Technical Communications of the 26th International Conference on Logic Programming

ICLP'10, July 16–19, 2010, Edinburgh, UK

Edited by

Manuel Hermenegildo

Torsten Schaub

Editors

Manuel Hermenegildo
School of Computer Science
Technical University of Madrid (UPM), Spain
herme@fi.upm.es

Torsten Schaub
Knowledge Processing and Information Systems
Institute for Computer Science
University of Potsdam, Germany
torsten@cs.uni-potsdam.de

ACM Classification 1998
D.1.6 Logic Programming, D.1.3 Concurrent Programming, D.2.7 Distribution, Maintenance, and Enhancement, D.3.2 Language Classifications , D.3.3 Language Constructs and Features, D.2.4 Software/Program Verification, F.1.1 Models of Computation, F.1.2 Modes of Computation, F.3.2 Semantics of Programming Languages, F.4.1 Mathematical Logic, F.4.3 Formal Languages, G.3 Probability and Statistics, H.2.4 Systems, I.2.2 Automatic Programming, I.2.3 Deduction and Theorem Proving, I.2.4 Knowledge Representation Formalisms and Methods, I.2.5 Programming Languages and Software, J.3 Life and Medical Sciences

ISBN 978-3-939897-17-0

Published online and open access by
Schloss Dagstuhl – Leibniz-Center for Informatics gGmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany.

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

Digital Object Identifier: 10.4230/LIPIcs.ICLP.2010.i

ISBN 978-3-939897-17-0 **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.

ISSN 1868-8969

www.dagstuhl.de/lipics

Contents

Technical Communications of the 26th International Conference on Logic Programming (ICLP'10).
Editors: M. Hermenegildo, T. Schaub; pp. v–viii
Leibniz International Proceedings in Informatics
Schloss Dagstuhl Publishing, Germany

Doctoral Consortium

Technical Communications of the International Conference on Logic Programming, 2010 (Edinburgh), pp. IX-X
http://www.floc-conference.org/ICLP-home.html

Conference Organization

Conference Chair

Veronica Dahl
(Simon Fraser University, Canada)

Programme Chairs

Manuel Hermenegildo
(IMDEA Software Institute and U. Politécnica de Madrid, Spain)
Torsten Schaub
(University of Potsdam, Germany)

Doctoral Consortium Chairs

Marcello Balduccini *(Kodak Research Labs, USA)*
Alessandro Dal Palù *(U. degli Studi di Parma, Italy)*

Prolog Programming Contest Chair

Tom Schrijvers *(K.U. Leuven, Belgium)*

Programme Committee

María Alpuente *(Technical U. of Valencia, Spain)*,
Pedro Cabalar *(Coruña University, Spain)*,
Manuel Carro *(Technical U. of Madrid, Spain)*,
Luc De Raedt *(K. U. Leuven, Belgium)*,
Marina De Vos *(University of Bath, UK)*,
James Delgrande *(Simon Fraser University, Canada)*,
Marc Denecker *(KU Leuven, Belgium)*,
Agostino Dovier *(University of Udine, Italy)*,
Esra Erdem *(Sabanci University, Istanbul, Turkey)*,
Wolfgang Faber *(University of Calabria, Italy)*,
Thom Fruehwirth *(University of Ulm, Germany)*,
Maurizio Gabbrielli *(University of Bologna, Italy)*,
John Gallagher *(Roskilde University, Denmark)*,
Samir Genaim *(Complutense University, Spain)*,
Haifeng Guo *(University of Nebraska at Omaha, USA)*,
Joxan Jaffar *(National U. of Singapore, Singapore)*,
Tomi Janhunen *(Helsinki U. of Technology, Finland)*,
Michael Leuschel *(U. of Duesseldorf, Germany)*,
Alan Mycroft *(U. of Cambridge, UK)*,
Gopalan Nadathur *(University of Minnesota, USA)*,
Lee Naish *(Melbourne University, Australia)*,
Enrico Pontelli *(New Mexico State University, USA)*,
Vitor Santos Costa *(University of Porto, Portugal)*,

Tom Schrijvers *(K.U. Leuven, Belgium)*,
Tran Cao Son *(New Mexico State University, USA)*,
Peter J. Stuckey *(Melbourne University, Australia)*,
Terrance Swift *(CENTRIA, Portugal)*,
Peter Szeredi *(Budapest U. of Tech. and E., Hungary)*,
Frank Valencia *(École Polytechnique, France)*,
Wim Vanhoof *(University of Namur, Belgium)*,
Kewen Wang *(Griffith University, Australia)*,
Stefan Woltran *(Vienna U. of Technology, Austria)*,
and Neng-Fa Zhou *(City University of New York, USA)*.

External Reviewers

Mario Alviano, David Baelde, Demis Ballis, Hariolf Betz, Stefano Bistarelli, Francesco Calimeri, Dario Campagna, Henning Christiansen, Raffaele Cipriano, Michael Codish, Marco Comini, Alvaro Cortés Calabuig, Céline Dandois, Minh Dao-Tran, Broes De Cat, Stef De Pooter, François Degrave, Bart Demoen, Inês Dutra, Ozan Erdem, Halit Erdogan, Marc Fontaine, Andrea Formisano, Andrew Gacek, Sarah Gaggl, Maria Garcia de la Banda, Miguel Gomez-Zamalloa, Gopal Gupta, Rémy Haemmerlé, Vlaeminck Hanne, Ángel Herranz, Stijn Heymans, Jacob Howe, Aaron Hunter, José Iborra, Jianmin Ji, Christophe Joubert, Angelika Kimmig, Zeynep Kiziltan, Thomas Krennwallner, Gergely Lukácsy, Michael Maher, Theofrastos Mantadelis, Jacopo Mauro, Wannes Meert, Maria Chiara Meo, Robert Mercer, José Morales, Sriraam Natarajan, Jorge Navas, Pascal Nicolas, Carlos Olarte, Carla Piazza, Alexandre Miguel Pinto, Daniel Plagge, Joerg Puehrer, Frank Raiser, M.J. Ramírez, Francesco Ricca, Fabrizio Riguzzi, Ricardo Rocha, Konstantinos Sagonas, Chiaki Sakama, Andrew Santosa, Beata Sarna-Starosta, Taisuke Sato, Peter Schachte, Maria I. Sessa, Fernando Silva, Mantas Simkus, Jon Sneyers, Zoltan Somogyi, Harald Sondergaard, Naoyuki Tamura, Paul Tarau, Ingo Thon, Alwen Tiu, Tansel Uras, Guy Van den Broeck, Martijn van Otterlo, Daniel Varro, Sven Verdoolaeghe, Zoltán Vámossy, Mark Wallace, Zhe Wang, David Warren, Herbert Wiklicky, Sebastian Will, Johan Wittocx, Dong Xu, Roland Yap, Jia-Huai You, Damiano Zanardini, Zhihu Zhang, Zsolt Zombori.

Technical Communications of the 26th International Conference on Logic Programming, Edinburgh, July, 2010
Editors: Manuel Hermenegildo, Torsten Schaub
LIPIcs - Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/LIPIcs.ICLP.2010.i

Technical Communications of the International Conference on Logic Programming, 2010 (Edinburgh), pp. XI-XIV
http://www.floc-conference.org/ICLP-home.html

INTRODUCTION TO THE TECHNICAL COMMUNICATIONS OF THE 26TH INTERNATIONAL CONFERENCE ON LOGIC PROGRAMMING

MANUEL HERMENEGILDO [1] AND TORSTEN SCHAUB [2]

[1] IMDEA Software Institute and U. Politécnica de Madrid, Spain
E-mail address: manuel.hermenegildo@{imdea.org|upm.es}

[2] University of Potsdam, Germany
E-mail address: torsten@cs.uni-potsdam.de

The Logic Programming (LP) community, through the Association for Logic Programming (ALP) and its Executive Committee, decided to introduce for 2010 important changes in the way the main yearly results in LP and related areas are published. Whereas such results have appeared to date in standalone volumes of proceedings of the yearly International Conferences on Logic Programming (ICLP), and this method –fully in the tradition of Computer Science (CS)– has served the community well, it was felt that an effort needed to be made to achieve a higher level of compatibility with the publishing mechanisms of other fields outside CS.

In order to achieve this goal without giving up the traditional CS conference format a different model has been adopted starting in 2010 in which the yearly ICLP call for submissions takes the form of a joint call for a) *full papers* to be considered for publication in a special issue of the journal, and b) shorter *technical communications* to be considered for publication in a separate, standalone volume, with both kinds of papers being presented by their authors at the conference. Together, the journal special issue and the volume of short technical communications constitute the *proceedings* of ICLP.

The journal proceedings of the *26th International Conference on Logic Programming* are the first of a series of yearly special issues of Theory and Practice of Logic Programming (TPLP) putting this new model into practice. It contains the papers accepted from those submitted as full papers (i.e., for TPLP) in the joint ICLP call for 2010. The collection of technical communications for 2010 in hand appears in turn as Volume 7 of the Leibniz International Proceedings in Informatics (LIPIcs) series, published on line through the Dagstuhl Research Online Publication Server (DROPS). Both sets of papers were presented by their authors at this 26th ICLP.

Papers describing original, previously unpublished research and not simultaneously submitted for publication elsewhere were solicited in all areas of logic programming including but not restricted to: *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), and *Applications* (Databases, Data Integration and Federation, Software Engineering, Natural Language Processing, Web and Semantic Web, Agents, Artificial Intelligence, Bioinformatics).

Technical Communications of the 26th International Conference on Logic Programming, Edinburgh, July, 2010
Editors: Manuel Hermenegildo, Torsten Schaub
LIPIcs - Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/LIPIcs.ICLP.2010.XI

Special categories were *application papers* (where the emphasis was on their impact on the application domain) and *system and tool papers* (where the emphasis was on the novelty, practicality, usability and general availability of the systems and tools described). In the *technical communications* the emphasis was on describing recent developments, new projects, and other materials not yet ready for publication as full papers. The length limit for full papers was set at 15 pages plus bibliography for full papers (approximately in line with the length of TPLP technical notes) and for technical communications at 10 pages total.

In response to the call for papers 104 abstracts were received, 81 of which remained finally as complete submissions. Of those, 69 were full papers submitted to the TPLP special issue track (21 of them applications or systems papers). The program chairs acting as guest editors organized the refereeing process with the help of the program committee and numerous external reviewers. Each paper was reviewed by at least three anonymous referees which provided full written evaluations. Competition was high and after the first round of refereeing only 25 full papers remained. Of these, 16 went through a full second round of refereeing with written referee reports. Finally, all 25 papers went through a final, copy-editing round. In the end the special issue contains 17 technical papers, 6 application papers, and 2 systems and tools papers. During the first phase of reviewing the papers submitted to the technical communications track were also reviewed by at least three anonymous referees providing full written evaluations. Also, a number of full paper submissions were moved during the reviewing process to the technical communications track. Finally, 22 papers were accepted as technical communications.

The list of the 25 accepted full papers, appearing in the special issue of TPLP, follows:

Regular Papers

Automated Termination Analysis for Logic Programs with Cut
Peter Schneider-Kamp, Jürgen Giesl, Thomas Stroeder, Alexander Serebrenik, René Thiemann

Transformations of Logic Programs on Infinite Lists
Alberto Pettorossi, Maurizio Proietti, Valerio Senni

Swapping Evaluation: A Memory-Scalable Solution for Answer-On-Demand Tabling
Pablo Chico de Guzmán, Manuel Carro Liñares, David S. Warren

Threads and Or-Parallelism Unified
Vítor Santos Costa, Inês Castro Dutra, Ricardo Rocha

CHR(PRISM)-based Probabilistic Logic Learning
Jon Sneyers, Wannes Meert, Joost Vennekens, Yoshitaka Kameya, Taisuke Sato

Inference with Constrained Hidden Markov Models in PRISM
Henning Christiansen, Christian Theil Have, Ole Torp Lassen, Matthieu Petit

A Translational Approach to Constraint Answer Set Solving
Christian Drescher, Toby Walsh

A Decidable Subclass of Finitary Programs
Sabrina Baselice, Piero Bonatti

We would like to thank very specially the members of the Program Committee and the external referees for their enthusiasm, hard work, and promptness, despite the higher load of the two rounds of refereeing plus the copy editing phase. The PC members were: María Alpuente, Pedro Cabalar, Manuel Carro, Luc De Raedt, Marina De Vos, James Delgrande, Marc Denecker, Agostino Dovier, Esra Erdem, Wolfgang Faber, Thom Fruehwirth, Maurizio Gabbrielli, John Gallagher, Samir Genaim, Haifeng Guo, Joxan Jaffar, Tomi Janhunen, Michael Leuschel, Alan Mycroft, Gopalan Nadathur, Lee Naish, Enrico Pontelli, Vitor Santos Costa, Tom Schrijvers, Tran Cao Son, Peter J. Stuckey, Terrance Swift, Peter Szeredi, Frank Valencia, Wim Vanhoof, Kewen Wang, Stefan Woltran, and Neng-Fa Zhou.

We would also like to thank David Basin, Francois Fages, Deepak Kapur, and Molham Aref for their invited talks and those that helped organize ICLP: Veronica Dahl (General Chair and Workshops Chair), Marcello Balduccini and Alessandro Dal Palù (Doctoral Consortium), and Tom Schrijvers (Prolog Programming Contest). ICLP'10 was held as part of the 2010 Federated Logic Conference, hosted by the School of Informatics at the U. of Edinburgh, Scotland. Support by the conference sponsors –EPSRC, NSF, Microsoft Research, Association for Symbolic Logic, Google, HP, Intel– is also gratefully acknowledged. We are also grateful to Andrei Voronkov for creating the EasyChair system.

Finally, we would like to thank very specially Ilkka Niemelä, editor in chief of Theory and Practice of Logic Programming, David Tranah, from Cambridge University Press, Marc Herbstritt, from LIPIcs, Leibniz Center for Informatics, all the members of the ALP Executive Committee, and the ALP community in general for having believed in and allowed us to put into practice this approach which we believe provides compatibility with the publishing mechanisms of other fields outside CS, without giving up the format and excitement of our conferences.

Manuel Hermenegildo and Torsten Schaub
Program Committee Chairs

Technical Communications of the International Conference on Logic Programming, 2010 (Edinburgh), pp. 1
http://www.floc-conference.org/ICLP-home.html

DATALOG FOR ENTERPRISE SOFTWARE: FROM INDUSTRIAL APPLICATIONS TO RESEARCH (INVITED TALK)

MOLHAM AREF

LogicBlox, Two Midtown Plaza, 1349 West Peachtree Street, N.W., Suite 1880, Atlanta, GA 30309
E-mail address: molham.aref@logicblox.com

LogicBlox is a platform for the rapid development of enterprise applications in the domains of decision automation, analytics, and planning. Although the LogicBlox platform embodies several components and technology decisions (e.g., an emphasis on software-as-a-service), the key substrate and glue is an implementation of the Datalog language. All application development on the LogicBlox platform is done declaratively in Datalog: The language is used to query large data sets, but also to develop web and desktop GUIs (with the help of pre-defined libraries), to interface with solvers, statistics tools, and optimizers for complex analytics solutions, and to express the overall business logic of the application. We believe that Datalog is at the sweet spot of the expressiveness/convenience trade-off. The language is high-level enough to allow fast development for increased productivity, and expressive enough to support implementing complex applications without a need to resort to imperative code.

The LogicBlox version of Datalog, LB-Datalog, is heavily influenced by several ideas and advanced Datalog extensions proposed by the research community. LB-Datalog is a Datalog with integrity constraints, state and incremental update, default values, higher-order predicates, existentially quantified head variables, constraint stratification, and more. Additionally, LogicBlox has active collaborations with several academic researchers who work on a variety of projects in nearly every aspect of LB-Datalog.

The goal of this talk is to present both the business case for Datalog and the fruitful interaction of research and industrial applications in the LogicBlox context.

Technical Communications of the 26th International Conference on Logic Programming, Edinburgh, July, 2010
Editors: Manuel Hermenegildo, Torsten Schaub
LIPIcs - Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/LIPIcs.ICLP.2010.1

Technical Communications of the International Conference on Logic Programming 2010 (Edinburgh), pp. 2–3
http://www.floc-conference.org/ICLP-home.html

A LOGICAL PARADIGM FOR SYSTEMS BIOLOGY (INVITED TALK)

FRANÇOIS FAGES

EPI Contraintes, INRIA Paris-Rocquencourt,
Domaine de Voluceau, 78150 Rocquencourt, France
E-mail address: Francois.Fages@inria.fr
URL: http://contraintes.inria.fr/

Biologists use diagrams to represent complex systems of interaction between molecular species. These graphical notations encompass two types of information: interactions (e.g. protein complexation, modification, binding to a gene, etc.) and regulations (of an interaction or a transcription). Based on these structures, mathematical models can be developed by equipping such molecular interaction networks with kinetic expressions leading to quantitative models of mainly two kinds: ordinary differential equations (ODE) for a continuous interpretation of the kinetics, and continuous-time Markov chains (CTMC) for a stochastic interpretation of the kinetics.

The Systems Biology Markup Language (SBML) [8] uses a syntax of reaction rules with kinetic expressions to define such reaction models in a precise way. Nowadays, an increasing collection of models of various biological processes is available in this format in model repositories, such as for instance www.biomodels.net [9], and an increasing collection of ODE simulation or analysis software platforms are now compatible with SBML.

Since 2002, we investigate the transposition of programming concepts and tools to the analysis of living processes at the cellular level. Our approach relies on a logical paradigm for systems biology which consists in making the following identifications:

$$\begin{array}{c} \textit{biological model} = \textit{quantitative state transition system} \\ \textit{biological properties} = \textit{temporal logic formulae} \\ \textit{biological validation} = \textit{model-checking} \\ \textit{model inference} = \textit{constraint solving} \end{array}$$

Our modelling software platform Biocham [7] (implemented in Prolog) is founded on this paradigm [6]. An SBML model can be interpreted in Biocham at three abstraction levels:

- the Boolean semantics (asynchronuous Boolean state transitions on the presence/absence of molecules),
- the continuous semantics (ODE on molecular concentration),
- the stochastic semantics (CTMC on numbers of molecules).

The Boolean semantics is the most abstract one, it can be used to analyse large interaction networks without known kinetics. These formal semantics have been related in the framework of abstract interpretation in [5], showing for instance that the Boolean semantics is an abstraction of the stochastic semantics, i.e. that the possible stochastic behaviors can be

1998 ACM Subject Classification: algorithm, theory,verification.
Key words and phrases: temporal logic, model-checking, systems biology, hybrid systems.

Technical Communications of the 26th International Conference on Logic Programming, Edinburgh, July, 2010
Editors: Manuel Hermenegildo, Torsten Schaub
LIPIcs - Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/LIPIcs.ICLP.2010.2

checked in the Boolean semantics, and that if a Boolean behavior is not possible, it cannot be achieved in the quantitative semantics for any kinetics.

The use of model-checking techniques developed in the last three decades for the analysis of circuits and programs is the most original feature of Biocham. The temporal logics used to formalize the properties of the behavior of the system are respectively the Computation Tree Logic (CTL) for the Boolean semantics, and a quantifier-free Linear Time Logic with constraints over the reals (LTL($\mathbb{R}$)) for the quantitative semantics.

Biocham has been used for querying large Boolean models of the cell cycle by symbolic model-checking [1], formalizing phenotypes in temporal logic [3], searching parameter values from temporal specification [10], measuring the robustness of a system w.r.t. temporal properties [11], and developping in this way quantitative models of cell signalling and cell cycle for cancer therapies [2].

For some time, an important limitation of this approach was due to the logical nature of temporal logic specifications and their Boolean interpretation by true or false. By generalizing model-checking techniques to temporal logic constraint solving [3, 4], a continuous degree of satisfaction could be defined for temporal logic formulae, opening the field of model-checking to optimization in high dimension.

We believe that this mixing of discrete logical and continuous dynamics, pioneered by constraint logic programming and hybrid systems, and illustrated here in systems biology, is a deep trend for the future in programming and verification.

References

[1] Nathalie Chabrier and François Fages. Symbolic model checking of biochemical networks. *CMSB'03: First WS on Computational Methods in Systems Biology*, LNCS, col. 2602, pages 149–162, Rovereto, Italy, March 2003. Springer-Verlag.

[2] Elisabetta De Maria, François Fages, and Sylvain Soliman. On coupling models using model-checking: Effects of irinotecan injections on the mammalian cell cycle. *CMSB'09: 7th Int'l.Conf. on Computational Methods in Systems Biology*, *LN in BioInformatics* Vol. 5688, pp. 142–157. Springer-Verlag, 2009.

[3] François Fages and Aurélien Rizk. On temporal logic constraint solving for the analysis of numerical data time series. *Theoretical Computer Science*, 408(1):55–65, November 2008.

[4] François Fages and Aurélien Rizk. From model-checking to temporal logic constraint solving. In *CP'2009* LNCS number 5732, pages 319–334. Springer-Verlag, September 2009.

[5] François Fages and Sylvain Soliman. Abstract interpretation and types for systems biology. *Theoretical Computer Science*, 403(1):52–70, 2008.

[6] François Fages and Sylvain Soliman. Formal cell biology in BIOCHAM. *8th Int. School on Formal Methods for the Design of Computer, Communication and Software Systems: Computational Systems Biology SFM'08*, LNCS Vol. 5016, pages 54–80, Bertinoro, Italy, February 2008. Springer-Verlag.

[7] François Fages, Sylvain Soliman, and Aurélien Rizk. *BIOCHAM v2.8 user's manual*. INRIA, 2009. `http://contraintes.inria.fr/BIOCHAM`.

[8] Michael Hucka et al. The systems biology markup language (SBML): A medium for representation and exchange of biochemical network models. *Bioinformatics*, 19(4):524–531, 2003.

[9] Nicolas le Novère, Benjamin Bornstein, Alexander Broicher, Mélanie Courtot, Marco Donizelli, Harish Dharuri, Lu Li, Herbert Sauro, Maria Schilstra, Bruce Shapiro, Jacky L. Snoep, and Michael Hucka. BioModels Database: a free, centralized database of curated, published, quantitative kinetic models of biochemical and cellular systems. *Nucleic Acid Research*, 1(34):D689–D691, January 2006.

[10] Aurélien Rizk, Grégory Batt, François Fages, and Sylvain Soliman. On a continuous degree of satisfaction of temporal logic formulae with applications to systems biology. *CMSB'08: 4th Int'l. Conf. on Computational Methods in Systems Biology*, LNCS Vol. 5307, pages 251–268. Springer-Verlag, 2008.

[11] Aurélien Rizk, Grégory Batt, François Fages, and Sylvain Soliman. A general computational method for robustness analysis with applications to synthetic gene networks. *Bioinformatics*, 12(25):il69–il78, June 2009.

Technical Communications of the International Conference on Logic Programming, 2010 (Edinburgh), pp. 4–13
http://www.floc-conference.org/ICLP-home.html

RUNTIME ADDITION OF INTEGRITY CONSTRAINTS IN AN ABDUCTIVE PROOF PROCEDURE

MARCO ALBERTI [1] AND MARCO GAVANELLI [2] AND EVELINA LAMMA [2]

[1] CENTRIA, DI-FCT, Universidade Nova de Lisboa, Portugal

[2] ENDIF, Università di Ferrara, Italy

Abstract. Abductive Logic Programming is a computationally founded representation of abductive reasoning. In most ALP frameworks, integrity constraints express domain-specific logical relationships that abductive answers are required to satisfy.

Integrity constraints are usually known *a priori*. However, in some applications (such as interactive abductive logic programming, multi-agent interactions, contracting) it makes sense to relax this assumption, in order to let the abductive reasoning start with incomplete knowledge of integrity constraints, and to continue without restarting when new integrity constraints become known.

In this paper, we propose a declarative semantics for abductive logic programming with addition of integrity constraints during the abductive reasoning process, an operational instantiation (with formal termination, soundness and completeness properties) and an implementation of such a framework based on the $\mathcal{S}$CIFF language and proof procedure.

