stable models and an alternative logic programming paradigm. In K.R. Apt, V.W. Marek, M. Truszczynski, and D.S. Warren, editors, *The Logic Programming Paradigm: a 25-Year Perspective*, pages 375–398. Springer, Berlin, 1999.

15 D.G. Mitchell and E. Ternovska. A framework for representing and solving NP search problems. In *Proceedings of the 20th National Conference on Artificial Intelligence, AAAI 2005*, pages 430–435, Palo Alto, CA, 2005. AAAI Press.

16 R. C. Moore. Semantical considerations on nonmonotonic logic. *Artificial Intelligence*, 25(1):75–94, 1985.

17 I. Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25:241–273, 1999.

18 D. Pearce and A. Valverde. Quantified equilibrium logic and foundations for answer set programs. In *Proceedings of the 24th International Conference on Logic Programming, ICLP 2008*, volume 5366 of *LNCS*, pages 546–560, Berlin, 2008. Springer.

19 N. Pelov, M. Denecker, and M. Bruynooghe. Well-founded and stable semantics of logic programs with aggregates. *Theory and Practice of Logic Programming*, 7(3):301–353, 2007.

20 R. Reiter. A logic for default reasoning. *Artificial Intelligence*, 13:81–132, 1980.

21 M. Truszczynski. Connecting first-order asp and the logic FO(ID) through reducts, In E. Erdem, Y. Lierler, Y. Lierler, and D. Pearce, editors, *Correct Reasoning, Essays of Logic-Based AI in Honour of Vladimir Lifschitz*, volume 7265 of *LNCS*, Berlin, 2012. Springer.

22 J. Vennekens, J. Wittocx, M. Mariën, and M. Denecker. Predicate introduction for logics with a fixpoint semantics. Part I: Logic programming. *Fundamenta Informaticae*, 79(1-2):187–208, 2007.

23 J. Wittocx, M. Mariën, and M. Denecker. The IDP system: a model expansion system for an extension of classical logic. In M. Denecker, editor, *Logic and Search, Computation of Structures from Declarative Descriptions, LaSh 2008*, pages 153–165, 2008.

Paving the Way for Temporal Grounding*

Felicidad Aguado, Pedro Cabalar, Martín Diéguez, Gilberto Pérez, and Concepción Vidal

University of Corunna
Corunna, Spain
`{aguado,cabalar,martin.dieguez,gperez,eicovima}@udc.es`

Abstract

In this paper we consider the problem of introducing variables in temporal logic programs under the formalism of *Temporal Equilibrium Logic* (TEL), an extension of Answer Set Programming (ASP) for dealing with linear-time modal operators. We provide several fundamental contributions that pave the way for the implementation of a grounding process, that is, a method that allows replacing variables by ground instances in all the possible (or better, relevant) ways.

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

Keywords and phrases ASP, linear temporal logic, grounding, temporal equilibrium logic

Digital Object Identifier 10.4230/LIPIcs.ICLP.2012.290

1 Introduction

Many application domains and example scenarios from Answer Set Programming (ASP) [13, 11] contain a dynamic component, frequently representing transition systems over discrete time. However, temporal reasoning in ASP tends to be quite rudimentary, just treating time as an integer variable which is grounded for a finite interval[1]. To cope with more elaborated temporal reasoning, in [1] a formalism called *Temporal Equilibrium Logic* (TEL) was proposed. TEL is syntactically identical to propositional *Linear-time Temporal Logic* (LTL) [16], but semantically, it relies on a temporal extension of *Equilibrium Logic* [14], the most general and best studied logical characterisation of stable models (or answer sets) [7]. A recent work [2] introduced a reduction of TEL into regular LTL, for a syntactic subclass of temporal theories called *Splitable Temporal Logic Programs*. Although this syntactic fragment is a strict subset of the TEL normal form obtained in [4], it deals with temporal rules in which, informally speaking, "past does not depend on the future," something general enough to cover most (if not all) existing examples of ASP temporal scenarios. The reduction was implemented in a tool, `STeLP`[2] [5], that computes the temporal stable models of a given program, showing the result in the form of a Büchi automaton.

Although the theoretical results on which `STeLP` is based are restricted to the propositional case, the input language was extended with the introduction of variables. This was done imposing some strict limitations on the syntax, forcing that any variable instance is not only *safe* (that is, occurring in the positive body of the rule) but also "typed" by a *static* predicate, i.e., a predicate whose extent does not vary along time. In many cases, this restriction implied the generation of irrelevant ground rules that increase the size of the

* This research was partially supported by Spanish MEC project TIN2009-14562-C05-04.

[1] More elaborated approaches [12] deal with arbitrary temporal distances by using a constraint satisfaction tool as a backend.

[2] `http://kr.irlab.org/stelp_online`

© Felicidad Aguado, Pedro Cabalar, Martín Diéguez, Gilberto Pérez, and Concepción Vidal; licensed under Creative Commons License ND
Technical Communications of the 28th International Conference on Logic Programming (ICLP'12).
Editors: A. Dovier and V. Santos Costa; pp. 290–300

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

```
static city/1, car/1, road/2.
                    o at(X,A)  ::- driveto(X,A), car(X), city(A).   % (1)
driveto(X,B) v no_driveto(X,B) ::- at(X,A), car(X), road(A,B).      % (2)
o at(X,A)    ::- at(X,A), not o no_at(X,A), car(X), city(A).        % (3)
  no_at(X,A) ::- at(X,B), A!=B, car(X), city(A), city(B).           % (4)
::- at(X,A), at(X,B), A!=B, car(X), city(A), city(B).               % (5)
```

Figure 1 A simple car driving scenario.

resulting ground LTL theory while they could be easily detected and removed by a simple analysis of the temporal program. Furthermore, the treatment of variables had not been proved to be sound with respect to the important property of *domain independence* [3] – essentially, a program is domain independent when its stable models do not vary under the arbitrary addition of new constants. Although the DLV definition of safe variables guarantees domain independence, there was no formal proof for temporal logic programs under TEL.

In this paper we provide several fundamental results that pave the way for an improved grounder for temporal logic programs with variables. The rest of the paper is organised as follows. In the next section, we explain our motivations using an illustrative example. In Section 3 we introduce the first order extension of TEL and provide some basic definitions, explaining the syntactic form for our input language. Next, we study the relaxed definition of safe variables and prove that it guarantees domain independence. Section 5 defines the concept of *derivable facts*, explaining how they can be computed and used afterwards to generate smaller ground theories. Finally, Section 6 concludes the paper.

2 A motivating example

For a better understanding of our motivations, let us consider a simple illustrative example.

▶ **Example 1.** Suppose we have a set of cars placed at different cities and, at each transition, we can drive a car from one city to another in a single step, provided that there is a road connecting them. ◀

Figure 1 contains a possible representation of this scenario in the language of STeLP. In the rules, operator ‘o’ stands for “next” whereas ‘::-’ corresponds to the standard ASP conditional ‘:-’, but holding at all time points. Rule (1) is the effect axiom for driving car X to city A. The disjunctive rule (2) is used to generate possible occurrences of actions in a non-deterministic way. Rules (3) and (4) represent the inertia[3] of fluent at(X,A). Finally, rule (5) just forbids that a car is at two different cities simultaneously.

As we can see in the first line, predicates city/1, car/1 and road/2 are declared to be static. The scenario would be completed with rules for static predicates. These rules conform what we call the *static program* and can only refer to static predicates without containing temporal operators. An example of a static program for this scenario could be:

```
road(A,B) :- road(B,A).  % roads are bidirectional
city(A) :- road(A,B).
car(1). car(2).
road(lisbon,madrid). road(madrid,paris). road(boston,ny). road(ny,nj).
```

[3] Auxiliary predicates no_driveto(X,B) and no_at(X,A) play the role here of strong negation.

Additionally, our temporal program would contain rules describing the initial state like, for instance, the pair of facts:

```
at(1,madrid). at(2,ny).
```

Note that all variables in a rule are always in some atom for a static predicate in the positive body. This sometimes makes rule bodies quite long and slightly redundant. The current grounding process performed by STeLP just consists in feeding the static program to DLV and, once it provides an extension for all the static predicates, each temporal rule is instantiated for each possible substitution of variables according to static predicates. In our running example, for instance, DLV provides a unique model[4] for the static program containing the facts:

```
car(1), car(2), city(lisbon), city(madrid), city(paris), city(boston),
city(ny), city(nj), road(lisbon,madrid), road(madrid,lisbon),
road(madrid,paris), road(paris,madrid), road(boston,ny),
road(ny,boston), road(ny,nj), road(nj,ny)
```

With these data, rule (1) generates 12 ground instances, since we have two possible cars for X and six possible cities for A. Similarly, rule (4) would generate 60 instances as there are 30 pairs A,B of different cities and two cars for X. Many of these ground rules, however, are irrelevant. Consider, for instance, the following pair of generated rules:

```
o at(1,ny) ::- driveto(1,ny).
no_at(1,paris) ::- at(1,ny).
```

corresponding, respectively, to possible instantiations of (1) and (4). In both cases, the body refers to a situation where car 1 is located or will drive to New York, while we can observe that it was initially at Madrid and that the European roadmap is disconnected from the American one. Of course, one could additionally encode a static reachability predicate to force that rule instances refer to reachable cities for a given car, but this would not be too transparent or elaboration tolerant. One would expect that the grounder was capable of detecting these "non-derivable" cases ignoring them in the final ground theory, if possible.

On the other hand, if we forget, for a moment, the temporal operators and we consider the definition of safe variables used in DLV, one may also wonder whether it is possible to simply require that each variable occurs in the positive body of rules, without needing to refer to static predicates mandatorily. Figure 2 contains a possible variation of the same scenario allowing this possibility. Our goal is allowing this new, more flexible definition of safe variables and exploiting, if possible, the information in the temporal program to reduce the set of generated ground rules.

3 Temporal Quantified Equilibrium Logic

Syntactically, we consider function-free first-order languages $\mathcal{L} = \langle C, P \rangle$ built over a set of *constant* symbols, C, and a set of *predicate* symbols, P. Using $\mathcal{L}$, connectors and variables, an $\mathcal{L} = \langle C, P \rangle$*-formula* F is defined following the grammar:

$$F ::= p \mid \bot \mid F_1 \wedge F_2 \mid F_1 \vee F_2 \mid F_1 \rightarrow F_2 \mid \bigcirc F \mid \Box F \mid \Diamond F \mid \forall x F(x) \mid \exists x F(x)$$

[4] If the static program yields several stable models, each one generates a different ground theory whose temporal stable models are computed independently.

```
static city/1, car/1, road/2.
                    o at(X,A)    ::- driveto(X,A).
driveto(X,B) v no_driveto(X,B) ::- at(X,A), road(A,B).
                    o at(X,A)    ::- at(X,A), not o no_at(X,A).
                      no_at(X,A) ::- at(X,B), A!=B, city(A).
                                 ::- at(X,A), at(X,B), A!=B.
```

Figure 2 A possible variation of the cars scenario.

where $p \in P$ is an atom, x is a variable and $\bigcirc$, $\Box$ and $\Diamond$ respectively stand for "next", "always" and "eventually." A *theory* is a finite set of formulas. We use the following derived operators and notation: $\neg F \stackrel{\text{def}}{=} F \to \bot$, $\top \stackrel{\text{def}}{=} \neg\bot$ and $F \leftrightarrow G \stackrel{\text{def}}{=} (F \to G) \wedge (G \to F)$ for any formulas F, G. An *atom* is any $p(t_1, \dots, t_n)$ where $p \in P$ is a predicate with n-arity and each t_i is a term (a constant or a variable) in its turn. We say that a term or a formula is *ground* if it does not contain variables. An $\mathcal{L}$*-sentence* or closed-formula is a formula without free-variables. The application of i consecutive $\bigcirc$'s is denoted as follows: $\bigcirc^i\varphi \stackrel{\text{def}}{=} \bigcirc(\bigcirc^{i-1}\varphi)$ for $i > 0$ and $\bigcirc^0\varphi \stackrel{\text{def}}{=} \varphi$. A *temporal fact* is a construction of the form $\bigcirc^i A$ where A is an atom. If D is a non-empty set, we denote by $At(D, P)$ the set of ground atomic sentences of the language $\langle D, P\rangle$. For the semantics, we will also define a mapping $\sigma\colon C \cup D \to D$ such that $\sigma(d) = d$ for all $d \in D$.

A first-order LTL-interpretation is a structure $\langle (D, \sigma), \mathbf{T}\rangle$ where D is a non-empty set (the *domain*), σ is a mapping as defined above (the interpretation of constants) and $\mathbf{T}$ is an infinite sequence of sets of ground atoms $\mathbf{T} = \{T_i\}_{i\geq 0}$. Intuitively, $T_i \subseteq At(D, P)$ contains those ground atoms that are true at situation i. Given two LTL-interpretations $\mathbf{H}$ and $\mathbf{T}$ we say that $\mathbf{H}$ is *smaller than* $\mathbf{T}$, written $\mathbf{H} \leq \mathbf{T}$, when $H_i \subseteq T_i$ for all $i \geq 0$. As usual, $\mathbf{H} < \mathbf{T}$ stands for: $\mathbf{H} \leq \mathbf{T}$ and $\mathbf{H} \neq \mathbf{T}$. We define the ground temporal facts associated to $\mathbf{T}$ as follows: $Facts(\mathbf{T}) \stackrel{\text{def}}{=} \{\bigcirc^i p \mid p \in T_i\}$. It is easy to see that $\mathbf{H} \leq \mathbf{T}$ iff $Facts(\mathbf{H}) \subseteq Facts(\mathbf{T})$.

▶ **Definition 2.** A *temporal-here-and-there* $\mathcal{L}$-structure with static domains, or a **TQHT**-*structure*, is a tuple $\mathcal{M} = \langle (D, \sigma), \mathbf{H}, \mathbf{T}\rangle$ where $\langle (D, \sigma), \mathbf{H}\rangle$ and $\langle (D, \sigma), \mathbf{T}\rangle$ are two LTL-interpretations satisfying $\mathbf{H} \leq \mathbf{T}$. ◀

A **TQHT**-structure of the form $\mathcal{M} = \langle (D, \sigma), \mathbf{T}, \mathbf{T}\rangle$ is said to be *total*. If $\mathcal{M} = \langle (D, \sigma), \mathbf{H}, \mathbf{T}\rangle$ is a **TQHT**-structure and k any positive integer, we denote by $(\mathcal{M}, k) = \langle (D, \sigma), (\mathbf{H}, k), (\mathbf{T}, k)\rangle$ the temporal-here-and-there $\mathcal{L}$-structure with $(\mathbf{H}, k) = \{H_i\}_{i\geq k}$ and $(\mathbf{T}, k) = \{T_i\}_{i\geq k}$. The satisfaction relation for $\mathcal{M} = \langle (D, \sigma), \mathbf{H}, \mathbf{T}\rangle$ is defined recursively forcing us to consider formulas from $\langle C \cup D, P\rangle$. Formally, if φ is an $\mathcal{L}$-sentence for the atoms in $At(C \cup D, P)$, then:

- If $\varphi = p(t_1, \dots, t_n) \in At(C \cup D, P)$, then

$$\mathcal{M} \models p(t_1, \dots, t_n) \quad \text{iff} \quad p(\sigma(t_1), \dots, \sigma(t_n)) \in H_0.$$
$$\mathcal{M} \models t = s \quad \text{iff} \quad \sigma(t) = \sigma(s)$$

- For $\bot$, $\wedge$ and $\vee$, as usual.
- $\mathcal{M} \models \varphi \to \psi$ iff $\langle (D, \sigma), w, \mathbf{T}\rangle \not\models \varphi$ or $\langle (D, \sigma), w, \mathbf{T}\rangle \models \psi$ for all $w \in \{\mathbf{H}, \mathbf{T}\}$
- $\mathcal{M} \models \bigcirc\varphi$ if $(\mathcal{M}, 1) \models \varphi$.
- $\mathcal{M} \models \Box\varphi$ if $\forall j \geq 0$, $(\mathcal{M}, j) \models \varphi$
- $\mathcal{M} \models \Diamond\varphi$ if $\exists j \geq 0$, $(\mathcal{M}, j) \models \varphi$

- $\langle (D,\sigma), \mathbf{H}, \mathbf{T} \rangle \models \forall x \varphi(x)$ iff $\langle (D,\sigma), w, \mathbf{T} \rangle \models \varphi(d)$ for all $d \in D$ and for all $w \in \{\mathbf{H}, \mathbf{T}\}$.
- $\mathcal{M} \models \exists x \varphi(x)$ iff $\mathcal{M} \models \varphi(d)$ for some $d \in D$.

The resulting logic is called *Quantified Temporal Here-and-There Logic with static domains*, and denoted by **SQTHT** or simply by **QTHT**. It is not difficult to see that, if we restrict to total **TQHT**-structures, $\langle (D,\sigma), \mathbf{T}, \mathbf{T} \rangle \models \varphi$ iff $\langle (D,\sigma), \mathbf{T}, \mathbf{T} \rangle \models \varphi$ in first-order LTL. Furthermore, the following property can be easily checked by structural induction.

▶ **Proposition 3.** *For any formula φ, if $\langle (D,\sigma), \mathbf{H}, \mathbf{T} \rangle \models \varphi$, then $\langle (D,\sigma), \mathbf{T}, \mathbf{T} \rangle \models \varphi$*

A *theory* Γ is a set of $\mathcal{L}$-sentences. An interpretation $\mathcal{M}$ is a model of a theory Γ, written $\mathcal{M} \models \Gamma$, if it satisfies all the sentences in Γ.

▶ **Definition 4** (Temporal Equilibrium Model)**.** A *temporal equilibrium* model of a theory Γ is a total model $\mathcal{M} = \langle (D,\sigma), \mathbf{T}, \mathbf{T} \rangle$ of Γ such that there is no $\mathbf{H} < \mathbf{T}$ satisfying $\langle (D,\sigma), \mathbf{H}, \mathbf{T} \rangle \models \Gamma$. ◀

If $\mathcal{M} = \langle (D,\sigma), \mathbf{T}, \mathbf{T} \rangle$ is a temporal equilibrium model of a theory Γ, we say that the First-Order LTL interpretation $\langle (D,\sigma), \mathbf{T} \rangle$ is a *temporal stable model* of Γ. We write $TSM(\Gamma)$ to denote the set of temporal stable models of Γ. The set of *credulous consequences* of a theory Γ, written $CredFacts(\Gamma)$ contains all the temporal facts that occur at some temporal stable model of Γ, that is:

$$CredFacts(\Gamma) \quad \stackrel{\text{def}}{=} \bigcup_{\langle (D,\sigma), \mathbf{T} \rangle \in TSM(\Pi)} Facts(\mathbf{T})$$

A property of TEL directly inherited from Equilibrium Logic (see Proposition 5 in [15]) is the following:

▶ **Proposition 5** (Cummulativity for negated formulas)**.** *Let Γ be some theory and let $\neg\varphi$ be some formula such that $\mathcal{M} \models \neg\varphi$ for all temporal equilibrium models of Γ. Then, the theories Γ and $\Gamma \cup \{\neg\varphi\}$ have the same set of temporal equilibrium models.* ◀

It is well-known that stable models (and so Equilibrium Logic) do not satisfy cummulativity in the general case: that is, if a formula is satisfied in all the stable models, adding it to the program may vary the consequences we obtain. However, when we deal with negated formulas, Proposition 5 tells us that cummulativity is guaranteed.

In this work, we will further restrict the study to a syntactic subset called *splitable*[5] temporal formulas (STF) which will be of one of the following types:

$$B \wedge N \quad \rightarrow \quad H \tag{1}$$

$$B \wedge \bigcirc B' \wedge N \wedge \bigcirc N' \quad \rightarrow \quad \bigcirc H' \tag{2}$$

$$\Box(B \wedge \bigcirc B' \wedge N \wedge \bigcirc N' \quad \rightarrow \quad \bigcirc H') \tag{3}$$

where B and B' are conjunctions of atomic formulas, N and N' are conjunctions of $\neg p$, being p an atomic formula and H and H' are disjunctions of atomic formulas.

[5] The name *splitable* refers to the fact that these programs can be splitted using [10] thanks to the property that rule heads never refer to a time point previous to those referred in the body.

▶ **Definition 6.** A *splitable temporal logic program* (*STL-program* for short) is a finite set of sentences like

$$\varphi = \forall x_1 \forall x_2 \ldots \forall x_n \psi,$$

where ψ is a splitable temporal formula with $x_1, x_2, \ldots, x_n$ free variables.

We will also accept in an STL-program an implication of the form $\Box(B \wedge N \rightarrow H)$ (that is, containing $\Box$ but not any $\bigcirc$) understood as an abbreviation of the pair of STL-formulas:

$$B \wedge N \rightarrow H$$
$$\Box(\bigcirc B \wedge \bigcirc N \rightarrow \bigcirc H)$$

▶ **Example 7.** The following theory Π_7 is an STL-program:

$$\neg p \rightarrow q \quad (4)$$
$$q \wedge \neg \bigcirc r \rightarrow \bigcirc p \quad (5)$$
$$\Box(q \wedge \neg \bigcirc p \rightarrow \bigcirc q) \quad (6)$$
$$\Box(r \wedge \neg \bigcirc p \rightarrow \bigcirc r \vee \bigcirc q) \quad (7)$$

For an example including variables, the encoding of Example 1 in Figure 2 is also an STL-program Π_1 whose logical representation corresponds to:

$$\Box(\ Driveto(x,a) \rightarrow \bigcirc At(x,a)\ ) \quad (8)$$
$$\Box(\ At(x,a) \wedge Road(a,b) \rightarrow Driveto(x,b) \vee NoDriveto(x,b)\ ) \quad (9)$$
$$\Box(\ At(x,a) \wedge \neg \bigcirc NoAt(x,a) \rightarrow \bigcirc At(x,a)\ ) \quad (10)$$
$$\Box(\ At(x,b) \wedge City(a) \wedge a \neq b \rightarrow NoAt(x,a)\ ) \quad (11)$$
$$\Box(\ At(x,a) \wedge At(x,b) \wedge a \neq b \rightarrow \bot\ ) \quad (12)$$

Remember that all rule variables are implicitly universally quantified. For simplicity, we assume that inequality is a predefined predicate.

An STL-program is said to be *positive* if for all rules (1)-(3), N and N' are empty (an empty conjunction is equivalent to $\top$). An STL-program is said to be *normal* if it contains no disjunctions, i.e., for all rules (1)-(3), H and H' are atoms. Given a propositional combination φ of temporal facts with $\wedge, \vee, \bot, \rightarrow$, we denote φ^i as the formula resulting from replacing each temporal fact A in φ by $\bigcirc^i A$. For a formula $r = \Box\varphi$ like (3), we denote by r^i the corresponding φ^i. For instance, $(6)^i = (\bigcirc^i q \wedge \neg \bigcirc^{i+1} p \rightarrow \bigcirc^{i+1} q)$. As $\bigcirc$ behaves as a linear operator in THT, in fact $F^i \leftrightarrow \bigcirc^i F$ is a THT tautology.

▶ **Definition 8** (expanded program). Given an STL-program Π for signature Σ we define its *expanded program* Π^∞ as the infinitary logic program containing all rules of the form (1), (2) in Π plus a rule r^i per each rule r of the form (3) in Π and each integer value $i \geq 0$. ◀

The program Π_7^∞ would therefore correspond to (4), (5) plus the infinite set of rules:

$$\bigcirc^i q \wedge \neg \bigcirc^{i+1} p \rightarrow \bigcirc^{i+1} q \qquad \bigcirc^i r \wedge \neg \bigcirc^{i+1} p \rightarrow \bigcirc^{i+1} r \vee \bigcirc^{i+1} q$$

for $i \geq 0$. We can interpret the expanded program as an infinite non-temporal program where the signature is the infinite set of atoms of the form $\bigcirc^i p$ with $p \in At$ and $i \geq 0$.

▶ **Theorem 9** (Theorem 1 in [2]). *$\langle \mathbf{T}, \mathbf{T} \rangle$ is a temporal equilibrium model of Π iff $\{\bigcirc^i p \mid p \in T_i, i \geq 0\}$ is a stable model of Π^∞ under the (infinite) signature $\{\bigcirc^i p \mid p \in \Sigma\}$.* ◀

▶ **Proposition 10.** *Any normal positive STL-program Π has a unique temporal stable model $\langle (D,\sigma), \mathbf{T} \rangle$ which coincides with its $\leq$-least LTL-model. We denote $LM(\Pi) = Facts(\mathbf{T})$.* ◀

4 Safe Variables and Domain Independence

In this section we consider the new definition of safe variables which does not refer to static predicates any more. As a result, we obtain a direct extrapolation of DLV-safety by just ignoring the temporal operators.

▶ **Definition 11.** A splitable temporal formula φ of type (1), (2) or (3) is said to be *safe* if, for any variable x occurring in φ, there exists an atomic formula p in B or B' such that x occurs in p. A formula $\forall x_1 \forall x_2 \ldots \forall x_n \psi$ is safe if the splitable temporal formula ψ is safe.

For instance, rules (8)-(12) are safe. A simple example of unsafe rule is the splitable temporal formula $\top \rightarrow P(x)$ where x does not occur in the positive body (in fact, the rule body is empty). Although an unsafe rule not always leads to lack of domain independence (see examples in [6]) it is frequently the case. We prove next that domain independence is, in fact, guaranteed for safe STL-programs.

▶ **Theorem 12.** *If φ is a safe sentence and $\langle (D,\sigma), \mathbf{T}, \mathbf{T} \rangle$ is a temporal equilibrium model of φ, then $\mathbf{T}|_C = \mathbf{T}$ and $T_i \subseteq At(\sigma(C), P)$ for any $i \geq 0$.*

Let (D,σ) be a domain and $D' \subseteq D$ a finite subset; the grounding over D' of a sentence φ, denoted by $\mathrm{Gr}_{D'}(\varphi)$, is defined recursively

$$\begin{aligned}
\mathrm{Gr}_{D'}(p) &\stackrel{\text{def}}{=} p, \text{ where } p \text{ denotes any atomic formula}\\
\mathrm{Gr}_{D'}(\varphi_1 \odot \varphi_2) &\stackrel{\text{def}}{=} \mathrm{Gr}_{D'}(\varphi_1) \odot \mathrm{Gr}_{D'}(\varphi_2), \text{ with } \odot \text{ any binary operator in } \{\wedge, \vee, \rightarrow\}\\
\mathrm{Gr}_{D'}(\forall x \varphi(x)) &\stackrel{\text{def}}{=} \bigwedge_{d \in D'} \mathrm{Gr}_{D'} \varphi(d)\\
\mathrm{Gr}_{D'}(\exists x \varphi(x)) &\stackrel{\text{def}}{=} \bigvee_{d \in D'} \mathrm{Gr}_{D'} \varphi(d)\\
\mathrm{Gr}_{D'}(\bigcirc \varphi) &\stackrel{\text{def}}{=} \bigcirc \mathrm{Gr}_{D'}(\varphi)\\
\mathrm{Gr}_{D'}(\square \varphi) &\stackrel{\text{def}}{=} \square \mathrm{Gr}_{D'}(\varphi)\\
\mathrm{Gr}_{D'}(\lozenge \varphi) &\stackrel{\text{def}}{=} \lozenge \mathrm{Gr}_{D'}(\varphi)
\end{aligned}$$

▶ **Proposition 13.** *Given any non-empty finite set D:*
$\langle (D,\sigma), \mathbf{H}, \mathbf{T} \rangle \models \varphi$ *iff* $\langle (D,\sigma), \mathbf{H}, \mathbf{T} \rangle \models \mathrm{Gr}_D(\varphi)$. ◀

▶ **Theorem 14** (Domain independence). *Let φ be safe splitable temporal sentence. Suppose we expand the language $\mathcal{L}$ by considering a set of constants $C' \supseteq C$. A total* **QTHT***-model $\langle (D,\sigma), \mathbf{T}, \mathbf{T} \rangle$ is a temporal equilibrium model of $\mathrm{Gr}_{C'}(\varphi)$ if and only if it is a temporal equilibrium model of $\mathrm{Gr}_C(\varphi)$.*

5 Derivable ground facts

In this section we present a technique for grounding safe temporal programs based on the construction a positive normal ASP program with variables. The least model of this program can be obtained by the ASP grounder[6] DLV and it can be used afterwards to provide the variable substitutions to be performed on the STL-program. Besides, in some cases, this technique means a reduction of the number of generated ground rules with respect to the previous strategy that relied on static predicates.

[6] Or any other ASP grounder, such as `gringo`, respecting DLV definition of safe variables.

The method is based on the idea of *derivable* ground temporal facts for an STL-program Π. This set, call it Δ, will be an upper estimation of the credulous consequences of the program, that is, $CredFacts(\Pi) \subseteq \Delta$. Of course, the ideal situation would be that $\Delta = CredFacts(\Pi)$, but the set $CredFacts(\Pi)$ requires the temporal stable models of Π and these (apart from being infinite sequences) will not be available at grounding time. In the worst case, we could choose Δ to contain the whole set of possible temporal facts, but this would not provide relevant information to improve grounding. So, we will try to obtain some superset of $CredFacts(\Pi)$ as small as possible, or if preferred, to obtain the largest set of *non-derivable* facts we can find. Note that a non-derivable fact $\bigcirc^i p \notin \Delta$ satisfies that $\bigcirc^i p \notin CredFacts(\Pi)$ and so, by Proposition 5, $\Pi \cup \{\neg \bigcirc^i p\}$ is equivalent to Π, that is, both theories have the same set of temporal equilibrium models. This information can be used to simplify the ground program either by removing rules or literals.

We begin defining several transformations on STL-programs. For any temporal rule r, we define $r^\wedge$ as the set of rules:

- If r has the form (1) then $r^\wedge \stackrel{\text{def}}{=} \{B \to p \mid \text{atom } p \text{ occurs in } H\}$
- If r has the form (2) then $r^\wedge \stackrel{\text{def}}{=} \{B \wedge \bigcirc B' \to \bigcirc p \mid \text{atom } p \text{ occurs in } H'\}$
- If r has the form (3) then $r^\wedge \stackrel{\text{def}}{=} \{\Box(B \wedge \bigcirc B' \to \bigcirc p) \mid \text{atom } p \text{ occurs in } H'\}$

In other words, $r^\wedge$ results from removing all negative literals in r and, informally speaking, transforming disjunctions in the head into conjunctions, so that $r^\wedge$ will imply *all* the original disjuncts in the disjunctive head of r. It is interesting to note that for any rule r with an empty head ($\bot$) this definition implies $r^\wedge = \emptyset$. Program $\Pi^\wedge$ is defined as the union of $r^\wedge$ for all rules $r \in \Pi$. As an example, $\Pi_7^\wedge$ consists of the rules:

$$\begin{array}{lll} \top \to q & \Box(q \to \bigcirc q) & \Box(r \to \bigcirc r) \\ q \to \bigcirc p & & \Box(r \to \bigcirc q) \end{array}$$

whereas $\Pi_1^\wedge$ would be the program:

$$\Box(\ Driveto(x,a) \to \bigcirc At(x,a)\ ) \tag{13}$$
$$\Box(\ At(x,a) \wedge Road(a,b) \to Driveto(x,b)\ ) \tag{14}$$
$$\Box(\ At(x,a) \wedge Road(a,b) \to NoDriveto(x,b)\ ) \tag{15}$$
$$\Box(\ At(x,a) \to \bigcirc At(x,a)\ ) \tag{16}$$
$$\Box(\ At(x,b) \wedge City(a) \wedge a \neq b \to NoAt(x,a)\ ) \tag{17}$$

If we look carefully at this example program, we are now moving each car x so that it will be at several cities at the same time (constraint (12) has been removed) and, at each step, it will additionally locate car x in all adjacent cities to the previous ones "visited" by x. In this way, if we conclude $\bigcirc^i At(x,a)$ from this program this is actually representing that car x *can reach* city a in i steps or less. In some sense, $\Pi^\wedge$ looks like a heuristic simplification[7] of the original problem obtained by removing some constraints (this is something common in the area of Planning in Artificial Intelligence).

Notice that, by definition, $\Pi^\wedge$ is always a positive normal STL-program and, by Proposition 10, it has a unique temporal stable model, $LM(\Pi^\wedge)$.

▶ **Proposition 15.** *For any STL-program* Π, $CredFacts(\Pi) \subseteq LM(\Pi^\wedge)$. ◀

[7] We could further simplify $\Pi^\wedge$ removing rules (15) and (17) by observing that their head predicates never occur in a positive body of Π_1. However, for the formal results in the paper, this is not essential, and would complicate the definitions.

Unfortunately, using $\Delta = LM(\Pi^\wedge)$ as set of derivable facts is unfeasible for practical purposes, since this set contains infinite temporal facts corresponding to an "infinite run" of the transition system described by $\Pi^\wedge$. Take for instance $\Pi_1^\wedge$ for the cars scenario. Imagine a roadmap with thousands of connected cities. $LM(\Pi^\wedge)$ can tell us that, for instance, car 1 cannot reach Berlin in less than 316 steps, so that $\bigcirc^{315} At(1, Berlin)$ is non-derivable, although $\bigcirc^{316} At(1, Berlin)$ is derivable. However, in order to exploit this information for grounding, we would be forced to expand the program up to some temporal distance, and we have no hint on where to stop. Note that, on the other hand, when we represent the transition system as usual in ASP, using a bounded integer variable for representing time, then this fine-grained optimization for grounding can be applied, because the temporal path *always has a finite length*.

As a result, we will adopt a compromise solution taking a superset of $LM(\Pi^\wedge)$ extracted from a new theory, Γ_Π. This theory will collapse all the temporal facts from situation 2 on, so that all the states T_i for $i \geq 2$ will be repeated. We define Γ_Π as the result of replacing each rule $\Box(B \wedge \bigcirc B' \rightarrow \bigcirc p)$ in $\Pi^\wedge$ by the formulas:

$$B \wedge \bigcirc B' \rightarrow \bigcirc p \tag{18}$$

$$\bigcirc B \wedge \bigcirc^2 B' \rightarrow \bigcirc^2 p \tag{19}$$

$$\bigcirc^2 B \wedge \bigcirc^2 B' \rightarrow \bigcirc^2 p \tag{20}$$

and adding the axiom schema:

$$\bigcirc^2 \Box(p \leftrightarrow \bigcirc p) \tag{21}$$

for any ground atom $p \in At(D, P)$ in the signature of Π. As we can see, (18) and (19) are the first two instances of the original rule $\Box(B \wedge \bigcirc B' \rightarrow \bigcirc p)$ corresponding to situations $i = 0$ and $i = 1$. Formula (20), however, differs from the instance we would get for $i = 2$ since, rather than having $\bigcirc^3 B'$ and $\bigcirc^3 p$, we use $\bigcirc^2 B'$ and $\bigcirc^2 p$ respectively. This can be done because axiom (21) is asserting that from situation 2 on all the states are repeated.

In the cars example, for instance, rule (13) in $\Pi_1^\wedge$ would be transformed in Γ_{Π_1} into the three rules:

$$Driveto(x, a) \rightarrow \bigcirc At(x, a) \qquad \bigcirc Driveto(x, a) \rightarrow \bigcirc^2 At(x, a)$$

$$\bigcirc^2 Driveto(x, a) \rightarrow \bigcirc^2 At(x, a)$$

It is not difficult to see that axiom (21) implies that checking that some $\mathcal{M}$ is a temporal equilibrium model of Γ_Π is equivalent to check that $\{\bigcirc^i p \mid p \in T_i\, ,\, i = 0, 1, 2\}$ is a stable model of $\Gamma_\Pi \setminus \{(21)\}$ and fixing $T_i = T_2$ for $i \geq 3$. This allows us to exclusively focus on the predicate extents in T_0, T_1 and T_2, so we can see the $\Box$-free program $\Gamma_\Pi \setminus \{(21)\}$ as a positive normal ASP (i.e., non-temporal) program for the propositional signature $\{p, \bigcirc p, \bigcirc^2 p \mid p \in At(D, P)\}$ that can be directly fed to DLV, after some simple renaming conventions.

▶ **Theorem 16.** Γ_Π *has a least LTL-model,* $LM(\Gamma_\Pi)$ *which is a superset of* $LM(\Pi^\wedge)$*.*

In other words $CredFacts(\Pi) \subseteq LM(\Pi^\wedge) \subseteq LM(\Gamma_\Pi) = \Delta$, i.e., we can use $LM(\Gamma_\Pi)$ as set of derivable facts and simplify the ground program accordingly. Note that this simplification does not mean that we first ground everything and then remove rules and literals: we simply do not generate the irrelevant ground cases.

A slight adaptation[8] is further required for this method: as we get ground facts of the form $p, \bigcirc p$ and $\bigcirc^2 p$ we have to unfold the original STL-program rules to refer to atoms in

[8] In fact, this means that we have to extend the definition of splitable temporal rule to cope with $\bigcirc^2$ atoms. This is not essential, but has forced to reprogram the translation into LTL performed by `STeLP`.

the scope of $\bigcirc^2$. For instance, given (9) we would first unfold it into:

$$\begin{array}{rcll} At(x,a) \wedge Road(a,b) & \rightarrow & Driveto(x,b) \vee NoDriveto(x,b) & (22) \\ \bigcirc At(x,a) \wedge \bigcirc Road(a,b) & \rightarrow & \bigcirc Driveto(x,b) \vee \bigcirc NoDriveto(x,b) & (23) \\ \Box(\ \bigcirc^2 At(x,a) \wedge \bigcirc^2 Road(a,b) & \rightarrow & \bigcirc^2 Driveto(x,b) & \\ & & \vee \bigcirc^2\ NoDriveto(x,b)\ ) & (24) \end{array}$$

and then check the possible extents for the positive bodies we get from the set of derivable facts $\Delta = LM(\Gamma_\Pi)$. For instance, for the last rule, we can make substitutions for x, a and b using the extents of $\bigcirc^2 At(x,a)$ and $\bigcirc^2 Road(a,b)$ we have in Δ. However, this still means making a join operation for both predicates. We can also use DLV for that purpose by just adding a rule that has as body, the positive body of the original temporal rule r, and as head, a new auxiliary predicate $Subst_r(x,a,b)$ referring to all variables in the rule. In the example, for rule (24) we would include in our DLV program:

$$\bigcirc^2 At(x,a) \wedge \bigcirc^2 Road(a,b) \rightarrow Subst_{(24)}(x,a,b)$$

In this way, each tuple of $Subst_r(x_1,\dots,x_n)$ directly points out the variable substitution to be performed on the temporal rule.

6 Conclusions

We have improved the grounding method for temporal logic programs with variables in different ways. First, we provided a safety condition that directly corresponds to extrapolating the usual concept of safe variable in Answer Set Programming as required, for instance, by the input language of DLV [9]. In this way, any variable occurring in a rule is considered to be safe if it also occurs in the positive body of the rule, regardless the possible scope of temporal operators. An interesting topic for future study is trying to extend [3, 8, 6] to the temporal case, providing a general safety condition for arbitrary quantified temporal theories. Second, we have designed a method for grounding the temporal logic program that consists in constructing a non-temporal normal positive program with variables that is fed to solver DLV to directly obtain the set of variable substitutions to be performed for each rule. The proposed method allows reducing in many cases the number of ground temporal rules generated as a result. For instance, in the cars scenario from Figure 2 and the small instance case described in the paper (2 cars and 6 cities) we reduce the number of generated ground rules in the scope of '$\Box$' from 160 using the current STeLP grounding method to 62 with the technique introduced here. Due to the combinatorial nature of this decrease, we do not include figures for other instances of the example. The reader may easily imagine that the higher degree of cities interconnection, the smaller obtained reduction of rule instances, as this approaches to the worst case of n^2, where the n cities are all pairwise connected. On the other hand, the example is general enough to illustrate the proposed technique, as rules for temporal predicates usually limit the possible combinations of variable values we must consider.

A stand-alone prototype for proving examples like the one in the paper has been constructed, showing promising results. The immediate future work is incorporating the new grounding method inside STeLP and analysing its performance on benchmark scenarios. We will also study different improvements like, for instance, detecting rules with variables that are irrelevant for grounding.

References

1 F. Aguado, P. Cabalar, G. Pérez, and C. Vidal. Strongly equivalent temporal logic programs. In *JELIA 2008*, volume 5293 of LNCS, pages 8–20.

2 F. Aguado, P. Cabalar, G. Pérez, and C. Vidal. Loop formulas for splitable temporal logic programs. In James P. Delgrande and Wolfgang Faber, editors, *LPNMR'11*, volume 6645 of LNCS, pages 80–92. Springer, 2011.

3 A. Bria, W. Faber, and N. Leone. Normal form nested programs. In S. Hölldobler *et al*, editor, *Proc. of the 11th European Conference on Logics in Artificial Intelligence (JELIA'08)*, Lecture Notes in Artificial Intelligence, pages 76–88. Springer, 2008.

4 P. Cabalar. A normal form for linear temporal equilibrium logic. In *JELIA'10*, volume 6341 of LNCS, pages 64–76. Springer, 2010.

5 P. Cabalar and M. Diéguez. STELP - a tool for temporal answer set programming. In *LPNMR'11*, volume 6645 of LNCS, pages 370–375, 2011.

6 P. Cabalar, D. Pearce, and A. Valverde. A revised concept of safety for general answer set programs. In *LPNMR'09*, volume 5753 of LNCS, pages 58–70. Springer, 2009.

7 M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In R. A. Kowalski and K. A. Bowen, editors, *Logic Programming: Proc. of the Fifth International Conference and Symposium (Volume 2)*, pages 1070–1080. MIT Press, 1988.

8 J. Lee, V. Lifschitz, and R. Palla. Safe formulas in the general theory of stable models. preliminary report. In *ICLP'08*, volume 5366 of *LNCS*, pages 672–676. Springer, 2008.

9 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*, 7:499–562, 2006.

10 Vladimir Lifschitz and Hudson Turner. Splitting a logic program. In *Proceedings of the eleventh international conference on Logic programming*, pages 23–37, Cambridge, MA, USA, 1994. MIT Press.

11 V. Marek and M. Truszczyński. *Stable models and an alternative logic programming paradigm*, pages 169–181. Springer-Verlag, 1999.

12 Veena S. Mellarkod, Michael Gelfond, and Yuanlin Zhang. Integrating answer set programming and constraint logic programming. *Annals of Maths and AI*, 53(1-4):251–287, 2008.

13 I. Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25:241–273, 1999.

14 D. Pearce. A new logical characterisation of stable models and answer sets. In *Non monotonic extensions of logic programming. Proc. NMELP'96. (LNAI 1216)*. 1996.

15 David Pearce. Equilibrium logic. *Annals of Maths and AI*, 47(1-2):3–41, 2006.

16 A. Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science*, pages 46–57. IEEE Computer Society Press, 1977.

Logic + control: An example

Włodzimierz Drabent

Institute of Computer Science, Polish Academy of Sciences, Poland, and
IDA, Linköping University, Sweden
`drabent@ipipan.waw.pl`

Abstract

We present a Prolog program – the SAT solver of Howe and King – as a (pure) logic program with added control. The control consists of a selection rule (delays of Prolog) and pruning the search space. We construct the logic program together with proofs of its correctness and completeness, with respect to a formal specification. Correctness and termination of the logic program are inherited by the Prolog program; the change of selection rule preserves completeness. We prove that completeness is also preserved by one case of pruning; for the other an informal justification is presented.

For proving correctness we use a method, which should be well known but is often neglected. For proving program completeness we employ a new, simpler variant of a method published previously. We point out usefulness of approximate specifications. We argue that the proof methods correspond to natural declarative thinking about programs, and that they can be used, formally or informally, in every-day programming.

1998 ACM Subject Classification D.1.6 Logic Programming, F.3.1 Specifying and Verifying and Reasoning about Programs, D.2.4 Software/Program Verification, D.2.5 Testing and Debugging

Keywords and phrases program correctness, program completeness, specification, declarative programming, declarative diagnosis.

Digital Object Identifier 10.4230/LIPIcs.ICLP.2012.301

1 Introduction

The purpose of this paper is to show to which extent the correctness related issues of a Prolog program can, in practice, be dealt with mathematical precision. We present a construction of a useful Prolog program. We view it as a logic program with added control. We formally prove that the logic program conforms to its specification and partly informally justify that adding control preserves this property. We argue that the employed methods are not difficult and can be used by actual programmers.

Howe and King [11] presented a SAT solver which is an elegant and concise Prolog program of 22 lines. It is not a (pure) logic program, as it includes `nonvar/1` and the if-then-else of Prolog; it was constructed as an implementation of an algorithm, using logical variables and coroutining. The algorithm is DPLL with watched literals and unit propagation (see [11] for references). Here we look at the program from a declarative point of view. We show how it can be obtained by adding control to a definite clause logic program.

We first present a simple logic program of five clauses, and then modify it in order to obtain a logic program on which the intended control can be imposed. The control involves fixing the selection rule (by means of the delay mechanisms of Prolog), and pruning some redundant fragments of the search space. In constructing both the introductory program and the final one, we begin with a specification, describing the relations to be defined by the program. We argue about usefulness of approximate specifications. For both logic

© Włodzimierz Drabent;
licensed under Creative Commons License ND
Technical Communications of the 28th International Conference on Logic Programming (ICLP'12).
Editors: A. Dovier and V. Santos Costa; pp. 301–311

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

programs we present formal proofs of their correctness and completeness. In the second case the proofs are performed together with the construction of the program. Both programs terminate under any selection rule. Adding control preserves correctness and termination. Completeness of the final program with control is justified partly informally.

To facilitate the proofs we present the underlying proof methods for correctness and completeness. For proving correctness we use the method of [4]; for completeness – a simplification of the method of [8]. We also employ a method of proving that a certain kind of pruning SLD-trees preserves completeness (from [7], an extended version of this paper).

Preliminaries. In this paper we consider definite clause programs (i.e. programs without negation). We use the standard notation and definitions, see e.g. [1]. In our main examples we assume a Herbrand universe $\mathcal{H}$ like in Prolog, based on an alphabet of infinitely many function symbols of each arity ≥ 0. However the theoretical considerations of Sect. 3 are valid for arbitrary nonempty Herbrand universe. By $ground(P)$ we mean the set of ground instances of a program P (under a given Herbrand universe).

We use the Prolog notation for lists. Names of variables begin with an upper-case letter. By a list we mean a term of the form $[t_1, \ldots, t_n]$ (so terms like $[a, a|X]$, or $[a, a|a]$ are not considered lists). As we deal with clauses as data, and clauses of programs, the latter will be called *rules* to avoid confusion. Given a predicate symbol p, by an *atom for* p we mean an atom whose predicate symbol is p, and by a *rule for* p – a rule whose head is an atom for p. By a *procedure* p we mean all the rules for p in the program under consideration.

Organization of the paper. The next section presents a simple and inefficient SAT solver. Section 3 formalizes the notion of a specification, and presents methods for proving program correctness and completeness. In Section 4 the final logic program is constructed hand in hand with its correctness and completeness proof. Section 5 considers adding control to the program. Section 6 discusses the presented approach and its relation to declarative diagnosis.

2 Propositional satisfiability – first logic program

Representation of propositional formulae. We first present the encoding of propositional formulae in CNF as terms, proposed by [11] and used in this paper.

Propositional variables are represented as logical variables; truth values – as constants `true`, `false`. A literal of a clause is represented as a pair of a truth value and a variable; a positive literal, say x, as `true-X` and a negative one, say $\neg x$, as `false-X`. A clause is represented as a list of (representations of) literals, and a conjunction of clauses as a list of their representations. For instance a formula $(x \vee \neg y \vee z) \wedge (\neg x \vee v)$ is represented as `[[true-X,false-Y,true-Z],[false-X,true-V]]`.

An assignment of truth values to variables can be represented as a substitution. Thus a clause (represented by) f is true under an assignment (represented by) θ iff the list $f\theta$ has an element of the form t-t, i.e. `false-false` or `true-true`. A formula in CNF is satisfiable iff its representation has an instance whose each element (is a list which) contains a t-t. We will often say “formula f” for a formula in CNF represented as a term f, similarly for clauses etc.

The program. Now we construct a simple logic program checking satisfiability of such formulae. We begin with describing the (unary) relations to be defined by the program. Let

$$L_1^0 = \{\, [t_1\text{-}u_1, \ldots, t_n\text{-}u_n] \in \mathcal{H} \mid n > 0,\ t_i = u_i \text{ for some } i \in \{1, \ldots, n\} \,\},$$
$$L_2^0 = \{\, [s_1, \ldots, s_n] \mid n \geq 0,\ s_1, \ldots, s_n \in L_1^0 \,\}.$$

(It may be additionally required that all t_j, u_j are in {`true`, `false`}, we do not impose this restriction). A clause f is true under an assignment θ iff the list $f\theta$ is in L_1^0. A formula in CNF is satisfiable iff it has an instance in L_2^0. However L_2^0 is not a unique set with this property. Moreover, a program defining (exactly) L_2^0 would be unnecessarily complicated and would involve unnecessary computations (like checking if the elements of a list are indeed closed lists of pairs). So we extend L_2^0.

A list of the form $[t_1\text{-}u_1, \ldots, t_n\text{-}u_n]$ $(n \geq 0)$ will be called a *list of pairs*. Let

$$L_1 = \left\{ t \in \mathcal{H} \mid \text{ if } t \text{ is a list of pairs then } t \in L_1^0 \right\},$$
$$L_2 = \left\{ s \in \mathcal{H} \mid \text{ if } s \text{ is a list of lists of pairs then } s \in L_2^0 \right\}.$$

Note that $L_1^0 \subseteq L_1, L_2^0 \subseteq L_2$, and that for any set L_2' such that $L_2^0 \subseteq L_2' \subseteq L_2$ it holds:

$$\text{A formula in CNF is satisfiable iff it has an instance in } L_2'. \tag{1}$$

(Because any its instance from L_2 is also in L_2^0, as the formula is a list of lists of pairs). Thus a program computing any such set L_2' would do.

A program P_1 defining such L_2' is constructed in a rather obvious way. Its main procedure is *sat_cnf*. It employs *sat_cl*, which defines an L_1' such that $L_1^0 \subseteq L_1' \subseteq L_1$. In the next section we prove that the sets defined by the program indeed satisfy these inclusions.

$$sat_cnf([\,]). \tag{2}$$
$$sat_cnf([Clause|Clauses]) \leftarrow sat_cl(Clause),\ sat_cnf(Clauses). \tag{3}$$
$$sat_cl([Pol\text{-}Var|Pairs]) \leftarrow Pol = Var. \tag{4}$$
$$sat_cl([H|Pairs]) \leftarrow sat_cl(Pairs). \tag{5}$$
$$=(X, X). \tag{6}$$

Let f be (a representation of) a CNF formula. Then $sat_cnf(f)$ succeeds iff f is satisfiable.

3 Correctness and completeness

Now we show how to prove that a program indeed defines the required relations. Basically we follow the approach of [8]. We present a special case of the correctness criterion used there, and we simplify and extend the method of proving completeness; see [7] for a wider presentation, and for proofs of the theorems.

3.1 Specifications

We provided a specification for the program P_1 by describing a set for each predicate; the predicate should define this set. In a general case, for an n-argument predicate p the specification describes an n-argument relation, to be defined by p. The specification in Sect. 2 is *approximate*: the relations are described not exactly, each one is specified by giving its superset and subset. It is convenient to view an approximate specification as two specifications (in our example the first one specifies L_1, L_2, and the second one L_1^0, L_2^0). The former describes the tuples that are allowed to be computed, the latter those that have to be computed. The former is related to program correctness, the latter to completeness.

In our example it was impossible to provide an exact specification, as it was not known which of the possible relations should be implemented. The usual procedure *append* provides a somehow different example of usefulness of approximate specifications [8]. In that case the relation is known, but it is not necessary (and a bit cumbersome) to specify it exactly.

To make it explicit which relation corresponds to which predicate, specifications will be represented as Herbrand interpretations. A (formal exact) **specification** is a Herbrand interpretation; given a specification S, each $A \in S$ is called a *specified* atom (by S). The fact that $p(t_1,\ldots,t_n) \in S$ is understood as that the tuple $(t_1,\ldots,t_n)$ is in the relation corresponding to p.

So the approximate specification in the example of Sect. 2 consists of two specifications S_1 and S_1^0 with the specified atoms of the form, respectively:

$$S_1\colon \begin{array}{lll} \mathit{sat_cnf}(t), & \text{where} & t \in L_2, \\ \mathit{sat_cl}(s), & & s \in L_1, \\ x = x, & & x \in \mathcal{H} \end{array} \qquad (7) \qquad\qquad S_1^0\colon \begin{array}{lll} \mathit{sat_cnf}(t), & \text{where} & t \in L_2^0, \\ \mathit{sat_cl}(s), & & s \in L_1^0, \\ x = x, & & x \in \mathcal{H} \end{array} \qquad (8)$$

Correctness and completeness. In imperative programming, correctness usually means that the program results are as specified. In logic programming, due to its non-deterministic nature, we actually have two issues: *correctness* (all the results are compatible with the specification) and *completeness* (all the results required by the specification are produced). In other words, correctness means that the relation defined by the program is a subset of the specified one, and completeness means inclusion in the opposite direction. Given a specification S and a program P, with its least Herbrand model M_P, we have: P is **correct** w.r.t. S iff $M_P \subseteq S$; it is **complete** w.r.t. S iff $M_P \supseteq S$.

Notice that if a program P is both correct and complete w.r.t. S then $M_P = S$ and the specification describes exactly the relations defined by P. An approximate specification, given by a pair S_{corr}, S_{compl} of Herbrand interpretations, means that for one of them the program has to be correct, for the other – complete. Formally, it is required that $S_{compl} \subseteq M_P \subseteq S_{corr}$.

It is useful to relate correctness and completeness with answers of programs.[1]

▶ Proposition 1. Let P be a program, Q a query, and S a specification.

If P is correct w.r.t. S and $Q\theta$ is an answer for P then $S \models Q\theta$.

If P is complete w.r.t. S and $S \models Q\sigma$, for a ground $Q\sigma$, then $Q\sigma$ is an answer for P, and is an instance of some computed answer for P and Q.

3.2 Correctness

To prove correctness we use the following property [4]; see [8] for further explanations.

▶ **Theorem 2** (Correctness). *A sufficient condition for a program P to be correct w.r.t. a specification S is $S \models P$.*

Note that $S \models P$ means that for each ground instance $H \leftarrow B_1,\ldots,B_n$ of a rule of P, if $B_1,\ldots,B_n \in S$ then $H \in S$.

Using Th. 2, it is easy to show that P_1 is correct w.r.t. S_1. For instance consider rule (5), and its arbitrary ground instance $\mathit{sat_cl}([u|s]) \leftarrow \mathit{sat_cl}(s)$, such that $\mathit{sat_cl}(s) \in S_1$. If $[u|s]$ is a list of pairs then s is; thus $s \in L_1^0$, and $[u|s] \in L_1^0$. So $[u|s] \in L_1$, and $\mathit{sat_cl}([u|s]) \in S_1$. We leave the rest of the proof to the reader.

---

[1] By a computed (respectively correct) **answer** for a program P and a query Q we mean an instance $Q\theta$ of Q where θ is a computed (correct) answer substitution [1] for Q and P. We often say just "answer", as each computed answer is a correct one, and each correct answer (for Q) is a computed answer (for Q or for some its instance $Q\sigma$). Thus, by soundness and completeness of SLD-resolution, $Q\theta$ is an answer for P iff $P \models Q\theta$.

3.3 Completeness

We begin with introducing a few auxiliary notions. Let us say that a program P is **complete for a query** $Q = A_1, \ldots, A_n$ w.r.t. S when $A_1\theta, \ldots, A_n\theta \in S$ implies $A_1\theta, \ldots, A_n\theta \in M_P$, for any ground instance $Q\theta$ of Q. Informally, complete for Q means that all the answers for Q required by the specification are computed. Note that a program is complete w.r.t. S iff it is complete w.r.t. S for any query iff it is complete w.r.t. S for any query $A \in S$.

We also say that a program P is **semi-complete** w.r.t. S if P is complete for any query Q for which there exists a finite SLD-tree. Note that the existence of a finite SLD-tree means that P with Q terminates under some selection rule. For a semi-complete program, if a computation for a query Q terminates then all the required by the specification answers for Q have been obtained. Here are conditions under which "semi-complete" implies "complete."

▶ Proposition 3. Let a program P be semi-complete w.r.t. S. P is complete w.r.t. S if

1. for each ground atomic query $A \in S$ there exists a finite SLD-tree, or
2. the program is recurrent or acceptable [1, Chapter 6].

A ground atom H is called **covered** [13] by a program P w.r.t. a specification S if H is the head of a ground instance $H \leftarrow B_1, \ldots, B_n$ of a rule of the program, such that all the atoms $B_1, \ldots, B_n$ are in S. For instance, given a specification $S = \{\, p(s^i(0)) \mid i \geq 0 \,\}$, atom $p(s(0))$ is covered both by a program $\{\, p(s(X)) \leftarrow p(X).\}$ and by $\{\, p(X) \leftarrow p(s(X)).\}$.

Now we are ready to present a sufficient condition for completeness.

▶ **Theorem 4** (Completeness). *Let P be a program, and S a specification.*
If all the atoms from S are covered by P then P is semi-complete w.r.t. S.

Hence, if such P satisfies one of the conditions from Prop. 3 then it is complete w.r.t. S.

Let us apply Th. 4 to our program. First let us show that all the atoms from S_1^0 are covered by P_1 (and thus P_1 is semi-complete). For instance consider a specified atom $A = sat_cnf(t)$. Thus $t \in L_2^0$. If t is nonempty then $t = [s|t']$, where $s \in L_1^0$, $t' \in L_2^0$. Hence a ground instance $A \leftarrow sat_cl(s), sat_cnf(t')$ of a clause of P_1 has all its body atoms in S_1^0, so A is covered. If t is empty then A is covered as it is the head of the rule $sat_cnf([\,])$. The reasoning for the remaining atoms of S_1 is similar, and left to the reader.

So the program is semi-complete w.r.t. S_1, and it remains to show its termination. An informal justification is that, for an intended initial query (or for an arbitrary ground initial query), the predicates are invoked with (closed) lists as arguments, and each recursive call employs a shorter list. For a formal proof that the program is recurrent [2],[1, Chapter 6.2], see [7]. Thus by Proposition 3, P_1 is complete w.r.t. S_1.

4 Preparing for adding control

To be able to influence the control of program P_1 in the intended way, in this section we construct a more sophisticated logic program P_3, with a program P_2 as an initial stage. The construction is guided by a formal specification, and done together with a correctness and semi-completeness proof. Most of the details are presented. However efficiency issues are outside of the scope of this work.

As explained in Sect. 2, it is sufficient that sat_cnf defines a set L_{sat_cnf} such that $L_2^0 \subseteq L_{sat_cnf} \subseteq L_2$ (similarly, sat_cl defines L_{sat_cl}, where $L_1^0 \subseteq L_{sat_cl} \subseteq L_1$). The rules for sat_cnf and $=$ from P_1, i.e. (2), (3), (6), are included in P_2. We modify the definition of sat_cl, introducing some new predicates. The new predicates would define the same propositional clauses as sat_cl, but represented in a different way.

To simplify the presentation, we provide now the specification for the new predicates. Explanations are given later on, while introducing each predicate. In the specification for correctness (respectively completeness) the new specified atoms are

$$\begin{array}{lll} \mathit{sat_cl3}(s,v,p), & \text{where} & [p\text{-}v|s] \in L_1 \text{ (respectively } \in L_1^0), \\ \mathit{sat_cl5}(v_1,p_1,v_2,p_2,s), & & \\ \mathit{sat_cl5a}(v_1,p_1,v_2,p_2,s), & & [p_1\text{-}v_1, p_2\text{-}v_2|s] \in L_1 \text{ (respectively } \in L_1^0). \end{array} \tag{9}$$

So specification S_2 for correctness is obtained by adding these literals to specification S_1 (cf. (7)), and specification S_2^0 for completeness – by adding to S_1^0 (cf. (8)) the literals of (9) with L_1 replaced by L_1^0. Note that $S_2^0 \subseteq S_2$.

In what follows, SC1 stands for the sufficient condition from Th. 2 for correctness w.r.t. S_2, and SC2 – for the sufficient condition from Th. 4 for semi-completeness w.r.t. S_2^0 (i.e. each atom from S_2^0 is covered). While discussing a procedure p, we consider SC2 for atoms of the form $p(\dots)$ from S_2^0. Let SC stand for SC1 and SC2. We perform the correctness and completeness proof hand in hand with introducing new rules of P_3. When checking a corresponding SC is not mentioned, it is simple and left to the reader. SC for *sat_cnf* and $=$ have been already shown.

Program P_1 performs inefficient search by means of backtracking. We are going to improve it by delaying unification of pairs *Pol*-*Var* in *sat_cl*. The idea is to perform such unification if *Var* is the only unbound variable of the clause. Otherwise, *sat_cl* is to be delayed until one of the first two variables of the clause becomes bound to `true` or `false`.

This idea will be implemented by separating two cases: the clause has one literal, or more. We want to distinguish these two cases by means of indexing the main symbol of the first argument. So the argument should be the tail of the list. We redefine *sat_cl*, introducing an auxiliary predicate *sat_cl*3. It defines the same set as *sat_cl*, but a clause $[Pol\text{-}Var|Pairs]$ is represented as three arguments $Pairs, Var, Pol$ of *sat_cl*3.

$$\mathit{sat_cl}([Pol\text{-}Var|Pairs]) \leftarrow \mathit{sat_cl3}(Pairs, Var, Pol). \tag{10}$$

Procedure *sat_cl*3 has to cover each atom $A = \mathit{sat_cl3}(s,v,p) \in S_2^0$, i.e. each A such that $[p\text{-}v|s] \in L_1^0$. Assume first $s = [\,]$. Then $p = v$; this suggests a rule

$$\mathit{sat_cl3}([\,], Var, Pol) \leftarrow Var = Pol. \tag{11}$$

Its ground instance $\mathit{sat_cl3}([\,],p,p) \leftarrow p{=}p$ covers A w.r.t. S_2^0. Conversely, each instance of (11) with the body atom in S_2 is of this form, its head is in S_2, hence SC1 holds.

When the first argument of *sat_cl*3 is not $[\,]$, we want to delay $\mathit{sat_cl3}(Pairs, Var, Pol)$ until Var or the first variable of $Pairs$ is bound. In order to do this in, say, Sicstus, we need to make the two variables to be separate arguments of a predicate. So we introduce a five-argument predicate *sat_cl*5, which is going to be delayed. It defines the set of the lists from L_{sat_cl} of length greater than 1; however a list $[Pol1\text{-}Var1, Pol2\text{-}Var2\,|\,Pairs]$ is represented as the five arguments $Var1, Pol1, Var2, Pol2, Pairs$ of *sat_cl*5. The intention is to delay selecting *sat_cl*5 until its first or third argument is bound (is not a variable). So the following rule completes the definition of *sat_cl*3.

$$\mathit{sat_cl3}([Pol2\text{-}Var2|Pairs], Var1, Pol1) \leftarrow \mathit{sat_cl5}(Var1, Pol1, Var2, Pol2, Pairs). \tag{12}$$

To check SC, let $S = S_2, L = L_1$ or $S = S_2^0, L = L_1^0$. For each ground instance of (12) the body is in S iff the head is in S. Hence SC1 holds for (12), and each $\mathit{sat_cl3}([p_2\text{-}v_2|s], v_1, p_1) \in S_2^0$ where $s \neq [\,]$ is covered by (12). So SC2 for *sat_cl*3 holds, due to (11) and (12).

In evaluating sat_cl5, we want to treat the bound variable (the first or the third argument) in a special way. So we make it the first argument of a new predicate sat_cl5a, with the same declarative semantics as sat_cl5.

$$sat_cl5(Var1, Pol1, Var2, Pol2, Pairs) \leftarrow sat_cl5a(Var1, Pol1, Var2, Pol2, Pairs). \quad (13)$$
$$sat_cl5(Var1, Pol1, Var2, Pol2, Pairs) \leftarrow sat_cl5a(Var2, Pol2, Var1, Pol1, Pairs). \quad (14)$$

SC are trivially satisfied. Moreover, SC2 is satisfied by each of the two rules alone. The control will choose the one that results in invoking sat_cl5a with its first argument bound.

To build a procedure sat_cl5a we have to provide rules which cover each atom $A = sat_cl5a(v_1, p_1, v_2, p_2, s) \in S_2^0$. Note that $A \in S_2^0$ iff $[p_1\text{-}v_1, p_2\text{-}v_2|s] \in L_1^0$ iff $p_1 = v_1$ or $[p_2\text{-}v_2|s] \in L_1^0$ iff $p_1 = v_1$ or $sat_cl3(s, v_2, p_2) \in S_2^0$. So two rules follow

$$sat_cl5a(Var1, Pol1, Var2, Pol2, Pairs) \leftarrow Var1 = Pol1. \quad (15)$$
$$sat_cl5a(Var1, Pol1, Var2, Pol2, Pairs) \leftarrow sat_cl3(Pairs, Var2, Pol2). \quad (16)$$

and SC2 holds for sat_cl5a. To check SC1, consider a ground instance of (15), with the body atom in S_2: $sat_cl5a(p, p, v_2, p_2, s) \leftarrow p = p$. As $[p\text{-}p, p_2\text{-}v_2|s] \in L_1$, the head of the clause is in S_2. Take a ground instance $sat_cl5a(v_1, p_1, v_2, p_2, s) \leftarrow sat_cl3(s, v_2, p_2)$. of (16), with the body atom in S_2. Then its head is in S_2, as $[p_2\text{-}v_2|s] \in L_1$ implies $[p_1\text{-}v_1, p_2\text{-}v_2|s] \in L_1$.

From a declarative point of view, our program is ready. The logic program P_2 consists of rules (2), (3), (6), and (10) – (16). It is correct w.r.t. S_2 and semi-complete w.r.t. S_2^0.

Avoiding floundering. When selecting sat_cl5 is delayed as described above, program P_2 may flounder; a nonempty query with no selected atom may appear in a computation. Floundering is a kind of pruning SLD-trees, and may cause incompleteness. To avoid it, we add a top level predicate sat. It defines the relation (a Cartesian product) in which the first argument is as defined by sat_cnf, and the second argument is a list of truth values.

$$sat(Clauses, Vars) \leftarrow sat_cnf(Clauses), tflist(Vars). \quad (17)$$

(Predicate $tflist$ will define the set of truth value lists.) The intended initial queries are of the form

$$sat(f, l), \text{ where } f \text{ is a (representation of a) propositional formula, } l \text{ is the list of variables in } f. \quad (18)$$

Such query succeeds iff the formula f is satisfiable. Floundering is avoided, as $tflist$ will eventually bind all the variables of f. More precisely, consider a node Q in an arbitrary SLD-tree for a $sat(f, l)$ of (18). We have three cases. (i) Q is the root, or its child. (ii) Q contains an atom derived from $tflist(l)$. Otherwise, (iii) the variables of l (and thus those of f) are bound; hence all the atoms of Q are ground (as no rule of P_2 introduces a new variable). So no such Q consists solely of non-ground sat_cl5 atoms.

We use auxiliary predicates to define the set of truth values, and the set of the lists of truth values. The extended formal specification S_3 for correctness consists of atoms

$$\begin{array}{lll} sat(t, u), & tflist(u), & \text{where } t \in L_2,\ u \text{ is a list whose elements are } \mathtt{true} \text{ or } \mathtt{false}, \\ tf(\mathtt{true}), & tf(\mathtt{false}), & \end{array} \quad (19)$$

and of those of S_2 (i.e. the atoms of (7), (9)). The extended specification S_3^0 for completeness consists of S_2^0 and of the atoms described by a modified (19) where L_2 is replaced by L_2^0. The three new predicates are defined in a rather obvious way, following [11]:

$$tflist([\,]). \qquad (20) \qquad\qquad tf(true). \qquad (22)$$

$$tflist([Var|Vars]) \leftarrow tflist(Vars), tf(Var). \qquad (21) \qquad\qquad tf(false). \qquad (23)$$

This completes our construction. The logic program P_3 consists of rules (2), (3), (6), (10) – (17), and (20) – (23). It is correct w.r.t. S_3 and semi-complete w.r.t. S_3^0. It terminates for the intended queries, under any selection rule, as it is recurrent under a suitable level mapping, see [7]. Thus by Prop. 3, the program is complete w.r.t. S_3^0.

5 The program with control

In this section we add control to program P_3. As the result we obtain the Prolog program of Howe and King [11]. (The predicate names differ, those in the original program are related to its operational semantics.) The idea is that P_3 with this control implements the DPLL algorithm with watched literals and unit propagation.[2]

The control added to P_3 modifies the default Prolog selection rule, and prunes some redundant parts of the search space (by the if-then-else construct). So correctness and termination of P_3 are preserved (as we proved termination for any selection rule).

To delay *sat_cl5* until its first or third argument is not a variable we use a declaration

```
:- block sat_cl5(-, ?, -, ?, ?).                                    (24)
```

of Sicstus. As informally discussed in Sect. 4, for the intended initial queries floundering is avoided; thus the completeness of P_3 is preserved.

The first case of pruning is to use only one of the two rules (13), (14), the one which invokes *sat_cl5a* with the first argument bound. According to [7, Corollary 6], this pruning preserves completeness (see [7] for a proof). The pruning is implemented by employing the *nonvar* built-in and the if-then-else construct of Prolog:

$$\begin{array}{l} sat_cl5(Var1, Pol1, Var2, Pol2, Pairs) \leftarrow \\ \quad nonvar(Var1) \rightarrow sat_cl5a(Var1, Pol1, Var2, Pol2, Pairs); \\ \qquad\qquad\qquad\qquad sat_cl5a(Var2, Pol2, Var1, Pol1, Pairs). \end{array} \qquad (25)$$

An efficiency improvement related to rules (15), (16) is possible. Procedure *sat_cl5a* is invoked with the first argument $Var1$ bound. If the first argument of the initial query $sat(f, l)$ is a (representation of a) propositional formula then *sat_cl5a* is called with its second argument $Pol1$ being `true` or `false`. So the unification $Var1 = Pol1$ in (15) works as a test, and the rule binds no variables.[3] Thus after a success of rule (15) there is no point in invoking (16), as the success of (15) produces the most general answer for $sat_cl5a(\ldots)$, which subsumes any other answer. Hence the search space can be pruned accordingly. We do this by converting the two rules into

$$\begin{array}{l} sat_cl5a(Var1, Pol1, Var2, Pol2, Pairs) \leftarrow \\ \quad Var1 = Pol1 \; \rightarrow \; true; \quad sat_cl3(Pairs, Var2, Pol2). \end{array} \qquad (26)$$

This completes our construction. The obtained Prolog program consists of declaration (24), the rules of P_3 except for those for *sat_cl5* and *sat_cl5a*, i.e. (2), (3), (6), (10) – (12), (17), (20) – (23), and Prolog rules (25), (26). It is correct w.r.t. S_3, and is complete w.r.t. S_3^0 for queries of the form (18).

---

[2] However, when a non-watched literal in a clause becomes true, the clause is not immediately removed.
[3] So = may be replaced by the built-in `==`, as in [11].

6 Discussion

Proof methods. The correctness proving method of [4] (further references in [8]) used here should be well-known, but is often neglected. For instance, an important monograph [1] uses a more complicated method (of [3]), which refers to the operational semantics (LD-resolution). See [8] for comparison and argumentation that the simpler method is sufficient.

Proving completeness has been seldom considered, especially within a framework of declarative semantics. For instance it is not discussed in [1]. (Instead, for a program P and an atomic query A, a characterization of the set of computed instances of A is studied, in a special case of the set being finite and the answers ground [1, Sect. 8.4].) Book [5] presents criteria for program completeness, in a sophisticated framework of relating logic programming and attribute grammars. The method presented here (Sect. 3.3, [7]) is a simplification of that from [8] (an initial version appeared in [6]). Our notion of completeness is slightly different, and programs with negation are excluded. We introduce a notion of semi-completeness, for which the corresponding sufficient condition deals with program procedures separately, while for completeness the whole program has to be taken into account.

Correctness and completeness are declarative properties, they are independent from the operational semantics. If dealing with them required reasoning in terms of operational semantics then logic programming would not deserve to be meant a declarative programming paradigm. The sufficient criteria of Th. 2, 4 for correctness and semi-completeness are purely declarative, they treat program rules as logical formulae, and abstract from any operational notions. However proving completeness refers to program termination. The reason is that in practice termination has to be concerned anyway, and a pure declarative approach to completeness [5, Th. 6.1] seems more complicated [7] (and it includes a condition similar to those for proving termination). Note that semi-completeness alone may be a useful property, as it guarantees that whenever the computation terminates, all the required answers have been computed.

We want to stress the simplicity and naturalness of the sufficient conditions for correctness and semi-completeness (Th. 2, 4). Informally, the first one says that the rules of a program should produce only correct conclusions, given correct premises. The other says that each ground atom that should be produced by P has to be the head of a rule instance, whose body atoms should be produced by P too. The author believes that this is a way a competent programmer reasons about (the declarative semantics of) a logic program.

Specifications. The examples of programs P_1 and P_3 show usefulness of approximate specifications (p. 303). They are crucial for avoiding unnecessary complications in constructing specifications and in correctness and completeness proofs. They are natural: when starting construction of a program, the relations it should compute are often known only approximately. Also, it is often difficult (and unnecessary) to exactly establish the relations computed by a program. As an example, the reader may try to describe the two (distinct) sets defined by the main procedures of P_1 and P_2 (cf. [7], where M_{P_1} is given.)

Specifications which are interpretations (as here and in [1]) have a limitation. They cannot express that e.g. for a given a there exists a b such that $p(a, b)$. In our case, we could not specify that it is sufficient for a SAT solver to find some variable assignment satisfying f, whenever f is satisfiable. Our specifications S_1^0, S_3^0 require that all such assignments are found. The problem seems to be solved by introducing specifications in a form of logical theories (where axioms like $\exists b. \, p(a, b)$ can be used). This idea is present in [5, 8].

Relations to declarative diagnosis. Declarative diagnosis methods (called sometimes declarative debugging) [13] (see also [9, 12] and references therein) locate in a program the

reason for its incorrectness or incompleteness. A diagnosis algorithm begins with a symptom (obtained from testing the program): an answer Q such that $S \not\models Q$, or a query Q for which computation terminates but some answers required by S are not produced. The located error turns out to be the program fragment (a rule or a procedure) which violates our sufficient condition for correctness or, respectively, semi-completeness. Roughly speaking, the diagnosis algorithm actually checks the sufficient conditions of Th. 2 (Th. 4), but only for some instances of program rules (for some specified atoms) – those involved in producing the symptom. (See [7] for further discussion.)

An attempt to prove a buggy program to be correct (complete) results in violating the corresponding sufficient condition for some rule (specified atom). For instance, in this way the author found an error in a former version of P_1 (there was $[Pairs]$ instead of $Pairs$). Any error located by diagnosis will also be found by a proof attempt; moreover no symptom is needed, and all the errors are found. However the sufficient condition has to be checked for all the rules of the program (for all specified atoms).

A serious difficulty in using declarative diagnosis methods is that an exact specification (a single intended model) of the program is needed. Then answering some diagnoser queries, like "is $append([a], b, [a|b])$ correct", may be difficult, as the programmer often does not know some details of the intended model, like those related to applying *append* on non-lists. The problem has been pointed out in [9] and discussed in [12] (see also references therein). A solution is to employ approximate specifications; incorrectness diagnosis should use the specification for correctness, and incompleteness diagnosis that for completeness. This seems simpler than introducing new diagnosis algorithms based on three logical values [12].

7 Conclusions

The central part of this paper is an example of a systematic construction of a Prolog program: the SAT solver of [11]. Starting from a formal specification, a definite clause program, called P_3, is constructed hand in hand with a proof of its correctness and completeness (Sect. 4). The final Prolog program is obtained from P_3 by adding control (delays and pruning SLD-trees, Sect. 5). Correctness, completeness and termination of a pure logic program can be dealt with formally, and we proved them for P_3. Adding control preserves correctness and (in this case) termination. We partly proved, and partly justified informally that completeness is preserved too. We point out usefulness of approximate specifications.

The employed proof methods are of separate interest. The method for correctness [4] is simple, should be well-known, but is often neglected. A contribution of this work is a method for proving completeness (Sect. 3.3, [7]), a simplification of that of [8]. Due to lack of space, taking pruning into account in proving completeness [7] is not discussed here.

We are interested in declarative programming. Our main example was intended to show how much of the programming task can be done without considering the operational semantics, how "logic" could be separated from "control." A substantial part of work could be done at the stage of a pure logic program, where correctness, completeness and termination could be dealt with formally. It is important that all the considerations and decisions about the program execution and efficiency (only superficially treated here) are independent from those related to the declarative semantics, to the correctness of the final program, and – to a substantial extent – its completeness.

We argue that the employed proof methods are simple, and correspond to a natural way of declarative thinking about programs. We believe that they can be actually used – maybe at an informal level – in practical programming; this is supported by our main example.

References

1. K. R. Apt. *From Logic Programming to Prolog.* International Series in Computer Science. Prentice-Hall, 1997.
2. M. Bezem. Strong termination of logic programs. *J. Log. Program.*, 15(1&2):79–97, 1993.
3. A. Bossi and N. Cocco. Verifying correctness of logic programs. In J. Díaz and F. Orejas, editors, *TAPSOFT, Vol.2*, volume 352 of *Lecture Notes in Computer Science*, pages 96–110. Springer, 1989.
4. K. L. Clark. Predicate logic as computational formalism. Technical Report 79/59, Imperial College, London, December 1979.
5. P. Deransart and J. Małuszyński. *A grammatical view of logic programming.* MIT Press, 1993.
6. W. Drabent. It is declarative. In *Logic Programming: The 1999 International Conference*, page 607. The MIT Press, 1999. Poster abstract. A technical report at `http://www.ipipan.waw.pl/~drabent/itsdeclarative3.pdf`.
7. W. Drabent. Logic + control: An example of program construction. *CoRR*, arXiv:1110.4978 [cs.LO], 2012. Corrected version. `http://arxiv.org/abs/1110.4978`.
8. W. Drabent and M. Miłkowska. Proving correctness and completeness of normal programs – a declarative approach. *Theory and Practice of Logic Programming*, 5(6):669–711, 2005.
9. W. Drabent, S. Nadjm-Tehrani, and J. Małuszyński. Algorithmic Debugging with Assertions. In H. Abramson and M. H. Rogers, editors, *Meta-Programming in Logic Programming*, pages 501–522. The MIT Press, 1989.
10. J. M. Howe and A. King. A pearl on SAT solving in Prolog (extended abstract). *Logic Programming Newsletter*, 24(1), March 31 2011. `http://www.cs.nmsu.edu/ALP/2011/03/a-pearl-on-sat-solving-in-prolog-extended-abstract/`.
11. J. M. Howe and A. King. A pearl on SAT and SMT solving in Prolog. *Theoretical Computer Science*, 2012. Special Issue on FLOPS 2010. Available online `http://dx.doi.org/10.1016/j.tcs.2012.02.024`. An earlier version is [10].
12. L. Naish. A three-valued declarative debugging scheme. In *23rd Australasian Computer Science Conference (ACSC 2000)*, pages 166–173. IEEE Computer Society, 2000.
13. E. Shapiro. *Algorithmic Program Debugging.* MIT Press, 1983.

Deriving a Fast Inverse of the Generalized Cantor N-tupling Bijection

Paul Tarau

Dept. of Computer Science and Engineering
University of North Texas, Denton, Texas, USA
`tarau@cs.unt.edu`

Abstract

We attack an interesting open problem (an efficient algorithm to invert the generalized Cantor N-tupling bijection) and solve it through a sequence of equivalence preserving transformations of logic programs, that take advantage of unique strengths of this programming paradigm. An extension to set and multiset tuple encodings, as well as a simple application to a "fair-search" mechanism illustrate practical uses of our algorithms.

The code in the paper (a literate Prolog program, tested with SWI-Prolog and Lean Prolog) is available at `http://logic.cse.unt.edu/tarau/research/2012/pcantor.pl`.

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

Keywords and phrases generalized Cantor n-tupling bijection, bijective data type transformations, combinatorial number system, solving combinatorial problems in Prolog, optimization through program transformation, logic programming and software engineering

Digital Object Identifier 10.4230/LIPIcs.ICLP.2012.312

1 Introduction

It is by no means a secret that logic programming is an ideal paradigm for solving combinatorial problems. Built-in backtracking, unification and availability of constraint solvers facilitates quick prototyping for problems involving search or generation of combinatorial objects. It also provides an easy path from executable specification to optimal implementation through a well-understood set of program transformations. From a software engineering perspective, problem solving with help of logic programming tools is a natural fit to *agile development* practices as it encourages a fast moving iterative process consisting of incremental refinements.

This paper reports on tackling a somewhat atypical problem solving instance: finding a fast inverse of a generalization of Cantor's pairing bijection to n-tuples. This generalization is mentioned in two relatively recent papers [2, 7] with a possible attribution in [2] to Skolem as a first reference.

The formula, given in [2] p.4, looks as follows:

$K_n(x_1,\ldots,x_n) = \binom{n-1+x_1+\ldots+x_n}{n} + \cdots + \binom{k-1+x_1+\ldots+x_k}{k} + \ldots + \binom{1+x_1+x_2}{2} + \binom{x_1}{1}$

where $\binom{n}{k}$ represents the number of subsets of k elements of a set of n elements and $K_n(x_1,\ldots,x_n)$ denotes the natural number associated to the tuple $x_1,\ldots,x_n$. So the problem of inverting it means finding a solution of the *Diophantine equation*

$$\binom{x_1}{1} + \binom{1+x_1+x_2}{2} + \ldots + \binom{n-1+x_1+\ldots+x_n}{n} = z \quad (1)$$

© Paul Tarau;
licensed under Creative Commons License ND
Technical Communications of the 28th International Conference on Logic Programming (ICLP'12).
Editors: A. Dovier and V. Santos Costa; pp. 312–322

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

and proving that it is unique. Unfortunately, despite extensive literature search, we have not found any attempt to devise an algorithm that computes the inverse of the function K_n, so we had to accept the fact that we were looking at an *open problem* with possibly interesting implications, given that for $n = 2$ it reduces to Cantor's pairing function that has been used in hundreds of papers on foundations of mathematics, logic, recursion theory as well as in some practical applications (dynamic n-dimensional arrays) like [11].

As an inductive proof that K_n is a bijection is given in [7] (Theorem 2.1), we know that a solution exists and is unique, so the problem reduces to computing the first solution of the Diophantine equation (1).

Unfortunately, solving an arbitrary Diophantine equation is Turing-equivalent. This is a consequence of the negative answer to Hilbert's 10-th problem, proven by Matiyasevich [8], based on earlier work by Robinson, Davis and Putnam [4, 10], and some fairly simple instances of it, like Fermat's $\exists x, y, z > 0, \exists n \geq 3, x^n + y^n = z^n$ have waited for centuries before being solved.

On the other hand, things do not look that bad in this case, as it is easy to show that in the equation (1), $\forall i, x_i \leq z$ holds. Therefore, an enumeration of all tuples $x_1, \ldots, x_n$ for $0 \leq x_i \leq z$ provides an obvious but dramatically inefficient solution.

So our open problem reduces to *finding an efficient, linear or low polynomial algorithm for computing the inverse*. This paper provides a surprisingly simple solution to it in section 7, after telling the story of our incremental refinements (as well as backtracking steps) leading to it. Section 2 overviews the well-known solution for $n = 2$. Section 3 provides the Prolog implementation of the mapping from n-tuples to natural numbers. Section 4 describes the successive refinements of the inverse function, from its specification to a moderately useful implementation. Section 5 introduces a *list-to-set bijection* that will turn out to be helpful in "connecting the dots" to a well-known combinatorial problem that leads to our solution described in section 7 (after a small "backtracking step" shown in section 6). Section 8 extends the bijection to sets and multisets. Section 10 discusses related work and section 11 concludes the paper.

2 The Classic Result: Cantor's Pairing Function and its Inverse

Cantor's pairing function is a polynomial of degree 2, obtained from the generalized one for $n = 2$, given by the formula $f(x_1, x_2) = x_1 + \frac{(x_1+x_2+1)(x_1+x_2)}{2}$.
The following Prolog code implements it:

```
cantor_pair(X1,X2,P) :- P is X1 + (((X1+X2+1) * (X1+X2)) // 2).
```

Note that by composing it n times, one can obtain an n-tupling function, but unfortunately the resulting polynomial is of degree 2^n, in contrast to the generalized n-tupling bijection which is a polynomial of degree n. On the other, hand, as the following Prolog code shows, the problem of finding its inverse efficiently is relatively easy. Basically, the inverse of Cantor's pairing function is obtained by solving a second degree equation while keeping in mind that solutions should be natural numbers [17].

```
cantor_unpair(P,K1,K2) :- E is 8*P+1, intSqrt(E,R), I is (R-1)//2,
  K1 is P-((I*(I+1))//2), K2 is ((I*(3+I))//2)-P.
```

We face a small bump here – Prolog's ordinary square root returning a fixed size float or double does not make sense when working with arbitrary size integers, so we need to

implement an "integer square root" of N returning the natural number that provides the largest perfect square $\leq N$. Fortunately, we can ensure fast convergence using Newton's method:

```
intSqrt(0,0).
intSqrt(N,R) :- N>0, iterate(N,N,K), K2 is K*K, (K2>N -> R is K-1 ; R=K).

iterate(N,X,NewR) :- R is (X+(N//X))//2, A is abs(R-X),
  (A<2 -> NewR=R ; iterate(N,R,NewR)).
```

As the following example shows, computations with larger than 64-bit operands are handled, provided that the underlying Prolog system supports arbitrary size integers.

```
?- cantor_pair(1234567890,9876543210,P),cantor_unpair(P,A,B).
P = 61728394953703703760, A = 1234567890, B = 9876543210.
```

3 Implementing the Generalized Cantor n-tupling Bijection

Tupling/untupling functions are a natural generalization of pairing/unpairing operations. They are called *ranking/unranking* functions by combinatorialists as they map bijectively various combinatorial objects to $\mathbb{N}$ (ranking) and back (unranking).

The natural generalization of Cantor's pairing bijection described in [2] is introduced using geometric considerations that make it obvious that it defines a bijection $K_n : \mathbb{N}^n \to \mathbb{N}$. More precisely, they observe that the enumeration in $\mathbb{N}^2$ of integer coordinate pairs laying on the anti-diagonals $x_1 + x_2 = c$ can be lifted to points with integer coordinates laying on hyperplanes of the form $x_1 + x_2 + \ldots + x_k = c$. The same result, using a slightly different formula is proven algebraically, by induction in [7]. We remind that the bijection K_n is defined by the formula

$$K_n(x_1, \ldots, x_n) = \sum_{k=1}^{n} \binom{k - 1 + x_1 + \ldots + x_k}{k} \tag{2}$$

where $\binom{n}{k}$, also called "binomial coefficient" denotes the number of subsets of n with k elements as well as the coefficient of x^k in the expansion of the binomial $(x + y)^n$.

It is easy to see that the generalized Cantor n-tupling function defined by equation (2) is a polynomial of degree n in its arguments, and a conjecture, attributed in [2] to Rudolf Fueter (1923), states that it is the only one, up to a permutation of the arguments. As mentioned in section 1, as we have found out through extensive literature search, while hoping for the contrary, it was also an *open problem* to find an efficient inverse for it.

Our first step is an efficient implementation of the function $K_n : \mathbb{N}^k \to \mathbb{N}$. By all means, this is the easy part, just summing up a set of binomial coefficients.

3.1 Binomial Coefficients, efficiently

Computing binomial coefficients efficiently is well-known

$$\binom{k}{n} = \frac{n!}{k!(n-k)!} = \frac{n(n-1)\ldots(n-(k-1))}{k!} \tag{3}$$

However, we will need to make sure that we avoid unnecessary computations and reduce memory requirements by using a tail-recursive loop. After simplifying the slow formula in the first part of the equation (3) with the faster one based on falling factorial $n(n-1)\ldots(n-(k-1))$, and performing divisions as early as possible to avoid generating excessively large intermediate results, one can derive the `binomial_loop` tail-recursive predicate:

```
binomial_loop(_,K,I,P,R) :- I>=K, !, R=P.
binomial_loop(N,K,I,P,R) :- I1 is I+1, P1 is ((N-I)*P) // I1,
   binomial_loop(N,K,I1,P1,R).
```

Note that, as a simple optimization, when $N - K \leq K$, the faster computation of $\binom{N}{N-K}$ is used to reduce the number of steps in `binomial_loop`.

The resulting predicate `binomial(N,K,R)` computes $\binom{N}{K}$ and unifies the result with `R`.

```
binomial(N,K,R) :- N<K, !, R=0.
binomial(N,K,R) :- K1 is N-K, K>K1, !, binomial_loop(N,K1,0,1,R).
binomial(N,K,R) :- binomial_loop(N,K,0,1,R).
```

3.2 The $\mathbb{N}^k \rightarrow \mathbb{N}$ bijection

We are ready to implement a first version of the $\mathbb{N}^k \rightarrow \mathbb{N}$ ranking function as a tail-recursive computation using the accumulator pairs `L1` $\rightarrow$ `L2`, that hold the states of the length of the list processed so far, and `S1` $\rightarrow$ `S2`, that hold the state of the prefix sum of $X_1, X_2, \ldots X_k$ computed so far.

```
from_cantor_tuple1(Xs,R) :- from_cantor_tuple1(Xs,0,0,0,R).

from_cantor_tuple1([],_L,_S,B,B).
from_cantor_tuple1([X|Xs],L1,S1,B1,Bn) :- L2 is L1+1, S2 is S1+X, N is S2+L1,
  binomial(N,L2,B), B2 is B1+B,
  from_cantor_tuple1(Xs,L2,S2,B2,Bn).
```

The following examples illustrate the fact that the values of the result are relatively small, independently of the length or the size of the values on the input list.

```
?- from_cantor_tuple([],N).
N = 0.
?- from_cantor_tuple([0,2012,999,0,10],N).
N = 2107259417045595.
?- from_cantor_tuple([9,8,7,6,5,4,3,2,1,0,0,1,2,3,4,5,6,7,8,9],N).
N = 3706225144988231392404.
```

4 Refining the Specification of the Inverse

We start with an executable specification of the inverse, seen as defining, for a given K, a bijection $g_K : \mathbb{N} \rightarrow \mathbb{N}^K$.

4.1 Enumerating, naively

The predicate `to_cantor_tuple1(K,N,Ns)` computes, for each `K`, the function g_K associating to the natural number `N` a tuple represented as a list `Ns` of length `K`.

```
to_cantor_tuple1(K,N,Ns) :- numlist(0,N,Is), cartesian_power(K,Is,Ns),
  from_cantor_tuple1(Ns,N),
  !. % just an optimization - no other solutions exist
```

Note that the built-in `numlist(From, To, Is)` is used to generate a list of integers in the interval `[From..To]`).

The predicate `to_cantor_tuple1` uses `cartesian_power(K,Is,Ns)` to enumerate candidates of length `K`, drawn from the initial segment `[0..N]` of $\mathbb{N}$.

```
cartesian_power(0,_,[]).
cartesian_power(K,Is,[X|Xs]) :- K>0, K1 is K-1, member(X,Is),
  cartesian_power(K1,Is,Xs).
```

As `cartesian_power` backtracks over this finite set of potential solutions, the predicate `from_cantor_tuple1(Ns,N)` is called until the first (and known to be unique) solution is found. Given the unicity of the solution, the CUT in the predicate `to_cantor_tuple1` is simply an optimization without an effect on the meaning of the program.

The following example illustrates the correctness of this executable specification.

```
?- to_cantor_tuple1(3,42,R), from_cantor_tuple(R,S).
R = [1, 2, 2], S = 42.
```

Unfortunately, performance deteriorates quickly around `K` larger than `5` and `N` larger than `100` as the time complexity of this program is at least $O(N^K)$. However, given our reliance on Prolog's backtracking, the search uses at most $O(K\,log(N))$ space when filtering through lists of length `K` containing numbers of at most the bitsize of `N`.

4.2 A better algorithm, using a tighter upper limit

The next step in deriving an efficient untupling function is a bit trickier. First we observe that, as `from_cantor_tuple(K,Ns,N)` runs through successive hyperplanes $X_1 + \ldots + X_k = M$, for each of them the sum maxes out when $X_1 = M$ and $X_k = 0$ for $1 \leq K \leq N$. We can compute directly this maximum value with the predicate `largest_binomial_sum` as follows:

```
largest_binomial_sum(K,M,R) :- largest_binomial_sum(K,M,0,R).

largest_binomial_sum(0,_,R,R).
largest_binomial_sum(K,M,R1,Rn) :- K>0, K1 is K-1, M1 is M+K1,
  binomial(M1,K,B), R2 is R1+B,
  largest_binomial_sum(K1,M,R2,Rn).
```

The predicate `largest_binomial_sum(K,M,R)` computes the same `R` as `cantor_tuple([M, 0,...,0], R)`, with `K-1 0`s following `M`.

Next we compute the upper limit for possible values of the sum `M` of $[X_1, ..., X_k]$ such that the relation `to_cantor_tuple([X1,...,Xk],N)` holds, i.e. we find the hyperplane $X_1 + \ldots + X_k = M$ defining the Cantor `K`-tuple. This computation, is implemented by the predicate `find_hyper_plane(K,N,M)` which, when given the inputs `K` and `M`, finds the value of the sum `M` that defines the hyperplane containing our tuple.

```
find_hyper_plane(0,_,0).
find_hyper_plane(K,N,M) :- K>0, between(0,N,M), largest_binomial_sum(K,M,R), R>=N,!.
```

Note the use of the built-in `between(From,To,I)` that backtracks over integers in the interval `[From..To]`.

We are now ready to define a more efficient inverse of the `from_cantor_tuple1` bijection, called `to_cantor_tuple2`, as a search through the set of lists such that the relation `from_cantor_tuple1(Xs,N)` holds.

```
to_cantor_tuple2(K,N,Ns) :- find_hyper_plane(K,N,M),
  sum_bounded_cartesian_power(K,M,Xs),
  from_cantor_tuple1(Xs,N),
  !,
  Ns=Xs.
```

The search, restricted this time to integers in the interval `[0..M]` is implemented by the predicate `sum_bounded_cartesian_power`.

```
sum_bounded_cartesian_power(0,0,[]).
sum_bounded_cartesian_power(K,M,[X|Xs]) :-  K>0, M>=0, K1 is K-1,
  between(0,M,X), M1 is M-X,
  sum_bounded_cartesian_power(K1,M1,Xs).
```

Note that, after applying the upper limit `M` computed by `find_hyper_plane`, to ensure that only tuples summing up to `M` are explored, we are using a customized cartesian product computation, in the predicate `sum_bounded_cartesian_power` backtracking over lists `[X1...Xk]` that sum-up to `M`. However, as the query

```
?- findall(M,(between(0,31,N),P is 2^N,find_hyper_plane(2,P,M)),Ms).
Ms = [1,1,2,3,5,7,10,15,22,31,44,63,90,127,180,255,361,511,723,1023,
     1447,2047,2895,4095,5792,8191,11584,16383,23169,32767,46340,65535]
```

indicates, while `M` grows significantly slower than `P` it can reach intractable ranges quite quickly.

The predicate `to_cantor_tuple2` is a good improvement over `to_cantor_tuple1`, but it is by no means the efficient algorithm we are seeking.

Clearly, a “paradigm shift” is needed at this point, as obvious optimizations only promise diminishing returns. The highest hope would be to find a deterministic predicate similar to the integer square root based inverse for the case $N = 2$, but this time the arbitrary degree N of our polynomial looks like an insurmountable obstacle.

5 The Missing Link: from Lists to Sets and Back

After rewriting the formula for the $\mathbb{N}^k \to \mathbb{N}$ bijection as:

$$K_n(x_1, \ldots, x_n) = \sum_{k=1}^{n} \binom{k-1+s_k}{k} \tag{4}$$

where $s_k = \sum_{i=1}^{k} x_i$, we recognize the *prefix sums* s_k incremented with values of k starting at 0.

At this point, as our key “Eureka step”, we instantly recognize here the “set side” of the bijection between sequences of n natural numbers and sets of n natural numbers described in [13][1]. We can compute the bijection `list2set` together with its inverse `set2list` as

```
list2set(Ns,Xs) :- list2set(Ns,-1,Xs).

list2set([],_,[]).
list2set([N|Ns],Y,[X|Xs]) :- X is (N+Y)+1, list2set(Ns,X,Xs).

set2list(Xs,Ns) :- set2list(Xs,-1,Ns).

set2list([],_,[]).
set2list([X|Xs],Y,[N|Ns]) :- N is (X-Y)-1, set2list(Xs,X,Ns).
```

[1] In [13] a general framework for bijective data transformations provides such conversion algorithms between a large number of fundamental data types.

The following examples illustrate how it works:

```
?- list2set([2,0,1,2],Set).
Set = [2, 3, 5, 8].

?- set2list([2, 3, 5, 8],List).
List = [2, 0, 1, 2].
```

As a side note, this bijection is mentioned in [5] and implicitly in [2], with indications that it might even go back to the early days of the theory of recursive functions.

6 Backtracking one step: revisiting the $\mathbb{N}^k \to \mathbb{N}$ bijection

It is time to step back at this point, and factor out `list2set` from our tail-recursive "untupling" loop `from_cantor_tuple1`.

The predicate `from_cantor_tuple` implements the the $\mathbb{N}^k \to \mathbb{N}$ bijection in Prolog, using the iterative computation of the binomial $\binom{n}{k}$ as well as the sequence to set transformer `list2set`. In contrast to `from_cantor_tuple1`, `untupling_loop` does not need to add the increments $1, 2, ..L-1$ as this task has been factored out and processed by `list2set`.

```
from_cantor_tuple(Ns,N) :-
  list2set(Ns,Xs),
  untupling_loop(Xs,0,0,N).

untupling_loop([],_L,B,B).
untupling_loop([X|Xs],L1,B1,Bn) :- L2 is L1+1, binomial(X,L2,B), B2 is B1+B,
  untupling_loop(Xs,L2,B2,Bn).
```

This shifts the problem of computing its inverse from lists to sets, an apparently minor use of a *bijective data type transformation*, that will turn out to be the single most critical step toward our solution.

7 The Efficient Inverse

We have now split our problem in two simpler ones: inverting `untupling_loop` and then applying `set2list` to get back from sets to lists.

Our first attempt was to try out constraint solving as it can sometime reverse arithmetic operations. Moreover, global constraints like `all_different` can take advantage of the fact that we are dealing with sets. However, the code (included as a comment in the companion Prolog file), turned out to be orders of magnitude slower than `to_cantor_tuple2`. This happened despite of the fact that we have tried also to take advantage of the optimizations implemented by the predicate `to_cantor_tuple2`, most likely because delaying computations brought unnecessary overhead without changing the essentially nondeterministic nature of the search.

The key "Eureka step" at this point is to observe that `untupling_loop` implements the sum of the combinations $\binom{X_1}{1} + \binom{X_2}{2} + \ldots + \binom{X_K}{K} = N$, which is nothing but the representation of N in the *combinatorial number system of degree* K, [16], due to [6]. Fortunately, efficient conversion algorithms between the conventional and the combinatorial number system are well known, [1, 5].

For instance, theorem **L** in [5] describes the precise position of a given sequence in the lexicographic order enumeration of all sequences of length k.

▶ **Theorem 1 (Knuth).** The combination $[c_k, \dots c_2, c_1]$ is visited after exactly $\binom{c_k}{k} + \dots + \binom{c_2}{2} + \binom{c_1}{1}$ other combinations have been visited.

We are ready to implement the Prolog predicate `tupling_loop(K,N,Ds)`, which, given the degree `K` indicating the number of "combination digits", finds and repeatedly subtracts the greatest binomial smaller than `N`.

```
tupling_loop(0,_,[]).
tupling_loop(K,N,[D|Ns]) :- K>0, NewK is K-1, I is K+N,
  between(NewK,I,M), binomial(M,K,B), B>N,
  !, % no more search is needed
  D is M-1, % the previous binomial gives the "digit" D
  binomial(D,K,BM), NewN is N-BM,
  tupling_loop(NewK,NewN,Ns).
```

The predicate `tupling_loop` implements a deterministic greedy search algorithm, by subtracting the combination containing the most significant "digit" `D` at each step from the variable `N`. At a given step, this results in the variable `NewN` that carries on the result in the tail-recursive loop. At the same time, the decreased value of `K`, used in the binomial is carried on as the variable `NewK`.

The efficient inverse of Cantor's N-tupling is now simply:

```
to_cantor_tuple(K,N,Ns) :- tupling_loop(K,N,Xs), reverse(Xs,Rs), set2list(Rs,Ns).
```

Note that we reverse the intermediate result `Xs` to ensure that `set2list` receives it in increasing order - our canonical representation for sets. The following example illustrates that it works as expected, including on very large numbers:

```
?- to_cantor_tuple(1234,666677778888999900003141 5,Ns), from_cantor_tuple(Ns,N).
Ns = [0, 0, 0, 0, 0, 0, 0, 0, 0...,0, 0, 1, 0], N = 6666777788889999000031415 .
```

8 Extending the Bijection to Sets and Multisets of `K` Natural Numbers

We obtain a bijection from natural numbers to *sets of* `K` *natural numbers*, canonically represented as lists of strictly increasing elements, by simply dropping the `set2list` and `list2set` operations.

```
from_cantor_set_tuple(Xs,N) :- untupling_loop(Xs,0,0,N).

to_cantor_set_tuple(K,N,Xs) :- tupling_loop(K,N,Ts), reverse(Ts,Xs).
```

Multisets of `K` *natural numbers* are represented canonically as sequences of nondecreasing, but possibly duplicated elements. Following [13], a transformation, similar to `list2set/set2list` can be derived for multisets. After a few unfoldings, the resulting code, using tail recursive helper predicates, becomes:

```
mset2set(Ns,Xs) :- mset2set(Ns,0,Xs).

mset2set([],_,[]).
mset2set([X|Xs],I,[M|Ms]) :- I1 is I+1, M is X+I, mset2set(Xs,I1,Ms).

set2mset(Xs,Ns) :- set2mset(Xs,0,Ns).

set2mset([],_,[]).
set2mset([X|Xs],I,[M|Ms]) :- I1 is I+1, M is X-I, set2mset(Xs,I1,Ms).
```

The two transformations can be seen as defining a bijection between strictly increasing and nondecreasing sequences of natural numbers:

```
?- set2mset([2,5,6,8,9],Mset), mset2set(Mset,Set).
Mset = [2, 4, 4, 5, 5], Set = [2, 5, 6, 8, 9].
```

We can combine this bijection with the Cantor n-tupling bijection and obtain

```
from_cantor_multiset_tuple(Ms,N) :- mset2set(Ms,Xs), from_cantor_set_tuple(Xs,N).
```

```
to_cantor_multiset_tuple(K,N,Ms) :- to_cantor_set_tuple(K,N,Xs), set2mset(Xs,Ms).
```

For instance, when dealing with *commutative and associative operations*, such multiset encodings turn out to be a natural match.

9 A Simple Application: Fair Search

One might ask, legitimately, why would one bother with pairing and n-tupling bijections. While the case has been made (see for instance [11]) for various applications besides theoretical computer science, that range from indexing multi-dimensional data and geographic information systems to cryptography and coding theory, we will focus here on a simple application with immediate relevance to logic programming: fair search through a multi-parameter search space.

A theorem conjectured by Bachet and proven by Lagrange, states that "*every natural number is the sum of at most four squares*". Let's assume that one wants to find, a "simple" solution to the equation (5), knowing that, as a consequence of this theorem, a solution always exists.

$$N = X^2 + Y^2 + Z^2 + U^2 \tag{5}$$

Let us define "simple solution" as a solution bounded by $O(X + Y + Z + U)$. We want to enumerate "simpler" candidates first, efficiently. To this end, we can use the fast inverse of the Cantor n-tupling function (specialized to multisets, given that both the "`*`" and "`+`" operations, involved in the equation 5, are associative and commutative). We can write a generic `fair_multiset_tuple_generator` as:

```
fair_multiset_tuple_generator(From,To,Length, Tuple) :- between(From,To,N),
  to_cantor_multiset_tuple(Length,N,Tuple).
```

We can specialize `fair_multiset_tuple_generator` for our specific problem as:

```
to_lagrange_squares(N,Ms) :- M is N^2, % conservative upper limit
  fair_multiset_tuple_generator(0,M,4,Ms),
  maplist(square,Ms,MMs), sumlist(MMs,N),
  !. % keep the first solution only

square(X,S) :- S is X*X.
```

The algorithm is quite efficient, for instance, it takes only a few seconds to find a decomposition for 2012:

```
?- time(to_lagrange_squares(2012,Xs)), maplist(square,Xs,Ns), sumlist(Ns,N).
% 9,685,955 inferences, 4.085 CPU in 4.085 seconds (100% CPU, 2371347 Lips)
Xs = [15, 23, 23, 27], Ns = [225, 529, 529, 729], N = 2012.
```

The algorithm is also simple enough to be used as an executable specification and it ensures optimality of the solution, in the sense that our search scans hyperplanes of the form $X_1 + X_2 + X_3 + X_4 = K$ for progressively larger and larger values of K. Also, given the multiset representation, the associativity and commutativity of "`*`" and "`+`" are factored in, reducing the search space significantly. However, our simple algorithm is no match to the $O(log^2(N))$ randomized algorithm of [9]. As a side note, deriving a faster algorithm for this decomposition is a fascinating task on its own, starting with the observation that it needs only to be computed for the prime factors of a number and involving some elegant identities holding for Hurwitz quaternions [18]. More importantly, the mechanism sketched here can also be used in iterative deepening algorithms as a fair a goal selector (for both conjunctions and disjunctions). This can be done initially in a meta-interpreter and possibly partially evaluated or moved to the underlying Prolog abstract machine.

Note also that, depending on the natural representation of the candidate data tuple (i.e. set, multiset or sequence), one can customize the fair tuple generator accordingly.

10 Related Work

We have found the first reference to the generalization of Cantor's pairing function to n-tuples in [2], and benefited from the extensive study of its properties in [7].

There are a large number of papers referring to the original "Cantor pairing function" among which we mention the surprising result that, together with the successor function it defines a decidable subset of arithmetic [3]. Combinatorial number systems can be traced back to [6] and one can find efficient conversion algorithms to conventional number systems in [5] and [1]. Finally, the "once you have seen it, obvious" `list2set / set2list` bijection is borrowed from [13], but not unlikely to be common knowledge of people working in combinatorics or recursion theory. This simple bijection between lists and sets of natural numbers shows the unexpected usefulness of the framework supporting bijective data type transformations [13, 15, 12], of which, a large Haskell-based[2] instance is described in [14].

11 Conclusion

We have derived through iterative refinements a fairly surprising solution to an open problem for which we had no a priori idea if it is solvable, or within which complexity bounds could be solved. The key "Eureka step" was to recognize a bijective data type transformation that suddenly brought us to a relatively well known equivalent problem for which efficient algorithms were available. Through the process, the ability to automate search algorithms relying directly on an executable declarative specification has been a major catalyst. The ability to derive equivalent logic programs using simple transformations has been also unusually helpful. From a software engineering perspective, this recommends logic programming as an ideal problem solving tool. Last but not least, proven sources of fundamental algorithms like [5] and the unusually high quality of Wikipedia articles on related topics have helped "connecting the dots" quickly and effectively.

Acknowledgement

This research has been supported by NSF research grant 1018172.

[2] but designed in a guarded Horn-clause style, for virtually automatic transliteration to Prolog

References

1. B. P. Buckles and M. Lybanon. Generation of a Vector form the Lexicagraphical Index [G6]. *ACM Transactions on Mathematical Software*, 5(2):180–182, June 1977.
2. Patrick Cegielski and Denis Richard. On arithmetical first-order theories allowing encoding and decoding of lists. *Theoretical Computer Science*, 222(1–2):55 – 75, 1999.
3. Patrick Cégielski and Denis Richard. Decidability of the Theory of the Natural Integers with the Cantor Pairing Function and the Successor. *Theor. Comput. Sci.*, 257(1-2):51–77, 2001.
4. Martin Davis, Hilary Putnam, and Julia Robinson. The decision problem for exponential diophantine equations. *The Annals of Mathematics*, 74(4):425–436, nov 1961.
5. Donald E. Knuth. *The Art of Computer Programming, Volume 4, Fascicle 3: Generating All Combinations and Partitions.* Addison-Wesley Professional, 2005.
6. D. H. Lehmer. The machine tools of combinatorics. In *Applied combinatorial mathematics*, pages 5–30. Wiley, New York, 1964.
7. Meri Lisi. Some remarks on the Cantor pairing function. *Le Matematiche*, 62(1), 2007.
8. Yuri Matiyasevich. *Hilbert's Tenth Problem.* MIT Press, Cambridge, London, 1993.
9. Michael O. Rabin and Jeffery O. Shallit. Randomized algorithms in number theory. *Communications on Pure and Applied Mathematics*, 39(S1):S239–S256, 1986.
10. Julia Robinson. Unsolvable diophantine problems. *Proceedings of the American Mathematical Society*, 22(2):534–538, aug 1969.
11. Arnold L. Rosenberg. Efficient pairing functions – and why you should care. *International Journal of Foundations of Computer Science*, 14(1):3–17, 2003.
12. Paul Tarau. A Groupoid of Isomorphic Data Transformations. In J. Carette, L. Dixon, C. S. Coen, and S. M. Watt, editors, *Intelligent Computer Mathematics, 16th Symposium, Calculemus 2009, 8th International Conference MKM 2009* , pages 170–185, Grand Bend, Canada, July 2009. Springer, LNAI 5625.
13. Paul Tarau. An Embedded Declarative Data Transformation Language. In *Proceedings of 11th International ACM SIGPLAN Symposium PPDP 2009*, pages 171–182, Coimbra, Portugal, September 2009. ACM.
14. Paul Tarau. Declarative Combinatorics: Isomorphisms, Hylomorphisms and Hereditarily Finite Data Types in Haskell, January 2009. Unpublished draft, `http://arXiv.org/abs/0808.2953`, updated version at `http://logic.cse.unt.edu/tarau/research/2010/ISO.pdf`, 150 pages.
15. Paul Tarau. Isomorphisms, Hylomorphisms and Hereditarily Finite Data Types in Haskell. In *Proceedings of ACM SAC'09*, pages 1898–1903, Honolulu, Hawaii, March 2009. ACM.
16. Wikipedia. Combinatorial number system — wikipedia, the free encyclopedia, 2011. [Online; accessed 21-March-2012].
17. Wikipedia. Pairing function — wikipedia, the free encyclopedia, 2011. [Online; accessed 23-March-2012].
18. Wikipedia. Lagrange's four-square theorem — wikipedia, the free encyclopedia, 2012. [Online; accessed 22-March-2012].

On the Termination of Logic Programs with Function Symbols

Sergio Greco, Francesca Spezzano, and Irina Trubitsyna

DEIS – Università della Calabria, 87036 Rende, Italy
{greco,fspezzano,irina}@deis.unical.it

Abstract

Recently there has been an increasing interest in the bottom-up evaluation of the semantics of logic programs with complex terms. The main problem due to the presence of functional symbols in the head of rules is that the corresponding ground program could be infinite and that finiteness of models and termination of the evaluation procedure is not guaranteed. This paper introduces, by deeply analyzing program structure, new decidable criteria, called *safety* and Γ-*acyclicity*, for checking termination of logic programs with function symbols under bottom-up evaluation. These criteria guarantee that stable models are finite and computable, as it is possible to generate a finitely ground program equivalent to the source program. We compare new criteria with other decidable criteria known in the literature and show that the Γ-*acyclicity* criterion is the most general one. We also discuss its application in answering bound queries.

1998 ACM Subject Classification D.1.6 Logic Programming

Keywords and phrases Logic Programming, Function Symbols, Bottom-up Execution, Program Termination, Stable Models

Digital Object Identifier 10.4230/LIPIcs.ICLP.2012.323

1 Introduction

Recently there has been an increasing interest in Answer Set Programming (ASP) with function symbols, and more in general on the bottom-up evaluation of the semantics of logic programs with complex terms [1, 2, 3]. Indeed, one of the main limitations of current ASP and datalog systems is the inability (or the limited power) to define programs with complex terms and function symbols [4, 5, 6, 7]. The main problem in extending logic programming under bottom-up evaluation with function symbols is that the corresponding ground program is infinite and that finiteness of models and termination of the evaluation procedure is not guaranteed.

The problem of checking whether the computation of a query terminates has been investigated since the beginning of logic programming. Most of the past work was devoted to the termination of programs under top-down evaluation or for SLD resolution [8, 9], although it also received a significant attention from the deductive database community [10]. The reason was that the only relevant logic programming implemented language was Prolog, which computes answers to queries using a specific SLDNF resolution algorithm. Recently, the attention has been concentrated on semantics which can be naturally computed by means of bottom-up evaluation, such as stable model semantics for programs with possibly unstratified negation, perfect model semantics for programs with stratified negation, and minimum model semantics for positive programs. The following example shows a very simple program which uses function symbols in rule heads and the problem is to decide if the fixpoint computation of the program terminates.

© Sergio Greco, Francesca Spezzano, and Irina Trubitsyna;
licensed under Creative Commons License ND
Technical Communications of the 28th International Conference on Logic Programming (ICLP'12).
Editors: A. Dovier and V. Santos Costa; pp. 323–333

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum f"ur Informatik, Dagstuhl Publishing, Germany

▶ **Example 1.** Consider the following logic program $\mathcal{P}_1$:

$r_1 : \mathtt{p(a,a)}.$
$r_2 : \mathtt{p(f(X), g(X))} \leftarrow \mathtt{p(X, X)}.$

The program has a unique minimal model $M_1 = \{\mathtt{p(a,a)}, \mathtt{p(f(a), g(a))}\}$, which can be computed using the classical bottom-up fixpoint algorithm. Current techniques are not able to establish (in advance, by analyzing the structure of the program) that the fixpoint computation terminates. ☐

The problem, known as *program termination*, (or *query termination*, when we refer to a specific query goal) is, in the general case, undecidable. Therefore, the recent research is investigating the identification of structural criteria that guarantee that the semantics can be computed using, for instance, bottom-up evaluators based on the grounding of programs. This is not a simple task as it is possible to have equivalent queries (i.e. queries computing the same answers, independently from the database) that have different structural properties and very basic changes to the syntax of programs, even without changing the semantics, may significantly alter the structural properties.

Current criteria analyze how values are propagated among predicate arguments, to understand whether the set of values associable with an argument is finite. However, these methods have limited capacity in comprehending finiteness of arguments appearing in recursive rules with function symbols in the head of rules. Considering the previous example, they are not able to understand that rule r_2 can be activated a finite number of times (actually, considering that there is only one exit rule, the recursive rule can be activated at most once).

Related works. As said before, the problem of checking whether the computation of a query terminates has been investigated since the beginning of logic programming.

Most of the past work was devoted to the termination of programs under top-down evaluation or for SLD resolution [8, 9]. The class of *finitary* programs, allowing decidable (ground) query computation using a top-down evaluation, has been proposed in [11]. A program $\mathcal{P}$ is finitary if (1) the number of cycles involving an odd number of negative subgoals is finite, and (2) it is *finitely recursive.* A program $\mathcal{P}$ is finitely recursive if each ground atom depends on finitely many ground atoms [12].

The problem of establishing whether the bottom-up based computation of logic programs terminates received a significant attention since the beginning of deductive databases [10] and recently has received an increasing interest. The class of *finitely ground ($\mathcal{FG}$) programs* has been proposed in [13]. The key property of this class is that stable models (answer sets) are computable. In fact, for each program $\mathcal{P}$ in this class there exists a finite and computable subset of its instantiation (grounding), called *intelligent instantiation*, having precisely the same answer sets as $\mathcal{P}$. As the problem of deciding whether a program is $\mathcal{FG}$ is not decidable, decidable subclasses, such as *finite domain ($\mathcal{FD}$) programs* [13], *ω-restricted programs* [14], *λ-restricted programs* [15] and *argument restricted ($\mathcal{AR}$) programs* [16] have been proposed. The query termination problem for ground query goals has been studied in [17]. Other approaches are the class of $\mathbb{FDNC}$ programs [2], i.e. programs having infinite answer sets in general, but a finite representation that can be exploited for knowledge compilation and fast query answering, and the proposal of [3], where functions are replaced by relations defined over finite domains.

Contribution. We first introduce the concept of *safe arguments* (a restriction of finite domain arguments), by also analyzing how rules may fire each other. As safe arguments can range only on a finite set of values, the instantiation of safe programs (that is, programs whose arguments are all safe) results in a finite ground program. Consequently, safe programs have a finite number of finite stable models and we show that the class of safe programs is decidable. We also show that the class of safe programs extends the class of finite domain programs, but is not comparable with the class of argument restricted programs.

Next we introduce a further criterion, called Γ-acyclicity, which analyzes the role of function symbols used in the program. We introduce the concept of *labelled propagation graph*, representing how complex terms in non-safe (or *affected*) arguments are created and used during bottom-up evaluation. The class of Γ-acyclic programs is defined by only considering affected arguments and cycles spelling strings of an underlying context free language. We show that this class is decidable, strictly extends both classes of safe programs and argument restricted programs and that it has a finite set of finite stable models which can be computed using current ASP systems, by a simple rewriting of the source program.

Finally, we discuss how the new criterion can be used in bound query answering.

Organization. The paper is organized as follows. Section 2 introduces basic notions on logic programming and recalls two main criteria guaranteeing the termination of logic programs with function symbols under bottom-up evaluation. Section 3 presents the class of safe programs. Section 4 introduces the class of Γ-acyclic programs. Section 5 shows how Γ-acyclic programs are rewritten so that their semantics can be computed by current ASP systems. Section 6 discusses bound query answering.

2 Logic programs with function symbols

Syntax. We assume to have infinite sets of constants, variables, predicate symbols and function symbols. Predicate and function symbols have associated a fixed arity. For a predicate p of arity n, we denote by $p[i]$, for $1 \leq i \leq n$, its i-th argument.

A *term* is either a constant, a variable or a complex term of the form $f(t_1, ..., t_m)$, where $t_1, ..., t_m$ are terms and f is a function symbol of arity m; each term t_i, for $1 \leq i \leq m$, is a *subterm* of $f(t_1, ..., t_m)$. The subterm relation is reflexive (each term is subterm of itself) and transitive (if t_i is subterm of t_j and t_j is subterm of t_k, then t_i is subterm of t_k). An *atom* is of the form $p(t_1, ..., t_n)$, where $t_1, ..., t_n$ are terms and p is a predicate symbols of arity n. A *literal* is either a (positive) atom A or its negation $\neg A$. A (*disjunctive*) rule r is a clause of the form:

$$a_1 \vee \cdots \vee a_m \leftarrow b_1, \cdots, b_k, \neg c_1, \cdots, \neg c_n$$

where $m > 0$ $k, n \geq 0$ and $a_1, \cdots, a_m, b_1, \cdots, b_k$, $c_1, \cdots, c_n$ are atoms. The disjunction $a_1 \vee \cdots \vee a_m$ is called the *head* of r and is denoted by *head*(r) while the conjunction $b_1, \cdots, b_k, \neg c_1, \cdots, \neg c_n$ is called the *body* and is denoted by *body*(r). If $m = 1$, then r is *normal* (i.e. ∨-free); if $n = 0$, then r is *positive* (i.e. ¬-free); if both $m = 1$ and $n = 0$, then r is *normal and positive.* A *program* $\mathcal{P}$ is a finite set of rules. A term (resp. an atom, a rule or a program) is said to be *ground* if no variables occur in it. A ground normal rule with an empty body is also called *fact.*

With a little abuse of notation we often use the same notation to denote a conjunction of body literals and a set of body literals, that is *body*(r) is also used to denote the set of literals appearing in the body of r. We also denote the *positive body* of r by *body*$^+(r) = \{b_1, \ldots, b_k\}$

and the *negative body* of r by $body^-(r) = \{c_1, \dots, c_n\}$. A predicate p depends on a predicate q if there is a rule r such that p appears in the head and q in the body, or there is a predicate s such that p depends on s and s depends on q. A predicate p is said to be recursive if it depends on itself, whereas two predicates p and q are said to be mutually recursive if p depends on q and q depends on p.

Generally, predicate symbols are partitioned into two different classes: extensional (or EDB or base), i.e. defined by the ground facts of a database, and intensional (or IDB or derived), i.e. defined by the rules of the program. The definition of a predicate p consists of all the rules (or facts) having p in the head. A database D consists of all the facts defining EDB predicates, whereas a program $\mathcal{P}$ consists of the rules defining IDB predicates. The program consisting of rules defining IDB predicates and facts defining EDB predicates is denoted by $\mathcal{P}_D$. When there is no ambiguity we shall use the symbol $\mathcal{P}$ to denote the complete set of rules and database facts. Given a set of ground atoms S and an atom $g(t)$, $S[g]$ (resp. $S[g(t)]$) denotes the set of g-tuples (resp. tuples matching $g(t)$) in S. Analogously, for a given set of sets of atoms M we shall use the following notations $M[g] = \{S[g] \mid S \in M\}$ and $M[g(t)] = \{S[g(t)] \mid S \in M\}$. We also assume that programs are *range restricted* [18], i.e. variables appearing in the head or in negated body literals are range restricted, that is they also appear in some positive body literal, and that possible constants in $\mathcal{P}$ are taken from the database domain[1].

Semantics. The Herbrand universe $H_\mathcal{P}$ of a program $\mathcal{P}$ is the possibly infinite set of ground terms which can be built using constants and function symbols appearing in $\mathcal{P}$. The Herbrand base $B_\mathcal{P}$ of a program $\mathcal{P}$ is the set of ground atoms which can be built using predicate symbols appearing in $\mathcal{P}$ and ground terms of $H_\mathcal{P}$. A rule r' is a *ground instance* of a rule r, if r' is obtained from r by replacing every variable in r with some ground term in $H_\mathcal{P}$; $ground(\mathcal{P})$ denotes the set of all ground instances of the rules in $\mathcal{P}$. An interpretation of a program $\mathcal{P}$ is any subset of $B_\mathcal{P}$. The value of a ground atom L w.r.t. an interpretation I is $value_I(L) = L \in I$, whereas $value_I(\neg L) = L \notin I$. The truth value of a conjunction of ground literals $C = L_1, \dots, L_n$ is the minimum over the values of L_i, i.e. $value_I(C) = min(\{value_I(L_i) \mid 1 \leq i \leq n\})$, while the value of a disjunction $D = L_1 \vee ... \vee L_n$ is its maximum, i.e. $value_I(D) = max(\{value_I(L_i) \mid 1 \leq i \leq n\})$; if $n = 0$, then $value_I(C) = true$ and $value_I(D) = false$. A ground rule r is *satisfied* by I if $value_I(head(r)) \geq value_I(body(r))$. Thus, a rule r with an empty body is satisfied by I if $value_I(head(r)) = true$. An interpretation M for $\mathcal{P}$ is a model of $\mathcal{P}$ if M satisfies all the rules in $ground(\mathcal{P})$.

The *model-theoretic semantics* for a positive program $\mathcal{P}$ assigns the set of its *minimal models* $\mathcal{MM}(\mathcal{P})$. A model M for $\mathcal{P}$ is minimal, if no proper subset of M is a model for $\mathcal{P}$. The more general *disjunctive stable model semantics* generalizes stable model semantics previously defined for normal programs [19] and also applies to programs with (unstratified) negation [20].

Let $\mathcal{P}$ be a logic program and let I be an interpretation for $\mathcal{P}$, $\mathcal{P}^I$ denotes the ground positive program derived from $ground(\mathcal{P})$ by (1) removing all the rules that contain a negative literal $\neg a$ in the body and $a \in I$, and (2) removing all the negative literals from the remaining rules. An interpretation I is a (disjunctive) stable model for $\mathcal{P}$ if and only if $I \in \mathcal{MM}(\mathcal{P}^I)$. The set of stable models of $\mathcal{P}$ is denoted by $\mathcal{SM}(\mathcal{P})$. It is well known

[1] Range restricted programs are often called safe programs. We will use the term safe to denote a set of program arguments.

that stable models are minimal models (i.e. $\mathcal{SM}(\mathcal{P}) \subseteq \mathcal{MM}(\mathcal{P})$) and that for negation-free programs minimal and stable model semantics coincide (i.e. $\mathcal{SM}(\mathcal{P}) = \mathcal{MM}(\mathcal{P})$) and that positive normal programs have a unique minimal model.

Finite domain programs. The class of finite domain programs is defined by analyzing the structure of programs and is based on the concept of argument graph.

The *argument graph* $G^A(\mathcal{P})$ of a program $\mathcal{P}$ is a direct graph containing a node for each argument $p[i]$ of an IDB predicate p of $\mathcal{P}$; there is an edge $(q[j], p[i])$ iff there is a rule $r \in \mathcal{P}$ such that: i) an atom $p(\bar{t})$ appears in the head of r; ii) an atom $q(\overline{v})$ appears in $body^+(r)$; iii) $p(\bar{t})$ and $q(\overline{v})$ share the same variable within the i-th and the j-th term, respectively. Given a program $\mathcal{P}$, an argument $p[i]$ is said to be *recursive* if it appears in a cycle of $G^A(\mathcal{P})$.

▶ **Definition 2** ($\mathcal{FD}$ Program [13]). Given a program $\mathcal{P}$, the set of *finite-domain arguments ($\mathcal{FD}$ arguments)* of $\mathcal{P}$ is the maximal set $FD(\mathcal{P})$ of arguments of $\mathcal{P}$ such that, for each argument $q[k] \in FD(\mathcal{P})$, every rule r with head predicate q satisfies the following condition. Let t be the term corresponding to argument $q[k]$ in the head of r. Then, either i) t is variable-free, or ii) t is a subterm of (the term of) an $\mathcal{FD}$ argument of a positive body predicate, or iii) every variable appearing in t also appears in (the term of) an $\mathcal{FD}$ argument of a positive body predicate which is not recursive with $q[k]$. If all arguments of the predicates of $\mathcal{P}$ are $\mathcal{FD}$, then $\mathcal{P}$ is said to be an $\mathcal{FD}$ program. □

The main properties of $\mathcal{FD}$ programs are the following: (i) recognizing whether $\mathcal{P}$ is an $\mathcal{FD}$ program is decidable, and (ii) every $\mathcal{FD}$ program is an $\mathcal{FG}$ program. Checking whether a program $\mathcal{P}$ is $\mathcal{FD}$ or not can be done by assuming that all arguments are in $FD(\mathcal{P})$ and eliminating, iteratively, arguments appearing in the head of a rule such that none of the three conditions of Definition 2 holds.

Argument Restricted programs. For any atom $p(t_1, ..., t_n)$, $p(t_1, ..., t_n)^0$ denotes the predicate symbol p, whereas $p(t_1, ..., t_n)^i$, for $1 \le i \le n$, denotes its argument term t_i. The depth of a variable X in a term t that contains X, denoted by $d(X, t)$, is defined recursively as follows:

$$d(X,t) = \begin{cases} 0 & if\ t = X \\ 1 + max_{i:t_i\ contains\ X} d(X, t_i) & if\ t = f(t_1, ..., t_n) \end{cases}$$

▶ **Definition 3** ($\mathcal{AR}$ Program [16]). An argument ranking for a program $\mathcal{P}$ is a function ϕ from arguments to integers such that, for every rule r of $\mathcal{P}$, every atom A occurring in the head of r, and every variable X occurring in an argument term A^i, $body^+(r)$ contains an atom B such that X occurs in an argument term B^j satisfying the condition

$$\phi(A^0[i]) - \phi(B^0[j]) \ge d(X, A^i) - d(X, B^j)$$

A program is argument restricted ($\mathcal{AR}$) if it has an argument ranking. □

▶ **Example 4.** Consider the following logic program $\mathcal{P}_4$:

r_1 : $\texttt{succ(X, f(X))} \leftarrow \texttt{nat(X)}$.
r_2 : $\texttt{nat(0)}$.
r_3 : $\texttt{nat(Y)} \leftarrow \texttt{succ(X, Y), bounded(Y)}$.

where `bounded` is a base predicate. The argument graph $G^A(\mathcal{P}_4)$ contains the following edges $(nat[1], succ[1])$, $(nat[1], succ[2])$, $(succ[2], nat[1])$ and $(bounded[1], nat[1])$. The program is not finite domain as the argument $succ[2]$ is not finite domain. However, $\mathcal{P}_4$ is

argument restricted as it is possible to assign the following consistent ranking to arguments: $\phi(bounded[1]) = \phi(nat[1]) = \phi(succ[1]) = 0$ and $\phi(succ[2]) = 1$. □

The class of argument restricted programs is contained in finitely ground, generalizes the finite domain class and is decidable.

3 Safe programs

In this section we introduce a new criterion guaranteeing that there is a finite instantiation, equivalent to the source program and, therefore, a finite set of finite stable models.

▶ **Definition 5** (Activation Graph). Let $\mathcal{P}$ be a program, the *activation graph* $\Omega(\mathcal{P}) = (\mathcal{P}, E)$ consists of a set of nodes denoting rules and a set of edges E defined as follows: for each pair of rules r and s there is an edge (r, s) from r to s if there is a set of ground facts DB_1 and two matchers θ_1 and θ_2 such that

1. $DB_1 \models body(r)\theta_1 \wedge DB_1 \not\models head(r)\theta_1$ and
2. let $DB_2 = DB_1 \cup head(r)\theta_1$, the following conditions hold:
   - $DB_2 \models body(s)\theta_2 \wedge DB_2 \not\models head(s)\theta_2$ and
   - $DB_1 \not\models body(s)\theta_2 \vee DB_1 \models head(s)\theta_2$. □

▶ **Example 6.** Consider the program $\mathcal{P}_1$ of Example 1 and let $\Omega(\mathcal{P}_1) = (\mathcal{P}_1, E)$ its activation graph. We have that $(r_1, r_2) \in E$, but $(r_2, r_2) \notin E$, as the firing of r_2 cannot fire r_2 again. Clearly, being r_1 a fact, it cannot be fired by another rule. Therefore, $\Omega(\mathcal{P}_1)$ is acyclic. □

▶ **Definition 7** (Safe Function). For any program $\mathcal{P}$, let A be a subset of arguments of $\mathcal{P}$, $\Psi_{\mathcal{P}}(A)$ denotes the set of arguments occurring in $\mathcal{P}$ such that for all rules $r \in \mathcal{P}$ where q appears in the head

1. r does not appear in a cycle of $\Omega(\mathcal{P})$, or
2. let t be the term corresponding to argument $q[k]$, for every variable X appearing in $q[k]$ in the head of r (considering all head occurrences), X also appears in some argument in $body^+(r)$ belonging to A. □

The function $\Psi_{\mathcal{P}}$ is monotonic and, for every set of arguments A occurring in $\mathcal{P}$, the sequence $\Psi_{\mathcal{P}}(A)$, $\Psi^2_{\mathcal{P}}(A)$, ... , $\Psi^i_{\mathcal{P}}(A)$, ... converges in a finite number of steps, that is, there is some finite n such that $\Psi^n_{\mathcal{P}}(A) = \Psi^{n+1}_{\mathcal{P}}(A)$.

▶ **Definition 8** (Safe Arguments). For any program $\mathcal{P}$, $safe(\mathcal{P}) = \Psi^{\infty}_{\mathcal{P}}(A)$, where $A = FD(\mathcal{P})$ is the set of finite domain arguments of $\mathcal{P}$, denotes the set of safe arguments of $\mathcal{P}$. A program $\mathcal{P}$ is said to be *safe* if all arguments are safe. □

It is worth noting that the starting set to compute safe arguments could be the set of finite domain arguments satisfying condition i) or ii) of Definition 2, that is condition iii) is not necessary to compute safe arguments. We shall denote by $args(\mathcal{P})$ the set of arguments of a program $\mathcal{P}$ and by $aff(\mathcal{P}) = args(\mathcal{P}) - safe(\mathcal{P})$ the set of affected arguments. The class of safe programs will be denoted by $\mathcal{SP}$.

▶ **Example 9.** Consider the program $\mathcal{P}_4$ of Example 4. Although the activation graph is not acyclic (there is a cycle between r_1 and r_3), we have that i) $bounded[1]$, $nat[1]$ and $succ[1]$ are safe as they are finite domain, and ii) $succ[2]$ is safe as the variable X in the first rule appears in a safe body argument. Since all arguments are safe, we have that the program $\mathcal{P}_4$ is safe. □

▶ **Example 10.** The program $\mathcal{P}_1$ of Example 1 is safe, as rule r_2 does not fire itself and the graph $\Omega(\mathcal{P}_1)$ is empty. Moreover, it is not argument-restricted as it is not possible to assign a rank to $p[1]$ and $p[2]$ such that $\phi(p[1]) - \phi(p[j]) \geq d(X, f(X)) - d(X, X)$ and $\phi(p[2]) - \phi(p[j]) \geq d(X, g(X)) - d(X, X)$, with $j = 1, 2$. □

▶ Proposition 11. The problem of deciding whether a program $\mathcal{P}$ is safe is decidable. □

The following theorem states that the class of safe programs i) strictly contains the class of finite domain programs, ii) is not comparable with the class of argument-restricted programs, and iii) is contained in the class of finitely ground programs.

▶ **Theorem 12.** $\mathcal{FD} \subsetneq \mathcal{SP} \subsetneq \mathcal{FG}$, $\mathcal{AR} \not\subseteq \mathcal{SP}$ *and* $\mathcal{SP} \not\subseteq \mathcal{AR}$. □

▶ **Corollary 13.** *For any safe program $\mathcal{P}$, the stable models of $\mathcal{P}$ are finite.* □

From Theorem 12 it also follows that any safe program $\mathcal{P}$ has finitely many stable models and both brave and cautious reasoning over safe programs are computable even for non-ground queries.

4 Exploiting function symbols

In this section we further improve our technique by exploiting the role of function symbols for checking program termination under bottom-up evaluation. We assume that if the same variable X appears in two terms occurring in the head and body of a rule, then one of the two terms must be a subterm of the other and that the nesting level of complex terms is at most one. There is no real restriction in such an assumption as every program could be rewritten into an equivalent program satisfying such a condition. For instance, a rule of the form $p(f(h(X))) \leftarrow q(g(X))$ could be rewritten into the following two rules: $p(f(X)) \leftarrow p'(X)$, $p'(h(X)) \leftarrow p''(X)$ and $p''(X) \leftarrow q(g(X))$.

The following example shows a program admitting finite stable models, but previous criteria, included the safety criterion, are not able to detect it.

▶ **Example 14.** Consider the below program $\mathcal{P}_{14}$:

$\mathtt{r(f(X))} \leftarrow \mathtt{s(X)}$.
$\mathtt{q(f(X))} \leftarrow \mathtt{r(X)}$.
$\mathtt{p(X)} \leftarrow \mathtt{q(X)}$.
$\mathtt{n(X)} \leftarrow \mathtt{p(g(X))}$.
$\mathtt{s(X)} \leftarrow \mathtt{n(X)}$.

The program is neither safe, as all arguments are affected, nor argument restricted. □

The *(labelled) propagation graph* $\Delta(\mathcal{P})$ is the graph derived from the argument graph $G^A(\mathcal{P})$ by only considering affected arguments and adding labels to arcs.

▶ **Definition 15** (Labelled argument and propagation graphs)**.** Let $\mathcal{P}$ be a program, the *labelled argument graph* $\mathcal{G}^A_L(\mathcal{P}) = (args(\mathcal{P}), E)$, where E is a set of labelled edges defined as follows. For each pair of nodes $p[j], q[i] \in args(\mathcal{P})$ and for every rule $r \in \mathcal{P}$ such that i) there is an atom $q(u) \in body^+(r)$, ii) $head(r) = p(v)$ and iii) the same variable X occurs in both $q[j]$ and $p[i]$, there is an arc $(q[j], p[i], \alpha) \in E$ where

- $\alpha = \epsilon$ if $q[j] = p[i]$ and both arguments contain variables;
- $\alpha = f$ if $q[j] = X$ and $p[i] = f(..., X, ...)$;
- $\alpha = \overline{f}$ if $q[j] = f(..., X, ...)$ and $q[j] = X$.

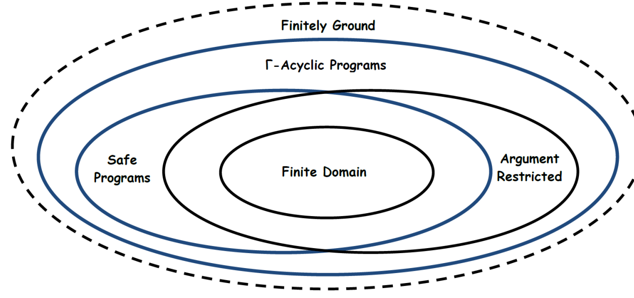


Figure 1 Criteria relationships.

The *(labelled) propagation graph* $\Delta(\mathcal{P})$ is the graph derived from the labelled argument graph $G_L^A(\mathcal{P})$ by only considering affected arguments. ◻

A path π is a sequence $n_1\,\alpha_1\,n_2\,\alpha_2\,...n_k\,\alpha_k\,n_{k+1}$, where $k \geq 1$ and for each $i \in [1..k]$ (n_i, n_{i+1}, α_i) is an edge of $\Delta(\mathcal{P})$. For any path $\pi = n_1\,\alpha_1\,n_2\,\alpha_2\,...\,n_k\,\alpha_k n_{k+1}$, we denote with $\lambda(\pi)$ the string $\alpha_1\,...\,\alpha_k$.

▶ **Definition 16.** Let $\mathcal{P}$ be a program and let $F = \{f_1, ..., f_m\}$ be the set of function symbols occurring in $\mathcal{P}$. The *grammar* $\Gamma_\mathcal{P}$ is a 4-tuple (N, T, R, S), where $N = \{S, S_1, S_2\}$ is the set of nonterminal symbols, $T = \{f \mid f \in F\} \cup \{\overline{f} \mid f \in F\}$ is the set of terminal symbols, S is the start symbol and R is the set of production rules below defined:

- $S \rightarrow S_1\,f_i\,S_2,\qquad \forall f_i \in F;$
- $S_1 \rightarrow f_i\,S_1\,\overline{f_i}\,S_1 \mid \epsilon,\qquad \forall f_i \in F;$
- $S_2 \rightarrow (S_1 \mid f_i)\,S_2 \mid \epsilon,\qquad \forall f_i \in F.$ ◻

The language $\mathcal{L}(\Gamma_\mathcal{P})$ is the set of strings generated by $\Gamma_\mathcal{P}$. As $\Gamma_\mathcal{P}$ is context free, the language $\mathcal{L}(\Gamma_\mathcal{P})$ can be recognized by means of a pushdown automaton. Given a grammar $\Gamma = \{N, T, R, S\}$ and a graph G, we say that path π in G spells a string $w \in \mathcal{L}(\Gamma)$ if $\lambda(\pi) = w$.

▶ **Definition 17** (Γ-acyclic Programs)**.** A program $\mathcal{P}$ is said Γ-*acyclic* if there is no cycle in $\Delta(\mathcal{P})$ spelling a string of $\mathcal{L}(\Gamma_\mathcal{P})$. ◻

Considering previous Example 14, the program $\mathcal{P}_{14}$ is Γ-acyclic, but not safe. Indeed, there is a cycle spelling the strings "$f\,f\,\overline{g}$", "$f\,\overline{g}\,f$" and "$\overline{g}\,f\,f$", but all strings do not belong to the language $\mathcal{L}(\Gamma_{\mathcal{P}_{14}})$. Observe that, in order to correctly recognize a cycle in $\Delta(\mathcal{P})$ spelling a string of $\mathcal{L}(\Gamma_\mathcal{P})$, we have to start from an edge with a positive label f (i.e. starting from an unlabelled edge or from an edge with a label $\overline{f}$ is not useful).

▶ Proposition 18. The problem of deciding whether a program is Γ-acyclic is decidable. ◻

The below theorem states that the class of acyclic programs strictly contains both classes of safe programs and argument restricted programs and is contained in the class of finitely ground programs.

▶ **Theorem 19.** $\mathcal{SP} \cup \mathcal{AR} \subsetneq \mathcal{AP} \subsetneq \mathcal{FG}$ ◻

▶ **Corollary 20.** *For any* Γ*-acyclic program* $\mathcal{P}$*, the stable models of* $\mathcal{P}$ *are finite.* ◻

The relationships among previous criteria and the new ones are reported in Fig. 1.

5 Computing stable models for Γ-acyclic programs

We now show how stable models for Γ-acyclic programs can be computed using current algorithms based on the grounding of programs. The idea is that, considering that positive normal Γ-acyclic programs have a finite minimum model, from a Γ-acyclic program $\mathcal{P}$, we first generate a standard Γ-acyclic program $st(\mathcal{P})$ such that all stable models of $\mathcal{P}$ are contained in the minimum model of $st(\mathcal{P})$ and next we generate a new program $ext(\mathcal{P})$ equivalent to $\mathcal{P}$, such that there is a ground, finite, equivalent version. The computation of the stable models of $ext(\mathcal{P})$ could be carried out by current answer set systems [4, 5, 6].

▶ **Definition 21** (Standard program). Let $\mathcal{P}$ be a logic program, $st(\mathcal{P})$ denote the normal, positive program, called standard version, obtained by replacing i) each disjunctive rule r having m atoms $a_1, ..., a_m$ in the head with m positive rules of the form $a_i \leftarrow body^+(r)$, for $1 \leq i \leq m$, and ii) each derived predicate symbol q with a new derived predicate symbol Q. □

▶ **Example 22.** Consider the program $\mathcal{P}_{22}$ consisting of the two rules

$$\texttt{p(X)} \vee \texttt{q(X)} \leftarrow \texttt{r(X)}, \neg\texttt{a(X)}.$$
$$\texttt{r(X)} \leftarrow \texttt{b(X)}, \neg\texttt{q(X)}.$$

where p, q and r are derived predicates (mutually recursive), whereas a and b are base predicates. The derived standard program $st(\mathcal{P}_{22})$ is:

$$\texttt{P(X)} \leftarrow \texttt{R(X)}.$$
$$\texttt{Q(X)} \leftarrow \texttt{R(X)}.$$
$$\texttt{R(X)} \leftarrow \texttt{b(X)}.$$

▶ **Lemma 23.** *Let $\mathcal{P}$ be a program and let $\mathcal{P}' = st(\mathcal{P}) \cup \{q(\bar{X}) \leftarrow Q(\bar{X}) \mid q \in dpred(\mathcal{P})\}$, where $dpred(\mathcal{P})$ denotes the set of derived predicate symbols in $\mathcal{P}$. For any stable model $M \in \mathcal{SM}(\mathcal{P})$, $M \subseteq \mathcal{MM}(\mathcal{P}')[S_\mathcal{P}]$, where $S_\mathcal{P}$ denotes the set of predicate symbols in $\mathcal{P}$.* □

For any rule r such that $head(r) = q_1(u_1) \vee \cdots \vee q_k(u_k)$, $headconj(r)$ denotes the conjunction $Q_1(u_1), ..., Q_k(u_k)$.

▶ **Definition 24** (Extended program). Let $\mathcal{P}$ be a disjunctive program and let r be a rule of $\mathcal{P}$, then, $ext(r)$ denotes the (disjunctive) extended rule $head(r) \leftarrow headconj(r), body(r)$ obtained by extending the body of r, whereas $ext(\mathcal{P}) = \{ext(r) \mid r \in \mathcal{P}\} \cup st(\mathcal{P})$ denotes the (disjunctive) program obtained by extending the rules of $\mathcal{P}$ and adding (standard) rules defining the new predicates. □

▶ **Example 25.** Consider the program $\mathcal{P}_{22}$ of Example 22. The extended program $ext(\mathcal{P}_{22})$ is as follows:

$$\texttt{p(X)} \vee \texttt{q(X)} \leftarrow \texttt{P(X)}, \texttt{Q(X)}, \texttt{r(X)}, \neg\texttt{a(X)}$$
$$\texttt{r(X)} \leftarrow \texttt{R(X)}, \texttt{b(X)}, \neg\texttt{q(X)}$$

plus the rules in $st(\mathcal{P}_{22})$ showed in Example 22. □

The following theorem states that $\mathcal{P}$ and $ext(\mathcal{P})$ are equivalent w.r.t. the set of predicate symbols in $\mathcal{P}$.

▶ **Theorem 26.** *For every program $\mathcal{P}$, $\mathcal{SM}(\mathcal{P})[S_\mathcal{P}] = \mathcal{SM}(ext(\mathcal{P}))[S_\mathcal{P}]$, where $S_\mathcal{P}$ is the set of predicate symbols occurring in $\mathcal{P}$.* □

6 Bound queries

The bottom-up computation of queries whose related programs are not range-restricted, could not be carried out, as the ground instantiation is infinite. The application of well known rewriting techniques, such as magic-set, may allow bottom-up evaluators to (efficiently) compute bounded queries, by rewriting queries so that the top-down evaluation is emulated [21, 22, 23, 17]. Before presenting our technique, let us introduce some notations.

A query is a pair $Q = \langle q(u_1,..,u_n), \mathcal{P}\rangle$, where $q(u_1,..,u_n)$ is an atom called query goal and $\mathcal{P}$ is a program. An *adornment* of predicate p with arity n is a string $\alpha \in \{b, f\}^*$ such that $|\alpha| = n$. The symbols b and f denote, respectively, bound and free arguments. Given a query $Q = \langle q(u_1,..,u_n), \mathcal{P}\rangle$, $MagicS(Q) = \langle q^\alpha(u_1,..,u_n), MagicS(q(u_1,..,u_n), \mathcal{P})\rangle$ denotes the rewriting of Q, where $MagicS(q(u_1,..,u_n), \mathcal{P})$ denotes the rewriting of rules in $\mathcal{P}$ with respect to the query goal $q(u_1,..,u_n)$ and α is the adornment associated with the query goal.

Since the magic-set rewriting technique has been defined for subclasses of queries (e.g. stratified queries), we assume that our queries are positive[2], although we could consider larger classes with the only necessary condition being that after their rewriting queries must be range restricted.

▶ **Definition 27.** A query $Q = \langle G, \mathcal{P}\rangle$ is said Γ-*acyclic* if either $\mathcal{P}$ or $MagicS(G, \mathcal{P})$ is Γ-acyclic. ◻

It is worth noting that it is possible to have a query $Q{=}\langle G, \mathcal{P}\rangle$ such that $\mathcal{P}$ is Γ-acyclic, but the rewritten program $MagicS(G, \mathcal{P})$ is not Γ-acyclic and vice versa.

▶ **Example 28.** Consider the query $Q = \langle \mathtt{p(f(f(a)))}, \mathcal{P}_{28}\rangle$, where $\mathcal{P}_{28}$ is defined below:

```
p(a).
p(f(X)) ← p(X).
```

$\mathcal{P}_{28}$ is not Γ-acyclic, but if we rewrite the program using the magic-set method, we obtain the Γ-acyclic program:

$\mathtt{p^b(a)} \leftarrow \mathtt{magic_p^b(a)}.$ $\mathtt{magic_p^b(f(f(a)))}.$
$\mathtt{p^b(f(X))} \leftarrow \mathtt{magic^b_p(f(X)),\ p^b(X)}.$ $\mathtt{magic_p^b(X)} \leftarrow \mathtt{magic_p^b(f(X))}.$

Consider now the query $Q = \langle \mathtt{p(a)}, \mathcal{P}'_{28}\rangle$, where $\mathcal{P}'_{28}$ is defined as follows:

```
p(f(f(a))).
p(X) ← p(f(X)).
```

The program is Γ-acyclic, but after the magic-set rewriting we obtain the below set of rules:

$\mathtt{p^b(f(f(a)))} \leftarrow \mathtt{magic_p^b(f(f(a)))}.$ $\mathtt{magic_p^b(a)}.$
$\mathtt{p^b(X)} \leftarrow \mathtt{magic^b_p(X), p^b(f(X))}.$ $\mathtt{magic_p^b(f(X))} \leftarrow \mathtt{magic_p^b(X)}.$

which is not Γ-acyclic. ◻

Thus, we propose to first check if the input program is Γ-acyclic and, if it does not satisfy Γ-acyclicity, to check the property on the rewritten program, which is query-equivalent to the original one.

[2] For positive queries we mean queries $\langle G, \mathcal{P}\rangle$ such that $\mathcal{P}$ is positive.