1. Introduction

The philosopher Peirce divides the reasoning schemes of humans into three types: *deduction* (reasoning from causes to effects), *induction* (synthesizing new rules from examples) and *abduction* (making hypotheses on possible causes from known effects).

Abductive Logic Programming [Kak93] is a computational representation of abductive reasoning that lets one express relationships between effects and possible causes (by means of a logic program), as well as logical constraints over the hypotheses (integrity constraints). In ALP possible hypotheses are represented by special predicates (called *abducibles*) that are not defined, but can be hypothesized, as long as they satisfy the integrity constraints. A positive answer to a query posed to an ALP system will typically contain the set of abducibles that are hypothesized in order for the query to succeed. Such an answer is called *abductive answer* in the ALP literature.

Several instances of ALP have been proposed in the literature [Kak90, Fun97, Den98, Alf99, Wan00], which differ for the logic language (and in particular for the type of abducibles and of integrity constraints that can be expressed).

While in many applications integrity constraints are known at the beginning of the reasoning process, it is sometimes useful to relax this assumption.

For instance, the classical application field of abductive reasoning is the diagnosis. However, in a realistic setting, a doctor does not simply listen to the patient enumerating

Key words and phrases: abduction, semantics, interactive computation, proof procedure.

Technical Communications of the 26th International Conference on Logic Programming, Edinburgh, July, 2010
Editors: Manuel Hermenegildo, Torsten Schaub
LIPIcs - Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/LIPIcs.ICLP.2010.4

all his/her symptoms, but they have a bidirectional and multi-stage interaction: the doctor asks questions, and refines his/her diagnosis based on the answers of the patient. So, there is the need to add information dynamically, often in the form of rules, that can rule out unrealistic sets of explanations.

In multi-agent reasoning, agents that employ abductive reasoning could exchange integrity constraints by a communication process, and continue operating with the newly acquired integrity constraints. In contracting, two agents try to reach an agreement and each agent tries to reach its goals. For example, one agent may want to buy a car, and the other wants to sell it; the first tries to get a price as low as possible, while the second has the opposite aim, and they negotiate on the model, the optionals, etc. Of course, each agent is unwilling to send all of its own knowledge, because the other would exploit it to get favourable conditions: if the buyer knew all the constraints of the seller, it would be able to compute the minimum possible price for the seller, and then propose such price. On the other hand, it is quite natural to tell some of the constraints only when needed, in order to speedup the negotiation, and avoid lingering on small variations of a meaningless solution. For instance, in case the buyer asks for a seat for children, the seller could reply: *"Ok, but you cannot install a children seat if you have the airbag"*, and the client has to take into consideration this constraint, when making new proposals. On the other hand, there is no reason for the seller to state such knowledge immediately from the beginning, as it still does not know if the buyer is interested at all in children seats.

An abductive reasoner might seek additional integrity constraints (possibly available from public repositories), depending on its current computation; for example, the number of integrity constraints could be very vast (as if one has to take into consideration all the EU rules for contracts), so only those strictly needed should be downloaded. Moreover, depending on the current state of the derivation one may choose to download regulations from one server or another: suppose I am deciding whether to buy a good from a service in Italy or in Portugal; I may first try to get the best price, but then check if the regulations of that country allow me to do such transaction. I will download the regulations of such country, check if my transaction is allowed, and, if it is not, I will backtrack and take the second choice.

Integrity constraints can also be obtained at runtime by means of an automated computational process; for instance, by inductive reasoning. Recently, extensions of Inductive Logic Programming techniques (ILP for short), and the DPML algorithm in particular [Lam07b], have been proposed to learn integrity constraints from labelled traces (a database of events recording happened interactions or activities, or a database collecting events at run-time). The DPML target language is the $\mathcal{S}$CIFF abductive logic language [Alb08], and this inductive approach has been experimented in various contexts (business processes, among others; see [Che09, Lam07a]).

Such applications motivate an abductive logic programming framework where some of the integrity constraints are known in advance, and some are added to the abductive logic program during the computation.

In this paper we propose a declarative semantics for such an extension, and its implementation based on the $\mathcal{S}$CIFF abductive logic language [Alb08]. $\mathcal{S}$CIFF is implemented using Constraint Handling Rules [Frü98]; in particular, integrity constraints are mapped to CHR constraints. Thanks to the properties of CHR, adding a new constraint at runtime amounts to the single operation of *calling* the new constraint, i.e., it can be delegated to the CHR solver.

The paper is structured as follows. In Section 2, we propose a declarative semantics of ALPs with dynamic addition of integrity constraints based on the $\mathcal{S}$CIFF language, and we show that it exhibits properties of termination, soundness and completeness. In Section 3 we describe the CHR-based implementation. In Section 4 we show some experimental results. Discussion of related work and conclusions follow.

2. Runtime addition of integrity constraints in $\mathcal{S}$CIFF

In this section, we give a semantics for the runtime addition of integrity constraints for the $\mathcal{S}$CIFF abductive logic language; however, the definitions can be easily generalized for other abductive logic languages.

2.1. $\mathcal{S}$CIFF language

We first provide a brief introduction to the $\mathcal{S}$CIFF language. A complete definition is available in [Alb08].

$\mathcal{S}$CIFF is a Computational Logic language, whose predicates can be defined or abducibles, and can contain variables. Variables can be constrained as in Constraint Logic Programming [Jaf94a].

A $\mathcal{S}$CIFF program $\mathcal{P}$ is composed of

- a knowledge base $\mathcal{KB}$;
- a set $\mathcal{IC}_S$ of *static integrity constraints*.

A $\mathcal{S}$CIFF knowledge base is a set of clauses of the form: $Head \leftarrow Body$, where *Head* is an atom built on a defined predicate, and *body* is a conjunction of literals (built on defined predicates or abducibles) and CLP constraints.

In $\mathcal{S}$CIFF, integrity constraints have the form: $Body \rightarrow Head$, where *Body* is a conjunction of abducible atoms, defined atoms and constraints, and *Head* is a disjunction of conjunctions of abducible atoms and CLP constraints, or *false*.

$\mathcal{S}$CIFF computations are goal-directed. A $\mathcal{S}$CIFF *Goal* has the same syntax of the body of a clause in the knowledge base.

2.2. Declarative semantics

The declarative semantics for runtime addition of integrity constraints is given in terms of *abductive explanation* as follows.

Given a $\mathcal{S}$CIFF program $\mathcal{P} = \langle \mathcal{KB}, \mathcal{IC}_S \rangle$ and a *goal* $\mathcal{G}$, a pair $\langle \Delta, \theta \rangle$, where Δ is a set of abducibles and θ is a substitution, is an *abductive explanation* for $\mathcal{G}$ with additional integrity constraints $\mathcal{IC}_D$ iff

(1) $\mathcal{KB} \cup \Delta \models \mathcal{G}\theta$
(2) $\mathcal{KB} \cup \Delta \models \mathcal{IC}_S \cup \mathcal{IC}_D$

where the symbol $\models$ is interpreted, in $\mathcal{S}$CIFF, as in the 3-valued completion semantics [Kun87]. If such conditions hold, we write $\langle \mathcal{KB}, \mathcal{IC}_S \rangle \models^{\Delta}_{\mathcal{IC}_D} \mathcal{G}$.

Example 2.1.

$$\begin{array}{rcl} p(X) & \leftarrow & q(X,Y), a(Y) \\ q(X,Y) & \leftarrow & r(Y), d(Y) \\ r(2) & & \end{array} \tag{2.1}$$

$$a(X) \quad \rightarrow \quad b(X) \vee c(X) \tag{2.2}$$

Given the knowledge base in equation (2.1) and the integrity constraint in equation (2.2), where $a/1$, $b/1$, $c/1$, and $d/1$ are abducibles, two abductive explanations are possible for the query $p(1)$: $\{a(2), b(2), d(2)\}$ and $\{a(2), c(2), d(2)\}$.

However, with the additional integrity constraint

$$c(X), d(X) \rightarrow \mathit{false},$$

only $\{a(2), b(2), d(2)\}$ is an abductive explanation.

2.3. Operational semantics

The $\mathcal{S}$CIFF proof-procedure consists of a set of transitions that rewrite a node into one or more child nodes. It encloses the transitions of the IFF proof-procedure [Fun97], and extends it in various directions. A complete description of $\mathcal{S}$CIFF proof procedure is in [Alb08], with proofs of soundness, completeness, and termination.

Each node of the proof is a tuple $T \equiv \langle R, CS, PSIC, \Delta \rangle$, where R is the resolvent, CS is the CLP constraint store, PSIC is a set of implications (called *Partially Solved Integrity Constraints*) derived from propagation of integrity constraints, and Δ is the current set of abduced literals. The main transitions, inherited from the IFF are:

Unfolding: replaces a (non abducible) atom with its definitions;

Propagation: if an abduced atom $a(X)$ occurs in the condition of an IC (e.g., $a(Y) \rightarrow p$), the atom is removed from the condition (generating $X = Y \rightarrow p$);

Case Analysis: given an implication containing an equality in the condition (e.g., $X = Y \rightarrow p$), generates two children in logical or (in the example, either $X = Y$ and p, or $X \neq Y$);

Equality rewriting: rewrites equalities as in the Clark's equality theory;

Logical simplifications: other simplifications like (true $\rightarrow A$) $\Leftrightarrow A$, etc.

$\mathcal{S}$CIFF also includes the transitions of CLP [Jaf94a, Jaf94b] for constraint solving.

To manage the run-time addition of integrity constraints, we extend $\mathcal{S}$CIFF with an additional transition defined as follows, and we call the resulting proof procedure $\mathcal{S}\text{CIFF}_D$.

Add-IC: Given a node $T \equiv \langle R, CS, PSIC, \Delta \rangle$ and an integrity constraint ic, transition *addIC* generates one node $T' \equiv \langle R, CS, PSIC \cup \{ic\}, \Delta \rangle$.

This transition picks integrity constraints from a queue of dynamic integrity constraints. The transition is applicable to any node in the proof tree, and it can be executed whenever the queue is not empty. More integrity constraints can be added to the queue during the computation.

A successful $\mathcal{S}\text{CIFF}_D$ derivation for an ALP $\langle \mathcal{KB}, \mathcal{IC}_S \rangle$, with additional integrity constraints $\mathcal{IC}_D$ and a goal $\mathcal{G}$ is a sequence of nodes where

- the root node is $\langle \mathcal{G}, \emptyset, \mathcal{IC}_S, \emptyset \rangle$
- each node is generated from the previous by a $\mathcal{S}\text{CIFF}_D$ transition
- the leaf node is $N \equiv \langle \text{true}, CS, PSIC, \Delta \rangle$

From the leaf node, a substitution θ is derived, that

- replaces all variables in N that are not universally quantified by a ground term;
- satisfies all the constraints in the store CS and the implications in $PSIC$.

If such a derivation exists, we write $\langle \mathcal{KB}, \mathcal{IC}_S \rangle \vdash_{\mathcal{IC}_D}^{\langle \Delta, \theta \rangle} \mathcal{G}$.

2.4. Properties

In this section, we state some relevant $\mathcal{S}$CIFF$_D$ properties. Due to lack of space, we omit the proofs, available in a companion technical report [Alb10].

Intuitively, $\mathcal{S}$CIFF$_D$ properties can be derived from $\mathcal{S}$CIFF properties, by showing that a $\mathcal{S}$CIFF$_D$ derivation for the program $\langle \mathcal{KB}, \mathcal{IC}_S \rangle$ with a finite set of additional integrity constraints $\mathcal{IC}_D$ can be transformed into an equivalent one, where a node is the root node of a $\mathcal{S}$CIFF derivation for the ALP $\langle \mathcal{KB}, \mathcal{IC}_S \cup \mathcal{IC}_D \rangle$.

The following proofs are based on these formal properties:

Proposition 2.2. *Let N_2 be the node generated from node N_1 by transition T_1, and N_3 be the node generated from node N_2 by* addIC*. Then, if N_4 is the node generated from node N_1 by* addIC*, transition T_1 is applicable to N_4, and the node N_5 generated from N_4 by T_1 is equal to N_3, modulo renaming of variables.*

$$N_1 \xrightarrow{T_1} N_2 \xrightarrow{\text{addIC}} N_3$$
$$N_1 \xrightarrow{\text{addIC}} N_4 \xrightarrow{T_1} N_5$$

Proposition 2.3. *Let D be a $\mathcal{S}$CIFF$_D$ derivation that has k applications of the* addIC *transition. Then there exists a derivation D' that has the following properties:*

- *the first k transitions of D' are* addIC*;*
- *each node of D', starting the transitions from $k+1$ is equal to the corresponding node of D.*

2.4.1. *Termination.* Being $\mathcal{S}$CIFF based on the 3-valued completion semantics, its termination is proven, as for SLDNF resolution [Apt91], for acyclic knowledge bases and bounded goals and implications. Of course, programs may also terminate in other cases as well. Other abductive proof-procedures are based on other semantics and can address also non-stratified programs [Lop06].

Intuitively, for SLD resolution a level mapping must be defined, such that the head of each clause has a higher level than the body. For $\mathcal{S}$CIFF, as well as for the IFF, since it contains integrity constraints that are propagated forward, the level mapping should also map atoms in the body of an integrity constraint to higher levels than the atoms in the head; moreover, this should also hold considering possible unfoldings of literals in the body of an integrity constraint [Xan03].

Termination is not affected in $\mathcal{S}$CIFF$_D$, as long as the newly added integrity constraints do not violate the termination conditions.

Proposition 2.4. *Let $\mathcal{G}$ be a query to an ALP $\langle \mathcal{KB}, \mathcal{IC}_S \rangle$, with additional integrity constraints $\mathcal{IC}_D$, where KB_S, $\mathcal{IC}_S \cup \mathcal{IC}_D$ and $\mathcal{G}$ are acyclic w.r.t. some level mapping, and $\mathcal{G}$ and all implications in $\mathcal{IC}_S \cup \mathcal{IC}_D$ are bounded w.r.t. the level-mapping. Then, every $\mathcal{S}$CIFF$_D$ derivation for each instance of $\mathcal{G}$ is finite.*

2.4.2. *Soundness.* As usual, the soundness property states that the abductive answer computed in a successful derivation is correct according to the declarative semantics.

Proposition 2.5. *Given an ALP* $\langle \mathcal{KB}, \mathcal{IC}_S \rangle$*, if*
$\langle \mathcal{KB}, \mathcal{IC}_S \rangle \vdash^{\langle \Delta, \theta \rangle}_{\mathcal{IC}_D} \mathcal{G}$
then
$\langle \mathcal{KB}, \mathcal{IC}_S \rangle \models^{\Delta}_{\mathcal{IC}_D} \mathcal{G}\theta$

2.4.3. *Completeness.* The completeness result states that $\mathcal{S}$CIFF$_D$ can compute a subset of any ground abductive answer that is correct according to the declarative semantics.

Proposition 2.6. *Given an ALP* $\langle \mathcal{KB}, \mathcal{IC}_S \rangle$ *and a set* $\mathcal{IC}_D$ *of integrity constraints, for any ground set* Δ *such that* $\langle \mathcal{KB}, \mathcal{IC}_S \rangle \models^{\Delta}_{\mathcal{IC}_D} \mathcal{G}$ *there exist* Δ' *and* θ *such that* $\langle \mathcal{KB}, \mathcal{IC}_S \rangle \vdash^{\langle \Delta', \theta \rangle}_{\mathcal{IC}_D} \mathcal{G}$ *and* $\Delta'\theta \subseteq \Delta$.

3. Implementation

The $\mathcal{S}$CIFF abductive proof procedure was implemented in Prolog, using extensively the Constraint Handling Rules [Frü98, Sch04] library. The implementation can be downloaded from the $\mathcal{S}$CIFF web site [SCI10] and runs on SICStus and SWI Prolog.

Constraint Handling Rules (CHR) is a logic language devoted to define new constraint solvers; however, it has been used as a general language for many different applications, not all strictly related to constraints.

A new solver is defined in CHR by means of rules. There exist two main types of rules: propagation and simplification[1]. A propagation rule is of the form

$$label@ \qquad Head_1, \ldots, Head_n \Rightarrow Guard | Body$$

and means that, if the optional *Guard* and the *Head*s are true, then the *Body* must be true. Operationally, whenever a set of constraints are in the store, matching $Head_1, \ldots, Head_n$, the *Guard* is checked; if it evaluates to *true*, the *Body* is executed (as a Prolog goal). The *label* is optional and serves only as an identifier of the rule.

Simplification rules have a similar syntax:

$$label@ \qquad Head_1, \ldots, Head_n \Leftrightarrow Guard | Body$$

and they state that if the *Guard* is true, then the conjunction $Head_1, \ldots, Head_n$ is equivalent to *Body*. Operationally, if $Head_1, \ldots, Head_n$ are in the store (and *Guard* is true), they are removed and substituted by *Body*.

$\mathcal{S}$CIFF represents most of its data structures as CHR constraints:

- an abducible atom $a(X)$ is represented with the CHR constraint `abd(a(X))`
- a (partially solved) integrity constraint $a(Y), q(Y) \rightarrow p(Y) \vee c(Y)$ is represented as the CHR constraint

$$\texttt{psic(}\underbrace{\texttt{[abd(a(Y)),q(Y)]}}_{Body}\texttt{,}\underbrace{\texttt{(p(Y) ; abd(c(Y)))}}_{Head}\texttt{)}$$

  The *Head* can be any Prolog goal (it has the same syntax).

[1]There are also *simpagation* rules, that are not logically necessary, but are important for efficiency; we will not go into details for lack of space.

The proof tree is explored in a depth-first fashion, using the Prolog stack for this purpose. Transitions are implemented as CHR rules; for example, transition *Propagation* is implemented with the following propagation CHR:

```
propagation @
    abd(A1),
    psic([abd(A2)|More],Head)
    ==> psic([A1=A2|More],Head).
```

Case Analysis handles the equality in the body of a PSIC

```
case_analysis @
    psic([A=B|More],Head)
    ==> impose A=B
        psic(More,Head)
    ;   % Open choice point
        impose A and B do not unify
```

and the logical simplification $(true \rightarrow A) \Leftrightarrow A$ manages implications with empty body:

```
logic_simplification @ psic([],Head) <=> call(Head).
```

Thanks to this implementation, adding a new integrity constraint is just a matter of *calling* the corresponding CHR constraint: if we want to dynamically add the integrity constraint (2.2) we execute the goal:

```
psic( [abd(a(X))], (abd(b(X));abd(c(X))) ).
```

In this way, the newly added integrity constraint is automatically subject to all the applicable transitions. Consider rule `propagation`: whenever two constraints matching the rule head (e.g., `abd(a(1))` and `psic([a(X)],b(X))`) are present in the CHR constraint store, the rule is fired, it generates `psic([a(X)=a(1)],b(X))`, that triggers `case analysis`, which in its turn generates two child nodes:

- one where unification is imposed between the abducible in the CHR constraint store and the abducible in the partially solved integrity constraint, and a new partially solved integrity constraint is imposed, with the abducible removed from the body;
- one where disunification between the abducible in the CHR constraint store and the abducible in the partially solved integrity constraint is imposed.

In the previous example, `psic([a(X)=a(1)],b(X))` is rewritten in the first case as $X = 1$ and $b(X)$ is executed; in the second case by imposing the CLP constraint $X \neq 1$.

The relevant point, here, is that rule `propagation` is fired whenever both the constraints (the abducible and the `psic`) are in the CHR store, regardless of which one entered the store first. So, if a partially solved integrity constraint is added by *addIC*, and some abducible in its body is already in the store, propagation will occur, as if the partially solved integrity constraint had been in the constraint store from the beginning of the computation.

4. Experiments

To show the effectiveness of the approach, we tested a simple benchmark problem, that is a simplified version of a contracting scenario. One agent needs to interact with some web service, and choose one that is able to provide the expected reply. In this example,

the agent will tell message m and will expect n as reply. The agent knows the address of a series of web services, given as facts:

$$known\_service(http://web.address.one/folder1/policy.ruleml).$$
$$known\_service(http://web.address.two/folder2/policy.ruleml).$$

In order to find the right service, the agent executes the following goal, where *tell* is abducible:

$$known\_service(Addr), download\_ic(Addr), tell(me, S, m), not(tell(S, me, A), A \neq n)$$

meaning that it will non-deterministically choose a service, download its integrity constraints, and then tell message m; it will fail if it gets any reply that is not n.

We generated 25^2 services, each with one integrity constraint

$$tell(Client, s, letter_1) \rightarrow tell(s, Client, letter_2)$$

where $letter_1$ and $letter_2$ are substituted with a ground term corresponding to one of the 25 letters of the alphabet.

We tried the goal on a slow network (mobile phone) and it took 173.350s to find the right service. As a comparison, a solution that first downloads the IC of all possible services before starting the solution takes 319.005s.

5. Related work

Among the many works on abduction in CHR by Christiansen and colleagues [Abd00, Chr05b], we emphasize an inspiring position paper [Chr05a], in which preliminary experiments are shown with integrity constraints mapped to CHR rules. In that work, Christiansen points out that through meta-rules it is possible to dynamically add integrity constraints. Here we extend the idea within the $\mathcal{S}$CIFF framework, which gives us a set of properties deemed crucial in the computational logic community. The operational semantics of $\mathcal{S}$CIFF is not based on that of CHR, but on the sound and complete semantics of the IFF [Fun97]: this allowed us to prove those properties also for $\mathcal{S}$CIFF. In this paper, we extend these proofs for the dynamic addition of integrity constraints, reaching the objective pointed out by Christiansen, but with soundness and completeness results.

EVOLP [Alf02] is a language to define logic programs able to evolve. A special atom $assert(Rule)$ can occur in the head or in the body of clauses; in case the stable model semantics assigns value *true* to some of these literals, the clause *Rule* is added to the program. Our instance can be considered as an evolving abductive program, in which only integrity constraints (and not clauses in the $\mathcal{KB}$) can be added, and based on the three-valued completion semantics, instead of the stable model semantics. Our language also features CLP constraints and, as the general CLP framework [Jaf94a], it is parametric with respect to the specific sort. The proof procedure lets the user choose the associated solver, and two state-of-the-art solvers are available in the current implementation: CLP($\mathcal{R}$), on the real values, and CLP(FD), on finite domains. EVOLP is a component of the ACORDA prospective logic programming system [Lop06], which also integrates abductive reasoning and preferences, to support interactive abductive logic programming, among other applications.

We can also easily extend the language in order to incorporate dynamic integrity constraints in the body of clauses, or in queries. Operationally, whenever an integrity constraint is part of the resolvent, the *addIC* transition would be applied. However, the impact of such extension on termination must be studied in future work. With reference to nested,

dynamic ICs, and this extension of the SCIFF language, it is worth to mention that in the literature, a lot of work was devoted to the treatment of embedded implications (due to Miller, et al. see [Mil89, Hod94] and McCarty, see [McC88]) based on the logic of Higher-Order Hereditary Harrop Formulas, a fragment of Intuitionistic logic. In this logic, and the λ system implemented [Nad88], they allow arbitrary lambda terms with full higher-order unification, and extend the formula language with arbitrarily nested universal quantifiers and implications. In our case, we can add integrity constraints at runtime, rather than program clauses as they do. We can therefore support abductive reasoning in an extended set of constraints.

In CR-Prolog [Bal03], new (consistency-restoring) rules can be added dynamically, as a part of an agent's Observe-Think-Act loop; if some inconsistency is detected then these constraints can be considered, according to their preferences. The semantics of CR-Prolog programs is defined as a transformation into abductive logic programs, where each consistency-restore rule has an abducible associated with it, and holds (only) if such abducible is abduced. In our framework, dynamically added integrity constraints must be satisfied, independently of the abductive answer.

6. Conclusions

In this paper we proposed a declarative semantics for abductive logic programs where additional integrity constraints can be added at runtime, based on the $\mathcal{S}$CIFF language.

We described $\mathcal{S}\text{CIFF}_D$, an extension of the $\mathcal{S}$CIFF proof procedure that supports runtime addition of integrity constraints, and we proved formal results of termination, soundness, and completeness for $\mathcal{S}\text{CIFF}_D$.