References

1 P. A. Bonatti, "On the decidability of fdnc programs," *Intellig. Artific.*, vol. 5, no. 1, 2011.
2 T. Eiter and M. Simkus, "Fdnc: Decidable nonmonotonic disjunctive logic programs with function symbols," *ACM Trans. Comput. Log.*, vol. 11, no. 2, 2010.
3 F. Lin and Y. Wang, "Answer set programming with functions," in *KR*, pp. 454–465, 2008.
4 N. Leone, G. Pfeifer, W. Faber, F. Calimeri, T. Dell'Armi, T. Eiter, G. Gottlob, G. Ianni, G. Ielpa, K. Koch, S. Perri, and A. Polleres, "The dlv system," in *Jelia*, pp. 537–540, 2002.
5 P. Simons, I. Niemel"a, and T. Soininen, "Extending and implementing the stable model semantics," *Artif. Intell.*, vol. 138, no. 1-2, pp. 181–234, 2002.
6 M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub, "*clasp* : A conflict-driven answer set solver," in *LPNMR*, pp. 260–265, 2007.
7 S. S. Huang, T. J. Green, and B. T. Loo, "Datalog and emerging applications: an interactive tutorial," in *SIGMOD Conference*, pp. 1213–1216, 2011.
8 D. D. Schreye and S. Decorte, "Termination of logic programs: The never-ending story," *J. Log. Program.*, vol. 19/20, pp. 199–260, 1994.
9 D. Voets and D. D. Schreye, "Non-termination analysis of logic programs with integer arithmetics," *TPLP*, vol. 11, no. 4-5, pp. 521–536, 2011.
10 R. Krishnamurthy, R. Ramakrishnan, and O. Shmueli, "A framework for testing safety and effective computability," *J. Comput. Syst. Sci.*, vol. 52, no. 1, pp. 100–124, 1996.
11 P. A. Bonatti, "Reasoning with infinite stable models," *Artif. Intell.*, vol. 156, no. 1, 2004.
12 S. Baselice, P. A. Bonatti, and G. Criscuolo, "On finitely recursive programs," *TPLP*, vol. 9, no. 2, pp. 213–238, 2009.
13 F. Calimeri, S. Cozza, G. Ianni, and N. Leone, "Computable functions in asp: Theory and implementation," in *ICLP*, pp. 407–424, 2008.
14 T. Syrjänen, "Omega-restricted logic programs," in *LPNMR*, pp. 267–279, 2001.
15 M. Gebser, T. Schaub, and S. Thiele, "Gringo : A new grounder for answer set programming," in *LPNMR*, pp. 266–271, 2007.
16 Y. Lierler and V. Lifschitz, "One more decidable class of finitely ground programs," in *ICLP*, pp. 489–493, 2009.
17 M. Alviano, W. Faber, and N. Leone, "Disjunctive asp with functions: Decidable queries and effective computation," *TPLP*, vol. 10, no. 4-6, pp. 497–512, 2010.
18 J. D. Ullman, *Principles of Database and Knowledge-Base Systems, Volume I.* Computer Science Press, 1988.
19 M. Gelfond and V. Lifschitz, "The stable model semantics for logic programming," in *ICLP/SLP*, pp. 1070–1080, 1988.
20 M. Gelfond and V. Lifschitz, "Classical negation in logic programs and disjunctive databases," *New Generation Comput.*, vol. 9, no. 3/4, pp. 365–386, 1991.
21 C. Beeri and R. Ramakrishnan, "On the power of magic," *J. Log. Program.*, vol. 10, no. 1/2/3&4, pp. 255–299, 1991.
22 S. Greco, "Binding propagation techniques for the optimization of bound disjunctive queries," *IEEE Trans. Knowl. Data Eng.*, vol. 15, no. 2, pp. 368–385, 2003.
23 G. Greco, S. Greco, I. Trubitsyna, and E. Zumpano, "Optimization of bound disjunctive queries with constraints," *TPLP*, vol. 5, no. 6, pp. 713–745, 2005.

Logic Programming in Tabular Allegories*

Emilio Jesús Gallego Arias[1] and James B. Lipton[2]

1 **Universidad Politécnica de Madrid**
2 **Wesleyan University**

Abstract

We develop a compilation scheme and categorical abstract machine for execution of logic programs based on allegories, the categorical version of the calculus of relations. Operational and denotational semantics are developed using the same formalism, and query execution is performed using algebraic reasoning. Our work serves two purposes: achieving a formal model of a logic programming compiler and efficient runtime; building the base for incorporating features typical of functional programming in a *declarative* way, while maintaining 100% compatibility with existing Prolog programs.

1998 ACM Subject Classification D.3.1 Formal Definitions and Theory, F.3.2 Semantics of Programming Languages, F.4.1 Mathematical Logic

Keywords and phrases Category Theory,Logic Programming,Lawvere Categories,Programming Language Semantics,Declarative Programming

Digital Object Identifier 10.4230/LIPIcs.ICLP.2012.334

1 Introduction

Relational algebras have a broad spectrum of applications in both theoretical and practical computer science. In particular, the calculus of binary relations [37], whose main operations are intersection ($\cup$), union ($\cap$), relative complement \, inversion $(_)^o$ and relation composition (;) was shown by Tarski and Givant [40] to be a complete and adequate model for capturing all first-order logic and set theory. The intuition is that conjunction is modeled by $\cap$, disjunction by $\cup$ and existential quantification by composition.

This correspondence is very useful for modeling logic programming. Logic programs are naturally interpreted by binary relations and relation algebra is a suitable framework for algebraic reasoning over them, including execution of queries.

Previous versions of this work [24, 32, 9, 22], developed operational and denotational semantics for constraint logic programming using distributive relational algebra with a quasi-projection operator. In this approach, all relations range over a unique domain or carrier: the set of hereditary sequences of terms generated by the signature of the program. For instance, the identity relation can be used to relate sequences of terms of an unbounded size.

Execution is performed using a rewriting system, but making it efficient is difficult given that untyped relations don't capture the exact number of logical variables in use. When a predicate call happens, the constraint store is duplicated, with one belonging to the caller environment and one used by the called predicate. At return time, the constraint stores are merged. The propagation of constraints posted inside a procedure call is delayed.

* The authors wants to acknowledge Wesleyan University for supporting this work with Van Vleck funds. This work is part of DESAFIOS10 (TIN2009-14599-C03-00)

© E.J. Gallego Arias and James B. Lipton;
licensed under Creative Commons License ND
Technical Communications of the 28th International Conference on Logic Programming (ICLP'12).
Editors: A. Dovier and V. Santos Costa; pp. 334–347

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

We propose to remedy this shortcoming by using *typed* relations. The theory of allegories [21], provides a categorical setting for distributive relational algebras. In this setting, relations are typed and the semantics for our relations becomes sequences of fixed-length. Now, the notion of categorical product and its associated projections interpret in an adequate way the shared context required to have an efficient execution model.

The most important concepts in our work are the notion of *strictly associative product* and *tabular relation*. Given types A, B (or *objects* in categorical language), we write $A \times B$ for their cartesian product. As usual $A \times (B \times C)$ is isomorphic ($\approx$) to $(A \times B) \times C$. We say our products are *strictly associative* if the isomorphism is an equality. That is, $(A \times B) \times C = A \times (B \times C)$. We are thus allowed to write $A \times B \times C$. This is a crucial fact for our machine, since if we interpret a chosen type H as a memory cell, then a memory region of size n is interpreted as H^n.

Second, we say a relation $R : A \leftrightarrow B$ is tabulated by an injective (monic) function (arrow) $f : C \to A \times B$ if every pair of the relation is in its image. We may split f into its components $f;\pi_1 : C \to A$ and $f;\pi_2 : C \to B$, and state that the pair $(f;\pi_1, f;\pi_2)$ tabulates R. Such a concept is fundamental for two reasons: the types of the tabulations carry important information about the memory use of the machine. The domain of the tabulations corresponds to *global storage* or heap and the co-domain represents the number of *registers* our machine is using at a given state.

The execution mechanism is entirely based on the composition of tabular relations, an operation fully characterized by the pullback of its tabulations. Relation composition models unification, parameter passing, renaming apart, allocation of new temporary variables and garbage collection.

The first important benefit of our use of categorical concepts is the small gap from the categorical specification to the actual machine and proposed implementation. This allows us to reason using a very convenient algebraic style, immediately witnessing the impact of such reasoning on the machine itself. Our philosophy is that in an fully algebraic framework, efficient execution should belong to regular reasoning. Real world implementations usually depart from this view in the name of efficiency, and one key objective of this work is to achieve efficiency without abandoning the algebraic approach. It is also worth noting that in our framework, we replace all the custom theory and meta-theory used in logic programming with category theory. The precise statement is that a Σ-allegory captures all the needed theory and meta-theory for a Logic Program with signature Σ, from set-theoretical semantics down to efficient execution.

The second — and in our opinion, most innovative benefit — is the possibility of seamlessly extending Prolog using constructions typical of functional programming in a fully *declarative way*. In [23], we sketch some of these extensions, adding algebraic data types, constraints, functions and monads to Prolog, all of it without losing source code compatibility with existing programs.

2 Logic Programming

Assume a permutative convention on symbols, i.e., unless otherwise stated explicitly, distinct names f, g stand for different entities (e.g. function symbols) and the same with distinct names i, j, for indices. A first-order language consists of a signature $\Sigma = \langle \mathcal{C}_\Sigma, \mathcal{F}_\Sigma \rangle$, given by $\mathcal{C}_\Sigma$, the set of constant symbols, and $\mathcal{F}_\Sigma$, the set of term formers or function symbols. $\mathcal{P}$ will denote the set of predicate symbols. Function $\alpha : \mathcal{P} \cup \mathcal{F}_\Sigma \to \mathbb{N}$ returns the arity of its predicate argument. We assume a set $\mathcal{X}$ of so-called *logic variables* whose members are

denoted x_i. We write $\mathcal{T}_\Sigma$ for the set of closed terms over Σ. We write $\mathcal{T}_\Sigma(\mathcal{X})$ for the set of open terms (in the variables in $\mathcal{X}$) over Σ. We drop Σ when understood from context. We write sequences of terms using vector notation: $\vec{t} = t_1, \ldots, t_n$. The length of such a sequence is written $|\vec{t}| = n$. We assume standard definitions for atoms, predicates, programs, clauses, and SLD resolution. For more details see [33].

3 Category Theory

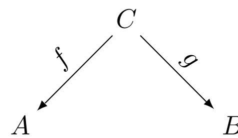
A category $\mathcal{C} = \langle \mathcal{O}, \mathcal{A} \rangle$ consists of a collection of objects $\mathcal{O}$ and typed arrows $\mathcal{A}$. For every object $A \in \mathcal{O}$, there is an identity arrow $id_A : A \to A \in \mathcal{A}$. Given arrows $f : A \to B$ and $g : B \to C$, its composition $f; g : A \to C$ is defined. For $f : A \to B$, we call A the domain of f and B its codomain. Composition is associative and $id_A; f = f; id_B = f$. We assume knowledge of the concepts of *commutative diagram*, product, equalizer, pullback, monic arrow and subobject [6, 5, 30].

For a product $A \times B$, we will write $\pi_1^{A \times B} : A \times B \to A$ and $\pi_2^{A \times B} : A \times B \to B$ for the projections. For arrows $f : C \to A$, $g : C \to B$ we write $\langle f, g \rangle$ for the unique product former. Several definitions exist for Regular Categories [11, 6, 27, 21]; we use the latter presentation.

▶ **Definition 1** (Regular Category). A category $\mathcal{C}$ is a Regular Category if it has products, equalizers, images and pullback transfer covers. A Regular Category can be used to generate a tabular allegory. Indeed, Regular Categories give rise to categories of relations.

3.1 Categorical Relations

▶ **Definition 2** (Monic Pair). $f : C \to A$ and $g : C \to B$ is a monic pair iff $\langle f, g \rangle : C \rightarrowtail A \times B$ is monic. A monic pair is a subobject of $A \times B$, thus we can see it as a relation from A to B:



▶ **Definition 3** (Composition of Relations). The composition (u, v) of a relation (f, g) with (h, i) is defined by the diagram on the left in Fig. 1. Note that the purpose of the cover in that diagram is to ensure that the resulting relation remains a monic pair. The right diagram shows the already composed relation.

▶ **Definition 4** (Categories of Relations). For a regular category $\mathcal{C}$, the category $Rel(\mathcal{C})$ of relations has the same objects as $\mathcal{C}$, arrows $A \to B$ are monic pairs $(f : C \to A, g : C \to B)$ and composition is defined as above. $\mathcal{C}$ is a sub-category of $Rel(\mathcal{C})$. The inclusion functor sends an arrow $f : A \to B$ to the pair (id, f). If a morphism of $Rel(\mathcal{C})$ is in $\mathcal{C}$, we call it a *map*.

Given a relation (f, g), its inverse, or reciprocal is (g, f). The natural order-isomorphism $Sub(A \times B) \approx Rel(A, B)$ yields a semi-lattice structure on $Rel(A, B)$.

3.2 Lawvere Categories

A Lawvere category is a category $\mathcal{C}$ with a denumerable set $\mathbb{N}$ of distinct objects, where each object N is the n-th power of the object 1. 0 is the terminal object. We write $!_A : A \to 0$ for the terminal arrow. The product of $T^m \times T^n$ is T^{m+n}. Products are *strictly associative*

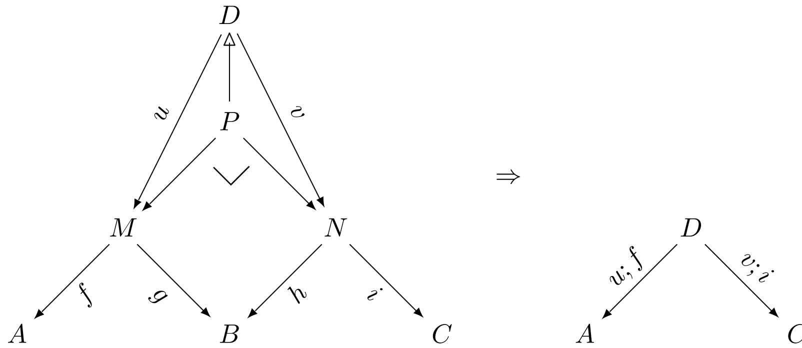


Figure 1 Composition of Relations.

since addition is associative, thus $((1 \times 1) \times 1) = (1 \times 2) = 3$. Note that this means $(id_2 \times id) : 2 \times 1 \to 2 \times 1 = id_3 : 3 \to 3$, or for $f : 2 \to 2$, $(f \times id_2) = \langle f; \pi_1, f; \pi_2, id_1, id_1 \rangle$, etc...

For a given signature Σ of a logic program, we build the corresponding (free or syntactic) Lawvere Category $\mathcal{C}_\Sigma$ as follows:

- For every constant $a \in \mathcal{T}_\Sigma$, we freely adjoin an arrow $a : 0 \to 1$.
- For every function symbol $f \in \mathcal{T}_\Sigma$ with arity $\alpha(f) = N$, we freely adjoin an arrow $f : N \to 1$.

A model of a Lawvere Category $\mathcal{C}$ is a functor $F : \mathcal{C} \to Set$ which preserves finite products and pullbacks. A homomorphism of $\mathcal{C}$-models is a natural transformation. The category of models $Mod(\mathcal{C}, Set)$ for $\mathcal{C}$ is the usual functor category.

Lawvere Categories are a natural framework for categorically representing algebraic theories. Examples of such categories $\mathcal{C}$ may be seen in [31], and some good treatments are in [6, 26].

3.3 Allegories

▶ **Definition 5** (Allegory). An allegory $\mathcal{R} = \{\mathcal{O}, \mathcal{A}\}$ is an enriched category, with objects $\mathcal{O}$ and *relations* $\mathcal{A}$. We write $R; S : A \to C$ for composition of relations $R : A \to B$ and $S : B \to C$. When there is no confusion possible we may also write RS for $R; S$. We add two new operations:

- For every relation $R : A \to B$ and $S : A \to B$, $(R \cap S) : A \to B$ is a relation.
- For every relation $R : A \to B$, $R^\circ : B \to A$ is a relation.

We write $R \subseteq S$ for $R \cap S = R$. The new operations obey the following laws:

$$
\begin{array}{rclrcl}
R \cap R & = & R & R \cap S & = & S \cap R \\
R \cap (S \cap T) & = & (R \cap S) \cap T & R^{\circ\circ} & = & R \\
(RS)^\circ & = & S^\circ; R^\circ & (R \cap S)^\circ & = & (R^\circ \cap S^\circ) \\
R; (S \cap T) & \subseteq & (R; S \cap R; T) & (R; S \cap T) & \subseteq & (R \cap T; S^\circ); S
\end{array}
$$

A *map* is a relation such that $R^\circ; R \subseteq id$ and $id \subseteq R; R^\circ$. We use capital letters for relations and small letters for maps. A relation R is coreflexive iff $R \subseteq id$. For an allegory $\mathcal{R}$, we shall denote its subcategory of maps by $Map(\mathcal{R})$. A pair of maps f, g tabulates a relation R iff $f^\circ; g = R$ and $f; f^\circ \cap g; g^\circ = 1$. The latter condition is equivalent to stating that f, g form a monic pair.

$$
\begin{array}{ccc}
A \xleftrightarrow{R} B \text{ with } f^\circ : A \to C,\ g : C \to B & = & A \xleftrightarrow{R} B \text{ with } f : C \to A,\ g : C \to B \qquad B \xleftrightarrow{R^\circ} A \text{ with } g : C \to B,\ f : C \to A
\end{array}
$$

It is easy to prove that a tabulation is unique up to isomorphism. A coreflexive relation $R \subseteq id$ is tabulated by a pair of the form (f, f). If $R = f^\circ; g$, then $R^\circ = g^\circ; f$.

An allegory is a tabular allegory iff every relation has a tabulation. For an allegory $\mathcal{R}$, $Map(\mathcal{R})$ is a regular category. The following lemma tells us that a tabular allegory really is the relational extension generated by its maps and that the concepts of regular category and tabular allegory intimately connected:

▶ **Lemma 6.** *If $\mathcal{R}$ is a tabular allegory then $\mathcal{R} \approx Rel(Map(\mathcal{R}))$. If $\mathcal{C}$ is a regular category then $\mathcal{C} \approx Map(Rel(\mathcal{C}))$. If $\mathcal{R} \approx Rel(\mathcal{C})$ then $Map(\mathcal{R}) \approx \mathcal{C}$.*

Proof. See [21] 2.147 and 2.148, 2.154. ◀

Composition of relations in a tabular allegory is thus defined in the same way as for categories of relations arising from a regular category, see Def. 3.

A distributive allegory is an allegory with a new relation denoted $\mathbf{0}_{AB}$ for every object A, B, and given relations R, S with the same type, $R \cup S$ is an arrow. They obey the following laws:

$$
\begin{array}{rclrcl}
R \cup R & = & R & R \cup S & = & S \cup R \\
R \cup (S \cup T) & = & (R \cup S) \cup T & \mathbf{0} \cup S & = & S \\
R \cup (R \cap S) & = & R & R; \mathbf{0} & = & \mathbf{0} \\
R(S \cup T) & = & RS \cup RT & R \cap (S \cup T) & = & (R \cap S) \cup (R \cup T)
\end{array}
$$

4 Regular Lawvere Categories and Σ-Allegories

The key idea is to use Lem. 6 to build an allegory from a Lawvere category. In order to do that, we need to define the concept of Regular Lawvere Category (RLC) $\mathcal{C}$ first. Then $Rel(\mathcal{C})$ generates a pre-Σ-allegory. However, this category is not distributive, so we $\cup$-complete it in order to obtain what we call a Σ-allegory.

▶ **Definition 7** (Regular Lawvere Category). Given a Lawvere Category $\mathcal{C}$, we build its regular completion $\hat{\mathcal{C}}$ by adjoining an initial object $\perp$, the corresponding initial arrows $?_A : \perp \to A$ for every object A and applying the quotient $?_A; f = ?_B$ for any arrow $f : A \to B$.

This completion effectively replaces the Lawvere Category concept of *existence* of an equalizer by the question: What is the domain of the equalizing arrow? Arrows not having an equalizer in $\mathcal{C}$ are equalized by $\perp$ in $\hat{\mathcal{C}}$.

▶ **Definition 8** (Initial Model). Given a choice $\langle , \rangle$ of product in Set, and a choice of symbols for the signature Σ generating the Regular Lawvere Category $\mathcal{C}$ and set $\mathcal{T}_\Sigma$, the initial model of a Regular Lawvere Category $\mathcal{C}$ — that is to say, the initial object in $Mod(\mathcal{C}, Set)$ — is the functor $[\![]\!]$, with object and arrow components $([\![]\!]_O, [\![]\!]_A)$:

$$[\![\bot]\!]_O = \emptyset \qquad [\![0]\!]_O = \{\bullet\} \qquad [\![N]\!]_O = \mathcal{T}_\Sigma^N \quad N > 0$$

$$\begin{array}{lcl}
[\![?_N]\!]_A & = & \emptyset \xrightarrow{\emptyset} [\![N]\!]_O \\
[\![!_N]\!]_A & = & \lambda x.\, \bullet \\
[\![c : 0 \to 1]\!]_A & = & \lambda \bullet .\, c \\
[\![f : N \to 1]\!]_A & = & \lambda(n_1, \ldots, n_N).\ f(n_1, \ldots, n_N) \\
[\![\pi_i : N \to 1]\!]_A & = & \lambda(n_1, \ldots, n_N).\ n_i \\
[\![\langle t_1, \ldots, t_N\rangle : M \to N]\!]_A & = & \lambda n.\langle [\![n]\!]_A ; [\![t_1]\!]_A, \ldots, [\![n]\!]_A ; [\![t_N]\!]_A \rangle
\end{array}$$

▶ **Lemma 9.** *The regular completion of a Lawvere Category is a regular category.*

4.1 Σ-Allegories

A RLC cannot model disjunctive clauses in logic programs, as it doesn't tabulate distributive allegories, which are tabulated by a Pre-Logos [21]: regular categories whose subobjects form a complete lattice, not just a semi-lattice.

▶ **Definition 10** (Σ-Allegory)**.** Given a Regular Lawvere Category $\mathcal{C}$, we define a Σ-allegory $\mathcal{R}_\cup$ as the distributive allegory generated from the allegory $\mathcal{R} \approx Rel(\mathcal{C})$ by freely adding all union arrows and taking the quotient by the distributive laws. An inclusion functor $F : \mathcal{R} \to \mathcal{R}_\cup$ exists, and it is easy to see that all the relations in $\mathcal{R}_\cup$ that possess a union-free representation are tabular.

5 Translation of the Program

The translation procedure is almost identical to the one defined in [22]. A predicate is translated to a coreflexive relation. We use two helper relations, a partial identity I, is meant to *create* and *destroy* local (or existentially quantified) variables, and a permutation W, which puts the arguments in the right order for relation composition.

▶ **Definition 11** (I Relation)**.** The relation I_{MN}, with $M < N$ is tabulated by $(\langle \pi_1, \ldots, \pi_M\rangle, id_N)$.

This relation formalizes the intuition that the reciprocal of a projection creates a new variable, indeed $I_{12} = \pi_1^\circ$.

▶ **Definition 12** (W Relation)**.** For a projection $w : N \to M$, with $N \geq M$ and $K = N - M$, we denote by $w' : N \to N$ any of its extensions to a permutation such that the following equations are satisfied: $\{w'(K) = w^{-1}(1), \ldots, w'(K + M) = w^{-1}(M)\}$. For a given w', W is tabulated by $(N, \langle \pi_{w'(1)}, \ldots, \pi_{w'(N)}\rangle)$.

First, we complete every predicate in a similar way to Clark's [15]. The set of n variables occurring in the terms is renamed from y_1 to y_n. Then, every term t_i occurring as an argument in the head and tail is replaced by a fresh variable x_i, and the equation $x_i = t_i$ is added to the clause. After that process, clauses are of the form:

$$p(\vec{x}') \leftarrow \vec{x} = \vec{t}(\vec{y}), p_1(\vec{x}_1), \ldots, p_n(\vec{x}_n).$$

$\vec{x}'$ a prefix of $\vec{x}$, $\vec{x}_i$ a selection of variables in $\vec{x}$ and $\vec{t}$ a sequence of terms using variables in $\vec{y}$. We replace $\vec{x}_i$ for projections $w_i(\vec{x})$ such $w_i(\vec{x}) = \vec{x}_i$. Clauses are now of the form:

$$p(\vec{x}') \leftarrow \vec{x} = \vec{t}(\vec{y}), p_1(w_1(\vec{x})), \ldots, p_n(w_n(\vec{x})).$$

Now we are ready to transform the clause into a relational term. The equation $\vec{x} = \vec{t}(\vec{y})$ is translated to a coreflexive relation between sequences of terms $K(\vec{t})$, of type $|\vec{t}| \to |\vec{t}|$, tabulated by an arrow $|\vec{y}| \to |\vec{t}|$.

▶ **Definition 13** (Term Translation)**.** The translation function K takes a sequence of terms $\vec{t}$, using $\vec{y} \equiv [y_1, \ldots, y_{|\vec{y}|}]$ variables and returns a coreflexive tabular relation $K(\vec{t}) : |\vec{t}| \to |\vec{t}|$ with tabulation $f : |\vec{y}| \to |\vec{t}|$.

$$\begin{array}{lcl} K(\vec{t}[\vec{y}]) & = & \langle K_{\vec{y}}(t_1), \ldots, K_{\vec{y}}(t_n)\rangle^\circ ; \langle K_{\vec{y}}(t_1), \ldots, K_{\vec{y}}(t_n)\rangle \\ \text{where} & & \\ K_{\vec{y}}(a) & = & !_{|\vec{y}|}; a : |\vec{y}| \to 1 \\ K_{\vec{y}}(y_i) & = & \pi_i : |\vec{y}| \to 1 \\ K_{\vec{y}}(f(t_1, \ldots, t_n)) & = & \langle K_{\vec{y}}(t_1), \ldots, K_{\vec{y}}(t_n)\rangle; f : \alpha(\vec{y}) \to 1 \end{array}$$

The tabulation could be seen as a constructor for $\vec{t}$ from a supply of fresh variables $\vec{y}$. We must wrap the predicates with the relational projection W_i generated from w_i, let $A_i = N - \alpha(p_i)$:

$$K(\vec{t}); W_1; (id_{A_1} \times \overline{p_1}); W_1^\circ; \ldots; W_n; (id_{A_n} \times \overline{p_n}); W_n^\circ$$

Note that we have replaced $\cup$ by composition. This is possible thanks to the fact that the relation $(id_{A_1} \times \overline{p_1})$ is coreflexive, thus the equation $A \cap B = A; B$ holds. Note that the presented arrow has type $N = |vt|$, while the arrow for the predicate should have a type of M. We use the $I_{MN} : M \to N$ to fix this and we obtain the final form. The final translation for the clause is:

$$I_{MN}; (K(\vec{t}); W_1; (id_{A_1} \times \overline{p_1}); W_1^\circ; \ldots; W_n; (id_{A_n} \times \overline{p_n}); W_n^\circ); I_{MN}^\circ$$

A predicate p consisting of several clauses is then translated using $\cup$:

$$\begin{array}{l} p(\vec{x}) \leftarrow cl_1 \vee \cdots \vee cl_m \quad \to \\ \overline{p} = C_1 \cup \cdots \cup C_m \end{array}$$

where C_i is the arrow corresponding to the translation of the clause cl_i.

▶ **Theorem 14** (Adequacy of the Translation)**.** *Given a predicate p of arity N translated to the arrow $\overline{p} : N \to N$, the initial model maps $\overline{p}$ to the subobject $\xrightarrow{[\![\overline{p}]\!]_A} \mathcal{T}_\Sigma^N$ such that its image is precisely the set of ground terms making p true.*

6 Specification of The Machine

We abuse notation to profit from the fact that a coreflexive relation is uniquely tabulated by a monic $f^\circ; f$ to write f for $f^\circ; f$ when it can be deduced from the context.

We define the categorical machine as a set of transition rules over relations. We write $(f \mid g)$ for tabular relations. Then, $(f \mid g); (f' \mid g')$ is rewrote to $(h; f \mid h'; g')$ using the pullback (h, h') of g, f'. This corresponds to a substitution, where the arrow $h : M \to N$ takes a current state of the machine using N variables to a state using M variables, and $h' : M' \to N'$ does the same, usually instantiating the translations of a clause to the right variables. This mechanism is also used for variable creation/destruction. The pair of arrows (h, h') above the transition arrow denotes the result of the pullback.

A union $R_1 \cup \cdots \cup R_n$ is used to represent disjunctive search, while predicate calls are represented as $(f \mid \langle g, [R]\rangle)$, where R is the relation pertaining to the call in-progress. Note

that g and the left tabulation of R share the same domain, allowing the propagation of substitutions resulting from reducing R to the outer context.

$$
\begin{array}{llll}
(f \mid g);(f' \mid g') & \xRightarrow{(h,h')} & (h;f \mid h';g') & \\
(f \mid \langle g_K, g_N\rangle);(id_K \times \overline{p_N}) & \Rightarrow & (f \mid \langle g_K, [(g_N \mid g_N);p_1]\rangle) \cup & \\
 & & \vdots \qquad\qquad\qquad\qquad \cup & \\
 & & (f \mid \langle g_K, [(g_N \mid g_N);p_n\rangle]) & \\
(f \mid \langle g, [(g' \mid g')]\rangle) & \Rightarrow & (f \mid \langle g, g\rangle) & \\
(f \mid \langle g, [E]\rangle) & \Rightarrow & (h;f \mid \langle h;g, [E']\rangle) & \text{iff } E \Rightarrow E' \\
R \cup S & \Rightarrow & R' \cup S & \text{iff } R \Rightarrow R' \\
\mathbf{0} \cup S & \Rightarrow & S &
\end{array}
$$

The first rule represents composition of tabular relations. The second one represents predicate call. First, disjunctive predicates are unfolded using the rule $f;(R \cup S) = f;R \cup f;S$. Computing the predicate call is performed by the relation $(g_N \mid g_N);p_1$. The third rule deals with return. The three last rules encode the search strategy of the machine. We include an example in Appendix A.

▶ **Theorem 15** (Operational equivalence). *$\langle p_1(\vec{u}_1), \ldots, p_n(\vec{u}_n)\rangle \to \cdots \to \Box$ is the SLD derivation with substitution σ iff*

$$K(\vec{u});W_1;\overline{p_1};W_1^\circ;\ldots;W_n;\overline{p_n};W_n^\circ \Rightarrow K(\sigma(\vec{u})) \cup R$$

7 The Pullback Algorithm

The core of the machine is pullback calculation. We present a pullback calculation algorithm for an arbitrary Regular Lawvere Category $\mathcal{C}$ generated from a signature Σ. The equational theory of $\mathcal{C}$ is the basis for the algorithm.

To improve the presentation, we reduce the pullback problem to its equivalent equalizer formulation. We start with a non commutative diagram and rewrite it until we reach a commutative one, which is an equalizer, and thus we obtain a pullback. The notion of substitution is an arrow composition followed by normalization modulo the product equational theory.

▶ **Definition 16** (Pullback Problem). A pullback problem is given by two arrows $f : N \to M$ and $g : N' \to M$.

▶ **Definition 17** (Arrow Normalization). We write $\to_R^!$ for the associated normalizing relation based on $\to_R$:

$$
\begin{array}{lllllll}
h;\langle f,g\rangle & \to_R & \langle h;f,h;g\rangle & \qquad \langle f,g\rangle;\pi_1 & \to_R & f & \\
\langle f,g\rangle;\pi_2 & \to_R & g & \qquad f;!_N & \to_R & !_M & \quad f : M \to N
\end{array}
$$

▶ **Definition 18** (Starting Diagram). For a pullback problem, its pre-starting diagram $\mathcal{P}$ is:

$$N \times N' \underset{\pi_2;g}{\overset{\pi_1;f}{\rightrightarrows}} M$$

Products are strictly associative, so π_2 is a renaming, for instance if $f = \langle \pi_1 \rangle$ and $g = \langle\langle \pi_1, \pi_2\rangle; f\rangle$, then $\pi_2 : 3 \to 2$ is equal to $\langle \pi_2, \pi_3\rangle$, and $\pi_2;g = \langle\langle \pi_2, \pi_3\rangle; f\rangle$. If $\pi_1;f \to_R^! f'$

and $\pi_2; g \to^!_R g'$, the starting diagram $\mathcal{P}$ is:

$$N + N' \xrightarrow{id = \langle \pi_1, \ldots, \pi_{N+N'} \rangle} N + N' \underset{g'}{\overset{f'}{\rightrightarrows}} M$$

$N + N'$ is the *type* of the pullback problem.

▶ **Definition 19** (Algorithm State)**.** For a pullback problem of type N, the algorithm state is $(S \mid h)$, $h : N \to N$ an arrow and S an ordered set of equations $f \approx g$ between arrows $f, g : N \to 1$.

▶ **Definition 20** (Auxiliary Substitution)**.** The helper substitution function is $S(i, f : N \to 1, h : N \to N) = h'$, where $\langle \pi_1, \ldots, \pi_{i-1}, f, \pi_{i+1}, \ldots, \pi_N \rangle; h \to^!_R h'$. This function replaces any π_i in h for f.

▶ **Definition 21** (Pullback Calculation Algorithm)**.** The input of the algorithm is two arrows $f_0 : N \to M$ and $g_0 : N' \to M$. First, build the starting diagram $\mathcal{P}$, which produces arrows f'_0 and g'_0, and a type of the problem $N + N' = N_T$. f'_0 and g'_0 are of the form $\langle f_1, \ldots, f_M \rangle$, $\langle g_1, \ldots, g_M \rangle$, then build the initial set $S = \{f_1 \approx g_1, \ldots, f_M \approx g_M\}$. The initial state is $(S \mid \langle \pi_1, \ldots, \pi_{N+N'} \rangle)$. The algorithm proceeds to transform the state $(S \mid h)$ iteratively until $S = \emptyset$ using the following rules

- Pick an equation from S such that $S = \{f \approx g\} \cup S'$. Compute $h; f \to^!_R f'$ and $h; g \to^!_R g'$. Then, do case analysis on $f' \approx g'$:

$$\begin{array}{llllllll}
!_M; a & \approx & !_M; b & \Rightarrow & \mathit{Fail} & \pi_i & \approx & \pi_j & \Rightarrow & (S' \mid S(j, \pi_i, h)) \\
!_M; a & \approx & h; f & \Rightarrow & \mathit{Fail} & \pi_i & \approx & g; f & \Rightarrow & (S' \mid S(i, g; f, h)) \\
g; f & \approx & g'; f' & \Rightarrow & \mathit{Fail} & !_M; a & \approx & \pi_i & \Rightarrow & (S' \mid S(i, !_M; a, h)) \\
!_M; a & \approx & !_M; a & \Rightarrow & (S' \mid h) & g; f & \approx & g'; f & \Rightarrow & (\{g_1 \approx g'_1\} \cup \ldots \\
 & & & & & & & & & \{g_n \approx g'_n\} \cup S' \mid h)
\end{array}$$

When $S = \emptyset$, our diagram is commutative but may not be an equalizer due to having an incorrect domain. We create a new arrow from h such that it is a monic. Discarding the K unused elements of M — is enough. Compose $h : M \to M$ with any extension of id_{M-K} to M to obtain $h' : (M - K) \to M$. This process is similar to *garbage collection* and memory de-fragmentation. If the algorithm fails, the equalizer is the initial arrow. Like many actual Prolog implementations, we don't implement occur-check. To get full soundness we would need to implement the occurs check in rule 7.

8 Implementation Discussion

We briefly present the most important points about the efficient implementation of the machine presented in Sec. 6 and Sec. 7. An implementation should be based on the interpretation of projections as pointers, with any π_i appearing inside a term being a pointer to a cell i.

The codomain of the tabulations may be seen as a set of registers, thus, for a pullback between $\langle !_1; f, \pi_1 \rangle$ and $\langle a, b \rangle$, we may assume that the registers are $X_1 = !_1; f$ and $X_2 = \pi_1$ and emit instructions `testc a, X1` and `testc b, X2`.

Note that the model presented here *forces* garbage collection and compaction. Every unused slot is eliminated by the pullback algorithm. We may fix our model by creating $\mathbb{N}$ copies of the object T with their corresponding products. Then, the T_i object becomes a representative of the memory cell i, and the denotational model captures the instantiation of

a variable as the variation of the tabulation domain from $(T_1 \times T_2 \times T_3)$ to $(T_1 \times T_3)$. This yields a memory behavior close to a standard WAM without garbage collection.

In order for the code to look reasonable we need to implement two optimizer engines. The first one is an algebraic one and perform tasks like statically computing the tabulation of $I_{MN}; K(\vec{t})$. The second one is a peephole optimizer.

9 Related Work

Algebraic approaches to logic programming have been tried in [29, 3, 20, 2, 16, 4]. The most important difference with our work is that all of them are based on the notion of *indexed category* and don't make a proposal for a concrete implementation. As in our proposal, the use of pullbacks is key point.

A different line of work is interpretation of logic programming as functional programs. The most representative works are [39, 41, 8, 36]. In [7], the authors study relational semantics for lazy functional logic programming language, modeling adequately the interactions between function call and non-determinism. In [10] the authors propose a diagram-based semantics for Logic Programming. An very interesting related work is [34]. This is the only proposal that we know of for the use of tabular allegories in programming. Unfortunately, McPhee's work does not develop an executable model. The use of category theory as a foundational tool for a machine is not new, the best known work is [17].

Several approaches to virtual machine generation [35, 18] and compiler verification [38] for Prolog exist. Several relation-based programming languages exist [19, 14, 13, 12, 25]. In [42], a similar effort to our semantics is developed, but the framework chosen is Tarski's cylindrical algebras instead Freyd's allegories. The author doesn't consider the implementation and efficiency of his approach.

In [1], the authors propose a first-order encoding for allegories. This is related our previous relation rewriting approach and indeed we consider their work very useful for mechanizing our theory. An encoding of allegories in a dependently-typed programming language is presented in [28]. We think Kahl's approach may help us to certify our compiler.

10 Conclusions and Future Work

We have presented an algebraic approach to Logic Programming, from the semantic base of category and allegory theory down to an actual machine based on which can be efficiently implemented. Our approach is new and has important advantages. First, as the algebraic connection between the different layers of the machine is not lost, reasoning in a layer is immediately reflected by the others. Additions on the semantics foster modifications to the algorithm as can be seen in [23]. In the other direction, a good example is the effect that memory layout has on incorporating T_i objects representing memory cells. Second, the correctness of the machine is easy to check. Composition of relations together with the equation $R; (S \cup T) = R; S \cup R; T)$ capture in a simple way the operational semantics and memory layout of Prolog. Our framework is well suited to prove semantic properties, given that our semantics are compositional and use the well established frameworks of category theory and relation algebra. Third, the use of such frameworks favors the reuse of existing technologies in other areas of programming.

We are actively working on an definitive instruction set. We don't want it to be specific to an operational choice like SLD, given that our approach is well suited to accommodate other strategies like breadth-first search. On the other hand, we are already developing extensions

to Prolog in [23], and some of them, such as higher-order types may require that we add second primitive of reduction to our machine.

In the future, we will mechanize all the theory presented here, and indeed we hope that effort will bring us close to the goal of having a fully verified implementation. We are working in extending Regular Lawvere Categories to Pre-Logos.

A An Example

We use as example the classical `add` predicate implementing Peano addition:

```
add(o,X,X).
add(s(X), Y, s(Z)) :- add(X, Y, Z).
```

A.1 Translation

We perform the renaming procedure similar to Clark's completion:

```
add(X1,X2,X3) :- X1 = o, X2 = Y1, X3 = Y1.
add(X1,X2,X3) :- X1 = s(Y1), X2 = Y2, X3 = s(Y3), X4 = Y1, X5 = Y3,
                 add(X4, X2, X5).
```

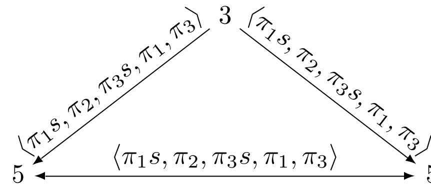
Note that we have two *kinds* of variables, the ones starting by X which may only appear as arguments to predicates and the Y variables, which represent the "real" variables used inside the predicate. Externally, `add` only uses three X variables, but internally it needs two more. In our relational translation, we will capture this fact by using a relation $I_{35} : 3 \to 5$ that takes care of creating $X4$ and $X5$. Recall that $\langle f, g\rangle$ is the categorical product constructor. Then, storing all our X variables in such a product, we may try to express `add` in a relational pseudo-notation:

$$\begin{aligned} \overline{add} &= \langle o, Y1, Y1\rangle \\ &\cup \; I_{35}; (\langle s(Y1), Y2, s(Y3), Y1, Y3\rangle \cap (id_2 \times \overline{add})); I_{35}^{\circ} \end{aligned}$$

the recursive call to $\overline{add}$ is wrapped into a vector of size 5, but we are calling it with the wrong parameters! The above expression is equivalent to $add(X3, X4, X5)$. We need to call it with the right parameters, so we compose the call with a permutation of the vector. We replace Y variables by categorical projections and the actual translation is:

$$\begin{aligned} \overline{add} &= \langle o, \pi_1, \pi_1\rangle^{\circ}; \langle o, \pi_1, \pi_1\rangle \\ &\cup \; I_{35}; \langle \pi_1 s, \pi_2, \pi_3 s, \pi_1, \pi_3\rangle^{\circ}; \langle \pi_1 s, \pi_2, \pi_3 s, \pi_1, \pi_3\rangle; W; (id_2 \times \overline{add}); W^{\circ}; I_{35}^{\circ} \end{aligned}$$

where $I_{35} : 3 \to 5 = \langle \pi_1, \pi_2, \pi_3\rangle^{\circ}$ and $W : 5 \to 5 = \langle \pi_1, \pi_3, \pi_4, \pi_2, \pi_5\rangle$. In order to save space we will abuse notation and will write f for a coreflexive relation $f^{\circ}; f$. With this abuse in mind, the tabulation of $\langle \pi_1 s, \pi_2, \pi_3 s, \pi_1, \pi_3\rangle$ is:



The reader can see how domain of the tabulations reflects the number of free variables in use by the machine, information which is usually associated to global storage. The codomain of the tabulations — the actual domain of the relations — should be interpreted as the number or working "temporal registers" that are used for parameter passing and unification.

A.2 Execution

A query $add(s(X), Y, Z)$ is translated to $\langle \pi_1 s, \pi_2, \pi_3\rangle; \overline{add}$ and its execution trace is:

$$
\begin{array}{ll}
\langle \pi_1 s, \pi_2, \pi_3\rangle; \overline{add} & \Rightarrow \\
(\langle \pi_1 s, \pi_2, \pi_3\rangle; \langle o, \pi_1, \pi_1\rangle) \cup \ldots & \Rightarrow \\
\mathbf{0} \cup \langle \pi_1 s, \pi_2, \pi_3\rangle; I_{35}; \langle \pi_1 s, \pi_2, \pi_3 s, \pi_1, \pi_3\rangle; W; (id_2 \times \overline{add}); W^\circ; I_{35}^\circ & \Rightarrow \\
\langle \pi_1 s, \pi_2, \pi_3\rangle; I_{35}; \langle \pi_1 s, \pi_2, \pi_3 s, \pi_1, \pi_3\rangle; W; (id_2 \times \overline{add}); W^\circ; I_{35}^\circ & \Rightarrow \\
(\langle \pi_1 s, \pi_2, \pi_3\rangle \mid \langle \pi_1 s, \pi_2, \pi_3, \pi_4, \pi_5\rangle); \langle \pi_1 s, \pi_2, \pi_3 s, \pi_1, \pi_3\rangle; W; (id_2 \times \overline{add}); W^\circ; I_{35}^\circ & \Rightarrow \\
(\langle \pi_1 s, \pi_2, \pi_3 s\rangle \mid \langle \pi_1 s, \pi_2, \pi_3 s, \pi_1, \pi_3\rangle); W; (id_2 \times \overline{add}); W^\circ; I_{35}^\circ & \Rightarrow \\
(\langle \pi_1 s, \pi_2, \pi_3 s\rangle \mid \langle \pi_1 s, \pi_3 s, \pi_1, \pi_2, \pi_3\rangle); (id_2 \times \overline{add}); W^\circ; I_{35}^\circ & \Rightarrow \\
(\langle \pi_1 s, \pi_2, \pi_3 s\rangle \mid \langle \pi_1 s, \pi_3 s, [\langle \pi_1, \pi_2, \pi_3\rangle; \langle o, \pi_1, \pi_1\rangle]\rangle; W^\circ; I_{35}^\circ \cup \ldots & \Rightarrow \\
(\langle os, \pi_1, \pi_1 s\rangle \mid \langle os, \pi_1 s, [\langle o, \pi_1, \pi_1\rangle]\rangle; W^\circ; I_{35}^\circ \cup \ldots & \Rightarrow \\
(\langle os, \pi_1, \pi_1 s\rangle \mid \langle os, \pi_1 s, o, \pi_1, \pi_1\rangle); W^\circ; I_{35}^\circ \cup \ldots & \Rightarrow \\
(\langle os, \pi_1, \pi_1 s\rangle \mid \langle os, \pi_1, \pi_1 s, o, \pi_1\rangle); I_{35}^\circ \cup \ldots & \Rightarrow \\
\langle os, \pi_1, \pi_1 s\rangle \cup \ldots &
\end{array}
$$

then $\langle os, \pi_1, \pi_1 s\rangle$ is translated back to the answer $X = o, Z = s(Y)$.

References

1. Bahar Aameri and Michael Winter. A first-order calculus for allegories. In Harrie de Swart, editor, *Relational and Algebraic Methods in Computer Science*, volume 6663 of *Lecture Notes in Computer Science*, pages 74–91. Springer Berlin / Heidelberg, 2011. 10.1007/978-3-642-21070-9_8.
2. Gianluca Amato and James Lipton. Indexed categories and bottom-up semantics of logic programs. In Robert Nieuwenhuis and Andrei Voronkov, editors, *LPAR*, volume 2250 of *Lecture Notes in Computer Science*, pages 438–454. Springer, 2001.
3. Gianluca Amato, James Lipton, and Robert McGrail. On the algebraic structure of declarative programming languages. *Theoretical Computer Science*, 410(46):4626 – 4671, 2009. Abstract Interpretation and Logic Programming: In honor of professor Giorgio Levi.
4. Andrea Asperti and Simone Martini. Projections instead of variables: A category theoretic interpretation of logic programs. In *ICLP*, pages 337–352, 1989.
5. Michael Barr and Charles Wells. *Category theory for computing science (3. ed.)*. Sentre de REcherches Mathématiques, 1999.
6. Francis Borceux. *Handbook of Categorical Algebra 2. Categories and Structures*, volume 51 of *Encyclopedia of Mathematics and its Applications*. Cambridge University Press, 1994.
7. Bernd Braßel and Jan Christiansen. A relation algebraic semantics for a lazy functional logic language. In Rudolf Berghammer, Bernhard Möller, and Georg Struth, editors, *RelMiCS*, volume 4988 of *Lecture Notes in Computer Science*, pages 37–53. Springer, 2008.
8. Pascal Brisset and Olivier Ridoux. Continuations in lambda-prolog. In *ICLP*, pages 27–43, 1993.
9. Paul Broome and James Lipton. Combinatory logic programming: computing in relation calculi. In *ILPS '94: Proceedings of the 1994 International Symposium on Logic programming*, pages 269–285, Cambridge, MA, USA, 1994. MIT Press.
10. Roberto Bruni, Ugo Montanari, and Francesca Rossi. An interactive semantics of logic programming. *THEORY AND PRACTICE OF LOGIC PROGRAMMING*, 1(6):647–690, 2001.
11. Carsten Butz. Regular categories and regular logic. Technical Report LS-98-2, BRICS, October 1998.

12 Dave Cattrall and Colin Runciman. Widening the representation bottleneck: A functional implementation of relational programming.

13 Dave Cattrall and Colin Runciman. A relational programming system with inferred representations. In Maurice Bruynooghe and Martin Wirsing, editors, *Programming Language Implementation and Logic Programming*, volume 631 of *Lecture Notes in Computer Science*, pages 475–476. Springer Berlin / Heidelberg, 1992. 10.1007/3-540-55844-6_156.

14 D.M. Cattrall. *The Design and Implementation of a Relational Programming Systema.* PhD thesis, University of York, 1992.

15 Keith L. Clark. Negation as failure. In Gallaire and Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, 1977.

16 Andrea Corradini and Andrea Asperti. A categorial model for logic programs: Indexed monoidal categories. In J. W. de Bakker, Willem P. de Roever, and Grzegorz Rozenberg, editors, *REX Workshop*, volume 666 of *Lecture Notes in Computer Science*, pages 110–137. Springer, 1992.

17 Guy Cousineau, Pierre-Louis Curien, and Michel Mauny. The categorical abstract machine. *Sci. Comput. Program.*, 8(2):173–202, 1987.

18 Stephan Diehl, Pieter H. Hartel, and Peter Sestoft. Abstract machines for programming language implementation. *Future Generation Comp. Syst.*, 16(7):739–751, 2000.

19 B. Dwyer. Programming using binary relations: a proposed programming languague. Technical report, University of Adelaide, 1994.

20 Stacy E. Finkelstein, Peter J. Freyd, and James Lipton. A new framework for declarative programming. *Theor. Comput. Sci.*, 300(1-3):91–160, 2003.

21 P.J. Freyd and A. Scedrov. *Categories, Allegories.* North Holland Publishing Company, 1991.

22 Emilio Jesús Gallego Arias, James Lipton, and Julio Mariño. Constraint logic programming with a relational machine. Technical report, Universidad Politécnica de Madrid, 2012. `http://babel.ls.fi.upm.es/~egallego/iclp/clprm.pdf`.

23 Emilio Jesús Gallego Arias, James Lipton, and Julio Mariño. Extensions to logic programming in tabular allegories: Algebraic data types, functions, constraints and monads. 2012. Submitted to ICLP 2012 `http://babel.ls.fi.upm.es/~egallego/iclp/lpta-ext.pdf`.

24 Emilio Jesús Gallego Arias, James Lipton, Julio Mariño, and Pablo Nogueira. First-order unification using variable-free relational algebra. *Logic Journal of IGPL*, 19(6):790–820, 2011.

25 Patrick A. V. Hall. Relational algebras, logic, and functional programming. In Beatrice Yormark, editor, *SIGMOD Conference*, pages 326–333. ACM Press, 1984.

26 Martin Hyland and John Power. The category theoretic understanding of universal algebra: Lawvere theories and monads. *Electr. Notes Theor. Comput. Sci.*, 172:437–458, 2007.

27 Peter T. Johnstone. *Sketches of an Elephant: A Topos Theory Compendium.* Oxford University Press, 2003.

28 Wolfram Kahl. Dependently-typed formalisation of relation-algebraic abstractions. In Harrie de Swart, editor, *Relational and Algebraic Methods in Computer Science*, volume 6663 of *Lecture Notes in Computer Science*, pages 230–247. Springer Berlin / Heidelberg, 2011. 10.1007/978-3-642-21070-9_18.

29 Yoshiki Kinoshita and A. John Power. A fibrational semantics for logic programs. In Roy Dyckhoff, Heinrich Herre, and Peter Schroeder-Heister, editors, *ELP*, volume 1050 of *Lecture Notes in Computer Science*, pages 177–191. Springer, 1996.

30 J. Lambek and P. J. Scott. *Introduction to Higher Order Categorical Logic.* Cambridge University Press, Cambridge, 1986.

31 F. William Lawvere. *Functorial Semantics of Algebraic Theories and Some Algebraic Problems in the context of Functorial Semantics of Algebraic Theories.* PhD thesis, Columbia University, 1968.

32 Jim Lipton and Emily Chapman. Some notes on logic programming with a relational machine. In Ali Jaoua, Peter Kempf, and Gunther Schmidt, editors, *Using Relational Methods in Computer Science*, Technical Report Nr. 1998-03, pages 1–34. Fakultät für Informatik, Universität der Bundeswehr München, July 1998.

33 J. W. Lloyd. *Foundations of logic programming.* Springer-Verlag New York, Inc., New York, NY, USA, 1984.

34 Richard McPhee and Richard Mcphee. Towards a relational programming language, 1995.

35 José F. Morales, Manuel Carro, Germán Puebla, and Manuel V. Hermenegildo. A generator of efficient abstract machine implementations and its application to emulator minimization. In Maurizio Gabbrielli and Gopal Gupta, editors, *ICLP*, volume 3668 of *Lecture Notes in Computer Science*, pages 21–36. Springer, 2005.

36 Maciej Pirog and Jeremy Gibbons. A functional derivation of the warren abstract machine. 2011. Submitted for publication.

37 Vaughan R. Pratt. Origins of the calculus of binary relations. In *Logic in Computer Science*, pages 248–254, 1992.

38 David M. Russinoff. A verified prolog compiler for the warren abstract machine. *Journal of Logic Programming*, 13:367–412, 1992.

39 Silvija Seres, J. Michael Spivey, and C. A. R. Hoare. Algebra of logic programming. In *ICLP*, pages 184–199, 1999.

40 Alfred Tarski and Steven Givant. *A Formalization of Set Theory Without Variables*, volume 41 of *Colloquium Publications.* American Mathematical Society, Providence, Rhode Island, 1987.

41 Eneia Todoran and Nikolaos S. Papaspyrou. Continuations for parallel logic programming. In *Proceedings of the 2nd ACM SIGPLAN international conference on Principles and practice of declarative programming*, PPDP '00, pages 257–267, New York, NY, USA, 2000. ACM.

42 Maarten H. van Emden. Compositional semantics for the procedural interpretation of logic. In Sandro Etalle and Miroslaw Truszczynski, editors, *ICLP*, volume 4079 of *Lecture Notes in Computer Science*, pages 315–329. Springer, 2006.

Tabling for infinite probability computation

Taisuke Sato[1] and Philipp Meyer[2]

1 **Tokyo Institute of Technology**
2-12-1 Ookayama, Meguro, Tokyo, Japan
sato@mi.cs.titech.ac.jp

2 **Technische Universität München**
Arcisstrasse 21, 80333 München, Germany
meyerphi@in.tum.de

Abstract

Tabling in logic programming has been used to eliminate redundant computation and also to stop infinite loop. In this paper we add the third usage of tabling, i.e. to make infinite computation possible for probabilistic logic programs. Using PRISM, a logic-based probabilistic modeling language with a tabling mechanism, we generalize prefix probability computation for PCFGs to probabilistic logic programs. Given a top-goal, we search for all SLD proofs by tabled search regardless of whether they contain loop or not. We then convert them to a set of linear probability equations and solve them by matrix operation. The solution gives us the probability of the top-goal, which, in nature, is an infinite sum of probabilities. Our generalized approach to prefix probability computation through tabling opens a way to logic-based probabilistic modeling of cyclic dependencies.

1998 ACM Subject Classification D.3.3 Language Constructs and Features

Keywords and phrases probability, tabling, PRISM

Digital Object Identifier 10.4230/LIPIcs.ICLP.2012.348

1 Introduction

Combining logic and probability in a logic programming language provides us with a powerful modeling tool for machine learning. The resulting language allows us to build complex yet comprehensible probabilistic models in a declarative way. PRISM [12, 13, 14] is one of the earliest attempts to develop such a language. It covers a large class of known models including BNs (Bayesian networks), HMMs (hidden Markov models) and PCFGs (probabilistic context free grammars) and computes probabilities with the same time complexity as their standard algorithms[1], as well as unexplored models such as probabilistic graph grammars [11].

The efficiency of probability computation in PRISM is attributed to tabling [16, 17, 10, 20, 19][2] that eliminates redundant computation. Given a top goal G, we search for all SLD proofs of G by tabled search and translating them to a set of propositional formulas with a graph structure called an *explanation graph* for G [13]. By applying dynamic programming to the explanation graph which is acyclic and partially ordered we can efficiently compute the probability of G in proportion to the size of the graph. The use of tabling also gives us another advantage over non-tabled computation: it stops infinite loop by detecting recurrence

[1] For example, the junction tree algorithm for BNs, the forward-backward algorithm for HMMs and the inside-outside algorithm for PCFGs.

[2] Tabling is also employed in other probabilistic logic programming languages such as ProbLog [5] and PITA [9].

© Taisuke Sato and Philipp Meyer;
licensed under Creative Commons License ND

Technical Communications of the 28th International Conference on Logic Programming (ICLP'12).
Editors: A. Dovier and V. Santos Costa; pp. 348–358

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

patterns of goals. Tabled logic programs thus allow us to directly use left recursive rules in CFGs without the need of converting them to right recursive ones.

In this paper we pursue yet the third advantage of tabling that has gone unnoticed in the non-probabilistic setting. We apply tabling to compute an infinite sum of probabilities that typically appears in the context of prefix probability computation in PCFGs. PCFGs (probabilistic context free grammars) are a probabilistic extension of CFGs in which CFG rules have probabilities and the probability of a sentence is computed as a sum-product of probabilities assigned to the rules used to derive the sentence [1, 4]. A prefix w is an initial substring of a sentence. Then the probability of the prefix w given by a PCFG is a sum of probabilities of infinitely many complete sentences of the form wv for some string v [3, 15, 7]. Prefix probabilities are useful in speech recognition as discussed in [3]. We generalize this prefix probability computation that originated in PCFGs to probability computation on *cyclic explanation graphs* which are generated by tabled search in PRISM. We emphasize that this approach, i.e. probability computation via cyclic explanation graphs, makes it possible to model probabilistic cyclic dependencies abundant in real life from economics to biological systems by probabilistic logic programs.

Technically the generation of cyclic explanation graphs is not difficult in PRISM. Just setting appropriately a certain PRISM flag that controls tabled search is enough. However computing probabilities from such graphs is difficult in general except for the case of *linear cyclic explanation graphs* that can be turned into a set of linear probability equations straightforwardly solvable by matrix operation. So the real problem is to guarantee the linearity of cyclic explanation graphs. We specifically examine a PRISM program for prefix probability computation in PCFGs and prove that the program always generates linear cyclic explanation graphs. We also prove that the probability equations obtained from the linear cyclic explanation graphs are solvable by matrix operation under some mild assumptions on PCFGs.

2 Probability computation in PRISM

2.1 Tabling and explanation graphs

PRISM is a probabilistic extension of Prolog with built-in predicates for machine learning tasks such as parameter learning and Bayesian inference [13, 14]. Theoretically a PRISM program DB is a union of a set of definite clauses and a set of probabilistic atoms of the form `msw(`id`,`v`)` that simulate dice throwing[3]. It defines a probability measure (an infinite joint distribution) $P_{DB}(\cdot)$ over possible Herbrand interpretations from which the probability of an arbitrary closed formula is calculated. Practically however PRISM programs are just Prolog programs that use `msw` atoms as probabilistic primitives. `msw` atoms are introduced by special declarations `values/3` specifying their properties as shown in the program DB_0 in Figure 1.

In PRISM, probabilities of ground atoms defined by a program DB are computed indirectly in two steps. In the first step, for a top-goal G of which we wish to compute the probability, we logically reduce it through DB by a top-down proof procedure, SLD search, to an equivalent propositional DNF formula $E_1 \vee \ldots \vee E_k$ such that $comp(DB) \vdash G \Leftrightarrow E_1 \vee \ldots \vee E_k$. Here each E_i $(1 \leq i \leq k)$, an *explanation for* G, corresponds to an SLD proof of G. It is a conjunction of ground `msw` atoms that records probabilistic choices made in the construction of the SLD

[3] `msw(`id`,`v`)` reads that throwing a dice named i yields an outcome v.

proof[4]. In the second step, using the fact that G and $E_1 \vee \ldots \vee E_k$ denote an identical random variable in terms of the distribution semantics for PRISM [13], we compute the probability of G as $P_{DB}(G) = P_{DB}(E_1 \vee \ldots \vee E_k)$ where $P_{DB}(\cdot)$ is the probability measure defined by *DB*.

In general there are exponentially many SLD proofs and so are explanations which result in an exponential size DNF. Nonetheless by introducing a tabling mechanism in the exhaustive SLD search process, we can often produce an equivalent but much smaller boolean formula by factoring out common sub-conjunctions in explanations as *intermediate goal*s [13, 20]. The resulting boolean formula is expressed as a conjunctive set of *defining formula*s that take the form $H \Leftrightarrow B_1 \vee \ldots \vee B_h$. Here H is the top-goal G or an intermediate goal. Hereafter the top-goal and intermediate goals are collectively called *defined goal*s. Each B_i $(1 \leq i \leq h)$ is a conjunction $C_1 \wedge \ldots \wedge C_m \wedge \mathtt{msw}_1 \wedge \ldots \wedge \mathtt{msw}_n$ $(0 \leq m, n)$ of defined goals $\{C_1, \ldots, C_m\}$ and `msw` atoms $\{\mathtt{msw}_1, \ldots, \mathtt{msw}_n\}$. We say that H is a *parent* of C_j $(1 \leq j \leq m)$. We call the closure of this parent-child relation the *ancestor relation* over ground atoms in *DB*. The whole set of defining formulas, denoted by $Exp(G)$, is called the *explanation graph* for G.

2.2 From explanation graphs to probability computation

The probability $P_{DB}(G)$ of a given goal G is precisely defined in terms of the distribution semantics for PRISM. But the problem is that the semantics is so abstractly defined and we cannot know the actual value of $P_{DB}(G)$ easily. Here we describe how to compute it under some assumptions.

To compute $P_{DB}(G)$, we convert each defining formula $H \Leftrightarrow B_1 \vee \ldots \vee B_h$ in $Exp(G)$ to *a set of probability equations for H*:

$$\begin{aligned} P(H) &= P(B_1) + \cdots + P(B_h) \\ &\quad \text{where} \\ P(B_i) &= P(C_1) \cdots P(C_m) P(\mathtt{msw}_1) \cdots P(\mathtt{msw}_n) \quad (1 \leq i \leq h) \\ &\quad \text{for } B_i = C_1 \wedge \ldots \wedge C_m \wedge \mathtt{msw}_1 \wedge \ldots \wedge \mathtt{msw}_n. \end{aligned}$$

We denote by $Eq(G)$ the entire set of probability equations thus obtained. Note that the conversion assumes exclusiveness among disjuncts $\{B_1, \ldots, B_k\}$ and independence among conjuncts $\{C_1, \ldots, C_m, \mathtt{msw}_1, \ldots, \mathtt{msw}_n\}$[5]. We consider $P(H)$s in $Eq(G)$ as numerical variables representing unknown probabilities and refer to them as *P-variable*s. What is important about $Eq(G)$ is that $Eq(G)$ *always has a solution $P(H) = P_{DB}(H)$ for every defined goal H*[13]. So if $Eq(G)$ has a unique solution for $P(G)$, it coincides with $P_{DB}(G)$.

When defined goals appearing in $Exp(G)$ are hierarchically ordered by the parent-child relation (with G as top-most element) as is usually the case, the P-variables in $Eq(G)$ are also hierarchically ordered so that $Eq(G)$ is uniquely and efficiently solved by dynamic programming using the generalized IO algorithm [13] in time linear in the size of $Eq(G)$. There are cases however in which $Exp(G)$ is not hierarchically ordered and some defined goals are their own ancestors. We say $Exp(G)$ is *cyclic* if there is a defined goal having itself as an ancestor in $Exp(G)$. If $Exp(G)$ is cyclic, $Eq(G)$ is also cyclic, and hence it is impossible to apply dynamic programming to $Eq(G)$, or even worse $Eq(G)$ may not have a unique solution.

---

[4] *comp*(*DB*) is the completion of *DB*. It is a union of if-and-only-if form of *DB* and so called Clark's equational theory.

[5] We assume in this paper that these conditions are always satisfied.

```
values(s,[[s,s],[a],[b]],set@[0.4,0.3,0.3]).
pre_pcfg(L):- pre_pcfg([s],L,[]).  % L is a ground list

pre_pcfg([A|R],L0,L2):-           % L0 ground, L2 variable when called
   ( values(A,_)-> msw(A,RHS),             % if A is a nonterminal
       pre_pcfg(RHS,L0,L1) ;  L0=[A|L1] ), % rule A->RHS selected
   ( L1=[] -> L2=[] ; pre_pcfg(R,L1,L2) ).
pre_pcfg([],L1,L1).
```

Figure 1 Prefix parser DB_0 for PCFGs.

In the next section, using a concrete example, we have a close look at cyclic $Eq(G)$s and investigate their properties.

3 Prefix computation for PCFGs using PRISM

In this section, we formulate prefix computation for PCFGs using PRISM.

3.1 A prefix parser

Before proceeding we introduce some terminology about CFGs for later use. Let X be a nonterminal in a CFG, α, β a mixed sequence of terminals and nonterminals. A rule for X is a production rule of the form $X \Rightarrow \alpha$. If there is a rule of the form $X \rightarrow Y\beta$, we say X and Y are in the direct left-corner relation. The *transitive closure* of the direct left-corner relation is called the *left-corner relation* and we write as $X \rightarrow_L Y$ if X and Y are in the left-corner relation. The left-corner relation is cyclic if $X \rightarrow_L X$ holds for some nonterminal X. We say that a rule is *useless* if it does not occur in any sentence derivation. A nonterminal is *useless* if every rule for it is useless. Otherwise it is *useful*. In this paper we assume that CFGs have "`s`" as a start symbol and have no epsilon rule and no useless nonterminal.

In addition let $\theta_1 : X \rightarrow \alpha_1, \ldots, \theta_n : X \rightarrow \alpha_n$ be the set of rules for X in a PCFG with *selection probabilities* $\theta_1, \ldots, \theta_n$ where $\sum_{i=1}^{n} \theta_i = 1$. We assume that every rule has a *positive* selection probability. If the sum of probabilities of sentences derived from the start symbol is unity, the PCFG is said to be *consistent* [18]. We also assume that PCFGs are consistent.

Now we here look at a concrete example of prefix probability computation based on cyclic explanation graphs. Consider a PCFG, $\mathbf{PG_0} = \{0.4 : \mathtt{s} \rightarrow \mathtt{s\,s}, 0.3 : \mathtt{s} \rightarrow \mathtt{a}, 0.3 : \mathtt{s} \rightarrow \mathtt{b}\}$. Here "`s`" is a start symbol and $0.4 : \mathtt{s} \rightarrow \mathtt{s\,s}$ says that the rule $\mathtt{s} \rightarrow \mathtt{s\,s}$ is selected with probability 0.4 when "`s`" is expanded in a sentence derivation.

A PRISM program DB_0 in Figure 1 is a prefix parser for $\mathbf{PG_0}$. It is a slight modification of a standard top-down CFG parser and parses prefixes acceptable by $\mathbf{PG_0}$ such as "`aab`" (as list `[a,a,b]`). The difference from the usual CFG parser is that it immediately terminates successfully as soon as the input prefix is consumed even if there remain nonterminals to be processed.

`values(s,[[s,s],[a],[b]],set@[0.4,0.3,0.3])` in Figure 1 is a value declaration which encodes $\mathbf{PG_0}$. `pre_pcfg([A|R],L0,L2)` is read that `[A|R]`, a substring of α in some

```
pre_pcfg([a]) <=> pre_pcfg([s],[a],[])
pre_pcfg([s],[a],[]) <=>
   pre_pcfg([s,s],[a],[]) & msw(s,[s,s]) v pre_pcfg([a],[a],[]) & msw(s,[a])
pre_pcfg([s,s],[a],[]) <=>
   pre_pcfg([a],[a],[]) & msw(s,[a]) v pre_pcfg([s,s],[a],[]) & msw(s,[s,s])
pre_pcfg([a],[a],[])
```

Figure 2 Explanation graph for prefix "a".

rule $X \to \alpha$, spans a d-list `L0-L2` as a sublist of the input list $[w_1, \ldots, w_N]$[6]. We remark that DB_0 is general, applicable to any PCFG just by replacing `values/3` with appropriate value declarations.

When this program is run with PRISM-flag `error_on_cycle` set to "`off`" for a command `?-`G where $G =$ `pre_pcfg(`$[w_1, \ldots, w_N]$`)` and $[w_1, \ldots, w_N]$ ($w_i \in \{\mathtt{a}, \mathtt{b}\}$) is a list representing a prefix $w_1, \ldots, w_N$, the proof procedure, the SLD search, simulates the leftmost derivation of the sentence by recursively calling the second clause. As soon as $[w_1, \ldots, w_N]$ is derived, the search terminates with success while ignoring nonterminals in `R` that may be non-empty as if `R` were successfully expanded to the remaining sentence[7]. We call this type of success *pseudo success*. During the search, a call to `pre_pcfg/3` is always of the form `pre_pcfg(`v`,`$[w_i, \ldots, w_N]$`,L2)` where v is a substring of RHS of some production rule and $i \leq N$. On return of the call, the variable `L2` is instantiated either to $[w_j, \ldots, w_N]$ ($i < j \leq N$) or to `[]` in the case of pseudo success. Therefore there are only a finitely many number of calling and returning patterns of `prefix_pcfg/3` and hence, the tabled search for all proofs of the top-goal G always terminates.

After all proof search done, PRISM constructs an explanation graph for the top-goal G by scanning the answer table in the memory. One thing to be noticed is that goals calling themselves and thereby suspended by tabling are also recorded in the table in addition to goals that normally succeeded. When PRISM encounters such goals, it looks at the PRISM-flag `error_on_cycle` and if the value is "`off`", those goals are treated as succeeded and as a result a cyclic explanation graph is generated.

3.2 Computing prefix probabilities: an example

In this subsection, we see, using a small example, how prefix probabilities are computed from cyclic explanation graphs. Figure 2 is the explanation graph for `pre_pcfg([a])` obtained by executing a command `?- probf(pre_pcfg([a]))`[8] w.r.t. DB_0. As can be seen, there is a cyclic goal `pre_pcfg([s,s],[a],[])` that calls itself. We convert the cyclic explanation graph to the corresponding set of probability equations shown in Figure 3. Here we used abbreviations: $\theta_{\mathtt{s} \to \mathtt{ss}} = P(\mathtt{msw}(\mathtt{s}, [\mathtt{s}, \mathtt{s}]))$ and $\theta_{\mathtt{s} \to \mathtt{a}} = P(\mathtt{msw}(\mathtt{s}, [\mathtt{a}]))$.

Although we know that the set of probability equations in Figure 3 are made true if we assign the probabilities defined by the distribution semantics[13] to `X`, `Y`, `Z` and `W`, we do not know their actual values. To know their actual values, we need to compute them by solving

[6] In the following strings beginning with lower case letters are ground terms.
[7] This is justifiable as we assume that every nonterminal is useful.
[8] `probf/1` is a built-in predicate in PRISM and `probf(`G`)` displays the explanation graph of G.

$$\left\{\begin{array}{lcl} \mathtt{X} & = & \mathtt{Y} \\ \mathtt{Y} & = & \mathtt{Z}\cdot\theta_{\mathtt{s}\to\mathtt{ss}} + \mathtt{W}\cdot\theta_{\mathtt{s}\to\mathtt{a}} \\ \mathtt{Z} & = & \mathtt{W}\cdot\theta_{\mathtt{s}\to\mathtt{a}} + \mathtt{Z}\cdot\theta_{\mathtt{s}\to\mathtt{ss}} \\ \mathtt{W} & = & 1 \end{array}\right. \quad \text{where} \quad \left\{\begin{array}{lcl} \mathtt{X} & = & P(\mathtt{pre_pcfg}([\mathtt{a}])) \\ \mathtt{Y} & = & P(\mathtt{pre_pcfg}([\mathtt{s}],[\mathtt{a}],[])) \\ \mathtt{Z} & = & P(\mathtt{pre_pcfg}([\mathtt{s},\mathtt{s}],[\mathtt{a}],[])) \\ \mathtt{W} & = & P(\mathtt{pre_pcfg}([\mathtt{a}],[\mathtt{a}],[])) \ = \ 1 \end{array}\right.$$

Figure 3 Probability equations for prefix "a".

the equations. Fortunately, equations are linear in the P-variables X, Y, Z and W and easily solvable.

By substituting $\theta_{\mathtt{s}\to\mathtt{ss}} = 0.4$ and $\theta_{\mathtt{s}\to\mathtt{a}} = 0.3$[9] for the equations and solving them, we obtain $\mathtt{X} = \mathtt{Y} = 0.5$, $P(\mathtt{pre_pcfg}([\mathtt{s},\mathtt{s}],[\mathtt{a}],[])) = \mathtt{Z} = 0.5$ and $\mathtt{W} = 1$, respectively. Hence the prefix probability of "a" is 0.5. Note that this prefix probability is larger than the probability of "a" as a sentence which is 0.3. This is because the prefix probability of "a" is the sum of the probability of sentence "a" *and* the probabilities of infinitely many sentences extending "a".

By looking at the set of probability equations in Figure 3 more closely, we can understand the way our approach computes prefix probabilities in PCFGs. For example, consider $\mathtt{Z} = P(\mathtt{pre_pcfg}([\mathtt{s},\mathtt{s}],[\mathtt{a}],[]))$ and the equation $\mathtt{Z} = \mathtt{W}\cdot\theta_{\mathtt{s}\to\mathtt{a}} + \mathtt{Z}\cdot\theta_{\mathtt{s}\to\mathtt{ss}}$. We can expand the solution Z into an infinite series:

$$\mathtt{Z} \quad = \quad \frac{1}{1-\theta_{\mathtt{s}\to\mathtt{ss}}}\mathtt{W}\cdot\theta_{\mathtt{s}\to\mathtt{a}} \quad = \quad (1+\theta_{\mathtt{s}\to\mathtt{ss}}+\theta^2_{\mathtt{s}\to\mathtt{ss}}+\cdots)\mathtt{W}\cdot\theta_{\mathtt{s}\to\mathtt{a}}$$

It is easy to see that this series represents the probability of infinitely many leftmost derivations of prefix "a" from nonterminals "s s" by partitioning the derivations based on the number of applications of rule $\mathtt{s} \to \mathtt{s\,s}$, i.e. 1 for no application ($\mathtt{s\,s} \Rightarrow_{\mathtt{s}\to\mathtt{a}} \mathtt{a\,s}$), $\theta_{\mathtt{s}\to\mathtt{ss}}$ for once ($\mathtt{s\,s} \Rightarrow_{\mathtt{s}\to\mathtt{ss}} \mathtt{s\,s\,s} \Rightarrow_{\mathtt{s}\to\mathtt{a}} \mathtt{a\,s\,s}$) and so on[10].

3.3 Properties of explanation graphs generated by a prefix parser

Let **PG** be a PCFG and **PG'** its backbone CFG. Also let $DB_{\mathbf{PG}}$ be a prefix parser for **PG** obtained by replacing the values/3 declaration in DB_0 in Figure 1 with an appropriate set of values/3 declarations encoding **PG**. In this section, we first prove that a necessary and sufficient condition under which a prefix parser $DB_{\mathbf{PG}}$ generates cyclic explanation graphs. We then prove that $DB_{\mathbf{PG}}$ always generates a system of linear equations for prefix probabilities. Finally we prove that the linear system is solvable by matrix operation under our assumptions on PCFGs.

▶ **Theorem 1.** *Let G_ℓ = pre_pcfg(ℓ) be a goal for a prefix $\ell = [w_1,\ldots,w_N]$ in* **PG'** *and $Exp(G_\ell)$ an explanation graph for G_ℓ generated by $DB_{\mathbf{PG}}$. Suppose there is no useless nonterminal in* **PG'**. *Then there exists a cyclic explanation graph $Exp(G_\ell)$ if-and-only-if the left-corner relation of* **PG'** *is cyclic.*

[9] values(s,[[s,s],[a],[b]],set@[0.4,0.3,0.3]) in the program sets $\theta_{\mathtt{s}\to\mathtt{ss}} = P(\mathtt{msw}(\mathtt{s},[\mathtt{s},\mathtt{s}])) = 0.4$, $\theta_{\mathtt{s}\to\mathtt{a}} = P(\mathtt{msw}(\mathtt{s},[\mathtt{a}])) = 0.3$ and $\theta_{\mathtt{s}\to\mathtt{b}} = P(\mathtt{msw}(\mathtt{s},[\mathtt{b}])) = 0.3$ respectively.

[10] Recall that we assume that PCFGs are consistent. So the sum of probabilities of sentences derived from "s" is 1. Consequently for example we may ignore s in "a s" when computing the probability of prefix "a" derived from "a s".

Proof. Suppose $Exp(G_\ell)$ is cyclic. Then some defined goal `pre_pcfg([`a`|`β`],`ℓ_0`,`ℓ_2`)` with a nonterminal "a" must call itself as a descendant in $Exp(G_\ell)$ where ℓ_0 and ℓ_2 are sublists of ℓ. So an SLD derivation exists from `:-prefix_pcfg([`a`|`β`],`ℓ_0`,L2),K` to its descendant `:-prefix_pcfg([`a`|`β`],`ℓ_0`,L2'),K'` that contains no return of goals because the list ℓ_0 is preserved. Consequently there is a corresponding leftmost derivation $\mathtt{s} \stackrel{*}{\Rightarrow} a\delta \stackrel{*}{\Rightarrow} a\delta'$ by **PG'**, the backbone CFG of **PG**. So the left-corner relation is cyclic.

Conversely suppose the left-corner relation of **PG'** is cyclic. Then there is a nonterminal "a" such that $a \rightarrow_L a$. As there is no useless nonterminal by our assumption, there is a leftmost derivation starting from "`s`" such that $\mathtt{s} \stackrel{*}{\Rightarrow} \gamma a\delta \stackrel{*}{\Rightarrow} \gamma a\delta' \stackrel{*}{\Rightarrow} w_1 \dots w_N$ for some sentence $w_1, \dots, w_N$. In what follows, for simplicity we assume that γ is empty (but generalization is straightforward). Let $\ell_0 = w_1, \dots, w_j$ $(j \leq N)$ be a prefix derived from a whose partial parse tree has a as the root and no a occurs below the root a. Then it is easy to see that the tabled search for all SLD proofs of G_{ℓ_0} generates $Exp(G_{\ell_0})$ containing a goal `prefix_pcfg([`a`|`β`],`ℓ_0`,[])` which is an ancestor of itself. So $Exp(G_{\ell_0})$ is cyclic. ◀

Let $Exp(G_\ell)$ be an explanation graph for G_ℓ. We introduce an equivalence relation $A \equiv B$ over defined goals appearing in $Exp(G_\ell)$: $A \equiv B$ if-and-only-if A is an ancestor of B and vice versa. We partition the set of defined goals into equivalent classes $[A]_\equiv$. Each $[A]_\equiv$ is called an *SCC* (*strongly connected component*). We say that *a defining formula* $H \Leftrightarrow B_1 \vee \dots \vee B_h$ *is linear* if there is no $B_i = C_1 \wedge \dots \wedge C_m \wedge \mathtt{msw}_1 \wedge \dots \wedge \mathtt{msw}_n$ $(1 \leq i \leq h,\ 0 \leq m, n)$ such that two defined goals, C_j and C_k $(j \neq k)$, belong to the same SCC. Also we say $Exp(G_\ell)$ *is linear* if every defining formula in $Exp(G_\ell)$ is linear.

▶ **Lemma 2.** *No two defined goals in the body of a defining formula in $Exp(G_\ell)$ belong to the same SCC.*

Proof. Let $H \Leftrightarrow B_1 \vee \dots \vee B_h$ be a defining formula in $Exp(G_\ell)$. Suppose some B_i contains two defined goals belonging to the same SCC. Looking at DB_0 in Figure 1, we know that the only possibility is such that $H \Leftrightarrow B_1 \vee \dots \vee B_h$ is a ground instantiation of the first (compound) clause about `pre_pcfg/3`:

```
pre_pcfg([a|β],ℓ0,ℓ2):-
    msw(a,α),pre_pcfg(α,ℓ0,ℓ1),pre_pcfg(β,ℓ1,ℓ2)        (1)
```

and the two defined goals, `pre_pcfg(`α`,`ℓ_0`,`ℓ_1`)` and `pre_pcfg(`β`,`ℓ_1`,`ℓ_2`)`, are in the same SCC. In this case, since `pre_pcfg(`α`,`ℓ_0`,`ℓ_1`)` is a proved goal, ℓ_1 is shorter than ℓ_0. On the other hand since `pre_pcfg(`β`,`ℓ_1`,`ℓ_2`)` is an ancestor of `pre_pcfg(`α`,`ℓ_0`,`ℓ_1`)` in $Exp(G_\ell)$, ℓ_0 is identical to or a part of ℓ_1, and hence ℓ_0 is equal to or shorter than ℓ_1. Contradiction. Therefore there is no such defining formula. Hence $Exp(G_\ell)$ is linear. ◀

▶ **Theorem 3.** *Let $Exp(G_\ell)$ be an explanation graph for a prefix ℓ generated by $DB_{\mathbf{PG}}$. $Exp(G_\ell)$ is linear.*

Proof. Immediate from Lemma 2. ◀

We next introduce a partial ordering $[A]_\equiv \succ [B]_\equiv$ over SCCs by $[A]_\equiv \succ [B]_\equiv$ if-and-only-if A is an ancestor of B but not vice versa in $Exp(G_\ell)$. We then extend this partial ordering to a total ordering $[A]_\equiv > [B]_\equiv$ over SCCs. Likewise we partition P-variables by the equivalence relation: $P(A) \equiv P(B)$ if-and-only-if $[A]_\equiv = [B]_\equiv$. We denote by $[P(A)]_\equiv$ the equivalence

class of P-variables corresponding to $[A]_\equiv$. By construction $[P(A)]_\equiv$s are totally ordered isomorphically to SCCs: $[P(A)]_\equiv > [P(B)]_\equiv$ if-and-only-if $[A]_\equiv > [B]_\equiv$. In the following we treat SCCs and P-variables as isomorphically stratified by this total ordering. We use $Eq([P(A)]_\equiv)$ to stand for the union of sets of probability equations for defined goals in $[A]_\equiv$.

Notice that $Eq([P(A)]_\equiv)$ is a system of linear equations by Theorem 3 if we consider P-variables in the lower strata as constants. Hence $Eq(G_\ell)$ is solvable inductively from lower strata to upper strata.

Now we prove that $Eq([P(A)]_\equiv)$ is always solvable by matrix operation under our assumptions on PCFGs. Let "a" be a nonterminal in the backbone CFG **PG'** and A a defined goal in $Exp(G_\ell)$. Put $A =$ `pre_pcfg(`$[a|\beta]$`,`ℓ_0`,`ℓ_2`)`. Since A is a proved goal, A successfully calls some ground goals $B_j =$ `pre_pcfg(`α_j`,`ℓ_{0j}`,`ℓ_{1j}`)` shown in (1) where $a \to \alpha_j$ is a CFG rule in **PG'**. By repeating a similar proof for Lemma 2, we can prove that the third goal `pre_pcfg(`β`,`ℓ_1`,`ℓ_2`)` in the clause body in (1) does not belong to $[A]_\equiv$, the SCC containing A. Thus $[A]_\equiv >$ [`pre_pcfg(`β`,`ℓ_1`,`ℓ_2`)`]$_\equiv$. So only some of the B_js can possibly belong to $[A]_\equiv$ as far as A is concerned.

Let $P(A_1), \ldots, P(A_K)$ be an enumeration of P-variables in $[P(A)]_\equiv$. Introduce a column vector $X_A = (P(A_1), \ldots, P(A_K))^T$. It follows from what we have argued that we can write $Eq([P(A)]_\equiv)$ as a system of linear equations $X_A = MX_A + Y_A$ where M is a $K \times K$ non-negative matrix and Y_A is a non-negative vector whose component is a sum of P-variables in the lower strata multiplied by constants. M is irreducible because in $Exp(G_\ell)$, every goal in $[A]_\equiv$ directly or indirectly calls every goal in $[A]_\equiv$. Y_A is non-zero because some A_i must have a proof tree that only contains defined goals in the lower strata. For vectors U, V, we write $U > 0$ (resp. $U \geq 0$) if every component of U is positive (resp. non-negative) and $U \geq V$ if $U - V \geq 0$ where 0 is a zero vector.

▶ **Theorem 4.** *Let* **PG** *be a consistent PCFG such that there is no epsilon rule and every production rule has a positive selection probability. Also let* $DB_{\mathbf{PG}}$ *be a prefix parser for* **PG** *and* $Exp(G_\ell)$ *an explanation graph for a prefix* ℓ*. Suppose* $Eq([P(A)]_\equiv)$ *is a system of linear equations for a defined goal* A *in* $Exp(G_\ell)$*. Put* $[P(A)]_\equiv = \{P(A_i) \mid 1 \leq i \leq K\}$ *and write* $Eq([P(A)]_\equiv)$ *as* $X_A = MX_A + Y_A$ *where* $X_A = (P(A_1), \ldots, P(A_K))^T$*. It has a unique solution* $X_A = (I - M)^{-1}Y_A$*.*

Proof. We prove that $I - M$ has an inverse matrix. To prove it, we assume hereafter that P-variables in $[P(A)]_\equiv$ are assigned as their values probabilities defined by the distribution semantics and hence all equations in $Eq([P(A)]_\equiv)$ are true.

By applying $X_A = MX_A + Y_A$ k repeatedly to itself, we have $X_A = M^k X_A + (M^{k-1} + \cdots + I)Y_A$ for $k = 1, 2, \ldots$ Since M, X_A, and Y_A are non-negative, we have $X_A \geq M^k X_A$ and $X_A \geq (M^{k-1} + \cdots + I)Y_A$ for every k. On the other hand since $\{(M^{k-1} + \cdots + I)Y_A\}_k$ is a monotonically increasing sequence of non-negative vectors bounded by X_A, it converges and so does $\{M^k X_A\}_k$.

Let $\rho(M)$ be the spectral radius of M[11]. Suppose $\rho(M) > 1$. In general $\rho(M) \leq \parallel M^k \parallel_\infty^{\frac{1}{k}}$ holds for every k where $\parallel \cdot \parallel_\infty$ is the matrix norm induced from the ∞ vector norm. It follows from $\rho(M)^k \leq \parallel M^k \parallel_\infty$ that $\lim_{k\to\infty} \parallel M^k \parallel_\infty = +\infty$. Consequently since $X_A > 0$ holds

[11] $\rho(M)$ is the largest eigenvalue of M. As M is irreducible, the right eigen vector and the left eigen vector associated with $\rho(M)$ are both positive by the Perron-Frobenius theorem.

because every proved goal has a positive probability from our assumption, some element of $M^k X_A$ goes to $+\infty$, which contradicts the convergence of $\{M^k X_A\}_k$. So $\rho(M) \leq 1$.

Suppose now $\rho(M) = 1$. Then in this case, we note that $\left\{ \dfrac{M^{k-1} + \cdots + I}{k} \right\}_k$ converges to a positive matrix (proof omitted), and hence $(M^{k-1} + \cdots + I)Y_A = \left(\dfrac{M^{k-1} + \cdots + I}{k} \right) \cdot kY_A$ diverges as k goes to infinity, which contradicts again the convergence of $\{(M^{k-1} + \cdots + I)Y_A\}_k$. Therefore $\rho(M) < 1$. So $(I - M)^{-1}$ exists. ◀

Note that $X_A = (I - M)^{-1} Y_A = (I + M + M^2 + \cdots) Y_A$. By further analyzing the matrix M, we understand that multiplying M by Y_A for example corresponds to growing partial parse trees by one step application of production rules (reduce operation in bottom-up parsing). Hence $P(A_i)$, a component of X_A, is an infinite sum of probabilities and so is the probability of the top prefix goal $P(\texttt{pre_pcfg}(\ell))$.

Summing up, we compute prefix probabilities for a PCFG **PG** as follows. Let *DB* be a prefix parser for **PG** and $G = \texttt{pre_pcfg}(\ell)$ a goal for a prefix ℓ.

[Step 1]: From G and *DB*, construct an explanation graph *Exp*(G).
[Step 2]: Extract the set of probability equations *Eq*(G) from *Exp*(G).
[Step 3]: Solve *Eq*(G) inductively from lower strata by matrix operation and obtain $P_{DB}(G)$, the prefix probability of ℓ.

The above procedure is general and applicable to arbitrary (cyclic) linear explanation graphs, not restricted to those generated by a PCFG prefix parser. We computed prefix probabilities for PLCGs (probabilistic left-corner grammars) similarly to PCFGs, but we omit the detail due to space limitations.

4 Related work

Prefix probability computation is mostly studied about PCFGs [3, 15, 7]. Jelinek and Lafferty [3] proposed a CKY like algorithm for prefix probability computation in PCFGs in CNF (Chomsky normal form). Their algorithm does not perform parsing but instead uses a single matrix whose dimension is the number of nonterminals which is constructed from a given PCFG. It runs in $O(N^3)$ where N is the length of an input prefix. Stolcke [15] applied the Earley style parsing to compute prefix probabilities. His algorithm uses a matrix of "probabilistic reflexive, transitive left-corner relation" computed from a given PCFG, independently of input sentences similarly to [3]. Our approach differs from them in that it works for probabilistic logic programs and it deals with explanation graphs constructed for each input prefix. Nederhof and Satta [7] generalized prefix probability computation for PCFGs to infix probability computation for PCFGs. They also studied prefix probability computation for a variant of PCFGs [8]. Nederhof et al. proposed prefix probability computation for stochastic tree adjoining grammars [6].

Approximate computation of prefix probabilities is possible for example by the iterative deepening algorithm used in ProbLog[2], but it is out of the scope of this paper.

5 Conclusion

We have proposed an innovative use of tabling: infinite probability computation based on cyclic explanation graphs generated by tabled search in PRISM. Our approach generalizes

prefix probability computation in PCFGs and is applicable to probabilistic models described by PRISM programs in general as well as PCFGs. In particular it is applicable to non-PCFG probabilistic grammars such as PLCGs though we omitted the result of prefix computation for PLCGs due to space limitations. We are developing a tool that generates a (cyclic) explanation graph for a given goal and computes its probability by solving the system linear equations associated with it. We expect that our approach provides a declarative way of logic-based probabilistic modeling of cyclic dependencies.

References

1. J. K. Baker. Trainable grammars for speech recognition. In *Proceedings of Spring Conference of the Acoustical Society of America*, pages 547–550, 1979.
2. L. De Raedt, A. Kimmig, and H. Toivonen. ProbLog: A probabilistic Prolog and its application in link discovery. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI'07)*, pages 2468–2473, 2007.
3. F. Jelinek and J. Lafferty. Computation of the probability of initial substring generation by stochastic context-free grammars. *Computational Linguistics*, 17(3):315–323, 1991.
4. C. D. Manning and H. Schütze. *Foundations of Statistical Natural Language Processing*. The MIT Press, 1999.
5. T. Mantadelis and G. Janssens. Dedicated tabling for a probabilistic setting. In *Proceedings of the 26th International Conference on Logic Programming (ICLP'10) (Technical Communications)*, pages 124–133, 2010.
6. M. Nederhof, A. Anoop Sarkar, and G. Satta. Prefix probabilities from stochastic tree adjoining grammars. In *Proceedings of the 36th Annual Meeting of the Association for Computational Linguistics (ACL'98)*, pages 953–959, 1998.
7. M. Nederhof and G. Satta. Computation of infix probabilities for probabilistic context-free grammars. In *Proceedings of the 2011 Conference on Empirical Methods in Natural Language Processing (EMNLP'11)*, pages 1213–1221, 2011.
8. M. Nederhof and G. Satta. Prefix probability for probabilistic synchronous context-free grammars. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics (ACL'11)*, pages 460–469, 2011.
9. F. Riguzzi and T. Terrance Swift. The PITA system: Tabling and answer subsumption for reasoning under uncertainty. *Theory and Practice of Logic Programming (TPLP)*, 11(4-5):433–449, 2011.
10. R. Rocha, F.M.A. Silva, and V.S. Costa. On applying or-parallelism and tabling to logic programs. *Theory and Practice of Logic Programming (TPLP)*, 5(1-2):161–205, 2005.
11. T. Sato. A glimpse of symbolic-statistical modeling by PRISM. *Journal of Intelligent Information Systems*, 31(2):161–176, 2008.
12. T. Sato and Y. Kameya. PRISM: a language for symbolic-statistical modeling. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI'97)*, pages 1330–1335, 1997.
13. T. Sato and Y. Kameya. Parameter learning of logic programs for symbolic-statistical modeling. *Journal of Artificial Intelligence Research*, 15:391–454, 2001.
14. T. Sato and Y. Kameya. New Advances in Logid-Based Probabilistic Modeling by PRISM. In L. De Raedt, P. Frasconi, K. Kersting, and S. Muggleton, editors, *Probabilistic Inductive Logic Programming*, pages 118–155. LNAI 4911, Springer, 2008.
15. A. Stolcke. An efficient probabilistic context-free parsing algorithm that computes prefix probabilities. *Computational Linguistics*, 21(2):165–201, 1995.

16 H. Tamaki and T. Sato. OLD resolution with tabulation. In *Proceedings of the 3rd International Conference on Logic Programming (ICLP'86)*, volume 225 of *Lecture Notes in Computer Science*, pages 84–98. Springer, 1986.

17 D. S. Warren. Memoing for logic programs. *Communications of the ACM*, 35(3):93–111, 1992.

18 C. S. Wetherell. Probabilistic languages: a review and some open questions. *Computing Surveys*, 12(4):361–379, 1980.

19 N.-F. Zhou, Y. Kameya, and T. Sato. Mode-directed tabling for dynamic programming, machine learning, and constraint solving. In *Proceedings of the 22th International Conference on Tools with Artificial Intelligence (ICTAI-2010)*, 2010.

20 N.-F. Zhou, T. Sato, and Y.-D. Shen. Linear tabling strategies and optimization. *Theory and Practice of Logic Programming (TPLP)*, 8(1):81–109, 2008.

ASP at Work: An ASP Implementation of PhyloWS*

Tiep Le, Hieu Nguyen, Enrico Pontelli, and Tran Cao Son

Department of Computer Science
New Mexico State University
(tile,nhieu,epontell,tson)@cs.nmsu.edu

Abstract

This paper continues the exploration started in [3], aimed at demonstrating the use of logic programming technology to support a large scale deployment and analysis of phylogenetic data from biological studies. This application paper illustrates the use of ASP technology in implementing the PhyloWS web service API—a recently proposed and community-agreed standard API to enable uniform access and inter-operation among phylogenetic applications and repositories. To date, only very incomplete implementations of PhyloWS have been realized; this paper demonstrates how ASP provides an ideal technology to support a more comprehensive realization of PhyloWS on a repository of semantically-described phylogenetic studies. The paper also presents a challenge for the developers of ASP-solvers.

1998 ACM Subject Classification J.3 Life and Medical Sciences, I.2.3 Logic programming

Keywords and phrases Answer sets, phylogenetic inference, systems, applications

Digital Object Identifier 10.4230/LIPIcs.ICLP.2012.359

1 Introduction

Phylogenetic inference is the task of constructing a phylogenetic tree that accurately characterizes the evolutionary lineages among a set of given species or genes. Phylogenetic trees allow us to understand the lineages of various species and how various functions evolved, to inform multiple alignments, and to identify what is the most conserved or important in some class of sequences. As such, phylogenetic trees have gained a central role in modern biology. They have become fundamental tools for building new knowledge, thanks to their explanatory and comparative-based predictive capabilities. In [9], 20 uses of evolutionary trees are discussed. Phylogenies are also used with increased frequency in several fields, e.g., genomics [5] and ecology [22]. Indeed, an ambitious goal in system biology is the construction of the *Tree of Life*, a phylogeny representing the evolutionary history of all species [2].

The explosive growth of phylogenetic data and the central role of phylogenetic knowledge in system biology led to the development of a database of phylogenies, called TreeBASE (e.g., [14, 18], `www.treebase.org`). The database contains phylogenetic trees and data matrices, together with information about the relevant publication, taxa, morphological and sequence-based characters, and published analyses. The trees are stored as text field strings structured in the Newick format [8]. The database provides retrieval capabilities via web interface, allowing users to locate phylogenies and to obtain datasets for different studies. Users can also retrieve data via a web service interface API (`sourceforge.net/`

* This work was partially supported by NSF grant IIS-0812267.

© Tiep Le, Hieu Nguyen, Enrico Pontelli, and Tran Cao Son;
licensed under Creative Commons License ND
Technical Communications of the 28th International Conference on Logic Programming (ICLP'12).
Editors: A. Dovier and V. Santos Costa; pp. 359–369

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

`apps/mediawiki/treebase/index.php?title=API`). This interface can deliver data in several different formats, including Newick, NEXUS [12], JSON, NeXML [21].

The creation of TreeBASE is a significant step towards the goal of creating the Tree of Life. Yet, it has been recognized that the lack of interoperability and standards in data and services between tools for the inference of phylogenies prevent large-scale and integrative analyses. To address these shortcomings, several efforts have been made. One of such efforts led to the development of an *interoperation stack* (*EvoIO Stack*) for the encoding and exchange of evolutionary structures. EvoIO comprises of *(i)* an ontology for data description (*Comparative Data Analysis Ontology (CDAO)*) [17], *(ii)* an exchange format (*NeXML*) [21], and *(iii)* a web service interface (*PhyloWS*) [11].

The PhyloWS interface specification [11] identifies several classes of queries specifically tied to phylogenies. The interface is comprehensive and represents the most extensive collection of queries and transformations for biological phylogenies ever proposed—in particular, it largely subsumes previous attempts to characterize access to phylogenetic databases (e.g., the approach of [15], implemented in Prolog by [3]). The implementation of these queries on an RDF representation of phylogenies proved to be challenging; in particular, traditional languages for RDF (e.g., SPARQL) do not provide the power to perform the type of computations on phylogenies required by PhyloWS—e.g., they lack the expressive power to capture recursive computations and transitive closures (which are essential, e.g., to determine ancestors and lineage in a phylogeny).

In this paper, we propose a new and modular implementation of PhyloWS using answer set programming (ASP) [13, 16]. We present the encoding of PhyloWS and evaluate the performance of the implementation using the data extracted from TreeBASE which shows that ASP is sufficiently expressive for answering various types queries of specified in the PhyloWS specification. The experimental results show that ASP-solver is efficient but pre-processing is needed for such a data intensive application.

2 Background—PhyloWS: Phylogenetic Web Service API

PhyloWS [11] is a web-services standard for accessing phylogenetic trees, data matrices, and their associated metadata from online phylogenetic data. Together with NeXML and CDAO, PhyloWS is a part of the platform, called *EvoIO Stack* [19], that combines support for exchange of data and their semantics and predictable programmatic access. PhyloWS is proposed to address the lack of a web-service API that allows for the integration of phylogenetic data and tools into new services for large-scale analysis. To date, PhyloWS only exists as a specification. The implementation proposed in this paper is its first implementation[1]. PhyloWS contains specification for a variety of tasks necessary for the creation, maintaining, retrieving, and manipulating of phylogenetic data (e.g., trees, matrices). In this paper, we will focus on the services for retrieving phylogenetic data. Based on the specification of PhyloWS [11], queries can be grouped into the following four categories:

- **Node-oriented queries**: Queries of this type ask for nodes satisfying certain conditions, e.g., the most recent common ancestor of two (or more) terminal nodes in the specified tree; or the nodes which have the distance to the root greater than a given distance; or determine the relationships among nodes, e.g., obtain the patristic distance between two nodes `tn707506` and `tn444001`; etc.

[1] The implementation of PhyloWS at `sourceforge.net/apps/mediawiki/treebase/index.php?title=API` is tailored to TreeBASE and limited to simple retrievals.

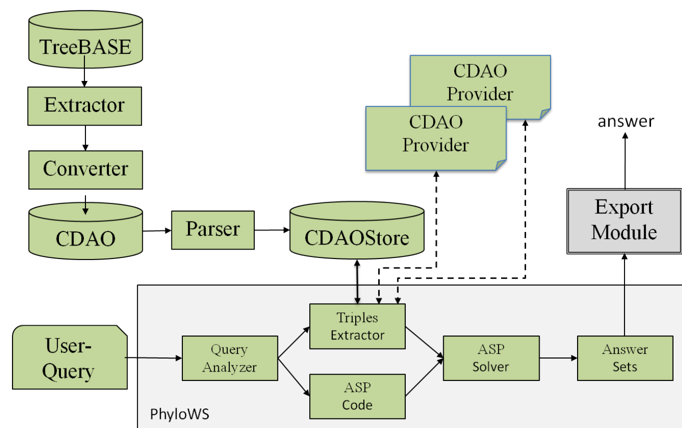


Figure 1 Overall Structure: System Implementation.

- **Clade-oriented queries**: Queries of this type are related to clades with some properties—e.g., determine the clades consisting of a species and all its descendants; or find the minimum spanning clade in a tree that contains the TUs (taxonomic units) *Ilexanomala* and *Ilex glabra*.
- **Tree-oriented queries**: These queries ask for trees satisfying some criteria, e.g., the trees created by *Knapp S.* no later than *2003-08-20*; or determine relationships among trees, e.g., the *Robinson-Foulds* distance between two trees.
- **Data-oriented queries**: These queries extract the metadata for different phylogenetic data, e.g., the *taxonVariant id* or *ncbi id* of a node; the creator of a tree; or the character matrix from all matrices containing a given set of OTUs.

3 System Organization

The overall implementation of PhyloWS is depicted in Figure 1. The top part shows the components of the system necessary to populate CDAOStore; CDAOStore [3] is triple store, build using CDAO, used for our experiments. First, data from current phylogenetic tree repositories (e.g., TreeBASE) is extracted into NeXML data files (*Extractor*) and converted to CDAO representation (*Converter*). This process is executed only once to populate the repository of phylogenetic data in CDAO representation. A standard XML-parser is used to generate triples from NeXML and import them into the CDAOStore.

The main contribution of this paper is the PhyloWS box. User queries are analyzed by a query analyzer that determines the actual ASP-code and the necessary data type from the CDAOStore repository. This information is passed on to the *Triples Extractor* module. The ASP program (facts and code) is sent to the ASP solver to compute its answer sets. The export module obtains the answer sets from the solver and generates answer for the user. Let us emphasize that the PhyloWS implementation in this paper is fully modular, and can be applied to any data source that can provide phylogenetic data as CDAO RDF triples.

Observe that in the construction of the CDAOStore, we were able to reuse the prototype described in [3]. The present system includes several improvements over the reported prototype—e.g., it is able to extract *all* studies from TreeBASE, the conversion to CDAO is more precise. The key differences between the current paper and the system described in [3] lie in (*i*) our focus here is on the full implementation of PhyloWS, instead of the simple querying covered in [3]; and (*ii*) the complete use of ASP technology in the implementation of the PhyloWS, instead of a mixture of Prolog and SPARQL.

4 PhyloWS in ASP

In this section, we will present the ASP implementation for the web services described in Section 2. For efficiency purpose, we develop a front-end that analyzes the queries and determines the type of phylogenetic data that needs to be extracted from the CDAO repository. Presently, the type of data corresponds to the type of queries. For example, for a node-oriented query, information about the trees (nodes and edges) will be extracted. The *Triples Extractor* module is responsible for extracting the necessary information from the CDAO representation and generating ASP facts for use to answer the query. Observe that this task can be combined and executed via ASP extensions such as `dlvhex` [4]. We will discuss the reason behind our design choice in the discussion part of the paper.

Representing CDAO in ASP. The information about phylogenies can be easily encoded as ASP facts. Following are some sample facts generated by the *Triples Extractor* module:

```
tree(t_id).                       % t_id is a tree
tree_label(t_id,lab).             % t_id has label "lab"
tree_is_defined_by(t_id,s_id).    % t_id is studied in s_id
tree_ntax(t_id,n_Taxa).           % t_id has n_Taxa taxa
edge(t_id,n1,n2).                 % t_id contains an edge  from n1 to n2
edge_length(t_id,n1,n2,l).        % l is the length of the edge (n1,n2) in t_id
represents_TU(t_id,n1,tu_id).     % node n1 of t-id represents tu_id
taxon_id(tu_id,taxon_id).         % tu_id represents taxon_id
matrix_type(m_id,m_type).         % matrix m_id is of the type "m_type"
belongs_to_TU(m_id,cell,tu_id).   % cell in matrix m_id belongs to tu_id
```

ASP Encoding of PhyloWS. The encoding of the PhyloWS in ASP starts with the definition of a set of rules that will be frequently used in several types of queries. The following code (syntax of `clingo` [10]) defines the predicates `node, leaf, root, parent`, and `ancestor` within a tree. It also defines the predicate `common_ancestor` of a set of taxa, identified by the predicate `set_of_taxa`, whose elements will be specified by `member/2`.

```
node(T,N):- edge(T,N,_).                      node(T,N):- edge(T,_,N).
leaf(T,N):- node(T,N),{edge(T,N,_)}0.         root(T,N):- node(T,N),{parent(T,_,N)}0.
parent(T,N1,N2):- tree(T), edge(T,N1,N2).
ancestor(T,N1,N2):- parent(T,N1,N2). ancestor(T,N1,N2):- ancestor(T,N1,Nb), parent(T,Nb,N2).
common_ancestor(T,N,S):- tree(T),node(T,N),set_of_taxa(S),{member(E,S):not ancestor(T,N,E)}0.
```

Most of the above rules are simple. The last rule states that node N in the tree T is a common ancestor of a set of taxa S if N is an ancestor of every member of S. We will next present the detailed encodings for the different types of queries.

Node-oriented Queries. We currently consider four frequently used node-oriented queries.

- *Query N1*: *compute the most recent common ancestor of two or more leaf nodes in a specified tree.* The input consists of a tree t and a set s of leaf nodes in t. The output should be the most recent common ancestor n of elements in s, denoted by $mrca(n,s)$, determined using the ASP rule:

  ```
  mrca(N, S):- tree(T), node(T,N), set_of_taxa(S), common_ancestor(T,N,S),
              {common_ancestor(T,Nb,S): ancestor(T,N,Nb)}0.
  ```

  The rule elegantly encodes that the most recent common ancestor of a set S is a common ancestor of S that does not have a descendant which is also a common ancestor of S.
- *Query N2*: *compute the patristic distance between two taxa in a given tree.* The patristic distance between taxa n_1 and n_2 of a tree t is defined as the sum of the distances from the most recent common ancestor to each node:

```
set_of_taxa(s). member(n1,s). member(n2, s).
distance_to_ancestor(T,N1,N2,L):- parent(T,N1,N2),edge_length(T,N1,N2,L).
distance_to_ancestor(T,N1,N2,D):- parent(T,Nb,N2),edge_length(T,Nb,N2,L),
        distance_to_ancestor(T,N1,Nb,L2), D=L+L2.
patristic_distance(T,N1,N2,D):- mrca(M, s), D=L1+L2,
        distance_to_ancestor(T,M,n1,L1), distance_to_ancestor(T,M,n2,L2).
```

The rules are simple thanks to the definition of the most recent common ancestor of a set in *Query N1*. Rules for computing the distance are standard.

- *Query N3*: *identify the set of* matching nodes *of a tree whose distance to the root is greater than a predefined distance.* Given a tree t (e.g., by identifier) and a distance c, output the matching nodes whose distance to the root is greater than c. This is implemented by the following rule:

```
matching_nodes(T,N):- root(T,R), distance_to_ancestor (T,R,N,L), L>=c.
```

- *Query N4*: *output the lineage of ancestors of a given node.* Given a tree t, a node n, the lineage of ancestors for n can be determine by the following rule, built using the facts $has_Ancestor(t, n, x)$ representing that x is an ancestor of n in the tree t.

```
lineage_node(t,n,Ancestor_node_id) :- has_Ancestor(t,n,Ancestor_node_id).
```

Clade-oriented Queries. Two typical clade-oriented queries are implemented.

- *Query C1*: *find the minimum spanning clade and the TUs of the clade for a set of taxa in a specified tree.* Given a set of taxa s, determine the minimum spanning clade of s and its TUs. Nodes belong to the minimum spanning clade are represented by the atoms of the form $minimum_clade(s, n)$. Atoms of the form $label(x, y)$ represent the label associated to the nodes in the clade. Since the answer is the tree whose root is the `mrca` n of s and all n's descendants, this can be implemented as follows.

```
minimum_clade(S,N):- tree(T), node(T,N), mrca(N, S).
minimum_clade(S,D):- tree(T), node(T,N), mrca(N, S), ancestor(T,N,D).
label(D,TU_Label):- tree(T), minimum_clade(_,D),
                    represents_TU(T,D,TU),tu_label(T,TU,TU_Label).
```

The first rule states that the given most recent common ancestor belongs to the minimum clade. The next rule obtains all of its descendants.

- *Query C2*: *find a clade in a tree whose taxa has a given character*, i.e., given a tree t and a character c, find a clade (or all) s of t s.t. every taxon in s has the character c.

```
clade(s).             {in_clade(s,N) : leaf(t,N)}.   member(N,s):- in_clade(s,N).
:- minimum_clade(s, N),not in_clade(s, N).
:- in_clade(s,N), represents_TU(T,N,TU),
        belongs_to_TU(M,Cell,TU), not belongs_to_Character(M,Cell,c).
```

The fact $clade(s)$ specifies the name s of the clade produced. The choice rule states that a leaf might or might not belong to the clade. The third rule defines the membership of the node in the set of taxa s that has been selected for use to determine the minimum clade (Query C1). The first constraint ensures that the elements of the clade are only those that are selected. The second removes clades that contain taxa that do not have the specified character.

Tree-oriented Queries. We consider eight types of tree-oriented queries.

- *Query T1*: *find trees matching a topology.* The topology can be given by (i) the range of the numbers of taxa count of the tree, i.e., between $n - c$ and $n + c$ for two constants n and c; (ii) the range of the width of the tree; etc.
  Most of the above queries can be straightforwardly encoded in ASP. For example, given a constant c and the tree $tr1386$, the following rule determines all trees with their taxa count in the range $[n - c, n + c]$ where n is the number of taxa of $tr1386$. The rule makes use of the predicate $tree_ntax(t, n)$ that represents the number of taxa of a tree.

```
matching_ntax(T,Cnt):- tree_ntax(tr1386,N),tree_ntax(T,Cnt),Cnt <= N+c,Cnt>=N-c.
```

- *Query T2*: *find trees whose length is shorter (or longer) than the length of a given tree (or a constant)* where the tree length is defined as the maximal distance from the root of the tree to its taxa (leaves). This type of queries can be answered with the definition of the tree length, implemented as follows.

```
distance_to_root(T,N,L):- root(T,R),leaf(T,N),distance_to_ancestor(T,R,N,L).
tree_length(T,L):- root(T,R), leaf(T,N), distance_to_root(T,N,L),
                   {distance_to_root(T,X,L1): L1>L}0.
```

- *Query T3*: *find trees with the shortest distance from the root to a given node n.* This can be easily implemented using the predicate *distance_to_root*.
- *Query T4*: *given a set s of OTUs (or taxa), find a tree containing this set.* We present the rules for identifying trees with a set of OTU, specified by the atom $otu_set(s)$ and the membership atoms $member(x, s)$.

```
connect_tu(TU,S,T):- tree(T),otu_set(S),member(TU,S),represents_TU(T,_,TU).
tree_otus(T,S):- tree(T),otu_set(S),{member(TU,S):not connect_tu(TU,S,T)}0.
```

- *Query T5*: *find trees based on tree metadata.* For example, trees that were *(i)* created by some author; *(ii)* created before a given date; *(iii)* built with a given type of data; etc. Since the data included in each study contains information such as *has_creator*, *has_creationDate*, *matrix_type*, etc. this type of queries can be easily encoded in ASP. Again, we omit the detail ASP rules for brevity.
- *Query T6*: *identify trees by the parsimony tree length* which is defined by the total number of characters of its taxa. This can be implemented as follows.

```
parsimony_length(T,L):- tree(T),
        L =  #count {belongs_to_Character(_,Cell_id,Character):
              belongs_to_TU(_,Cell_id,TU_id):represents_TU(T,_,TU_id)}.
```

- *Query T7*: *determine trees with size greater (or smaller) than a given constant c, or a certain ratio r of internal to external nodes.* Again, using the aggregate function *#count*, this type of query can be implemented as follows[2].

```
matching_tree_size(T,S):- tree(T), S = #count {node(T,_)}, S>=c.
internal_node(T,N):- node(T,N), not leaf(T,N).
matching_tree_ratio(T):- tree(T),R=R1/R2, R>=r,
        R1 = #count {internal_node(T,_)}, R2 = #count {leaf(T,_)}.
```

---

[2] The code assumes that the ration is an integer. Using the scripting feature available for `clingo`, this assumption can be removed.

- *Query T8*: *computing the Robinson-Foulds distance [1] between two trees.* The Robinson-Foulds distance is frequently used to compare phylogenetic trees. It measures the number of clusters of descendant leaves that are not shared by the two trees. The Robinson-Foulds distance of two trees T_1 and T_2 can be computed using the following algorithm:
  - Compute the multi-set of clusters of each tree, where each cluster is a set of taxa (leaves) that are descendants of an internal node n. Let us denote the set of clusters of T_1 and T_2 by $C(T_1)$ and $C(T_2)$, respectively.
  - Compute D_1 (resp. D_2), the number of clusters which belong $C(T_1) \setminus C(T_2)$ (resp. $C(T_2) \setminus C(T_1)$).

  The Robinson-Foulds distance is then defined by $(D_1 + D_2)/2$. Given two trees T_1 and T_2, we define the predicate $rf_distance(T_1, T_2, D_1, D_2)$ that encodes the Robinson-Foulds distance. This can be implemented using the following set of ASP rules.

```
in_cluster(T,N,L):- internal(T,N), leaf(T,L), ancestor(T,N,L).
neq_cluster(X,Y):- internal(T1,X),internal(T2,Y),T1!=T2,in_cluster(T1,X,L),
                                                    not in_cluster(T2,Y,L).
neq_cluster(X,Y):- internal(T1,X),internal(T2,Y),T1!=T2,not in_cluster(T1,X,L),
                                                    in_cluster(T2,Y,L).
eq_cluster(X,Y,T1,T2):- internal(T1,X),internal(T2,Y),T1!=T2,not neq_cluster(X,Y).
{matched(X,Y,T1,T2) : eq_cluster(X,Y,T1,T2)}.
2{used(T1,X), used(T2,Y)}:- matched(X,Y,T1,T2).
matched(X,Y,T1,T2):- matched(Y,X,T2,T1).
:-matched(X,Y,T1,T2),matched(X,Z,T1,T2),Y!=Z.
:-matched(Y,X,T1,T2),matched(Z,X,T1,T2),Y!=Z.
:-eq_cluster(X,Y,T1,T2), not used(T1,X), not used(T2,Y).
not_matched(T,N):- internal(T,N),not used(T,N).
rf_distance(T1,T2,D1,D2):- tree(T1), tree(T2), T1!=T2,
        D1 = #count {not_matched(T1,N)}, D2 = #count{not_matched(T2,N)}.
```

  The clusters are named by the internal nodes. The first rule defines the elements of a cluster. Next two rules state that two clusters from different trees are different when their sets of taxa are different. The third rule defines when two clusters are identical. The choice rule defines the predicate $matched(X, Y, T_1, T_2)$ among identical clusters of the trees. The next two rules define the predicates $matched(X, Y, T_1, T_2)$ and $used(X, T_1)$ ($used(Y, T_2)$) which say that the cluster X of tree T_1 is identical to the cluster Y of tree T_2 and will not be counted towards D_1 and D_2 respectively. The constraints ensure that each cluster is used to match with at most one cluster and the matching should be done as long as it is possible. $not_matched(T, N)$ indicates the cluster that is not matched with any cluster of another tree. The last rule encodes the Robinson-Foulds distance.

Data-oriented Queries. We consider four types of data-oriented queries.

- *Query D1*: *list metadata for a given taxon n or a given tree t.* Such information can be obtained from the facts associated to the tree. Some of the rules are:

```
metadata_belongs_to(n,T,Study_id):- node(T,n), tree_is_defined_by(T,Study_id).
metadata_represents(n,TU_id,TU_label,Taxon_id,TaxonVariant_id,Ncbi_id,Ubio_id):-
        represents_TU(T,n,TU_id), tu_label(T,TU_id,TU_label),
        taxon_id(TU_id,Taxon_id),taxonVariant_id(TU_id,TaxonVariant_id),
        ncbi_id(TU_id,Ncbi_id), ubio_id(TU_id,Ubio_id).
metadata_character(n,Character_id):-
        represents_TU(_,n,TU_id),belongs_to_TU(_,Cell_id,Tu_id),
        belongs_to_Character(_,Cell_id,Character_id).
```

- *Query D2*: *identify all matrices containing a given OTU (tu_id); or determine all characters in a matrix (m_id) that have data for an OTU.* This query is encoded as follows.

```
matrices_with_otu(M,tu_id):- has_TU(M,tu_id).
character_has_otu(C,m_id,tu_id):- belongs_to_TU(m_id,Cell,tu_id),
                                         belongs_to_Character(m_id,Cell,C).
```

- *Query D3*: *identify all OTUs in a matrix which have a given set of characters*:

```
has_character(Tu,C):- belongs_to_TU(M,Cell,Tu), belongs_to_Character(M,Cell,C).
obtain_otus_having_characters(Tu,S):- has_TU(M,Tu), set_characters(S),
        {member(C,S): not has_character(Tu,C)}0.
obtain_otus_having_characters_belonging_matrix(matrix_id,Tu,S):-
        has_TU(M,Tu), set_characters(S),{member(C,S): not has_character(Tu,C)}0.
```

- *Query D4*: *identify the character that appears in all matrices containing data for a given set of OTUs.* The ASP rules for this query are:

```
matching_matrices(M,S):- otu_set(S), has_TU(M,_),{member(E,S): not has_TU(M,E)}0.
has_character(M,C):- belongs_to_Character(M,_,C).
character_in_all_matrices(C):- matching_matrices(M,S), has_character(M,C),
        {matching_matrices(M1,S): not has_character(M1,C)}0.
```

The first rule identifies the matrix that contains all the given OTUs. The other rules search for the characters that appear in all those matrices.

5 Evaluation

We have successfully extracted all data from the TreeBASE and created various types of NeXML files for studies (2989 files, 2.55GB), matrices (5794 files, 1.96GB), and trees (8621 files, 500MB). So far, we have converted them into 9558 CDAO files and stored them in study (861), matrix (76), and tree (8621) files. The space requirement for CDAO study, matrix, and tree files is 18GB, 3410MB and 2214MB, respectively. We observe that the conversion is fairly time consuming and space demanding. However, most of the time is spent in the conversion of `Character State Data Matrix`, e.g., the program takes 3.25 hours to convert the `Character State Data Matrix` of the study `S715`, stored in a 3.36MB file, that has 44 OTUs and each OTU has 2721 characters. On the other hand, the conversion of the largest tree (identifier `Tr47158`), stored in a 3MB file, into CDAO took less than 5 minutes. From this data, we have built around 4GB fact data from study and matrix files and 172MB fact data from tree files and populated CDAOStore with both sets of data. The huge size of the data is the main reason for the design decisions discussed in the next section.

Table 1 contains sample results of the system for several queries discussed in the previous section. The experiment is conducted using 261 CDAO files. The ASP solver used in the experiment is `clingo` version 3.0.3. The machine used in the experiment uses Linux OS with a Genuine Intel(R) CPU T2400 @ 1.83GHz and 1015 MB. Because the *Triples Extractor* is fairly efficient (it takes usually less than 15 seconds to extract the necessary data from the CDAOStore), we only report the time (in ms.) used by the ASP solver.

6 Related Work and Discussion

Related Work. ASP has been used in the construction of phylogenetic network such as the evolutionary history of Indo-European languages [7]. The method was later applied to the analysis of parasite-host systems [6]. Our use of ASP in this paper is different in that we use ASP in the development of phylogenetic web services.

The present work is most closely related to our previous work [3]. As we have indicated in Section 3, the present work is much advanced comparing to the early work. In particular,

Table 1 Evaluation of queries.

Query	Data size	Execution time	Query	Data size	Execution time
$N1$	644.5 KB	2.360	$N2$	698.8 KB	2.840
$N3$	685.2 KB	1.060			
$C1$	1.5 MB	2.940	$C2$	31.9 MB	13.420
$T1$	698.8 KB	1.210	$T2$	698.8 KB	1.000
$T3$	698.8 KB	0.810	$T4$	1.2 MB	0.450
$T5$	65.6 KB	0.020	$T6$	31.9 MB	1913.820
$T7$	644.5 KB	0.830	$T8$	6.2 KB	0.820
$D1$	44.0 MB	15.270	$D2$	34.9 MB	10.310
$D3$	34.9 MB	14.020	$D4$	19.1 MB	5.940

the set of queries implemented in this paper—as indicated in the PhyloWS specification and agreed by the community—is broader and addresses the need of the community. Furthermore, the implementation described in this paper employs only ASP for the query evaluation.

Design Choices. ASP technologies have been extended to allow ASP programs interact with ontologies such as the system `dlvhex` [4]. As such, it is natural to ask the question of whether PhyloWS could be implemented using `dlvhex` and how would the system perform. To answer these questions, we have experimented with the web interface at the URL `http://asptut.gibbi.com/`. With a few changes in the syntax to conform with the `dlvhex` syntax, most queries can be executed with sample data. The difficulty arises when we attempt to run with the real data. As it turns out, converting everything into triples using `dlvhex` using the command: `triple(X,Y,Z):-&rdf[file_URI](X,Y,Z)` and then defining necessary predicates such as `has_TU, belongs_to_TU` using standard LP rules such as[3]

```
has_TU(X,Z) :- triple(X,"<http://___/cdao.owl#has_TU>",Z).
belongs_to_TU(X1,Y1,Z1) :-
    triple(X1,"<http://___/cdao.owl#has_Character>",Z2),
    triple(Y1,"<http://___/cdao.owl#belongs_to_Character>",Z2),
    triple(Y1,"<http://___/cdao.owl#belongs_to_TU>", Z1).
```

does not provide the desired efficiency. For example, our parser took 30 minutes to process the study `S261` (12 MB in CDAO representation); the web-interface does not return the result after 1.5 hours. This indicates that a straightforward application of `dlvhex` features to simplify the amount of programming will not yield an acceptable result. We are planning to further experiment with `dlvhex` without using the web-interface.

The huge size of the CDAO files and the lack of an efficient interface between ASP and ontologies led to the use of the parser (using JAVA and the Jena framework) to generate facts from CDAO and store them in the CDAOStore.

As noted, the current size of the CDAOStore is about 5GB. Intuitively, any query listed in Section 4 could have been processed using this data. However, `clingo` cannot deal with file of 70 MB. We observed this during our experiment: whenever the amount of data is more than 70MB, a *killed* message is displayed and the computation is aborted. The *Query Analyzer* and *Triple Extractor* modules are developed to deal with this issue.

Limitations and Challenges. The previous discussion details some limitations of the current system. While it would be interesting whether the use of `dlvhex` will help us to

[3] ___ stands for `www.evolutionaryontology.org/cdao/1.0`.

eliminate the intermediate steps of the *Query Analyzer* and `Triple Extractor` modules, the critical limitation lies in the scalability of ASP-solver. As we have mentioned, `clingo` cannot yet deal with input larger than 70MB. Considering that in the current experiment, we only use data from 261 studies (around 1/10 of the total number of studies) and the necessary data could go up to 44MB, a full fledged implementation of PhyloWS using ASP will require additional techniques and/or better ASP-solvers. This also raises the question of whether other ASP extensions (e.g., DLVDB [20]) will provide a more scalable implementation.

7 Conclusion and Future Work

We described an ASP based implementation of PhyloWS, a web services API for phylogenetic applications. The implementation focuses on retrieval services, expressed by four different types of queries. We discussed the ASP implementation of the queries and evaluated with data from 261 studies extracted from TreeBASE. We detailed the design choices and discussed the limitation of the implementation that presents a challenge to the ASP community.

To continue with the development of PhyloWS, we plan to exploit the strengths of ASP to enrich PhyloWS with (*i*) constraints over the answers; and (*ii*) preferences between answers. We envision that this can be achieved via a web-interface that not only allows users to specify their queries but also the additional constraints and preferences. We plan to experiment with other ASP-extensions such as `dlvhex` or DLVDB to identify a more scalable system. In addition, we will also investigate whether different methods of computing answer sets (e.g., using reactive answer set solver) could be useful. Finally, we plan to complete the import of data from the 9558 CDAO files to CDAOStore.

References

1. T. Asano, J. Jansson, K. Sadakane, R. Uehara, and G. Valiente. Faster computation of the Robinson-Foulds distance between phylogenetic networks, 2010.
2. O. R. P. Bininda-Emonds. *Phylogenetic Supertrees: Combining Information to Reveal the Tree of Life,* Computational Biology Series, Vol. 4. Kluwer Academic Publisher, 2004.
3. B. Chisham, E. Pontelli, T. C. Son, and B. Wright. Cdaostore: A phylogenetic repository using logic programming and web services. In *Technical Communications of the 27th ICLP*, Vol. 11, *LIPIcs*, 209–219. 2011.
4. T. Eiter, G. Ianni, R. Schindlauer, and H. Tompits. ACM international conference on web intelligence. In *Web Intelligence*, pages 1073–1074. IEEE Computer Society, 2006.
5. H. Ellegren. Comparative genomics and the study of evolution by natural selection. *Molecular Ecology*, 17(21):4586–4596, 2008.
6. E. Erdem. PHYLO-ASP: Phylogenetic Systematics with Answer Set Programming. In *LPNMR*, 567–572. Springer, 2009.
7. E. Erdem, V. Lifschitz, and D. Ringe. Temporal phylogenetic networks and logic programming. *TPLP*, 6(5):539–558, 2006.
8. J. Felsenstein. The newick tree format, 1986. `http://evolution.genetics.washington.edu/phylip/newicktree.html`.
9. W. M. Fitch. Uses for evolutionary tree. *Phi. Trans. R. Soc. Lond. B*, 349:93–102, 1995.
10. M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. clasp: A conflict-driven answer set solver. In *LPNMR*, 260–265. Springer-Verlag, 2007.
11. H. Lapp and R. Vos. Phyloinformatics Web Services API: Overview. `https://www.nescent.org/wg/evoinfo/index.php?title=PhyloWS`, NESCE, 2009.
12. D. Maddison, D. Swofford, and W. Maddison. NEXUS: an Extensible File Format for Systematic Information. *Syst. Biol.*, 46(4):590–621, 1997.

13 V. Marek and M. Truszczyński. Stable models and an alternative logic programming paradigm. In *The Logic Programming Paradigm: a 25-year Perspective*, 375–398, 1999.

14 V. Morell. TreeBASE: the roots of phylogeny. *Science*, pages 273–569, 1996.

15 L. Nakhleh, D. Miranker, F. Barbancon, W. Piel, and M. Donoghue. Requirements of phylogenetic databases. In *3rd IEEE Symposium on Bioinf. and Bioeng.*, 141–148, 2003.

16 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.

17 F. Prosdocimi, B. Chisham, E. Pontelli, J.D. Thompson, and A. Stoltzfus. Initial implementation of a comparative data analysis ontology. *Evol. Bioinfor.*, 5:47–66, 2009.

18 M. Sanderson, B. G. Baldwin, G. Bharathan, C. S. Campbell, D. Ferguson, J. M. Porter, C. VonDohlen, M. F. Wojciechowski, and M. J. Donoghue. The growth of phylogenetic information and the need for a phylogenetic database. *Syst. Biol.*, 42:562–568, 1993.

19 A. Stoltzfus, N. Cellinese, K. Cranston, H. Lapp, S. McKay, E. Pontelli, and R. Vos. The evoio interop project. `http://www.evoio.org/wiki/Main_Page`, NESCE, 2009.

20 G. Terracina, N. Leone, V. Lio, and C. Panetta. Experimenting with recursive queries in database and logic programming systems. *TPLP*, 8(2):129–165, 2008.

21 R. Vos. `nexml`: Phylogenetic data in xml. `http://www.nexml.org`, 2008.

22 C. Webb, D. Ackerly, M. McPeek, and M. Donoghue. Phylogenies and communtiy ecology. *Annu. Rev. Ecol. Syst.*, 33(1), 2002.

CHR for Social Responsibility

Veronica Dahl[1], Bradley Coleman[2], J. Emilio Miralles[3], and Erez Maharshak[4]

1 **School of Computing Science, Simon Fraser University**
Burnaby, B.C. Canada
veronica@cs.sfu.ca

2 **School of Computing Science, Simon Fraser University**
Burnaby, B.C. Canada
bradley@proxydemocracy.org

3 **Department of Physics, Simon Fraser University**
Burnaby, B.C. Canada
emiralle@sfu.ca

4 **Cognitive Science Department and School of Computing Science, Simon Fraser University**
Burnaby, B.C. Canada
erez@proxydemocracy.org

Abstract

Publicly traded corporations often operate against the public's interest, serving a very limited group of stakeholders. This is counter-intuitive, since the public as a whole owns these corporations through direct investment in the stock-market, as well as indirect investment in mutual, index, and pension funds. Interestingly, the public's role in the proxy voting process, which allows shareholders to influence their company's direction and decisions, is essentially ignored by individual investors. We speculate that a prime reason for this lack of participation is information overload, and the disproportionate efforts required for an investor to make an informed decision. In this paper we propose a CHR based model that significantly simplifies the decision making process, allowing users to set general guidelines that can be applied to every company they own to produce voting recommendations. The use of CHR here is particularly advantageous as it allows users to easily track back the most relevant data that was used to formulate the decision, without the user having to go through large amounts of irrelevant information. Finally we describe a simplified algorithm that could be used as part of this model.

1998 ACM Subject Classification D.3.2 Language Classifications - Constraint and logic languages, H.4.2 Types of Systems – Decision support, K.4.3 [Computers and Society] Organizational Impacts

Keywords and phrases Constraint handling rules, principled decision making, informed voting, client directed voting, social responsibility.

Digital Object Identifier 10.4230/LIPIcs.ICLP.2012.370

1 Introduction

The financial crisis of 2008 showed us all how deeply corporations impact our lives, and in turn the importance of Corporate Social Responsibility (CSR). In this paper we argue that modern information technology can promote CSR, both from the perspective of investing in socially responsible companies or investment vehicles, and in using the voting rights that shareholders are given to impact companies in a way that can make them both more profitable and more socially responsible.

© Veronica Dahl, Bradley Coleman, J. Emilio Miralles, and Erez Maharshak;
licensed under Creative Commons License ND
Technical Communications of the 28th International Conference on Logic Programming (ICLP'12).
Editors: A. Dovier and V. Santos Costa; pp. 370–380

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

In particular we argue that Constraint Handling Rules (CHR) [3] is an effective tool for this, and we propose a CHR-based model of Informed Advice capable of recommending investment and voting decisions that are consistent with a user's stated preferences (values), and of explaining the reasons for these recommendations.

As well, we propose a practical and novel method of making automatic proxy voting recommendations based on a user's values. Currently, only around 5% [4] of retail investors vote their proxies, and the rate of informed participation is even lower. Our tool addresses the root causes of this dearth, namely information overload and under-load, and could increase the quantity and quality of decisions. We propose a method that only asks users to choose how aggressively they wish to vote on various classes of proposals, like sweat-shop labor, the environment, and director elections.

The paper is structured as follows: in section 2 we discuss the main issues around making effective and principled decisions. Section 3 presents the model itself, exemplified through principled decision making on investment in a company or mutual/index fund. In section 4, we describe our novel method of making automatic proxy voting recommendations. Section 5 presents our concluding remarks.

Consistent use of our proposed model would inform users who might otherwise be unaware of which decisions would clash with their value system and why. More importantly, it would spare users from going through the incredible amount of information needed in order to make informed decisions. Our model potentially allows presenting users with specific and focused information relevant to the decision at hand. Insofar as users become empowered to vote and decide according to their value systems, a wider range of stakeholders and owners would take part in the corporate governance process. This could promote a healthy evolution in management concepts and higher corporate governance standards.

2 Making effective and principled decisions

There is growing recognition of the fact that the public needs to be given choices that reflect their values and principles. For instance, ethical investment is increasing in popularity. Ethical investing indices are those that only include companies satisfying environmental or social criteria. Among these, Wikipedia names as examples those of FTSE4Good Index, and Dow Jones Sustainability Index. It also explains that strict mechanical criteria for inclusion and exclusion in such indices is important to prevent accusations of ideological bias in selection, as well as to prevent market manipulation, e.g. in Canada when Nortel was permitted to rise to over 30% of the TSE 300 index value[1].

However, it is not easy to effectively recognize which choices really do reflect one's values and principles. As also pointed out in Wikipedia, mechanical criteria can yield misleading results, since a firm can satisfy mechanical "ethical criteria", e.g. regarding board composition or hiring practices, but fail to perform ethically with respect to shareholders, e.g. Enron. There is indeed the risk that the seeming "seal of approval" of an ethical index may facilitate scams by putting investors more at ease.

For instance, self descriptions regarding sustainability or ethics could induce mechanical criteria to include the self-describing company among ethical ones, even though their ethical traits may be questioned elsewhere. Thus, a company such as Dow Chemical, which Wikipedia describes as responsible for actions that some consider unethical or against sustainability[2],

[1] `http://en.wikipedia.org/wiki/Stock_market_index#Ethical_stock_market_indices`
[2] Such as managing a nuclear weapons production facility that produced plutonium triggers for hydrogen

would probably be rated as ethical by an automatic rating agent looking at Dow Chemical's own website.

Even if we instruct automatic criteria to avoid using self descriptions, indices deemed to be sustainable could give high ratings even to companies that are believed not to be, and this inclusion alone could induce mechanical criteria to advise ethically-conscious investors to invest in those companies. A case in point, the Dow Jones Sustainability World Index recently rated the Dow Chemical Company as "one of the top performers in the global chemical industry", giving it the highest scores in the sector for operational eco-efficiency, customer relationship management and environmental reporting.

An alternative to mechanical criteria might be provided by market transparency and disclosure, but the problem remains how to collect the data into a knowledge base and how to use it together with users' stated principles and preferences in order to guide the search for advice that is consistent with those values. We shall argue that to solve this problem, we can resort to databases where the information is verified by specialists (e.g., international lawyers in charge of lawsuits concerning a questionable firm can verify whether the firm was found guilty), together with a CHR program which can be tailored for specialization into various applications that consult those databases (e.g. informed voting, informed investment, etc.).

3 Our proposed model for responsibly-informed decision making

As discussed, mechanical selection and disclosure and transparency are the present options for dealing with the fact that corporations are, in general, not trustworthy. None of these options is satisfactory, since mechanical selection can actually perpetuate scam by creating a false sense of security, and there always will be corporations that pretend to disclose and to exhibit transparency while not being totally sincere.

This dilemma between unsatisfying options can be overridden by a) placing relevant and reliable information in a database which can then be consulted, b) placing the description of users' values, companies' values and any other relevant information as initial constraints for a CHR program, and c) letting the CHR program run over the database when given some kind of "question", e.g. who should a given user vote for, or what decision should he/she make when faced with some specific problem.

We next introduce our proposed model through examples in the specific financial domain of choosing companies to invest on. There will be a system-defined part of the program, which the user needs not be concerned with, and which will adequately process the user's definitions.

3.1 Priority definitions

These are done through propagation rules, in which each criterion is associated with either a high, low, or medium priority[3]. To exemplify:

bombs (the Rocky Flats Plant), manufacturing napalm B, supplying the dioxin containing Agent Orange that was used as a weapon, or producing a soil fumigant, DBCP, which was responsible for sterilizing male workers in banana plantations in Latin America after most domestic uses of DBCP were banned in 1977 due to the successful lawsuits from workers at Dow's DBCP production who were made sterile by exposure to the compound.

[3] Alternatively, we could allow for numeric measures such as 0.9, or perhaps automatically translate less precise measures such as high, low, and medium into numeric values according to some algorithm that takes all the user's priorities into account.

```
priorities ==> priority(environmentalSafety,high),
 priority(humanSafety,high),
 priority(transparency,medium),
 priority(goodHistoricYield,medium), ...
```

3.2 Goal definitions

These are also described through propagation rules, e.g. the user can set goals such as to get at least 5% on average yearly, e.g.:

```
goals ==> goal(minimumAvgeYield,5), ...
```

3.3 The knowledge base: its sources, reasons, and trustworthiness

We use a knowledge base that lists for all candidate companies specific incidents that justify a given score with respect to each of the criteria. We do this through a 5-ary predicate "criterion", whose first argument names the criterion in question, whose second argument refers to a company, whose third argument rates the level at which the company satisfies the criterion (i.e., low, medium, or high), whose fourth argument summarizes the reason for said rating, and whose fifth and last argument records the URL from which this summary was extracted, e.g.:

```
criterion(humanSafety,'DowChemical',low,'because it sells
 chemicals that damage the human nervous system and have been
 banned from the US for that reason, to third world countries
 that do not yet have protective regulations',
 'http://en.wikipedia.org/wiki/Dow_Chemical_Company#DBCP').
```

The database also lists historic average yields and any other goal defined under "goals", e.g.[4]:

```
achievedHistorically(minimumAvgeYield,DowChemical,40).
```

It is important to consider from what sources the database will be constructed in each case. For instance having consulted Dow Chemical's own description, which presents itself as a sustainable company, the following contradicting information could co-exist in the same database:

```
criterion(humanSafety,'DowChemical',high,'because it addresses
 many of the world's most challenging problems such as the need
 for clean water, renewable energy generation and conservation,
 and increasing agricultural productivity',
 'http://www.dow.com/news/corporate/2011/20110908a.htm').
```

As we can see, the information to be included in the knowledge base can be unreliable in some cases, or partial, or even contradictory (as in the case of Dow Chemical listing itself as sustainable, which clashes with the belief of some that several of its actions were far from sustainable or ethical). We can adopt some criteria to decrease the risk of error in rating, such as adding a marker of trustworthiness to each of the above rules related to the quality of independent verification, e.g. the international lawyers that participated in the legal actions against Dow Chemical could give faith that the first rule is accurate.

[4] This is an example, not an actual yield.

We postulate that in the long run, the best way to achieve enough information is to allow for information to come from various sources, to allow for the potential contradictions that this can generate, and to simply output both the (perhaps contradicting) advice and its rationale in each case. The human user can then make up his mind on which of the info to follow, or in a later stage of our system, we could resort to argumentation theory [7, 1, 2] to weigh the merits of each argument and counter-argument automatically.

Our proposal is innovative because previous solutions to the problem of preventing market manipulation are based on either mechanical selection, which is always less discriminative than humans', and as we saw facilitates scams, or on just trusting the corporations, which many people believe to be imprudent at the least.

It is important to note that in a world in which web documents can contain timely information not easily found elsewhere, and in which concept extraction from web documents is becoming more and more efficient, we will likely tend to rely increasingly on them as sources of information. For this additional reason we believe that it is important to allow contradicting information to enter our database, while giving the user appropriate tools to deal with it a posteriori, such as argumentation theory. In any case, since some of the issues at hand might be controversial, we feel it is best to allow diverse points of view to be reflected, and leave it to the user to decide which one he or she wants to adhere to. The important thing for this is that we provide the rationale and the URL source which will allow them to make a truly informed decision.

3.4 The system

The system's shell itself can be created in just a few, relatively simple, CHR rules. To exemplify, the following CHR rule expresses that a Company meets a given criterion and goal if the user's values for that criterion and goal coincide with the values of the company. Now since the goal is met, the reason and its justifying link get printed as well.

```
priority(Criterion,Value), goal(G,N)
 ==> criterion(Criterion,Company,Value,Reason,Link),
 achievedHistorically(G,Company,N1),
 N1> N,
 print_reason(Company,Reason,Link)
 | ok(Company).
```

Of course, we could modify such rules in various ways as needed, e.g. demanding that the user's value either coincides with, or is less than, the value the company gets, or we could check all goals, not just the single one of the example.

The result will be a list of companies that are eligible according to each goal and criterion. This can be sorted out by the user or further processed by the system.

4 Principled and informed voting

The above will rapidly turn relevant as financial data-tagging becomes the standard practice mandated by the U.S. Securities and Exchange Commission[5]. This will allow matching various complex criteria in large data-sets. One especially practical and interesting implementation

[5] http://www.forbes.com/sites/tomgroenfeldt/2011/09/16/mandated-data-tagging-makes-sec-reports-useful-to-investors/

would be matching individuals' beliefs and voting principles with financial and proxy data. This would assist users in arriving at a voting decision when a large volume of data is available and only a tiny subset is relevant for the decision. One simplified possibility for this will be explored in the next section.

4.1 Motivation

As corporate power grows and the power of governments falls, mechanisms to govern corporations become more important. As governmental power falls, their *power* to regulate corporations falls as well. Further, as the influence of corporations over governments increases (eg. lobbying), the *will* of governments to regulate corporations also falls. This can form a positive feedback loop.

Hope is not lost though, since there is an existing structure at every publicly traded company wherein its shareholders – the corporation's actual owners – vote on high level decisions at the company. This mechanism is aptly referred to as corporate governance. Loosely, shareholders do not vote on which product the company should release, but instead they help elect a board of directors that will steer management well.

Furthermore, if a shareholder owns a fixed amount of shares, they may place a proposal of their own on the company's ballot for all of the shareholders to vote on. This is known as a shareholder proposal. As a policy, management supports management proposals and opposes shareholder proposals, because if management agreed with a shareholder proposal, then they would implement it themselves. Shareholder proposals can range in subject matter from the genocide in Darfur, to the environment, to sweat-shop labor, to executive compensation, to disclosing political contributions, to amending the corporate by-laws to, for instance, separate the Chairman and CEO positions. These proposals can call for an action, or for disclosure from the company. Since the range of management and shareholder proposals is so wide, good voting can lead to corporations that are both more profitable, and more socially responsible.

Shareholders are the literal owners of a company. Indeed, the ownership relationship is very powerful, arguably more so than the citizenship relationship between citizens and their governments, yet only about 5% of retail investors vote their shares [4], and even fewer investors cast informed votes. This is because of apathy, information overload, and information under-load.

ProxyDemocracy.org is a non-partisan, non-profit website that attempts to solve these (and related) issues, primarily in the U.S.. The website is essentially an interactive database of votes cast by institutional investors. Most of these votes were collected by scraping mandatory disclosure filings which are disclosed after the votes have been cast. However, for the purposes of this paper, we will focus on institutions who disclose votes prior to meetings. There are ten such institutions whose votes can be found on ProxyDemocracy. These institutions primarily predisclose in an effort to gain support for their positions by getting out ahead of other investors. Other investors, very large and very small, can and do see these early votes on ProxyDemocracy.org and use them to inform their own voting decisions.

4.2 Client directed voting

Client directed voting (CDV) seeks to improve voting rates and/or voting quality by automating the voting process. Mark Latham introduced this concept in [5], but [6] is more current. Still, the term "Client Directed Voting" was coined by Stephen Norman of American

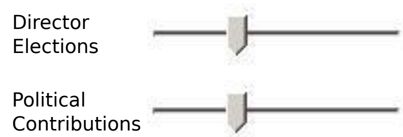


Figure 1 Left is passive, right is activist.

Express in 2006. There are many ways to envision this and to implement it. Here we propose a method which we argue will improve both participation and voting quality.

The state-of-the-art in computing sciences is such that modern information systems could aggregate the votes of institutions that publish their votes before meetings and use them to make voting recommendations congruent with a user's stated values. The user need only declare his voting preferences once and these can be used to automatically generate suggested votes on an on-going basis. The user would still likely need to approve each suggested vote for regulatory purposes. Automated voting is surely not foolproof, but the automation largely overcomes information overload, and the basis of respected institutional investors (that have the expertise and resources to vote well) largely overcomes information under-load.

We will use the term *passive* to describe a vote in favor of management, and *activist* to describe a vote against management. There is no value judgement intended by this word choice. We use these two terms to abstract the type of proposal so that we do not have to distinguish between management and shareholder proposals and votes that are "For", "Against", or "Withhold". Each vote will simply be considered to be either activist or passive.

The back-end of this CDV implementation requires predisclosing institutions, their voting histories, and a simple mechanism for automatically classifying proposals by type (e.g. director elections, political contributions). ProxyDemocracy currently shows the early votes of ten institutions, as well as their voting histories broken down by issue type, so these three requirements are straight-forward and attainable with a moderate effort.

The front-end requires only that for each issue type, the user indicates how activist or passive they want to vote. The user will indicate how activist or passive they wish to vote on each issue type by moving a slider, where the leftmost position will be as passive as possible, and the rightmost position will be as activist as possible (see Figure 1).

Voting decisions will be based entirely on the early votes of these predisclosers, their voting histories, and the user's slider positions.

This algorithm does not ask users to decide which predisclosing institutions they want factored into their voting decisions. This is a tempting, but problematic design choice, because most users are not familiar with these institutions, nor do they have opinions on their voting records. In fact, this algorithm only asks users the one question that any CDV scheme that aims to help users vote their values must ask, namely how they want to vote on the issues.

Furthermore, the algorithm we propose uses each and every predisclosing institution to make the decision. This is regardless of whether, for some issue, an institution's voting record aligns with the user's given values or not. This approach is inspired by the mathematical field of Information Theory, in that there is useful information in each and every voting decision, and thus each one should be factored into the final decision.

4.3 Our algorithm

This algorithm only decides a users' vote for one single proposal. For this vote, assume that k institutions have disclosed their vote by this time. We will use all k votes to make our decision. Let v_i be the vote of institution i, where $v_i = \{\text{activist, passive}\}$. These votes are represented by:

$$v_1, v_2, \ldots, v_k.$$

Also, let p_i, where $0 \leq p_i \leq 1$, be the historical probability of institution i casting an activist vote. More specifically, p_i is exactly the frequency with which institution i has cast activist votes on this issue in the past, which we know from this institution's voting record. For example, if some p_i is close to 0 on this issue, then institution i is passive on this issue. These probabilities are represented by:

$$p_1, p_2, \ldots, p_k.$$

For a given issue (like director elections, see figure 1), the position the user chooses will become the value s, where $-1 \leq s \leq 1$. For example, if $s = -1$, then the the user is maximally passive on this issue, if $s = 0$, then the user is neutral, and if $s = 1$, then the user is maximally activist. The users voting preference for this issue is represented by s.

Note that the the p_i for these k institutions may conform to any probability distribution. For instance, all k institutions might be very passive. We will use a weighting function $f : \{1, \ldots, k\} \to \mathbb{R}$ to attempt to mitigate any bias in this probability distribution by assigning each vote a weight. The function f is just one possible weighting function, there may be many that work well.

$$f_i = \begin{cases} 1 - p_i & \text{if } v_i = \text{activist}, \\ p_i & \text{if } v_i = \text{passive}. \end{cases}$$

The function f assigns a very small weight to a passive vote of an institution that votes passively on this issue, and similarly assigns a very small weight to an activist vote from an institution that is an activist on this issue. Conversely, f will assign a large weight to a passive vote from an activist institution, and the symmetric. Thus, if all of the predisclosers are historically passive on an issue, but the user wants to be activist, then if at least one prediscloser cast an active vote, this scheme will weight that vote high and the passive votes low.

The intuition behind this is that if an institution that is passive on directors votes against a director, then this director must be significantly unsatisfactory. Similarly, if a very environmentalist institution votes against an environmental proposal, there is a high likelihood that there is a significant problem with this proposal.

Now we will explain the second weighting function $g : \{1, \ldots, k\} \to \mathbb{R}$. This function scales some of the weightings that f created, and using the slider value s, creates new weightings, g_i:

$$g_i = \begin{cases} f_i & \text{if } v_i = \text{activist and } s \leq 0, \\ f_i(2 - (1 - s)) & \text{if } v_i = \text{activist and } s > 0, \\ f_i(2 - (1 + s)) & \text{if } v_i = \text{passive and } s < 0, \\ f_i & \text{if } v_i = \text{passive and } s \geq 0. \end{cases}$$

Again, function g is just one of many possible weighting functions. Further, it is limited in that, if the user is passive on this issue, it will only scale the passive votes by at most two. Observe that if the user picks the value -1, it will multiply the weight of each passive vote by 2. The function g can indeed scale either the passive or activist votes by at least 1 and at most 2. This scaling factor is an arbitrary choice, and can likely only be justified with rigorous experimentation with this and other weighting functions.

The intuition behind f and g is that first f helps to mitigate the bias from the vote sources, and then g applies the user's bias. Finally if $\sum_{v_i=Activist} g_i > \sum_{v_i=Passive} g_i$, then we cast an activist vote, and otherwise we vote passively. Below is an example.

Prior to demonstrating this simple approach in the next section, it is worth noting that we have tested it on a very large data set of approximately one million historical proxy votes, which has so far showed the effectiveness of the general approach. You can find a detailed account of the trial in this URL: `https://docs.google.com/open?id=0B57uHUYhCLdCRi15RGtoQlRvX2c`.

4.4 Example

In this example, we show how four institutions voted in a 2010 proposal to elect Charles Prince to serve on the board of directors of Johnson & Johnson. Some believe Mr. Prince to be a controversial director. He served as the chairman and chief executive officer of the investment bank Citigroup from 2002 until his resignation in November of 2007, shortly before the financial crisis of 2008. Thus, some claim he might have been involved in the decisions that, shortly after his resignation, brought about Citigroup's collapse (market capitalization crashed from $244B down to $20B) and subsequent bailout by the U.S. federal government. While some people may question his competence, his experience and influence as a chief executive and banker may be important to Johnson & Johnson. He was first appointed in 2006 and continues to serve on Johnson & Johnson's board.

The institutional voters for this proposal include CalSTRS, the pension fund for the teachers of California, AFSCME, a labor union pension fund, CBIS, a Catholic ministry pension fund, and Vanguard, the world's largest mutual fund company with more than $2T in assets under management. The first three institutions predisclose their proxy votes, but Vanguard does not. Hence, we would not have been privy to their vote (vi value) in advance. Yet, being a past proposal we can include it just to represent a more conservative school of thought.

In the first table, we are given the p_i's and the v_i's of the predisclosing voters (including Vanguard, which did not actually predisclose), and we compute the f_i's and the g_i's for four different values of s.