Such an extension can support interesting applications such as interactive abductive logic programming and contracting in service-oriented architecture.

References

[Abd00] Slim Abdennadher and Henning Christiansen. An experimental CLP platform for integrity constraints and abduction. In Henrik Legind Larsen, Janusz Kacprzyk, Slawomir Zadrozny, Troels Andreasen, and Henning Christiansen (eds.), *FQAS*, pp. 141–152. Physica-Verlag, Heidelberg, 2000.

[Alb08] Marco Alberti, Federico Chesani, Marco Gavanelli, Evelina Lamma, Paola Mello, and Paolo Torroni. Verifiable agent interaction in abductive logic programming: the SCIFF framework. *ACM Transactions on Computational Logics*, 9(4), 2008.

[Alb10] Marco Alberti, Marco Gavanelli, and Evelina Lamma. Runtime addition of integrity constraints in SCIFF. Tech. Rep. cs-2010-01, Università degli Studi di Ferrara, Dipartimento di Ingegneria, 2010. Available at http://www.unife.it/dipartimento/ingegneria/informazione/informatica/rapporti-tecnici-1.

[Alf99] José Júlio Alferes, Luís Moniz Pereira, and Terrance Swift. Well-founded abduction via tabled dual programs. In D. De Schreye (ed.), *ICLP*, pp. 426–440. 1999.

[Alf02] José Júlio Alferes, Antonio Brogi, João Alexandre Leite, and Luís Moniz Pereira. Evolving logic programs. In Sergio Flesca, Sergio Greco, Nicola Leone, and Giovambattista Ianni (eds.), *JELIA*, *Lecture Notes in Computer Science*, vol. 2424, pp. 50–61. Springer, 2002.

[Apt91] Krzysztof R. Apt and Marc Bezem. Acyclic programs. *New Generation Computing*, 9(3/4):335–364, 1991.

[Bal03] Marcello Balduccini and Michael Gelfond. Logic programs with consistency-restoring rules. In *AAAI Spring 2003 Symposium*, pp. 9–18. 2003.

[Che09] Federico Chesani, Evelina Lamma, Paola Mello, Marco Montali, Fabrizio Riguzzi, and Sergio Storari. Exploiting inductive logic programming techniques for declarative process mining. *T. Petri Nets and Other Models of Concurrency*, 2:278–295, 2009.

[Chr05a] Henning Christiansen. Experiences and directions for abduction and induction using constraint handling rules. In *Workshop on abduction and induction AIAI'05*. Edinburgh, Scotland, 2005.

[Chr05b] Henning Christiansen and Verónica Dahl. HYPROLOG: A new logic programming language with assumptions and abduction. In Maurizio Gabbrielli and Gopal Gupta (eds.), *ICLP*, *Lecture Notes in Computer Science*, vol. 3668, pp. 159–173. Springer, 2005.

[Den98] Marc Denecker and Danny De Schreye. SLDNFA: An abductive procedure for abductive logic programs. *J. Log. Program.*, 34(2):111–167, 1998.

[Frü98] T. Frühwirth. Theory and practice of constraint handling rules. *Journal of Logic Programming*, 37(1-3):95–138, 1998.

[Fun97] T. H. Fung and R. A. Kowalski. The IFF proof procedure for abductive logic programming. *Journal of Logic Programming*, 33(2):151–165, 1997.

[Hod94] Joshua S. Hodas and Dale Miller. Logic programming in a fragment of intuitionistic linear logic. *Inf. Comput.*, 110(2):327–365, 1994.

[Jaf94a] J. Jaffar and M.J. Maher. Constraint logic programming: a survey. *Journal of Logic Programming*, 19-20:503–582, 1994.

[Jaf94b] Joxan Jaffar, Michael Maher, Kim Marriott, and Peter Stuckey. The semantics of constraint logic programs. *Journal of Logic Programming*, 1994.

[Kak90] A. C. Kakas and Paolo Mancarella. On the relation between Truth Maintenance and Abduction. In T. Fukumura (ed.), *Proceedings of the 1st Pacific Rim International Conference on Artificial Intelligence, PRICAI-90, Nagoya, Japan*, pp. 438–443. Ohmsha Ltd., 1990.

[Kak93] A. C. Kakas, R. A. Kowalski, and Francesca Toni. Abductive Logic Programming. *Journal of Logic and Computation*, 2(6):719–770, 1993.

[Kun87] Kenneth Kunen. Negation in logic programming. *J. Log. Program.*, 4(4):289–308, 1987.

[Lam07a] Evelina Lamma, Paola Mello, Marco Montali, Fabrizio Riguzzi, and Sergio Storari. Inducing declarative logic-based models from labeled traces. In Gustavo Alonso, Peter Dadam, and Michael Rosemann (eds.), *BPM*, *Lecture Notes in Computer Science*, vol. 4714, pp. 344–359. Springer, 2007.

[Lam07b] Evelina Lamma, Paola Mello, Fabrizio Riguzzi, and Sergio Storari. Applying inductive logic programming to process mining. In Hendrik Blockeel, Jan Ramon, Jude W. Shavlik, and Prasad Tadepalli (eds.), *ILP*, *Lecture Notes in Computer Science*, vol. 4894, pp. 132–146. Springer, 2007.

[Lop06] Gonçalo Lopes and Luís Moniz Pereira. Prospective programming with ACORDA. In *Empirically Successful Computerized Reasoning (ESCoR'06) workshop at The 3rd International Joint Conference on Automated Reasoning (IJCAR'06)*. Seattle, USA, 2006.

[McC88] L. Thorne McCarty. Clausal intuitionistic logic I - fixed-point semantics. *J. Log. Program.*, 5(1):1–31, 1988.

[Mil89] Dale Miller. A logical analysis of modules in logic programming. *J. Log. Program.*, 6(1&2):79–108, 1989.

[Nad88] Gopalan Nadathur and Dale Miller. An overview of lambda-prolog. In *ICLP/SLP*, pp. 810–827. 1988.

[Sch04] T. Schrijvers and B. Demoen. The K.U. Leuven CHR system: implementation and application. In T. Fruhwirth and M. Meister (eds.), *First Workshop on Constraint Handling Rules*. 2004.

[SCI10] The *SCIFF* abductive proof procedure, 2010. `http://lia.deis.unibo.it/research/sciff/`.

[Wan00] Kewen Wang. Argumentation-based abduction in disjunctive logic programming. *J. Log. Program.*, 45(1-3):105–141, 2000.

[Xan03] I. Xanthakos. *Semantic Integration of Information by Abduction*. Ph.D. thesis, Imperial College London, 2003. Available at `http://www.doc.ic.ac.uk/~ix98/PhD.zip`.

Technical Communications of the International Conference on Logic Programming, 2010 (Edinburgh), pp. 14–23
http://www.floc-conference.org/ICLP-home.html

LEARNING DOMAIN-SPECIFIC HEURISTICS FOR ANSWER SET SOLVERS

MARCELLO BALDUCCINI [1]

[1] Intelligent Systems, KRL
Eastman Kodak Company
Rochester, NY 14650-2102 USA
E-mail address: marcello.balduccini@gmail.com

ABSTRACT. In spite of the recent improvements in the performance of Answer Set Programming (ASP) solvers, when the search space is sufficiently large, it is still possible for the search algorithm to mistakenly focus on areas of the search space that contain no solutions or very few. When that happens, performance degrades substantially, even to the point that the solver may need to be terminated before returning an answer. This prospect is a concern when one is considering using such a solver in an industrial setting, where users typically expect consistent performance. To overcome this problem, in this paper we propose a technique that allows learning domain-specific heuristics for ASP solvers. The learning is done off-line, on representative instances from the target domain, and the learned heuristics are then used for choice-point selection. In our experiments, the introduction of domain-specific heuristics improved performance on hard instances by up to 3 orders of magnitude (and 2 on average), nearly completely eliminating the cases in which the solver had to be terminated because the wait for an answer had become unacceptable.

1. Introduction

In recent years, solvers for Answer Set Programming (ASP) [Gel91, Mar99] have become amazingly fast. Mostly, that is due to good heuristics that direct the search toward the most promising areas of the search space, and to learning algorithms that discover features of the search space on-the-fly (see e.g. [Geb07]). Unfortunately, when the search space is sufficiently large, it is still possible for the search algorithm to mistakenly focus on areas of the search space that contain no solutions or very few. When that happens, performance degrades substantially, even to the point that the solver may need to be terminated before returning an answer. This prospect is a concern when one is considering using such a solver in an industrial application, in which the solver will act as part of a black-box from which users typically expect consistent performance. It should be noted that the phenomenon of performance degradation is often due to the fact that the heuristics used in choice-point selection are general-purpose, and thus can be side-tracked by peculiar features of a given domain. To overcome this problem, in this paper we propose a technique that allows learning domain-specific heuristics for ASP solvers. The technique is mainly aimed at improving the efficiency of the computation of one answer set (as opposed to multiple answer sets of a program) of consistent programs, but could be extended further. The learning is done off-line, on representative instances from the target domain. In our experiments, the introduction of domain-specific heuristics improved performance on hard instances by up to 3 orders of magnitude (and 2 on average), nearly

1998 ACM Subject Classification: I.2.3, I.2.4, I.2.5.
Key words and phrases: answer set programming, solvers, domain-specific heuristics.

Technical Communications of the 26th International Conference on Logic Programming, Edinburgh, July, 2010
Editors: Manuel Hermenegildo, Torsten Schaub
LIPIcs - Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/LIPIcs.ICLP.2010.14

completely eliminating the situations in which the solver had to be terminated because the wait for an answer had become unacceptable.

This paper is organized as follows. In the next section we give some background on ASP. Next, we discuss the basic search algorithm used in most ASP solvers. Then, in Section 3, we present our technique for learning domain-specific heuristics. Experimental results are discussed in Section 4. Finally, in Section 5, we draw conclusions.

2. Answer Set Programming

Let us start by giving some background on ASP. We define the syntax of the language precisely, but only give the informal semantics of the language in order to save space. We refer the reader to [Gel91, Nie00] for a specification of the formal semantics. Let Σ be a signature containing constant, function and predicate symbols. Terms and atoms are formed as usual in first-order logic. A (basic) literal is either an atom a or its strong (also called classical or epistemic) negation $\neg a$. The set of literals formed from Σ is denoted by $lit(\Sigma)$. A *rule* is a statement of the form:

$$h_1 \ \vee \ \ldots \ \vee \ h_k \leftarrow l_1, \ldots, l_m, \text{not } l_{m+1}, \ldots, \text{not } l_n$$

where h_i's and l_i's are ground literals and *not* is the so-called *default negation*. The intuitive meaning of the rule is that a reasoner who believes $\{l_1, \ldots, l_m\}$ and has no reason to believe $\{l_{m+1}, \ldots, l_n\}$, has to believe one of h_i's. The part of the statement to the left of $\leftarrow$ is called *head*; the part to its right is called *body*. Symbol $\leftarrow$ can be omitted if no l_i's are specified. Often, rules of the form $h \leftarrow \text{not } h, l_1, \ldots, \text{not } l_n$ are abbreviated into $\leftarrow l_1, \ldots, \text{not } l_n$, and called *constraints*. The intuitive meaning of a constraint is that its body must not be satisfied. A rule containing variables is interpreted as the shorthand for the set of rules obtained by replacing the variables with all the possible ground terms (called *grounding* of the rule). A *program* is a pair $\langle \Sigma, \Pi \rangle$, where Σ is a signature and Π is a set of rules over Σ. We often denote programs just by the second element of the pair, and let the signature be defined implicitly. In that case, the signature of Π is denoted by $\Sigma(\Pi)$. Finally, an *answer set* (or *model*) of a program Π is one of the possible collections of its consequences under the answer set semantics. Notice that the semantics of ASP is defined in such a way that programs may have multiple answer sets, intuitively corresponding to alternative views of the specification given by the program. In that respect, the semantics of default negation allows for a simple way to encode choices. For example, the set of rules $\{p \leftarrow \text{not } q. \ q \leftarrow \text{not } p.\}$ intuitively states that either p or q hold, and the corresponding programs has two answer sets, $\{p\}$, $\{q\}$. Because a convenient representation of alternatives is often important in the formalization of knowledge, the language of ASP has been extended with *constraint literals* [Nie00], which are expressions of the form $m\{l_1, l_2, \ldots, l_k\}n$, where m, n are arithmetic expressions and l_i's are basic literals as defined above. A constraint literal is satisfied whenever the number of literals that hold from $\{l_1, \ldots, l_k\}$ is between m and n, inclusive. Using constraint literals, the choice between p and q, under some set of conditions Γ, can be compactly encoded by the rule $1\{p, q\}1 \leftarrow \Gamma$. A rule of this form is called *choice rule*. When solving sets of problems from a given domain of interest, ASP programs are often divided into a *domain description* and a *problem instance*. Intuitively, the domain description encodes a description of the problem domain and of the solutions, while each problem instance encodes a different problem from the domain. In this paper we will make the simplifying assumption (usually satisfied even in practical applications) that the signature of every problem instance of interest is contained in the signature of the domain description. Another notion that is important for practical purposes is that of *domain predicate*. Domain predicates are relations whose definition is given with rules following syntactic restrictions, in such a way that the definition

of the relation can be derived from the rules without performing a complete answer set computation for the containing program. Domain predicates are used by the grounding procedures in order to determine the ranges of the variables that occur in the program. The precise definition of the syntactic restrictions varies depending on the grounding procedure used. A commonly used definition is the one given in [Syr98].

3. Learning Domain-Specific Heuristics

The search algorithm used by most ASP solvers (e.g. SMODELS [Nie02], DLV [Cal02], CLASP [Geb07]) builds upon the DPLL procedure [Dav60, Dav62]. The basic algorithm for the computation of a single answer set, which we will later refer to as *standard algorithm*, is show in Figure 1. The term *extended literal*, used in the algorithm, identifies a literal l or the expression *not* l (intuitively meaning that l is known not to hold in the answer set, but its complement, $\bar{l}$, may or may not hold). Given an extended literal e, $not(e)$ denotes the expression *not* l if $e = l$ and it denotes l if $e =$ *not* l. The algorithm is based on the idea of growing a particular set of (ground) literals, often

```
function solve ( Π : Program, A : Set of Extended Literals )
    B := expand(Π, A);
    if (B is answer set of Π) then return B;
    if (B is not consistent or B is complete) then return ⊥;
    e := choose_literal(Π, B);
    B' := solve(Π, B ∪ {e});
    if (B' = ⊥) then B' := solve(Π, B ∪ {not(e)});
    return B';
```

Figure 1: Basic Search Algorithm for ASP

called partial answer set, until it is either shown to be an answer set of the program, or it becomes inconsistent. To achieve this, guesses have to be made as to which literals may be in the answer set.

It is not difficult to see how the choices made by `choose_literal` greatly influence the number of choice points picked by the algorithm, and ultimately its performance. In order to reduce the chances of $choose\_literal$ making "wrong" selections, modern solvers base literal selection on carefully designed heuristics. For example, in SMODELS the selection is roughly based on maximizing the number of consequences that can be derived after selecting the given extended literal [Nie02]. These techniques work well in a number of cases, but not always. In fact, particular features of the program can confuse the heuristics. When that happens in the early stage of the search process, the effect is often disastrous, causing the solver to fail to return an answer in an acceptable amount of time. Particularly frustrating is the fact that the efficiency of the heuristics may change largely in correspondence of small elaborations of the program in input. For example, the $choose\_literal$ heuristics may make good selections for one problem instance, while they may cause the search to take an unacceptable amount of time for a not-too-different problem instance.

One way to limit the effect of wrong selections by $choose\_literal$ is that of allowing the solver to learn about relevant conflicts at run-time. Once learned, the information about conflicts can be used for the early pruning of other branches of the search space (e.g. [Geb07]). Although this technique has proven to be extremely effective, it does not address directly the issue of $choose\_literal$ making wrong choices, but rather curbs the problem by making some of those choices impossible after learning has taken place, or by allowing to quickly backtrack after a wrong choice has been made.

Furthermore, because the learning occurs at run-time, during the initial phase of the computation in which learning has not yet occurred, $choose\_literal$ may once again affect efficiency negatively by taking the search process in the wrong direction.

A different, more straightforward, way of limiting the wrong selections made by $choose\_literal$ is to directly improve the choice-making algorithm. In this paper, we adopt the approach of learning domain-specific heuristics from a number of sample problems, and of using them for literal selection in a modified version of $choose\_literal$. This technique is suitable for situations in which one is interested in solving a number of problem instances from a given problem domain. Such situations are very common in the ASP community – see e.g. the Second Answer Set Programming Competition [Den09]. Moreover, this is particularly the case in industrial applications, where the application contains the domain description, and the user describes the instance using some interface (refer e.g. to [Bal06]), which then automatically encodes the problem instance.

Consider program P_1:

$$P_1 = \begin{cases} p \leftarrow \text{not } q. \quad q \leftarrow \text{not } p. \\ \\ r. \\ \leftarrow p, r. \\ \leftarrow q, \text{not } s. \\ \\ u(X) \leftarrow t(X), \text{not } v(X). \\ v(X) \leftarrow t(X), \text{not } u(X). \\ \\ t(0).\ t(1).\ \ldots\ t(1000). \end{cases}$$

The program can be viewed as consisting of a domain description and a problem instance: the first 7 rules constitute the former, while the definition of predicate t is the problem instance. A different problem instance might then define t as $\{t(5),\ t(6),\ t(7)\}$. In this case, it is obvious that a good strategy for the selection of the literals consists in first choosing among $\{p, \text{not } p, q, \text{not } q\}$ and only later (if necessary) considering the extended literals formed by u and v.

In general, the domain-specific heuristics for $choose\_literal$ will be learned – rather than manually specified – by analyzing the choices made by the standard solver $solve$ when solving representative problem instances from the domain. This approach is particularly useful in applications in which a number of problem instances from the same class of problems will have to be solved over time – for example, in the setting of an industrial application, or in a programming/solver competition in which benchmarking is involved – and computational power is available off-line to allow learning the domain-specific heuristics (e.g. before deploying the application, or before submitting the solver or solutions to a competition).

Let us now describe in more detail our technique for learning and using domain-specific heuristics. We start with the learning phase. First of all, the algorithm from Figure 1 is modified to maintain a record of the choice points, and to return the list of choice points together with the answer set, when one is found. The modified algorithm is shown in Figure 2. In the algorithm, the list of choice points is stored in variable S. Symbol $\circ$ represents concatenation. When $solvecp$ is initially invoked, S is the empty list.

Now we turn our attention to how the information collected by $solvecp$ is used to guide the domain-specific heuristics. Given the domain description M and a problem instance I that is to be used to learn the domain-specific heuristics, the *decision-sequence* of I (denoted by $d(I)$) is $\perp$ if $solvecp(I \cup M, \emptyset, \emptyset) = \perp$ and S if $solvecp(I \cup M, \emptyset, \emptyset) = \langle A, S \rangle$ for some A. From now on, given a decision-sequence d, we denote its nth element by d_n. Moreover, given an extended literal

```
function solvecp ( Π : Program, A : Set of Extended Literals, S : Ordered List of Extended Literals )
    B := expand(Π, A);
    if (B is answer set of Π) then return ⟨B, S⟩;
    if (B is not consistent or B is complete) then return ⊥;
    e := choose_literal(Π, B);
    ⟨B′, S′⟩ := solve(Π, B ∪ {e}, S ∘ e);
    if (B′ ≠ ⊥) then return ⟨B′, S′⟩;
    ⟨B′, S′⟩ := solve(Π, B ∪ {not(e)}, S ∘ not(e));
    return ⟨B′, S′⟩;
```

Figure 2: Search Algorithm for ASP with Explicit Tracking of Choice Points

e, $level(e, d)$ denotes the index i such that $d_i = e$ (e is guaranteed not to occur at more than one position by construction of the decision-sequence in $solvecp$). Intuitively, $level(e, d)$ represents the level in the decision tree at which e was selected. Notice that, by construction of the sequence of choice points in $solvecp$, if $d(I) \neq \bot$, then $d(I)$ only enumerates the choice points that led directly to the answer sets. All the choice points that did not lead directly to it, in the sense that they were later backtracked upon, are in fact discarded every time the algorithm backtracks.

In order to improve the efficiency of the learned heuristics, we divide the class of problem instances in subclasses, and associate with each problem instance I an expression σ denoting the subclass it belongs to. The intuition is that using subclasses allows to further tailor the literal selection heuristics to the peculiar features of the problem instances. For example, in a planning domain, σ might be the maximum length of the plan (often called $lasttime$ or $maxtime$ in ASP-based planning). The subclass of a problem instance I is denoted by $\sigma(I)$.

Let $\mathcal{I}$ denote the set of all problem instances that will be used for the learning of the domain-specific heuristics. Next, we specify a way to determine how many times an extended literal e was selected at a certain level of the decision-sequences for the problem instances in $\mathcal{I}$. More precisely, given a positive integer δ, called the *scaling factor*, and subclass σ, the *occurrence count* of an extended literal e w.r.t. a level l and set of instances $\mathcal{I}$ is

$$o_{\delta,\sigma}(e, l, \mathcal{I}) = |\ \{\ I \,|\, I \in \mathcal{I}\ \wedge\ \sigma(I) = \sigma\ \wedge\ d(I) \neq \bot \wedge \\ l - \delta/2 \leq index(e, d(I)) < l + \delta/2\ \}\ |.$$

The scaling factor δ allows taking into account all the occurrences of e at a level in the interval $[l - \delta/2, l + \delta/2)$. If $\delta = 1$, then only the occurrences of e with level equal to l are considered. Values of δ greater than 1 can be useful in those cases in which all or most permutations of a sub-sequence of choice points lead to an answer set.

Let now $E = \{e_1, e_2, \ldots, e_k\}$ be a set of extended literals, representing possible choice points at some level l of the decision tree. The *set of best choice points* among E is:

$$best_\delta(l, E, \sigma, \mathcal{I}) = \{e \,|\, e \in E\ \wedge\ \forall e' \in E\ \ o_{\delta,\sigma}(e, l, \mathcal{I}) \geq o_{\delta,\sigma}(e', l, \mathcal{I})\}.$$

Intuitively, $best_\delta(l, E, \sigma, \mathcal{I})$ returns the choice points that, if taken at level l, are most likely to lead to an answer set without backtracking, based on the information collected about the instances of subclass σ in $\mathcal{I}$. Algorithms for the computation of $best_\delta(l, E, \sigma, \mathcal{I})$ and $o_{\delta,\sigma}(e, l, \mathcal{I})$ are simple and are omitted to save space.

Function $best_\delta(l, E, \sigma, \mathcal{I})$ encodes the essence of the domain-specific heuristics. Algorithm $choose\_literal$ can now be extended to perform literal selection guided by the domain-specific heuristics. The modified algorithm, $choose\_literal\_dspec$, is shown in Figure 3. In

```
function choose_literal_dspec ( Π : Program,
                                σ : Problem Subclass,
                                A : Set of Extended Literals,
                                level : Integer /* Current Level in the Decision Tree */,
                                T : Set of Extended Literals,
                                I : Set of Instances,
                                δ : Integer /* Scaling Factor*/ )

    L := lit(Σ(Π));
    E := L ∪ {not l | l ∈ L};
    E' = ∅;
    for each e ∈ E
        if (e ∉ A ∧ not(e) ∉ A ∧ e ∉ T) then
            E' := E' ∪ {e};
        end if
    end for
    B := best_δ(level, E', σ, I);
    if (B ≠ ∅) then
        chosen := one_element_of(B);
    else
        chosen := choose_literal(Π, A);
    end if

    return chosen;
```

Figure 3: Function for Literal Selection with Domain-Specific Heuristics

$choose\_literal\_dspec$, argument T is the set of extended literals that have previously been selected by $choose\_literal\_dspec$. If $best_\delta(level, E', \sigma(I), \mathcal{I})$ is the empty set, then $choose\_literal\_dspec$ falls back to performing standard extended literal selection via $choose\_literal$. This is for instances in which the learned heuristics do not prescribe any extended literal for the current decision level, or in which all the extended literals that the learned heuristics prescribed have already been tried. Modifying the standard solver's algorithm in order to use the domain-specific heuristics for choice-point selection is rather straightforward. A simple version, which for the most part follows the well-known iterative version of the SMODELS algorithm, is shown in Figure 4.