Institution	p_i	v_i	f_i	g_i (s=-.7)	g_i (s=-.3)	g_i (s=0)	g_i (s=1)
CalSTRS	.470	Activist	.530	.530	.530	.530	1.060
AFSCME	.438	Passive	.438	.745	.569	.438	.438
CBIS	.500	Passive	.500	.850	.650	.500	.500
Vanguard	.085	Passive	.085	.145	.111	.085	.085

In the second table, we show the sums of the activist and passive votes for each of the four values of s and the vote that is actually cast. With these four predisclosers, when $s = 1$, the algorithm casts an activist vote. This is because the sum of the weighted passive votes is less than the one weighted and scaled activist vote cast by CalSTRS (an activist vote is a vote against Mr. Prince). Similarly, the algorithm casts a passive vote when s is $-.7, -.3$

or 0. Note that if the f value of CalSTRS, which is .530 had been only slightly higher, the algorithm would have voted for Mr. Prince regardless of the s (slider value) chosen by the user.

s	$\sum_{v_i=Activist} g_i$	$\sum_{v_i=Passive} g_i$	vote cast
-.7	.530	1.739	Passive
-.3	.530	1.330	Passive
0	.530	1.023	Passive
1	1.060	1.023	Activist

5 Conclusion

We have presented arguments in favor of developing computerized systems for responsible decision making based on CHR, and exemplified our ideas in the context of automatically helping a user choose among investment possibilities in accordance with the user's values. We have also proposed an algorithm for client directed voting which can be readily incorporated as well into our CHR based system, thanks to its high modularity, and implemented a toy CHR program as proof of concept (see: `https://docs.google.com/open?id=0B57uHUYhCLdCRi15RGtoQlRvX2c`)The idea that the world is a symbol for thought, or thought materialized, can be postulated at various levels, from the most literal to the most metaphysical. By allowing principled thought guided by humanistic ends to become a matter of fact embedded in computer systems we can contribute both to humanize computers and to expand human consciousness in ways direly needed at the present juncture in our civilization. In particular, automating informed decision making could, through forcing corporations to choose between meeting users' values or losing their support, revolutionize the way corporations are run enough to transform them into agents of positive change. With this paper we hope to stimulate further work along these lines.

It is also important to note that, while we have focused on the specific areas of financial advice and voting, our described methodology can be readily adapted to other areas where informed decision-making can be supported by computers. Thus, other than by empowering voters and investors, the social implications of our proposed model's research could be mind-boggling from the point of view of potential systematic contributions to societal participation (through making wider consultation possible) and to the elevation of the world's educational levels, given that even poor areas of the world are gaining affordable access to mobile phones to which the needed databases could be fed and consulted. For instance our informed advice system adapted to medicine could allow patients to make more educated, conscious, and personally germane decisions on their treatments through relevant automatic expansion of the information received at a doctor's visit. In the long run, our research will make it possible to implement a model of machine informed human cognition around guidelines that consistently focus on human values and concerns, hence promoting an expansion of global consciousness around humanistic lines, with profound transformational effects.

6 Acknowledgements

This work was supported by V. Dahl's NSERC grant 31611024. We would like to thank the anonymous referees for their feedback, Henry Saint Dahl for timely advice, and Andy Eggers at ProxyDemocracy/the London School of Commerce for his work on the database of votes.

References

1 Xiuyi Fan and Francesca Toni. Assumption-based argumentation dialogues. In Toby Walsh, editor, *IJCAI*, pages 198–203. IJCAI/AAAI, 2011.

2 Xiuyi Fan and Francesca Toni. Conflict resolution with argumentation dialogues. In Liz Sonenberg, Peter Stone, Kagan Tumer, and Pinar Yolum, editors, *AAMAS*, pages 1095–1096. IFAAMAS, 2011.

3 Thom Frühwirth and Frank Raiser, editors. *Constraint Handling Rules: Compilation, Execution, and Analysis*. March 2011.

4 Chris Kentouris. Broadridge to u.s. corporations: Tell employees to vote. `http://www.securitiestechnologymonitor.com/news/-27412-1.html`, March 2011.

5 Mark Latham. The internet will drive corporate monitoring. *Corporate Governance International*, 3(2), 2000. Available at votermedia.org/publications. 1999 version at SSRN eLibrary".

6 Mark Latham. Proxy voting brand competition. *Journal of Investment Management*, 5(1), 2007. Available at votermedia.org/publications.

7 Iyad Rahwan and Guillermo R. Simari, editors. *Argumentation in Artificial Intelligence*. Springer Publishing Company, Incorporated, 1st edition, 2009.

A Logic Programming approach for Access Control over RDF

Nuno Lopes[1], Sabrina Kirrane[2], Antoine Zimmermann[3], Axel Polleres[4], and Alessandra Mileo[1]

1 **Digital Enterprise Research Institute**
`{nuno.lopes,alessandra.mileo}@deri.org`
2 **Digital Enterprise Research Institute and Storm Technology**
`sabrina.kirrane@deri.org`
3 **École Nationale Supérieure des Mines, FAYOL-ENSMSE, LSTI, F–42023 Saint-Étienne, France**
`antoine.zimmermann@emse.fr`
4 **Siemens AG Österreich, Siemensstrasse 90, 1210 Vienna, Austria**
`axel.polleres@siemens.com`

Abstract

The Resource Description Framework (RDF) is an interoperable data representation format suitable for interchange and integration of data, especially in Open Data contexts. However, RDF is also becoming increasingly attractive in scenarios involving sensitive data, where data protection is a major concern. At its core, RDF does not support any form of access control and current proposals for extending RDF with access control do not fit well with the RDF representation model. Considering an enterprise scenario, we present a modelling that caters for access control over the stored RDF data in an intuitive and transparent manner. For this paper we rely on Annotated RDF, which introduces concepts from Annotated Logic Programming into RDF. Based on this model of the access control annotation domain, we propose a mechanism to manage permissions via application-specific logic rules. Furthermore, we illustrate how our Annotated Query Language (AnQL) provides a secure way to query this access control annotated RDF data.

1998 ACM Subject Classification I.2.4 Knowledge Representation Formalisms and Methods

Keywords and phrases Logic Programming, Annotation, Access Control, RDF

Digital Object Identifier 10.4230/LIPIcs.ICLP.2012.381

1 Introduction

Enterprises rely on stand-alone systems, commonly known as Line Of Business (LOB) applications, to efficiently perform day-to-day activities: interactions with clients in a Customer Relationship Management (CRM) application, employee information in a Human Resources (HR) application, project documentation in a Document Management System (DMS), etc. These systems, although independent, often contain different information regarding the same entities; for example, if an organisation needs to know the projects commissioned by a customer, the employees that worked on those projects and the revenue that was generated, they need to obtain information across these systems. However, such integration is not a simple task, not only due to the heterogeneity of the systems, but also due to the presence of access control mechanisms in each system. In fact, since much of the information within the enterprise is highly sensitive, this integration step could result in information leakage to unauthorised individuals.

© Nuno Lopes, Sabrina Kirrane, Antoine Zimmerman, Axel Polleres, and Alessandra Mileo;
licensed under Creative Commons License ND

RDF is a flexible format for representing such integrated data, however it does not provide any mechanisms to avoid the problem of information leakage. In this paper we rely on an integration solution that extracts information from the underlying LOB applications into RDF. Based on this integrated data, we define a mechanism to enforce access control over the resulting RDF graph, implemented via logic programming. Our approach provides a flexible representation for the access control policies and also caters for permission propagation via logic inference rules.

The solution we present builds upon an extension of the RDF data model to supply context information (called Annotated RDF), that provides a backwards compatible model to attach domain-specific metadata to each RDF triple. The main contribution of this paper in relation to access control over RDF data consists of defining an annotation domain that models access control permissions in RDF. Based on this model, access control can be enforced by relying on an extension of SPARQL, the standard query language for RDF. Although in this paper we are considering that the access control annotated data stems from the integration of the data from LOB applications, the presented model can be applied as a general model for access control in RDF, without requiring the information integration step.

The remainder of the paper is structured as follows: in Section 2 we briefly introduce concepts from the Semantic Web research area and their extension to the annotated case. Section 3 formalises the access control annotation domain and details our implementation of the domain in logic programming. Finally, we describe the related work in Section 4 and present conclusions and directions for future work in Section 5.

2 Preliminaries

In this section we provide the necessary background information regarding the semantics of Annotated RDFS. We start by presenting the data model, giving an overview of RDF and its extension towards Annotated RDFS which draws inspiration from Annotated Logic Programming [13]. We then present the extension of the RDF Schema (RDFS) inference rules for the annotated case and the extension of the SPARQL query language for querying Annotated RDFS, AnQL. Finally, we present the current prototype implementation of Annotated RDFS and AnQL which is implemented in SWI Prolog.

2.1 Annotated RDFS Data Model

We present an overview of the concepts of RDF and its extension to Annotated RDFS.

▶ **Definition 1** (RDF triple, RDF graph)**.** Considering the disjoint sets $\mathbf{U}$, $\mathbf{B}$ and $\mathbf{L}$, representing respectively URIs, blank nodes and literals, an *RDF triple* is a tuple $(s, p, o) \in \mathbf{UB} \times \mathbf{U} \times \mathbf{UBL}$,[1] where s is called the *subject*, p the *predicate*, and o the *object*. An *RDF graph* G is a finite set of RDF triples.

An RDF triple has the intuitive meaning that the *subject* is connected to the *object* by the *predicate* relation. In this work, we avoid introducing details about the concrete syntaxes of RDF, and we omit minutiae. Please refer to [15] and [9] for specifics.

Several extensions were presented to introduce meta-information into the RDF data model. For example, [7] define temporal RDF, which allows for the allocation of a validity

[1] For conciseness, we represent the union of sets simply by concatenating their names.

interval to an RDF triple; [20] presents fuzzy RDF in order to attach a confidence or membership value to a triple. These and other approaches can be represented within a common framework, called Annotated RDF [23] and further extended to include RDFS inference rules by [21]. Annotated RDFS introduces the notion of an *annotation domain* into the RDF model and defines an extension of the RDFS inference rules that, by relying on the $\otimes$ and $\oplus$ (cf Definition 2) operations defined by the annotation domain, can be specified in a *domain independent* fashion. Next we present the definition of an annotation domain

▶ **Definition 2** (Annotation Domain). Let L be a non-empty set, whose elements are considered the *annotation values*. We say that an *annotation domain* for RDFS is an idempotent, commutative semi-ring $D = \langle L, \oplus, \otimes, \bot, \top \rangle$, where $\oplus$ is $\top$-annihilating. That is, for $\lambda, \lambda_1, \lambda_2 \in L$:

1. $\oplus$ is idempotent, commutative, associative;
2. $\otimes$ is commutative and associative;
3. $\bot \oplus \lambda = \lambda$, $\top \otimes \lambda = \lambda$, $\bot \otimes \lambda = \bot$, and $\top \oplus \lambda = \top$;
4. $\otimes$ is distributive over $\oplus$, i.e. $\lambda_1 \otimes (\lambda_2 \oplus \lambda_3) = (\lambda_1 \otimes \lambda_2) \oplus (\lambda_1 \otimes \lambda_3)$;

An annotation domain $D = \langle L, \oplus, \otimes, \bot, \top \rangle$ induces a partial order $\preceq$ over L defined as: $\lambda_1 \preceq \lambda_2$ iff $\lambda_1 \oplus \lambda_2 = \lambda_2$.

▶ **Example 3** (Annotation Domain). The Fuzzy Annotation Domain is defined as $D_{[0,1]} = \langle [0, 1], \max, \min, 0, 1 \rangle$. We can specify that `:joeBloggs` is a part-time employee of `:westportCars` as follows:

$$(\texttt{:joeBloggs}, \texttt{:worksFor}, \texttt{:westportCars})\colon 0.5$$

For the definitions of other domains, such as the temporal domain, the reader is referred to [21]. In Section 3.1 we present the definition of an annotation domain to model access control. Further to the above annotation domain definition, we extend RDF towards annotated RDFS:

▶ **Definition 4** (Annotated triple, graph). An *annotated triple* is an expression $\tau\colon \lambda$, where τ is an RDF triple and λ is an *annotation value*. An *annotated RDFS graph* is a finite set of annotated triples.

The entailment between two Annotated RDFS graphs, $G \models H$ is defined by a model-theoretic semantics presented in [21].

2.2 Inference Rules

RDF Schema (RDFS) [4] consists of a predefined vocabulary that assigns specific meaning to certain URIs, allowing a reasoner to infer new triples from existing ones. A set of inference rules can be used to provide a sound and complete reasoner for RDFS [22]. These rules can be extended to support Annotated RDFS reasoning, in a domain-independent fashion, simply by relying on the $\otimes$ and $\oplus$ operations (presented in Definition 2). Such rules can be represented by the following meta-rule:

$$\frac{\tau_1\colon \lambda_1,\ \ldots,\ \tau_n\colon \lambda_n, \{\tau_1, \ldots \tau_n\} \vdash_{\mathsf{RDFS}} \tau}{\tau\colon \bigotimes_i \lambda_i} \quad . \tag{1}$$

This rule reads that if a classical RDFS triple τ can be inferred by applying an RDFS inference rule to triples $\tau_1, \ldots \tau_n$ (denoted $\{\tau_1, \ldots, \tau_n\} \vdash_{\mathsf{RDFS}} \tau$), the same triple can be

inferred in the annotated case with annotation term $\bigotimes_i \lambda_i$, where λ_i is the annotation of triple τ_i. The $\oplus$ operation is used to combine information about the same statement: if the same triple is inferred from different rules or steps in the inference, the following rule is applied:

$$\frac{\tau\colon \lambda_1,\ \tau\colon \lambda_2}{\tau\colon \lambda_1 \oplus \lambda_2}\ . \tag{2}$$

It is also possible to specify a custom set of rules in order to provide application specific inferencing.

2.3 AnQL: Annotated Query Language

The proposed query language for Annotated RDFS is AnQL [14], which consists of an extension to the W3C recommended query language for RDF, SPARQL [18], while also taking into consideration features from the upcoming SPARQL 1.1 language revision. Consider $\mathbf{V}$ a set of variables disjoint from $\mathbf{UBL}$. In SPARQL, a *triple pattern* consists of an RDF triple with optionally a variable $v \in \mathbf{V}$ as the subject, predicate and/or object. Sets of triple patterns are called *basic graph patterns* (BGP) and BGPs can be combined to create generic *graph patterns*. The semantics of SPARQL is based on the notion of *basic graph pattern* matching, where a *substitution* is a partial function $\mu\colon \mathbf{V} \to \mathbf{UBL}$.

For the extension of SPARQL towards the AnQL query language, we propose a specific annotation domain instance of D of the form $\langle L, \oplus, \otimes, \bot, \top\rangle$. Let $\mathbf{A}$ denote the set annotation variables, disjoint from $\mathbf{UBLV}$ and λ be an annotation value from L or an annotation variable from $\mathbf{A}$, called an *annotation label*. For a SPARQL triple pattern τ, we call $\tau\colon \lambda$ an *annotated triple pattern* and sets of annotated triple patterns are called *basic annotated patterns* (BAP). Similar to SPARQL, BAPs can be combined to create an *annotated graph pattern* and for further details we refer the reader to [14].

An *AnQL query* is defined as a triple $Q = (P, G, V)$ where: (1) P is an annotated graph pattern; (2) G is an annotated RDF graph; and (3) $V \subseteq \mathbf{VA}$ is the set of variables to be returned by the query. Given an annotated graph pattern P, we further denote by $var(P) \subseteq \mathbf{V}$ and $avar(P) \subseteq \mathbf{A}$ the set of variables and annotation variables respectively present in a graph pattern P. As presented in Example 5, the annotated graph pattern P is specified following the WHERE keyword where the variables are specified after the SELECT keyword.

▶ **Example 5** (AnQL query)**.** Considering the fuzzy domain presented in Example 3, we can pose the following query:

```
SELECT ?v ?av WHERE { ?v a :Company ?av }
```

where ?v is a variable from $\mathbf{V}$ and ?av is an annotation variable from $\mathbf{A}$.

The semantics of AnQL BAP matching is defined by extending the notion of SPARQL *basic graph pattern* matching to cater for annotation variables and their mapping to annotation values. For any substitution μ and variable v, $\mu(v)$ corresponds to the value assigned to v by μ. For a BAP P, $\mu(P)$ represents the annotated triples that correspond to P except that any variable $v \in vars(P) \cup avars(P)$ is replaced with $\mu(v)$.

▶ **Definition 6** (BAP matching, extends [16, Definition 2])**.** Let P be a BAP and G an Annotated RDFS graph. We define the *evaluation* of P over G, denoted $[\![P]\!]_G$, as the list of substitutions that are *solutions* of P, i.e. $[\![P]\!]_G = \{\mu \mid G \models \mu(P)\}$, according to the model-theoretic definition of entailment presented by [21].

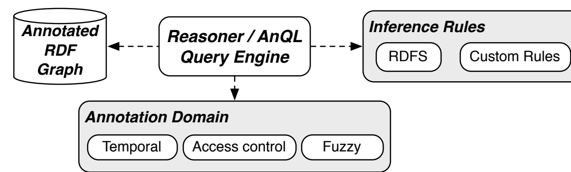


Figure 1 Annotated RDFS implementation schema.

The semantics of arbitrary annotated graph patterns is defined by an algebra that is built on top of this *BAP matching*. For further details we refer the reader to [14] and a combined overview of Annotated RDFS and AnQL is provided by [25].

2.4 Implementation

The system architecture of our prototype implementation, based on SWI-Prolog's Semantic Web library [24], is sketched in Figure 1. The main component of the system consists of the **Reasoner / AnQL Query Engine**, which is composed of a *forward-chaining* reasoner engine with a fix-point semantics that calculates the closure of a given **Annotated RDF Graph** [21] and an implementation of the AnQL query language. This main component can be tailored to a specific **Annotation Domain** and to include different **Inference Rules** describing how triples and their annotation values are propagated. Such inference rules can be specified, in domain independent fashion, by using a high-level language that abstracts the specific details of each domain. An example of an Annotated RDFS rule is presented in Example 7.

▶ **Example 7** (Annotated RDFS Inference Rule)**.** The following rule provides *subclass inference* in the RDFS ruleset:

```
rdf(O, rdf:type, C2, V) <==  rdf(O, rdf:type, C1, V1),
                             rdf(C1, rdfs:subClassOf, C2, V2),
                             infimum(V1, V2, V).
```

where the `rdf/4` predicate is used to represent the annotated triples and the `infimum/3` predicate corresponds to the implementation of the $\otimes$ domain operation (c.f. Definition 2).

More information and downloads of the prototype implementation can be found at `http://anql.deri.org/`.

3 Access Control Annotation Domain

In this section we formalise our access control annotation domain, following the definitions presented in Section 2.1, starting by defining the entities and annotation values and then presenting the $\otimes$ and $\oplus$ domain operations. Finally, we briefly describe the implementation of the presented annotation domain.

3.1 Entities and Annotations

For the modelling of the Access Control Domain (ACD) consider, in addition to the previously presented sets of URIs $\mathbf{U}$, blank nodes $\mathbf{B}$, and literals $\mathbf{L}$, a set of credential elements $\mathbf{C}$. The elements of $\mathbf{C}$ are used to represent *usernames*, *roles*, and *groups*. To represent *attributes*, we propose a set $\mathbf{T}$ of pairs of form (k, v), represented as key–value pairs where

$k \in \mathbf{U}$ and $v \in \mathbf{L}$. For example "$(:age, 30)$" or "$(:institute, DERI)$" are elements of $\mathbf{T}$.[2] We allow shortcuts to represent intervals of integers, for example "$(:age, [25, 30])$" to indicate that all entities with attribute "$:age$" between 25 and 30 are allowed access to the triple.

Considering an element $e \in \mathbf{CT}$, e and $\neg e$ are *access control elements*, where e is called a *positive* element and $\neg e$ is called a *negative* element.[3] An *access control statement* S consists of a set of access control elements and an *Access Control List* (ACL) consists of a set of access control statements. An access control statement S is *consistent* if and only if, for any element $e \in \mathbf{CT}$, only one of e and $\neg e$ may appear in S. This restriction avoids *conflicts*, where a statement is attempting to both *grant* and *deny* access to a triple. Furthermore, we can define a partial order between access control statements S_1 and S_2, as $S_1 \leq S_2$ *iff* $S_1 \subseteq S_2$. This partial order can be used to eliminate *redundant* access statements within an ACL: if a user is granted access by statement S_2, he will also be granted access by statement S_1 (and thus S_2 can be removed). Finally, an ACL is *consistent* if and only if all statements therein are consistent and not redundant. In our domain representation, only consistent ACLs are considered as annotation values. Intuitively, an annotation value specifies which entities have read permission to the triple, or are denied access when the annotation is preceded by $\neg$.

▶ **Example 8** (Access Control List). Assume a set of entities $\mathbf{C} = \{jb, js, hr, it\}$, where jb and js are employee usernames and hr and it are shorthand for *humanResources* and *informationTechnology*, respectively. The following annotated triple:

$$\tau : [[it], [hr, \neg js]]$$

states that the entities identified with it or hr (except if the js credential is also present) have read access to the triple τ.

An ACL A can be considered as a non-recursive Datalog with negation (nr-datalog$^\neg$) program, where each of the access control statements $S \in A$ corresponds to the body of a rule in the Datalog program. The head of each Datalog rule is a reserved element $access \notin \mathbf{CT}$ and the evaluation of the Datalog program determines the access permission to a triple given a specific set of credentials. The set of user credentials is assumed to be provided by an external authentication service and consists of elements of $\mathbf{CT}$ which equates to a non-empty ACL representing the entities associated with the user. As expected, we assume that this ACL consists of only one positive statement, i.e. the ACL will contain one statement with all the entities associated with the user and does not contain any negative elements.

▶ **Example 9** (Datalog Representation of an ACL). Taking into account the annotation example presented in Example 8. The nr-datalog$^\neg$ program corresponding to the ACL $[[it], [hr, \neg js]]$ is:

$access \leftarrow it.$
$access \leftarrow hr, \neg js.$

The set of credentials of the user *session*, provided by the external authentication system eg. $[[js, it]]$, are facts in the nr-datalog$^\neg$ program.

---

[2] In these examples, the default URI prefix is `http://urq.deri.org/enterprise#`.

[3] Here we are using $\neg e$ to represent strong negation. In our access control domain representation, $\neg e$ indicates that e will be specifically *denied* access.

Further domain specific information, for example the encoding of hierarchies between the credential elements, can be encoded as extra rules within the nr-datalog$^{\neg}$ program. These extra rules can be used to provide *implicit* credentials to a user, allowing the access control to be specified based on credentials that the authentication system does not necessarily assign to a user.

▶ **Example 10** (Credential Hierarchies)**.** If the entity *emp* represents all the employees within a specific company, and that jb and js correspond to employee usernames (as presented in Example 8), the following rules can be added to the nr-datalog$^{\neg}$ program from Example 9:

$emp \leftarrow js.$
$emp \leftarrow jb.$

These rules ensure that both jb and js are given access when the credential *emp* is required in an annotation value.

These rules can be used not only to express hierarchies between entities but any form of nr-datalog$^{\neg}$ rules are allowed.

3.2 Annotation Domain

We now turn to the annotation domain operations $\otimes$ and $\oplus$ that, as presented in Section 2.2, allow for the combination of annotation values catering for RDFS inferences. A naive implementation of these domain operations may produce ACLs which are not consistent (and would not be considered valid annotation values). To avoid such invalid ACLs, we rely on a normalisation step that ensures the result is a valid annotation value by checking for redundant statements and applying a conflict resolution policy if necessary. If an annotation statement contains a positive and negative access control element for the same entity, e.g $[jb, \neg jb]$, there is a *conflict*. There are two different ways to resolve conflicts in the annotation statements: (i) apply a *brave conflict resolution* (allow access); or (ii) *safe conflict resolution* (deny access). This is achieved during the normalisation step, through the *resolve* function, by removing the appropriate element: $\neg jb$ for brave or jb for safe conflict resolution. In our current modelling, we are assuming safe conflict resolution. The normalisation process is defined as follows:

▶ **Definition 11** (Normalise)**.** Let A be an ACL. We define the reduction of A into its consistent form, denoted $norm(A)$, as:

$$normalise(A) = \{resolve(S_i) \mid S_i \in A \text{ and } \nexists S_j \in A, i \neq j \text{ such that } S_i \leq S_j\} \quad .$$

The $\oplus$ operation is used to combine annotations when the same triple is deduced from different inference steps (cf. Rule (2)). For the access control domain, the $\oplus_{ac}$ operation involves the union of the annotations and the subsequent normalisation operation. The result of this operation intuitively creates a new nr-datalog$^{\neg}$ program consisting of the union of all the rules from the original nr-datalog$^{\neg}$ programs. Formally,

$$A_1 \oplus_{ac} A_2 = normalise\,(A_1 \cup A_2) \quad .$$

The following example presents an application of the $\oplus_{ac}$ operation:

▶ **Example 12** ($\oplus_{ac}$ operation)**.** Consider the triples $\tau_1 = (\texttt{:johnSmith}, \texttt{:salary}, \texttt{40000})\colon [[js]]$ and $\tau_2 = (\texttt{:johnSmith}, \texttt{:salary}, \texttt{40000})\colon [[hr]]$. Combining these triples with the $\oplus_{ac}$ operation (by applying Rule (2)) should result in providing access to all the entities which are allowed to access the premises:

$$(\texttt{:johnSmith}, \texttt{:salary}, \texttt{40000})\colon [[js], [hr]] \quad .$$

In turn, the $\otimes$ operation is used when inferring new triples, with the application of Rule (1), and for the access control domain, this operation ($\otimes_{ac}$) consists of merging the rules belonging to both annotation programs and then performing the normalisation and conflict resolution. This equates to restricting access to inferred statements to only those entities that have access to the both the original statements. Thus, the $\otimes$ operation corresponds to:

$$A_1 \otimes_{ac} A_2 = \mathit{normalise}\left(\{S_1 \cup S_2 \mid S_1 \in A_1 \text{ and } S_2 \in A_2\}\right) \ ,$$

where $S_1 \cup S_2$ represents the set theoretical union. Unlike the $\oplus_{ac}$ operation, the $\otimes_{ac}$ may produce conflicts in the annotation statements. For example, the application of the $\otimes_{ac}$ operation with the Annotated RDFS dom rule is as follows:

▶ **Example 13** ($\otimes_{ac}$ operation)**.** Let $\tau_1 = (\texttt{:westportCars}, \texttt{:netIncome}, \texttt{1000000})\colon [[hr, \neg jb]]$ and $\tau_2 = (\texttt{:netIncome}, \mathsf{dom}, \texttt{:Company})\colon [[it, jb]]$ be triples. The annotation resulting from applying the $\otimes_{ac}$ operation should provide access to the resulting triple only to entities which are allowed to access all the premisses. Thus we can infer, not only that `:westportCars` is of type `:Company`, but also the appropriate annotation value:

$$(\texttt{:westportCars}, \mathsf{type}, \texttt{:Company})\colon [[hr, it, \neg jb]] \ .$$

Please note that the aforementioned conflict resolution mechanism simplifies $[\neg jb, jb]$ to $[\neg jb]$.

Lastly, the smallest and largest annotation values in the access control domain, $\bot_{ac}$ and $\top_{ac}$ respectively, correspond in turn to an empty nr-datalog$^\neg$ program and another that provides access to all entities $e \in \mathbf{CT}$: $\bot_{ac} = []$ and $\top_{ac} = [[]]$. The $\bot_{ac}$ annotation value element indicates that the annotated triple is not accessible to any entity, since no annotation statements will provide access to the triple, and an annotation value of $\top_{ac}$ states that the triple is *public*, since any credential contained in the user session will trivially provide access to the triple. Intuitively, the $\top_{ac}$ annotation is translated into the nr-datalog$^\neg$ program containing only the "access" fact, while $\bot_{ac}$ corresponds to an empty program. However, for practical reasons, it might be necessary to assume a "super-user" role, for example represented as the reserved element "su", which will be allowed access to all triples and therefore would be used as the $\bot_{ac}$ annotation.

▶ **Definition 14** (Access Control Annotation Domain)**.** Let $\mathbf{F}$ be the set of annotation values over $\mathbf{CT}$, i.e. consistent ACLs. The access control annotation domain is formally defined as: $D_{ac} = \langle \mathbf{F}, \oplus_{ac}, \otimes_{ac}, \bot_{ac}, \top_{ac} \rangle$.

The presented modelling of the access control domain can be easily extended to handle other permissions, like *update*, and *delete* by representing the annotation as an n-tuple of ACL $\langle P, Q, \ldots \rangle$, where P specifies the formula for *read* permission, Q for *update* permission, etc. In this extended domain modelling, the domain operations can also be extended to operate over the corresponding elements of the annotation tuple. A *create* permission has a different behaviour as it would not be attached to any specific triple but rather as a graph-wide permission and thus is out of scope for this modelling. In this paper, we are considering only *read* permissions in the description of the domain and thus restrict the modelling to a single access control list. It is worth noting that the support for *create* and *update* of RDF is only included in the forthcoming W3C SPARQL 1.1 Recommendation [8].

3.3 Prolog Implementation

Considering the prototype described in Section 2.4, the implementation of the access control annotation domain consists of a Prolog module that is imported by the reasoner. This

```
@prefix : <http://urq.deri.org/enterprise#> .
:westportCars rdf:type :Company  "[[jb]]".
:westportCars :netIncome 1000000 .
:joeBloggs :worksFor :westportCars .
:joeBloggs :salary 80000 "[[jb]]".
:johnSmith :worksFor :westportCars .
:johnSmith :salary 40000 "[[js]].
```

Figure 2 RDF triples annotated with access control permissions.

module defines the domain operations $\otimes_{ac}$ and $\oplus_{ac}$, represented as the predicates `infimum/3` and `supremum/3` respectively. The annotation values are represented by using *lists* (in this case lists of lists), following the notions presented in the previous section.

The implementation of the $\oplus_{ac}$ operation involves concatenating the list representation of both annotations and then performing the normalisation operation. As for the $\otimes_{ac}$ operation, we follow a similar procedure to the $\oplus_{ac}$ operation, with the additional step of applying either the *brave* or the *safe* conflict resolution method. The evaluation of the nr-datalog$^\neg$ program can be performed based on the representation of the annotation values, by checking if the list of credentials of a user is a superset of any of the positive literals of the statements of our annotation values and also that it does not contain any of the negative literals of the statement.

An example of RDF data annotated with access control information is presented in Figure 2, where the salary information is only available to the respective employee. In this figure we are representing the RDF triples and annotation element using the NQuads RDF serialisation.[4] Using AnQL, the extension of the SPARQL query language described in Section 2.3, it is possible to perform queries that take into consideration the access control annotations. An example of an AnQL query over this data is presented in the following example:

▶ **Example 15** (AnQL Query Example). This query specifies that we are interested in the salary of employees that someone with the permissions $[[jb, hr, it]]$ is allowed to access.

```
SELECT * WHERE { ?p :salary ?s "[[jb, hr, it]]" }
```

The answers for this query (when matched against the data from Figure 2) under SPARQL semantics, i.e. if the annotation was omitted, would be:

$$\{\{?\mathtt{p} \rightarrow \mathtt{:joeBloggs}, ?\mathtt{s} \rightarrow 80000\}, \{?\mathtt{p} \rightarrow \mathtt{:johnSmith}, ?\mathtt{s} \rightarrow 40000\}\} \ .$$

However, when the domain annotations are present, an AnQL query engine must also perform the following check: $[[jb, hr, it]]$ satisfies the nr-datalog$^\neg$ program λ, where λ is the program represented by the annotation of each matched triple, thus yielding only the following answer:

$$\{\{?\mathtt{p} \rightarrow \mathtt{:joeBloggs}, ?\mathtt{s} \rightarrow 80000\}\} \ .$$

[4] http://sw.deri.org/2008/07/n-quads/

4 Related Work

The topic of access control has been long studied in relational databases and the approach of enforcing access policies by query rewriting was also considered for the Quel query language by [19]. However, the presented system does not rely on annotating the relational data but rather access control is specified using constraints over the user credentials which are then included in the rewritten query. A good overview of common issues, existing models and languages for access control is provided by [5], who focus on topics also discussed in this paper such as user hierarchy, allowing and denying access and conflict resolution.

For the Semantic Web, well known policy languages such as KAoS [3], Rei [12] and PROTUNE [2] are based on logical formalisms and consequently have well defined semantics. Although such policy languages enable policy specification using semantic web languages in their current form, they do not support reasoning based on RDF data relations.

In contrast, [11], [17], and [1] propose access control models for RDF graphs and like us allow for policy propagation and inference based on semantic relations. The policy language proposed by [11] is not based on well defined semantics and no implementation details are provided. [17] propose a path-based approach to policy composition. [1] state that they use an analytical tableaux system, however they do not provide a mechanism for merging or for inference of permissions based on RDF structure.

[6] describe the requirements an RDF store needs from a Semantic Wiki perspective. Apart from efficiency and scalability, the authors refer to the need for access control on a triple level and to integrate the structure of the organisation in the access control methods. The described system relies on a query engine (SPARQL is mentioned but no details are given) and a rule processor to decide the access control enforcement at query time. [10] present the possibility of maintaing metadata for RDF to enforce access control and touch upon of the work presented here, such as using rules for specifying access control, as possible extensions of their model. Providing access control on a resource level is also left as an open question, one we are tackling by the specification of rules.

5 Conclusions and Future Work

The Resource Description Framework (RDF) can be used for large scale integration of information from existing LOB applications. In this paper, we propose an access control model that can be used to protect RDF data and demonstrate how a combination of Annotated RDF and SPARQL can be used to control access to integrated enterprise data. Our model is based on the previously proposed Annotated RDF framework and attaches the access control information on a triple basis i.e. each RDF triple can contain different annotation values. The proposed solution provides a flexible representation method for the access control annotations, based on access control rules that define which entities have access to the triple. However, on very large datasets, challenges will arise with respect to optimal access control policy administration. To tackle this issue we propose managing permissions by specifying domain-specific inference rules for the annotation domain. We also suggest a possible implementation structure for a framework to enforce the access control based on rewriting a SPARQL query into an Annotated SPARQL query (AnQL) which relies on a secure authentication service.

Our initial work touches on how rules can be used to simplify the management of RDF access control permissions. In future work, we propose to investigate the interdependencies between usernames, groups, roles, and attributes and how we can further exploit the RDF graph structure to streamline the management of RDF access control policies. Although the

modelling presented in this paper provides a suitable representation model for the annotation values, its implementation and evaluation for large RDF graphs remains an open issue. To provide acceptable query performance when compared to its non-annotated counterpart, different optimisation strategies for both annotation storage and query evaluation will be necessary.

Acknowledgements. This work is supported in part by the SFI under Grant No. SFI/08/CE/I1380 (Líon-2), the IRCSET EPS and Storm Technology Ltd. We would like to thank Gergely Lukácsy, Aidan Hogan, and Umberto Straccia for their comments on this paper.

References

1. M. Amini and R. Jalili. Multi-level authorisation model and framework for distributed semantic-aware environments. *IET Information Security*, 4(4):301, 2010.
2. P.A. Bonatti, J.L. De Coi, Daniel Olmedilla, and Luigi Sauro. Rule-based policy representations and reasoning. In *Semantic techniques for the web*, pages 201–232, 2009.
3. J.M. Bradshaw, Stewart Dutfield, Pete Benoit, and J.D. Woolley. KAoS: Toward an industrial-strength open agent architecture. In *Software Agents*, pages 375–418, 1997.
4. Dan Brickley and R.V. Guha. RDF Vocabulary Description Language 1.0: RDF Schema. W3C Recommendation, W3C, February 2004. Available at `http://www.w3.org/TR/rdf-schema/`.
5. Sabrina De Capitani di Vimercati, Pierangela Samarati, and Sushil Jajodia. Policies, Models, and Languages for Access Control. In Subhash Bhalla, editor, *Databases in Networked Information Systems, 4th International Workshop, DNIS 2005, Aizu-Wakamatsu, Japan, March 28-30, 2005, Proceedings*, volume 3433, pages 225–237. Springer, 2005.
6. Sebastian Dietzold and Sören Auer. Access Control on RDF Triple Stores from a Semantic Wiki Perspective. In Chris Bizer, Sören Auer, and Libby Miller, editors, *Proc. of 2nd Workshop on Scripting for the Semantic Web at ESWC, Budva, Montenegro.*, volume 183, June 2006.
7. Claudio Gutierrez, Carlos A. Hurtado, and Alejandro A. Vaisman. Introducing Time into RDF. *IEEE Transactions on Knowledge and Data Engineering*, 19(2):207–218, February 2007.
8. Steve Harris and Andy Seaborne. SPARQL 1.1 Query Language. W3C working draft, W3C, January 2012. Available at `http://www.w3.org/TR/2012/WD-sparql11-query-20120105/`.
9. Patrick Hayes. RDF Semantics. W3C Recommendation, W3C, February 2004. Available at `http://www.w3.org/TR/rdf-mt/`.
10. James Hollenbach, Joe Presbrey, and Tim Berners-Lee. Using RDF Metadata To Enable Access Control on the Social Semantic Web. In Tania Tudorache, Gianluca Correndo, Natasha Noy, Harith Alani, and Mark Greaves, editors, *Proceedings of the Workshop on Collaborative Construction, Management and Linking of Structured Knowledge (CK2009)*, volume 514. CEUR-WS.org, 2009.
11. S Javanmardi, M Amini, R Jalili, and Y. GanjiSaffar. SBAC: A Semantic Based Access Control Model. In *11th Nordic Workshop on Secure IT-systems (NordSec'06), Linkping, Sweden*, 2006.
12. L. Kagal and T. Finin. A policy language for a pervasive computing environment. In *Proceedings POLICY 2003. IEEE 4th International Workshop on Policies for Distributed Systems and Networks*, pages 63–74. IEEE Comput. Soc, 2003.
13. Michael Kifer and V. S. Subrahmanian. Theory of Generalized Annotated Logic Programming and its Applications. *J. Log. Program.*, 12(3&4):335–367, 1992.

14 Nuno Lopes, Axel Polleres, Umberto Straccia, and Antoine Zimmermann. AnQL: SPARQLing Up Annotated RDF. In *Proceedings of the International Semantic Web Conference (ISWC-10)*, number 6496 in LNCS, pages 518–533. Springer-Verlag, 2010.

15 Frank Manola and Eric Miller. RDF Primer. W3C Recommendation, `http://www.w3.org/TR/rdf-primer/`, W3C, February 2004.

16 Jorge Pérez, Marcelo Arenas, and Claudio Gutiérrez. Semantics and complexity of SPARQL. *ACM Transactions on Database Systems*, 34(3):1–45, 2009.

17 Tatyana Ryutov, Tatiana Kichkaylo, and Robert Neches. Access Control Policies for Semantic Networks. In *2009 IEEE International Symposium on Policies for Distributed Systems and Networks*, pages 150–157. IEEE, July 2009.

18 Andy Seaborne and Eric Prud'hommeaux. SPARQL Query Language for RDF. W3C Recommendation, W3C, January 15 2008. Available at `http://www.w3.org/TR/rdf-sparql-query/`.

19 Michael Stonebraker and Eugene Wong. Access control in a relational data base management system by query modification. In *Proceedings of the 1974 annual conference - Volume 1*, ACM '74, pages 180–186, New York, NY, USA, 1974. ACM.

20 Umberto Straccia. A Minimal Deductive System for General Fuzzy RDF. In Axel Polleres and Terrance Swift, editors, *RR*, volume 5837, pages 166–181. Springer, 2009.

21 Umberto Straccia, Nuno Lopes, Gergely Lukacsy, and Axel Polleres. A General Framework for Representing and Reasoning with Annotated Semantic Web Data. In Maria Fox and David Poole, editors, *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010, Atlanta, Georgia, USA, July 11-15, 2010*. AAAI Press, July 2010.

22 Herman J. ter Horst. Completeness, decidability and complexity of entailment for RDF Schema and a semantic extension involving the OWL vocabulary. *J. Web Sem.*, 3(2-3):79–115, 2005.

23 Octavian Udrea, Diego Reforgiato Recupero, and V. S. Subrahmanian. Annotated RDF. *ACM Trans. Comput. Logic*, 11(2):1–41, 2010.

24 Jan Wielemaker, Zhisheng Huang, and Lourens van der Meij. SWI-Prolog and the Web. *Theory and Practice of Logic Programming*, 8(3):363–392, 2008.

25 Antoine Zimmermann, Nuno Lopes, Axel Polleres, and Umberto Straccia. A general framework for representing, reasoning and querying with annotated Semantic Web data. *Web Semantics: Science, Services and Agents on the World Wide Web*, 11(0):72 – 95, 2012.

LOG-IDEAH: ASP for Architectonic Asset Preservation

Viviana Novelli[1], Marina De Vos[2], Julian Padget[2], and Dina D'Ayala[1]

1 **Department of Architecture and Civil Engineering**
University of Bath, BA2 7AY
Bath, UK
E-mail: `{v.i.novelli,d.f.d'ayala}@bath.ac.uk`

2 **Department of Computer Science**
University of Bath, BA2 7AY
Bath, UK
E-mail: `{mdv,jap}@cs.bath.ac.uk`

Abstract

To preserve our cultural heritage, it is important to preserve our architectonic assets, comprising buildings, their decorations and the spaces they encompass. In some geographical areas, occasional natural disasters, specifically earthquakes, damage these cultural assets. Perpetuate is a European Union funded project aimed at establishing a methodology for the classification of the damage to these buildings, expressed as "collapse mechanisms". Structural engineering research has identified 17 different collapse mechanisms for masonry buildings damaged by earthquakes. Following established structural engineering practice, paper-based decisions trees have been specified to encode the recognition process for each of the various collapse mechanisms. In this paper, we report on how answer set programming has been applied to the construction of a machine-processable representation of these collapse mechanisms as an alternative for these decision-trees and their subsequent verification and application to building records from L'Aquila, Algiers and Rhodes. As a result, we advocate that structural engineers do not require the time-consuming and error-prone method of decisions trees, but can instead specify the properties of collapse mechanisms directly as an answer set program.

1998 ACM Subject Classification D.1.6 Logic Programming

Keywords and phrases Answer set programming, structural engineering, knowledge representation

Digital Object Identifier 10.4230/LIPIcs.ICLP.2012.393

1 Introduction

It is useful to be able to make a rapid and reliable assessment of the vulnerability of a building after a seismic shock, to determine (i) the stability of the building and hence the acceptable proximity of public access (ii) what immediate preservation actions are appropriate, and (iii) potential risks from subsequent seismic activity.

The Perpetuate project aims to combine two approaches in order to deliver assessments with a higher degree of confidence. The one on which we report here takes the form of an expert survey, based on an approach called LOG-IDEAH (LOGic trees for the Identification of Damage due to Earthquakes for Architectural Heritage) [10]. The complementary mechanical model-based approach is called FaMIVE (Failure Mechanism Identification and Vulnerability Evaluation) [2, 3, 4]. Both procedures are aimed at identifying the seismic

© Viviana Novelli, Marina De Vos, Julian Padget, and Dina D'Ayala;
licensed under Creative Commons License ND
Technical Communications of the 28th International Conference on Logic Programming (ICLP'12).
Editors: A. Dovier and V. Santos Costa; pp. 393–403

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

behaviour of an architectonic asset on the basis of data collected by rapid survey or by photographic observation. The difference between the two methodologies is that LOG-IDEAH is an intuitive (human) logic procedure that relies on seismic damage collection for the identification of the failure modes of the architectonic assets, while FaMIVE is a numerical approach which uses geometric and mechanical (based on the properties of the building materials involved) data for the calculation of the performance of historical buildings.

The remainder of the paper is structured as follows: (i) we next (section 2) provide the context for the application, introduce the necessary domain terminology used in the rest of the paper and explain the hierarchical approach to the identification of individual building elements that forms the basis for the constructive procedure for the recognition of collapse mechanisms (ii) we then set out (section 3) how this damage and asset data is used to determine the possible collapse mechanisms; we start from the traditional structural engineering method of decision trees; this is followed by a much easier, declarative and computational approach of representing the building data and requirements for the individual collapse mechanisms as an answer set program (iii) in section 4 we briefly describe the data capture mechanism used to acquire the data to be used by the analysis, and (iv) in section 5 review the process and outline plans for future development.

2 Architectonic Asset Analysis

LOG-IDEAH depends upon an hierarchical approach, in which the architectonic asset is deconstructed into façades, the façades into structural elements and the structural elements into artistic assets. On the basis of this hierarchical approach, a logical methodology for the acquisition of the data has been developed in order to collect seismic damage data on site or by photographic observation.

Data collection for LOG-IDEAH entails the recording of information related to the damage position, damage type and damage level, that are observed at the level of the structural elements and artistic assets of the architectonic asset under inspection.

The collected data is then interpreted by means of logic trees that represent the knowledge and expertise of structural engineers, as they would use it for the identification of the global behaviour of an architectonic asset, and to recognise the failure modes of the architectonic asset in question.

2.1 Ontology

As with all disciplines, a comprehensive ontology[1] has been developed to capture and precisely define domain concepts and their relationships. Because the description of the analysis process is necessarily expressed in terms of this ontology, we give a brief overview of the elements required for reading the remainder of this paper. A formal representation in OWL has been developed, but that is not the subject of this paper and is not used directly in what follows. There are four top-level concept classes:

Architectonic asset (AA): This covers seven classes of buildings (A, ..., G) from mansions, through mosques, aqueducts, city walls and obelisks to historical centres (such as L'Aquila, in which the case-study building treated here is located).

[1] While using the word ontology we are not referring to semantic annotation in terms of an XML description, but an establied set of related concepts in the field of structural engineering

Macro-element (ME): This is a set of abstract concepts, used to group structural elements, comprising four classes: vertical, horizontal, vaulted and staircases.

Structural element (SE): This comprises four groups, corresponding to the MEs above, of concrete classes such as piers and spandrels (vertical), rafters and tie-beams (horizontal), buttresses and bosses (vaulted) and cantilever and steps (staircases).

Artistic asset (AA): This is a set of three groups of three abstract classes, used to categorize the various forms of decoration that may be attached to a structural element. Fresco's, friezes are just two examples of artistic assests.

2.2 Localisation and Identification of Damage

The essence of the the survey approach is to construct a rectilinear map of each façade, based on the structural elements (SE) of which it is composed, label each SE by type (pier, pillar, spandrel, arch) and associate with it the kind of damage (vertical crack, horizontal crack, diagonal crack) and the degree of damage (light, severe, near collapse, collapsed). This data then forms the input to the encoding of the structural engineer's decision tree that identifies the collapse mechanism.

The zone under consideration (and the buildings within it) may be thought of as a n-ary tree, rooted at the zone identifier, with whichever district the building is located, whose leaves are, in the extreme, the artistic assets, such as a carving or a balcony, that decorate the building and the interior nodes are everything in between.

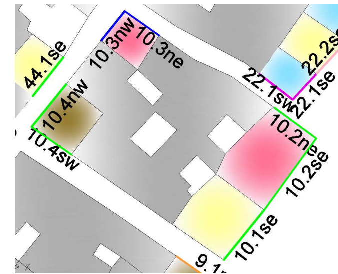
The ultimate objective, from an engineering perspective, is to establish a relationship between each artistic asset and the façade to which it is attached, since this is how it may be damaged if the façade is subject to a collapse mechanism. This is achieved by defining a hierarchical naming scheme. The method starts by dividing the urban map into blocks and enumerating the blocks and the buildings located therein. Thus, the name associated with a particular asset is given by (block number + building number), followed by the façade orientation. The final suffix is the number of the region (in a rectilinear map) of the façade plus a letter that refers to the type of topological relationship between the asset and the façade. For example (see Figure 1), in the map of the historical centre of L'Aquila – damaged by an earthquake in 2009 – the building highlighted in red is named 10.4, since this building is the fourth building of the block number 10. Furthermore, a sequence of façades is associated with this building, describing those which have been inspected, namely 10.4sw and 10.4nw, while those on the remaining sides are still to be inspected.

The relationship between façades and the structural elements is established after identifying vertical (piers/pillars/columns) and horizontal (spandrels/arches) structural elements of the façade, as shown in Figure 1. Each structural element is labelled by a pair of positive integers, where the first is the number of the floor, encoding the horizontal alignment of the elements and the second is the position of the element, encoding the vertical alignment. Thus, looking at Building 10.4 (Figure 1d) and façade 10.4sw, the identification of its structural elements is clear.

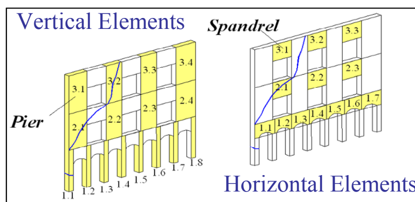
Once these relationships have been established, seismic damage information at the level of SEs is collected, then interpreted, first at the level of MEs (for example, façade) and then at the level of AA (building).

The damage type classification is given by: (i) H, denoting a horizontal crack (ii) V, a vertical crack (iii) D1, a diagonal crack from upper right to bottom left (iv) D2, a diagonal crack from upper left to bottom right (v) X, being the occurrence of D1 and D2 in the same structural element (vi) Sp, denoting spalling, which indicates surface fragmentation of the (building) material, and (vii) Cr, denoting crushing, which indicates interior fragmentation.

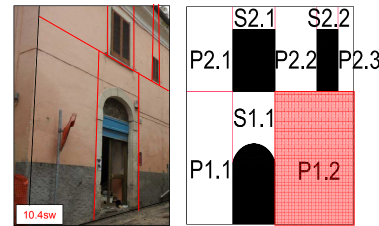
(a) The division of L'Aquila into districts



(b) Identification of the façades on individual buildigs in district 10



(c) The structural enumeration convention



(d) Facade sw of building 10.4 is broken into structural elements to enable reference to pier 2 on floor 1

Figure 1 The division of L'Aquila into districts, façade identification on individual buildings in district 10, the structural enumeration convention and the referencing of pier 2 on floor 1.

The damage level severity classification is given by: (i) LD: light damage (ii) SD: severe damage (iii) NC: near collapse, and (iv) C: collapsed.

All of the above is carried over directly into the ASP encoding (see Figure 4).

3 Representation and Reasoning

3.1 Decision Trees

One of the primary deliverables of the Perpetuate project is a process for determining the various collapse mechanisms for earthquake-damaged masonry buildings, by examining the damage to the structural elements and the artistic assets of the building. So far, the architects have identified 19 distinct such mechanisms, each of which is depicted in Figure 2.

The original (manual) method developed for identifying the collapse mechanism uses the traditional structural engineering approach of decision trees. Figure 3 shows the decision tree for collapse mechanism A. This mechanism occurs when there are vertical cracks on either side of a façade starting from the top floor. The graphics on the right highlight on which floor the decision tree is operating at a given time.

Such a manual approach, which also requires specialized knowledge on the part of the observer, is not very efficient at scale for dealing with an earthquake zone where several hundreds of buildings are damaged, like for example the sites involved in the Perpetuate Project: L'Aquila and the Casbah of Algiers. This led to the requirement for a computational mechanism to support the survey process by non-experts. Given the intrinsically procedural nature of decision trees, a procedural programming approach could have been chosen. However, given the declarative description of the collapse mechanisms (e.g. vertical

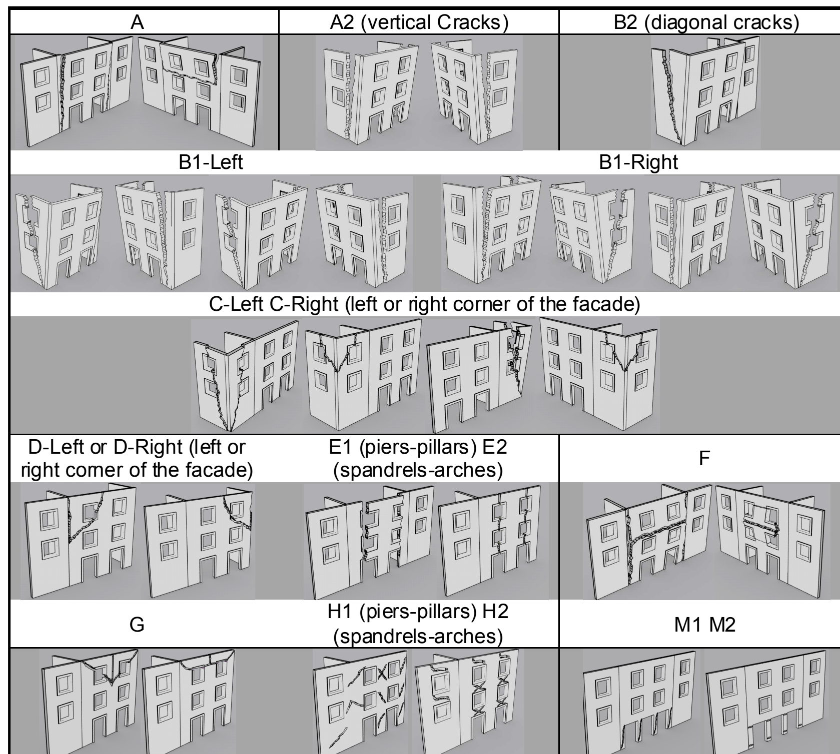


Figure 2 Illustration of the variety of collapse mechanisms.

cracks on either side of a façade) provided by the architects on the project, we believed the declarative paradigm would be more suitable. In addition to providing a computational model, it also allowed us to verify and validate the decision trees provided by the arhcitects. We have chosen to implement the collapse mechanism inference procedure and the description of the buildings using answer set programming[8, 9] with *AnsProlog* as the implementation language[1].

3.2 Answer Set Programming

Answer-set programming (ASP) [8] is a declarative programming paradigm in which a logic program is used to describe the requirements that must be fulfilled by the solutions of a certain problem. The solutions of the problem can be obtained through the interpretation of the answer sets of the program, usually defined through a variant or extension of the stable-model semantics [8].

In this paper we use *AnsProlog* as our implementation language. Its basis component are atoms, elements that can be either true or false. An atom a can be negated using *negation as failure*. A *literal* is an atom a or a negated atom *not* a. We say that *not* a is true if we cannot find evidence supporting the truth of a. Using atoms and literals as building blocks we can create rules. In their general form they are represented as:

$$\mathtt{a} :- \mathtt{b}_1, \ldots, \mathtt{b}_m, \mathtt{not}\ \mathtt{c}_1, \ldots, \mathtt{not}\ \mathtt{c}_n.$$

where a, b_i, and c_j are atoms. Intuitively, this can be read as: *if all atoms b_i are known/true and no atom c_i is known/true, then a must be known/true.*

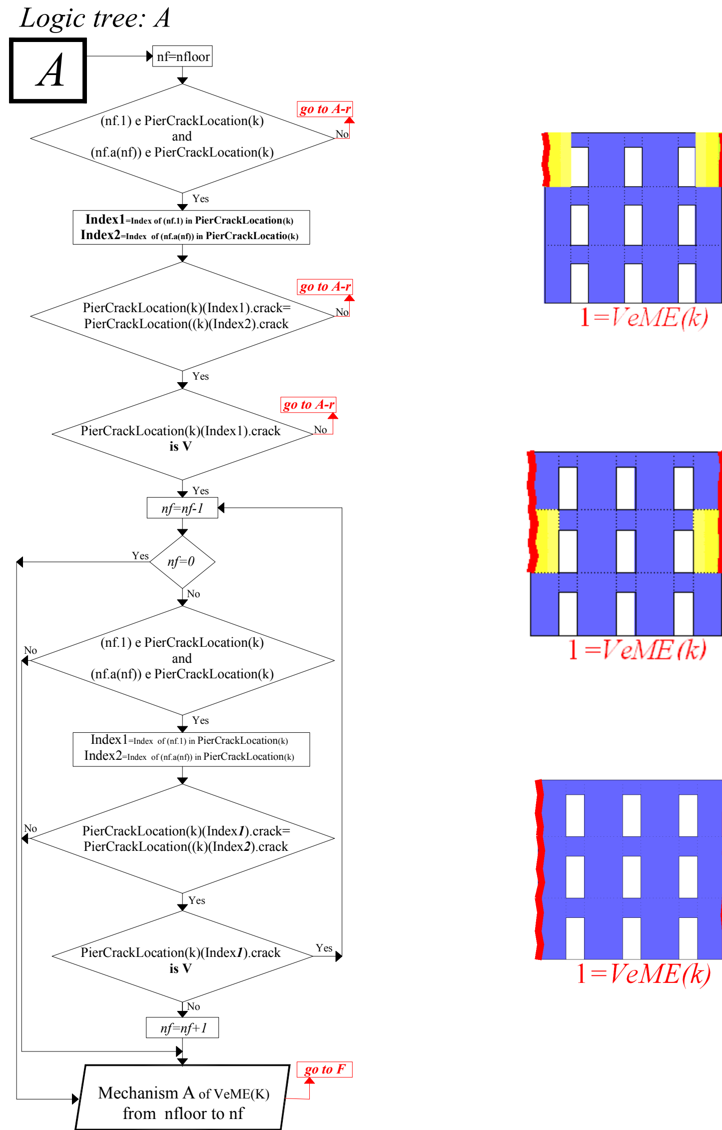


Figure 3 The logic tree for mechanism A (left) and sketches (right).

We refer to a as the head and $b_1, \ldots, b_m, not\ c_1, \ldots, not\ c_n$ as the body of the rule. Rules with empty bodies are called *facts*. Rules with empty heads are referred to as *constraints*, indicating that no solution should be able to satisfy the body. A *program* is a set of rules. The semantics is defined in terms of *answer sets*, i.e. assignments of true and false to all atoms in the program that satisfy the rules in a minimal and consistent fashion, taking into account that the truth of an atom cannot be based on the absence of proof (i.e. the truth of an atom cannot indirectly be inferred by its own negation). A program has zero or more answer sets, each corresponding to a solution.

Algorithms and implementations for obtaining answer sets of logic programs are referred to as *answer-set solvers*. The most popular and widely used solvers are DLV [6], providing solver capabilities for disjunctive programs, and the SAT inspired CLASP [7].

```
% @block buildingconstants {
%       provides the constants used in damage description of buildings
% @atom damageType(T)
%       type of damage from vertical;horizontal;diagonal crack / ;
%       diagonal crack \ ;
%       x shape;spalling;crushing
% @atom damageLevel(L)
%       severity of damage from damage limitation;significant damage;
%       near collapse;collapse

damageType(v;h;d1;d2;x;s;cr).
damageLevel(ld;sd;nc;c).
```

Figure 4 The facts describing the damage types and levels of buildings.

3.3 *AnsProlog* for Collapse Mechanisms

Instead of using the procedural decision-trees as our starting point, we used the sketches of the collapse mechanisms together with a discussion with a domain expert as our starting point. Since the answer set program will be integrated with web-interface (see next section) that collects data for each building individually, our answer set program only needs to consider the representation of a single building at any given time.

To start the modelling process, it was necessary to decide upon the representation of the various structural elements of the building. With the objective of making the logic accessible to the architects, we annotated all our program fragments with a description of the atoms used. To do so, we used a subset of the annotation language LANA[5]. This language uses program comments plus semantic tags in the style of Javadoc t o describe the various components of the program. We only used the `@block` tag, indicating a collection of rules, and the `@atom` tag to describe individual atoms and their terms.

We first defined facts to denote the various damage types and damage levels. The encoding is shown in Figure 4.

This was followed with facts for the description of the structural elements themselves. The ultimate goal is that this information is automatically generated on the basis of information gathered by non-expert surveyors on site (see next section for more information on the data collection). For some of the collapse mechanisms, numerical information is required, such as the number of piers with a vertical crack or the percentage of piers that are damaged. Since this data can be generated during the data collection process, we choose to incorporate it as facts rather than compute it in the answer set program.

Figure 5 contains the description of a synthetic building with one visible façade which exhibits an out-of-plane collapse mechanism of type A.

Having the description of the building, the different collapse mechanisms can be encoded. Rather than using the rather procedural decision trees as a starting point, the encoding is derived solely from the schematic pictures (see right-hand side of Figure 5 for an example). Each collapse mechanism is encoded in a separate file and LANA block for ease of testing, flexibility and readability.

In this paper we illustrate this process with the intra-façade collapse mechanism A. To demonstrate the ease of use of the ASP-encoding, we subsequently extend this to the inter-façade mechanism A2.

Figure 6 shows the block of *AnsProlog* code that allows us to detect collapse mechanism A. Most encodings of collapse mechanisms, and mechanism A is not an exception, start from the top floor of a façade and try to identify a certain crack pattern. If found, lower floors are tested until a floor is found which does not have this desired pattern. The system will then return the specific pattern with the range of floors involved in the pattern. For the

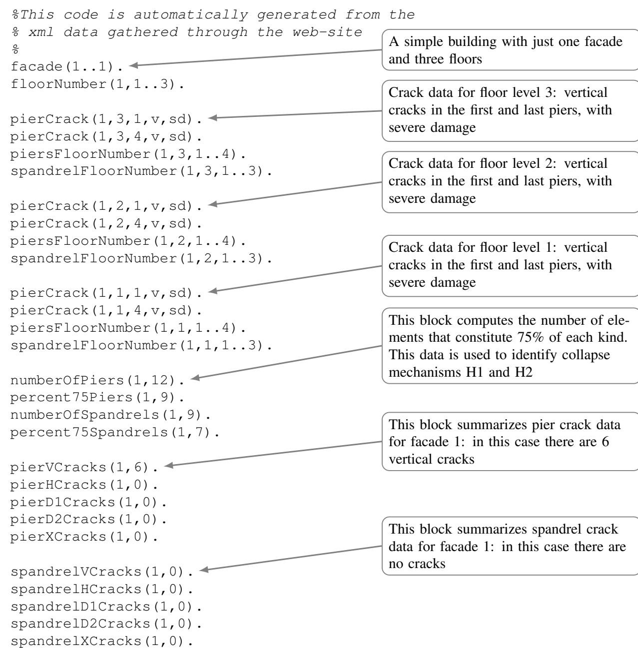


Figure 5 The encoding of a single façade exhibiting collapse mechanism A (out of plane).

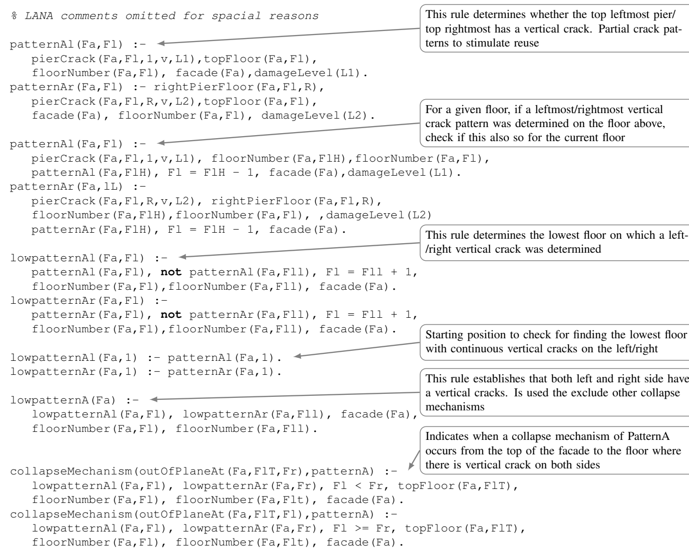


Figure 6 The rules used to recognise collapse mechanism A.

encoding, distinct parts of a sought crack pattern are tested separately (e.g. `patternAr` and `patternAl`) to support reuse for other mechanism using the same partial patterns. For each partial pattern, the lowest floor (e.g. `lowpatternAl(Fa, Fl)` and `lowpatternAr(Fa, Fl)`) where the partial patterns occurs, is determined to be combined to form the collapse mechanism (e.g. `collapseMechanism(outOfPlaneAt(Fa, FlT, Fr), patternAa)`).

The scenario described here in detail is one relatively simple type of collapse mechanism where, there are vertical cracks down both sides within a façade, leading to the collapse of the entire front of the building. Most collapse mechanisms are intra-façade, but there

```
% LANA comments omitted for spacial reasons.

lowpatternAa(Fa) :-
   lowpatternAl(FaR,Fl), lowpatternAr(FaL,Fll),   <- Uses atoms derived in scenarioA
   floorNumber(FaR,Fl),floorNumber(FaL,Fll),
   rightFacade(Fa,FaR),leftFacade(Fa,FaL),facade(Fa;FaR;FaL).

collapseMechanism(
      outOfPlanePortion(Fa,FlT,Fll),
      patternAb) :-   <- Uses atoms derived in scenarioA
   lowpatternAl(FaR,Fl), lowpatternAr(FaL,Fll),Fl<Fll,
   floorNumber(FaR,Fl),floorNumber(FaL,Fll),
   floorNumber(FaL,FlT),
   rightFacade(Fa,FaR),leftFacade(Fa,FaL),facade(Fa;FaR;FaL).

collapseMechanism(
      outOfPlanePortion(Fa,FlT,Fl),
      patternA2) :-
   lowpatternAl(FaR,Fl), lowpatternAr(FaL,Fll),Fll<=Fl,
   floorNumber(FaR,Fl),floorNumber(FaL,Fll),
   floorNumber(FaL,FlT),
   rightFacade(Fa,FaR),leftFacade(Fa,FaL),facade(Fa;FaR;FaL).
```

Figure 7 The A2 detection rules.

is a more complicated class of inter-façade mechanisms, such as illustrated by mechanism A2 (Figure mechanisms). This has much in common with mechanism A, in that there are vertical cracks on either side of the façade concerned, but those cracks are in the *adjacent* façades, and in consequence, the reasoning process must be applied not at the macro-element level, but across the architectonic asset as a whole. The relative ease with which is it possible to extend the analysis illustrates the benefit of the hierarchical approach to the representation of the building and the declarative nature of the encoding. In the encoding (Figure 7), we use reuse the partial patterns for vertical cracks on the left and right side of a façade, something which is not possible in the decision trees.

With all the 19 collapse mechanisms implemented, we can pass the building description, the encodings of each mechanism and some auxiliary files for computing, for example, the rightmost pier of a floor or the top floor of a façade, and for showing only the `collapsemechanism/2` to the answer set solver CLINGO[7] to determine which mechanism(s) are found for this building. Applying this to our synthetic Mechanism A Building, as encoded in Figure 5, the result is the identification of an out-of-plane collapse across three floors, as expected.

The verification process was approached by the creation of unit test cases for each of the 19 collapse mechanisms. While initially created manually, they were later re-created through the web-site described in the next section. Determining the collapse mechanisms for an entire building takes at most 1 or 2 seconds, which is signicantly faster than experienced human can manually go through the 19 decision trees.

4 Data Capture

The record of the collected seismic data has been facilitated by the use of a web-site[2] which permits expert and non-expert users to: (i) create of new architectonic asset records, effectively from anywhere (ii) draw simplified sketches of inspected façades (iii) record damage type and damage level to structural elements and artistic assets, (iv) upload photographic records of assets and asset damage, and (v) assess probable collapse mechanisms using the reasoning process described in the previous section.

Surveys are in progress and at this stage have collected records from buildings in L'Aquila and the Casbah in Algiers. Information regarding the buildings and their structural elements is stored in XML format, which can be converted to *AnsProlog* code, like in Figure 5. Integration between the Perpetuate web-site and LOG-IDEAH is currently in progress.

[2] `http://perpetuate.cs.bath.ac.uk/`

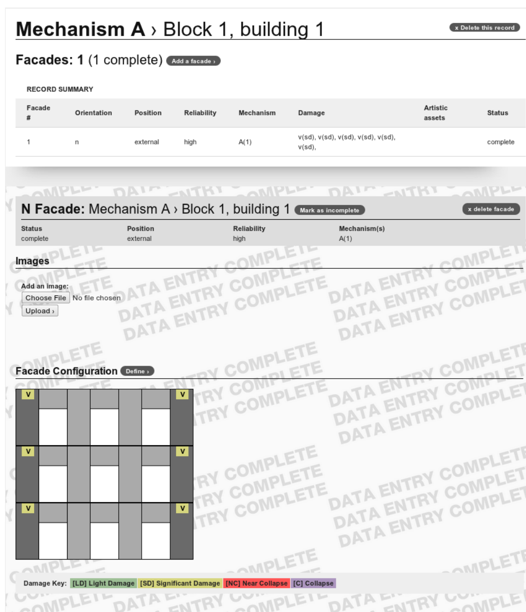


Figure 8 Extract from completed data entry in web browser.

5 Discussion and Future Work

In this paper, we introduced a computational representation and an alternative approach to the established structural engineering methodology of decision trees, which we believe is more efficient, flexible, intuitive and less error-prone. In the process of writing the answer set programs, a number of subtle errors were uncovered in the decision trees, as well as some oversights and shortcomings of building's representation, that would have been hard to find using the traditional pen-and-paper verification technique or a procedural implementation of the decision trees. Encoding the collapse mechanisms took us only a few days, including the time to get aquinted with the domain ontology. We hope in the future to apply a similar approach to other problem for which structural engineers use decision trees. By doing so, we hope to demonstrate ASP might be a good alternative to decision trees.

At the moment buildings are assumed to be relative regular for the purpose of continuous cracks. Piers and spandrels on different floors have similar size, lined-up are consecutive. The next version of the model and software will relax this constraint by using (structural) element as the basic component of a building rather pier or spandrel. In this way, wider piers can conceptually be encoded as three elements of type pier to allow for elements on several floors to be lined up.

The Perpetuate project is attracting attention from conservationists from across the world who would also like to enter building information into the repository. This will provide more data for architects to identify and specify more collapse mechanisms and to extend the approach to other types of buildings, such as stone, wood or earth.

Acknowledgements. This work is partially supported by the European Commission funded FP7 project PERPETUATE (ENV.2009.3.2.1.1). See `http://www.perpetuate.eu` for more information.

References

1. Chitta Baral. *Knowledge Representation, Reasoning, and Declarative Problem Solving.* Cambridge University Press, Cambridge, England, 2003.
2. D D'Ayala and E Speranza. Definition of collapse mechanisms and seismic vulnerability of historic masonry buildings. *Earthquake Spectra*, 19:479—509, August 2003.
3. D. F. D'Ayala. Force and displacement based vulnerability assessment for traditional buildings. *Bulletin Of Earthquake Engineering*, 3:235—265, December 2005.
4. Dina D'Ayala and Sara Paganoni. Assessment and analysis of damage in l'aquila historic city centre after 6th april 2009. *Bulletin of Earthquake Engineering*, 9:81–104, 2011. 10.1007/s10518-010-9224-4.
5. Marina De Vos, Doga Kiza, Johannes Oetsch, Jörg Pührer, and Hans Tompits. Annotating answer-set programs in lana. *Theory and Practice of Logic Programming*, 2012.
6. Thomas Eiter, Nicola Leone, Cristinel Mateis, Gerald Pfeifer, and Francesco Scarcello. The KR system dlv: Progress report, comparisons and benchmarks. In *Proceedings of the 6th International Conference on Principles of Knowledge Representation and Reasoning* (*KR 1998*), pages 406–417. Morgan Kaufmann, 1998.
7. M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. Conflict-driven answer set solving. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence* (*IJCAI 2007*), pages 386–392. AAAI Press/The MIT Press, 2007.
8. Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *Proceedings of the 5th International Conference and Symposium on Logic Programming*, pages 1070–1080. MIT Press, 1988.
9. Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9(3-4):365–386, 1991.
10. Viviana Novelli and Dina D'Ayala. Seismic damage identification of cultural heritage assets. In *Seismic Protection of Curtural Heritage*, page 13, Antalya, Turkey, 2011.

Extending $\mathcal{C}+$ with Composite Actions for Robotic Task Planning

Xiaoping Chen[1], Guoqiang Jin[1], and Fangkai Yang[2]

1 **School of Computer Science, University of Science and Technology of China**
2 **Department of Computer Science, University of Texas at Austin**

Abstract

This paper extends action language $\mathcal{C}+$ by introducing *composite actions* as sequential execution of other actions, leading to a more intuitive and flexible way to represent action domains, better exploit a general-purpose formalization, and improve the reasoning efficiency for large domains. Our experiments show that the composite actions can be seen as a method of knowledge acquisition for intelligent robots.

1998 ACM Subject Classification I.2.4 Knowledge Representation Formalisms and Methods

Keywords and phrases Reasoning about Actions, Action Languages, Robotic Task Planning

Digital Object Identifier 10.4230/LIPIcs.ICLP.2012.404

1 Introduction

The problem of describing changes caused by the execution of actions plays an important role in knowledge representation. Actions may be described

1. by specifying their preconditions and effects, as in STRIPS [5], PDDL-like languages, action languages such as $\mathcal{B}$ and $\mathcal{C}$ [7], $\mathcal{C}+$ [8], situation calculus [14];
2. in terms of execution of primitive actions, such as programs in GoLog [10], ASP [17], extended event calculus [16], ABStrips [15] and HTN [4]; or
3. as a special case of actions of more general kind, as in MAD [11] and $\mathcal{ALM}$ [6].

Actions formalized in the first and third approach are used to automate planning, and more generally, to automate commonsense reasoning tasks such as temporal projection and postdiction, with an emphasis on addressing the problem of generality in AI [12]. However, actions formalized in the second approach are usually used for complementary purposes: they are abstractions or aggregates that characterize the hierarchical structure of the domain and improve search efficiency. This paper extends action language $\mathcal{C}+$ with *composite actions* defined as sequential execution of other actions, and shows that these composite actions can be used for the purposes of the first and third approaches as well.

The extended $\mathcal{C}+$ has three advantages. First, it provides one more way to formalize actions in $\mathcal{C}+$. Second, composite actions can be defined by exploiting the general purpose formalization of actions, a step of addressing the problem of generality in AI, or by exploiting natural language information for knowledge acquisition. Third, composite actions can be used to characterize the hierarchical structure of problem and improve planning efficiency.

To achieve this goal, we add a new construct to $\mathcal{C}+$ that defines *composite actions* as sequential executions of actions $a_0, \dots a_k$ under conditions (written as formulas) $E_0, \dots E_k$. For instance, consider a domain of a robot with a hand which can deliver small objects from one place to another. The primitive actions represent the basic functions of the robot such

© Xiaoping Chen, Guoqiang Jin, and Fangkai Yang;
licensed under Creative Commons License ND

as `move`, `pickup`, `putdown`. We can define a composite action $Fetch(s, l)$ as the consecutive executing of the actions $Move(l_1)$ and $Bring(s, l)$

$$Fetch(s, l) \textbf{ is } Move(l_1) \textbf{ if } Loc(s) = l_1 \wedge Loc(Robot) \neq l_1; Bring(s, l).$$

However, to define the semantics of the construct is not a trivial task. In $\mathcal{C}+$, all actions are assumed to be executed over 1 time interval. This assumption affects the design of both description language and query language of CCalc[1]: descriptions of action domains don't involve formalizing time, leading to a concise representation; when formulating queries, time instances can be named explicitly and conveniently based on the assumption. In presence of composite actions, it is natural to talk about their lengths and how their lengths affect the design of the language. It happens that defining the length of a composite action in terms of the number of primitive actions it involves leads to a cumbersome query language, since the number is not fixed for a composite action: the length of $Fetch(s, l)$ depends on the location of the robot and s. Therefore, when formulating queries, the user may need to explicitly name the indefinite lengths of actions, which becomes complicated when doing distant projection, postdiction or planning. For simplicity of the syntax, we extend the assumption to composite actions so that it is fully compatible with CCalc input, but use a notion of *subintervals* in their semantics to characterize execution trajectories of composite actions.

The new language is implemented by modifying the software cplus2asp [1], which translates the input into an incremental answer set program and calls the solver iclingo[2]. We formalize a version of a service robot domain with composite actions, and show that composite actions can be used for knowledge acquisition, and improve planning efficiency for large problems.

The work presented in this paper is somewhat similar to [9] but composite actions defined there have fixed and explicitly specified length.

2 Preliminaries

The review of action language $\mathcal{C}+$ follows [8]. A (multi-valued) signature is a set σ of symbols, called (multi-valued) constants, along with a non-emtpy finite set $Dom(c)$ of symbols, disjoint from σ, assigned to each constant c. Each constant belongs to one of the three groups: *action* constants, *simple fluent* constants and *statically determined fluent* constants.

Consider a fixed multi-valued signature σ. An *atom* is an expression of the form $c = v$ ("the value of c is v") where $c \in \sigma$ and $v \in Dom(c)$. A *formula* is a propositional combination of atoms. An interpretation maps every constant in σ to an element of its domain. A formula is called *fluent formula* if it does not contain action constants, and *action formula* if it contains at least one action constant and no fluent constants.

An *action description* consists of a set of *causal laws* of the form

$$\textbf{caused } F \textbf{ if } G \tag{1}$$

where F and G are formulas. The rule is called *static law* if F and G are fluent formulas, or *action dynamic law* if F is an action formula; and rules of the form

$$\textbf{caused } F \textbf{ if } G \textbf{ after } H \tag{2}$$

where F and G are fluent formulas, and H is a formula, called *fluent dynamic law*.

Many useful constructs are defined as abbreviations for the basic forms (1) and (2) shown above. For instance, the law

[1] `http://www.cs.utexas.edu/users/tag/cc/`
[2] `http://potassco.sourceforge.net/`

$$a \textbf{ causes } F \textbf{ if } G, \qquad \text{for an action constant } a, \tag{3}$$

stands for **caused** F **if** $\top$ **after** $a \wedge G$;

$$\textbf{inertial } c, \qquad \text{for a fluent constant } c, \tag{4}$$

stands for **caused** c **if** c **after** c;

$$\textbf{exogenous } a, \qquad \text{for an action constant } a, \tag{5}$$

stands for **caused** a **if** a and **caused** $\neg a$ **if** $\neg a$;

$$\textbf{default } a, \qquad \text{for an action constant } a, \tag{6}$$

stands for **caused** a **if** a; and

$$\textbf{nonexecutable } H \textbf{ if } F, \qquad \text{for an action formula } H, \tag{7}$$

stands for **caused** $\bot$ **after** $H \wedge F$.

A *causal theory* contains a finite set of *causal rules* of the form $F \Leftarrow G$ where F and G are formulas. Following [8], the semantics of an action description D is defined by a translation to the union of an infinite sequence of causal theories D_m $(m \geq 0)$. The signature of D_m consists of pairs of form $i : c$ such that $i \in \{0, \ldots, m\}$ and c is a fluent constant of D, or $i \in \{0, \ldots, m-1\}$ and c is an action constant of D. The rules of D_m are

- $i : F \Leftarrow i : G$, for static law (1) in D and $i \in \{0, \ldots, m\}$, and action dynamic law (1) in D and $i \in \{0, \ldots, m-1\}$;
- $i{+}1 : F \Leftarrow (i{+}1 : G) \wedge (i : H)$, for every fluent dynamic law (2) and $i \in \{0, \ldots, m-1\}$;
- $0 : c = v \Leftarrow 0 : c = v$, for simple fluent constant c and $v \in Dom(c)$.

A model of causal theory D_m can be seen as a path of length m in the transition diagram, as described in proposition 8 of [8].

Example 1. Consider a robot that uses a manipulator to transfer small objects from one place to another. It can perform actions $Move(l)$, $Pickup(s)$, $Putdown(s)$ which affects inertial fluents $Loc(o)$, $Hold(s)$, where l denotes the places in the domain, o the objects, s the small objects which can be grasped by the robot. The action description is

$$\textbf{inertial } Loc(o) = l \quad \textbf{inertial } Hold(s)$$
$$\textbf{exogenous } Move(l) \quad \textbf{exogenous } Pickup(s) \quad \textbf{exogenous } Putdown(s)$$
$$\textbf{caused } Loc(s) = l \textbf{ if } Hold(s) \wedge Loc(Robot) = l$$
$$Move(l) \textbf{ causes } Loc(Robot) = l \quad \textbf{nonexecutable } Move(l) \textbf{ if } Loc(Robot) = l$$
$$Pickup(s) \textbf{ causes } Hold(s) \quad \textbf{nonexecutable } Pickup(s) \textbf{ if } \neg Hold(Nothing)$$
$$\textbf{nonexecutable } Pickup(s) \textbf{ if } Loc(Robot) \neq Loc(s)$$
$$Putdown(s) \textbf{ causes } Hold(Nothing) \quad \textbf{nonexecutable } Putdown(s) \textbf{ if } \neg Hold(s)$$

The action desription D^0 is obtained by setting the variables $l \in \{L_1, L_2\}, o \in \{Robot, S\}, s \in \{S\}$. A model of D^0_4 can be represented as a path of length 4 in the transition diagram of D^0.

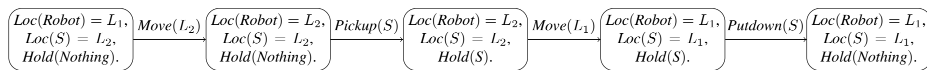


Figure 1 One path in the transition diagram D^0.

3 Defining Composite Actions

3.1 Syntax

We consider a fragment of general action descriptions in $\mathcal{C}+$ containing static laws of the form (1), action dynamic laws of the form (5) and (6), and fluent dynamic laws of the form (3), (4) and (7).

Given an action description D with a set of fluent constants σ^{fl} and a set of action constants σ^{act}, an *extended action description* D^+ introduces a set of *composite action constants* σ^{comp} and *composite action definition laws* of the form

$$b \textbf{ is } (a_0 \textbf{ if } E_0); \ldots; (a_k \textbf{ if } E_k) \tag{8}$$

where $b \in \sigma^{comp}$ is the *head* of the law, called a *composite action constant.* $a_0, \ldots, a_k \in \sigma^{act} \cup \sigma^{comp}$, and $E_0, \ldots, E_k$ are fluent formulas. Intuitively, this law means executing composite actions b is defined as executing a_0 if E_0 holds, then executing a_1 if E_1 holds, ..., then executing a_k if E_k holds. If E_i does not hold, action a_i will be skipped.

A composite action defined in (8) is *acyclic* if there exists a mapping $\lambda : \sigma^{act} \cup \sigma^{comp} \to \{0, 1, 2, \ldots\}$, such that $\lambda(b) > \lambda(a_i)$ for every $i \in \{0, \ldots, k\}$. In the following we assume composite actions are acyclic to forbid infinite recursion such as b **is** $b; a$.

An action description is acyclic if there exists one mapping λ such that every composite action definition law in the action description is acyclic.

Example 1, continued. We would like to extend the action description D^0 by introducing two composite actions $Fetch(s, l)$, and $Bring(s, l)$:

$$\begin{array}{l} Fetch(s, l) \textbf{ is } Move(l_1) \textbf{ if } Loc(s) = l_1 \wedge Loc(Robot) \neq l_1; Bring(s, l). \\ Bring(s, l) \textbf{ is } Pickup(s); Move(l); Putdown(s). \end{array} \tag{9}$$

Intuitively, $Fetch(s, l)$ means "fetch the object s from some other location to l", and $Bring(s, l)$ means "bring the object s from here to location l".

3.2 Semantics

Given an acyclic action description D^+, let S be the set of composite action definition laws in D^+. For each $r \in S$, an *associate action tuple* $t(r)$ is a pair $\langle b, A \rangle$ where b is the head of r, A is an ordered list over $\sigma^{act} \cup \{\epsilon\}$.each $t(r)$ is defined sequentially on the ordered list of $[r_1, r_2, \ldots, r_m]$, where $r_i \in S$ and $\lambda(head(r_i)) \leq \lambda(head(r_j))$ for $i < j$ such that

- $t(r) = \langle b, [a_0, \ldots, a_k] \rangle$ if for every $i \in \{0, \ldots, k\}$, $a_i \in \sigma^{act}$ of r.
- otherwise, $t(r) = \alpha(\langle b, [a_0, \ldots, a_k] \rangle)$. For all $\alpha(\langle b, A \rangle)$ of the form $\langle b, A' \rangle$, A' is a list obtained from replacing every $a_i \in \sigma^{comp}$ in A with all elements of an corresponding ordered list B_i such that
  - for every $a_i \in \sigma^{comp}$ in A, there is an associate action tuple $t' = \langle a_i, A_i \rangle$ which is already defined for some $r \in S$, and
  - B_i is an ordered list of the same length as A_i, with the first element a_i and the remaining elements ϵ.

For example, the associate action tuples of the two rules in (9) are:

$$t(r_1) = \langle Bring, [Pickup, Move, Putdown] \rangle\,,$$
$$t(r_2) = \alpha(\langle Fetch, [Move, Bring] \rangle) = \langle Fetch, [Move, Bring, \epsilon, \epsilon] \rangle\,.$$

For a composite action definition law r and its associate action tuple $\langle b, [a_0, a_1, \ldots, a_{k'}] \rangle$, $index(b, a_i) = i$ if $a_i \neq \epsilon$.

For instance, in (9), we have

$$index(Fetch, Move) = 0, index(Fetch, Bring) = 1,$$
$$index(Bring, Pickup) = 0, index(Bring, Move) = 1, index(Bring, Putdown) = 2.$$

The intuitive meaning of $index(b, a) = t$ is that a is the t-th action that defines b.

Let k^* be the maximal length of A in the associate action tuples of S, σ_0 be the set of all the actions that defines the composite actions. Intuitively, it is the maximal number of primitive actions expanded by a composite action. e.g $k^* = 4$ for (9), the action $Fetch(s, l)$ can be expanded to 4 primitive actions at most. Since we specify that a composite action is executed in 1 time interval as well as a primitive action, we can only represent its executing trajectory in a different dimension to specify time. As a result, a time interval $(i, i+1)$ is divided by subtime points $i = i.0, \ldots, i.k^* = i+1$ and into k^* subintervals $(i, i.1), (i.1, i.2) \ldots, (i.k^*-1, i+1)$, and fluents have values in all subtime points.

Formally, an extended action description D^+ can be translated into an infinite sequence of causal theories D^+_m $(m \geq 0)$.

The signature of D^+_m contains all the symbols occurring in the signature of D_m, and in addition, for each composite action definition law (8), the triples:

- $i.j : a_t$, where $i \in \{0, \ldots, m-1\}$, $j \in \{0, \ldots, k^*-1\}$ and $a_t \in \sigma_0$, and
- $i.j : c$, where $i \in \{0, \ldots, m-1\}$, $j \in \{0, \ldots, k^*\}$ and c is a fluent constant.

The causal theory translated by D^+_m contains rules of the following parts (assuming $i \in \{0, \ldots m-1\}$, $j \in \{0, \ldots, k^*-1\}$ unless stated otherwise):

1. all rules in D_m except rules obtained from (4). That means the primitive actions are executed in 1 time interval.
2. for every fluent dynamic law (4) and $v \in Dom(c)$, rules
   $i.j{+}1 : c = v \Leftarrow (i.j{+}1 : c = v) \wedge (i.j : c = v)$.
   The rules state that the original inertial laws form (4) are replaced by a group of inertial laws specifying the values of fluents at subtime points.
3. for every $v \in Dom(c)$, the synonymity rules
   $i.0 : c = v \leftrightarrow i : c = v \Leftarrow \top, \qquad i{+}1 : c = v \leftrightarrow i.k^* : c = v \Leftarrow \top.$
   These rules states that every simple fluent has the same value at time point i and $i.0$, as well as $i.k^*$ and $i{+}1$.
4. for each static law (1) and $t \in \{1, \ldots, k^*-1\}$, rules
   $i.t : F \Leftarrow i.t : G$.
   The rules mean that the static laws defining the relationship between fluents at time points are also used for subtime points.
5. for every law (3), rules
   $i.j{+}1 : F \Leftarrow (i.j{+}1 : G) \wedge (i.j : H)$.
   These rules say that the action a_j leads to the same effect in the subinterval.
6. for every law (7) where H contains only one action symbol, rules
   $\bot \Leftarrow (i.j : H \wedge F)$.
   The rules state that when an action is nonexecutable at some timepoint, it is also nonexecutable at the subtime point with the same condition.
7. for each law (8),

   a. for each fluent dynamic rule (7) and there is at least one action symbol other than a_0 occurs in H, rules
      $\bot \Leftarrow (i : H^b_{a_0} \wedge F)$,
      where $H^b_{a_0}$ means to replace every occurrance of a_0 with b in H. The rules say that any action that can not be concurrently executed with the first action of the composite action can also not be executed concurrently with the composite action itself.

b. for $0 \leq n \leq k$, set of rules

$$i : b \Leftarrow i : b \qquad i : \neg b \Leftarrow i : \neg b \qquad i.j : \neg a_t \Leftarrow i.j : \neg a_n$$
$$i.t : b_j \Leftarrow (i : b) \wedge (i.t : E_j) \wedge index(b, b_j) = t$$
$$i.j{+}t : b_j \Leftarrow (i.j : b) \wedge (i.j{+}t : E_j) \wedge index(b, b_j) = t$$
$$\bot \Leftarrow i : a_n \wedge i : b.$$

These rules say that any composite action is exogenous, and its primitive actions can only be "triggered" when the condition E_j is true at the shifted subtime point, which is determined by the value of *index* over the action pair. Also, we state that the composite action can not be executed concurrently with its primitive actions.

8. for $b_m, b_n \in \sigma^{comp}$, rules

$$\bot \Leftarrow i : b_m \wedge i : b_n.$$

The rules state that composite actions cannot be concurrently executed.

4 Properties of Extended Action Description

In this section we investigate the properties of the semantics of extended action descriptions by generalizing the notion of using a transition diagram to characterize the model of an action description proposed in [8]. We will identify an interpretation I of a causal theory with the set of atoms that are satisfied by this interpretation, that is to say, with the set of atoms of the form $c = I(c)$. Such a convention allows us to represent a model of an extended action description D^+_m as

$$\bigcup_{0 \leq i \leq m} i : s_i \cup \bigcup_{0 \leq i \leq m-1} i : e_i \cup \bigcup_{0 \leq i \leq m-1} (\bigcup_{0 \leq j \leq k^*} i.j : s_{i.j} \cup \bigcup_{0 \leq j \leq k^*-1} i.j : \widehat{e}_{i.j}) \qquad (10)$$

where $e_0, \ldots, e_{m-1}$ are interpretations of $\sigma^{act} \cup \sigma^{comp}$, $s_0, \ldots s_m, s_{i.1}, \ldots, s_{i.k}$ are interpretations of σ^{fl}, and $\widehat{e}_{i.0}, \ldots, \widehat{e}_{i.k^*}$ are interpretations of σ_0.

A *state* is an interpretation s of σ^{fl} such that $0 : s$ is a model of D^+_0. States are vertexes of the transition diagram represented by D^+.

The transitions are defined by models of D^+_1, a model of D^+_1 can be represented in (10) with $m = 1$.

An *explicit transition* is a triple $\langle s, e, s' \rangle$ where s and s' are interpretations of σ^{fl} and e is an interpretation of $\sigma^{act} \cup \sigma^{comp}$ such that $(0 : s) \cup (0 : e) \cup (1 : s')$ belongs to a model of D^+_1. If for some $b \in \sigma^{comp}$, $e(b) = \mathbf{t}$, then $\langle s, e, s' \rangle$ is called a *composite transition*, otherwise it is called a *simple transition*.

An *elaboration* is a tuple of the form $\langle s, \widehat{e}_0, s_1, \ldots, s_{k^*}, \widehat{e}_{k^*}, s' \rangle$, where $\widehat{e}_i$ is an interpretation of σ_0 and s_i is an interpretation of σ^{fl}, such that

$$(0 : s) \cup (0.0 : \widehat{e}_0) \cup (0.1 : s_1) \cup \ldots \cup (0.k^*{-}1 : s_{k^*-1}) \cup (0.k^*{-}1 : \widehat{e}_{k^*-1}) \cup (1 : s')$$

belongs to a model of D^+_1. An elaboration can be seen as a list of k^* triples $\langle s, \widehat{e}_0, s_1 \rangle, \ldots, \langle s_{k^*-1}, \widehat{e}_{k^*-1}, s' \rangle$. Each of the triples is called an *implicit transition*. If $\widehat{e}_j(a_j) = \mathbf{f}$ for any a_j occurring in (8) for $j \in \{0, \ldots, k\}$, the elaboration is called a *trivial elaboration for b*. The edge of the transition diagram of D^+ are the transitions in the models of D^+_1.

The above definition implicitly relies on the following properties of transitions.

▶ Proposition 1. For any explicit transition $\langle s, e, s' \rangle$ or implicit transition $\langle s, \widehat{e}_i, s' \rangle$, s and s' are states.

This proposition is a generalization of Proposition 7 in [8]. Again, the validity of this proposition depends on the fact that statically determined fluents are not allow to occur in the head of a fluent dynamic law (2).

To relate the model of the causal theory obtained from an extended action description, Proposition 8 of [8] is generalized to include composite transitions and elaborations.

▶ Proposition 2. For any $m > 0$, an interpretation (10) on the signature of D_m^+ is a model of D_m^+ iff for $0 \leq i \leq m-1$ each triple $\langle s_i, e, s_{i+1}\rangle$ is an explicit transition, and each tuple $\langle s_i, \widehat{e}_i, s_{i.1}, \ldots, s_{i.k^*-1}, \widehat{e}_{i.k^*-1}, s_{i+1}\rangle$ is an elaboration.

Proposition 1 and Proposition 2 allow us to represent an extended action description as a transition graph.

Now we investigate the soundness of the new language. Following [3], for action description D and D' such that the signature of D is a part of the signature of D', D is a *residue* of D' if restricting the states and events of the transition system for D' to the signature of D establishes an isomorphism between the transition systems for D' and D.

▶ Proposition 3. Let D be an action description of a signature σ and b be a constant such that $b \notin \sigma$. If D' is an action description of the signature $\sigma \cup \{b\}$ obtained from D by adding a composite action definition law of b in terms of σ, then D is a residue of D'.

For instance, in the simple robotic domain, the transition system represented by $(D^0)^+$ is isomorphic to the transition system represented by D^0, by restricting the events of the transition system for $(D^0)^+$ to the action constants other than $Fetch(s,l)$, $Bring(s,l)$.

In addition to showing that an extended action description inherits all "good" things from the original action description, we also show that it doesn't introduce anything "bad": a primitive action a_j, if executed at subtime point, is the exact simulation of the action a_j executed at some time point as a primitive action, their transitions are in 1-1 correspondence.

▶ Proposition 4. Each implicit transition $\langle s, \widehat{e}, s'\rangle$ of D^+ corresponds to a transition $\langle s, e, s'\rangle$ of D.

Based on this proposition, it is easy to see that an elaboration in D^+ corresponds to a path of length k^* in the transition diagram of D. Figure 2 shows the transitions of a model of $(D_1^0)^+$, where the implicit transitions are represented as dashed arrows. It can be seen that every implicit transition corresponds to a transition in D^0, as shown in Figure 1.

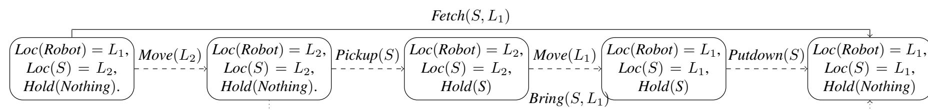


Figure 2 A model of $(D_1^0)^+$ represented as transitions.

5 Experiments—KeJia's Domain

5.1 Formalizing and Reasoning with Composite Actions

In this section, we use composite actions to formalize the domain of the robot KeJia [2]. The robot has a manipulator that can operate various kinds of appliances. The actions that he can perform include $Move(l)$, $Pickup(s)$, $Putdown(s)$, $Open(m)$, $Close(m)$, $Putin(s,m)$, $Takeout(s,m)$, $Start(m)$. Typical scenarios include fetching objects from different places according to the requests of humans and doing other housework such as heating the food with the microwave oven. In addition to do usual task planning, KeJia can acquire knowledge from either human user or textual materials to enrich its knowledge base and planning abilities. For instance, when KeJia is asked to heat the food with microwave oven while he doesn't know how to use the appliance, he can either try to download microwave manuals from internet, did textual analysis to extract instructions, or ask help from humans.[3]

[3] A video of using microwave is at `http://wrighteagle.org/en/demo/ServiceRobot_oven.php`

The instructions of using many household appliances is usually acquired from either textual manuals or humans. The structure of these instructions are usually quite similar, such as “first put the object into the machine, then close the door of the machine, and start the machine, after a while, open the door, finally take out the object from the machine”. Instructions of this kind can be converted to composite action definition law by KeJia's natural language understanding module:

$$Use(o, m) \textbf{ is } Putin(o, m); Close(m); Start(m); Open(m); Takeout(o, m).$$

Therefore, heating food with a microwave oven and washing clothes with a washer can be formalized by refering to the knowledge of using the machine as:

$$Heat(f) \textbf{ is } Move(l) \textbf{ if } Loc(Microwave) = l \wedge Loc(Robot) \neq l; Use(f, Microwave).$$

$$Wash(c) \textbf{ is } Move(l) \textbf{ if } Loc(Washer) = l \wedge Loc(Robot) \neq l; Use(c, Washer).$$

These laws are added into the knowledge base incrementally without modifying any other parts in the knowledge base, due to the feature of elaboration tolerance of the formalism. Composite actions make it easier for a robot to gain useful procedural knowledge in many ways, such as oral instructions, or information from internet. More generally, the actions can be defined by referring to actions in a general-purpose library.

A complete formalization of the domain is available at `http://wrighteagle.org/kejiaexp/`. In the following we assume four places (l_1, l_2, l_3, l_4).

Prediction. *Initially, the robot is at l_2, the popcorn is at l_3 and not heated, the microwave oven is at l_2 with the door open, the washer is at l_4 and the door is closed, and the milk is in the robot's hand. The robot heats the milk with the microwave oven, and then put to milk into her plate. Does it follow that in the resulting state, the robot, the milk and the microwave oven are at the same location?*

To solve this problem, we add the following query rules into the causal theory

```
:- query
maxstep :: 2;
0:loc(robot)=l2, loc(microwave)=l2, loc(popcorn)=l3, -heated(popcorn),
  -heated(milk), dooropen(microwave), loc(washer)=l4, doorclosed(washer),
  inside(hand)=milk, heat(milk,microwave);
1:putintoplate(milk).
2:loc(robot) \= loc(milk) ++ loc(milk) \= loc(microwave).
```

The extended CPLUS2ASP return “`UNSATISFIABLE`”, indicating that at time 2, the robot is at the same location with the milk and the microwave.

Planning. *Given the same initial state as above, find a plan within 10 steps so that the milk and the popcorn are both heated by the robot.*

When the corresponding query is specified, one of the answer sets returned by the extended CPLUS2ASP contains atoms:

```
0:heat(milk), 0.1:use(milk,microwave), 0.1:putin(milk,microwave),
0.2:close(microwave), 0.3:operate(microwave), 0.4:open(microwave),
0.5:takeout(milk,microwave), 1:toplate(milk), 2:move(l3), 3:pickup(popcorn),
4:heat(popcorn), 4.0:move(l2), 4.1:use(popcorn,microwave),
4.1:putin(popcorn,microwave), 4.2:close(microwave), 4.3:operate(microwave),
4.4:open(microwave), 4.5:takeout(popcorn,microwave).
```

We have three observations. First, composite actions occur as building blocks of the plan, for example, we see `0:heat(milk)`, `0.0:use(milk,microwave)`, etc in the result. Second, when a composite action is executed, all details about the executions of the primitive actions in the composite action are also included, for instance, when `0:heat(milk)` is executed, we also have the details `0.0:use(milk,microwave)`, . . . , `0.4:takeout(milk,microwave)`.

Table 1 The results of the *KeJia* domain.

Length of Plans	#Instances	#Time-Outs	Time ratio
≤20	35	0	0.138
21–25	13	0	0.274
26–30	24	0	1.505
31–35	23	3	1.553
36–40	6	2	2.096
41–45	1	1	–

Third, composite actions can have different kinds of execution trajectory, for instance, the execution trajectory of the action `4:heat(popcorn)` has the action `4.0:move(l2)` more than that of `0:heat(milk)`.

5.2 Performance

We test planning performance by two representations of the domain KeJia: a traditional representation $KeJia_1$ without any composite actions, and an extended representation $KeJia_2$ by adding some composite actions into $KeJia_1$. We consider 120 different instances, for every instance, the numbers of locations and objects, the initial states and the goal states are randomly generated. We set the longest acceptable length of a plan for a instance using $KeJia_1$ to 50 and time limit for computing to be 30min[4].

The result is shown in Table 1. There are 18 problems which can be solved by neither representations. We classify the other instances into 6 categories by the length of the plans generated using $KeJia_1$. For each category, the third column shows the number of instances that cannot be computed using $KeJia_1$. The last column shows the average ratio of times on computing a instance using $KeJia_1$ and $KeJia_2$ where time-out instances are excluded. There are no time-outs using $KeJia_2$.

In Table 1, we notice that when the plan length increases from ≤20 to 36–40, the ratio increases simultaneously, especially, when the length of a plan is up to 26-30, the time ratio is always > 1, indicating that the composite actions help improve the efficiency as the complexity of domain tasks increases. The reason the time ratio is < 1 is that there are more rules introduced by composite actions, which may also become overhead of computation. For large domains, the composite actions in the plan contain a lot of consecutive executions of the primitive actions. Making use of composite actions allows the solver iClingo to find the "cumulative effects" at earlier stages of grounding.

Therefore, when the task domain has a "hierarchical structure" such that its plan consists of many consecutive executions of primitive actions which can compose to an action in a different abstraction space, composite actions may be worthwhile and can improve the efficiency.

6 Conclusion

In this paper we introduce composite actions into a fragment of $\mathcal{C}+$. Action description equipped with composite actions leads to a more intuitive and flexible way to formalize action domains by exploiting general-purpose formalization, a step to address the problem of

[4] The detailed representation, instances and logs, as well as the extended Cplus2ASP system can be found at `http://wrighteagle.org/kejiaexp/`

generality, and improve efficiency of reasoning and planning by characterizing the hierarchical structure of the problem domain. Extended action descriptions can be processed by the extended CPLUS2ASP system.

A direct next step is to apply CPLUS2ASP on robot KeJia to solve the real-life problems for real-time computation. In the future, we would like to introduce composite action definition to MAD, where modular actions can be defined as special case or sequential executions of actions, by referring to a general-purpose library. Composite actions should also be defined on $\mathcal{C}+$ in its full generality.

Acknowledgements. This work is supported by the National Hi-Tech Project of China under grant 2008AA01Z150 and the Natural Science Foundation of China under grant 60745002 and 61175057, as well as the USTC Key Direction Project and the USTC 985 Project. The authors are grateful to Vladimir Lifschitz, Michael Gelfond, Alfredo Gabaldon, Daniela Inclezan and the anonymous reviewers for their constructive comments and suggestions.

References

1 Michael Casolary and Joohyung Lee. Representing the language of the causal calculator in answer set programming. In *Technical Communications of the 27th International Conference on Logic Programming (ICLP 2011)*, pages 51–61, 2011.

2 X. Chen, J. Ji, J. Jiang, G. Jin, F. Wang, and J. Xie. Developing high-level cognitive functions for service robots. In *Proc. of 9th Int. Conf. on Autonomous Agents and Multi-agent Systems (AAMAS 2010)*, 2010.

3 Selim T. Erdoğan and Vladimir Lifschitz. Actions as special cases. In *Proceedings of International Conference on Principles of Knowledge Representation and Reasoning (KR)*, pages 377–387, 2006.

4 Kutluhan Erol, James A. Hendler, and Dana S. Nau. Htn planning: Complexity and expressivity. In *AAAI*, pages 1123–1128, 1994.

5 Richard Fikes and Nils Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3–4):189–208, 1971.

6 Michael Gelfond and Daniela Inclezan. Yet another modular action language. In *Proceedings of the Second International Workshop on Software Engineering for Answer Set Programming*, pages 64–78, 2009.

7 Michael Gelfond and Vladimir Lifschitz. Action languages. *Electronic Transactions on Artificial Intelligence*, 3:195–210, 1998.

8 Enrico Giunchiglia, Joohyung Lee, Vladimir Lifschitz, Norman McCain, and Hudson Turner. Nonmonotonic causal theories. *Artificial Intelligence*, 153(1–2):49–104, 2004.

9 Daniela Inclezan and Michael Gelfond. Representing Biological Processes in Modular Action Language ALM. In *Proceedings of the 2011 AAAI Spring Symposium on Formalizing Commonsense*, pages 49–55. AAAI Press, 2011.

10 Hector J. Levesque, Raymond Reiter, Yves Lespérance, Fangzhen Lin, and Richard B. Scherl. Golog: A logic programming language for dynamic domains. *J. Log. Program.*, 31(1-3):59–83, 1997.

11 Vladimir Lifschitz and Wanwan Ren. A modular action description language. In *Proceedings of National Conference on Artificial Intelligence (AAAI)*, pages 853–859, 2006.

12 John McCarthy. Generality in Artificial Intelligence. *Communications of the ACM*, 30(12):1030–1035, 1987. Reproduced in [13].

13 John McCarthy. *Formalizing Common Sense: Papers by John McCarthy.* Ablex, Norwood, NJ, 1990.

14 John McCarthy and Patrick Hayes. Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, volume 4, pages 463–502. Edinburgh University Press, Edinburgh, 1969.

15 Earl D. Sacerdoti. Planning in a hierarchy of abstraction spaces. In *Proceedings of the 3rd international joint conference on Artificial intelligence*. Morgan Kaufmann Publishers Inc., 1973.

16 Murray Shanahan. Event calculus planning revisited. In *Proceedings 4th European Conference on Planning (ECP 97), Springer Lecture Notes in Artificial Intelligence no. 1348*, pages 390–402. Springer, 1997.

17 Tran Cao Son, Chitta Baral, and Sheila A. McIlraith. Planning with different forms of domain-dependent control knowledge - an answer set programming approach. In *LPNMR*, pages 226–239, 2001.

Improving Quality and Efficiency in Home Health Care: an application of Constraint Logic Programming for the Ferrara NHS unit*

Massimiliano Cattafi[1], Rosa Herrero[2], Marco Gavanelli[1], Maddalena Nonato[1], Federico Malucelli[3], and Juan José Ramos[2]

1 **ENDIF, Università di Ferrara, Italy**
`{massimiliano.cattafi, marco.gavanelli, maddalena.nonato}@unife.it`

2 **Dept. de Telecomunicació i Enginyeria de Sistemes, UAB, Spain**
`rherrero.math@gmail.com, juanjose.ramos@uab.cat`

3 **Dipartimento di Elettronica e Informazione, Politecnico di Milano, Italy**
`malucell@elet.polimi.it`

Abstract

Although sometimes it is necessary, no one likes to stay in a hospital, and patients who need to stay in bed but do not require constant medical surveillance prefer their own bed at home. At the same time, a patient in a hospital has a high cost for the community, that is not acceptable if the patient needs service only a few minutes a day.

For these reasons, the current trend in Europe and North-America is to send nurses to visit patients in their home: this choice reduces costs for the community and gives better quality of life to patients. On the other hand, it introduces the combinatorial problem of assigning patients to the available nurses in order to maximize the quality of service, without having nurses travel for overly long distances.

In this paper, we describe the problem as a practical application of Constraint Logic Programming. We first introduce the problem, as it is currently addressed by the nurses in the National Health Service (NHS) in Ferrara, a mid-sized city in the North of Italy. Currently, the nurses solve the problem by hand, and this introduces several inefficiencies in the schedules.

We formalize the problem, obtained by interacting with the nurses in the NHS, into a Constraint Logic Programming model. In order to solve the problem efficiently, we implemented a new constraint that tackles with the routing part of the problem. We propose a declarative semantics for the new constraint, and an implementation based on an external solver.

1998 ACM Subject Classification D.3.2 Constraint and logic languages

Keywords and phrases CLP(FD), Nurse Scheduling Applications, Home Health Care

Digital Object Identifier 10.4230/LIPIcs.ICLP.2012.415

1 Introduction

One of current trends to reduce costs and maintain service quality of health services is to close peripheral hospitals, reduce patients hospitalization, and concentrate the service at

* We thank Andrea Peano for his help with the significance tests. This work was partially supported by EU project *ePolicy*, FP7-ICT-2011-7, grant agreement 288147. Possible inaccuracies of information are under the responsibility of the project team. The text reflects solely the views of its authors. The European Commission is not liable for any use that may be made of the information contained in this paper.

© M. Cattafi, R. Herrero, M. Gavanelli, M. Nonato, F. Malucelli, and J.J. Ramos;
licensed under Creative Commons License ND

few, big structures, able to provide specialized treatments and high quality consultancy. At the same time, though, those patients who do not need to be treated in a hospital, must be provided health care directly at their homes. The challenge is to keep costs at a low level, while achieving high service quality standards, comparable to those achievable at a hospital.

In this paper we describe an application concerning home health care in the city of Ferrara, Italy. We describe the problem as stated by the workers in the local agency of the National Health Service (NHS), together with the data they provided us. Then, we model the problem in Constraint Logic Programming on finite domains (CLP(FD)).

1.1 The home health care service in Ferrara

At present, the home health care (HHC) service in the city of Ferrara, Italy, is managed by the local agency of the National Health Service (NHS), namely AUSL 109. All patients who are not self sufficient and in need for medical treatment are eligible for HHC. Each request is thus characterized by a patient identifier (name and address), a medical treatment, and the specific day of the week when the treatment must be delivered (each patient can have more than one request per week).

Service is provided by a set of qualified nurses. Every day, each nurse starts his/her duty at the city hospital, visits the patients in his/her list, delivers the required treatments and travels by car from one patient's home to the next, until (s)he finally returns to the hospital.

A treatment lasts from 5 to 60 minutes, depending on its specific characteristics.

While Ferrara is a medium-size town (about 150,000), the area administered by AUSL 109 is rather large and its population ageing. Although most of the population is concentrated in town, a number of elderly people live in the countryside and they are those more likely to be enrolled in the service. Therefore, the service is characterized at the same time by a high variance of duration and a significant geographical dispersion of the requests.

Scheduling such a service poses several challenges; good solution should achieve:

- from the NHS point of view, the minimization of the travel time over the service time; in fact during travel a nurse is on duty but is not delivering any service.
- from the nurses point of view, the equidistribution of the workload, which can not be guaranteed by simply equally subdividing patients, due to heterogeneous requests.
- from the patient point of view, a good degree of *loyalty*, i.e., the number of different nurses who are in charge of a single patient should be kept as low as possible.

A total of 15 nurses is involved. A duty should last up to 7 hours and 12 minutes. As a representative sample, in February 2010 there were 3323 requests, subdivided among 458 patients.

At present, nurses organize their duties themselves. In order to simplify the subdivision of the patients to the nurses, the territory pertaining AUSL 109 has been partitioned statically into 9 zones Each nurse receives in charge most of the patients belonging to one such area. Then the nurse tries to fit the patients requests into the working shifts while complying with the maximum workload allowed. Such decisions are not driven by any optimization criteria, and the routing is not necessarily optimal within the day, leaving apart what could be gained in terms of travelled distance if requests would be exchanged between nurses. Due to the greedy procedure followed, the nurse weekly schedules have very different workloads and balancing this load over the months leads to a detriment in loyalty. Nurses complain about such disparities, and have difficulties adapting their schedule to new incoming patients, new treatments, or to any other change. Moreover, if the workload balance could be improved by optimizing the routing component, nurses could be available at the hospital for others tasks,

thus reducing the overall costs. In addition, an improved routing plan would impact on the direct expenses related to gas and car usage, which contribute to the overall cost.

2 A Constraint for the Traveling Salesman Problem

Before delving into the actual CLP model of the problem, we present a new constraint useful to address efficiently the routing component of the problem. Intuitively, the new constraint provides the length of the shortest Hamiltonian cycle connecting a given subset of the nodes in a graph.

Given a fully-connected graph $G \equiv (N, E)$ with a special node $0 \in N$, a weighting function $d : N \times N \mapsto \mathbb{R}$ (also represented in matrix notation $D = (d_{i,j})$), and a selection function $s : N \mapsto \{0, 1\}$ (also represented as a list, or a 1-dimensional matrix S) constraint

$$traveltime(N, D, S, T^{trv}) \tag{1}$$

solves a TSP and computes the length of the shortest Hamiltonian cycle associated to the set of nodes $N' = \{n \in N | S(n) = 1\}$. More precisely, given a list of nodes $N = (p_1, p_2, \ldots, p_m)$, a matrix of distances D, and a list of values $S = (s_1, \ldots, s_m)$, constraint (1) is true iff

$$T^{trv} = \min_{Path} \sum_{(i,j)\in Path} d_{i,j}$$

such that

- $Path$ is a sequence of the form

$$0, (0, p_{k_1}), p_{k_1}, (p_{k_1}, p_{k_2}), p_{k_2}, \ldots (p_{k_{n-1}}, p_{k_n}), p_{k_n}, (p_{k_n}, 0), 0$$

  that alternates nodes with edges, starting and ending at node 0;
- the visited nodes are exactly those corresponding to elements in the list S:

$$p_i \in Path \Longleftrightarrow S_i = 1.$$

Pseudocode in Figure 1 outlines the implementation of the *traveltime* constraint. Operationally, it awakes every time one of the s_i variables is instantiated. If $s_i = 1$, it means node p_i must be visited. Note that, in a generic node of the search tree, some of the s_i variables will have value 1, some will have been set to 0, and some will still be unassigned. The predicate in line 2 selects in $DefinitelyVisited$ the nodes that have definitely to be visited. This variable is, in turn, passed as an argument to the predicate which computes the corresponding TSP (line 3). The TSP thus takes into account at this stage the nodes which are currently known to be visited, i.e., those for which the S variable has been set to 1.

If not all of the S variables are ground (test on line 4), the TSP cost provides a valid lower bound to the actual travel time (line 6) and can be used in a branch-and-bound search. In fact, if the lower bound is higher than the elements in the domain of the variable T^{trv} (for example, because the working hours of the nurse are almost all used for servicing the patients, so there is not enough time for traveling), we can immediately fail and backtrack, avoiding to continue the search in a wrong branch of the search tree.

When all of the S variables are ground, the cost of the TSP becomes the real travel time, and we are able to finally fix the value of T^{trv} to the TSP cost (line 5).

```
1  traveltime(P,D,S,T^trv):-
2     select_definitely_visited_nodes(P,S,DefinitelyVisited),
3     compute_tsp(DefinitelyVisited,D,LowerBound),
4     (ground(S)
5        -> T^trv = LowerBound
6        ;  T^trv ≥ LowerBound,
7           suspend(traveltime(P,D,S,T^trv))
8     ).
9  select_definitely_visited_nodes([],[],[]).
10 select_definitely_visited_nodes([P_i|P],[S_i|S],Definitely):-
11    (ground(S_i), S_i=1 -> Definitely = [P_i|LV] ; Definitely = LV),
12    select_definitely_visited_nodes(LP,LS,LV).
```

Figure 1 Pseudocode for the *traveltime* constraint.

3 A Constraint Logic Programming Model

Formally, the input data consists of:

- a set $\mathcal{S}_{serv}$ of services, of size N_s; for each service s we know the patient pat_s, the day day_s and the duration dur_s
- a matrix of distances D; the element $d_{i,j}$ is the travel time from patient i to patient j (if i and j are both greater than 0), or from/to the hospital (if $i = 0$ or $j = 0$)
- $\mathcal{S}_{nurse} = \{1, \dots, N_n\}$ is the set of available nurses
- N_d is the number of days considered in the scheduling
- MpD is the amount of minutes available per day for each nurse; this includes both service time and travel time

A solution is an assignment of a nurse to each service, respecting all the constraints. The quality of the solution depends on how balanced the week workloads of the nurses are and on how many different nurses take care of the same patient during the week.

3.1 The CLP Model

To each service s we associate a decision variable $Nurse_s$ that can take a value between 1 and the number of available nurses N_n.

It can be useful to represent the nurses variables also in their Boolean channeling version, using constraint reification; this simplifies the definition of some requirements, as will be clear in the following. We have a matrix SN of size $N_s \times N_n$ such that

$$SN_{s,n} = 1 \iff Nurse_s = n. \qquad (\forall s \in \mathcal{S}_{serv}, \forall n \in \mathcal{S}_{nurse}) \tag{2}$$

We are interested in computing the workload of each nurse n in each day d: $DayWL_{n,d}$. Each day workload is the sum of the total service time and the travel time of that nurse:

$$DayWL_{n,d} \quad = \quad T^{svc}_{n,d} + T^{trv}_{n,d} \qquad (\forall n \in \mathcal{S}_{nurse},\ \forall d \in 1 \dots N_d). \tag{3}$$

The total day workload for each nurse cannot exceed MpD, so for each day d and each nurse n, the domains of variables $DayWL_{n,d}$, $T^{svc}_{n,d}$ and $T^{trv}_{n,d}$ are from 0 to MpD.

The week workload $WeekWL_n$ of nurse n is the sum of the respective day workloads

$$WeekWL_n \quad = \quad \sum_{d=1}^{N_d} DayWL_{n,d} \qquad (\forall n \in \mathcal{S}_{nurse}). \tag{4}$$

The service time is the total time of the durations of the services given by nurse n in day d:

$$T^{svc}_{n,d} = \sum_{s \in \mathcal{S}_{serv}, day_s = d} dur_s \cdot SN_{s,n}. \tag{5}$$

As mentioned in Section 2, the routing part is addressed by constraint *traveltime* (Eq. 1) that solves the TSP of a nurse that visits a subset of the patients. In order to compute the travel time $T^{trv}_{n,d}$ of nurse n in day d, we need to provide to such constraint

1. the nodes of the graph, that are the patients' locations
2. the matrix of distances D,
3. the selection function S (in its list representation), that is a sub-matrix of the SN matrix,
4. and the (finite domain) variable that represents the travel time: $T^{trv}_{n,d}$.

More precisely, since we want to compute the travel time for day d, item 1 will be the set $Patients^d \triangleq \{pat_s | s \in \mathcal{S}_{serv}, day_s = d\}$ of those patients that will be visited in day d, while item 3 will be the sub-matrix $SN^d_n \triangleq \{SN_{s,n} | s \in \mathcal{S}_{serv}, day_s = d\}$ containing only those services to be given in day d. In other words, the actual parameters passed to constraint *traveltime* to compute the traveltime of nurse n in day d will be:

$$traveltime(Patients^d, D, SN^d_n, T^{trv}_{n,d}).$$

3.2 The Objective Function

The requirements given by the chief nurse are to optimize two main objectives, namely the equal repartition of the workload to the various nurses and the loyalty, although psychological factors or tiredness can also affect the quality of the service.

Concerning the first objective, there are various ways to achieve balanced week workloads for the nurses [14]. We chose to minimize the maximum week workload, obtained as

$$MaxWeekWL = \max_{n \in \mathcal{S}_{nurse}} WeekWL_n$$

One way to obtain maximum loyalty is to minimize the number of nurses that visit a same patient. Let $ServicePat_p$ be the set of services of patient p. The information if a patient p is visited during the week by nurse n is given by:

$$PN_{p,n} = \bigvee\nolimits_{s \in ServicePat_p} SN_{s,n} \qquad \forall p \in \mathcal{S}_{patient}, \forall n \in \mathcal{S}_{nurse}$$

(where we identify truth values *true* and *false* with 1 and 0, respectively); then

$$LoyaltyPenalty = \sum_{p \in \mathcal{S}_{patient}, n \in \mathcal{S}_{nurse}} PN_{p,n}$$

The global objective can be given as a weighted sum of the two components

$$\min(\alpha_1 \cdot MaxWeekWL + \alpha_2 \cdot LoyaltyPenalty), \tag{6}$$

where α_1 and α_2 are positive real numbers that can be chosen by the user in order to reflect the current priorities adopted in the AUSL. Of course, such values can be tuned later on.

4 Example

As an example, we have three patients, requesting a total of 5 services, whose durations are in Fig 2 and the distance matrix (in upper triangular form) is in Figure 3. Assume we have two nurses, n_1 and n_2, and that the limit on the day workload is $MpD = 30$.

One solution could be to assign

patient	mon	thu
p_1	10	5
p_2		20
p_3	20	5

Figure 2 Patients' requests with durations.

	h	p_1	p_2	p_3
h		3	3	5
p_1			2	7
p_2				8

Figure 3 Distance Matrix.

- on *mon*, nurse n_1 to patient p_1 (formally, $Nurse_{(p_1,mon)} = n_1$) and nurse n_2 to p_3
- on *tue*, nurse n_1 to patients p_1 and p_3, and nurse n_2 to p_2.

In this assignment, we have $DayWL_{n_1,mon} = T^{svc}_{n_1,mon} + T^{trv}_{n_1,mon} = 10 + (3 + 3) = 16$ (going from the hospital h to p_1 and coming back); $DayWL_{n_2,mon} = 20 + (5 + 5) = 30$; $DayWL_{n_1,tue} = (5 + 5) + (3 + 8 + 5) = 26$; $DayWL_{n_2,tue} = 20 + (3 + 3) = 26$. The total week workload is $WeekWL_{n_1} = 16 + 26 = 42$ for nurse n_1 and $WeekWL_{n_2} = 30 + 26 = 52$ for n_2. The loyalty penalty will be 1 for patients p_1 and p_2 (that are visited by one nurse) and 2 for p_3, that is visited by both nurses. So, the value of the objective function will be $\alpha_1 \cdot \max\{42, 52\} + \alpha_2 \cdot (1 + 1 + 2) = 52\alpha_1 + 4\alpha_2$.

5 Implementation

The TSP solving algorithm (predicate `compute_tsp` in Figure 1) can be implemented in CLP(FD), with different constraint models.

One model assigns a decision variable for each city to be visited. We have a sequence of decision variables $X_1, \ldots, X_n$, each of them ranging on the set of cities to be visited, and where X_1 is the first city to be visited, X_2 is the second, ..., X_n is the last city to be visited. The fact that all cities must be visited is imposed through an `alldifferent` constraint [13]. In this model, the cost is the sum of the distances $d(X_1, X_2) + d(X_2, X_3) + \cdots + d(X_{n-1}, X_n) + d(X_n, X_1)$.

A second model uses the `circuit` constraint [4] for which various propagation algorithms have been proposed [6, 10]. Again, we have a sequence of decision variables $X_1, \ldots, X_n$, each ranging on the set of possible cities, but in this case the meaning is different: X_1 is the city to be visited immediately after city number 1, X_2 is the city to be visited after city whose name is "2", ..., X_n is the city that is visited after the city named n. The `circuit` constraint ensures that allowed assignments form a Hamiltonian circuit, and the cost is the sum $d(1, X_1) + d(2, X_2) + \cdots + d(n, X_n)$.

However, it is well known in the literature [6] that solving large TSPs in CLP(FD) is very demanding in terms of computing time, so we decided to implement predicate `compute_tsp` as an invocation of an efficient TSP solver [9], based on the Lagrangian Relaxation technique used in Operations Research. We could have used other solvers, but we found that our choice performed well on the typical size of the considered TSP instances. Although the TSP instances were very difficult for a CLP(FD) implementation, they were rather easy for Lagrangian Relaxation, and solving them through LR did not show a deterioration in performance with respect to state-of-the-art TSP solvers [1], so we preferred to use a solver for which we had access to the source code. A detailed description of the Lagrangian Relaxation technique is out of the scope of this work; the interested reader can refer to [9].

6 Search Strategies

We tried our model with five search strategies. The first four were developed with the goal of obtaining a good general-purpose search strategy; then we tried to improve by exploiting better the structure of the problem.

The **Generic Search (GS)** strategy is a depth-first search in which the next variable to be assigned is selected with the smallest domain heuristic. Since the decision variables are the $Nurse$ variables (Section 3.1), we select first the service that can be assigned to the smallest number of nurses. The assigned nurse is selected at random. The **Generic Search with Restarts (GSR)** also applies restarts with the optimal timeout sequence [11].

We also modified the variable selection heuristics; instead of selecting the variable with the smallest domain within the services of the whole week, we try to complete the assignments of a single day before starting with another day (first assign all the services of the first day, then the second day, etc). The idea is that if we made some wrong decisions in one day, so that it is impossible to assign all the patients of that day, we want to fail as soon as possible before initiating the assignments in other days. Within each day, we select first the variable with the smallest domain. This strategy was applied without restarts, **Generic Search by Day (GSD)** and with restarts **Generic Search by Day with Restart (GSDR)**.

Finally, we defined a search strategy more tailored to the problem at hand, called **Loyalty Guided Search (LGS)**. We first sort the services in decreasing order of duration, so that those services with higher duration will be assigned first. The idea is to try to fit first the most difficult services into the available time for the nurses. Then, given a service s of patient pat_s, we try to assign him/her a nurse who is already visiting this patient, in order to minimize the *LoyaltyPenalty*. The domain of $Nurse_s$ is divided into two parts: the nurses who are already visiting this patient and those who are not; we try first the first part, then the second. Moreover, both parts are sorted by the current $WeekWL$ of the nurse; in this way, we try first the nurses that are less occupied, in order to balance the workload.

7 Experiments and Results

The program was implemented in the open-source CLP system ECL^{i}PS^e 6.0 [2], and linked to a Java part implementing Lagrangian Relaxation (LR) for the TSP. All tests were performed on an Intel i5 processor 2.40GHz computer with 4GB of RAM on four weekly instances.

Figure 4 shows the computation time required by the routing aspect of the problem with the various methods described in Section 5. The values are obtained by imposing a full weekly assignment of services to nurses and then computing the $N_n \times N_d$ resulting TSPs with each of the different methods. It can be noticed that the circuit-based one is more efficient than the alldifferent-based one, however solving the TSPs with Lagrangian Relaxation is orders of magnitude faster than both of them (times are reported on a logarithmic scale). Using CLP(FD) as a unifying framework, it is practical and convenient to take advantage of the efficiency of LR on this specific subproblem by enclosing it in our *traveltime* constraint.

Table 1 shows the best results obtained for the five search strategies described in Section 6 running them for a maximum of 10 minutes. The randomized algorithms (the GS* strategies) were run 20 times each. For each week, we show the Objective and the corresponding $MaxWeekWL$ and *LoyaltyPenalty*. The Objective is given by the weighted sum in Eq. 6; in these particular experiments, we used $\alpha_1 = \alpha_2 = 1$, so Objective is simply the sum of the two subsequent columns. The results are compared to the solution hand-made (HMS) by the nurses considering the division of the city into 9 areas. We can see that the model was very effective, as all the search strategies were able to improve on the hand-made

Table 1 Best results for each search strategy.

	First Week	Second Week	Third Week	Fourth Week
	Objective=WL+LP	Objective=WL+LP	Objective=WL+LP	Objective=WL+LP
GS	2203 1918 + 285	2371 2064 + 307	2331 2033 + 298	2387 2063 + 324
GSR	2125 1841 + 284	2347 2040 + 307	2270 1963 + 307	2345 2022 + 323
GSD	2185 1905 + 280	2351 2052 + 299	2255 1963 + 292	2389 2071 + 318
GSDR	2097 1811 + 286	2345 2033 + 312	2263 1958 + 305	2342 2011 + 331
LGS	1982 1782 + 200	2277 2042 + 235	2181 1954 + 227	2290 2034 + 256
HMS	2356 2124 + 232	2405 2153 + 252	2395 2141 + 254	2433 2146 + 287

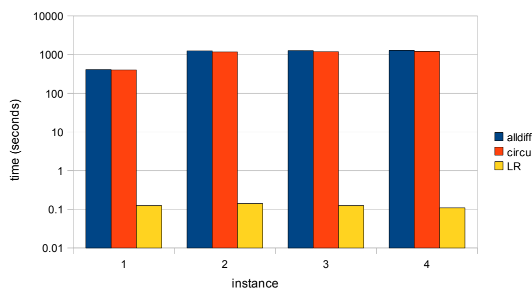


Figure 4 Comparison of computing time required to solve the TSPs with the different methods.

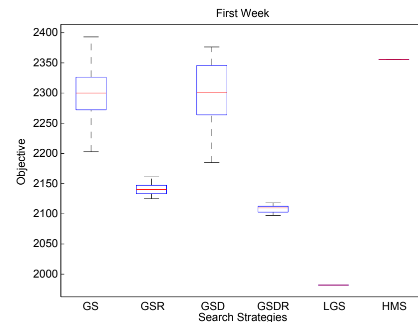


Figure 5 Box plot of the 5 strategies and the hand-made solution (HMS).

solution. Moreover, LGS was able to improve on the hand-made solution both in terms of equidistribution of the workload and in terms of loyalty, thus improving both the working conditions of the nurses and the service quality for the patients. Unluckily, we were not able to compute the optimal solution, so we cannot compare with it.

Since some of the search strategies use randomization, we also show the box plots of the 20 repetitions (Figure 5). The plots reveal that restarts are the most important factor in the general purpose strategies. The strategies not using restarts often were unable to improve the hand-made solution. Labeling on single days can sometimes give a further improvement. However, a search strategy tailored for the problem is able to provide a large improvement with respect to the general purpose ones. Notice that LGS and HMS are strategies that do not include randomization, so they always provide the same solution each time they are executed; this explains why the box plot reduces to a single line. A significance test supports the conjecture that LGS improves upon the hand made solution. The Wilcoxon-Mann-Whitney test [3] rejects the hypothesis that LGS is worse than HMS with a p-value of 0.01429, well below the usual significance threshold of 0.05.

8 Related Work

The efficient delivery of HHC service attracted the attention of the CLP and the Operations Research communities. Application papers are generally focused on the particular type of service that has to be optimized. In many countries for example the Home Health Care service is managed together with the Home Care that involves other types of services and

most of the times requires the specification of time windows in which the services have to be delivered and this is one of the main differences that we found with our case.

For example, Bertels and Fahle [5] adopt a hybrid approach, combining Constraint Programming, local search and Linear Programming. The approach takes advantage of the presence of tight time windows: *"Due to time window constraints, in the HHC only few permutations correspond to feasible orderings. In our approach we enumerate those orderings by a CP approach, and we use an LP to find optimal start times ..."*. In our case time windows are not present thus enumerating the feasible orderings is not viable.

Laps Care [7] is a system adopted in Sweden for Home Care, although it is also able to consider some of the issues in HHC. Laps Care uses an iterative method, in which an initial solution uses a single route for each service; then routes are joined until no further improvement is possible. To escape from the local optimum, one of the routes is split into one route for each patient, and the joining phase restarts.

Looking at the problem from a more abstract viewpoint, one may see some similarities with the classical Capacitated Vehicle Routing Problem (CVRP). In the CVRP a set of disjoint routes for a fleet of vehicles has to be found so that all customers (nodes) are visited, the required quantity of goods is delivered to each customer, the capacity of the vehicles is not exceeded and the objective function is minimized. The usual objective function is the overall traveled distance or the number of vehicles. In our case we can see nurses as vehicles and patients as customers. There are some important differences with CVRP that make all the efficient method developed for the classical problem not applicable in our case. One difference concerns the capacity. As in CVRP we may consider nurse daily duty time as a capacity constraint, however, unlike the CVRP, in our case the sequence in which patients are visited matters on how the capacity is consumed. This actually turns our problem into a time constrained VRP which is not as easy as the CVRP and for which the classical CVRP methods are not so efficiently adapted.

The other difference concerns the objective function. On the one hand, as in VRP, we would have to minimize the total traveled distance, in order to make the service as efficient as possible, on the other hand we have to balance the workload among nurses. Thus this component of the objective function is a kind of bottleneck (min-max), that is difficult to address with OR methods. Finally the loyalty factor is component of the objective function that makes our problem very peculiar, not to say unique.

Various works consider how to solve the TSP, or its variant with Time Windows, in CP or with hybrid algorithms [6, 12, 8]; the TSP is only a component of the HHC problem.

9 Conclusions

In this paper, we presented a Constraint Logic Programming application for a Home Health Care problem. We modelled the problem that is currently solved by hand by the nurses of the National Health Service unit of the city of Ferrara, in Italy. The novelty of the model stands in two issues that are peculiar of the problem in Ferrara. The first issue is the objective: reducing the disparities in workload of the nurses, while at the same time improving the quality of service from the patients' viewpoint, by keeping minimal the number of different nurses that take care of a same patient. The second issue is the implementation of a new constraint that addresses the routing component of the problem. The constraint was implemented by embedding into a constraint an efficient solver for the Travelling Salesperson Problem. Although the new constraint is implemented through a technique used in Operations Research (namely, Lagrangian Relaxation), it has a clear logical

semantics, that smoothly integrates into the constraint model.

We implemented various general-purpose search-strategies, then we moved to a new search strategy that is more tailored to the given problem, obtaining strong improvements.

Experimental results show a large improvement with respect to the currently used solutions, showing that Logic Programming can be effective to address real life problems.

The logic program consists of about 300 lines of ECL^{i}PS^e code, including custom constraints, and the interfacing to the Java TSP solver, plus about 600 lines of Java code. The main objective of the application was to provide to nurses more balanced workloads, and to patients more continuity (loyalty) in the service. In other words, the objective was to improve the feeling of the quality of working conditions for the nurses and the perception of the quality of service for the patients. However, as a by-product, we also reduced the workload of the nurses, in terms of travel time. With respect to the hand-made solution, a nurse saves about 3 hours per week. In this way, the working time of the nurses is used more effectively to provide service to patients, instead of travelling on sub-optimal routes. The saved time could be used to provide better service to the patients, or to serve more patients, which is a strong improvement in a period of crisis and governmental cuts.

The application was mainly designed, developed and tested by two PhD students in about six months. As a rough estimate, the person-months for the development will be returned in about 8 months, which shows that Logic Programming can be an economically affordable technology to improve working conditions and service quality.

References

1. David L. Applegate, Robert E. Bixby, Vasek Chvátal, and William J. Cook. *The Traveling Salesman Problem: A Computational Study.* Princeton University Press, 2006.
2. K. Apt and M. Wallace. *Constraint logic programming using ECL^{i}PS^e.* 2007.
3. Thomas Bartz-Beielstein, Marco Chiarandini, Luís Paquete, and Mike Preuss, editors. *Experimental Methods for the Analysis of Optimization Algorithms.* Springer, Germany, 2010.
4. N. Beldiceanu and E. Contejean. Introducing global constraints in CHIP. *Mathematical and Computer Modelling*, 20(12):97 – 123, 1994.
5. S. Bertels and T. Fahle. A hybrid setup for a hybrid scenario: combining heuristics for the home health care problem. *Computers & OR*, 33(10), 2006.
6. Yves Caseau and François Laburthe. Solving small TSPs with constraints. In L. Naish, editor, *ICLP*. The MIT Press, 1997.
7. P. Eveborn, M. Rönnqvist, H. Einarsdóttir, M. Eklund, K. Lidén, and M. Almroth. Operations research improves quality and efficiency in home care. *Interfaces*, 2009.
8. Filippo Focacci, Michela Milano, and Andrea Lodi. Soving TSP with time windows with constraints. In Danny De Schreye, editor, *ICLP*, pages 515–529. MIT Press, 1999.
9. R. Herrero, J.J. Ramos, and D. Guimarans. Lagrangian metaheuristic for the travelling salesman problem. In *Extended abstracts of Operational Research Conference 52*, Royal Holloway, University of London, September 2010.
10. L. Kaya and J. Hooker. A filter for the circuit constraint. In F. Benhamou, editor, *CP*, volume 4204 of *Lecture Notes in Computer Science*, pages 706–710. Springer, 2006.
11. M. Luby, A. Sinclair, and D. Zuckerman. Optimal speedup of Las Vegas algorithms. *Inf. Process. Lett.*, 1993.
12. G. Pesant, M. Gendreau, J-Y. Potvin, and J-M. Rousseau. An exact constraint logic programming algorithm for the TSP with time windows. *Transp. Science*, 32(1), 1998.
13. J.-C. Régin. A filtering algorithm for constraints of difference in CSPs. In B. Hayes-Roth and R. Korf, editors, *AAAI*, pages 362–367. AAAI Press / The MIT Press, 1994.
14. H. Simonis. Models for global constraint applications. *Constraints*, 12:63–92, 2007.

A Flexible Solver for Finite Arithmetic Circuits

Nathaniel Wesley Filardo and Jason Eisner

Department of Computer Science
Johns Hopkins University
3400 N. Charles St., Baltimore, MD 21218, USA
`http://cs.jhu.edu/~{nwf,jason}/`
{nwf, jason}@cs.jhu.edu

Abstract

Arithmetic circuits arise in the context of weighted logic programming languages, such as Datalog with aggregation, or Dyna. A weighted logic program defines a generalized arithmetic circuit—the weighted version of a proof forest, with nodes having arbitrary rather than boolean values. In this paper, we focus on finite circuits. We present a flexible algorithm for efficiently *querying* node values as they change under *updates* to the circuit's inputs. Unlike traditional algorithms, ours is agnostic about which nodes are tabled (materialized), and can vary smoothly between the traditional strategies of forward and backward chaining. Our algorithm is designed to admit future generalizations, including cyclic and infinite circuits and propagation of delta updates.

1998 ACM Subject Classification F.1.1 Models of Computation, I.2.3 Deduction and Theorem Proving

Keywords and phrases arithmetic circuits, memoization, view maintenance, logic programming

Digital Object Identifier 10.4230/LIPIcs.ICLP.2012.425

1 Introduction

The weighted logic programming language Dyna [10] is a convenient and modular notation for specifying derived data. In this paper, we begin to consider efficient algorithms for answering queries against Dyna programs. Our methods treat arithmetic circuits, and are relevant to other variants of logic programming, such as Datalog with aggregation [15, 5].

Many tasks in computer science involve computing and maintaining derived data. *Deductive databases* [21] store **extensional** (i.e., provided) data but also define additional **intensional** data specified by formulas. Algorithms in artificial intelligence or business analytics can often be written in this form [10]. The extensional data are observed facts, and the resulting cascades of intensional data arise from aggregation, record linkage, analysis, logical reasoning, statistical inference, or machine learning.

If the extensional data can change over time, keeping the intensional data up to date is called *view maintenance* or *stream processing* [23]. This pattern includes traditional *abstract data types*, which maintain derived data under operations such as "insert" and "remove." For example, a priority queue maintains the argmax of a function over an extensional set.

A Dyna program is a *declarative* specification of derived data. Like an abstract data type, it admits many correct implementations of its update and query methods. These execution strategies range from the laziest ("store the update stream and scan it when queried") to the most eager ("recompute all intensional data upon every update"). A particular strategy might trade time for space, or more time now for less time later (e.g., investing time in finding a faster query plan or maintaining an index). We seek a unified algorithm that subsumes as many reasonable strategies as possible, and which supports transitioning smoothly between

© Nathaniel Wesley Filardo and Jason Eisner;
licensed under Creative Commons License ND
Technical Communications of the 28th International Conference on Logic Programming (ICLP'12).
Editors: A. Dovier and V. Santos Costa; pp. 425–438

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

```
1  Compute(j ∈ I_int)
2    return f_j ({i ↦ Lookup(i) | i ∈ P_j})
3
4  Lookup(j ∈ I)
5    v ← M[j]
6    if v = UNK then v ← Compute(j)
7    maybe M[j] ← v
8    return v
```

Listing 1 Internals of basic backward chaining with optional memoization. $\mathcal{M}$ stores values for extensional items and initially stores UNK for intensional items.

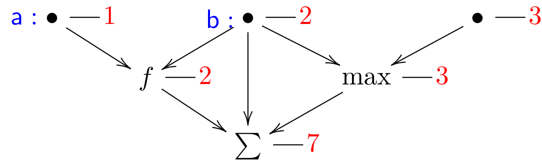


Figure 1 An example arithmetic circuit on the natural numbers, showing the function for each intensional item (f, max, $\sum$, where $f(a, b) = b^a$) and the symbol • for each extensional item. Item values are shown in red, and selected item names in blue.

them. This allows different strategies to be selected for different parts of the program (static analysis) or for different workloads (dynamic analysis).

In the present work, we discuss the development of a generic algorithm for finding and maintaining solutions to finite *arithmetic circuits*, which are a subset of Datalog and Dyna programs. Our algorithm offers several degrees of freedom, which will allow us to compare various static and adaptive strategies in future. The algorithm can choose any initial guess for the circuit's solution; its agenda of pending computations may be processed in any order; and it contains **maybe** directives where the algorithm has an additional free choice. Thus, our algorithm smoothly interpolates among traditional strategies such as forward chaining, backward chaining, and backward chaining with memoization.

2 Arithmetic Circuits

An **arithmetic circuit** [4] is a finite directed acyclic graph on nodes $\mathcal{I}$ with edges $\mathcal{E}$. We refer to the nodes as **items**. We denote item j's set of **parents** by $P_j \stackrel{\text{def}}{=} \{i \mid (i \to j) \in \mathcal{E}\}$, and its set of **children** by $C_j \stackrel{\text{def}}{=} \{k \mid (j \to k) \in \mathcal{E}\}$. Transitive parents are called **ancestors**, and transitive children are called **descendants**. We use the term **generalized arithmetic circuit** for the more general case where the graph may be infinite and/or cyclic.

Each item $i \in \mathcal{I}$ has a **value** in some set $\mathcal{V}$. Figure 1 shows a small arithmetic circuit over integer values. The **root** or **input** items, those without parents, are denoted $\mathcal{I}_{\text{ext}}$ ("extensional") and receive their values from the environment.[1] The remaining items are denoted $\mathcal{I}_{\text{int}}$ ("intensional") and derive their values by rule from their parents' values. Each item $j \in \mathcal{I}_{\text{int}}$ is equipped with a function f_j to combine its parents' values. The input to f_j is not merely an unordered collection of the parents' values. Rather, to specify which parent has which value, it is a *map* $P_j \to \mathcal{V}$, consisting of a collection of pairs $i \mapsto v$.

A Datalog or pure Prolog program can be regarded as a concise specification of a **generalized boolean circuit**, which is the case where $\mathcal{V} = \{\text{TRUE}, \text{FALSE}\}$. The items $\mathcal{I}$ correspond to propositional terms of the logic program, and clauses of the logic program describe how to discover the parents or children of a given item (on demand). Specifically, each grounding of a clause corresponds to an AND node whose parents are the body items, and whose child is an OR node corresponding to the head item. This kind of circuit is called an AND/OR graph.

[1] The literature varies as to whether extensional items are called roots or leaves, whether they are regarded as ancestors or descendants, and whether they are drawn at the top or the bottom of a figure. We treat them as roots and ancestors and draw them at the top. So edges and information flow downward in our drawings. As a result, "bottom-up" reasoning (forward chaining) actually proceeds from the top of the drawing down.

Datalog is sometimes extended to allow limited use of NOT nodes as well.

Arithmetic circuits are the natural analogue of boolean circuits for *weighted* logic programming languages, such as Datalog with aggregation [15, 5] and our own Dyna [11, 9].

Suppose we are given a generalized arithmetic circuit, along with a map $\mathcal{S} : \mathcal{I}_{\text{ext}} \to \mathcal{V}$ that specifies the extensional data. A **solution** to the circuit is an extension $\overline{\mathcal{S}}$ of this map over all of $\mathcal{I}$ such that $\overline{\mathcal{S}}[j] = f_j\left(\{i \mapsto \overline{\mathcal{S}}[i]\} \mid i \in P_j\right)$ for each $j \in \mathcal{I}_{\text{int}}$. In the traditional case where the circuit is finite and acyclic a solution $\overline{\mathcal{S}}$ always exists and is unique. This paper considers only that case, which also ensures that our algorithms always terminate. However, we will avoid using methods that rely strongly on finiteness or acyclicity. This makes our methods relevant to the harder problem of solving *generalized* arithmetic circuits (see section 7), as needed for the general case of weighted logic programming languages.

3 Backward Chaining

We begin with some basic strategies for querying an item's solution value $\overline{\mathcal{S}}[j]$, based on **backward chaining** from the item to its ancestors. We construct a map $\mathcal{M}$ from items to their solution values, known as the **memo table** or **chart**. For each extensional item $j \in \mathcal{I}_{\text{ext}}$, we initialize $\mathcal{M}[j]$ to $\mathcal{S}[j]$ $(= \overline{\mathcal{S}}[j])$. For each intensional items $j \in \mathcal{I}_{\text{int}}$, the solution value $\overline{\mathcal{S}}[j]$ is initially *unknown*, so we initialize $\mathcal{M}[j]$ to the special object UNK $\notin \mathcal{V}$. We may regard the map $\mathcal{M} : \mathcal{I} \to \mathcal{V} \cup \{\text{UNK}\}$ as a *partial* map $\mathcal{I} \to \mathcal{V}$ that stores actual values for only some items—initially just the extensional items.

We define mutually recursive functions LOOKUP and COMPUTE as in Listing 1. A user may query the solution with LOOKUP(j). This returns $\mathcal{M}[j]$ if it is known, but otherwise calls COMPUTE(j) to compute j's value using f_j, which in turn requires LOOKUPs at j's parents.

Pure Backward Chaining The simplest form of backward chaining simply recurses through ancestors until LOOKUP reaches the roots. Line 7 is never used in this case, so $\mathcal{M}$ never changes and intensional items remain as UNK. Clearly LOOKUP(j) returns $\overline{\mathcal{S}}[j]$.

Unfortunately, pure backward chaining can have runtime exponential in the size of the circuit. Each call to LOOKUP(j) will in effect enumerate all *paths* to j. For example, consider a circuit for computing Fibonacci numbers, where each item $\mathsf{fib}(n)$ for $n \geq 2$ is the sum of its parents $\mathsf{fib}(n-1)$ and $\mathsf{fib}(n-2)$. Then LOOKUP($\mathsf{fib}(n)$) has runtime that is exponential in n, with $\mathsf{fib}(n-t)$ being repeatedly computed $\mathsf{fib}(t)$ $(\approx O(1.618^t))$ times during the recursion.

Optional Memoization To avoid such repeated computation, a call to LOOKUP(j) can **memoize** its work by caching the result of COMPUTE(j) in $\mathcal{M}[j]$ for use by future calls, via line 7 of Listing 1. This is the backward-chaining version of dynamic programming. It generalizes the node-marking strategy that depth-first search uses to avoid re-exploring a subgraph. However, the `maybe` keyword in line 7 indicates that the memoization step is not required for correctness; it merely commits space in hopes of a future speedup. LOOKUP($\mathsf{fib}(n)$) can even achieve $O(n)$ expected runtime without memoizing all recursive LOOKUPs: instead it can memoize LOOKUPs on a systematic subset of items, or on a random subset of calls.

4 Reactive Circuits: Change Propagation

Our goal is to design a dynamic algorithm for arithmetic circuits that supports not just **queries** but also **updates**. It must handle a stream of operations of the form QUERY(j) for

```
 1  RunAgenda()
 2    until 𝒜 = ∅
 3      pop i : ⇐v from 𝒜
 4      if v = UNK then v ← Compute(i)
 5      if v ≠ ℳ[i] then   % else discard
 6        ℳ[i] ← v
 7        foreach j ∈ C_i
 8          w ← UNK
 9          maybe w ← Compute(j)
10          Update(j, w)
```

Listing 2 The core of an agenda-driven, tuple-at-a-time variant of the traditional forward chaining algorithm. $\mathcal{M}$ is initialized to an arbitrary but total guess and remains total (no UNK values) thereafter. Hence, though Compute calls Lookup, Lookup never recurses back to Compute.

```
1  Update(j ∈ ℐ, w ∈ 𝒱 ⊔ {UNK})
2    delete 𝒜[j]
3    if w ≠ ℳ[j] then   % else discard
4      𝒜[j] ← ⇐w
```

Listing 3 Updates requested by the user or by RunAgenda are enqueued on the agenda $\mathcal{A}$ as replacement updates.

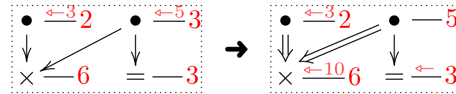


Figure 2 An example iteration of the loop in RunAgenda. We apply the update $\Leftarrow 5$ to the right parent, which makes the children inconsistent with their parents, and enqueue new updates that will fix the inconsistencies. Double arrows indicate the edges used to Compute the new value in the replacement update: 10 is 2×5.

any j, which returns $\overline{\mathcal{S}}[j]$, and Update($i$,$v$) for $i \in \mathcal{I}_{\text{ext}}$, which modifies $\mathcal{S}[i]$ to $v \in \mathcal{V}$.[2]

In the case of pure backward chaining, we only have to maintain the stored extensional data, as intensional values are not stored, but are derived from the extensional data on demand. In our terminology from above, Update(i,v) can just set $\mathcal{M}[i] \leftarrow v$, and Query($j$) can just call Lookup($j$).

However, handling updates is harder once we allow memoization of intensional values. The memos in $\mathcal{M}$ grow **stale** as external inputs change, yet Lookup would continue to return outdated results based on these memos. That is, updating i may make its intensional descendants inconsistent; this must be rectified before subsequent queries are answered. We therefore need some mechanism for restoring consistency in $\mathcal{M}$, by propagating changes to memoized descendants.

Formally, we say that $j \in \mathcal{I}_{\text{ext}}$ is **consistent** iff Lookup(j)$= \mathcal{S}[j]$, and that $j \in \mathcal{I}_{\text{int}}$ is **consistent** iff Lookup(j) = Compute(j). Notice that un-memoized intensional items (those with $\mathcal{M}[j] = $ UNK) are always consistent. We call $\mathcal{M}$ consistent if all items are consistent—in this case Lookup(j) will return the solution $\overline{\mathcal{S}}[j]$ as desired. Equivalently, the memo table $\mathcal{M}$ is consistent iff each extensional memo is correct and each intensional memo is in agreement with its visible ancestors. Here i and k are said to be **visible** to each other whenever there is a directed path from i to its descendant k that goes only through un-memoized (UNK) items. Thus, calling Compute(k) eventually recurses to Lookup(i) at each visible parent i.

5 Pure Forward Chaining

An alternative solution strategy, **forward chaining**, propagates updates. We will use it in section 6 to solve the update problem. First we present forward chaining in its pure form.