4. Experimental Evaluation

In this section we discuss the experiments we ran in order to evaluate our technique for learning domain-specific heuristics and using them in computing answer sets. To ensure coverage of a wide variety of cases, we have tested our implementation on both abstract problems and on problems from industrial applications of ASP. Here we show the results of testing on the task of planning for the Reaction Control System of the Space Shuttle.

The system used in the experiments is LPARSE+SMODELS, which we modified to obtain implementations of algorithms $solvecp$ and $solve\_dspec$. One complication of the implementation process is due to the fact that LPARSE often introduces unnamed atoms during the grounding of rules containing constraint literals, where by unnamed atoms we mean atoms that do not occur in the original program, and that are assigned an identifier that is only meaningful in the context of the current computation. Dealing with unnamed atoms is problematic because, in order to be used in the learning of the domain heuristics, all atoms must be assigned identifiers that are meaningful throughout

```
function solve_dspec ( Π : Program,
                       σ : Problem Subclass,
                       ℐ : Set of Instances,
                       δ : Scaling Factor )
 var S : Stack of Sets of Extended Literals;
 var B,T : Set of Extended Literals;
 var terminate : Boolean;

     S := ∅; B := ∅; T := ∅;
     terminate := false;
     while (terminate = false)
           B := expand(Π, B);
           if (B is answer set of Π) then
                 terminate := true;
           else
                 if (B is not consistent or B is complete) then
                       if (S = ∅) then
                             B := ⊥;
                             terminate := true;
                       else
                             /* Backtrack */
                             B := top(S);
                             S := pop(S);
                       end if
                 else
                       /* Select a choice point */
                       e := choose_literal_dspec(Π, σ, B, level, T, ℐ, δ);
                       T := T ∪ {e};
                       S := push(B ∪ {not(e)}, S);
                       B := B ∪ {e};
                 end if
           end if
     end while
     return B;
```

Figure 4: Search Algorithm for ASP with Domain-Specific Heuristics for Choice-Point Selection

multiple computations (normally, the atoms' own string representation satisfies this requirement). We have thus developed a technique that uses pre-processing and post-processing for the execution of LPARSE to assign unnamed atoms identifiers satisfying this requirement. Space limitations prevent us from giving more details on this technique.

It should also be noted that we did not use CLASP for our experiments: although CLASP is based, like SMODELS, on the DPLL procedure, and thus technically viable for the implementation of our algorithms, such implementation is complicated by the fact that, in CLASP, literal selection is allowed to select special literals denoting the whole body of a rule. A further complication of the implementation is due to the use of clause learning in CLASP. Work is ongoing on implementing $solvecp$ and $solve\_dspec$ within this solver, and results will be discussed in a longer paper. In the rest of the discussion, we refer to the implementation of $solve\_dspec$ within SMODELS as DSPEC.

As described in e.g. [Nog03, Bal06], the RCS is the Shuttle's system that has primary responsibility for maneuvering the Shuttle while it is in space. It consists of fuel and oxidizer tanks, valves, and other plumbing needed to provide propellant to the maneuvering jets of the Shuttle. The RCS also

includes electronic circuitry, both to control the valves in the fuel lines and to prepare the jets to receive firing commands.

In order to configure the Shuttle for an orbital maneuver, the RCS must be configured by opening and closing appropriate valves. This is accomplished by either changing the position of the associated switches, or by issuing computer commands. In normal conditions, the procedures for the configuration of the RCS for a given maneuver are known in advance by the astronauts. However, if components of the RCS are faulty, then the standard procedures may not be applicable. Moreover, because of the amount of possible combinations of faults, it is impossible to prepare in advance a set of configuration procedures for faulty situations. In those cases, ground control needs to carefully examine the problem and manually come up with a configuration procedure. The system described in [Nog03, Bal06] uses a model of the RCS, as well as ASP-based reasoning algorithms, to provide ground control with a decision-support system that automatically generates configuration procedures for the RCS and that can be used when faulty components are present (incidentally, the system can also perform diagnostic reasoning [Bal06]).

A collection of problem instances from the domain of the RCS is publicly available, together with the ASP encoding of the model of the RCS.[1] The interested reader may refer to [Nog03] for a description of the instances. For our testing, we have selected a set of 425 instances from the collection, corresponding to the public instances with no electrical faults and 3, 8, and 10 mechanical faults respectively, for which a plan of length 6 or less (determined by parameter $lasttime$) was found in the experiments discussed in [Nog03, Bal06], and we have analyzed the performance of the solver on planning with maximum lengths ranging between 6 and 10.

The comparison between SMODELS and DSPEC was conducted as follows. First of all, for each instance we found one plan using SMODELS. Each computation was set up in such a way as to timeout after 6000 seconds, if no answer set had yet been found. Next, we generated the domain-specific heuristics. The set of instances used for learning consisted of all the instances for which our implementation of $solvecp$ found a solution in 50 seconds of less, while the remaining "hard instances" were used for the evaluation phase. The problem subclasses were defined by the pair $\langle lasttime, maneuver\rangle$, where $lasttime$ specifies the maximum plan length and $maneuver$ is the maneuver that the RCS must be configured for (in our experiments, using the maneuver in the subclass definition substantially improved the performance of the learned heuristics). Figure 5 shows the results of the comparison for the 58 hard instances with 8 mechanical faults and values of lasttime of 9 and 10. The results were obtained with $\delta = 1$. We believe the speedup obtained with the domain-specific heuristics is remarkable. First of all, out of 32 instances for which the standard solver timed out before finding a solution, in 28 cases the domain-specific heuristics allowed to find a solution within the time limit, and in some cases in under 10 seconds. The average speedup is 232.3, with a peak of 1253.1 for an instance for which SMODELS timed out[2], and a peak of 544.5 for an instance for which SMODELS did not time out. In 4 cases (out of 32) DSPEC performed worse than the standard solver. We believe that these outliers can be eliminated if more samples are made available for learning.

[1] The files are available from `http://www.krlab.cs.ttu.edu/Software/Download/`.

[2] The actual speedup could in fact be higher, since SMODELS timed out. As a test, we have let SMODELS run on some of these instances for over $60,000$ seconds (16 hours) without getting a solution.

8 Mechanical Faults

Lasttime/ Instance	SMODELS (sec)	DSPEC (sec)	Speedup (times)
9 / 025	6000	17.643	340.1
9 / 027	6000	9.597	625.2
9 / 038	125.244	8.616	14.5
9 / 044	1439.027	6.846	210.2
9 / 053	6000	13.599	441.2
9 / 059	85.151	551.806	0.2
9 / 074	6000	8.961	669.6
9 / 075	736.134	3.837	191.9
9 / 087	6000	6000	1.0
9 / 090	6000	14.111	425.2
9 / 093	2451.649	6.477	378.5
9 / 098	114.643	10.529	10.9
9 / 103	52.219	12.544	4.2
9 / 122	6000	4.788	1253.1
9 / 140	6000	11.493	522.1
9 / 165	6000	13.027	460.6
9 / 170	6000	6000	1.0
9 / 179	6000	14.304	419.5
9 / 184	6000	20.254	296.2
9 / 188	6000	6000	1.0
9 / 191	4829.019	8.869	544.5
9 / 199	437.379	7.144	61.2
10 / 013	94.623	21.663	4.4
10 / 022	6000	423.565	14.2
10 / 025	6000	2035.089	2.9
10 / 027	6000	10.248	585.5
10 / 032	2949.169	13.82	213.4
10 / 037	6000	12.218	491.1
10 / 044	6000	18.162	330.4

Lasttime/ Instance	SMODELS (sec)	DSPEC (sec)	Speedup (times)
10 / 050	72.596	12.521	5.8
10 / 053	1907.445	23.37	81.6
10 / 059	6000	15.163	395.7
10 / 061	266.024	7.756	34.3
10 / 070	519.583	16.343	31.8
10 / 074	6000	13.903	431.6
10 / 077	251.754	7.518	33.5
10 / 087	6000	24.962	240.4
10 / 088	3830.141	18.512	206.9
10 / 092	318.83	11.712	27.2
10 / 093	6000	494.85	12.1
10 / 096	789.351	13.787	57.3
10 / 103	6000	16.781	357.5
10 / 110	6000	255.421	23.5
10 / 113	264.419	6000	0.044
10 / 120	1983.466	20.254	97.9
10 / 140	64.451	6000	0.011
10 / 147	187.8	7.125	26.4
10 / 154	942.008	6000	0.157
10 / 165	6000	30.008	199.9
10 / 166	6000	820.789	7.3
10 / 177	6000	12.605	476.0
10 / 178	6000	6000	1.0
10 / 179	6000	16.74	358.4
10 / 188	5235.985	12.74	411.0
10 / 189	3773.981	11.765	320.8
10 / 190	6000	1010.51	5.9
10 / 194	6000	12.407	483.6
10 / 199	6000	9.452	634.8

Figure 5: Performance Comparison on the RCS Domain. Machine specs: Intel i7 CPU, 2.93GHz, 8GB RAM.

5. Conclusions

In this paper we have demonstrated how domain-specific heuristics for choice-point selection can be learned and used in ASP solvers. Our experimental evaluation has shown that domain-specific heuristics can give remarkable speedups, and allow to find answer sets that otherwise cannot be computed in a reasonable time. In the case of the RCS domain, a large number of the instances for which the standard solver timed out, could be solver in a matter of seconds using the domain-specific heuristics, with an average speedup of more than 2 orders of magnitude and peaks of more than 3. This is the type of consistent performance that makes a solver viable for industrial applications.

We believe that an appealing feature of our approach is that in principle it can be applied to any solver built around the DPLL procedure. Hence, it is technically possible to apply the same approach shown here to other ASP solvers, or even to, say, SAT solvers and constraint solvers. Work is ongoing on implementing our technique within CLASP.

As a final note, we would like to point out that the method used here to learn the domain-specific heuristics is a very simple instance of policy learning. It will be interesting to investigate how

more sophisticated techniques from reinforcement learning, but also from machine learning and data mining, can be applied to the learning of the domain-specific heuristics. We expect that doing so will allow to improve performance of the solvers even further.

References

[Bal06] Marcello Balduccini, Michael Gelfond, and Monica Nogueira. Answer Set Based Design of Knowledge Systems. *Annals of Mathematics and Artificial Intelligence*, 2006.

[Cal02] Francesco Calimeri, Tina Dell'Armi, Thomas Eiter, Wolfgang Faber, Georg Gottlob, Giovanbattista Ianni, Giuseppe Ielpa, Christoph Koch, Nicola Leone, Simona Perri, Gerard Pfeifer, and Axel Polleres. The DLV System. In Sergio Flesca and Giovanbattista Ianni (eds.), *Proceedings of the 8th European Conference on Artificial Intelligence (JELIA 2002)*. 2002.

[Dav60] Martin Davis and Hillary Putnam. A Computing Procedure for Quantification Theory. *Communications of the ACM*, 7:201–215, 1960.

[Dav62] Martin Davis, Geroge Logemann, and Donald Loveland. A Machine program for theorem proving. *Communications of the ACM*, 5(7):394–397, 1962.

[Den09] Marc Denecker, Joost Vennekens, Stephen Bond, Martin Gebser, and Miroslaw Truszczynski. The Second Answer Set Programming Competition. In *10th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR09)*, pp. 637–654. 2009.

[Geb07] Martin Gebser, B. Kaufmann, A. Neumann, and Torsten Schaub. Conflict-driven answer set solving. In Manuela M. Veloso (ed.), *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI'07)*, pp. 386–392. MIT Press, 2007.

[Gel91] Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9:365–385, 1991.

[Mar99] Victor W. Marek and Miroslaw Truszczynski. *The Logic Programming Paradigm: a 25-Year Perspective*, chap. Stable models and an alternative logic programming paradigm, pp. 375–398. Springer Verlag, Berlin, 1999.

[Nie00] Ilkka Niemela and Patrik Simons. *Logic-Based Artificial Intelligence*, chap. Extending the Smodels System with Cardinality and Weight Constraints, pp. 491–521. Kluwer Academic Publishers, 2000.

[Nie02] Ilkka Niemela, Patrik Simons, and Timo Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1–2):181–234, 2002.

[Nog03] Monica Nogueira. *Building Knowledge Systems in A-Prolog*. Ph.D. thesis, University of Texas at El Paso, 2003.

[Syr98] Tommi Syrjanen. Implementation of logical grounding for logic programs with stable model semantics. Tech. Rep. 18, Digital Systems Laboratory, Helsinki University of Technology, 1998.

Technical Communications of the International Conference on Logic Programming, 2010 (Edinburgh), pp. 24–33
http://www.floc-conference.org/ICLP-home.html

HEX PROGRAMS WITH ACTION ATOMS

SELEN BASOL [1] AND OZAN ERDEM [1] AND MICHAEL FINK [2] AND GIOVAMBATTISTA IANNI [3]

[1] Faculty of Engineering and Natural Sciences, Sabancı University, Istanbul 34956, Turkey
E-mail address: {selenbasol,ozanerdem}@su.sabanciuniv.edu

[2] Institut für Informationssysteme, TU-Wien, Favoritenstraße 9-11, 1040 Wien, Austria
E-mail address: fink@kr.tuwien.ac.at

[3] Dipartimento di Matematica, Univ. della Calabria, P.te P. Bucci, Cubo 30B, 87036 Rende, Italy
E-mail address: ianni@mat.unical.it

ABSTRACT. HEX programs were originally introduced as a general framework for extending declarative logic programming, under the stable model semantics, with the possibility of bidirectionally accessing external sources of knowledge and/or computation. The original framework, however, does not deal satisfactorily with stateful external environments: the possibility of predictably influencing external environments has thus not yet been considered explicitly. This paper lifts HEX programs to ACTHEX programs: ACTHEX programs introduce the notion of action atoms, which are associated to corresponding functions capable of actually changing the state of external environments. The execution of specific sequences of action atoms can be declaratively programmed. Furthermore, ACTHEX programs allow for selecting preferred actions, building on weights and corresponding cost functions. We introduce syntax and semantics of ACTHEX programs; ACTHEX programs can successfully be exploited as a general purpose language for the declarative implementation of executable specifications, which we illustrate by encodings of knowledge bases updates, action languages, and an agent programming language. A system capable of executing ACTHEX programs has been implemented and is publicly available.

1. Introduction

HEX programs [Eit05], were originally introduced as a general framework for extending declarative logic programming, under the stable model semantics, with the possibility of bidirectionally accessing external sources of knowledge and/or computation. For instance, a rule like

$$pointsTo(X, Y) \leftarrow \&hasHyperlink[X](Y), url(X).$$

1998 ACM Subject Classification: I.2.4 [**Knowledge Representation Formalisms and Methods**]: Representation languages; I.2.3 [**Deduction and Theorem Proving**]: Inference engines, Logic programming, Nonmonotonic reasoning and belief revision; F.4.1 [**Mathematical Logic**]: Computational logic.

Key words and phrases: Answer Set Programming, Logic programming interoperability, Action languages.

This work was partially supported by the Vienna Science and Technology Fund (WWTF) under grant ICT08-020, by the Italian Research Ministry (MIUR) under project INTERLINK II04CG8AGG, and by the Regione Calabria and the EU under POR Calabria FESR 2007-2013 (PIA project of DLVSYSTEM s.r.l.).

Technical Communications of the 26th International Conference on Logic Programming, Edinburgh, July, 2010
Editors: Manuel Hermenegildo, Torsten Schaub
LIPIcs - Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/LIPIcs.ICLP.2010.24

might be devised for obtaining pairs of URLs (X, Y), where X actually links Y on the Web, and $\&hasHyperlink$ is an *external predicate* construct.

The possibility of accessing multiple external sources of knowledge has no significant constraint in HEX programs: in particular, besides constant values, relational knowledge (predicate extensions) can flow from external sources to the logic program at hand and viceversa, and recursion involving external predicates is allowed under reasonable safety assumptions.

It has been illustrated how HEX-programs qualify themselves for actual implementation of action and/or planning languages. As an example, in [Eit05] it is shown how the so called *code call* construct of agent programs as defined in [Eit99] can be embedded in HEX-programs using the notion of external predicate.

As a further example, HEX-programs constitute a generalization of description logic programs as defined in [Eit08]: it is made possible to push additional, hypothetical assertions to an external description logic knowledge base L, and then subsequently query the augmented knowledge base L'. However, it is not possible to push persistent assertions to L: in fact, HEX-programs do not contemplate the possibility of changing the state of external sources. For instance, it can be desirable having a program fragment like

$$new(X) \vee old(X) \leftarrow \&addToFavorites[X], new(X).$$

where intuitively $\&addToFavorites[X]$ is *a)* true for all (and only) the values of X which do not appear in a given external list L of favorite URLs, and *b)* has the side effect of adding X to L if X is not already in L. However, one might wonder what the semantics of a program including the above fragment should be, noticing that $\&addToFavorites$ changes its outcome depending on its state (the list L). Hence, the sequence of state changes due to $\&addFavorites$ would be predictable only if the rule evaluation order in the logic program at hand is operationally specified and known by the programmers.

Updates on external environments changing their state are desired in a variety of contexts, mainly: *1)*, when the actual execution of a plan is expected: in this setting, a change in the environment the agent at hand is acting in is implicitly prescribed; also, the order of execution of plan actions and their effect must be predictable and, indeed, this is the general setting which logic-based action languages are devised to reason about [Gel93]; *2)* when an answer set solver is interfaced with other (stateful) applications: the latter usually elaborate on data depending on answer sets computed, which can be then subsequently exploited for synthetis of new logic programs and evaluation thereof.

In the former case, the logic programming community (and particularly, the nonmonotonic reasoning community), has devoted extensive research towards reasoning about actions and planning, but only a few works (see e.g. [Sub00]) considered the support for actual execution of agent actions explicitly. In the latter case, applications have been developed by the Answer Set Programming community usually resorting to handcrafted solutions, like ad hoc post-parsing of answer sets[1], or developing ad hoc libraries for invoking answer set solvers from other development environments (see, e.g. [Ric03, Pir08]).

Although HEX-programs interface well with external sources of knowledge, it turns out that some structural limitations prevent addressing the issue of having impact on external environments in a satisfactory way: first, external functions associated to external predicates are inherently stateless; second, but more importantly, HEX-programs are fully declarative:

[1]An extensive list of known applications of ASP can be found at
`http://www.kr.tuwien.ac.at/research/projects/WASP/showcase.html`

this implies that when writing an HEX program, it is not predictable whether and in which order an external function will be evaluated.

To this end, we lift HEX programs to ACTHEX programs. ACTHEX programs introduce the notion of *action predicate* and *action atom*. Differently from external predicates, action predicates have impact on external environments and might trigger state changes and side effects. Action predicates are associated to corresponding (executable) functions. The framework allows *a)* to express and infer a predictable order of execution for action atoms, *b)* to express soft (and hard) preferences among a set of possible action atoms, and *c)* to actually execute a set of action atoms according to a predictable schedule. It is worth remarking that ACTHEX programs do not represent an action language in a strict sense. The main goal of the language is *1)* to provide a complementary extension to logic programming over which existing action, planning and agent languages can be grounded, and *2)* to provide a tighter and semantically sound framework for interfacing logic programs with applications of arbitrary nature.

2. Syntax and Semantics

Intuitively, ACTHEX programs extend HEX programs allowing rules like

$$\#robot[move, D]\{b, T\}[2:1] \leftarrow direction(D), time(T).$$

the above can be seen as a rule for scheduling a movement of a given robot in direction D with execution order T. Action atoms are executed according to *execution schedules*. The latter in turn depend on answer sets, which in their generalized form, can contain action atoms. The order of execution within a schedule can be specified using a *precedence* attribute (which in the above rule is set by the variable T); also actions can be associated with weights and priority levels (the values 2 and 1 above, respectively). Action atoms allow to specify whether they have to be executed *bravely* (the b switch above), *cautiously* or *preferred cautiously*, respectively meaning that an action atom a can get executed if it appears in at least one, all, or all *best cost* answer sets. We give next the formal syntax and semantics of the language.

Syntax. Given a finite alphabet Σ, we denote as $\mathcal{C}$, $\mathcal{X}$, $\mathcal{G}$, and $\mathcal{A}$ mutually disjoint subsets of Σ^* whose elements are respectively called constant names, variable names, external predicate names, and action predicate names. Elements from $\mathcal{X}$ (resp., $\mathcal{C}$) are denoted with first letter in upper case (resp., lower case), while elements from $\mathcal{G}$ (resp., $\mathcal{A}$) are prefixed with "&" (resp. "#"). Note that names in $\mathcal{C}$ serve both as constant and predicate names.

Elements from $\mathcal{C} \cup \mathcal{X}$ are called *terms*. A *higher-order atom* (or *atom*) is a tuple $(Y_0, Y_1, \ldots Y_n)$, where $Y_0, Y_1, \ldots Y_n$ are terms; $n \geq 0$ is the arity of the atom. Intuitively, Y_0 is the predicate name, and we thus also use the more familiar notation $Y_0(Y_1 \ldots Y_n)$. The atom is ordinary, if Y_0 is a constant. For example, $(x, type, c)$, $node(X)$, and $D(a, b)$, are atoms; the first two are ordinary atoms. An external atom [Eit05] is of the form $\&g[Y_1, \ldots, Y_n](X_1, \ldots, X_m)$ where $Y_1, \ldots, Y_n$ and $X_1, \ldots, X_m$ are two lists of terms (called input and output lists, respectively), and $\&g \in \mathcal{G}$ is an external predicate name. We assume that $\&g$ has fixed lengths $in(\&g) = n$ and $out(\&g) = m$ for input and output lists, respectively. An action atom is of the form $\#g\,[Y_1, \ldots Y_n]\,\{o, r\}\,[w : l]$ where $Y_1, \ldots, Y_n$ is a list of terms (called input list), and $\#g$ is an action predicate name. We assume that $\#g$ has fixed length $in(\#g) = n$ for its input list. $o \in \{b, c, c_p\}$ is called the *action option*.

Depending on the value of o, the action atom is called *brave, cautious, preferred cautious*, respectively.

Optional attributes r, w and l range over positive integers and variables[2], and are called *action precedence*, *action weight* and *action level* respectively. For an action atom a, we denote by $pr(a), w(a)$, and $l(a)$ its precedence, weight, and level, respectively. Concerning the latter two, we remark that they are reminiscent of the corresponding attributes of so-called weak constraints, but refrain from further illustration for space reasons.

Example 2.1. The action atom $\#robot[move, left]\{b, 1\}$ may be devised for moving a robot to the left. Here, we have that $in(\#robot) = 2$. This atom features the option b executed with precedence 1, while weight and level information are not given.

A rule r is of the form $\alpha_1 \vee \ldots \vee \alpha_k \leftarrow \beta_1, \ldots, \beta_n, not\, \beta_{n+1}, \ldots, not\, \beta_m$, where $m, n, k \geq 0$, $m \geq n$, $\alpha_1, \ldots \alpha_k$ are atoms or action atoms, and $\beta_1, \ldots \beta_m$ are either atoms or external atoms. We define $H(r) = \{\alpha_1, \ldots, \alpha_k\}$ and $B(r) = B^+(r) \cup B^-(r)$, where $B^+(r) = \{\beta_1, \ldots, \beta_n\}$ and $B^-(r) = \{not\, \beta_{n+1}, \ldots, not\, \beta_m\}$. If $H(r) = \emptyset$ and $B(r) \neq \emptyset$, then r is a *constraint*, and if $B(r) = \emptyset$, and $H(r) \neq \emptyset$, then r is a *fact*; r is *ordinary*, if it does not contain external or action atoms. An ACTHEX *program* is a finite set P of rules. It is *ordinary*, if all rules are ordinary.