Pure forward chaining *eagerly* fills in the *entire* chart $\mathcal{M}$, starting at the roots and visiting children after their parents. Eventually $\mathcal{M}$ converges to $\overline{\mathcal{S}}$. Forward chaining algorithms include natural-order recalculation in spreadsheets [29] and semi-naive bottom-up evaluation for Datalog [28]. We use the "tuple-at-a-time" algorithm of Listing 2. It uses an **agenda** $\mathcal{A}$

[2] We also wish to support *continuous queries*, in which the user may request (asynchronous) notifications when specified items change value. This is, however, beyond the scope of the current paper.

that enqueues *future* **updates** to the chart [17, 11]. $\mathcal{A}$ contains at most one update for each item i, which we denote $\mathcal{A}[i]$, and supports modification or deletion of this update.[3]

Our updates are **replacement updates** of the form $i : \underline{\leftarrow v}$ (where $i \in \mathcal{I}$ and $v \in \mathcal{V}$).[4] Iteratively, until the agenda is empty, our forward chaining algorithm **pops** (selects and removes) any update $i : \underline{\leftarrow v}$ from the agenda, and **applies** it to the chart by setting $\mathcal{M}[i] \leftarrow v$. The algorithm then **propagates** this update to i's children, by **pushing** an appropriate update $j : \underline{\leftarrow w}$ onto the agenda for each child j. This push operation overwrites any previous update to j, so we write it as $\mathcal{A}[j] \leftarrow \underline{\leftarrow w}$.

The new value w is obtained by `Compute(j)`, meaning it is recomputed from the values at j's parents (including the changed value at i). If $\mathcal{M}[j]$ already had value w, the update is immediately discarded and does not propagate further. Ordinarily, w is `Compute`d in line 9 when the update is constructed and pushed. But if line 9 is optionally skipped, the update specifies w as UNK, meaning to compute the new value only when the update is popped and actually applied (line 4). Such a **refresh update** $j : \underline{\leftarrow \text{UNK}}$ may be abbreviated as $j : \underline{\leftarrow}$ and simply says to refresh j's value so it is consistent.[5] In any case, *an inconsistent item always has an update pending on the agenda*, which will eventually make it consistent.[6]

Figure 2 shows one step of pure forward chaining. In our visual notation for circuits, we draw the state of item i as $i : f_i \;—\mathcal{M}[i]$, where i (if present) names the item, f_i is the item's function (or $\bullet$ if $i \in \mathcal{I}_{\text{ext}}$), and $\mathcal{M}[i]$ is the current memo if any. If an update to i is waiting on the agenda, we display it over i's line as $i : f_i \overset{\leftarrow v}{—}\mathcal{M}[i]$, omitting the new value v if it is UNK. Since information flows downward in our drawings, being *above* i's line indicates that the update has yet to be applied to $\mathcal{M}[i]$. (In section 6.1 we will introduce a below-the-line notation.) Our textual update notation $i : \underline{\leftarrow v}$ is intended to resemble the drawing.

The process can be started from any total (UNK-free) initial chart $\mathcal{M}$, provided that the initial agenda $\mathcal{A}$ is sufficient to correct any inconsistencies in this $\mathcal{M}$. $\mathcal{A}$ is always sufficient if it updates *every* item: so the **conservative initialization strategy** defines each $\mathcal{A}[i]$ to be $i : \underline{\leftarrow \mathcal{S}[i]}$ for extensional i, and either $j : \underline{\leftarrow \texttt{Compute}(j)}$ or $j : \underline{\leftarrow}$ for intensional j. However, just as Listings 2–3 discard unnecessary updates, we can also omit as unnecessary any initial updates to items that are consistent in the initial $\mathcal{M}$. So we may wish to choose our initial $\mathcal{M}$ to be mostly consistent. For example, under the **NULL initialization strategy**, we initialize $\mathcal{M}[i]$ to a special value NULL $\in \mathcal{V}$ for all $i \in \mathcal{I}$. Provided that each function f_j outputs NULL whenever all its inputs are NULL, each intensional j is initially consistent and hence requires no initial update.[7]

The user method `Query(j)` is now defined as `RunAgenda(); return Lookup(j)`. This runs

---

[3] The agenda can be implemented as a simple dictionary. However, using an adaptable priority queue [14] can speed convergence, if one orders the updates topologically or by some informed heuristic [18, 12].

[4] It will be explained shortly why an underline appears in the notation for this type of update.

[5] Why are there two kinds of updates? Both have potential advantages. Refresh updates ensure that j is only recomputed once, even if the parents change repeatedly before the update pops. On the other hand, ordinary updates have the chance of being discarded immediately, which avoids the expense of pushing and popping any update at all; and if they are not discarded, their priority order can be affected by knowledge of w. Later algorithms in this paper cache item values temporarily, with the result that the cost of computing w may vary depending on when `Compute(j)` is called. Finally, delta updates (section 7) must be computed at push time.

[6] A consistent item might also have an update pending—a refresh that is not yet known to be unnecessary.

[7] In a logic programming setting, updating $\mathcal{M}[j]$ from NULL to non-NULL may be regarded as "proving j." Forward chaining proves the extensional items from the initial agenda, and then propagation causes it to prove some or all of the intensional items. In *unweighted* logic programming, NULL may be interpreted as "not proven" and identified with `FALSE`. Both `AND` and `OR` functions then have the necessary property. Similarly, in *weighted* logic programming, NULL means "no proven value." Here Dyna [9] again uses functions that guarantee the necessary property, by extending arithmetic functions (which generalize `AND`) as well as aggregation functions (which generalize `OR`) over the domain that includes NULL.

the agenda to completion and then returns $\mathcal{M}[j]$. As for the user method $\textsc{Update}(i, v)$, the user is permitted to call Listing 3 in this case, thereby pushing a new update onto the agenda. Forward chaining processes all such updates at the start of the next query. This does not require recomputing the whole circuit.

It may be instructive at this point to contemplate the physical storage of the map $\mathcal{M} : \mathcal{I} \to \mathcal{V} \cup \{\text{UNK}\}$ (where $\text{NULL} \in \mathcal{V}$). A large circuit may be compactly represented by a much smaller logic program (section 2). In this case one might also hope to store $\mathcal{M}$ compactly in space $o(|\mathcal{I}|)$, using a sparse data structure such as a hash table. The "natural" storage strategy is to treat UNK as the default value in the case of backward chaining, but to treat NULL as the default value in the case of forward chaining. In each case this means that initialization is fast because intensional items are not *initially* stored. Backward chaining then adds items to the hash table only if they are queried (and memoized), while forward chaining adds them only if they are provable (see footnote 7). The *final* storage size of $\mathcal{M}$ may differ in these two cases owing to the different choice of default. It can be more space-efficient—particularly in our hybrid strategy below—to choose different defaults for different types of items, reflecting the fact that some type of item is "usually" UNK or NULL (or even 0). One stores the pair $(i, \mathcal{M}[i])$ only when $\mathcal{M}[i]$ differs from the default for i. The datatype used to store $\mathcal{M}[i]$ does not need to be able to represent the default value.

6 Mixed Chaining With Selective Memoization

Both pure algorithms above are fully reactive, but sometimes inefficient. Backward chaining may redo work. Forward chaining requires storage for all items, and updates fully before answering a query. Yet each has advantages. Backward chaining visits only the nodes that are needed for a given query; forward chaining visits only the nodes that need updating.

A hybrid algorithm should combine the best of both, visiting nodes only as necessary and using $\mathcal{M}$ to materialize some useful subset of them. Our core insight is that

- The job of backward chaining is to *compute values for which the memo is missing* (UNK).
- The job of forward chaining is to *refresh any memos that are present but potentially stale.*
- Pure backward chaining is the case where *all memos are missing*. So a query triggers a cascade of backward computation; but forward chaining is unnecessary (no *stale* memos).
- Pure forward chaining is the dual case where *all memos are present*. So an initial or subsequent update triggers a cascade of forward computation; but backward chaining is unnecessary (no *missing* memos). We regard the arbitrarily initialized chart of section 5 as a complete but potentially stale memo table.

We will develop a hybrid algorithm that can memoize any subset of the intensional items. This subset can change over time: memos are optionally created while answering queries by backward chaining, and can be freely created or flushed at any time. Why bother? Computing only values that are needed for a given query can reduce asymptotic time and space requirements, a fact exploited by the magic sets technique [25]. Furthermore, materializing some or all of these values only *temporarily* can reduce the cost of storing and maintaining many memos. For example, [30] thereby solve the arithmetic circuit for the forward-backward algorithm in $O(\log n)$ rather than $O(n)$ space, while increasing runtime only from $O(n)$ to $O(n \log n)$.

6.1 Updates vs. Notifications

The essential (and novel) challenge here is to make forward chaining work with an incomplete memo table $\mathcal{M}$. Intuitively, we merely need to propagate updates as usual down through

unmemoized regions of the circuit, so that they reach and refresh any stale memos below.

However, updates in such a region have a different nature. When we update the memo for an item i, each *visible* unmemoized descendant j remains *consistent* (in the terminology of section 4). After all, the result of calling Lookup(j) would *already* reflect the change to i.

Thus, what we propagate to j is not really an update but a **notification**. It does not say "change the value of j," but rather "the value of j has already [implicitly] changed." Crucially, this notification must be propagated to the descendants of j. When it finally reaches i's visible *memoized* descendants k—which became inconsistent the moment that i's memo was updated—it will trigger updates there to repair the inconsistencies.[8]

The agenda $\mathcal{A}$ now contains two kinds of **messages**: $\mathcal{A}[j]$ may be either an update to j or a notification from j. Recall from section 5 that an update to j is graphically displayed *above* the line. A notification from j is drawn as $j : f \,\overline{\leftarrow}$, with the change displayed *below* the line to indicate that it has already descended through item j. In this paper, the change is always a replacement by an unspecified (UNK) value, written textually as $j : \overline{\leftarrow}$.[9]

6.2 Push-Time Updates and Invalidations

The resulting code is shown in Figure 3. Our code also takes the opportunity to exploit notifications even for memoized items. In the old Listing 3, Update(j, w) always enqueued $j : \underline{\leftarrow w}$ for later. Our new Update(j, w) in Listing 4 can still choose that option provided that j is memoized (Line 4:8, a **pop-time update**), but its default is to Apply the update immediately (Line 4:14, a **push-time update**). If so, it pushes only the notification $j : \overline{\leftarrow}$ and there is no need to Apply the update at pop time. What *does* still happen at pop time is propagation: it is not until we pop an update *or* a notification to j (Line 5:5) that we visit j's children (Line 6:2).[10]

What happens if Line 6:4 is optionally skipped (so that $w = \text{UNK}$)? Then the resulting Update is a refresh update as before (section 5) if processed at pop time. However, if processed at push time, it is an **invalidation** update that deletes a memo instead of correcting it. Propagating invalidations can clear out stale portions of the chart at lower computational cost. Separately, the Flush method can also be called by the user or by FreelyManipulateM (Listing 5) to delete individual memos without the need to propagate.

Like the forward chaining algorithm, the hybrid algorithm may start from any initial chart $\mathcal{M}$—but intensional items j now have the option of $\mathcal{M}[j] = \text{UNK}$. The initial agenda does not contain any notifications, but as before, it must include enough updates to correct any inconsistencies in the initial chart. Since unmemoized intensional UNK items are always consistent by definition (section 4), the initial agenda never needs to have updates for them. For example, the **UNK initialization strategy** initializes just as in backward chaining (section 3), with extensional items set correctly and everything else initially UNK.

---

[8] This algorithm has a more complicated invariant than that of section 5: When k is inconsistent, the agenda contains *either* an update at k (as in section 5) *or* a notification at some visible ancestor of k.

[9] However, in general, *any* change that can appear in an update could appear in a notification, e.g., a more specific replacement $\overline{\leftarrow w}$, or a delta $\overline{\oplus w}$ (see section 7).

[10] Why allow both pop-time and push-time updates? Pop-time updates are required for correctness in certain settings involving delta updates (section 7). Also, pop-time updates include refresh updates, which are useful in avoiding premature computation of the new value w (footnote 5). On the other hand, push-time updates ensure fresher lookup results by immediately updating $\mathcal{M}[j]$ to a new value (or invalidating it to UNK). If the same update is deferred to pop time, then any calls to Lookup(j) while the update is waiting on the agenda will unfortunately get a stale memo for j, resulting in stale descendants that must be updated after the update pops.

6.3 Correctness: Avoiding A Subtle Bug

Returning to the setting of section 6.1, again suppose that i was updated, making its descendant k inconsistent, and that j is an unmemoized intermediate item on an i-to-k path.

Updating some *other* visible descendant of j (i.e., other than k) may cause j to get recursively looked up and optionally memoized before its notification arrives. If j gets memoized, it will receive an update rather than a notification. But the COMPUTE(j) that computes the update value will get the same answer as the COMPUTE(j) that computed the memo. That is, the memo $\mathcal{M}[j]$ was not stale but already reflected the change to i. This causes a subtle bug: forward chaining will discard the apparently unnecessary update, rather than propagating it on downward to k. Thus, k may remain inconsistent forever.

To prevent this bug, memoizing j must also enqueue a notification that the memo at j has been updated. The correct behavior is illustrated in Figure 4. This notification reflects the past update to i; it restores the invariant mentioned in footnote 8, and it will propagate down to k as desired. Such a notification must be enqueued when memoizing any item j such that COMPUTE(j) recursed to some item that had a notification on the agenda. The functions in Listing 7 return (as a second value) a flag that is TRUE if this condition holds, and enqueue the required notification at Line 7:22.

6.4 Efficiency: Obligation Tracking

Recall our challenge in section 4: backward chaining with optional memoization was a good algorithm, but to support UPDATE, we needed change propagation to refresh stale memos.

In our hybrid algorithm, we can use the UNK initialization strategy (section 6.2) to recover backward chaining. Change propagation will now be handled correctly.

Unfortunately, our propagation of notification through unmemoized regions is overly aggressive. For example, if *no intensional items have been memoized*, then change propagation should be completely unnecessary—this is the pure backward chaining case of section 4—and yet our algorithm will visit all descendants of an UPDATEd item! Our method visits all children of an updated item to check whether they too may need updating. In pure forward chaining, we can stop propagating (discard the update) at a child whose value is consistent; but for an UNK child the value is unknown, so we conservatively keep propagating.

In general, we should propagate down along an edge only when this may eventually reach a memoized descendant. This requires **obligation tracking**: for every item in the circuit, we desire to know if it has descendant memos which must be visited if its value changes.

We define the predicate $\text{obl}(\mathcal{A}[i], j)$ (used on Line 6:2) to mean that i is **obligated to** inform its child j of the update $\mathcal{A}[i]$. By definition, this is so if j is memoized or is in turn obligated to any of j's children. As a result, obligation of items in an arithmetic circuit $\mathcal{C}$ is naturally expressed as a boolean circuit $\mathcal{C}_{\text{obl}}$ that determines transitive reachability. Roughly speaking, $\mathcal{C}_{\text{obl}}$ has the same topology as $\mathcal{C}$ but with the edge direction reversed.

We can be even more precise about determining obligation. Specifically, in the recursive definition, i is *not* obligated to its child j if there is a notification at i or a refresh update at j. In these cases, j and its descendants are guaranteed to get refreshed anyway, so it is not necessary to propagate messages to j from i or its ancestors. One can again express this tighter definition as a boolean circuit $\mathcal{C}_{\text{obl}}$, whose boolean inputs are updated as $\mathcal{M}$ and $\mathcal{A}$ evolve. Line 6:2 then queries this $\mathcal{C}_{\text{obl}}$ using our algorithm.

We can maintain $\mathcal{C}_{\text{obl}}$ in turn using $(\mathcal{C}_{\text{obl}})_{\text{obl}}$, or by falling back to a *cheaper* obligation tracking strategy at this stage. For example, obligation tracking is cheap on a circuit that uses a memoization and flushing policy such that the memoized items always have memoized

```
1   Query(i ∈ I)
2     RunAgenda()
3     (v, ·) ← Lookup(i)    % · will be FALSE
4     return v
5
6   Update(j ∈ I, w ∈ V ⊔ {UNK})
7     maybe
8       if (A[j] ≠ ⥪) ∧ (M[j] ≠ UNK) then
9         delete A[j]
10        if (w = UNK) ∨ (M[j] ≠ w) then
11          A[j] ← ←w
12        return
13      % else fall through
14    Apply(j, w)
15
16  Flush(j ∈ I_int)
17    if A[j] = ←· then A[j] ← ⥪
18    M[j] ← UNK
```

Listing 4 User interface methods. (A user call to Update must have $j \in \mathcal{I}_{\text{ext}}, w \in \mathcal{V}$.)

```
1   Propagate(i ∈ I)
2     foreach j ∈ C_i such that obl(A[i], j)
3       w ← UNK
4       maybe (w, ·) ← Compute(j)
5       Update(j, w)
6     delete A[i]
7
8   % Convert update to notification
9   HandleUpdate(i : ←v)
10    v_cur ← M[i]   % will not be UNK
11    maybe if v = UNK then
12      (v, ·) ← Compute(i)
13    if v ≠ v_cur then   % else discard
14      foreach j ∈ C_i maybe Lookup(j)
15      maybe v ← UNK
16      Apply(i, v)
17
18  Apply(j ∈ I_int, w ∈ V ⊔ {UNK})
19    M[j] ← w
20    A[j] ← ⥪
```

Listing 6 Forward chaining internals.

```
1   RunAgenda()
2    until A = ∅
3     FreelyManipulateM()
4     peek u from A
5     case u of
6       i : ⥪ → Propagate(i)
7       · : ←· → HandleUpdate(u)
8
9   FreelyManipulateM()
10   done ← FALSE
11   until done
12    foreach i ∈ I_int maybe Lookup(i)
13    foreach i ∈ I_int maybe Flush(i)
14    maybe done ← TRUE
```

Listing 5 Nondeterministic high-level control.

```
1   % Derive j's value from parents
2   Compute(j ∈ I_int)
3     foreach i ∈ P_j
4       (v_i, m_i) ← LookupFromBelow(i)
5     FreelyManipulateM()
6     return (f_j ({i ↦ v_i | i ∈ P_j}),
7               max_{i∈P_j} m_i)
8
9   % Interaction with forward chaining
10  LookupFromBelow(i ∈ I)
11    m_c ← (A[i] = ⥪)
12    (v, m_t) ← Lookup(i)
13    return (v, m_c ∨ m_t)
14
15  % Derive i's value from memo or parents
16  Lookup(i ∈ I)
17    if M[i] ≠ UNK then
18      return (M[i], FALSE)
19    (v,m) ← Compute(i)
20    maybe
21      M[i] ← v
22      if m then A[i] ← ⥪
23    return (v, m)
```

Listing 7 Backward chaining internals.

Figure 3 The internals of our initial mixed-chaining algorithm, which combines forward and backward reasoning and supports arbitrary Flushes during execution.

parents. In that case, i is obligated to its child j only when j is memoized, which can be checked directly without an auxiliary circuit. Also, obligation tracking tolerates one-sided error: it is always safe for an obligation query to conservatively return TRUE, which at worst just results in unnecessary propagation. This leads to cheap approximate obligation tracking strategies, such as always returning TRUE, or coarse-to-fine approximations where the circuits $\mathcal{C}, \mathcal{C}_{\text{obl}}, (\mathcal{C}_{\text{obl}})_{\text{obl}}, \ldots$ are progressively smaller because a node in one circuit corresponds to a *set* of nodes in the previous circuit and is TRUE if *any* of them are obligated.

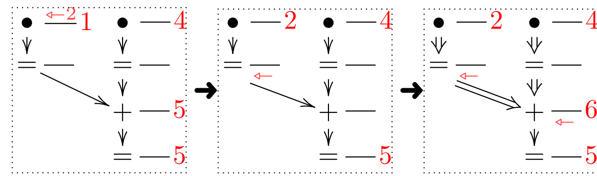


Figure 4 Backward chaining may need to enqueue notifications. After the top left update ("i") propagates to a notification $\overline{\Leftarrow}$ at its child, the + item ("j") is flushed. Backward chaining from the + item (through un-memoized items) memoizes an up-to-date result of 6. Because backward chaining encountered a $\overline{\Leftarrow}$, the memoization enqueues another $\overline{\Leftarrow}$ at the + item, which ensures that its child ("k") will later be updated from 5 to 6.

6.5 Related Work

The recent constraint solver Kangaroo [24] was independently motivated by similar concerns. Like us, it mixes backward and forward chaining. In Kangaroo, queries seek out relevant updates—the reverse of our obligation approach, in which updates seek out relevant memoized queries. We are more selective about storage than Kangaroo, which stores memos at *all* nodes of the circuit.[11] On the other hand, Kangaroo is more selective about runtime. While it may have more memos, it updates only stale memos that are relevant to current queries, whereas our current algorithm updates all stale memos.

Previous mixed-chaining algorithms have been simpler. For functional programming, Acar et al. [1, 3] answer queries by backward chaining with *full* memoization; they update these memos by forward chaining of *replacement* updates. The same strategy is used for Prolog by Saha and Ramakrishnan [26, 27], who contrast it with the "DRed" strategy that forward-chains *invalidation* updates [16]. The "magic sets" transformation for Datalog [25] can be seen as a variant of these strategies. It uses only forward chaining, but restricted to items that would have been visited by backward chaining from the given query. All of these strategies memoize every computed item. In contrast, we are more economical with space.

Acar et al. [2] do separately consider *selective* memoization, but do not handle updates in this more challenging case (see section 6). A different selective strategy [19] relies primarily on *unmemoized* backward chaining. It first performs forward chaining on a given sub-circuit to identify and memoize a subset of TRUE values. However, this relies on the special property of Datalog that a TRUE node of a sub-circuit is also TRUE in the full circuit.

We believe that our framework can naturally be extended with richer computational strategies (see section 7). This is because it integrates fully selective memoization with a mixed chaining strategy, and because it has a general notion of an agenda of pending work, which can support a variety of update types, prioritization heuristics, and parallelizations.

7 Extensions

Some of the following extensions to our hybrid algorithm of section 6 are not too difficult, and we sketch them here. We defer full treatments to a longer version of this paper.

Richer Vocabulary of Updates For simplicity, this paper has focused on replacement updates $i : \underline{\leftarrow v}$. However, our prototype of Dyna [11] actually used agenda-based forward chaining with **delta updates** such as $i : \underline{\oplus\, v}$ for some operator $\oplus$. Applying this update at

[11] Selective memoization is an added reason for mixed chaining. Our forward chaining sometimes invokes backward chaining, in order to re-COMPUTE the value of a stale item with an unmemoized parent.

pop time increments the old memo $\mathcal{M}[i]$ to $\mathcal{M}[i] \oplus v$. Similarly, Dijkstra's shortest-path algorithm [7] chooses to use forward chaining with push-time delta updates, which are applied immediately and push delta notifications $i : \overline{\oplus v}$ onto the agenda (where $\oplus$ is min). A delta update at i is sometimes cheap to propagate to j, compared to a replacement update. This is because one can sometimes avoid a full call to COMPUTE(j) at Line 6:4—which looks up or computes all the parents of j—by exploiting arithmetic properties such as distributivity of f_j over $\oplus$, or associativity and commutativity of $\oplus$ if $f_j = \oplus$.[12] Also, associativity and commutativity of $\oplus$ updates can be used to simplify the agenda data structure.

Circuit Transformation Even replacement updates can sometimes be propagated to j without a full call to COMPUTE(j). Consider the case where f_j aggregates a large set of parents P_j using an associative binary operator. We can statically or dynamically transform the circuit to replace the direct edges from P_j to j with a binary **aggregation tree**. As this tree is just part of the circuit, it can use any strategy. In particular, if we maintain memos at the tree's internal nodes, then we can propagate a change from $i \in P_j$ to j in time $O(\log |P_j|)$.

Circuits can also be rearranged into more efficient forms by refactoring arithmetic expressions. It is possible to carry out such rearrangements by transforming the weighted logic program from which the circuit is derived [8]. But in principle, one might also rearrange the circuit locally as inference proceeds.

Aborting Backward Chaining by Guessing Our algorithm can be extended to handle cyclic arithmetic circuits.[13] Pure forward chaining can propagate updates around cycles indefinitely in hopes that the memos will converge [11]. If so, it finds a fixed-point solution $\overline{\mathcal{S}}$. But backward chaining does not work on the same circuit: it can recurse around the cycles forever without ever making progress by creating a memo. There is an interesting solution.

In general, we can interrupt any long backward-chaining recursion by allowing COMPUTE(j) to optionally *guess* an arbitrary memo for $\mathcal{M}[j]$ (perhaps NULL). In this case we must enqueue a refresh update $j : \underline{\leftarrow}$, which serves as a continuation. Popping this update later will resume backward chaining and check that our guess at j is consistent with j's ancestors (perhaps including j itself, cyclically). If not, it will use the agenda to propagate a fix by forward chaining (perhaps cyclically until convergence). If j is already obligated to any children, we must also enqueue a notification $j : \overline{\leftarrow}$ to alert them that guessing $\mathcal{M}[j]$ may have changed it from the previous value of LOOKUP(j).

Fine-Grained Obligation Suppose j is an OR node whose parent i has value TRUE, or a $\times$ node whose parent i has value 0. As long as i has this value, j is *insensitive* to its other parents, who should not be obligated to propagate their updates to j. This generalizes the **watched variable trick** from the satisfiability community [22].

On-Demand Propagation Our current algorithm calls RUNAGENDA at the start of every QUERY, which refreshes all stale memos—including those that are not relevant to this query. This can be especially inefficient for cyclic or infinite circuits. We would prefer to propagate only the currently relevant updates, as in Kangaroo [24].

---

[12] This is slightly tricky when $\oplus$ is not idempotent, but solved in [11].

[13] Our current definition of obligation is overly broad in the cyclic case. It can create *self-supporting obligation*, where updates are unnecessarily propagated around cycles without actually refreshing any memos, merely because each item believes it is obligated to the next. Restoring efficiency in this case has been considered by [20].

Continuous Queries and Snapshots A **continuous query** of item i in an arithmetic circuit is a request to be notified (e.g., via callback) whenever UPDATES have caused QUERY(i) to change. Continuous queries are also used in databases and from functional reactive programming [13, 6]. Some users may also like to be notified of any updates that reach i as our algorithm runs, allowing them to **peek** at intermediate states $\mathcal{M}[i]$.

Programs We are actively working to extend the algorithms presented here to work not on arithmetic circuit descriptions directly but on Prolog-like weighted rules of Datalog with Aggregation [15, 5] and Dyna [9]. These programs can describe *infinite* generalized arithmetic circuits with value-dependent structure and with infinite fan-in or fan-out. A query, update, or memo may now be specified using a pattern that makes it apply to infinitely many items. This is the most challenging extension we have discussed.

8 Conclusion

We have developed a dynamic algorithm for solving arithmetic circuits and maintaining the solution under updates to the inputs. The solver can smoothly mix backward and forward chaining, while selectively memoizing results (and flushing memos). Different chaining and memoization strategies can be used as needed for different parts of the circuit, which does not affect correctness but can potentially improve time or space efficiency. Our framework also provides a basis for several extensions.

Acknowledgements This research was funded in part by the JHU Human Language Technology Center of Excellence. We would further like to thank John Blatz for useful early discussions, and an anonymous reviewer for calling our attention to Kangaroo [24].

References

1. Umut A. Acar, Guy E. Blelloch, and Robert Harper. Adaptive functional programming. In *Proc. of POPL*, pages 247–259, 2002.
2. Umut A. Acar, Guy E. Blelloch, and Robert Harper. Selective memoization. In *Proc. of POPL*, pages 14–25, 2003.
3. Umut A. Acar and Ruy Ley-Wild. Self-adjusting computation with Delta ML. In Pieter W. M. Koopman, Rinus Plasmeijer, and S. Doaitse Swierstra, editors, *Advanced Functional Programming*, volume 5832 of *Lecture Notes in Computer Science*, pages 1–38. Springer, 2008.
4. A. Borodin and I. Munro. *The computational complexity of algebraic and numeric problems.* Elsevier, 1975.
5. Sara Cohen, Werner Nutt, and Alexander Serebrenik. Algorithms for rewriting aggregate queries using views. In *Proc. of ADBIS-DASFAA*, pages 65–78, London, UK, 2000. Springer-Verlag.
6. Antony Courtney and Conal Elliott. Genuinely functional user interfaces. In *2001 Haskell Workshop*, September 2001.
7. Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.
8. Jason Eisner and John Blatz. Program transformations for optimization of parsing algorithms and other weighted logic programs. In Shuly Wintner, editor, *Proc. of FG 2006: The 11th Conference on Formal Grammar*, pages 45–85. CSLI Publications, 2007.

9 Jason Eisner and Nathaniel W. Filardo. Dyna: Extending Datalog for modern AI. In Tim Furche, Georg Gottlob, Oege de Moor, and Andrew Sellers, editors, *Datalog 2.0*, volume (to be published) of *LNCS*. Springer, 2011.

10 Jason Eisner and Nathaniel W. Filardo. Dyna: Extending Datalog for modern AI (full version). Technical report, Johns Hopkins University, 2011. Available at `dyna.org/Publications`. A condensed version appeared as [9].

11 Jason Eisner, Eric Goldlust, and Noah A. Smith. Compiling comp ling: Weighted dynamic programming and the Dyna language. In *Proc. of HLT-EMNLP*, pages 281–290, Vancouver, October 2005. Association for Computational Linguistics.

12 Gal Elidan, Ian Mcgraw, and Daphne Koller. Residual belief propagation: Informed scheduling for asynchronous message passing. In *Proceedings of the 22nd Conference on Uncertainty in Artificial Intelligence*, 2006.

13 Conal Elliott and Paul Hudak. Functional reactive animation. In *International Conference on Functional Programming*, 1997.

14 Michael T. Goodrich and Roberto Tamassia. *Data Structures and Algorithms in JAVA*. Wiley, 1998.

15 Sergio Greco. Dynamic programming in datalog with aggregates. *IEEE Transactions on Knowledge and Data Engineering*, 11(2):265–283, 1999.

16 Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In Peter Buneman and Sushil Jajodia, editors, *SIGMOD Conference*, pages 157–166. ACM Press, May 1993.

17 Martin Kay. Algorithm schemata and data structures in syntactic processing. In B. J. Grosz, K. Sparck Jones, and B. L. Webber, editors, *Readings in Natural Language Processing*, pages 35–70. Kaufmann, Los Altos, CA, 1986. First published in 1980 as Xerox PARC Technical Report CSL-80-12 and in the Proceedings of the Nobel Symposium on Text Processing, Gothenburg.

18 Dan Klein and Christopher D. Manning. A* parsing: Fast exact Viterbi parse selection. In *Proc. of HLT-NAACL*, 2003.

19 Thomas Labish. Developing a combined forward/backward-chaining system for logic programs in a hybrid expertsystem shell. Master's thesis, Universität Kaiserlautern, June 1993. In German.

20 Mengmeng Liu, Nicholas E. Taylor, Wenchao Zhou, Zachary G. Ives, and Boon Thau Loo. Recursive computation of regions and connectivity in networks. *IEEE 25th International Conference on Data Engineering*, 2009.

21 J.W. Lloyd and R.W. Topor. A basis for deductive database systems. *The Journal of Logic Programming*, 2(2):93 – 109, 1985.

22 Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient sat solver. In *DAC*, pages 530–535, Las Vegas, NV, USA, June 2001. ACM.

23 Rajeev Motwani, Jennifer Widom, Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Gurmeet Singh Manku, Chris Olston, Justin Rosenstein, and Rohit Varma. Query processing, approximation, and resource management in a data stream management system. In *CIDR*, 2003.

24 M. A. Hakim Newton, Duc Nghia Pham, Abdul Sattar, and Michael Maher. Kangaroo: An efficient constraint-based local search system using lazy propagation. In *17th International Conference on Principles and Practice of Constraint Programming*, pages 645–659, Perugia/Italy, September 2011.

25 Raghu Ramakrishnan. Magic templates: a spellbinding approach to logic programs. *Journal of Logic Programming*, 11(3-4):189–216, 1991.

26 Diptikalyan Saha. *Incremental Evaluation of Tabled Logic Programs*. PhD thesis, Stony Brook University, December 2006.

27 Terrance Swift and David Scott Warren. XSB: Extending Prolog with tabled logic programming. *CoRR*, abs/1012.5123, 2010. Under consideration for publication in Theory and Practice of Logic Programming.

28 Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume 1. Computer Science Press, 1988.

29 Alan G. Yoder and David L. Cohn. Domain-specific and general-purpose aspects of spreadsheet languages. In *Proceedings of the Workshop on Domain-Specific Languages*, 1997.

30 G. Zweig and M. Padmanabhan. Exact alpha-beta computation in logarithmic space with application to map word graph construction. In *Proceedings of ICSLP*, 2000.

Software Model Checking by Program Specialization

Emanuele De Angelis

University 'G. d'Annunzio' of Chieti-Pescara,
Viale Pindaro 42, I–65127 Pescara, Italy
Email: deangelis@sci.unich.it

Abstract

We introduce a general verification framework based on program specialization to prove properties of the runtime behaviour of imperative programs. Given a program P written in a programming language L and a property φ in a logic M, we can verify that φ holds for P by: (i) writing an interpreter I for L and a semantics S for M in a suitable metalanguage, (ii) specializing I and S with respect to P and φ, and (iii) analysing the specialized program by performing a further specialization. We have instantiated our framework to verify safety properties of a simple imperative language, called SIMP, extended with a nondeterministic choice operator. The method is fully automatic and it has been implemented using the MAP transformation system [14].

1998 ACM Subject Classification D.2.4 Software/Program Verification

Keywords and phrases Software model checking, program specialization, constraint logic programming.

Digital Object Identifier 10.4230/LIPIcs.ICLP.2012.439

1 Introduction and problem statement

Formal verification techniques allow us to prove that software artefacts (e.g., analysis and design models and source code) satisfy some given specifications. These techniques have recently received a growing attention as the basis for a promising methodology to increase the reliability and reducing the cost of software production (e.g., by reducing time to market).

Software model checking is a body of formal verification techniques for imperative programs that combine and extend ideas and techniques developed in the fields of static program analysis and model checking (see [12] for a recent survey). In order to prove that a program satisfies a given specification, software model checking methods automatically construct a program model which is sound, in the sense that if the model satisfies the given specification, then so does the actual program. Constructing such a model is a critical aspect of software model checking, since it tries to meet two somewhat conflicting requirements. On one hand, in order to make the verification process of large programs viable in practice, it has to construct a model by abstracting away as many details as possible. On the other hand, it would be desirable to have a model which is as precise as possible to reduce the number of wrong detections. Unfortunately, even for small programs operating over integer variables, an exhaustive exploration of the state space generated by the execution of programs is practically infeasible, and simple properties such as safety (which essentially states that 'something bad never happens') are undecidable. Despite this undecidability limitation, software model checking techniques work in many practical cases.

A huge amount of imperative languages is nowadays available. These languages provide sophisticated features which continuously change. Thus, software model checkers are required to be rapidly adapted to those changes. In order to develop tools which meet such a

© Emanuele De Angelis;
licensed under Creative Commons License ND
Technical Communications of the 28th International Conference on Logic Programming (ICLP'12).
Editors: A. Dovier and V. Santos Costa; pp. 439–444

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

requirement, agile development methodologies should be employed. In this paper we consider program specialization as a framework in which model checking of imperative programs may be performed in a very *agile*, effective way. Program specialization is a program transformation technique whose objective is the adaptation of a program to the context of use. In particular, it turns out to be a very flexible and general methodology through which variations of the semantics of the considered imperative language and to different logics for expressing the properties of interest may be rapidly implemented into the software model checker. Indeed, by following this approach, which can be regarded as a generalization of the one proposed in [16], given a program P written in a programming language L, and a property φ in a logic M, we can verify that φ holds for P by: (i) writing an interpreter I for L and a semantics S for M in a suitable metalanguage, (ii) specializing the interpreter and the semantics with respect to P and φ, and finally (iii) analysing the specialized program. We choose *Constraint Logic programming* (CLP), which has been shown to be a very suitable language for implementing symbolic evaluation and analysis of imperative programs [10, 11, 16, 17], as a metalanguage.

2 Background and overview of the existing literature

Constraint logic programming has been successfully applied to perform model checking of both finite state [15] and infinite state [4] systems. In [6] a framework for the verification of safety properties of infinite reactive systems based on CLP program specialization has been introduced. Moreover, in [7] it has been shown that the termination of reachability analyses of infinite state systems can be improved by encoding reachability as a CLP program and then specializing the CLP program by incorporating the information about the initial and the unsafe states. The use of CLP to analyse simple imperative programs has been proposed in [16], where a CLP-based interpreter for the operational semantics of a simple imperative language is partially evaluated w.r.t. an input program. The result of the analysis of the residual CLP program can thus be used for annotating the original imperative program with relations among variables occurring in the imperative program. In [8] a method is presented for translating imperative programs supporting heap-allocated mutable data structures and recursive procedures to CLP.

A widely used technique implemented by software model checkers (e.g. SLAM and BLAST) is the *Counter-Example Guided Abstraction Refinement* (CEGAR) [12] which, given a program P and a safety property φ, uses an abstract model $\alpha(P)$ to check whether or not P satisfies φ. If $\alpha(P)$ satisfies φ then P satisfies φ, otherwise a counterexample, i.e., an execution which makes the program unsafe, is produced. The counterexample is then analysed: if it turns out to be a real execution of P (*genuine* counterexample) then the program is proved to be unsafe, otherwise it has been generated due to a too coarse abstraction (*spurious* counterexample) and $\alpha(P)$ needs to be refined. The CEGAR approach has also been implemented by using CLP. In particular, in [17], the authors have designed a CEGAR-based software model checker for C programs, called ARMC. In [10], another CLP-based software model checker for C programs, called TRACER, is presented. It integrates an abstraction refinement phase within a symbolic execution process.

3 Goal of the research

The goal of our research is to introduce a software model checking *framework*, based on the specialization of CLP programs, which is *parametric* with respect to: (i) the imperative language of the programs to be verified, and (ii) the specification language of the property

to be checked. Regardless of the actual languages, the software model checker consists of a *front-end* module, which handles source code of programs, and a *verification engine* module which actually performs the verification task. Since our objective is performing software model checking of real programs we have to deal with issues arising from handling programs consisting of thousands of lines of code and using advanced features of imperative languages. Handling large programs introduces scalability issues which are to be carefully considered during the realization of the front-end. Indeed, the choices made during the design of the front-end may heavily affect the performance of the verification engine. More complex issues arise from handling programs using static and dynamic data structures, procedures, concurrency and objects. Consequently, a large portion of our research activity is devoted to designing abstraction techniques to be integrated in the CLP specialization process to prove properties which range from simple safety properties, such as array bound checking, to more sophisticated properties dealing with contents of data structures (e.g., sortedness), class relations (e.g., inheritance) and object interactions (e.g., effects of method invocations on object fields) and concurrent processes.

Our first mid-term objective is to realize a software model checker for the C language, which is still very popular, especially among device drivers and operating systems developers. A renewed attention to the C language is demonstrated in the TACAS 2012 competition (`http://sv-comp.sosy-lab.org/2012/`) in which several C software model checkers have been tested on very large programs preprocessed by using the CIL (C Intermediate Language) front-end (`http://cil.sourceforge.net/`). Thus, in order to ease the comparison with other tools we have decided to instantiate our framework to perform model checking of the C language by: (i) using CIL to translate the source language, and (ii) introducing a verification engine to prove safety properties of C programs.

4 Current status of the research

As a first step of our research activity we have instantiated our framework to perform the model checking of programs written in a simple imperative language with nondeterministic choice (SIMP), which is an abstraction of a subset of the C language. In particular, we have considered integer variables and the common control flow statements: `while(b) {···}` and `if(b) {···} else {···}`. The operational, or transition, semantics of SIMP is defined in terms of a transition relation $\Rightarrow$ over *state*s, that is, pairs of the form $\langle p, e\rangle$, where p is a command and e is an *environment*, i.e, a function which maps variables occurring in p to their values. A state s' is said to be *reachable* from s if $s \Rightarrow^* s'$. A *specification* S is a triple $\langle$*initial-prop*$, p,$ *unsafe-prop*$\rangle$, where p is a command and *initial-prop* and *unsafe-prop* are two formulas describing environments. A state $\langle p, e\rangle$ is said to be *initial* (*unsafe*) if e satisfies *initial-prop* (*unsafe-prop*). We say that S holds, or p is *safe* w.r.t. S, if there is no unsafe state which is reachable from an initial state. Performing model checking of p consists in verifying whether or not S holds. In order to perform model checking of SIMP commands we have defined a CLP-based interpreter of the operational semantics of SIMP as follows:

```
t( s(asgn(loc(X),A),E1), s(skip,E2) ) :- aeval(A,E1,V), update(loc(X),V,E1,E2).
t( s(ite(B,S1,_),E), s(S1,E) ) :- beval(B,E).
t( s(ite(B,_,S2),E), s(S2,E) ) :- beval(not(B),E).
t( s(ite(ndc,S1,_),E), s(S1,E) ).
t( s(ite(ndc,_,S2),E), s(S2,E) ).
t( s(while(B,S1),E), s(ite(B,comp(S1,while(B,S1)),skip),E) ).
t( s(comp(skip,S),E), s(S,E) ).
t( s(comp(S1,S3),E1), s(comp(S2,S3),E2) ) :- t( s(S1,E1), s(S2,E2) ).
```

where `t(s(P,E), s(P1,E1))` holds iff ⟨P,E⟩ ⇒ ⟨P1,E1⟩, that is, the execution of the command `P` in the environment `E` leads to the execution of the command `P1` in the environment `E1`. Terms of the form `s(P,E)` denote states, where `P` ranges over ground terms built out of the following functors: `asgn` for the assignment, `ite` for the if-then-else statement (`ndc` represents the nondeterministic choice operator), `while` for the while loop, `skip` for the empty statement, `comp` for the statement composition, and `E` is a list of terms encoding the environment. We also have that: (i) `aeval(A,E,V)` holds iff `V` is the value of the arithmetic expression `A` in the environment `E`, (ii) `beval(B,E)` holds iff the boolean expression `B` evaluates to **true** in the environment `E`, and (iii) `update(loc(X),V,E1,E2)` holds iff `E2` is equal to `E1` except in `X` which takes the value `V` (`loc` encodes a variable identifier).

Let $S = \langle \textit{initial-prop}, p, \textit{unsafe-prop} \rangle$ be a specification. We reduce the problem of verifying whether or not a command is safe to a reachability problem by using the CLP program SMC which consists of the following clauses:

```
unsafeProg :- initial(X), reachable(X).
reachable(X) :- unsafe(X).
reachable(X) :- t(X,X1), reachable(X1).
initial(s(p,E)) :- initial-prop.
unsafe(s(_,E)) :- unsafe-prop.
```

together with the clauses defining the semantics of SIMP. In the above program, `p` stands for the ground term encoding the command p, while `initial-prop` and `unsafe-prop` stand for constraints encoding the formulas *initial-prop* and *unsafe-prop*, respectively.

Our software model checking method consists in:

(Step 1) encoding the specification S into the clauses for `initial`, and `unsafe`,

(Step 2) specializing SMC with respect to `unsafeProg`, and

(Step 3) computing the least model M(SpSMC) of the specialized program SpSMC, and checking whether or not `unsafeProg` belongs to the least model M(SpSMC).

The objective of Step 2 is to modify the initial program SMC by propagating the information specified by `initial`, so that by exploiting this information, the computation of the least model M(SpSMC) may be more effective and terminate more often than the computation of the least model M(SMC). In particular, the interpretation overhead is compiled away by specialization, thereby producing a CLP program where no terms encoding the command p are present. Step 2 is performed by a rule-based program specialization strategy which makes use of the following rules: definition introduction, unfolding, folding, and clause removal. These rules preserve the least model semantics of CLP programs [5], thus, the specialization strategy yield a program which is guaranteed to satisfy the same set of properties satisfied by the original program. Indeed, we have that `unsafeProg`$\in M$(SMC) iff `unsafeProg`$\in M$(SpSMC). In order to ensure the termination of Step 2, we use suitable generalization operators, related to widen operators used in abstract interpretation techniques [2].

▶ **Example 1.** Let us consider the following specification $\langle x = 0 \wedge y \geq 0,\ p,\ error = 1 \rangle$, where p stands for: `while (x < 10) { y = y+1;  x = x+1; } if ( y + x < 10) { error = 1; }`. We encode p into the term

```
comp( while( lt(loc(x),int(10)), comp(asgn(loc(y),plus(loc(y),int(1))),
                                      asgn(loc(x),plus(loc(x),int(1))) ) ),
      ite( lt(plus(loc(y),loc(x)),int(10)), asgn(loc(error),int(1)), skip) )
```

where the functor `int` encodes an integer value. By also translating the remaining components of the specification we obtain the following clauses:

```
initial(s(p,[X,Y,E])) :- X=0, Y>=0, E=0.
unsafe(s(_,[X,Y,E])) :- E=1.
```

where p stands for the term encoding p. By specializing SMC with respect to unsafeProg, we obtain the following program

```
new3(X,Y,E) :- X>=0, X<10, Y>=X, E=0, X1=X+1, Y1=Y+1, new3(X1,Y1,E).
new3(X,Y,E) :- X>=10, Y>=X, E=0, new5(X,Y,E).
new2(X,Y,E) :- X=0, Y>=0, E=0, X1=1, Y1=Y+1, new3(X1,Y1,E).
unsafeProg  :- X=0, Y>=0, E=0, new2(X,Y,E).
```

whose model is used to verify whether or not p is safe. Since the program contains no constrained fact, we have that $M(\mathsf{SpSMC})$ is empty, and thus p is safe.

In [16], the interpreter is specialized w.r.t. the input program and a static analyser for CLP programs is used to derive relations among variables occurring in the imperative program. In our method, we discover these relations *during* the specialization process by means of generalization techniques defined in terms of relations and operators on constraints such as widening, convex-hull, and well-quasi orders.

5 Preliminary results accomplished

We have implemented our software model checking method using the MAP transformation system [14] for the verification engine module and the Lex and Yacc tools for the front-end module (it is currently being rewritten in CIL). We have shown the effectiveness of our method by applying it to some examples (available at http://map.uniroma2.it/smc) taken from the literature [9, 10], and we have compared its performance with that of ARMC and TRACER. Among the wide variety of software model checkers nowadays available, we choose ARMC and TRACER because they provide CLP based implementations of two dual verification approaches: ARMC starts with a very coarse abstraction and uses counterexamples to increase the level of details of the model and, conversely, TRACER starts with a very detailed model and uses counterexamples to abstract away as many details as possible and, possibly, refines it if the model is too coarse to prove the property. Our preliminary results (see Table 1) show that our approach is viable and competitive in practice.

Table 1 Time (in seconds) for performing model checking (TRACER was run by using the option `-intp wp`). $\perp$ denotes 'terminating with error', ∞ means 'No answer within 20 minutes'.

Tool	*f2*	*substring*	*daggerP*	*seesaw*	*tracerP*	*interp*	*widen*	*selectSort*
ARMC	∞	719.39	∞	3.98	∞	0.13	∞	0.48
TRACER	1.35	227.28	1.27	1.46	1.04	1.32	1.35	$\perp$
MAP	0.21	10.20	5.37	0.03	0.03	0.06	0.07	0.06

6 Open issues and expected achievements

A challenging issue is the extension of our framework to deal with more complex language features provided by imperative languages such as arrays, lists, procedure calls, and concurrency. Moreover, since it is possible to deal with imperative languages at different levels of abstraction, it would be interesting to extend the framework to verify properties of both: (i) high-level languages with object-oriented features, such as Java, PHP, Objective C or C# [13], and (ii) low-level languages such as bytecode for Java [1] or for the .NET platform [3]. From the verification point of view, such extensions would require the design of suitable interpreters for handling the newly introduced language features, posing new theoretical and experimental challenges.

Another challenging issue is the extension of the set of properties which can be proved. In Section 3 we listed some interesting properties depending on the content of data structures, the relations among objects and among classes, and the behavior of concurrent processes. Handling these properties not only requires the investigation of suitable logics in which they may be expressed, but also raises the issue of integrating them into the verification process. In particular, it could be necessary to resort to more sophisticated logic program transformations based on the unfold/fold method.

References

1. E. Albert, M. Gómez-Zamalloa, L. Hubert, and G. Puebla. Verification of Java bytecode using analysis and transformation of logic programs. In *Proc. of PADL'07*, volume 4354 of *LNCS*, pp. 124–139, 2007.
2. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixpoints. In *Proc. of POPL'77*, pp. 238–252. ACM Press, 1977.
3. P. Cousot, R. Cousot, and F. Logozzo. A parametric segmentation functor for fully automatic and scalable array content analysis. In *Proc. of POPL'11*, pp. 105–118, 2011.
4. G. Delzanno and A. Podelski. Model checking in CLP. In *Proc. of TACAS'99*, LNCS 1579, pp. 223–239, 1999.
5. S. Etalle and M. Gabbrielli. Transformations of CLP modules. *Theoretical Computer Science*, 166:101–146, 1996.
6. F. Fioravanti, A. Pettorossi, and M. Proietti. Verifying CTL Properties of Infinite State Systems by Specializing Constraint Logic Programs. In *Proc. of VCL'01*, pp. 85-96. Revised and extended Version: Technical Report R.657, IASI - CNR, 2007, Roma, Italy.
7. F. Fioravanti, A. Pettorossi, M. Proietti, and V. Senni. Improving Reachability Analysis of Infinite State Systems by Specialization. In *Proc. of RP'11*, pp. 165–179, 2011.
8. C. Flanagan. Automatic software model checking via constraint logic. *Sci. Comput. Program.*, 50(1-3):253–270, March 2004.
9. B. S. Gulavani, S. Chakraborty, A. V. Nori, and S. K. Rajamani. Automatically Refining Abstract Interpretations. In *Proc. of TACAS'08*, LNCS 4963, pp. 443–458, 2008.
10. J. Jaffar, J. A. Navas, and A. E. Santosa. TRACER: A symbolic execution tool for verification. `http://paella.d1.comp.nus.edu.sg/tracer/`.
11. J. Jaffar, J. A. Navas, and A. E. Santosa. Symbolic execution for verification. *Computing Research Repository*, 2011.
12. R. Jhala and R. Majumdar. Software model checking. *ACM Comput. Surv.*, 41(4):21:1–21:54, October 2009.
13. G.T. Leavens, K. R. M. Leino, and P. Müller. Specification and verification challenges for sequential object-oriented programs. *Form. Asp. Comput.*, 19(2):159–189, June 2007.
14. The MAP transformation system. `http://www.map.uniroma2.it/mapweb`.
15. U. Nilsson and J. Lübcke. Constraint logic programming for local and symbolic model-checking. In *Proc. of CL 2000*, LNAI 1861, pp. 384–398, 2000.
16. J. C. Peralta, J. P. Gallagher, and H. Saglam. Analysis of imperative programs through analysis of constraint logic programs. In *Proc. of SAS'98*, LNCS 1503, pp. 246–261,1998.
17. A. Podelski and A. Rybalchenko. ARMC: The logical choice for software model checking with abstraction refinement. In *Proc. of PADL'07*, LNCS 4354, pp. 245–259, 2007.

Temporal Answer Set Programming *

Martín Diéguez

University of Corunna
Corunna, Spain
`martin.dieguez@udc.es`

Abstract

Answer Set Programming (ASP) has become a popular way for representing different kinds of scenarios from knowledge representation in Artificial Intelligence. Frequently, these scenarios involve a temporal component which must be considered. In ASP, time is usually represented as a variable whose values are defined by an extensional predicate with a finite domain. Dealing with a finite temporal interval has some disadvantages. First, checking the existence of a plan is not possible and second, it also makes difficult to decide whether two programs are *strongly equivalent*.

If we extend the syntax of Answer Set Programming by using temporal operators from temporal modal logics, then infinite time can be considered, so the aforementioned disadvantages can be overcome. This extension constitutes, in fact, a formalism called *Temporal Equilibrium Logic*, which is based on *Equilibrium Logic* (a logical characterisation of ASP).

Although recent contributions have shown promising results, Temporal Equilibrium Logic is still a novel paradigm and there are many gaps to fill. Our goal is to keep developing this paradigm, filling those gaps and turning it into a suitable framework for temporal reasoning.

1998 ACM Subject Classification F.4.1 Mathematical Logic/Temporal logic, I.2.3 Deduction and Theorem Proving/Logic programming

Keywords and phrases Answer Set Programming, Temporal Equilibrium Logic, Linear Temporal Logic

Digital Object Identifier 10.4230/LIPIcs.ICLP.2012.445

1 Introduction and motivation

Modal Temporal Logics have become a popular paradigm used in protocol specification and representation of temporal scenarios. There are many techniques and tools that allow checking some desirable properties over temporal systems defined by this kind of logics, but most of them usually involve representational issues such as the limitation to propositional definition of the scenarios and several inherent problems like the frame problem [24].

Another paradigm used for temporal reasoning is *Answer Set Programming* [25, 23], whose roots are based on Logic Programming and Non-monotonic Reasoning. Thanks to the use of variables in the representation, reasoning with incomplete information and the existence of high performance solvers to compute the models (solutions) of the problem, ASP has become a suitable paradigm for representing and solving search problems in Artificial Intelligence. ASP has been commonly used to represent temporal scenarios [19] due to both the use of variables in the specifications and the easy definition of inertia rules which avoid the frame problem inherent to the formalisation of temporal problems. In fact, there are several action languages that use an ASP tool as a backend [14]. Despite all these

* This research was partially supported by Spanish MEC project TIN2009-14562-C05-04.

© Martín Diéguez;
licensed under Creative Commons License ND

features, its main disadvantage is that time is represented by an extensional predicate with a finite domain, and consequently, a desired property like checking the existence of a plan for a particular representation is not possible. Furthermore, it makes checking whether two temporal theories are strongly equivalent more difficult (or impossible if arbitrary temporal formulas are allowed). In order to motivate our research, let us consider the following example:

▶ **Example 1.** A wolf, a goat and a cabbage are on one bank of a river and the boatman has to take them, safe and sound to the other side. The boatman can take at most one of them at a time. Finally, the following condition is applied: the wolf will eat the goat, and the goat will eat the cabbage, if they are left alone at the same side of the river.

```
1. time(0..max). opp(l,r). opp(r,l).
2. item(w). item(g). item(c).
3. eats(w,g). eats(g,c). object(b).
4. object(Z) :- item(Z).
5. next(I,I1) :- I1 = I+1, time(I1), time(I).

6. % Effect axiom for moving
7. at(X,A,T2):- at(X,B,T1), m(X,T1),
8.            opp(A,B), next(T1,T2).

9. % The boat is always moving
10. at(b,A,T2):- at(b,B,T1), opp(A,B),
11.                        next(T1,T2).

12. % Inertia
13. at(Y,A,T2) :- at(Y,A,T1), not at(Y,B,T2),
14.                     opp(A,B), next(T1,T2).

15. % State constraint
16. :- eats(X,Y), at(X,A,T), at(Y,A,T),
17.                at(b,B,T), opp(A,B).

18. % Unique value constraint
19. :- at(Y,A,T), at(Y,B,T), opp(A,B).

20. % Choice rules for action execution
21. m(X,T) :- not a(X,T), time(T),
22.                 item(X), T < max.
23. a(X,T) :- not m(X,T), time(T),
24.                 item(X), T < max.

25. % Action executability
26. :- m(X,T), at(b,A,T), at(X,B,T),
27.                         opp(A,B).

28. % Non-concurrent actions
29. :- m(X,T), m(Z,T), X != Z.

30. % Initial state: everything at
31. % left bank
32. at(Y,l,0):- object(Y).

33. % Goal: all items at right bank
34. g :- at(w,r,T), at(g,r,T), at(c,r,T).
35. :- not g.
```

As shown above, this representation encodes time in a variable `I` which is appended to every temporal predicate. This variable takes its values from {0,1,2,...,`max`}, being `max` a constant value which fixes the extension of the predicate `time/1`. Furthermore an example of an inertia rule is shown in lines 13 and 14. In an LTL formalisation, the information of this rule should be encoded in every fluent, considering every possible indirect effect.

Our proposal extends the syntax of ASP with operators to talk about time, like those defined in *Linear Temporal Logic* (LTL) [22], whose semantics is shown in Definition 2. To achieve this, we will make use of *Temporal Equilibrium Logic* (TEL) [10], which combines *Equilibrium Logic* [26] (a logical characterization of Answer Set Programming) and the aforementioned Linear Temporal Logic. This formalism introduces the definition of *Temporal Stable Models* (analogous to stable models [15] for any temporal theory). In TEL, every reference to time is encoded through modal operators. Therefore it is possible to consider an infinite-length narrative, overcoming the disadvantages of the current ASP approaches.

▶ **Definition 2. LTL semantics**
Let $\mathcal{I} = \{s_0, s_1, ..., s_n\}$ a set of set of atoms, i an integer ($i \geq 0$) and α an LTL formula. We say that $\mathcal{I}, i \models \alpha$ if:

- $\mathcal{I}, i \models p$ if $p \in s_i$.
- $\mathcal{I}, i \models \neg\alpha$ if $\mathcal{I}, i \not\models \alpha$.
- $\mathcal{I}, i \models \alpha \wedge \beta$ if $\mathcal{I}, i \models \alpha$ and $\mathcal{I}, i \models \beta$.
- $\mathcal{I}, i \models \alpha \vee \beta$ if $\mathcal{I}, i \models \alpha$ or $\mathcal{I}, i \models \beta$.
- $\mathcal{I}, i \models \Box\alpha$ if $\forall j \geq i, \mathcal{I}, j \models \alpha$.
- $\mathcal{I}, i \models \Diamond\alpha$ if $\exists j \geq i, \mathcal{I}, j \models \alpha$.
- $\mathcal{I}, i \models \bigcirc\alpha$ if $\mathcal{I}, i+1 \models \alpha$.
- $\mathcal{I}, i \models \alpha\mathcal{U}\beta$ if $\exists n \geq i, \mathcal{I}, n \models \beta$ and $\forall j, i \leq j < n, \mathcal{I}, j \models \alpha$.
- $\mathcal{I}, i \models \alpha\mathcal{R}\beta \iff \mathcal{I}, i \models \neg(\neg\alpha\mathcal{U}\neg\beta)$.

2 Goal of research and preliminary results accomplished

As we have mentioned before, Temporal Equilibrium Logic is a novel research topic and of course, maturity of the results obtained so far is not comparable to the state of the art in Answer Set Programming. The foundations of Temporal Equilibrium Logic are defined in [10]. The property of strong equivalence among temporal logic programs has been studied in [3] where the authors define a translation that allows us to check this property in Linear Temporal Logic using an LTL model checker. The normal form of TEL programs is defined in [7]. Every TEL theory can be translated, like in the non-temporal case, into a set of implications, but in the case of TEL normal form, some of those can be under the scope of the LTL operator "$\Box$" (read "forever").

Recent results have focused on computing temporal equilibrium models. A first contribution [4] studies the computation of temporal equilibrium models for a subset of the already mentioned TEL normal form, which receives the name of *Splitable Temporal Logic Programs* (STLP's). These programs are characterised by the informal condition saying that the future does not depend on the past, that is, if the LTL operator "$\bigcirc$" (read "next state") appears in the body of a rule, it also applies to every atom in its head. By the splitting theorem [20], the authors have proven that the technique of *Loop Formulas* [13] can be applied to this kind of programs, so, its temporal equilibrium models can be computed by an LTL model checker. Such models are represented in terms of a Büchi automaton [6], which captures the whole behaviour of the system.

`STeLP` [1] [9] is the first tool designed to compute Temporal Equilibrium Models. It has its roots in the aforementioned work but it also adds some features to the input language like the use of variables in the specifications (like most ASP tools) and the use of arbitrary LTL expressions in the constraints. Example 1 would be formalised in `STeLP` as follows:

```
1. domain item(X).                                          % Domain declaration
2. static eats/2,object/1, opp/2.                           % Static predicates
3. fluent at/2.                                             % Fluent declaration
4. action m/1.                                              % Action declaration
5. opp(l,r). opp(r,l). eats(w,g). eats(g,c).
6. item(w). item(g). item(c). object(b).
7. object(Z) :- item(Z).
8. o at(X,A)::- at(X,B), m(X), opp(A,B).                    % Effect axiom
9. o at(b,A)::- at(b,B), opp(A,B).                          % Effect axiom
10. ::- m(X), at(b,A), at(X,B), opp(A,B).                   % Action executability
11. ::- at(X,A), at(X,B), opp(A,B).                         % Unique value
12. o at(X,A) ::- at(X,A), not o at(X,B), opp(A,B).         % Inertia
13. ::- eats(X,Y), at(X,A), at(Y,A), at(b,B), opp(A,B).     % State constraint
14. m(X) ::- not a(X).                                      % Choice action
15. a(X) ::- not m(X).                                      % execution
```

[1] `http://kr.irlab.org/stelp_online`

```
16. ::- m(X), item(Z), m(Z), X != Z.                    % Non-concurrent actions
17. at(Y,l):- object(Y).                                % Initial state
18. g ::- at(w,r), at(g,r), at(c,r).                    % Goal state
19. :- always not g.                                    % Goal must be satisfied
```

The input language of this tool is very similar to common ASP language but introduces two new operators in the definition of the rules: "o" which corresponds to "$\bigcirc$" in LTL and "::-" which replaces the traditional ":-" of ASP but under the implicit scope of the LTL operator "$\Box$" , that is, the rule must be satisfied in every instant of time. Furthermore, `STeLP` deals with arbitrary LTL formulas in the body of constraints (as shown in line 30, in the example above), which is useful, for instance, to check whether a temporal representation has a plan.

Like most ASP solvers, `STeLP` grounds the input program before computing the models. In this case, traditional grounding algorithms are not directly applicable because ground atoms may change their truth value along time. As a first approach to temporal grounding, the user has to identify a family of predicates, called `static`, whose truth value remains constant along time. Furthermore, to guarantee the domain independence property of the program[2], the input program must satisfy the following *safety* condition:

1. Any variable X occurring in a rule $B \to H$ or $\Box(B \to H)$, also occurs in some positive static predicate in B.
2. Rules of the form $B \to H$, where at least one static predicate occurs in H, only contain static predicates.

It is easy to see that, under this condition, static predicates cannot depend on the non-static ones. `STeLP` first computes a stable model of the "static program" and then uses it to ground the rest of the rules by instantiating any positive static predicate in their bodies.

Since static predicates must occur in any rule, this tool allows defining global variable names with a fixed domain (by the keyword *domain*), in a similar way to lparse[3]. For instance, in the example above `item(X)` means that any rule referring to variable X is implicitly extended by including the implicitly declared static predicate `item(X)`.

Apart from splitable programs, arbitrary temporal theories have been studied in [8]. Here, the authors present an algorithm for computing temporal equilibrium models based on several operations over Büchi automata. Furthermore, that paper achieves first results about the complexity of TEL, whose satisfiability is decidable in ExpSpace and, at least, PSpace-hard.

3 Current status of research and expected achievements

As we have said before, several results have been accomplished but there is still much work in progress. We are currently trying to improve the grounding algorithm implemented in `STeLP`. Until now, only static predicates had been considered during the instantiation of a rule. This causes a generation of irrelevant ground rules that increase the size of the resulting ground LTL theory while they could be easily detected and removed by a simple analysis of the temporal program. A new algorithm for temporal grounding has been considered

[2] that is, its set of stable models does not change with respect to the addition of constants
[3] `http://www.tcs.hut.fi/Software/smodels/src/lparse-1.1.2.tar.gz`

in [2]. In this paper, the authors have proven that the safety condition of DLV [17] (every variable which occurs in a rule, must occur in a positive predicate in its body) is enough to guarantee the property of domain independence for STLP's. Considering an STLP under this safety condition, authors also define a translation to compute its set of *derivable facts* (a set of ground atoms that are potentially members of at least one of its temporal equilibrium models) which is used to determine, in grounding time, if a ground rule can be removed from the final ground program. If a ground rule contains an atom in its positive body which does not belong to the set of derivable facts of the program, this rule can be omitted is not generated.

As a future work, on the one hand, we expect to keep developing the foundations of Temporal Equilibrium Logic. In [16], it has been shown that LTL has the same expressive power as a First Order theory of linear ordering. We conjecture that there is a corresponding relation between Temporal Equilibrium Logic and *Quantified Equilibrium Logic*.

On the other hand, we are also concerned about practical results. For instance, one of our constant goals is improving the performance of `STeLP` and its scalability for larger programs. The main bottlenecks are the grounding algorithm (it has already been improved) and the computation of the temporal equilibrium models. The latter is made translating a final LTL formula into a Büchi automaton (which is the costliest process). In some cases, if we just want to check the satisfiability of a formula, there are tools like TSPASS [21] that are based on resolution techniques and can check satisfiability without computing the whole automaton. Depending on users' requirements, these tools are more suitable than SPOT [11], the current backend used by `STeLP`, due to their shorter response time.

In order to find practical applications for our work, one possibility is using STeLP as a backend for ASP-based action languages such as $\texttt{DLV}^k$ [12] or `ALM` [14]. In the current implementation of ASP-based action languages, checking whether a temporal or planning problem has a solution would mean exhausting the set of possible transitions (assuming we deal with a finite set of fluents and actions) and checking the appearance of repeated states outside the ASP program. STeLP offers the possibility of a fully automated method that builds the automaton from the generated temporal formulas. It could be plugged as an alternative backend for action languages when we are interested in verifying complex temporal properties or checking the existence of a plan. In fact our plan is to adapt the implementation of the action language `ALM` to use `STeLP` instead of its current backend.

Finally, like in every Ph.D. thesis, it is very important to make a comparison of our proposal with other frameworks for nonmonotonic temporal reasoning present in the literature, for instance [1], [18] and [5]. Finding a connection between Temporal Equilibrium Logic and those approaches would be useful to perceive the advantages and disadvantages of TEL with respect to other strategies.

References

1. M. Abadi and Z. Manna. Temporal Logic Programming. *Journal of Symbolic Computation*, 8(3):277–295, 1989.
2. F. Aguado, P. Cabalar, M. Diéguez, G. Pérez, and C. Vidal. Paving the Way for Temporal Grounding. In *ICLP '12*, 2012.
3. F. Aguado, P. Cabalar, G. Pérez, and C. Vidal. Strongly Equivalent Temporal Logic Programs. In *JELIA'08*, volume 5293 of *LNCS*, pages 8–20. Springer, 2008.
4. F. Aguado, P. Cabalar, G. Pérez, and C. Vidal. Loop Formulas for Splitable Temporal Logic Programs. In *LPNMR'11*, volume 6645 of *LNCS*, pages 80–92. Springer, 2011.
5. C. Baral and J. Zhao. Non-monotonic Temporal Logics for Goal Specification. *IJCAI 2007*.

6 R. Büchi. On a decision method in restricted second-order arithmetic. In *International Congress on Logic, Method and Philosophical Science'60*, pages 1–11, 1962.
7 P. Cabalar. A Normal Form for Linear Temporal Equilibrium Logic. In *JELIA'10*, volume 6341 of *LNCS*, pages 64–76. Springer, 2010.
8 P. Cabalar and S. Demri. Automata-based computation of temporal equilibrium models. In *21st International Workshop on Logic Program Synthesis and Transformation (LOPSTR'11)*, LNCS. Springer, 2011.
9 P. Cabalar and M. Diéguez. STeLP - a Tool for Temporal Answer Set Programming. In *LPNMR'11*, volume 6645 of *LNCS*, pages 370–375. Springer, 2011.
10 P. Cabalar and G. P. Vega. Temporal Equilibrium Logic: a first approach. In *11th International Conference on Computer Aided Systems Theory (EUROCAST'07)*, volume 4739 of *LNCS*, pages 241–248. Springer, 2007.
11 A. Duret-Lutz and D. Poitrenaud. SPOT: an Extensible Model Checking Library Using Transition-based Generalized Büchi Automata. In *Proc. of the IEEE Computer Society's 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS'04)*. IEEE Computer Society, 2004.
12 T. Eiter, W. Faber, N. Leone, G. Pfeifer, and A. Polleres. Planning under Incomplete Knowledge. In *1st International Conference on Computational Logic (CL '00)*. Springer, 2000.
13 P. Ferraris, J. Lee, and V. Lifschitz. A generalization of the Lin-Zhao theorem. *Annals of Mathematics and Artificial Intelligence*, 47(1-2):79–101, 2006.
14 M. Gelfond and D. Inclezan. Yet Another Modular Action Language. In *2nd International Workshop on Software Engineering for Answer Set Programming (SEA'09)*, pages 64–78, 2009.
15 M. Gelfond and V. Lifschitz. The Stable Model Semantics for Logic Programming. In *ICLP'88*, pages 1070–1080. MIT Press, Cambridge, MA, 1988.
16 J. Kamp. *Tense Logic and the Theory of Linear Order*. PhD thesis, University of California at Los Angeles, 1968.
17 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*, 7:499–562, 2006.
18 H. J. Levesque, F. Pirri, and R. Reiter. Foundations for the situation calculus. *Electronic Transactions on Artificial Intelligence*, 2:159–178, 1998.
19 V. Lifschitz. Answer Set Programming and Plan Generation. *Artificial Intelligence*, 138(1-2):39–54, 2002.
20 V. Lifschitz and H. Turner. Splitting a Logic Program. In *ICLP'94*, pages 23–37. MIT press, 1994.
21 M. Ludwig and U. Hustadt. Implementing a fair monodic temporal logic prover. *AI Communications*, 23(2-3):69–96, 2010.
22 Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer, 1992.
23 V. Marek and M. Truszczyński. *Stable models and an alternative logic programming paradigm*, pages 169–181. Springer-Verlag, 1999.
24 J. McCarthy and P. Hayes. Some Philosophical Problems from the Standpoint of Artificial Intelligence. *Machine Intelligence Journal*, 4:463–512, 1969.
25 I. Niemelä. Logic Programs with Stable Model Semantics as a Constraint Programming Paradigm. *Annals of Mathematics and Artificial Intelligence*, 25:241–273, 1999.
26 D. Pearce. A new logical characterisation of stable models and answer sets. In *Selected papers from the Non-Monotonic Extensions of Logic Programming (NMELP'96)*, volume 1216 of *LNAI*, pages 57–70. Springer, 1996.

A Gradual Polymorphic Type System with Subtyping for Prolog

Spyros Hadjichristodoulou

Computer Science Department, Stony Brook University
New York, U.S.A.

Abstract

Although Prolog was designed and developed as an untyped language, there have been numerous attempts at proposing type systems suitable for it. The goal of research in this area has been to make Prolog programming easier and less error-prone not only for novice users, but for the experienced programmer as well. Despite the fact that many of the proposed systems have deep theoretical foundations that add types to Prolog, most Prolog vendors are still unwilling to include any of them in their compiler's releases. Hence standard Prolog remains an untyped language. Our work can be understood as a step towards typed Prolog. We propose an extension to one of the most extensively studied type systems proposed for Prolog, the Mycroft-O'Keefe type system, and present an implementation in XSB-Prolog. The resulting type system can be characterized as a Gradual type system, where the user begins with a completely untyped version of his program, and incrementally obtains information about the possible types of the predicates he defines from the system itself, until type signatures are found for all the predicates in the source code.

1998 ACM Subject Classification D.1.6. Logic Programming, D.3.3. Language Constructs and Features

Keywords and phrases Type Inference, Polymorphic Type System, Gradual Typing, Tabling, Answer Subsumption

Digital Object Identifier 10.4230/LIPIcs.ICLP.2012.451

1 Introduction and problem description

Since the seminal work of Mycroft and O'Keefe [5] in introducing a polymorphic type system for Prolog, there has been vast research on the area. Some of it followed their direction, and is essentially about extending or reconstructing the Mycroft-O'Keefe type system [7, 2], while others take different paths for introducing types in logic programs [6, 1, 3].

The common denominator in these approaches, however, is that most of them are about *theoretically* defining and constructing a type system for Logic Programming. Although some early Prolog implementations contained type checking mechanisms based on the Mycroft-O'Keefe type system (e.g. the DEC-10 compiler), most modern systems tend to keep Prolog as an untyped language [1]. Of course, there is the exception of Mercury (`http://www.mercury.csse.unimelb.edu.au/`), which has become famous for the type-checking abilities it offers to programmers; however, this comes with the price of strong typing and "limited" polymorphism of Mercury programs.

As discussed in [5], the main purpose of developing a polymorphic type system for Prolog is to provide the programmer with another tool which will make programming easier and

[1] As we will discuss later, an implementation for the Mycroft-O'Keefe type system was developed with the intention of being distributed with SWI-Prolog and YAP

© Spyros Hadjichristodoulou;
licensed under Creative Commons License ND
Technical Communications of the 28th International Conference on Logic Programming (ICLP'12).
Editors: A. Dovier and V. Santos Costa; pp. 451–457

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

less error-prone. We consider our research as a step towards that direction; our goal is to build a working type system which will enable programmers to write correct Prolog programs more easily than before. The challenge is to somewhat combine various aspects of the approaches introduced in the literature and use modern techniques to implement a robust type inference system which will be distributed with XSB Prolog. Our type checking and inference mechanism will offer users two modes of operation; firstly, they will be able to type-check their program, if they provide a type signature for every predicate they define. Secondly, if some of these signatures are missing, the type inference engine will be able to infer types for the respective predicates, in order to provide the user with information about their newly defined predicates. This process will be conducted in an incremental, *gradual* manner; the user will start with a completely untyped version of his program, and type inference will gradually give more information about what the types of the defined predicates may be. If the user is satisfied with the type inference engine's suggestions, then these types will be considered by the system as if they were declared by the user as type signatures. This process will continue until each defined predicate in the source code has a type signature. This kind of type systems, called *Gradual Type Systems* was introduced in [8].

2 Background and overview of the existing literature

The first work introducing some kind of type checking and inference in Prolog was Mycroft and O'Keefe's Polymorphic type system, [5]. It is based on the seminal work by Robin Milner, [4], who created a polymorphic type system for the ML family of functional programming languages, and first introduced the notion of "Well-typedness". In the Mycroft-O'Keefe type system, type signatures are provided by the user for each defined predicate, and the type checker's task is to verify that each definition respects the signature declaration. The only notion of *inference* in this type system is with **Variables**; when a predicate `p(X,Y)` is type-checked against its signature, a type is inferred for both `X` and `Y`. Also, it allows for *polymorphism* in the usual meaning; arguments of predicates can be of *any* type, denoted by type variables, and allows for user-defined type constructors as well as some predefined ones (i.e. for lists, `type list(A) --> [] ; [A|list(A)]`). Moreover, a connection is made between the Prolog program to be type-checked and the type checker, which is itself another Prolog meta-program. Finally, the notion of "Well-typedness" is introduced, and has the same meaning as in the Hindley-Milner type system: "Well-typed programs can't go wrong". In the context of Prolog, this means that no predicate will be called with arguments that don't respect the type signature declared by the user. The following example illustrates the operations described above:

▶ **Example 1** (`append/3`)**.** The user declares a type for the well-known `append/3` predicate, as `:- pred append(list(A),list(A),list(A))`. This means that `append/3` has 3 arguments, and each has the same type, namely `list(A)`. So, each argument can be a list of anything, as long as all 3 arguments have the same type.

The user may have the definition of `append/3` available:

```
append([],L,L).
append([X|L1],L2,[X|L3]) :- append(L1,L2,L3).
```

Each clause of `append/3` will now be type-checked against the given signature. It's easy to see that each clause respects the declared signature, and so `append/3` is well-typed. Finally, if the following clause appears in the program, `p(X,Y,Z) :- ... , append(X,Y,Z), ...`, then types for `X`, `Y` and `Z` are inferred according to `append/3`'s signature (and all are `list(A)`).

A recent implementation of the Mycroft-O'Keefe type system was developed in 2009 [7]. The authors' aim was to gradually[2] introduce types in Prolog using a type-checking library that was planned to be shipped with two of the most popular Prolog implementations, SWI-Prolog and YAP. This type checking library makes it easy to interface typed and untyped code, by performing runtime checks when typed predicates (i.e. predicates for which the user has provided a type signature) are called. By doing that, the authors make sure that the type system can be used somewhat "on-demand", i.e. only when the programmer gives type signatures for certain predicates, and this makes the migration from untyped to typed Prolog easier.

A rather different approach was introduced in [1]. The authors employ a fixed-point, bottom-up, abstract interpretation technique in order to infer types for Horn-Clause programs. Type declarations for predicates and constructors are very similar to the Mycroft-O'Keefe type system [5], but in this approach, the system can infer types for predicates while not depending on a type signature provided by the user. For example, the type of `append/3` can be inferred only by its definition and the definition of the `list(A)` constructor, as given above.

3 Goal of the research

Our goal is to combine the advantages of each of these approaches in order to build a robust inference system for types of both predicates and constructors. The system will be distributed along with XSB-Prolog, and is implemented as a preprocessor of the original source code. This allows the user to start with a completely untyped Prolog program, and with the help of the inference engine to *gradually* learn more about the types of the predicates he defines. We believe that the existence of such a system will make programming in Prolog easier and less error-prone, while at the same time maintaining the flexibility that Prolog gives.

4 Current status of the research

Our first task was to port the type-checking library from [7] which was written for SWI-Prolog and YAP, to work in XSB-Prolog. Based on the XPP-Preprocessor [3], we implemented a preprocessor that gets invoked by the user with a compiler flag. If the user desires to provide type signatures for any of the predicates he has defined, the only thing to do is include the following declaration in his source code: `:- compiler_options(xpp_on(typecheck)).`

After successfuly porting the type-checking library to XSB-Prolog, we used the approach in [1] to build a type-inference engine. We use the same notations for declaring the types as in [5]. Defined predicates for which a type signature has been provided get type-checked, whereas the type of the others is inferred. Despite the similarities between our approach and the ones discussed previously, there are basic differences:

- We extended the Mycroft-O'Keefe type system in order to lift the limitation that each clause of a predicate must have the same type. Prolog programmers often want to define facts which may have different types, and we didn't want our engine to infer that this

[2] The use of the term "gradually" here has a different meaning that the one we used to describe our approach in the previous section. In [7] it is used to describe the *migration* process from untyped Prolog to typed Prolog, whereas in our approach, it is used to describe the process of adding types to a single program, starting from a completely untyped program and moving towards a fully typed program

[3] `http://www.cross-browser.com/x/docs/xpp_reference.php`

kind of predicates is ill-typed. If, for example, the user has defined two facts as `p(42)` and `p(a)`, the type-inference engine will give `p/1` the type `p(atomic)`, instead of failing. For this scheme to work, we have introduced simple-fixed subtyping rules between the primitive types that each program can have. For example, `integer`, `atom` and `float` are all subtypes of `atomic`.

- In [1], the authors use a "cut-off" point to stop their inference when the type of a predicate grows bigger at each step. Instead of doing this, we are using `unify_with_occurs_check`, so that when two clauses of a predicate give types that can't be unified with the occurs check, our system infers that the predicate is ill-typed. This approach was also taken in type-checking in [7].
- None of the previous approaches were able to handle the case of inferring types for type constructors. We are currently developing a type inference mechanism which will be invoked when the user requests, which will try to infer what the type of a defined constructor may be. This may be particularly useful when using large libraries with many new constructors but no documentation on what each constructor does. We hope that being able to see the type of each constructor will be useful to the programmer for better understanding external code.

Algorithm 1 Outer fixed-point

1: `do_type_inference_batch(PredList,TypedListIn,TypedList) :-` {Let `PredList` be the list of predicates we want to infer types for, `TypedListIn` be the list of types for the predicates in PredList, `TypedList` be the list of types that will be inferred in one step}
2: **for all** `Pred` in `PredList` **do**
3:   `type_inference_batch(Pred,Type,TypedListIn)`
4: **end for**
5: Let `TypeListTemp` be the list consisting of all the returned `Types`
6: **if** `TypeListIn` != `TypeListTemp` **then**
7:   call `do_type_inference(PredList,TypeListTemp,TypedList)`
8: **else**
9:   set `TypedList` = `TypeListIn`
10: **end if**

Algorithm 2 Inner fixed-point

1: `type_inference_batch(Pred,Type,TypedList) :-` {Let `Pred` be the predicate we want to infer types for, `Type` be the type we will infer for `Pred`, `TypedList` be the list of types that were inferred in the previous step, `TypeIn` be the type of `Pred` computed in the previous step as it resides in `TypedList`}
2: **for all** clauses of `Pred` **do**
3:   call `type_inference(Pred,TypedList,TypeIn)`
4: **end for**
5: Gather all newly constructed `TypeIns` in a list, `TList`
6: Unify all elements of `TList` wich each other
7: Unify `Type` with the Head of `TList`

We currently have 3 versions of the type-inference engine. In the first, which is described in algorithms 1, 2 and 3 below, the predicates that perform type-inference are all non-tabled. The problem with this approach, is that we needed to pass around a list of all the

Algorithm 3 Find a type from only one clause of the predicate

1: `type_inference(Pred,TypedList,Type) :-` {Let `Pred` be the predicate we want to infer types for, `Type` be the type we will infer for `Pred`, `TypedList` be the list of types that will be inferred in one step}
2: **for** each clause of `Pred` **do**
3: Find a type for `Pred` from the body of the clause and store it in `Type`
4: Find a type for `Pred` from the head of the clause and store it in `PredType`
5: **if** `Type` and `PredType` can be unified with occurs check **then**
6: succeed
7: **else**
8: throw error
9: **end if**
10: **end for**

(intermediate) types that had been inferred for each predicate of the source code, in order to use the newest information at each step.

In order to remedy this, we re-wrote the basic type-inference predicates into a tabled version. Each time a new type is inferred for a predicate, a record is entered in the global table kept by XSB, so when the type of any predicate is requested during the process, it can be obtained by looking-up the table, instead of passing around a list. This approach also enabled us to be able to give some type to mutually recursive predicates.

For the final version of our type inference engine, we employed the principle of `answer subsumption` as described in [9]. In essence, whenever a new answer is produced for a predicate that is tabled with answer subsumption, it's joined with the answer that already resides in the table, and that join is now the only answer for that predicate. Now, instead of using `findall/3` in lines 5-7 and 6-8 of algorithms 1 and 2 respectively to get all the types for each clause and until the fixed point is reached, we use `answer subsumption` and always keep the most *specific* type found for each predicate. This may seem rather illogical in the beginning, since when two answers are produced for a predicate, we tend to keep the most general one. The reason is that when trying to find answers for goals in Prolog, we don't care which clause of the goal will make the answer found true, as long as there is one that does. However, in type inference, the type inferred must respect *all* the clauses of the predicate. We can think of this difference as the duality between *union* and *intersection*; when we want answers for a goal we are looking for the *union* of answers, whereas when we want to find a type for a predicate, we want the *intersection* of types found.

5 Preliminary results accomplished

▶ **Example 2.** We will start with a simple recursive predicate, `reverse_acc/3`. It's the tail-recursive version of `reverse/2`, which binds the output with the input list, reversed.

```
reverse_acc([], Acc, Acc).
reverse_acc([Head|Tail], Acc, Reversed) :-
      reverse_acc(Tail, [Head|Acc], Reversed).
```

Asking our engine to infer the type of `reverse_acc/3` will yield:

```
| ?- infer_types('test.P').

Inferred types for the following 1 predicates:
reverse_acc(list(A),list(A),list(A))
```

▶ **Example 3.** In this second example, we will show how our extensions to the Mycroft-O'Keefe type system behave. The predicate we want to infer a type for is `foo/1`:

```
foo(42).
foo(bar).
```

The original Mycroft-O'Keefe type system would not be able to give a type to `foo/1`. Our extensions make it possible for the engine to assign the `atomic` type:

```
| ?- infer_types('test.P').

Inferred types for the following 1 predicates:
foo(atomic)
```

▶ **Example 4.** For this last example, we will show some preliminary results of our constructor type inference engine. We assume the following code snippet:

```
:- type natural ---> 0 ; s(natural).

formula(0).
formula(s(0)).
formula(s(s(0))).
formula(0 + s(0)).
formula(s(s(0)) - s(0)).
```

The above code declares a new type constructor `natural` for natural numbers, and various versions of the same predicate, `formula/1`. The task is to find what is the type of the constructors of `formula/1`'s argument:

```
| ?- infer_constructors('test.P',formula(_)).

Inferred the following constructors:
formula(natural)
formula(+(natural,natural))
formula(-(natural,natural))
```

The engine has managed to infer that the argument of `formula/1` can be either a `natural` (as per the type constructor above), a `+(natural,natural)` or a `-(natural,natural)`.

6 Open issues and expected achievements

Although the implementation of the type inference engine has progressed over the few last months, there are still issues that need to be resolved

- The final version of the code where answer subsumption is used must be tested thoroughly and compared to the other versions. It will be interesting to see the differences in both runtime and table usage for large source files
- The type inference for constructors must be refined; the correct type constructors for the last example of the previous section would be `expr --> natural ; expr + expr ; expr - expr`, so our engine must somehow recognize that the `+` and `-` combine more complex things than simple `naturals`

References

1. R. Barbuti and R. Giacobazzi. A bottom-up polymorphic type inference in logic programming. *Sci. Comput. Program.*, 19(3):281–313, 1992.
2. T.K. Lakshman and U.S. Reddy. Typed Prolog: A semantic reconstruction of the Mycroft-O'Keefe type system. In *Int. Logic Programming Symp*, pages 202–217, 1991.
3. L. Lu. Polymorphic type analysis in logic programs by abstract interpretation. *The Journal of Logic Programming*, 36(1):1–54, 1998.
4. R. Milner. A theory of type polymorphism in programming. *Journal of computer and system sciences*, 17(3):348–375, 1978.
5. A. Mycroft and R.A. O'Keefe. A polymorphic type system for Prolog. *Artificial Intelligence*, 23(3):295–307, 1984.
6. T. Schrijvers and M. Bruynooghe. Towards constraint-based type inference with polymorphic recursion for functional and logic languages. In *Proceedings of the 17th International Workshop on Implementation and Application of Functional Languages*, pages 1–16, 2005.
7. T. Schrijvers, V. Santos Costa, J. Wielemaker, and B. Demoen. Towards Typed Prolog. In *Proceedings of the 24th International Conference on Logic Programming*, ICLP '08, pages 693–697. Springer-Verlag, 2008.
8. J.G. Siek and M. Vachharajani. Gradual typing with unification-based inference. In *Proceedings of the 2008 symposium on Dynamic languages*, page 7. ACM, 2008.
9. T. Swift and D. S. Warren. Tabling with answer subsumption: implementation, applications and performance. In *Proceedings of the 12th European conference on Logics in artificial intelligence*, JELIA'10, pages 300–312. Springer-Verlag, 2010.

ASP modulo CSP: The clingcon system

Max Ostrowski

Institut für Informatik, Universität Potsdam, August-Bebel-Str. 89, D–14482 Potsdam, Germany ostrowsk@cs.uni-potsdam.de

Abstract

Answer Set Programming (ASP; [1]) has become a prime paradigm for declarative problem solving due to its combination of an easy yet expressive modeling language with high-performance Boolean constraint solving technology. However, certain applications are more naturally modeled by mixing Boolean with non-Boolean constructs, for instance, accounting for resources, fine timings, or functions over finite domains. The challenge lies in combining the elaborated solving capacities of ASP, like backjumping and conflict-driven learning, with advanced techniques from the area of constraint programming (CP). I therefore developed the solver *clingcon*, which follows the approach of modern Satisfiability Modulo Theories (SMT; [2, Chapter 26]). My research shall contribute to bridging the gap between Boolean and Non-Boolean reasoning, in order to bring out the best of both worlds.

1998 ACM Subject Classification D.1.6. Logic Programming, I.2.3 Deduction and Theorem proving/Logic programming, D.3.2 Language Classifications/Constraint and logic languages

Keywords and phrases Answer Set Programming, Constraint Programming

Digital Object Identifier 10.4230/LIPIcs.ICLP.2012.458

1 Introduction and Motivation

clingcon is a hybrid solver for ASP, combining the simple modeling language and the high performance Boolean solving capacities of ASP with techniques for using non-Boolean constraints from the area of Constraint Programming (CP). Although *clingcon*'s solving components follow the approach of modern Satisfiability Modulo Theories (SMT; [2, Chapter 26]) solvers when combining the ASP solver *clasp* with the CP solver *gecode* [3], *clingcon* furthermore adheres to the tradition of ASP in supporting a corresponding modeling language by appeal to the ASP grounder *gringo*. Although in the current implementation the theory solver is instantiated with the CP solver *gecode*, the principal design of *clingcon* along with the corresponding interfaces are conceived in a generic way, aiming at arbitrary theory solvers.

I will first give a general background over the theory of ASP and CP and will afterwards describe the architecture of *clingcon* which follows the approach of SMT. Given this, the main contribution of my work is a comparison of simple methods to compute minimal inconsistencies and explanations for any black-box CP system. These minimal conflicts and reasons can then be used for driving the conflict-driven learning process of the system. These methods have been implemented in the system *clingcon* and yield a performance improvement of an order of magnitude on a broad range of benchmarks.

2 Background

A (*normal*) *logic program* over an alphabet $\mathcal{A}$ is a finite set of *rules* of the form

$$a_0 \leftarrow a_1, \ldots, a_m, not\ a_{m+1}, \ldots, not\ a_n \ , \tag{1}$$

© Max Ostrowski;
licensed under Creative Commons License ND
Technical Communications of the 28th International Conference on Logic Programming (ICLP'12).
Editors: A. Dovier and V. Santos Costa; pp. 458–463

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

where $a_i \in \mathcal{A}$ is an *atom* for $0 \leq i \leq n$.[1] A *literal* is an atom a or its (default) negation *not* a. For a rule r as in (1), let $head(r) = a_0$ be the *head* of r and $body(r) = \{a_1, \ldots, a_m, not\ a_{m+1}, \ldots, not\ a_n\}$ be the *body* of r. Given a set B of literals, let $B^+ = \{a \in \mathcal{A} \mid a \in B\}$ and $B^- = \{a \in \mathcal{A} \mid not\ a \in B\}$. Furthermore, given some set $\mathcal{B}$ of atoms, define $B|_{\mathcal{B}} = (B^+ \cap \mathcal{B}) \cup \{not\ a \mid a \in B^- \cap \mathcal{B}\}$. The set of atoms occurring in a logic program P is denoted by $atom(P)$. A set $X \subseteq \mathcal{A}$ is an *answer set* of a program P over $\mathcal{A}$, if X is the $\subseteq$-smallest model of the *reduct* $P^X = \{head(r) \leftarrow body(r)^+ \mid r \in P, body(r)^- \cap X = \emptyset\}$. An answer set can also be seen as a Boolean assignment satisfying all conditions induced by program P (cf. [4]).

A *constraint satisfaction problem* (CSP) is a triple (V, D, C), where V is a set of *variables* with respective *domains* D, and C is a set of *constraints*. Each variable $v \in V$ has an associated domain $dom(v) \in D$. Following [5], a constraint c is a pair (S, R) consisting of a k-ary *relation* R defined on a vector $S \subseteq V^k$ of variables, called the *scope* of R. That is, for $S = (v_1, \ldots, v_k)$, we have $R \subseteq dom(v_1) \times \cdots \times dom(v_k)$. We use $S(c) = S$ and $R(c) = R$ to access the scope and the relation of $c = (S, R)$. For an assignment $A : V \to \bigcup_{v \in V} dom(v)$ and a constraint (S, R) with $S = (v_1, \ldots, v_k)$, define $A(S) = (A(v_1), \ldots, A(v_k))$, and let $sat_C(A) = \{c \in C \mid A(S(c)) \in R(c)\}$.

The input language of *clingcon* extends the one of *gringo* (cf. [6]) by CP-specific operators marked with a preceding `$` symbol. After grounding, a propositional program is then composed of regular and constraint atoms, denoted by $\mathcal{A}$ and $\mathcal{C}$, respectively. The set of constraint atoms induces an ordinary constraint satisfaction problem (CSP) (V, D, C). This CSP is to be addressed by the corresponding CP solver, in our case *gecode*. As detailed in [7], the semantics of such constraint logic programs is defined by appeal to a two-step reduction. For this purpose, we consider a regular Boolean assignment over $\mathcal{A} \cup \mathcal{C}$ (in other words, an interpretation) and an assignment of V to D (for interpreting the variables V in the underlying CSP). In the first step, the constraint logic program is reduced to a regular logic program by evaluating its constraint atoms. To this end, the constraints in C associated with the program's constraint atoms $\mathcal{C}$ are evaluated w.r.t. the assignment of V to D. In the second step, the common Gelfond-Lifschitz reduct [8] is performed to determine whether the Boolean assignment is an answer set of the obtained regular logic program. If this is the case, the two assignments constitute a (hybrid) constraint answer set of the original constraint logic program.

In what follows, we rely upon the following terminology. We use signed literals of form $\mathbf{T}a$ and $\mathbf{F}a$ to express that an atom a is assigned $\mathbf{T}$ or $\mathbf{F}$, respectively. That is, $\mathbf{T}a$ and $\mathbf{F}a$ stand for the Boolean assignments $a \mapsto \mathbf{T}$ and $a \mapsto \mathbf{F}$, respectively. We denote the complement of such a literal ℓ by $\overline{\ell}$. That is, $\overline{\mathbf{T}a} = \mathbf{F}a$ and $\overline{\mathbf{F}a} = \mathbf{T}a$. We represent a Boolean assignment simply by a set of signed literals. Sometimes we restrict such an assignment A to its regular or constraint atoms by writing $A|_{\mathcal{A}}$ or $A|_{\mathcal{C}}$, respectively. For instance, given the regular atom '`person(adam)`' and the constraint atom '`work(adam) $> 4`', we may form the Boolean assignment $\{\mathbf{T}\texttt{person(adam)}, \mathbf{F}work(adam)\ \$>\ 4\}$.

We identify constraint atoms in $\mathcal{C}$ with constraints in (V, D, C) via a function $\gamma : \mathcal{C} \to C$. Provided that each constraint $c \in C$ has a complement $\overline{c} \in C$, like $\overline{`x = y\text{'}} = `x \neq y\text{'}$ or $\overline{`x < y\text{'}} = `x \geq y\text{'}$ and vice versa, we can extend γ to signed constraint atoms over $\mathcal{C}$ as

[1] The semantics of choice rules and integrity constraints is given through program transformations. For instance, $\{a\} \leftarrow$ is a shorthand for $a \leftarrow not\ a'$ plus $a' \leftarrow not\ a$ and similarly $\leftarrow a$ for $a' \leftarrow a, not\ a'$, for a new atom a'.

follows.

$$\gamma(\ell) = \begin{cases} c & \text{if } \ell = \mathbf{T}c \\ \overline{c} & \text{if } \ell = \mathbf{F}c \end{cases}$$

For instance, we get $\gamma(\mathbf{F}work(adam)\ \$>\ 4) = work(adam) \leq 4$, where `work(adam)` $\in V$ is a constraint variable and $(work(adam) \leq 4) \in C$ is a constraint. An assignment satisfying the last constraint is $\{work(adam) \mapsto 3\}$.

Following [4], we represent Boolean constraints issuing from a logic program under ASP semantics in terms of *nogoods* [5]. This allows us to view inferences in ASP as unit propagation on nogoods. A *nogood* is a set $\{\sigma_1, \dots, \sigma_m\}$ of signed literals, expressing that any assignment containing $\sigma_1, \dots, \sigma_m$ is unintended. Accordingly, a total assignment A is a *solution* for a set Δ of nogoods if $\delta \not\subseteq A$ for all $\delta \in \Delta$. Whenever $\delta \subseteq A$, the nogood δ is said to be *conflicting* with A. For instance, given atoms a, b, the total assignment $\{\mathbf{T}a, \mathbf{F}b\}$ is a solution for the set of nogoods containing $\{\mathbf{T}a, \mathbf{T}b\}$ and $\{\mathbf{F}a, \mathbf{F}b\}$. Likewise, $\{\mathbf{F}a, \mathbf{T}b\}$ is another solution. Importantly, nogoods provide us with reasons explaining why entries must (not) belong to a solution, and lookback techniques can be used to analyze and recombine inherent reasons for conflicts. We refer the interested reader to [4] for details on how logic programs are translated into nogoods within ASP.

3 Research Program

My research work started from the question how to combine the advantages of ASP with Non-Boolean constraint processing techniques. In the process of writing my diploma thesis I developed the hybrid solver *clingcon.* I discovered that the major difficulties lay within the conflict-driven learning techniques of ASP. A black-box CSP solver like *gecode* is not able to provide any useful evidence for its propagation. Such an evidence is needed in an advanced learning setting to provide useful reasons and conflicts for the ASP solver. I addressed this shortcoming by developing mechanisms for extracting minimal reasons and conflicts from any CP solver. The method of minimizing sets of constraints that we present are similar to the ones depicted in [9], Our method furthermore take the incremental nature of the CP solver into account. In contrast to [10], we do not incorporate the learning mechanism into the CP solver but rather use it for the interaction between the ASP and the CP solver. Furthermore, we cope with the difficulty having a black-box system as a CP solver.

Clingcon is based on an algorithm for computing constraint answer sets that extends a previous algorithm to compute standard answer sets [4] by a CP "oracle." The basic algorithm for finding standard answer sets is called Conflict-Driven Nogood Learning (CDNL); it includes conflict-driven learning and backjumping according to the First-UIP scheme [11, 12, 13]. That is, whenever a conflict happens, a conflict nogood containing a Unique Implication Point (UIP) is identified by iteratively resolving a violated nogood against a second nogood that is a reason for some literal in it. A basic CDNL algorithm is depicted in Algorithm 1.

The principal design of *clingcon* along with the corresponding interfaces are conceived in a generic way, aiming at arbitrary theory solvers. The first extension concerns the input language of *gringo* with theory-specific language constructs. Just as with regular atoms, the grounding capabilities of *gringo* can be used for dealing with constraint atoms containing first-order variables. As regards the current *clingcon* system, the language extensions allow for expressing constraints over integer variables. This involves arithmetic constraints as well as global constraints and optimization statements. These constraints are treated as atoms

Algorithm 1: CDNL-ASPmCSP

```
   input  : A program Π.
   output : A constraint answer set of Π.
 1 loop
 2   Propagation
 3   if hasConflict then
 4     if decisionLevel = 0 then return no Answer Set
 5     ConflictAnalysis
 6     Backjump
 7   else if complete Assignment then
 8     Labeling
 9     if hasConflict then
10       Backjump
11     else
12       return Constraint Answer Set
13   else
14     Select
```

and passed to the ASP solver. Information about these constraints is furthermore directly shared with the theory propagator and in turn the theory solver, viz. *gecode*. The theory propagator is implemented as a post propagator. Theory propagation is done by the theory solver until a fixpoint is reached. In doing so, decided constraint atoms are transferred to the theory solver, and conversely constraints whose truth values are determined by the theory solver are sent back to the ASP solver using a corresponding nogood. Note that theory propagation is not only invoked when propagating partial assignments but also whenever a total Boolean assignment is found. Whenever the theory solver detects a conflict, the theory propagator is in charge of conflict analysis. Apart from reverting the state of the theory solver upon backjumping, this involves the crucial task of determining a conflict nogood (which is usually not provided by theory solvers, as in the case of *gecode*). Similarly, the theory propagator is in charge of enumerating constraint variable assignments, whenever needed. Determining a good conflict nogood is the main part of my research that I want to present.

After doing theory propagation either a conflict occurs or some constraints (boolean literals in our case) could be evaluated to true or false. In both cases an explanation is needed, either in form of a conflict or a reason nogood. The *simple* version of generating the conflicting nogood N, is just to take the entire assignment of constraint literals. In this way, all yet decided constraint atoms constitute $N = \{\ell \mid \ell \in A|_{\mathcal{C}}\}$. The corresponding list of inconsistent constraints is

$$I = [\,\gamma(\ell) \mid \ell \in A|_{\mathcal{C}}\,]. \tag{2}$$

In order to reduce this list of inconsistent constraints and to find the real cause of the conflict, we apply an *Irreducible Inconsistent Set* (IIS) algorithm. The term IIS was coined in [14] for describing inconsistent sets of constraints having consistent subsets only. We use the concept of an IIS to find the minimal cause of a conflict. With this technique, [9] showed that it is actually possible to drastically reduce such exhaustive sets of inconsistent constraints as in (2) and to create a much smaller conflict nogood. Similar to the algorithm

Algorithm 2: FORWARD_FILTERING

input : An inconsistent list of constraints $I = [c_1, \dots, c_n]$.
output : An irreducible inconsistent list of constraints I'.

```
1 I' ← []
2 while I' is consistent do
3 |  T ← I'
4 |  i ← 1
5 |  while T is consistent do
6 |  |  T ← T ∘ c_i
7 |  |  i ← i + 1
8 |  I' ← I' ∘ c_i
9 return I'
```

in [9], we developed a set of algorithms that exploits the features of an incremental CSP solver even more. I will shortly explain one of these algorithms. Algorithm 2 is called Forward Filtering; it is designed to avoid resetting the search space of the CP solver. It incrementally adds constraints to a testing list T, starting from the first assigned constraint to the last one (lines 5 and 6). Remember that incrementally adding constraints is easy for a CP solver as it can only further restrict the domains. If our test list T becomes inconsistent we add the currently tested constraint to the result I' (lines 5 and 8). If this result is inconsistent (Line 2), we have found a minimal list of inconsistent constraints. Otherwise, we start again, this time adding all yet found constraints I' to our testing list T (Line 1). Now we have to create a new constraint space. But by incrementally increasing the testing list, we already reduced the number of potential candidates that contribute to the IIS, as we never have to check a constraint behind the last added constraint. We illustrate this again on a little example. We start Algorithm 2 with $T = I' = []$ and

$$
\begin{aligned}
I =&[work(lea) = work(adam), work(john) = 0, work(smith) = 0] \\
&\circ[work(adam) + work(lea) > 6, work(lea) - work(adam) = 1]
\end{aligned}
$$

in Line 3. We add $work(lea) = work(adam)$ to T, as this constraint alone is consistent, we loop and add constraints until $T = I$. As this list is inconsistent, we add the last constraint $work(lea) - work(adam) = 1$ to I' in Line 8. We can do so, as we know that the last constraint is indispensable for the inconsistency. As I' is consistent we restart the whole procedure, but this time setting $T = I' = [work(lea) - work(adam) = 1]$ in Line 3. Please note that, even if I would contain further constraints, we would never have to check a constraint behind $work(lea) - work(adam) = 1$. Our testing list already contained an inconsistent set of constraints, consequently we can restrict ourself to this subset. Now we start the loop again, adding $work(lea) = work(adam)$ to T. On their own, those two constraints are inconsistent, as there exists no valid pair of values for the variables. So we add $work(lea) = work(adam)$ to I', resulting in $I' = [work(lea) - work(adam) = 1, work(lea) = work(adam)]$. With this much smaller conflict we hope to speed up the search process.

But we can do even more. Up to now we only considered reducing an inconsistent list of constraints to reduce the size of a conflicting nogood. If the CP solver propagates the literal l, a *simple* reason nogood is $N = \{\ell \mid \ell \in A|_{\mathcal{C}}\} \cup \{\bar{l}\}$. If we have for example $A|_{\mathcal{C}} = \{\mathbf{T}work(john)\$ == 0, \mathbf{T}work(lea) - work(adam)\$ == 1\}$, the CP solver propagates the literal $\mathbf{F}work(lea)\$ == work(adam)$. To use the proposed algorithms

to reduce a reason nogood we first have to create an inconsistent list of constraints. As $J = [\gamma(\ell) \mid \ell \in A|_{\mathcal{C}}]$ implies $\gamma(l)$, this inconsistent list is $I = J \circ [\overline{\gamma(l)}] = [work(john) = 0, work(lea) - work(adam) = 1, work(lea) = work(adam)]$. So we can now use these various filtering methods also to reduce reasons generated by the CP solver. In this case the reduced reason is
$\{\mathbf{T}work(lea) - work(adam)\$ == 1, \mathbf{T}work(lea)\$ == work(adam)\}$. Smaller reasons reduce the size of conflicts even more, as they are constructed using unit resolution.

Evaluation on various benchmarks showed that, also filtering conflicts and reasons is a very time consuming process, it can speed up search by order of magnitudes.

4 Future Work

In my future work I want to focus on these reasons and also on a combination of the so called lazy approach which is implemented in *clingcon* with a translational approach. Currently only the ASP solver profits from the additional knowledge of the CP solver. I want to strengthen the CP solving capabilities with features from ASP such as dedicated heuristics like VSIDS and BerkMin and learning.

References

1. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press (2003)
2. Biere, A., Heule, M., van Maaren, H., Walsh, T., eds.: Handbook of Satisfiability. Volume 185 of Frontiers in Artificial Intelligence and Applications. IOS Press (2009)
3. `http://www.gecode.org`
4. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Conflict-driven answer set solving. In Veloso, M., ed.: Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI'07), AAAI Press/The MIT Press (2007) 386–392
5. Dechter, R.: Constraint Processing. Morgan Kaufmann Publishers (2003)
6. Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., Thiele, S.: A user's guide to `gringo`, `clasp`, `clingo`, and `iclingo`. Available at `http://potassco.sourceforge.net`
7. Gebser, M., Ostrowski, M., Schaub, T.: Constraint answer set solving. In Hill, P., Warren, D., eds.: Proc. of the 25th International Conference on Logic Programming (ICLP'09). Volume 5649 of Lecture Notes in Computer Science., Springer-Verlag (2009) 235–249
8. Gelfond, M., Lifschitz, V.: Classical negation in logic programs and disjunctive databases. New Generation Computing **9** (1991) 365–385
9. Junker, U.: QuickXPlain: Conflict detection for arbitrary constraint propagation algorithms. IJCAI'01 Workshop on Modelling and Solving problems with constraints (2001)
10. Moore, N.: Improving the Efficiency of Learning CSP Solvers. University of St Andrews thesis. University of St Andrews (2011)
11. Marques-Silva, J., Sakallah, K.: GRASP: A search algorithm for propositional satisfiability. IEEE Transactions on Computers **48**(5) (1999) 506–521
12. Zhang, L., Madigan, C., Moskewicz, M., Malik, S.: Efficient conflict driven learning in a Boolean satisfiability solver. In: Proceedings of the International Conference on Computer-Aided Design (ICCAD'01). (2001) 279–285
13. Mitchell, D.: A SAT solver primer. Bulletin of the European Association for Theoretical Computer Science **85** (2005) 112–133
14. van Loon, J.: Irreducible inconsistent systems of linear inequalities. In: European Journal of Operational Research. Volume 8., Elsevier Science (1981) 283–288

An ASP Approach for the Optimal Placement of the Isolation Valves in a Water Distribution System

Andrea Peano

EnDiF, Università degli Studi di Ferrara
via G. Saragat 1 – 44122, Ferrara, Italy
andrea.peano@unife.it

Abstract

Several design issues of Water Distribution Systems can be represented as combinatorial optimization problems, and then addressed by means of opportune techniques and technologies available in Computational Logic and Operational Research. My Ph.D. Thesis relates to achieve (near-)optimal solutions to such real-life problems either by exploiting potentialities of existing techniques and by developing ad hoc algorithms. Among all the design issues above mentioned, the Isolation Valve Location Problem is defined as the problem of computing the optimal placement, on the hydraulic network, of a limited number of *isolation valves*, so that any pipe can be isolable in case of failure and the maximum service disruption (varying the broken pipe) is minimized. About Computational Logic, different Answer Set Programming encodings to such a problem have been developed during the first stage of my research activity, and more suitable encodings are currently under study.

1998 ACM Subject Classification D.1.6 Logic Programming

Keywords and phrases Answer Set Programming, Isolation Valves Positioning, Hydroinformatics

Digital Object Identifier 10.4230/LIPIcs.ICLP.2012.464

1 Introduction

My Ph.D. Thesis relates to real-life optimization problems in the hydraulic engineering field. More precisely, with the collaboration of computer scientists, operational researchers and hydraulic engineers, I investigate and exploit potentialities of various Operational Research and Artificial Intelligence techniques in order to achieve good (and, whenever possible, optimal) solutions for those particular design issues of the urban hydraulic network that can be effectively modelled as known combinatorial optimization problems. Furthermore, such design issues often require to devise new specialized variants of the known combinatorial optimization problems.

For example, the problem of minimizing the impact of a contamination in a hydraulic network can be seen, under opportune assumptions, as a variant of the well known *Multiple Traveling Salesman Problem* (MTSP); since the quality of feasible solutions must be computed through a burdensome hydraulic simulation, such optimization problem was addressed by us by means of several genetic algorithms [6]. In particular, in [6], we proposed a novel genetic encoding for the MTSP for which we defined new genetic crossover operators based on ad hoc mixed integer linear programming (sub-)optimizations, obtaining a hybrid genetic algorithm.

Another real-life combinatorial optimization problem which is a typical issue during the design of a hydraulic network is finding the optimal positioning of a limited number of isolation valves on the network. Up to now, we exploited two different technologies, following

© Andrea Peano;
licensed under Creative Commons License ND
Technical Communications of the 28th International Conference on Logic Programming (ICLP'12).
Editors: A. Dovier and V. Santos Costa; pp. 464–468

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

two independent approaches: in the first we modelled the above mentioned problem by means of a Bilevel (Mixed Integer) Linear Programming [3] formalization, discussed in [13]; I presented the study at the 3rd Student Conference on Operational Research (SCOR 2012). In the second approach, we addressed such optimization problem by defining several Answer Set Programming (ASP)[1, 11, 8] programs, discussed in [5]. Both the optimization approaches compute the globally optimum placement of the valves.

In the next section I will briefly describe the problem of the isolation valve placement on hydraulic networks and the ASP approach designed to solve it.

2 The Isolation Valves Location Problem

Water Distributions Systems (WDSs) are strategic urban infrastructures. Their planning is, in turn, a strategic task in terms of costs control and to assure a fair degree of reliability. For example, during the design of a water distribution network, one of the choices is the design of the isolation system. It is a real-life problem for hydraulic engineers, and in recent years it has been studied through computational methods in the *hydroinformatics* literature [9, 4].

A water distribution system has the main objective of providing water to homes and facilities that require it. The water distribution network can be thought of as a labelled indirected graph, in which the edges represent the pipes in the network. There is at least one special node that represents the source of water (node 1 in Figure 1), and the users' homes are connected to the edges. For each edge, we assume to have knowledge about the average amount of water (in litres per second) that is drawn by the users insisting on that edge (during the day); such value is the label associated to the edge, and it is called the users' *demand*.

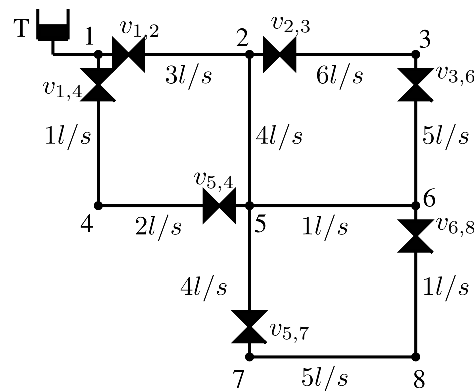


Figure 1 A water distribution network with valves.

The isolation system is mainly used during repair operations: in case some pipe is damaged, it has to be fixed or substituted. However, no repair work can be done while the water is flowing at high pressure in the pipe: first the part of the network containing the broken pipe should be de-watered, then workers can fix the pipe. The de-watering is performed by closing an opportune set of *isolation valves* (that make up the so-called *isolation system* of the water distribution network) so that the damaged pipe is disconnected from the sources. For example, in Figure 1, if the edge connecting nodes 2 and 3 (let us call it $e_{2,3}$) is broken, workers can close valves $v_{2,3}$ and $v_{3,6}$ and de-water the broken pipe. Of course, during this pipe substitution the users that take water from edge $e_{2,3}$ cannot be serviced.

The usual measure of disruption is the *undelivered demand*: in this case, it corresponds to the demand of the users insisting on the broken pipe, namely $6l/s$.

However, we are not always this lucky: in case the damaged pipe is $e_{7,8}$, workers will have to close valves $v_{5,7}$ and $v_{6,8}$, de-watering pipes $e_{7,8}$ and $e_{6,8}$, with a total cost of $5+1=6l/s$. In fact, the minimum set of pipes that will be de-watered is that belonging to the so-called *sector* of the broken pipe, i.e., the set of pipes encircled by a same set of valves. But there can be even worse situations: if the broken pipe is $e_{2,5}$, workers have to close valves $v_{1,2}$ and $v_{5,4}$, which means disconnecting all the pipes except $e_{1,4}$ and $e_{4,5}$, with an undelivered demand of $3+4+6+5+1+4+5+1=29l/s$. Notice in particular that the edges $e_{2,3}$, $e_{7,8}$ and $e_{6,8}$ are disconnected in this way, although they do not belong to the same sector as the broken pipe. This effect is called *unintended isolation*, and usually means that the isolation system was poorly designed.

One common value used by hydraulic engineers [9] to measure the quality of the isolation system is the undelivered demand in the worst case. In the example of Figure 1, the worst case happens when the broken pipe is in the set $\{e_{1,2}, e_{2,5}, e_{5,6}, e_{5,7}\}$; in this case, as we have seen, the undelivered demand is $29l/s$.

In a previous work, [2] developed a system, based on Constraint Logic Programming [10] on Finite Domains (CLP(FD)), that finds the optimal positioning of a given number of valves in a water distribution network. The assignments found by [2] improved the state-of-the-art in hydraulic engineering for this problem, finding solutions with a lower (worst-case) undelivered demand than the best solutions known in the literature of hydraulic engineering [9], obtained through genetic algorithms.

In the current work, we address the same problem in Answer Set Programming [1, 11, 8], which is a suitable technology to address combinatorial graph problems[12], and, in particular, we have already defined two different ASP programs [5]. One program explicitly defines the *sectors* as clusters of (isolated) pipes and minimizes the undelivered demand of the worst sector; instead, in the other program, sectors are left implicit and the aim is to maximize the minimum satisfied demand in case of pipe isolation, by considering that a pipe is isolated if it is not reachable from any source. In the next section we show the most important results obtained by first experiments.

3 Results

The first experiments, presented in [5], show, in general, that the developed programs take more computation time than the CLP(FD) approach [2]. However, we must say that the CLP(FD) model was developed by two CLP experts, during some person-months and was trimmed for efficiency. Instead, the two ASP formulations were mainly developed by a first-year PhD student in about one week; this shows that ASP is very intuitive and easy to understand even for non experts, that it is indeed very declarative. The two implemented ASP programs consist of respectively about 20 and 25 rules, which shows that ASP is a very interesting technology for rapid prototyping.

Experiments have been performed on a *Intel* based architecture with two P8400 CPUs; as ASP solver we used the Potassco's solver *Clasp* [7]. The two programs have been optimized using a real-life instance based on the Apulian hydraulic network [9] (Figure 2) and varying the number of available isolation valves.

Figure 3 shows the optimization performance of the two ASP programs (the one based on sectors and the one that, instead, does not define sectors) for several number of available valves. In particular, the better effectiveness of the sector-based program can be noticed.

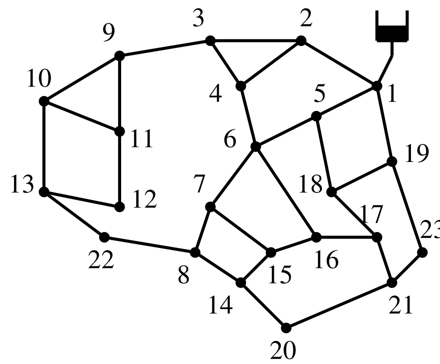


Figure 2 The Apulian hydraulic network.

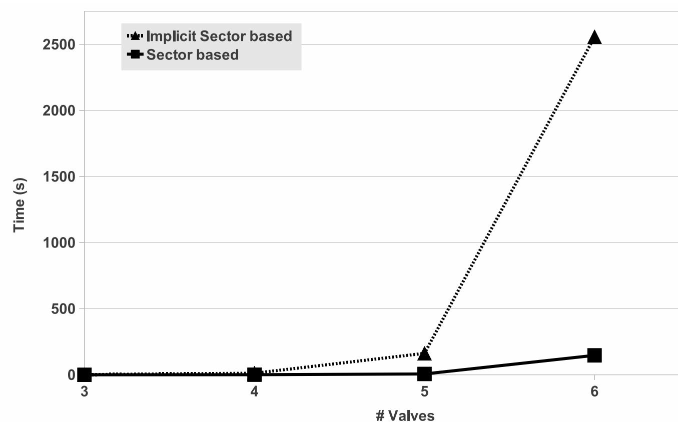


Figure 3 Computing times of the optimization processes of the two ASP programs.

4 Future Work

In future work, we plan to improve the sector-based ASP program by defining opportune rules in order to break the symmetries determined by the current formalization, and we will experiment the resulting program also with other available ASP solvers. Another appealing challenge is to compute the solution that minimizes the undelivered demand of the worst sector as well as the undelivered demands of the other (no worst) sectors, which are not actually optimized neither by our current MILP formalization [13] nor by the CLP(FD) one [2]. We are also interested in trying to integrate the ASP programs with a CLP approach, to take advantage of the strengths of the two approaches. Finally, I plan to delve into the ASP theory and techniques in order to consolidate my competence in such Artificial Intelligence field.

Acknowledgements. This work was partially supported by EU project *ePolicy*, FP7-ICT-2011-7, grant agreement 288147. Possible inaccuracies of information are under the responsibility of the project team. The text reflects solely the views of its authors. The European Commission is not liable for any use that may be made of the information contained in this paper.

References

1. Chitta Baral. *Knowledge representation, reasoning and declarative problem solving.* Cambridge University Press, 2003.
2. Massimiliano Cattafi, Marco Gavanelli, Maddalena Nonato, Stefano Alvisi, and Marco Franchini. Optimal placement of valves in a water distribution network with CLP(FD). *Theory and Practice of Logic Programming*, 11(4-5):731–747, 2011.
3. Benoît Colson, Patrice Marcotte, and Gilles Savard. Bilevel programming: A survey. *4OR: A Quarterly Journal of Operations Research*, 3:87–107, 2005. 10.1007/s10288-005-0071-0.
4. Enrico Creaco, Marco Franchini, and Stefano Alvisi. Optimal placement of isolation valves in water distribution systems based on valve cost and weighted average demand shortfall. *Water Resources Management*, 24:4317–4338, 2010. 10.1007/s11269-010-9661-5.
5. Marco Gavanelli, Maddalena Nonato, Andrea Peano, Stefano Alvisi, and Marco Franchini. An ASP approach for the valves positioning optimization in a water distribution system. In Francesca Lisi, editor, *9th Italian Convention on Computational Logic (CILC 2012), Rome, Italy*, volume 857 of *CEUR workshop proceedings*, pages 134–148, 2012.
6. Marco Gavanelli, Maddalena Nonato, Andrea Peano, Stefano Alvisi, and Marco Franchini. Genetic algorithms for scheduling devices operation in a water distribution system in response to contamination events. In Jin-Kao Hao and Martin Middendorf, editors, *Evolutionary Computation in Combinatorial Optimization*, volume 7245 of *Lecture Notes in Computer Science*, pages 124–135. Springer Berlin / Heidelberg, 2012. 10.1007/978-3-642-29124-1_11.
7. Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Max Ostrowski, Torsten Schaub, and Marius Schneider. Potassco: The Potsdam answer set solving collection. *AI Communications*, 24(2):105–124, 2011.
8. Michael Gelfond. Answer sets. In *Handbook of Knowledge Representation*, chapter 7. Elsevier, 2007.
9. Orazio Giustolisi and Dragan A. Savić. Identification of segments and optimal isolation valve system design in water distribution networks. *Urban Water Journal*, 7(1):1–15, 2010.
10. Joxan Jaffar and Michael J. Maher. Constraint logic programming: A survey. *J. Log. Program.*, 19/20:503–581, 1994.
11. Nicola Leone. Logic programming and nonmonotonic reasoning: From theory to systems and applications. In Chitta Baral, Gerhard Brewka, and John Schlipf, editors, *Proceedings of the 9th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'07)*, volume 4483 of *Lecture Notes in Computer Science*. Springer, 2007.
12. Ilkka Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25:241–273, 1999. 10.1023/A:1018930122475.
13. Andrea Peano, Maddalena Nonato, Marco Gavanelli, Stefano Alvisi, and Marco Franchini. A Bilevel Mixed Integer Linear Programming Model for Valves Location in Water Distribution Systems. In Stefan Ravizza and Penny Holborn, editors, *3rd Student Conference on Operational Research*, volume 22 of *OpenAccess Series in Informatics (OASIcs)*, pages 103–112, Dagstuhl, Germany, 2012. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

Answer Set Programming with External Sources

Christoph Redl

Institute of Information Systems, TU Vienna
Karlsplatz 13, 1040 Vienna, Austria

Abstract

Answer Set Programming (ASP) is a well-known problem solving approach based on nonmonotonic logic programs and efficient solvers. To enable access to external information, HEX-programs extend programs with external atoms, which allow for a bidirectional communication between the logic program and external sources of computation (e.g., description logic reasoners and Web resources). Current solvers evaluate HEX-programs by a translation to ASP itself, in which values of external atoms are guessed and verified after the ordinary answer set computation. This elegant approach does not scale with the number of external accesses in general, in particular in presence of nondeterminism (which is instrumental for ASP). Hence, there is a need for genuine algorithms which handle external atoms as first-class citizens, which is the main focus of this PhD project.

In the first phase of the project, state-of-the-art conflict driven algorithms were already integrated into the prototype system dlvhex and extended to external sources. In particular, the evaluation of external sources may trigger a learning procedure, such that the reasoner gets additional information about the internals of external sources. Moreover, problems on the second level of the polynomial hierarchy were addressed by integrating a minimality check, based on unfounded sets. First experimental results show already clear improvements.

1998 ACM Subject Classification I.2.3 Deduction and Theorem Proving; I.2.4 Knowledge Representation Formalisms and Methods

Keywords and phrases Answer Set Programming, Nonmonotonic Reasoning, External Computation, FLP Semantics

Digital Object Identifier 10.4230/LIPIcs.ICLP.2012.469

1 Introduction and Problem Description

1.1 The Answer-Set Programming Paradigm

In recent years, the *Answer Set Programming* (ASP) paradigm has emerged as an important approach for declarative problem solving Problems are represented in terms of nonmonotonic logic programs, such that the models of the latter encode the solutions of a problem at hand. The most widely used notions of models in this context are *stable models* [11] and the generalized notion of *answer sets* for a (possible disjunctive) logic program [12]. One of the main reasons for the increasing popularity of ASP is the availability of sophisticated solvers for the respective languages, including `DLV` [14] and and `clasp` [10].

Formally, a propositional *answer set program* is a set of rules r of form

$$a_1 \vee \cdots \vee a_k \leftarrow b_1, \ldots, b_m, not\ b_{m+1}, \ldots, not\ b_n \qquad (1)$$

where *not* denotes default-negation, and $a_i, 1 \leq i \leq k$, $b_j, 1 \leq j \leq n$ are literals, i.e., propositional atoms p or strongly (classically) negated atoms $\neg p$. Intuitively, a rule is satisfied by an interpretation, i.e. set of classical literals, I, iff either $b_i \notin I$ for some

© Christoph Redl;
licensed under Creative Commons License ND

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

$1 \leq i \leq m$, $b_i \notin I$ for some $m+1 \leq i \leq n$, or $a_i \in I$ for some $1 \leq i \leq k$. A set of literals is an answer set of a program P, iff it is subset-minimal and satisfies all rules of P.

1.2 External Sources

The rise of the World Wide Web and distributed systems fostered the development of formalisms that are geared towards modularity and integrating multiple data sources. Various extensions of ASP that allow to access information in external sources different from logic programs have been proposed, e.g., the DLV^{DB} system [20] and HEX-*programs* [5]. The latter accommodate a universal bidirectional interface for arbitrary sources of external computation through the notion of an external atom and are in the focus of this PhD project. For example, an external atom of the form $\&synonym[thes_file, car](X)$ might be used to access an external thesaurus "*thes_file*", and to retrieve all terms X in it that are synonyms of "*car*" (i.e., the atom evaluates to true). Using such external atoms, whose semantics is abstractly modeled by an input-output relationship, one can easily access different kinds of information resources and reason about them in a single program.

HEX-programs have been successfully used in various application domains; e.g., in the Semantic Web. A solver for HEX-programs is dlvhex [4], which implements a compilation approach for evaluation. First external atoms are replaced by ordinary atoms, and their truth value is guessed nondeterministically by disjunctive rules. The resulting program is then evaluated by the use of existing reasoners for answer set programs. In a postprocessing step, the guess for the external atoms is validated, i.e., the result is filtered to keep only those answer sets which are stable also under external sources. However, the performance of the reasoner is unsatisfactory. This is caused by the fact that the actual model building is hidden in the ASP solver employed at the backend. The development of scalable genuine algorithms is therefore the overall goal of the PhD project.

2 Background and Overview of the Existing Literature

The following subsections summarize approaches for three related topics which will play a role for the proposed project.

2.1 Model Finding for Propositional Ordinary Answer Set Programs

Model finding strategies for ASP programs can be classified in two major groups. The first one consists of algorithms that *reduce* the problem to another host logic, for which one can apply specialized SAT solvers. Approaches of the second kind search directly for models and are called *genuine* algorithms.

Genuine algorithms (as e.g. implemented by DLV) essentially boil down to an intelligent (restricted) enumeration of truth assignments. Deterministic consequences of the rules under partial truth assignments are computed in order to set the truth values of further atoms. If the assignment is still partial after this step, the value of an yet undefined atom is guessed in the style of DPLL algorithms for SAT, and again deterministic consequences are determined, etc.; in case the guess fails, the computation backtracks and the alternative value is considered. In contrast to DLV, the CLASP system [10] employs a conflict-driven method corresponding to conflict driven SAT solvers [15]. The distinguishing feature is *learning*: Whenever a conflict emerges, the literals which were initially responsible for the conflict are determined and recorded to prevent the reasoner from running into the same conflict again.

2.2 Intelligent Grounding

Non-ground answer set programs are like propositional programs but the atoms in a rule (1) are of the form $p(t_1, \ldots, t_n)$, where p is a (first-order) predicate and the t_i are terms (usually function-free) in a first order language. The semantics of such a program P is defined in terms of its *grounding*, which consists of all possible ground instances of the rules in P.

Most current ASP solvers (including DLV and CLASP) step to the grounding of a program before the actual model finding algorithms are started. In contrast, *lazy grounding*, as used for instance in GASP [16] and the ASPeRiX solver [13], is an alternative to pregrounding. Rules are only grounded when the body is satisfied, and consequently it prevents the grounding of rules which are potentially unnecessary.

2.3 External Domains

Clingo (gringo+CLASP) provides an interface which allows the user to call Lua functions (http://www.lua.org), a lightweight and embeddable scripting language, at certain points during evaluation, e.g., before grounding, after a model has been found, and after termination [9]. While communication between the reasoner and external scripts is possible, it is constrained to happen between specific evaluation phases and is not tightly coupled to and interleaved with model building, in contrast to HEX-programs.

DLV-EX and DLV-Complex are ASP systems extending DLV by external predicates and complex values like lists and sets. This allows to use sources of computation that are defined outside the logic program [1], which is especially useful for functions that are difficult or not efficiently expressible by rules (e.g. string manipulation functions).

3 Goal of the Research

I now describe the expected main outcomes of the PhD project.

3.1 Algorithms

One of the main results are newly developed efficient, native model-finding algorithms for ASP programs with external source access, which push the frontier of applicability. This allows to evaluate programs with external calls of reasonable size efficiently, relative to the intrinsic complexity.

3.2 Scalable Computation

The novel algorithms will relieve the current evaluation bottleneck of external accesses, and will contribute to position ASP programs with external access better as a formalism for declarative problem solving in real-world application settings. For programs of benign structure, I expect an exponential speedup in certain situations. In the general case, I aim at computation times comparable to ordinary ASP solvers, modulo complexity of external sources.

3.3 Prototype Implementation

The theoretical contributions are then realized in the dlvhex reasoner. The main practical outcome will be a prototype implementation of the developed evaluation algorithms as a proof of concept. The implementation will be evaluated using existing and newly developed benchmarks, and will be made publicly available through a website.

4 Current Status of the Research

At this point of the project, a state-of-the-art conflict-driven ASP solver was integrated into our prototype system dlvhex and extended by *additional learning techniques* to capture the semantics of external atoms. Empirical experiments show already a significant speedup compared to the traditional evaluation method, which is based on a translation to ordinary ASP programs. Results were published in an accepted ICLP 2012 paper [3].

A current topic is the development of techniques for *minimality checking* of answer set candidates in order to extend the algorithms to problems on the second level of the polynomial hierarchy. While the minimality check is polynomial for ordinary disjunction-free ASP programs, it becomes intractable in presence of disjunctions or nonmonotonic external sources. Hence, the development of efficient algorithms is not straightforward. While the traditional evaluation algorithm for HEX does the check explicitly, i.e., it directly searches for smaller models, I am currently working on a new algorithm based on *unfounded sets* [8]. Experiments show that this approach is superior to explicit minimality checking, and that optimization techniques for search space pruning are applicable. The approach is under preparation for being submitted as a conference paper.

Another current issue concerns *syntactic criteria* which allow for a simpler evaluation. The idea is that for programs of a certain structure, tailored and more efficient algorithms may employed.

5 Preliminary Results Accomplished

After finishing my master's studies in July 2010 (*Computational Intelligence*) and October 2010 (*Medical Computer Science*), I started my PhD studies in October 2010 and got a position as a research assistant at the Institute of Information Systems at TU Vienna, while the research project at our institute (*Evaluation of ASP Programs with External Source Access*), which I am currently involved in, started in January 2012 and is planned for 3 years.

In the first year my PhD studies I have published the main results of my two master's theses in three conference papers. The first thesis in my studies of *Computational Intelligence* is on the development of a belief merging framework, called MELD, for the HEX-program solver dlvhex [17]. More specifically, the integration of heterogeneous data sources is abstractly described by tree-shaped *merging plans*, where the data sources appear in the leaf nodes and different *merging operators* in the inner nodes [19].

The MELD system is based on an extension of HEX-programs which allows for *program nesting*. That is, a HEX-program can contain calls to other programs and reason about their answer sets. Hence, answer sets are turned into accessible objects. Besides its application in the belief merging framework, program nesting has turned out to be useful also for the implementation of user-defined aggregation functions, which need to reason over multiple answer sets of subprograms. The subsystem for the evaluation of nested programs was described in another publication [7].

My second master's thesis in my studies of *Medical Computer Science* deals with a particular application of the belief merging framework MELD in medical and biological applications [18]. In particular, I developed *concrete merging operators* which are useful for the integration of *decision diagrams*, which are frequently used in medicine and related sciences to describe decision processes (e.g. for DNA classification). More results of this thesis were published in [6].

Besides writing the mentioned publications, I participated in writing the project proposal for our project to the Austrian science fond during the first year of my PhD studies. The

overall goal of the project is the development of new evaluation algorithms for HEX-programs. This turned out to be necessary in order to make HEX-programs practically useful, as I discovered during my work on my master's theses that the scalability of the current algorithms is limited. The project was approved in September 2011 and started in January 2012.

In the first phase of the project, the ASP solver clasp was integrated into dlvhex as new backend, replacing DLV. This allows for a much tighter coupling of the solver with external source evaluation, exploiting clasp's extensible SMT interface, and is the foundation for the further development. At this point of the project I have developed *external behavior learning* (EBL) as a new learning strategy besides classical conflict-driven learning as used by ordinary ASP solvers. While conflict-driven learning gains new knowledge from conflict situations, EBL learns from evaluations of external sources. This knowledge can be used in the further search to exclude model candidates beforehand if they are not compliant with the external sources. The learned knowledge about external sources is itself represented as a nogood clause and recorded in the solver. As the algorithm proceeds, this knowledge is used to guide the search.

The prototype implementation of *external behavior learning* (EBL) shows already positive effects wrt. performance. Depending on the program structure and the external atoms involved, I could observe an up to exponential speedup in some cases, e.g., for some programs with DL-atoms; DL-atoms allow for querying description logic knowledge bases from the logic program by the use of external atoms. Also for string manipulation functions, which are conceptually simple but important from a practical point of view, I could observe very promising improvements by EBL and exploiting functionality. Details of EBL are described in an accepted ICLP 2012 and TPLP paper [3].

6 Open Issues and Expected Achievements

The current translation-based HEX-evaluation algorithm does not scale well to real-world applications because of the high number of generated model candidates. The situation is comparable to the evolution of the answer set semantics as a programming paradigm: at the very beginning, the lack of efficient ASP solvers prevented its use in practice. But since efficient algorithms and ASP systems have become available, ASP has gained momentum in a range of applications, from science over humanities to industrial use. The main goal of the PhD project is significantly better scalability. It is expected that an exponential speedup is achievable in some cases, whereas a prediction in the average case is difficult to make.

One step towards this goal was the development of external behavior learning (EBL). In the next step, the basic idea shall be refined. Additionally to deriving knowledge from known properties of external sources, it shall also be possible to write *customized learning functions* for specific sources. The idea is that the provider of an external source often has a better insight to the behavior of the source, which can be used to help the reasoner excluding model candidates early. For this purpose, a user-friendly language shall be developed, which is an open issue.

Another important open issue, which was disregarded until now, concerns the minimality check of model candidates. While I have recently developed a minimality checking algorithm based on unfounded sets [8] (instead of the explicit minimality check, which was used until now), it is still realized as a post-check. That is, the algorithm first constructs an answer set candidate, and then filters out those candidates which contain unfounded sets. An open issue is *interleaving* the unfounded set search with the main search for answer sets. The idea is that one can in some cases already identify unfounded sets when the interpretation is

still partial. Then it makes no sense to further complete the interpretation, but it is more reasonable to immediately backtrack.

Case-based reasoning (CBR) is solving of current computational problems by the use of relevant aspects of past problem solutions [2]. This topic seems to be a related to HEX-program evaluation using EBL. An open issue is therefore also the investigation to what extend CBR techniques can be adopted for our purposes.

References

1. F. Calimeri, S. Cozza, and G. Ianni. External Sources of Knowledge and Value Invention in Logic Programming. *Annals of Mathematics and Artificial Intelligence*, 50(3–4):333–361, Aug. 2007.
2. R. L. de Mántaras and E. Plaza. Case-based reasoning: An overview. *AI Communications Journal*, 10(1):21–29, 1997.
3. T. Eiter, M. Fink, T. Krennwallner, and C. Redl. Conflict-driven ASP solving with external sources. *Theory and Practice of Logic Programming: Special Issue ICLP*, 2012. To appear.
4. T. Eiter, G. Ianni, R. Schindlauer, and H. Tompits. dlvhex: A Prover for Semantic-Web Reasoning under the Answer-Set Semantics. In *Proceedings of the ICLP'06 Workshop on Applications of Logic Programming in the Semantic Web and Semantic Web Services (ALPSWS2006)*, pages 33–39. CEUR WS, 2006.
5. T. Eiter, G. Ianni, R. Schindlauer, and H. Tompits. Effective Integration of Declarative Rules with External Evaluations for Semantic-Web Reasoning. In *Proceedings of the 3rd European Conference on Semantic Web (ESWC 2006)*, volume 4011 of *LNCS*, pages 273–287. Springer, 2006.
6. T. Eiter, T. Krennwallner, and C. Redl. Declarative merging of and reasoning about decision diagrams. In A. D. Palù, A. Dovier, and A. Formisano, editors, *Workshop on Constraint Based Methods for Bioinformatics (WCB 2011), Perugia, Italy, September 12, 2011*, pages 3–15. Dipartimento di Matematica e Informatica, Universita degli Studi di Perugia, September 2011.
7. T. Eiter, T. Krennwallner, and C. Redl. Nested hex-programs. *CoRR*, abs/1108.5626, 2011.
8. W. Faber. Unfounded sets for disjunctive logic programs with arbitrary aggregates. In *In Logic Programming and Nonmonotonic Reasoning, 8th International Conference (LPNMR'05), 2005*, pages 40–52. Springer Verlag, 2005.
9. M. Gebser, B. Kaufmann, R. Kaminski, M. Ostrowski, T. Schaub, and M. Schneider. Potassco: The Potsdam Answer Set Solving Collection. *AI Commun.*, 24(2):107–124, 2011.
10. M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. *Clasp* : A conflict-driven answer set solver. In C. Baral, G. Brewka, and J. S. Schlipf, editors, *LPNMR*, volume 4483 of *Lecture Notes in Computer Science*, pages 260–265. Springer, 2007.
11. M. Gelfond and V. Lifschitz. The Stable Model Semantics for Logic Programming. In R. Kowalski and K. Bowen, editors, *Logic Programming: Proceedings of the 5th International Conference and Symposium*, pages 1070–1080. MIT Press, 1988.
12. M. Gelfond and V. Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, 9(3–4):365–386, 1991.
13. C. Lefèvre and P. Nicolas. The First Version of a New ASP Solver: ASPeRiX. In E. Erdem, F. Lin, and T. Schaub, editors, *Proceedings of the 10th International Conference on Logic Programming an Nonmonotonic Reasoning (LPNMR 2009), Potsdam, Germany, September 14-18, 2009*, volume 5753 of *LNCS*, pages 522–527. Springer, 2009.
14. 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 (TOCL)*, 7(3), July 2006.

15 D. G. Mitchell. A SAT solver primer. *EATCS Bulletin (The Logic in Computer Science Column)*, 85:112–133, February 2005.

16 A. Palù, A. Dovier, E. Pontelli, and G. Rossi. Answer set programming with constraints using lazy grounding. In P. Hill and D. S. Warren, editors, *Proceedings of the 25th International Conference on Logic Programming (ICLP 2009)*, volume 5649 of *LNCS*, pages 115–129. Springer, 2009.

17 C. Redl. Development of a belief merging framewerk for dlvhex. Master's thesis, Vienna University of Technology, Institute of Information Systems, Knowledge-Based Systems Group, A-1040 Vienna, Karlsplatz 13, July 2010.

18 C. Redl. Merging of biomedical decision diagrams. Master's thesis, Vienna University of Technology, Institute of Information Systems, Knowledge-Based Systems Group, A-1040 Vienna, Karlsplatz 13, October 2010.

19 C. Redl, T. Eiter, and T. Krennwallner. Declarative Belief Set Merging using Merging Plans. In R. Rocha and J. Launchbury, editors, *13th International Symposium on Practical Aspects of Declarative Languages (PADL'11), Austin, Texas, U.S.A., January 24-25, 2011*, volume 6539 of *LNCS*, pages 99–114. Springer, January 2011.

20 G. Terracina, N. Leone, V. Lio, and C. Panetta. Experimenting with recursive queries in database and logic programming systems. *CoRR*, abs/0704.3157, 2007.

Together, Is Anything Possible? A Look at Collective Commitments for Agents

Ben Wright

Department of Computer Science, New Mexico State University
Las Cruces, New Mexico, USA
bwright@cs.nmsu.edu

Abstract

In this research, commitments – specifically collective commitments – are looked at as a way to model connections between agents in groups. Using the concepts and ideas from action languages, we propose to model these commitments as actions along with the other basic actions that autonomous agents are capable of performing. The languages developed will be tested against different examples from various multi-agent system (MAS) areas and implemented to run in answer set programming.

1998 ACM Subject Classification I.2.4 Knowledge Representation Formalisms and Methods, I.2.11 Distributed Artificial Intelligence

Keywords and phrases Reasoning About Knowledge, Action Languages, Commitments, Multi-agent systems, Modal Logic

Digital Object Identifier 10.4230/LIPIcs.ICLP.2012.476

1 Introduction and problem description

Nowadays, it is becoming more frequent to see research mentioning 'distributed this' or 'that network'. Artificial Intelligence (AI) and knowledge representation are no different. For decades now, multi-agent systems and distributed AI have been vastly popular areas of study.

This project looks to study the organization and representation of *groups* - which we use loosely as a term. Borrowing from many well known research areas, we believe that a simple and declarative approach using ***action languages*** and ***commitments*** will be the key to modeling many of the concepts involved in modeling certain groups.

We wish to represent this idea in both a *declarative* and *logical* way. By declarative, we mean to say in the sense that the rules and constraints of the group will be represented formally. When we say in a logical way, we mean a way in which the rules and how the group may come to *reason* are done using aspects of traditional logics. This also comes into play when we wish to define things the group may do either in a temporal sense, for instance: "The group may do something in the future.", or in an epistemic way, like "The group already knew that it was done." These two areas have very active research areas in ***temporal*** and ***epistemic logic***.

In order to test our system, we will use answer set programming as it models *declarative* and *logical* concepts easily. Once finished, we believe this type of model will be able to represent features in group for different areas such as negotiations, normative systems, joint action plans, multi-agent planning, and argumentation to name a few.

© Ben Wright;
licensed under Creative Commons License ND
Technical Communications of the 28th International Conference on Logic Programming (ICLP'12).
Editors: A. Dovier and V. Santos Costa; pp. 476–480

Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

2 Background and Overview of Existing Literature

2.1 Epistemic & Temporal Logics

Epistemic logic is focused on reasoning about the knowledge that agents may have about themselves, the world, or other agents. Two good references that define this area are [6] and [3]. Epistemic logic captures the essence of knowledge through modal operators. For instance $K_i p$ would read 'Agent i knows that the fluent p is true.' Operators are then defined on how knowledge may change or how knowledge can be *inferred* from other knowledge. Knowledge can also be nested, and this can be shown by a well known axiom rule called 'positive introspection' which is as follows: $K_i p \implies K_i K_i p$ which can be read as "If Agent i know p, then it knows that it knows p".

A big area of complexity and interest in this area is the idea of *common knowledge*. Common knowledge plays off of the idea "What does everyone know and knows that everyone knows?". From the previous example of positive introspection alone, we can see that this type of 'everyone knows' can become quite large in its representation.

Like epistemic logic, temporal logic is focusing on reasoning about truths that pertain to time. Many times in more 'real world' settings, fluents or values may change as the course of time or a program runs. Expressing these snippets of time may prove difficult if there isn't a mechanism to frame time in. There are two main ways to approach temporal logic: linear or branching. Linear temporal logic (LTL) treats time as a path along a string while branching temporal logic views time as a tree in which many different branches are possible, a well known system for this is Computation Tree Logic (CTL). There is actually a system known as CTL* which attempts to combine both of these ideas of representation together [5].

2.2 Action Languages

Representing what agents are capable of doing or how they may change their beliefs has always been a large area of research. One such idea for this representation is action languages. Action languages have been around for sometime now and have been shown as a way to set up *automated reasoning* or *planning* [9]. Various features have been implemented for agents using action languages like incomplete knowledge, reasoning about knowledge, and time.

Action languages work based on transitions from a state of fluents to another state of fluents. For instance, if "*move_left* **causes** *left*" is an action then when it occurs, $left$ would be true. Likewise, other features like static causal laws may show indirect effects like "*left* **if** $\neg$*right*" which would mean that "$left$ is only true when *right* is false" [7].

In addition *multi-agent* action languages (MALs) has been growing as well in which communication, coordination, and dispersion of fluentspaces/actionspaces all become issues of concern when developing. Some example MALs can be found in [12] and [1].

2.3 Commitments

Commitments have been studied for sometime in the MAS research area. Commitments formalize obligations or contracts between agents in a way that is logically representable. For instance if Agent A says that "I'll return the book to B within 2 days" then something like '$c(A, B, has_book(B), 0, 2)$.' can be used to represent this.

By using this idea of commitments, more complicated ideas can be formalized. For instance, protocols for agent communication can be formed like "If someone sends me a message, I will respond with an acknowledgement". With protocols, concepts such as *negotiation* or *argumentation* can be reasoned about.

In addition to this *building up* idea, commitments can also represent the tone or 'mood' of a system. Much like in *normative systems* where norms or social choice assign 'basic behaviors' to all agents much like "All cars stop at a red light".

When commitments are moved beyond the scope of agent to agent (that is one of the sides is a group) there is some debate as to how these occur. These group, or *collective*, commitments become difficult to represent. What does it mean for "If the doorbell rings, someone in the door will answer it."? Will everyone in the house answer the door? Will everyone assume someone else answers the door? These are still open issues and there are some differences in how researchers approach them. For instance, some attach *intentionality* to commitments [4] while others do not [13].

3 Research Accomplishments, Goals, and Future

3.1 Goal

The goal of this research is to capture the behavior of groups effectively. To do this, we will use *collective commitments* and *action languages*. By using commitments and action languages, we will be able to represent the ideas in both a logical and declarative way that leads to simple transitions among state fluents.

An example of what we are trying to represent is the idea of having a group of merchants (suppose m_1, m_2, and m_3) and a group of consumers (suppose c_1, c_2, and c_3). With actions, we can define stuff similar to '*purchase_item* **causes** *item_bought*' for the consumers and '*offer_item* **causes** *item_known*'. Where *purchase_item* and *offer_item* are actions and *item_bought* and *item_known* are fluents for the consumers and merchants respectively.

If we now have a collective commitment by the merchants such as "If an item is offered, at least one of us has that item for sale" this can model an interesting behavior in a succinct way. Rather than stating "If m_1 offers an item, then either m_1 or m_2 or m_3 has said item in their store at the time of a purchase by a consumer" and then have similar rules for m_2 and m_3 as well.

In addition to this type of commitment, another form we are looking into are *epistemic commitments* which can represent things like "Consumers will only purchase items from merchants they believe are giving them a fair deal". In this example, 'fair deal' would have to be defined more explicitly. However the interesting parts of this commitment are twofold — first, it does not pertain to a specific agent, but a *group* of agents and second, it plays off of the *beliefs* that that agent holds.

By intertwining these areas together in a logical and declarative way with action languages and commitments, we feel that expressing the behaviors in groups of agents will be reduced greatly.

3.2 Current Status of the Research

The concept of collective commitments has already been approached in some research [4, 2]. However, these do not use the idea of *action languages*. [13] proposes an action language to model basic commitments, that is non-collective commitments. No implementation of the action language exists yet with commitments.

In addition to the concept of commitments, *beliefs* have also been researched for action languages. [8, 10, 11] all consider knowledge or belief in action languages. These however only focus on single agents. [1] proposes an action language that models Kripke Structures.

An implementation of *epistemic commitments* could not be found. Creating and implementation an action theory that allows for commitments and beliefs will constitute a large portion of this research.

Current progress on this research is attempting to implement base level commitment from action languages into an answer set program model. In use currently is the MAL $\mathcal{L}^{mt}$ defined in [13]. From this language we currently have delayed and reversible processes implemented. So ideas like "The payment should arrive in 3 to 5 days" and "I sent the payment 2 days ago, but wish to cancel it now" can be represented.

Connection these processes to simple commitments still needs to be done in addition to adding in belief systems. Following this, these ideas can be raised to the level of "groups".

3.3 Open Issues and Expected Achievements

Building up the system implementation to allow for more flexibility and group variance seems to exponentially grow the size of our stable models returned in ASP. As many of the smaller features in the logics and representation areas already belong in higher complexity classes, bringing it all together in a straightforward manner may prove difficult.

Also, consideration on the upper bound for the number of agents a system may have has not been considered in depth. With the aforementioned complexity issues, this will need to be considered, especially for the implementation part.

In addition, some of the features we wish to add —such as epistemic commitments— are still vague in what they actually mean or do. Some further research into specific case studies will need to be done.

At the end of this research, we expect to have a basic implementation of epistemic and group commitments using action theories and answer set programming. As these ideas bridge many different areas of MAS and Logic, we will use examples from these different areas to test our ideas and implementation. At this stage of research, the implementation will only be concerned with *satisfiability* of models, rather than *optimal* models.

References

1. Chitta Baral, Gregory Gelfond, Enrico Pontelli, and Tran Cao Son. Logic programming for finding models in the logics of knowledge and its applications: A case study. *TPLP*, 10(4-6):675–690, 2010.
2. Cristiano Castelfranchi. Commitments : From Individual Intentions to Groups and Organizations. In *Proceedings of the First International Conference on Multiagent Systems*, pages 41–48, 1995.
3. Hans van Ditmarsch, Wiebe van der Hoek, and Barteld Kooi. *Dynamic Epistemic Logic*. Springer Publishing Company, Incorporated, 1st edition, 2007.
4. Barbara Maria Dunin-Keplicz and Rineke Verbrugge. *Teamwork in Multi-Agent Systems: A Formal Approach*. Wiley Publishing, 1st edition, 2010.
5. E. Allen Emerson and Joseph Y. Halpern. "sometimes" and "not never" revisited: On branching versus linear time temporal logic. *J. ACM*, 33(1):151–178, January 1986.
6. Ronald Fagin, Joseph Y. Halpern, Yoram Moses, and Moshe Y. Vardi. *Reasoning About Knowledge*. MIT Press, 1995.
7. Michael Gelfond and Vladimir Lifschitz. Action languages. *Electronic Transactions on AI*, 3, 1998.
8. Aaron Hunter and James P Delgrande. An Action Description Language for Iterated Belief Change. In *IJAIC 2007*, pages 2498–2503, 2007.

9 Vladimir Lifschitz. Action languages, answer sets and planning. In *In The Logic Programming Paradigm: a 25-Year Perspective*, pages 357–373. Springer Verlag, 1999.

10 Jorge Lobo, Gisela Mendez, and Stuart R. Taylor. Adding Knowledge to the Action Description Language A. In *AAAI-97*, number 3, pages 454–459, 1997.

11 Marek Sergot and Robert Craven. The Deontic Component of Action Language n C +. In *DEON 2006*, pages 222–237, 2006.

12 Tran Cao Son, Enrico Pontelli, and Ngoc-Hieu Nguyen. Planning for multiagent using asp-prolog. In Jürgen Dix, Michael Fisher, and Peter Novák, editors, *Computational Logic in Multi-Agent Systems*, volume 6214 of *Lecture Notes in Computer Science*, pages 1–21. Springer Berlin / Heidelberg, 2010. 10.1007/978-3-642-16867-3_1.

13 Tran Cao Son, Enrico Pontelli, and Chiaki Sakama. Formalizing commitments using action languages. In Chiaki Sakama, Sebastian Sardiña, Wamberto Vasconcelos, and Michael Winikoff, editors, *DALT*, volume 7169 of *Lecture Notes in Computer Science*, pages 67–83. Springer, 2011.