Example 2.2. The following is a valid ACTHEX program:

$$
\begin{aligned}
evening \vee morning.& \\
\#robot[turnAlarm, on]\{c, 2\} &\leftarrow evening. \\
\#robot[turnAlarm, off]\{c, 2\} &\leftarrow morning. \\
\#robot[move, all]\{b, 1\} &\leftarrow \&getFuel[](high). \\
\#robot[move, left]\{b, 1\} &\leftarrow \&getFuel[](low).
\end{aligned}
$$

Semantics. The semantics of ACTHEX programs generalizes that of HEX-programs given in [Eit05], which in turn generalizes traditional answer-set semantics [Gel91]. In the sequel, let P be an ACTHEX program. We will assume that P acts in a *external environment* E, over which action atoms potentially triggered by P might have some effects. ACTHEX programs can in practice be exploited in a variety of different environments (e.g. a relational database, a file system, or the entire Web): we focus here on the semantics of P, and thus we will make no particular assumption on the nature of E besides assuming it as a finite collection of data structures of unspecified nature and size (to take the most general view, assume E as a finite, arbitrarily large, portion of a Turing machine tape surrounded by blanks on both sides).

The *Herbrand base* of P, denoted HB_P, is the set of all possible ground versions of atoms, external atoms and action atoms occurring in P obtained by replacing variables with constants from $\mathcal{C}$. The grounding of a rule r, $grnd(r)$, is defined accordingly, and the grounding of program P is given by $grnd(P) = \bigcup_{r \in P} grnd(r)$. Unless specified otherwise, $\mathcal{C}, \mathcal{X}, \mathcal{G}$, and $\mathcal{A}$ are implicitly given by P.

Example 2.3. Given $\mathcal{C} = \{edge, arc, d, e, 1, 2\}$, some ground instances of $E(X, c)$ are $edge(d, e)$, $arc(arc, e)$; $\#robot[d, N]\{b, X\}$ has ground instances $\#robot\ [d, e]\{b, 1\}$, $\#robot\ [d, d]\{b, 2\}$.

[2]We assume here that $\mathcal{C}$ contains a finite subset of consecutive integers $S = \{0, \ldots, n_{max}\}$.

An *interpretation relative to* P is any subset $I \subseteq HB_P$ containing (ordinary) atoms and action atoms. We say that I is a *model* of atom (or action atom) $a \in HB_P$, denoted $I \models a$, if $a \in I$. With every external predicate name $\&g \in \mathcal{G}$, we associate an (n+m+1)-ary Boolean function $f_{\&g}$, assigning each tuple $(I, y_1, \ldots, y_n, x_1, , \ldots, x_m)$ either 0 or 1, where $n = in(\&g)$, $m = out(\&g)$, $I \subseteq HB_P$, and $x_i, y_j \in \mathcal{C}$. Similarly, with every action predicate name $\#g \in \mathcal{A}$, we associate a $(n+2)$-ary function $f_{\#g}$ with input $(E, I, y_1, \ldots, y_n)$ and returning a new external environment $E' = f_{\#g}(E, I, y_1, \ldots, y_n)$. Note that functions that are associated with action atoms do not have output lists. We say that $I \subseteq HB_P$ is a model of a ground external atom $a = \&g\,[y_1, \ldots, y_n]\,(x_1, \ldots, x_m)$, denoted $I \models a$, iff $f_{\&g}(I, y_1 \ldots, y_n, x_1, \ldots, x_m) = 1$.

Intuitively, functions associated with external atoms model (stateless) calls to external code and/or external sources of knowledge, as originally defined in [Eit05]. The newly introduced notion here is that of action predicates: action atoms can appear in answer sets or not depending on whether they are a consequence of the program at hand or not; functions associated with action predicates serve the purpose of modelling the actual execution of entailed action atoms, i.e., the respective changes on E.

Example 2.4. We associate with $\&reach$ a function $f_{\&reach}$, s.t. $f_{\&reach}(I, G, A, B) = 1$ iff node B is reachable from node A in the graph encoded by means of the binary predicate G . Let $I = \{e(b, c), e(c, d)\}$. Then, I is a model of $\&reach[e, b](d)$, since $f_{\&reach}(I, e, b, d) = 1$. Also, let us associate with $\#insert$ a function $f_{\#insert}$, and assume that E contains an encoding of a knowledge base K expressed as a set of facts. When action atom $\#insert\,[edge, arc]\,\{b, 1\}$ needs to be executed, then the function $f_{\#insert}$ is called with inputs $(E, I, edge, arc)$, for an interpretation I. Intuitively, $\#insert$ might correspond to the act of adding to the extension of the predicate $edge$ in K the extension of the predicate arc in I.

Let r be a ground rule. We define (i) $I \models H(r)$ iff there is some $a \in H(r)$ such that $I \models a$, (ii) $I \models B(r)$ iff $I \models a$ for all $a \in B^+(r)$ and $I \not\models a$ for all $a \in B^-(r)$, and (iii) $I \models r$ iff $I \models H(r)$ or $I \not\models B(r)$. We say that I is a *model* of an ACTHEX program P, denoted $I \models P$, iff $I \models r$ for all $r \in grnd(P)$. We call P $satisfiable$, if it has some model. Given an ACTHEX program P, the *FLP-reduct* of P with respect to $I \subseteq HB_P$, denoted fP^I , is the set of all $r \in grnd(P)$ such that $I \models B(r)$. $I \subseteq HB_P$ is an *answer set* of P iff I is a minimal model of fP^I.

Note that we inherit from the framework of HEX programs the adoption of the notion of reduct as defined by [Fab04] (referred to as *FLP-reduct* henceforth). The FLP-reduct is equivalent to the traditional Gelfond-Lifschitz reduct for ordinary programs, and in our context ensures answer-set minimality, even in the presence of external atoms (see [Eit05] for details). Let $\mathcal{AS}(P)$ be the collection of all the answer sets of program P; the set of *best models* $\mathcal{BM}(P)$ contains the answer sets of P minimizing an objective function H_P. $H_P(A)$ intuitively weighs an answer set A depending on the weights (and levels) of action atoms which are contained in A[3].

Let a be an action atom of the form $\#g\,[y_1, \ldots y_n]\,\{o, r\}$, and $A \in \mathcal{AS}(P)$; a is said to be *executable* in A, if *i)* a is brave (i.e., $o = b$) and $a \in A$, or *ii)* a is cautious (i.e., $o = c$) and $a \in B$ for every $B \in \mathcal{AS}(P)$, or *iii)* a is preferred cautious (i.e., $o = c_p$) and $a \in B$ for every $B \in \mathcal{BM}(P)$. Roughly speaking, once an answer set A is chosen as the one to be

[3]For space reasons, the reader can find the definition of H_P at
`http://www.kr.tuwien.ac.at/research/systems/dlvhex/actionplugin/preferences.html`

executed, action atoms to be executed are selected depending on their action option. Note that, in this respect, the notion of *brave executability* depends on the answer set at hand and thus slightly differs from the traditional notion of brave entailment.

Given an answer set $A \in \mathcal{BM}(P)$, an *execution schedule* $E_{A,P} = [a_1, \ldots, a_n]$ is an ordered list containing all the action atoms executable in A, such that $i < j$ if $pr(a_i) < pr(a_j)$, for each pair of atoms a_i, a_j appearing in $E_{A,P}$.

Intuitively, an execution schedule for a program gives an order for the action execution compatible with the precedences specified in the program. Note that for action atoms with the same precedence the execution order is not specified.

Given an execution schedule $E_{A,P} = [a_1, \ldots, a_n]$, let $E_0 = E$, and for $i > 0$, $E_i = f_{a_i}(E_{i-1}, A, y_1, \ldots, y_m)$. We define $EX(E_{A,P}) = E_n$ as the *execution outcome* of $E_{A,P}$, and $\mathcal{EX}(P) = \{E_{A,P} \mid A \in \mathcal{BM}(P)\}$.

In general, given a program P, we consider $\mathcal{AS}(P)$, $\mathcal{BM}(P)$ and $\mathcal{EX}(P)$ as different facets of the semantics of P. In particular, the *execution outcome* of P is $EX(E_{A,P})$ for an execution schedule $E_{A,P} \in \mathcal{EX}(P)$ of choice. We simply assume that a deterministic rule for choosing $E_{A,P}$ is given[4].

Example 2.5. Let A_1, A_2, A_3 be three answer sets of a given program $P_{ex2.5}$, where A_1, $A_2 \in \mathcal{BM}(P_{ex2.5})$. Let $a_1 = \#insert\,[e, g_1]\,\{b, 1\}$, $a_2 = \#insert\,[e, g_2]\,\{c, 5\}$, $a_3 = \#insert\,[e, g_3]\,\{c, 2\}$, $a_4 = \#insert\,[e, g_4]\,\{c_p, 2\}$, $a_5 = \#insert\,[e, g_5]\,\{b, 1\}$, and let $A_1 = \{a_1, a_2, a_3, a_4, a_5\}$, $A_2 = \{a_2, a_4\}$, $A_3 = \{a_2, a_5\}$.

Since $A_3 \notin \mathcal{BM}(P_{ex5})$, possible choices of answer sets are A_1 and A_2. If we choose A_1, brave atoms a_1, a_5, cautious atom a_2 and preferred cautious atom a_4 are executable since $a_1, a_5 \in A_1$, where a_2 appears in all the answer sets and a_4 appears in both A_1 and A_2 . A_1 has two possible execution schedules which are $[a_1, a_5, a_4, a_2]$, and $[a_5, a_1, a_4, a_2]$.

For the case that A_2 is selected, cautious atom a_2 and preferred cautious atom a_4 are executable since a_2 appears in all answer sets, and a_4 appears in A_1 and A_2. Thus, the only possible execution schedule for A_2 is $[a_4, a_2]$.

3. Applications of ACTHEX programs

In this section, we provide evidence for the versatility of ACTHEX by discussing several application scenarios, including encodings of existing action-based KR formalisms.

Action languages. We use action language $\mathcal{C}$ [Giu98] as a representative for sketching how action languages can be reduced to ACTHEX programs. The relationship to logic programming is well-known: we follow a transformation from [Lif99].

The semantics of $\mathcal{C}$ is defined in terms of transition diagrams which put in relationship propositional *action* and *fluent* atoms. The possible state evolution specified in transition diagrams can equivalently be characterized as a logic program expressed in terms of predicates having a time attribute, which are used for encoding truth values of different action and fluent variables at different times. Not surprisingly, the precedence attribute of action atoms can intuitively capture the notion of time as in [Lif99].

Consider causal laws defined as either a *static law* of the form "**caused** F **if** $L_1 \wedge \cdots \wedge L_m$", or a *dynamic law* of the form "**caused** F **if** $L_1 \wedge \cdots \wedge L_m$ **after** $L_{m+1} \wedge \cdots L_n \wedge L_{n+1} \wedge$

[4]For the sake of efficiency, our implementation executes the first execution schedule obtained from the first computed answer set: other selection criteria are of course possible.

$\cdots \wedge L_k$", where F is a fluent literal, L_i is a fluent literal for $1 \leq i \leq n$, and respectively an action name for $n+1 \leq i \leq k$. An *action description* is a set of causal laws.

Given an action description D and a *maximum time* t, following [Lif99], a dynamic law $l \in D$ of the form above can be translated to the ordinary rule $F'(T+1) \leftarrow \ not\ \overline{L}'_1(T+1),\ \ldots\ , not\ \overline{L}'_m(T+1), L'_{m+1}(T), \ldots, L'_k(T)$, where F' is a unary predicate associated to fluent F, while L'_i, $\overline{L}'_i$ are unary predicates associated to fluents L_i, $1 \leq i \leq n$, respectively to actions L_i, $n+1 \leq i \leq k$, and their complements[5]. We then put in connection action atoms with actions by means of rules $\#L_i\{o,T\} \leftarrow L_i(T).$, $n+1 \leq i \leq k$, where $\#L_i$ is a newly introduced action atom which is responsible of executing the action L_i, and o is an action option. By adding other auxiliary rules (e.g. guessing rules $b(T) \vee \overline{b}(T) \leftarrow T \leq t$ for each action b), and setting $o = b$, we obtain a program P_D whose execution schedules $\mathcal{EX}(P_D)$ correspond to so-called *histories* (paths) of length t in D. An execution plan $e \in \mathcal{EX}(P_D)$ can then be materially executed. Similarly, preference orderings between actions as in the language $\mathcal{PP}$ and variants thereof [Son06], can be attached to action atoms: for an ordering $L_1 < \cdots < L_n$ among actions one can introduce corresponding integer weights $w_1 < \cdots < w_n$ and rules $\#L_i\{O,T\}[w_i : 1] \leftarrow L_i(T)$.

Knowledge Base Updates. As another potential usage of ACTHEX programs, we mention the possibility of updating knowledge bases, e.g., as achieved by the predicates *assert* and *retract* in Prolog. We assume that external environments contain a collection C of knowledge bases accessible by names, and consider abstract action constructs $assert(kb, f)$ and $retract(kb, f)$, which respectively should add or remove a statement f from a given knowledge base kb. The above can be grounded to ACTHEX programs, introducing action predicates $\#assert_k$ and $\#retract_k$, for $k > 0$[6]. An atom $\#assert_k[kb, a_1, \ldots, a_k]\{o,p\}$, (resp. $\#retract_k[kb, a_1, \ldots, a_k]\{o,p\}$) adds to (resp. removes from) the knowledge base kb the assertion $a_1|\ldots|a_k$, for $a_i|a_j$, being the string concatenation of a_i and a_j.

For instance, the rule $\#assert_3[kb, \text{“}n(\text{”}, X, \text{“}).\text{”}]\{b,1\} \leftarrow node(X).$ encodes the possible addition of facts $n(c)$ for each c such that $node(c) \in A$, for an answer set A. The above constructs can be fruitfully combined with reasoning over the given knowledge bases: to this end, we introduce the action atom $\#execute[kb]\{o,p\}$. Assuming the kb is a valid ACTHEX program, when such an atom belongs to the current execution schedule, it gets executed by evaluating kb and the resulting execution schedule. Note that whether $\#assert$, $\#retract$ and $\#execute$ actions will be executed depend on reasoning on the program at hand: this opens a variety of possibilities, e.g. belief revisions, and, in general, observe-think-act cycles [Kow99][7]. Note that the evaluation of programs with this kind of construct might not terminate in general: this issue is subject of ongoing study.

Translation of Agent Programs. Agent programs can also be realized in the ACTHEX framework. We consider logic-based *agent programs* as developed in [Sub00], consisting of rules of

[5]We can assume a constraint $\leftarrow L'_i(T), \overline{L}'_i(T)$ is added for each L_i. Note that the current implementation of ACTHEX programs allows for strong negation, by which an atom $\overline{L}'(T)$ can be conveniently modelled as $\neg L'(T)$.

[6]Our implementation of ACTHEX programs conveniently allows to program and group families of action atoms, like the above, using variable length parameter lists.

[7]An example ACTHEX program containing update actions is given at
`http://www.kr.tuwien.ac.at/research/systems/dlvhex/actionplugin/actionplugin_example1.html`

the form $Op_0\alpha_0 \leftarrow \chi, [\neg]Op_1\alpha_1, \ldots, [\neg]Op_m\alpha_m$, governing an agent's behaviour. The Op_i are *deontic modalities*, the α_i are *action status atoms*, and χ is a *code-call condition*.

For instance, $Do\ dial(N) \leftarrow in(N, phone(P)),\ O\ call(P)$, intuitively states that the agent should dial phone number N if she is obliged to call P. In [Sub00], a translation of an agent program $AG(P)$ to a logic program P is given, such that the answer sets of P correspond to the so-called *reasonable status sets* of $AG(P)$. We build on this transformation and model code-call conditions (which, e.g., provide access to actual sensor readings) using external atoms as already described in [Eit05]. Similarly, we model Do atoms as action atoms in our framework using rules of the sort $\#action_\alpha[\ldots]\{b\} \leftarrow Do\ \alpha$. A framework implementing this translation is available[8], featuring *a)* the translation of agent programs to ACTHEX programs, *b)* incorporating the actual execution of Do-able actions and *c)* an implementation of message box facilities for agents.

Other applications. ACTHEX programs can be exploited in a variety of other contexts, ranging from database access to interaction with actual web sources. We developed an example[9] illustrating how to exploit reasoning in ASP for choosing meeting schedules of two teams. Events are extracted from actual Google Calendars[10] of two teams; meeting dates are selected using ASP reasoning; eventually, the chosen events are posted to the calendars of the teams using an action atom of the form

$\#createEvent[Team, Url, \text{“}ActHexMeeting\text{”}, Date, User, Password]\{b, 1\}$.

4. Implementation Notes

An implementation of ACTHEX programs has been realized and is available[11] as an extension to the dlvhex system[12]. With respect to the traditional workflow of an answer set solver, the system computes execution schedules and executes one of it according to: *i)* the semantics of ACTHEX programs, *ii)* the selection policy of execution schedules described in Section 2, and *iii)* the associated executable functions provided for action predicates. The system is equipped with a toolkit enabling users to develop their own libraries of action predicates: some example libraries are available. In particular, the `KBModaddon` library constitutes a generalization of update action atoms as shown in Section 3 (it is, e.g., possible to execute arbitrary command line statements, and to assert and retract arbitrary statements from knowledge bases). An example library allowing access and modification to Google Calendars is also publicly available.

5. Related Work and Conclusions

Our work has points of contact with some lines of research which can be grouped as follows. *Action languages* serve the purpose of providing a declarative language for specifying causal theories [Giu98, McC97], allowing to assert not only the truth of a proposition, but also that there is a cause for it to be true. In this respect, they provide a formalism for the declarative representation of dynamic domains and gave rise to logic-based planning

[8] `http://students.sabanciuniv.edu/~ozanerdem/AgentToHex.html`

[9] `http://www.kr.tuwien.ac.at/research/systems/dlvhex/actionplugin/actionplugin_example2.html`

[10] `http://www.google.com/calendar`

[11] `http://www.kr.tuwien.ac.at/research/systems/dlvhex/actionplugin.html`

[12] `http://www.kr.tuwien.ac.at/research/systems/dlvhex/`

systems such as CCLAC [Giu04] and DLV$^{\mathcal{K}}$ [Eit03]. The two systems mentioned are based on transformations [Lif99, Gel93] to logic programming under the answer set semantics, however other (nonmonotonic) reasoning engines can be exploited for causal reasoning in action domains as well (cf, e.g., [Tur96, Kak01, Lin00]).

ACTHEX programs generalize HEX programs which in turn generalize ASP programs, and thus can be similarly used to implement planning systems based on action languages (as shown in Section 3). When resorting to ACTHEX, however, action atoms also encode their actual execution, enabling a variety of applications. For instance, this allows for interweaving plan generation and action execution seamlessly within a coherent declarative framework, which may, e.g., be utilized for an integrated approach to monitoring plan execution. For instance, [Nie07] extends the action language $\mathcal{K}$ towards conditional planning: building on HEX programs, they introduce external function calls in causal rules to import fluent information from an external source. The introduction of action atoms makes it possible to extend the framework coping with action execution and monitoring their success.

Logic-based agent programming constitutes a further natural application domain for ACTHEX programs: intelligent agents require reasoning and/or planning capabilities for acting in dynamic environments, and using logic programming for the declarative specification of a respective observe-think-act cycle [Kow99] is a reasonable choice. ACTHEX may serve as an implementation layer for agent systems built according to this paradigm. We exemplified its suitability providing a transformation of IMPACT agent programs [Sub00] into corresponding ACTHEX programs.

The evaluation of IMPACT agent programs is restricted to stratified negation in its current implementation: the given ACTHEX encoding does not require such a restriction and can handle general agent programs as formally conceived. Similarly, compared to ACTHEX, agent-oriented logic programming languages based on Horn clause languages (e.g., DALI [Cos04], or ALP [Dre09]) lack a declarative concept of negation, which is important from an expressive and practical modelling point of view, for instance to express exceptions. On the other hand, most nonmonotonic logic programming based approaches to agent-oriented programming, (e.g.[Alf06, Alf08, Nie06, Vos05, Lei01]), detach the reasoning process from the actual execution of an agent's actions (which often are termed 'external') and only their (expected) effects are taken into account for further deliberation. For such agent frameworks, ACTHEX can provide the platform for an integrated implementation. In conclusion, ACTHEX is a declarative logic programming framework including a representation for actions that are executed and have an impact on an external environment. Formal properties of the language and further extensions (e.g. parallel execution schedules) are subject to ongoing work. Corresponding results, as well as a more rigorous treatment of the given encodings, will be subject of follow-up work and/or an extended version of this paper.

References

[Alf06] J. J. Alferes, F. Banti, and A. Brogi. An event-condition-action logic programming language. In *JELIA*, pp. 29–42. 2006.

[Alf08] J. J. Alferes, A. Gabaldon, and J. Leite. Evolving logic programming based agents with temporal operators. In *IAT*, pp. 238–244. 2008.

[Cos04] S. Costantini and A. Tocchio. The DALI logic programming agent-oriented language. In *JELIA*, pp. 685–688. 2004.

[Dre09] C. Drescher, S. Schiffel, and M. Thielscher. A declarative agent programming language based on action theories. In *FroCos*, pp. 230–245. 2009.

[Eit99] T. Eiter, V. S. Subrahmanian, and G. Pick. Heterogeneous active agents, I: semantics. *Artif. Intell.*, 108(1-2):179–255, 1999. doi:http://dx.doi.org/10.1016/S0004-3702(99)00005-3.

[Eit03] T. Eiter, W. Faber, N. Leone, G. Pfeifer, and A. Polleres. A logic programming approach to knowledge-state planning, II: The DLV$^{\mathcal{K}}$ system. *Artif. Intell.*, 144(1-2):157–211, 2003.

[Eit05] T. Eiter, G. Ianni, R. Schindlauer, and H. Tompits. A uniform integration of higher-order reasoning and external evaluations in answer-set programming. In *IJCAI*, pp. 90–96. 2005.

[Eit08] T. Eiter, G. Ianni, T. Lukasiewicz, R. Schindlauer, and H. Tompits. Combining answer set programming with description logics for the semantic web. *Artif. Intell.*, 172(12-13):1495–1539, 2008.

[Fab04] W. Faber, N. Leone, and G. Pfeifer. Recursive aggregates in disjunctive logic programs: Semantics and complexity. In *JELIA*, pp. 200–212. 2004.

[Gel91] M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Comput.*, 9(3/4):365–386, 1991.

[Gel93] M. Gelfond and V. Lifschitz. Representing action and change by logic programs. *JLP*, 17:301–322, 1993.

[Giu98] E. Giunchiglia and V. Lifschitz. An action language based on causal explanation: Preliminary report. In *AAAI/IAAI*, pp. 623–630. 1998.

[Giu04] E. Giunchiglia, J. Lee, V. Lifschitz, N. McCain, and H. Turner. Nonmonotonic causal theories. *AI*, 153(1-2):49–104, 2004. doi:http://dx.doi.org/10.1016/j.artint.2002.12.001.

[Kak01] A. C. Kakas, R. Miller, and F. Toni. E-RES: Reasoning about actions, events and observations. In *LPNMR*, pp. 254–266. 2001.

[Kow99] R. A. Kowalski and F. Sadri. From logic programming towards multi-agent systems. *Ann. Math. Artif. Intell.*, 25(3-4):391–419, 1999.

[Lei01] J. A. Leite, J. J. Alferes, and L. M. Pereira. Minerva - a dynamic logic programming agent architecture. In *ATAL*, pp. 141–157. 2001.

[Lif99] V. Lifschitz and H. Turner. Representing transition systems by logic programs. In *LPNMR*, pp. 92–106. 1999.

[Lin00] F. Lin. From causal theories to successor state axioms and strips-like systems. In *AAAI/IAAI*, pp. 786–791. 2000.

[McC97] N. McCain and H. Turner. Causal theories of action and change. In *AAAI/IAAI*, pp. 460–465. 1997.

[Nie06] D. Van Nieuwenborgh, M. De Vos, S. Heymans, and D. Vermeir. Hierarchical decision making in multi-agent systems using answer set programming. In *CLIMA VII*, pp. 20–40. 2006.

[Nie07] D. Van Nieuwenborgh, T. Eiter, and D. Vermeir. Conditional planning with external functions. In *LPNMR*, pp. 214–227. 2007.

[Pir08] G. Pirrotta and A. Provetti. A Java wrapper for answer set programming inferential engines. In *CILC 2008*.

[Ric03] F. Ricca. The DLV Java wrapper. In *APPIA-GULP-PRODE*, pp. 263–274. 2003.

[Son06] T. C. Son and E. Pontelli. Planning with preferences using logic programming. *TPLP*, 6(5):559–607, 2006.

[Sub00] V. S. Subrahmanian, P. A. Bonatti, J. Dix, T. Eiter, S. Kraus, F. Ozcan, and R. B. Ross. *Heterogenous Active Agents*. MIT Press, 2000.

[Tur96] H. Turner. Representing actions in default logic: A situation calculus approach. In *In Proceedings of the Symposium in honor of Michael Gelfond's 50th birthday (also in Common Sense 96)*. 1996.

[Vos05] M. De Vos, T. Crick, J. A. Padget, M. Brain, O. Cliffe, and J. Needham. LAIMA: A multi-agent platform using ordered choice logic programming. In *DALT*, pp. 72–88. 2005.

Technical Communications of the International Conference on Logic Programming, 2010 (Edinburgh), pp. 34–43
http://www.floc-conference.org/ICLP-home.html

COMMUNICATING ANSWER SET PROGRAMS

KIM BAUTERS [1] AND JEROEN JANSSEN [2] AND STEVEN SCHOCKAERT [1] AND DIRK VERMEIR [2] AND MARTINE DE COCK [3,1]

[1] Department of Applied Mathematics and Computer Science, Universiteit Gent
Krijgslaan 281, 9000 Gent, Belgium
E-mail address: {kim.bauters,steven.schockaert}@ugent.be

[2] Department of Computer Science, Vrije Universiteit Brussel
Pleinlaan 2, 1050 Brussel, Belgium
E-mail address: {jeroen.janssen,dvermeir}@vub.ac.be

[3] Institute of Technology, University of Washington
1900 Commerce Street, WA-98402 Tacoma, USA
E-mail address: mdecock@u.washington.edu

Abstract. Answer set programming is a form of declarative programming that has proven very successful in succinctly formulating and solving complex problems. Although mechanisms for representing and reasoning with the combined answer set programs of multiple agents have already been proposed, the actual gain in expressivity when adding communication has not been thoroughly studied. We show that allowing simple programs to talk to each other results in the same expressivity as adding negation-as-failure. Furthermore, we show that the ability to focus on one program in a network of simple programs results in the same expressivity as adding disjunction in the head of the rules.

1. Introduction

The idea of answer set programming (ASP) is to represent the requirements of a computational problem by a logic program P such that particular minimal models of P, called answer sets and usually defined using some form of the stable model semantics [Gel88], correspond to the solutions of the original problem [Lif02]. The research on multi-context systems has, among other things, been concerned with studying how a group of simple agents can cooperate to find the solutions of global problems [Roe05, Bre07]. We start with an introductory example to illustrate how the ideas of multi-context systems can be used to solve problems in the ASP setting.

Kim Bauters and Jeroen Janssen are funded by a joint Research Foundation-Flanders (FWO) project. Steven Schockaert is a postdoctoral fellow of the Research Foundation-Flanders (FWO).

Technical Communications of the 26th International Conference on Logic Programming, Edinburgh, July, 2010
Editors: Manuel Hermenegildo, Torsten Schaub
LIPIcs - Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/LIPIcs.ICLP.2010.34

Example 1.1. A hotspot network consists of two hotspots H_1 and H_2. The hotspots are wired to each other to share an internet connection and provide wireless access to users in the area. A user U tries to connect to the closest detectable hotspot *e.g.* H_1. Now assume that H_1 is no longer accessible. H_1 cannot find this out by itself, nor can it rely on users telling this since they cannot connect. The rules below illustrate how we can model this knowledge using the communicating programs we describe in Section 2.2. For compactness, we abbreviate *accessible* as a, *access* as c, *problem* as p and *optimal* as o. Consider the program $\mathcal{P}_{intro}$ with the rules:

$$H_1 : \neg a \leftarrow H_2 : p \quad \text{[r1]} \qquad U : o \leftarrow H_1 : a \quad \text{[r5]}$$
$$H_2 : a \leftarrow \quad \text{[r2]} \qquad U : \neg o \leftarrow H_2 : a, not\ H_1 : a \quad \text{[r6]}$$
$$H_2 : \neg a \leftarrow H_1 : p \quad \text{[r3]} \qquad U : c \leftarrow H_1 : a \quad \text{[r7]}$$
$$H_2 : p \leftarrow U : \neg o \quad \text{[r4]} \qquad U : c \leftarrow H_2 : a \quad \text{[r8]}$$

Let $H_1 = \{\mathsf{r1}\}$, $H_2 = \{\mathsf{r2}, \mathsf{r3}, \mathsf{r4}\}$ and $U = \{\mathsf{r5}, \mathsf{r6}, \mathsf{r7}, \mathsf{r8}\}$. Note how the rules of the first hotspot H_1 differ from those of the second hotspot H_2, *i.e.* they are in different states. Indeed, the first hotspot H_1 cannot rely on the user to tell that there is a problem and it is not accessible. The second hotspot does not have these restrictions. It is easy to see that user U can deduce that she has access (r2, r8), though this access is not optimal (r6). The second hotspot detects this (r4) and concludes that there is a problem, allowing H_1 to derive that it is not accessible (r1).

In this paper we systematically study the effect of adding such kind of communication to ASP in terms of expressiveness. The communication between ASP programs that we propose is similar in spirit to the work in [Roe05, Bre07, Buc08]. Studying the expressiveness with a focus on simple ASP programs, however, is in contrast to approaches such as [De05, Van07] that start from expressive ASP variants, which obscures the analysis of the effect of communication on the expressiveness. A first contribution of this paper is that communicating simple programs can solve problems at the first level of the polynomial hierarchy and that communicating normal ASP programs do not offer any additional expressiveness. The second contribution is the introduction of a new, intuitive form of communication that allows for communicating simple ASP programs to solve problems at the second level of the polynomial hierarchy. The hardness results that we present in this paper in a sense complement the membership results from [Bre07]. However, our definitions of communicating ASP and minimality differ slightly, complicating a direct comparison of the results.

The remainder of this paper is organized as follows. Section 2 recalls the basic concepts and results from ASP which we use in this paper, and explores the syntax and semantics of communicating programs. In Section 3 we show that these communicating simple programs are capable of simulating normal programs that have negation-as-failure. In Section 4 we introduce focused communicating programs and show how networks of simple agents can simulate disjunctive ASP programs. Related work is discussed in Section 5 and Section 6 provides some final remarks.

2. Preliminaries

2.1. Answer set programming

We first recall the basic concepts and results from ASP that are used in this paper. To define ASP programs, we start from a countable set of atoms and we define a *literal* l as an atom a or its classical negation $\neg a$. If L is a set of literals, we use $\neg L$ to denote the set $\{\neg l \mid l \in L\}$ where, by definition, $\neg\neg a = a$. A set of literals L is *consistent* if $L \cap \neg L = \emptyset$. An *extended literal* is either a literal or a literal preceded by *not* which we call the negation-as-failure operator. For a set of literals L, we use $not(L)$ to denote the set $\{not\ l \mid l \in L\}$.

A *disjunctive rule* is an expression of the form $\gamma \leftarrow (\alpha \cup not(\beta))$ where γ is a set of literals (interpreted as a disjunction, denoted as $l_1; \ldots; l_n$) called the head of the rule and $(\alpha \cup not(\beta))$ (interpreted as a conjunction) is the body of the rule with α and β sets of literals. A *positive disjunctive rule* is a disjunctive rule without negation-as-failure in the body, *i.e.* with $\beta = \emptyset$. A *disjunctive program* P is a finite set of disjunctive rules. The *Herbrand base* $\mathcal{B}_P$ of P is the set of atoms appearing in program P. A (partial) *interpretation* I of P is any consistent set of literals $I \subseteq (\mathcal{B}_P \cup \neg\mathcal{B}_P)$. I is *total* iff $I \cup \neg I = \mathcal{B}_P \cup \neg\mathcal{B}_P$.

A *normal rule* is a disjunctive rule with at most one literal l in the head. A *normal program* P is a finite set of normal rules. A *simple rule* is a normal rule without negation-as-failure in the body. A *simple program* P is a finite set of simple rules. The *immediate consequence operator* T_P of a simple program P *w.r.t.* an interpretation I is defined as

$$T_P(I) = I \cup \{l \mid ((l \leftarrow \alpha) \in P) \wedge (\alpha \subseteq I)\}. \tag{2.1}$$

We use $P^\star$ to denote the fixpoint which is obtained by repeatedly applying T_P starting from the empty interpretation, *i.e.* the least fixpoint of T_P *w.r.t.* set inclusion. An interpretation I is an *answer set* of a simple program P iff $I = P^\star$.

The *reduct* P^I of a disjunctive program P *w.r.t.* the interpretation I is defined as $P^I = \{\gamma \leftarrow \alpha \mid (\gamma \leftarrow \alpha \cup not(\beta)) \in P, \beta \cap I = \emptyset\}$. I is an answer set of the disjunctive program P when I is the minimal model *w.r.t.* set inclusion of P^I. In the specific case of normal programs, answer sets can also be characterized in terms of fixpoints. Specifically, it is easy to see that the reduct P^I is a simple program. I is an answer set of the normal program P iff $\left(P^I\right)^\star = I$, *i.e.* if I is the answer set of the reduct P^I.

2.2. Communicating programs

The underlying intuition of communication between ASP programs is that of a function call or, in terms of agents, asking questions to other agents. This communication is based on a new kind of literal '$Q{:}l$', as in [Roe05, Bre07]. If the literal l is not in the answer set of Q then $Q{:}l$ is false; otherwise $Q{:}l$ is true. The semantics are closely related to the minimal semantics in [Bre07] and especially the semantics in [Buc08].

Let $\mathcal{P}$ be a finite set of program names. A *$\mathcal{P}$-situated literal* is an expression of the form $Q{:}l$ with $Q \in \mathcal{P}$ and l a literal. For $R \in \mathcal{P}$, a $\mathcal{P}$-situated literal $Q{:}l$ is called *R-local* if $Q = R$. For a set of literals L, we use $Q{:}L$ as a shorthand for $\{Q{:}l \mid l \in L\}$. For a set of $\mathcal{P}$-situated literals X and $Q \in \mathcal{P}$, we use $X_{\downarrow Q}$ to denote $\{l \mid Q{:}l \in X\}$, i.e. the projection of X on Q. A set of $\mathcal{P}$-situated literals X is *consistent* iff $X_{\downarrow Q}$ is consistent for all $Q \in \mathcal{P}$. By $\neg X$ we denote the set $\{Q{:}\neg l \mid Q{:}l \in X\}$ where we define $Q{:}\neg\neg l = Q{:}l$. An *extended $\mathcal{P}$-situated literal* is either a $\mathcal{P}$-situated literal or a $\mathcal{P}$-situated literal preceded by *not*. For

a set of $\mathcal{P}$-situated literals X, we use $not(X)$ to denote the set $\{not\ Q{:}l \mid Q{:}l \in X\}$. For a set of extended $\mathcal{P}$-situated literals X we denote by X_{pos} the set of $\mathcal{P}$-situated literals in X, *i.e.* those extended $\mathcal{P}$-situated literals in X that are not preceded by negation-as-failure, while $X_{\text{neg}} = \{Q{:}l \mid not\ Q{:}l \in X\}$.

A *$\mathcal{P}$-situated normal rule* is an expression of the form $Q : l \leftarrow (\alpha \cup not(\beta))$ where $Q : l$ is a single $\mathcal{P}$-situated literal, called the head of the rule, and $(\alpha \cup not(\beta))$ is called the body of the rule with α and β sets of $\mathcal{P}$-situated literals. A $\mathcal{P}$-situated normal rule $Q{:}l \leftarrow (\alpha \cup not\ (\beta))$ is called R-local whenever $Q = R$. A *$\mathcal{P}$-component normal program* Q is a finite set of Q-local $\mathcal{P}$-situated normal rules. Henceforth we shall use $\mathcal{P}$ to both denote the set of program names and to denote the set of actual $\mathcal{P}$-component normal programs. A *communicating normal program* $\mathcal{P}$ is then a finite set of $\mathcal{P}$-component normal programs.

A *$\mathcal{P}$-situated simple rule* is an expression of the form $Q{:}l \leftarrow \alpha$, *i.e.* a $\mathcal{P}$-situated normal rule without negation-as-failure in the body. A *$\mathcal{P}$-component simple program* Q is a finite set of Q-local $\mathcal{P}$-situated simple rules. A *communicating simple program* $\mathcal{P}$ is then a finite set of $\mathcal{P}$-component simple programs.

In the remainder of this paper we drop the $\mathcal{P}$-prefix whenever the set $\mathcal{P}$ is clear from the context. Whenever the name of the component normal program Q is clear, we write l instead of $Q : l$ for Q-local situated literals. For notational convenience, we write communicating program for communicating normal program. Finally note that a communicating normal (simple) program with only one component program trivially corresponds to a normal (simple) program.

Similar as for a normal program, we can define the *Herbrand base* for a component program Q as the set of atoms occurring in Q, which we denote as $\mathcal{B}_Q$. The Herbrand base of a communicating program $\mathcal{P}$ is defined as $\mathcal{B}_{\mathcal{P}} = \left\{Q{:}a \mid Q \in \mathcal{P} \text{ and } a \in \bigcup_{R \in \mathcal{P}} \mathcal{B}_R\right\}$. We say that a (partial) interpretation I of a communicating program $\mathcal{P}$ is any consistent subset $I \subseteq (\mathcal{B}_{\mathcal{P}} \cup \neg\mathcal{B}_{\mathcal{P}})$. Given an interpretation I of a communicating program $\mathcal{P}$, the reduct Q^I for $Q \in \mathcal{P}$ is the component simple program obtained by deleting

- each rule with an extended situated literal $not\ R{:}l$ in the body such that $R{:}l \in I$;
- each remaining extended situated literal of the form $not\ R{:}l$;
- each rule with a situated literal $R{:}l$ in the body that is not Q-local with $R{:}l \notin I$;
- each situated literal $R{:}l$ that is not Q-local and such that $R{:}l \in I$.

The underlying intuition of the reduct is clear. Analogous to the definition of a reduct of a normal programs [Gel88], the reduct of a communicating program defines a way to reduce this program relative to some guess I. The reduct of a communicating program is a communicating simple program that only contains component simple programs Q with Q-local situated literals. That is, each component simple program Q corresponds to a classical simple program. We tackle the problem of self-references in [Buc08] by treating Q-local situated literals in a different way. Since the communication is based on belief and internal reasoning is based on knowledge, this allows for "mutual influence" as in [Bre07, Buc08] where the belief of an agent can be supported by the agent itself, via belief in other agents. Also note that the belief between agents is the belief as identified in [Lif99], *i.e.* $Q : l$ is true whenever "$\neg not\ Q{:}l$" is true under the syntax and semantics introduced in [Lif99] for nested logic programs and when treating $Q{:}l$ as a fresh atom.

Definition 2.1. We say that an interpretation I of a communicating program $\mathcal{P}$ is an *answer set* of $\mathcal{P}$ if and only if we have that $\forall Q \in \mathcal{P} \cdot (Q{:}I_{\downarrow Q}) = \left(Q^I\right)^\star$.

Example 2.2. Consider the communicating program $\mathcal{P}_{intro}$ from Example 1.1. It is easy to see that $M = \{H_1 : \neg a, H_2 : a, H_2 : p, U : \neg o, U : c\}$ is the unique answer set of $\mathcal{P}_{intro}$. Indeed, we obtain the reducts $(H_1)^M = \{\neg a \leftarrow\}$, $(H_2)^M = \{a \leftarrow, p \leftarrow\}$ and $(U)^M = \{\neg o \leftarrow, c \leftarrow\}$ which have the answer sets $\{\neg a\}, \{a, p\}$ and $\{\neg o, c\}$, respectively.

3. Simulating Negation-as-Failure with Communication

The addition of communication to ASP programs provides added expressiveness and an increase in computational complexity, which we illustrate in this section. We show that a communicating simple program can simulate normal programs, where simple programs are P-complete and normal programs are NP-complete [Bar03]. Furthermore, we illustrate that, surprisingly, there is no difference in terms of computational complexity between communicating simple programs and communicating normal programs.

We start by giving an example of the transformation that allows to simulate (communicating) normal programs using communicating simple programs. Afterwards, we give a formal definition of this transformation.

Example 3.1. Consider the communicating normal program $\mathcal{E}$ with the rules

$$Q_1 : a \leftarrow not\ Q_2 : b$$
$$Q_2 : b \leftarrow not\ Q_1 : a.$$

When $Q_1 = Q_2$ this example corresponds to a normal program. The transformation we propose below results in the communicating simple program $\mathcal{P} = \{Q_1', Q_2', N_1, N_2\}$:

$$\begin{aligned} Q_1' : a &\leftarrow N_2 : \neg(b)^\dagger & \qquad N_1 : (a)^\dagger &\leftarrow Q_1' : a \\ Q_2' : b &\leftarrow N_1 : \neg(a)^\dagger & \qquad N_2 : (b)^\dagger &\leftarrow Q_2' : b \\ Q_1' : \neg(a)^\dagger &\leftarrow N_1 : \neg(a)^\dagger & \qquad N_1 : \neg(a)^\dagger &\leftarrow Q_1' : \neg(a)^\dagger \\ Q_2' : \neg(b)^\dagger &\leftarrow N_2 : \neg(b)^\dagger & \qquad N_2 : \neg(b)^\dagger &\leftarrow Q_2' : \neg(b)^\dagger. \end{aligned}$$

The transformation creates two types of *'worlds'*, Q_i' and N_i with $1 \leq i \leq 2$, which are all component programs. Q_i' is similar to Q_i, although occurrences of extended situated literals of the form $not\ Q_i : l$ are replaced by $N_i : \neg(l)^\dagger$, with $(l)^\dagger$ a fresh literal. The non-monotonicity associated with negation-as-failure is simulated by introducing the rules $\neg(l)^\dagger \leftarrow N_i : \neg(l)^\dagger$ and $\neg(l)^\dagger \leftarrow Q_i' : \neg(l)^\dagger$ in Q_i' and N_i, respectively. Finally, we add rules of the form $(l)^\dagger \leftarrow Q_i' : l$ to N_i, creating an inconsistency when N_i believes $\neg(l)^\dagger$ when Q_i' believes l.

The resulting communicating simple program $\mathcal{P}$ is an equivalent program in that its answer sets correspond to those of the original communicating program, yet without using negation-as-failure. Indeed, the answer sets of $\mathcal{E}$ are $\{Q_1 : a\}$ and $\{Q_2 : b\}$ and the answer sets of $\mathcal{P}$ are $\left\{Q_1' : a, Q_2' : \neg(b)^\dagger, N_2 : \neg(b)^\dagger, N_1 : (a)^\dagger\right\}$ and $\left\{Q_2' : b, Q_1' : \neg(a)^\dagger, N_1 : \neg(a)^\dagger, N_2 : (b)^\dagger\right\}$. Note furthermore how this is a polynomial transformation with at most $3 \cdot |\mathcal{E}_{\text{neg}}|$ additional rules with $\mathcal{E}_{\text{neg}}$ as defined in Definition 3.2.

Definition 3.2. Let $\mathcal{E} = \{Q_1, \ldots, Q_n\}$ be a communicating program. The communicating simple program $\mathcal{P} = \{Q_1', \ldots, Q_n', N_1, \ldots, N_n\}$ with $1 \leq i, j \leq n$ that simulates $\mathcal{E}$ is defined

by

$$
\begin{aligned}
Q'_i &= \left\{ l \leftarrow \alpha'_{\text{pos}} \cup \left\{ N_j{:}\neg(k)^\dagger \mid Q_j{:}k \in \alpha_{\text{neg}} \right\} \mid (l \leftarrow \alpha) \in Q_i \right\} && (3.1)\\
&\cup \left\{ \neg(b)^\dagger \leftarrow N_i{:}\neg(b)^\dagger \mid Q_i{:}b \in \mathcal{E}_{\text{neg}} \right\} && (3.2)\\
N_i &= \left\{ \neg(b)^\dagger \leftarrow Q'_i{:}\neg(b)^\dagger \mid Q_i{:}b \in \mathcal{E}_{\text{neg}} \right\} && (3.3)\\
&\cup \left\{ (b)^\dagger \leftarrow Q'_i{:}b \mid Q_i{:}b \in \mathcal{E}_{\text{neg}} \right\} && (3.4)
\end{aligned}
$$

with $\alpha'_{pos} = \left\{ Q'_j{:}l \mid Q_j{:}l \in \alpha_{pos} \right\}$ and $\mathcal{E}_{\text{neg}} = \bigcup_{i=1}^{n} \left(\bigcup_{(a \leftarrow \alpha) \in Q_i} \alpha_{\text{neg}} \right)$.

Recall that both $\neg(b)^\dagger$ and $(b)^\dagger$ are fresh literals that intuitively correspond to $\neg b$ and b. We use Q'_i+ to denote the rules in Q'_i defined by (3.1) and Q'_i- to denote the rules in Q'_i defined by (3.2).

Intuitively, the transformation employs the non-monotonic property of the belief underlying the situated literals to simulate negation-as-failure. This is obtained from the interplay between the rules (3.2) and (3.3). As such, we can use the new literal '$\neg(b)^\dagger$' instead of the original extended (situated) literal '*not* b', allowing us to rewrite the rules as we do in (3.1). In order to ensure that the simulation works, even when the program we want to simulate contains true negation, we need to specify some additional bookkeeping (3.4).

As becomes clear from Proposition 3.3 and Proposition 3.4, the above transformation preserves the semantics of the original program. Since we can easily rewrite any normal program as a communicating normal program, the importance of this is thus twofold. On one hand, we reveal that communicating normal programs do not have any additional expressive power over communicating simple programs. On the other hand, it follows that the expressiveness of communicating simple programs allows us to solve NP-complete problems, since finding the answer set of normal programs is an NP-complete problem [Bar03].

Proposition 3.3. *Let $\mathcal{P} = \{Q_1, \ldots, Q_n\}$ and let $\mathcal{P}' = \{Q'_1, \ldots, Q'_n, N_1, \ldots, N_n\}$ with $\mathcal{P}$ a communicating program and $\mathcal{P}'$ the communicating simple program that simulates $\mathcal{P}$ as defined in Definition 3.2. If M is an answer set of $\mathcal{P}$, then M' is an answer set of $\mathcal{P}'$ with M' defined as:*

$$
\begin{aligned}
M' = &\left\{ Q'_i{:}a \mid a \in M_{\downarrow Q_i}, Q_i \in \mathcal{P} \right\}\\
&\cup \left\{ Q'_i{:}\neg(b)^\dagger \mid b \notin M_{\downarrow Q_i}, Q_i \in \mathcal{P} \right\}\\
&\cup \left\{ N_i{:}\neg(b)^\dagger \mid b \notin M_{\downarrow Q_i}, Q_i \in \mathcal{P} \right\}\\
&\cup \left\{ N_i{:}(a)^\dagger \mid a \in M_{\downarrow Q_i}, Q_i \in \mathcal{P} \right\}.
\end{aligned} \tag{3.5}
$$

Proposition 3.4. *Let $\mathcal{P} = \{Q_1, \ldots, Q_n\}$ and let $\mathcal{P}' = \{Q'_1, \ldots, Q'_n, N_1, \ldots, N_n\}$ with $\mathcal{P}$ a communicating program and $\mathcal{P}'$ the communicating simple program that simulates $\mathcal{P}$. Assume that M' is an answer set of $\mathcal{P}'$ and that $(M')_{\downarrow N_i}$ is total* w.r.t. *$\mathcal{B}_{N_i}$ for all $i \in \{1, \ldots, n\}$. Then the interpretation M defined as*

$$
M = \left\{ Q_i{:}b \mid b \in \left(\left(Q'_i+ \right)^{M'} \right)^\star \right\} \tag{3.6}
$$

is an answer set of $\mathcal{P}$. ∎

Note that the requirement for M' to be a total answer set of $\mathcal{P}$ in N_i is necessary in this last proposition, as demonstrated by the following example.

Example 3.5. Consider the normal program $R = \{a \leftarrow not\ a\}$ which has no answer sets. The corresponding communicating simple program $\mathcal{P} = \{Q, N\}$ has the following rules:

$$Q : a \leftarrow N : \neg(a)^\dagger \qquad N : \neg(a)^\dagger \leftarrow Q : \neg(a)^\dagger$$
$$Q : \neg(a)^\dagger \leftarrow N : \neg(a)^\dagger \qquad N : (a)^\dagger \leftarrow Q : a.$$

It is easy to see that $I = \emptyset$ is an answer set of $\mathcal{P}$ since we have $Q^I = N^I = \emptyset$.

4. Focused Communicating Programs

In this section, we extend the semantics of communicating programs in such a way that it is possible to focus on a single component program. That is, we indicate that we are not interested in the answer sets of the entire network of component programs, but only in answer sets of a single component program. The underlying intuition is that of auxiliary functions or, in terms of agents, a team governed by a leader who forwards (and possibly amends) the conclusions. We are thus varying the communication mechanism, without altering the expressiveness of the agents in the network.

Definition 4.1. Let $\mathcal{P}$ be a communicating program and $Q \in \mathcal{P}$ a component program. A *Q-focused answer set* of $\mathcal{P}$ is any subset-minimal element of

$$\{M_{\downarrow Q} \mid M \text{ an answer set of } \mathcal{P}\}.$$

If we are only interested in Q-focused answer sets, then $\mathcal{P}$ is called a Q-focused communicating program, denoted as $\mathcal{P}_{\downarrow Q}$. As before, we drop the Q-prefix when the component program Q is clear from the context.

Example 4.2. Consider the communicating program $\mathcal{P}_{focus} = \{Q, R\}$ with the rules

$$Q = \{a \leftarrow, b \leftarrow, c \leftarrow not\ R : c\}$$
$$R = \{a \leftarrow not\ c, c \leftarrow not\ a, d \leftarrow c\}.$$

The communicating program $\mathcal{P}_{focus}$ has two answer sets, namely $M_1 = Q : \{a, b, c\} \cup \{R : a\}$ and $M_2 = Q : \{a, b\} \cup R : \{c, d\}$. The only Q-focused answer set of $\mathcal{P}_{focus}$ is $\{a, b\}$ since $M_{1\downarrow Q} = \{a, b, c\}$ and $M_{2\downarrow Q} = \{a, b\}$.

This simple extension is all that is needed to take another step in the complexity hierarchy. That is, the complexity of finding the answer sets of a focused communicating program is Σ_2^{P}-hard.[1] Before we state this result, we first explain that any positive disjunctive program can be simulated using focused communicating programs. The underlying intuition is straightforward. We delegate the disjunction in the head to a new component program where we simulate the corresponding choice using negation-as-failure. The results of these component programs are then grouped in an aggregate component program on which we focus to ensure that we only retain the minimal models that correspond with the answer sets of the original positive disjunctive program. We start with an example to illustrate the simulation.

[1]Recall that Σ_2^{P} is the class of problems that can be solved in polynomial time on a non-deterministic machine with an NP oracle, *i.e.* $\Sigma_2^{\mathsf{P}} = \mathsf{NP}^{\mathsf{NP}}$.

Example 4.3. Consider the positive disjunctive program $D = \{a; b \leftarrow, a \leftarrow b, b \leftarrow a\}$. The corresponding focused program $(\mathcal{P}_{simulate})_{\downarrow Q} = \{Q, R_1\}$ has the following rules:

$$\begin{array}{ll} R_1{:}a \leftarrow not\ R_1{:}b & Q{:}a \leftarrow R_1{:}a \\ R_1{:}b \leftarrow not\ R_1{:}a & Q{:}b \leftarrow R_1{:}b \\ & Q{:}a \leftarrow Q{:}b \\ & Q{:}b \leftarrow Q{:}a \end{array}$$

The answer sets of $\mathcal{P}_{simulate}$ are $\{R_1{:}a\} \cup Q{:}\{a, b\}$ and $\{R_1{:}b\} \cup Q{:}\{a, b\}$. The unique answer set of $(\mathcal{P}_{simulate})_{\downarrow Q}$ is therefore $\{a, b\}$, which is also the unique answer set of D.

Definition 4.4. Let $D = \{r_1, \ldots, r_n, r_{n+1}, \ldots, r_s\}$ be a positive disjunctive program where $r_i = \gamma_i \leftarrow \alpha_i$ such that $|\gamma_i| > 1$ for $i \in \{1, \ldots, n\}$ and $|\gamma_i| \in \{0, 1\}$ for $i \in \{n+1, \ldots, s\}$. The focused program that simulates D, $\mathcal{P}_{\downarrow Q} = \{Q, R_1, \ldots, R_n\}$, is defined by

$$\begin{aligned} Q = &\{r_i \mid i \in \{n+1, \ldots, s\}\} \\ &\cup \{l \leftarrow \{R_i{:}l\} \cup \alpha_i \mid i \in \{1, \ldots, n\}, l \in \gamma_i\} \end{aligned} \tag{4.1}$$

where for $i \in \{1, \ldots, n\}$ we have

$$R_i = \{l \leftarrow not(\gamma_i \setminus \{l\}) \mid l \in \gamma_i\}. \tag{4.2}$$

Proposition 4.5. *Let D be a positive disjunctive program and $\mathcal{P}_{\downarrow Q}$ the focused communicating program that simulates D. M is an answer set of D iff M is an answer set of $\mathcal{P}_{\downarrow Q}$.* ∎

We can thus use focused communicating programs to solve existential-universal quantifiable boolean formulae (*e.g.* by simulating the disjunctive ASP program proposed in [Bar03]). This can be used as the basis of a proof to show that finding the answer sets of focused communicating programs is in Σ_2^{P}.

Corollary 4.6. *Deciding whether a Q-focused communicating (simple) program $\mathcal{P}_{\downarrow Q}$ with two or more component programs has an answer set containing a specific literal l is Σ_2^{P}-hard. Membership in Σ_2^{P} can also be shown, thus this problem is Σ_2^{P}-complete.* ∎

5. Related Work

A large body of research has been devoted to combining logic programming with multi-agent or multi-context ideas for various reasons. Among others, the logic can be used to describe the (rational) behaviour of the agents in a multi-agent network, as in [Del99]. It can be used to combine different flavours of logic programming languages [Luo05, Eit08]. It can be used to externally solve tasks for which ASP is not suited, yet remaining in a declarative framework [Eit06]. It can also be used as a form of cooperation, where multiple agents or contexts collaborate to solve a difficult problem [De05, Van07].

The approach described in this paper falls into this last category and studies the expressiveness of the communication component in communicating ASP. In contrast to [De05, Van07] our approach is based on simple programs and on asking for information instead of pushing (partial) answer sets to the next ASP program in the network. Like in [De05], but in contrast with [Van07], we allow circular communication between programs and do not force a linear network of ASP programs that in turn refine the results of previous steps.

Table 1: Complexity of Communicating Answer Set Programming

	no communication	with communication	focused communication
simple program	P-hard	NP-hard	Σ_2^P-hard
normal program	NP-hard	NP-hard	Σ_2^P-hard

Complexity studies in this setting have been performed but with some notable differences. For example, [Bre07] generalises towards heterogenous non-monotonic multi-context systems in which different flavours of logic programming languages work together to solve a problem.

It is shown that the complexity of verifying whether some literal is contained in some (resp. all) solutions is in Σ_k^P (resp. Π_k^P), where the value of k depends on the underlying logic that is used.

In [DT09], recursive modular nonmonotonic logic programs (MLP) under the ASP semantics are considered. The main difference between MLP and our simple communication is that our communication is parameter-less, *i.e.* the truth of a situated literal is not dependent on parameters passed by the situated literal to the target component program.

The work in this paper is different from all of the above in that it studies the expressiveness of communicating answer set programs with simple rules while varying the mechanisms for parameter-less communication between the agents.

6. Conclusion

In this paper we have systematically studied the effect of adding communication to ASP in terms of expressiveness and computational complexity. One of the most interesting results is that communicating simple programs (without negation-as-failure) are expressive enough to simulate communicating normal programs (with negation-as-failure). To show this, we have provided an actual translation of a communication normal ASP program into an equivalent communicating ASP program with only simple rules. Since normal programs are a special case of communicating normal programs, and solving normal programs is known to be NP-complete, this entails that solving communicating simple programs is an NP-hard problem.

Additionally, we introduce focused communicating programs where we "focus" on the results of a single component program. The other component programs can still contribute to solving the problem at hand, but they no longer have a direct influence over the resulting answer set. Indeed, the component program on which we focus can override any and all conclusions. Such focused communicating programs can easily be obtained by varying the parameter-less communication mechanism found in the communicating programs introduced in the first part of this paper. Focused communicating programs can be used to simulate programs with disjunctive rules without negation-as-failure and are able to solve problems in Σ_2^P. Table 1 summarises our main results.

Acknowledgment

The authors wish to thank the anonymous reviewers for their references to related work, as well as for their comments and suggestions that helped to improve the quality of this

paper. Special thanks go out to Pascal Nicolas, Claire Lefèvre and Laurent Garcia for their fruitful discussions that led to new insights.

References

[Bar03] Chitta Baral. *Knowledge, Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2003.

[Bre07] Gerhard Brewka and Thomas Eiter. Equilibria in heterogeneous nonmonotonic multi-context systems. In *Proc. of AAAI07*, pp. 385–390. 2007.

[Buc08] Francesco Buccafurri, Gianluca Caminiti, and Rosario Laurendi. A logic language with stable model semantics for social reasoning. In *Proc. of ICLP08*, pp. 718–723. 2008.

[De05] Marina De Vos, Tom Crick, Julian Padget, Martin Brain, Owen Cliffe, and Jonathan Needham. LAIMA: A multi-agent platform using ordered choice logic programming. In *Declarative Agent Languages and Technologies III*, pp. 72–88. 2005.

[Del99] Pierangelo Dell'Acqua, Fariba Sadri, and Francesca Toni. Communicating agents. In *Proceedings of the International Workshop on Multi-Agent Systems in Logic Programming*. 1999.

[DT09] Minh Dao-Tran, Thomas Eiter, Michael Fink, and Thomas Krennwallner. Modular nonmonotonic logic programming revisited. In *Proc. of ICLP*, pp. 145–159. 2009.

[Eit06] Thomas Eiter, Giovambattista Ianni, Roman Schindlauer, and Hans Tompits. dlvhex: A tool for semantic-web reasoning under the answer-set semantics. In *Proceedings of International Workshop on Applications of Logic Programming in the Semantic Web and Semantic Web Services*, pp. 33–39. 2006.

[Eit08] Thomas Eiter, Giovambattista Ianni, Thomas Lukasiewicz, Roman Schindlauer, and Hans Tompits. Combining answer set programming with description logics for the semantic web. *Artifial Intelligence*, 172(12–13):1495–1539, 2008.

[Gel88] Michael Gelfond and Vladimir Lifzchitz. The stable model semantics for logic programming. In *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pp. 1081–1086. 1988.

[Lif99] Vladimir Lifschitz, Lappoon R. Tang, and Hudson Turner. Nested expressions in logic programs. *Ann. Math. Artif. Intell.*, 25(3-4):369–389, 1999.

[Lif02] Vladimir Lifschitz. Answer set programming and plan generation. *Artificial Intelligence*, 138:39–54, 2002.

[Luo05] Jiewen Luo, Zhongzhi Shi, Maoguang Wang, and He Huang. Multi-agent cooperation: A description logic view. In *Proc. of PRIMA05*, pp. 365–379. 2005.

[Roe05] Floris Roelofsen and Luciano Serafini. Minimal and absent information in contexts. In *Proc. of IJCAI05*, pp. 558–563. 2005.

[Van07] Davy Van Nieuwenborgh, Marina De Vos, Stijn Heymans, and Dirk Vermeir. Hierarchical decision making in multi-agent systems using answer set programming. In *Proc. of CLIMA07*. 2007.

Technical Communications of the International Conference on Logic Programming, 2010 (Edinburgh), pp. 44–53
http://www.floc-conference.org/ICLP-home.html

IMPLEMENTATION ALTERNATIVES FOR BOTTOM-UP EVALUATION

STEFAN BRASS [1]

[1] Martin-Luther-Universität Halle-Wittenberg, Institut für Informatik,
Von-Seckendorff-Platz 1, D-06099 Halle (Saale), Germany
E-mail address: brass@informatik.uni-halle.de

Abstract. Bottom-up evaluation is a central part of query evaluation / program execution in deductive databases. It is used after a source code optimization like magic sets or SLDmagic that ensures that only facts relevant for the query can be derived. Then bottom-up evaluation simply performs the iteration of the standard T_P-operator to compute the minimal model. However, there are different ways to implement bottom-up evaluation efficiently. Since this is most critical for the performance of a deductive database system, and since performance is critical for the acceptance of deductive database technology, this question deserves a thorough analysis. In this paper we start this work by discussing several different implementation alternatives. Especially, we propose a new implementation of bottom-up evaluation called "Push-Method".

1. Introduction

Deductive databases [Min88, Ull90, Fre91, Ram94, Vag94, Fri95, Ram95] have not yet been very successful in practice (at least in terms of market share). However, their basic idea is practically very important: Deductive databases aim at an integrated system of database and programming language that is based on the declarative paradigm which was so successful in database languages. Currently, database programming is typically done in languages like PHP or Java. The programs construct SQL statements, send them to the database server, fetch the results, and process them. The interface is not very smooth, and although the situation can be improved with specific database languages like PL/SQL and server-side procedures / user-defined functions within the DBMS, the language paradigms remain different. Object-oriented databases were one approach to develop an integrated system based on a single paradigm, but there the declarativity of the database query part was sacrificed, and they did not get a significant market share, either. Nevertheless, there is an obvious demand for integrated database/programming systems, and this demand has even grown because of object-relational features that need programming inside the database server, and because of web and XML applications.

One of the reasons why deductive databases were not yet successful is the non-satisfying performance of many prototypes. This is also related to the impression that most deductive database prototypes have not really tried to be useful also as a programming platform — they were concentrated only on recursive query evaluation.

1998 ACM Subject Classification: I.2.3 Logic Programming, H.2.4 Query processing.
Key words and phrases: deductive databases, bottom-up evaluation, implementation.

Technical Communications of the 26th International Conference on Logic Programming, Edinburgh, July, 2010
Editors: Manuel Hermenegildo, Torsten Schaub
LIPIcs - Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/LIPIcs.ICLP.2010.44

Of course, recursive query evaluation is an important task, because many applications use tree-structured or graph-structured data. There has been a lot of progress over the years in this area [Ban86b]. A large part of this work was about source-level optimizations, like the well-known magic-set method [Ban86a, Bee91] and its many optimizations, including the SLDMagic-method of the author [Bra00].

However, this all depends on an efficient implementation of bottom-up evaluation. If one wants to build a new deductive database system which a real chance for acceptance in practice, one needs to clarify first how bottom-up evaluation should be done. This is not obvious, and several alternatives will be discussed in this paper.

Note that programming in deductive databases is not the same as programming in Pure Prolog. In deductive databases one thinks in the direction of the arrow, because they are based on bottom-up evaluation. For instance left recursion is very natural in this way, whereas in Prolog it must be avoided.

Top-down systems with tabling (like the XSB system [Sag94]) have a middle position between Prolog and deductive databases. Currently they have better performance than systems based on bottom-up evaluation. Our belief is that the bottom-up approach has still room for improvement in order to deliver competitive performance. In [Bra00] we proposed a source-level transformation that is for tail-recursions asymptotically faster than the standard magic set method (and also than the tabling method underlying the XSB system). It is also interesting because it unifies many improvements which were proposed for the magic set method over time.

After a source-level transformation like SLDmagic, which solves the problem of goal-direction, one needs an efficient implementation of bottom-up evaluation. The research reported in this paper is a step in this direction.

The approach we want to follow is to translate Datalog into C++, which can then be compiled to machine code. We did first performance tests with the methods described in this paper, but because of space restrictions, we must refer to

`http://www.informatik.uni-halle.de/~brass/botup/`

for the results.

2. Basic Framework

There are three types of predicates:

- EDB predicates ("extensional database"), the given database relations,
- IDB predicates ("intensional database"), which are defined by means of rules,
- built-in predicates like $<$, which usually have an infinite extension, and are defined by means of program code inside the system.

The purpose of bottom-up evaluation is to compute the extensions of the IDB relations. Actually, only one of them is the "answer predicate", the extension of which must be printed, or otherwise made available to the user.

Bottom-up evaluation works by applying the rules from right to left, so basically it computes the minimal model by iterating the standard T_P-operator. Of course, an important goal is to apply every applicable rule instance only once via rule-ordering and managing deltas for recursive rules ("seminaive evaluation"). However, slight exceptions are possible, because there is a tradeoff with the work needed for storing and accessing again intermediate facts.

Because of the infinite extension, built-in predicates can only be called when certain arguments are bound (i.e. input arguments, known values). In contrast, a free argument position permits a variable (output argument). The restrictions for the predicates are described by binding patterns (modes, adornments), e.g. $<$ can be called for the binding pattern `bb` only (every letter in a binding pattern corresponds to an argument position, `b` means bound, `f` means free).

A basic interface for relations is that it is possible to open a cursor (scan, iterator) over the relation, which permits to loop over all tuples. We assume that for every normal predicate p, there is a class p`_cursor` with the following methods:

- `void open()`: Open a scan over the relation (cursor is placed before the first tuple).
- `bool fetch()`: Move the cursor to the next tuple. This function must also be called to access the first tuple. It returns `true` if there is a first/next tuple, or `false`, if the cursor is at the end of the relation.
- T `col_`i`()`: Get the value of the i-th column/attribute (with data type T).
- `close()`: Close the cursor.

For the push method, we will also need

- `push()`: Save the state of the cursor on a global stack.
- `pop()`: Restore the state of the cursor.

(A merge join sometimes needs the possibility to return to a saved position in a scan, too.) A relation may have special access structures (e.g. it might be stored in a B-tree or an array). Then not only a full scan (corresponding to binding pattern `ff...f`) is possible, but also scans only over tuples with a given value for certain arguments. We assume that in such cases there are additional cursor classes called p`_cursor_`β, with a binding pattern β. These classes have the same methods as the other cursor classes, only the `open`-method has parameters for the bound arguments. E.g. if `p` is a predicate of arity 3 which permits especially fast access to tuples with a given value of the first argument, and if this argument has type `int`, the class `p_cursor_bff` would have the method `open(int x)`.

Actually, some access structures can efficiently evaluate small conjunctions with parameters, e.g. a B-tree over the first argument of `p` would also support a query of the form `p(X,Y,Z)` $\wedge$ `X` $\geq c_1$ $\wedge$ `X` $< c_2$, where c_1 and c_2 are integer constants or bound variables. This is not in the focus of the current paper, but a realistic system must be able to make use of such possibilities.

In addition, we need a possibility to create new tuples for predicates defined by rules (IDB predicates). We assume that for each predicate p there is a class p with a class method `insert` that creates a new tuple in p. Of course, it is possible that the objects of class p correspond to individual tuples, but since we only use the cursor interface, this is only one possible implementation.

For seminaive evaluation of recursive programs, additional cursor types are needed (e.g. `p_cursor_diff` runs only over tuples generated in the previous step of the fixpoint iteration), and a method to switch to the next iteration step (class method `next_iter` of `p`).

3. Materializing Derived Predicates

The first, most basic method for implementing bottom-up evaluation is to explicitly create a stored relation for every IDB-prediate. As an example, let us consider

`p(X, Z, 2)` $\leftarrow$ `q(X, Y)` $\wedge$ `r(Y, 5, Z).`

```
q_cursor q1;
q1.open();
while(q1.fetch()) {
    int X = q1.col_1();
    int Y = q1.col_2();
    r_cursor_bff r1; // if there is an index on the first argument
                     // (and none for the binding pattern bbf)
    r1.open(Y);
    while(r1.fetch()) {
        if(r1.col_2() == 5) {
            int Z = r1.col_3();
            p::insert(X, Z, 2);
        }
    }
}
```

Figure 1: Materializing an IDB-Predicate: `p(X, Z, 2)` $\leftarrow$ `q(X, Y)` $\wedge$ `r(Y, 5, Z).`

We assume that all columns in the examples have type `int`.

Of course, one option is to use a standard relational database, create a table for every IDB predicate, and send SQL statements to the database to execute the rules. But using a separate system for the management of facts causes performance penalties. Furthermore, for the seminaive evaluation of recursive rules, there is no good and efficient way to manage the deltas with SQL (the set of tuples newly derived in an iteration). Another interesting problem is that a good sideways information passing (SIP) strategy in the magic set method or selection function in the SLDMagic method needs already knowledge about existing indexes and relation sizes. Therefore it is not a good idea to do query optimization in two completely separate systems: The chosen SIP strategy/selection function more or less prescribes the evaluation of the resulting rules. For instance, if one wants to use a merge join, less "sideways information passing" is possible than with a nested loop/index join.

Therefore, one would do basic rule evaluation in the deductive database system itself, although it might be possible to use parts of a standard relational system (e.g. the storage manager). E.g. with a nested loop/index join, the implementation of the above rule would look as shown in Figure 1.

Of course, the materialization method causes a lot of copying. E.g., disjunctions must be expressed in standard Datalog with a derived predicate:

```
p(X, Y) ← q(X, Y).
p(X, Y) ← r(X, Y).
```

The materialization method would copy all `q`- and `r`-facts. This is especially expensive if the data values `X` and `Y` are large (e.g. longer strings). The problem can be reduced by working only with pointers to the real data values (but that might not make optimal use of the memory cache in the CPU, because values are more scattered around in memory).

4. Pull-Method

Of course, explicitly materializing every intermediate predicate needs a lot of memory. Therefore, it is a standard technique in databases to compute tuples only on demand, or

```
class p_cursor {
public:
    void open()
    {
        q1.open();
        q1_more = q1.fetch();
        if(q1_more)
            r1.open(q1.col_2()); // Assuming again index on first argument
    }
    bool fetch()
    {
        while(q1_more) {
            while(r1.fetch()) {
                if(r1.col_2() == 5)
                    return true;
            }
            r1.close();
            q1_more = q1.fetch();
            if(q1_more)
                r1.open(q1.col_2());
        }
        return false;
    }
    int col_1() { return q1.col_1(); }
    int col_2() { return r1.col_3(); }
    int col_3() { return 2; }
private:
    q_cursor q1;
    r_cursor_bff r1;
    bool q1_more;
};
```

Figure 2: Pull-Method (Lazy Evaluation): `p(X, Z, 2)` $\leftarrow$ `q(X, Y)` $\wedge$ `r(Y, 5, Z).`

actually not even compute the entire tuple, but permit access to its columns (in this way, possibly large data values do not have to be copied). In order to get such "lazy" evaluation, one only needs to support the cursor interface for each predicate. Let us consider again

`p(X, Z, 2)` $\leftarrow$ `q(X, Y)` $\wedge$ `r(Y, 5, Z).`

If `p` is nonrecursive, and this is the only rule, and no duplicate elimination is needed, the code would look as shown in Figure 2.

If duplicate elimination is needed, storing all tuples is necessary (unless the tuples for the body literals are generated in a fitting sort order). One can then apply the materialization method, or extend the pull-method by building a hash table of all previously returned tuples, and adding a check that the derived tuple is new.

An important disadvantage of the pull-method is that it causes recomputation if multiple scans over a predicate are performed. This does not only happen when there are several

body literals with the same predicate p, but also when a single p-literal appears in the inner loop of a nested loop join. However, recomputation is not necessarily something evil that must be avoided at any price. If the recomputation is not expensive, as in the example with the predicate describing a disjunction, it is a possible alternative.

Recursion with the pure pull-method is not possible: If one tries to implement recursion with the recursive opening of cursors, this leads to an infinite recursion even for acyclic relations since no bindings are passed to the recursive call: Each call does the same work again. Of course, it is possible to integrate standard seminaive evaluation, but this simply means to use the materialization method at least once in each recursive cycle.

5. Push-Method

It is also possible to apply the rules strictly from right to left, and move generated facts immediately to the place where they are needed. In contrast to the materialization method, a rule is not applied to produce all consequences in the current state, but only a single fact is derived each time. This reduces the need for intermediate storage and copying, which was also the main motivation for the pull-method. But here the producer of facts is in control, not the consumer as in the pull-method.

Of course, usually several facts can be derived with a rule. Therefore, once a fact is derived, one must store the current state of rule application for later backtracking. Then control jumps to a rule where this fact matches a body literal. There can be several rules that might use the produced fact, in which case again a backtrack point is generated.

This method basically works only with rules that have at most one body literal with IDB-predicate, because then matching facts for the other (EDB) body literals are available when a fact for the IDB body literal arrives. The SLDMagic method [Bra00] produces such rules as output of the program transformation, therefore this case is practically interesting. Furthermore, when there are several body literals with IDB-predicates, it is often possible to use the materialization or pull method for the predicates of all but one body literal.

The push method is applicable to linear recursion, and that is in fact one of its strengths (it can be very efficient in this case).

Let us explain how it works. First one creates a variable for every column of an IDB predicate. Consider again the example rule:

$$\texttt{p(X, Z, 2)} \leftarrow \texttt{q(X, Y)} \wedge \texttt{r(Y, 5, Z).}$$

If q is an IDB predicate, r is an EDB-predicate, and all arguments have type int, we get:

```
int q_1, q_2;
int p_1, p_2, p_3;
```

In addition, there is a code piece for every body literal with IDB-predicate. Control jumps to this code piece when a new fact for this predicate was derived. The argument values of the fact are stored in the above variables. The purpose of the code is to check whether new facts can be derived with this rule with the given instantiation of the IDB body literal, and if yes, to store the arguments of the derived fact in the corresponding variables and to jump to every place where the newly derived fact is used. For the example rule, the code looks as shown in Figure 3 (there are in fact many optimization possibilities, which we cannot discuss here for space reasons). Rules with only EDB-predicates in the body act as starting points. For instance, consider

$$\texttt{q(X, Y)} \leftarrow \texttt{s(X, Y)} \wedge \texttt{Y} \geq \texttt{0.}$$

```
q:  r_cursor_bff r1;
    r1.open(q_2); // Assuming index on first arg
    while(r1.fetch()) {
        if(r1.col_2() == 5) {
            p_1 = q_1;
            p_2 = r1.col_3();
            p_3 = 2;
            if(r1.fetch()) {
                push_int(q_1);
                push_int(q_2);
                r1.push();
                push_cont(CONT_q);
            }
            goto p;
        }
    }
    goto backtrack; // if this is the last place where q is used
cont_q:
    r1.pop();
    q_2 = pop_int();
    q_1 = pop_int();
    do {
        if(r1.col_2() == 5) {
            p_1 = q_1;
            p_2 = r1.col_3();
            p_3 = 2;
            if(r1.fetch()) {
                push_int(q_1);
                push_int(q_2);
                r1.push();
                push_cont(CONT_q);
            }
            goto p;
        }
    } while(r1.fetch());
    goto backtrack; // if this is the last place where q is used
backtrack:
    if(stack_empty()) return false;
    switch(pop_task()) {
    case CONT_q; goto cont_q;
    ...
    }
```

Figure 3: Push-Method: `p(X, Z, 2)` $\leftarrow$ `q(X, Y)` $\wedge$ `r(Y, 5, Z)`.

```
init:
    s_cursor s1;
    s1.open();
    while(s1.fetch()) {
        if(s1.col_2() >= 0) {
            q_1 = s1.col_1();
            q_2 = s1.col_2();
            if(s1.fetch()) {
                s1.push();
                push_cont(CONT_init);
            }
            goto q;
        }
    }
    return false; // if this is the last/only initialization rule
cont_init:
    s1.pop();
    do {
        if(s1.col_2() >= 0) {
            q_1 = s1.col_1();
            q_2 = s1.col_2();
            if(s1.fetch()) {
                s1.push();
                push_cont(CONT_init);
            }
            goto q;
        }
    } while(s1.fetch());
    return false; // if this is the last/only initialization rule
```

Figure 4: Push-Method: Initialization with `q(X, Y)` $\leftarrow$ `s(X, Y)` $\wedge$ `Y` $\geq$ `0`.

Then for every `s`-fact, we would fill the variables `q_1` and `q_2` and jump to the place where `q`-facts are used (label `q:`). Again, this loop is implemented with backtracking.
Note that the push method can be made to fit into the cursor interface: If for instance `p` needs to be queried with a cursor, the above code is inside the `fetch` method. The code jumps to label `p:` when a new `p`-fact is derived, the `fetch`-method returns `true` to the caller. Each call to the `fetch`-method starts at the label `backtrack`. In the `open`-method the backtrack-stack is initialized with a value that causes a jump to the initialization (`init`).

6. Pull-Method with Passing of Bindings

In deductive databases, one normally uses first a program transformation like magic sets, which is responsible for passing bindings from the caller to the callee, so that only relevant facts are computed when the transformed rules are evaluated strictly bottom-up (i.e. the entire minimal model of the transformed program is computed). The "magic predicates" contain values for the input/bound arguments of a predicate. Calls to these

predicates are added as conditions to the rule body, so that the rule can only "fire" when the result is needed.

With the magic set transformation, all calls to a predicate are put together in one set. This is good, if there are several calls to a complex predicate with the same input values: Then the answer is computed only once. But it is also bad, because the results for different calls to a predicate are all in one set, from which one has to select the result matching the current input arguments.

The Pull Method works already not completely bottom-up, but is controlled from the caller (top-down) who requests the next tuple. Therefore it is very natural that the caller passes all information he has about the required tuples. This would replace the magic set transformation, but it is not exactly the same, because now there is not one big magic set for a predicate (and a binding pattern), but for each call there are given values for certain arguments. This has positive and negative effects: When the call returns, one gets a tuple with the required values in the given positions, so no further check/selection is necessary. This is especially important since the pull method repeats computations, so non-matching tuples would simply be wasted. On the negative side, if the same call appears more than once, the result is computed repeatedly.

Passing bindings to called predicates fits nicely into the cursor interface, because for EDB-predicates with special access structures, it is already possible. In this way, this is also possible for IDB predicates.

For the materialization method and the push method, magic sets (or one of its variants) works well. However, if one wants to combine the different methods in one program (which is advisable, since each has its strengths and weaknesses), it would be possible to treat the magic set specially and to initialize it each time with only a single tuple.

7. Related Work

There are still more variants of bottom-up evaluation proposed in the literature, which we intend to include in our comparison in a future version of this article:

- In [Liu03], an extreme form of materialization is proposed: Not only facts about the derived predicates are explicitly stored, but also intermediate results during rule evaluation. This is combined with a clever selection of data structures.
- In [Wun95], a method similar to the push method is used, but with a materialization of the derived predicates (our push method avoids this). Similar methods are also used to propagate changes from base relations to materialized views.
- In [Cod99], bottom-up evaluation is implemented with a meta-interpreter running in Prolog. An important idea is also how to handle rules where the body of one rule is prefix of a body of another rule (as generated by the magic set method).

8. Conclusions

Our long-term goal is to develop a deductive database system that supports stepwise migration from classical SQL. Of course, the system will use our SLDmagic method [Bra00] for goal-direction, but it also needs an efficient bottom-up engine to run the transformed program. In this paper, we investigated several implementation variants based on a translation to C++. In summary, the methods differ in what they materialize (store in memory for a longer time), what they recompute, and the order in which applicable rule instances are

considered (and also the duration: the pull method has a rule instance "open" for a longer time). It turned out that the optimal method depends on the input program and also on the compiler and the hardware, as well as keys and access structures for the relations. But the push method performed constantly quite well. It has the restriction that it can work directly only with rules having only a single IDB-literal in the body, but it can be combined with other methods, or applied in several steps with different components of the program. Also, the SLDmagic method produces rules with only one IDB-predicate in the body.

The source code and performance results for the tests are available at

`http://www.informatik.uni-halle.de/~brass/botup/`

Results of future tests will also be posted on this page.

References

[Ban86a] Francois Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D. Ullman. Magic sets and other strange ways to implement logic programs. In *Proc. of the 5th ACM Symp. on Principles of Database Systems (PODS'86)*, pp. 1–15. ACM Press, 1986.

[Ban86b] Francois Bancilhon and Raghu Ramakrishnan. An amateur's introduction to recursive query processing. In Carlo Zaniolo (ed.), *Proceedings of the 1986 ACM SIGMOD international conference on Management of Data (SIGMOD'86)*, pp. 16–52. 1986.

[Bee91] Catril Beeri and Raghu Ramakrishnan. On the power of magic. *The Journal of Logic Programming*, 10:255–299, 1991.

[Bra00] Stefan Brass. SLDMagic — the real magic (with applications to web queries). In W. Lloyd et al. (eds.), *First International Conference on Computational Logic (CL'2000/DOOD'2000)*, no. 1861 in LNCS, pp. 1063–1077. Springer, Heidelberg, Berlin, 2000.

[Cod99] Michael Codish. Efficient goal directed bottom-up evaluation of logic programs. *Journal of Logic Programming*, 38(3):355–370, 1999.

[Fre91] Burkhard Freitag, Heribert Schütz, and Günter Specht. LOLA — A logic language for deductive databases and its implementation. In *Proc. of 2nd Int. Symp. on Database Systems for Advanced Applications (DASFAA'91)*, pp. 216–225. World Scientific, 1991.

[Fri95] Oris Friesen, Gilles Gauthier-Villars, Alexandre Lefebvre, and Laurent Vieille. Applications of deductive object-oriented databases using DEL. In Raghu Ramakrishnan (ed.), *Applications of Logic Databases*, pp. 1–22. Kluwer, 1995.

[Liu03] Yanhong A. Liu and Scott D. Stoller. From datalog rules to efficient programs with time and space guarantees. In *Proceedings of the 5th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'03)*, pp. 172–183. ACM, 2003.

[Min88] Jack Minker. Perspectives in deductive databases. *The Journal of Logic Programming*, 5:33–60, 1988.

[Ram94] Raghu Ramakrishnan, Divesh Srivastava, S. Sudarshan, and Praveen Seshadri. The CORAL deductive system. *The VLDB Journal*, 3:161–210, 1994.

[Ram95] Raghu Ramakrishnan and Jeffrey D. Ullman. A survey of deductive database systems. *The Journal of Logic Programming*, 23:125–149, 1995.

[Sag94] Konstantinos Sagonas, Terrance Swift, and David S. Warren. XSB as an efficient deductive database engine. In Richard T. Snodgrass and Marianne Winslett (eds.), *Proc. of the 1994 ACM SIGMOD Int. Conf. on Management of Data (SIGMOD'94)*, pp. 442–453. 1994.

[Ull90] Jeffery D. Ullman and Carlo Zaniolo. Deductive databases: Achievements and future directions. *ACM SIGMOD Record*, 19:75–82, 1990.

[Vag94] Jayen Vaghani, Kotagiri Ramamohanarao, David Kemp, Zoltan Somogyi, Peter J. Stuckey, Tim S. Leask, and James Harland. The Aditi deductive database system. *The VLDB Journal*, 3:245–288, 1994.

[Wun95] Jens E. Wunderwald. Memoing evaluation by source-to-source transformation. In Maurizio Proietti (ed.), *Logic Programming Synthesis and Transformation, 5th International Workshop (LOPSTR'95)*, no. 1048 in LNCS, pp. 17–32. Springer-Verlag, 1995.

Technical Communications of the International Conference on Logic Programming, 2010 (Edinburgh), pp. 54–63
http://www.floc-conference.org/ICLP-home.html

INDUCTIVE LOGIC PROGRAMMING AS ABDUCTIVE SEARCH

DOMENICO CORAPI [1] AND ALESSANDRA RUSSO [1] AND EMIL LUPU [1]

[1] Department of Computing
Imperial College London
180 Queen's Gate, SW7 2AZ
London, UK
E-mail address: {d.corapi,a.russo,e.c.lupu}@ic.ac.uk

Abstract. We present a novel approach to non-monotonic ILP and its implementation called TAL (Top-directed Abductive Learning). TAL overcomes some of the completeness problems of ILP systems based on Inverse Entailment and is the first top-down ILP system that allows background theories and hypotheses to be normal logic programs. The approach relies on mapping an ILP problem into an equivalent ALP one. This enables the use of established ALP proof procedures and the specification of richer language bias with integrity constraints. The mapping provides a principled search space for an ILP problem, over which an abductive search is used to compute inductive solutions.

Introduction

Inductive Logic Programming (ILP) [Lav94] is a machine learning technique concerned with the induction of logic theories from positive and negative examples and has been successfully applied to a wide range of problems [Dŏŏ]. Its main virtue, the highly expressive representation language, is also the cause of its high computational complexity. Some ILP systems attempt to efficiently find a less then perfect hypothesis by using heuristics to navigate the search space effectively [Qui96], [Ric95]. Others focus on completeness and aim for perfect accuracy with respect to the examples, searching the space thoroughly for an optimal solution. Among these XHAIL [Ray09a] has identified Abductive Logic Programming (ALP) [Kak92] as a means to deal with incomplete theories and provide semantics for negation as failure (NAF) [Cla77]. XHAIL, like other *inverse entailment* (IE) based systems, abductively derives a lower bound for the search space that is then generalised. In contrast, *Top-down* ILP systems like [Mug08, Bra99, Bos94] construct the hypothesis by specialising an overly general theory without a lower bound. However existing top-down systems limit the expressiveness of the language and the possible outcome of the learning (e.g. concepts learned must be observed in the training data, recursion is not allowed and the use of negation is limited).

Abductive proof procedures have been extensively employed as part of ILP systems (e.g. [Esp00]) or extended for inductive reasoning (e.g. [Adé95]). In contrast to these existing approaches, we propose a novel mechanism that *maps an ILP problem into an equivalent ALP*

Key words and phrases: Inductive Logic Programming, Abductive Logic Programming, Non-monotonic Reasoning.

Technical Communications of the 26th International Conference on Logic Programming, Edinburgh, July, 2010
Editors: Manuel Hermenegildo, Torsten Schaub
LIPIcs - Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/LIPIcs.ICLP.2010.54

instance. An ILP task is thus translated into an ALP problem whose solution is translated back into a solution of the original problem. The resulting top-down ILP system, called TAL (Top-directed Abductive Learning), offers several advantages over existing techniques. TAL is able to handle negation within the learning process and is able to learn non-monotonic hypotheses, relying on the semantics of the underlying abductive proof procedure employed; allows expressive language bias specifications that subsume mode declarations and can be combined with integrity constraints; performs non-observational [Moy03] and multiple predicate learning [Mal98]; and makes use of constraint solving techniques. Non-monotonic ILP has been successfully applied to bioinformatics [Ray08] and requirement engineering [Alr09] and as showed in [Cor09] and [Ray09b] can also be employed to perform theory revision.

In the particular case of definite theories, the TAL search space includes hypotheses that are not found by established Inverse Entailment based systems like PROGOL [Mug95] or ALECTO [Moy03] and provides a more effective solution to learning interdependent concepts compared to the state of art ILP systems, e.g. [Kim09, Ray09a]. Though not explored in depth here, the principled search space characterised by ALP includes abductive solutions that represent partial inductive solutions, which can be measured in terms of some scoring function (e.g. accuracy on the given set of examples) thereby enabling the use of heuristic based search strategies.

The paper is organised as follows. First, we introduce the notation and relevant background concepts. We then describe the representation underlying the learning system and discuss the learning mechanism. Then, we present through examples some of the main features of the system, and discuss related work. We conclude with final remarks and directions for future work.

1. Abductive and Inductive Logic Programming

ALP and ILP are extensions of logic programming. They both search for a hypothesis that is able to account for some given evidence. ALP constructs hypotheses in the form of ground facts. ILP systems generate rules that are able to discriminate between positive and negative examples that represent the training data. In general, ILP is regarded as a machine learning technique and used when a certain knowledge base must be enriched with rules that are also able to classify new examples. We assume in the following that the reader is familiar with first-order logic and logic programming [Llo87]. Following Prolog [Sha94] conventions, predicates, terms and functions are represented with an initial lower case letter and variables are represented with an initial capital letter.

Definition 1.1. An *ALP task* is defined as $\langle g, T, A, I\rangle$ where T is a normal logic program, A is a set of *abducible* facts, I is a set of *integrity constraints* and g is a ground goal. $\Delta \in ALP\langle g, T, A, I\rangle$ is an abductive solution for the ALP task $\langle g, T, A, I\rangle$, if $\Delta \subseteq A$, $T \cup \Delta$ is consistent, $T \cup \Delta \models g$ and $T \cup \Delta \models I$. $ALP\langle g, T, A, I\rangle$ denotes the set of all abductive solutions for the given ALP task.

Note that the abductive task, as defined, deals with ground goals, thus being a specific case of the setting in [Kak92]. The notion of entailment is not fixed since, as discussed later, the approach proposed in this paper is not committed to a particular semantics.

Definition 1.2. An *ILP task* is defined as $\langle E, B, S\rangle$ where E is a set of ground positive or negative literals, called *examples*, B is a *background theory* and S is a set of clauses called *language bias*. The theory $H \in ILP\langle E, B, S\rangle$, called *hypothesis*, is an inductive solution

for the task $\langle E, B, S\rangle$, if $H \subseteq S$, H is consistent with B and $B \cup H \models E$. $ILP\langle E, B, S\rangle$ denotes the set of all inductive solutions for the given task.

We consider the case where B and H are normal logic programs and E is a set of ground literals (with positive and negative ground literals representing positive and negative examples, respectively).

The space of possible solutions is inherently large for all meaningful applications so different levels of constraints are imposed to restrict the search for hypotheses.When possible, besides the background knowledge about the modelled world, some *a priori* knowledge on the structure of the hypothesis can be employed to impose an instance-specific *language bias* S. Mode declarations are a common tool to specify a language bias.

Definition 1.3. A *mode declaration* is either a head or body declaration, respectively $modeh(s)$ and $modeb(s)$ where s is called a *schema*. A schema s is a ground literal containing placemarkers. A *placemarker* is either '$+type$' (input), '$-type$' (output), '$\#type$' (ground) where *type* is a constant.

Given a schema s, s^* is the literal obtained from s by replacing all placemarkers with different variables $X_1, ..., X_n$; $\texttt{type}(\texttt{s}^*)$ denotes the conjunction of literals $t_1(X_1), ..., t_n(X_n)$ such that t_i is the type of the placemarker replaced by the variable X_i; $\texttt{ground}(\texttt{s}^*)$ is the list of the variables that replace the ground placemarkers in s, listed in order of appearance left to right. Similarly $\texttt{inputs}(\texttt{s}^*)$ and $\texttt{outputs}(\texttt{s}^*)$ are, respectively, the lists of the variables replacing input and output placemarkers in s. For example, for mode declaration $m3$ in Sec. 3, $s = even(+nat)$, $s^* = even(X)$, $\texttt{type}(\texttt{s}^*) = nat(X)$, $\texttt{outputs}(\texttt{s}^*) = \texttt{ground}(\texttt{s}^*) = []$, $\texttt{inputs}(\texttt{s}^*) = [X]$.

A rule r is *compatible* with a set M of mode declarations iff (a) there is a mapping from each head/body literal l in r to a corresponding head/body declaration $m \in M$ with schema s and l is subsumed by s^*; (b) each output placemarker is bound to an *output variable*; (c) each input placemarker is bound to an output variable appearing in the body before or to a variable in the head; (d) each ground placemarker is bound to a ground term; (e) all variables and terms are of the corresponding type. The use of head variables in output mode declarations is not discussed here for space constraints.

In the next sections the language bias S is specified in terms of a set M of mode declarations, and denoted as $s(M)$. Each mode declaration $m \in M$ is uniquely identified by a label id_m, called *mode declaration identifier*.

2. TAL

An ILP problem can be seen as a search for a hypothesis where we choose how many rules to use and for each rule which predicates to use in the head and in each of the body conditions and how many body conditions are used. Additionally, different choices are possible on how arguments are unified or grounded. In this section, we present the mapping of a set M of mode declarations into a *top theory* $\top$ that constrains the search by imposing a generality upper bound on the inductive solution. An abductive proof procedure is instantiated on this top theory together with the background theory. The abductive derivation identifies the heads of the rules (of a hypothesis solution) and the conditions needed to cover positive examples and exclude negative examples, ensuring consistency. The abductive solution is guaranteed to have a corresponding inductive hypothesis H that is a solution with respect to the examples.

2.1. Hypothesis representation

We use a list-based representation to encode an inductive solution, as a set of rules, into a set of facts by mapping each literal in the clauses into instances of its corresponding mode declaration. This allows the representation of rules as abducible facts. An inductive hypothesis $H \subseteq s(M)$ is composed of a set of rules $\{r_1, ..., r_n\}$. Each rule r is associated with an *output list*, i.e. an ordered list of the variables in r that replace output placemarkers in body literals or input placemarkers in the head. The *output identifier* associated with each of these variables is the position of the variable in the output list. Given a set M of mode declarations, each rule r of the form $l_1 \leftarrow l_2, ..., l_n$, compatible with M, can be represented as an ordered list $l = [\mathtt{l_1}, \mathtt{l_2}, ..., \mathtt{l_n}]$ where each $\mathtt{l_i}$ is a tuple $(id_m, [c_1, ..., c_p], [o_1, ..., o_q])$; id_m is the identifier of the mode declaration in M that l_i maps to; each c_j is a ground term replacing a ground placemarker in the mode declaration identified by id_m; each o_k is an output identifier and encodes the fact that the k^{th} input variable in l_i (in order of appearance left to right) unifies with the variable indicated by o_k. We refer to this transformation of a rule r into a list l as the function $l = t_M(r)$.

Example 2.1. Given the following three mode declarations $M = \{m1 : modeh(p(+any)),$ $m2 : modeb(q(+any, \#any)),$ $m3 : modeb(q(+any, -any))\}$ the rule $r = p(X) \leftarrow q(X, Y), q(Y, a)$, compatible with M, is associated to the output list $[X, Y]$ where X replaces the input placemarker in $m1$ and Y replaces the output placemarker in $m3$. The output identifier of X is the integer 1 and the output identifier of Y is 2. r is represented as the list $l = [(m1, [], []), (m3, [], [1]), (m2, [a], [2])] = t_M(r)$. The first element of the list associates the head $p(X)$ to mode declaration $m1$. The second element associates the condition $q(X, Y)$ to mode declaration $m3$ and links the first and only input variable to X. The third element of the list associates the condition $q(Y, a)$ to mode declaration $m2$, links the input variable to Y and instantiates the second argument to a.

It is easy to see that every possible rule within $s(M)$ can be encoded according to this representation. Also, it is always possible to derive a unique rule r from a well-formed list l. We refer to this transformation as $r = {t_M}^{-1}(l)$.

Rules in a final inductive hypothesis theory H, are associated with a unique *rule identifier* r_{id}, an integer from 1 to the maximum number, MNR, of rules allowed in H. The abductive representation $\Delta = T_M(H)$ of an hypothesis theory H, is the set of facts $rule(r_{id}, l) \in \Delta$, one for each rule $r \in H$, such that (a) r_{id} is the rule identifier of r; and (b) $l = t_M(r)$. The inverse transformation $H = {T_M}^{-1}(\Delta)$ is similarly defined.

2.2. Mapping mode declarations into an abductive top theory

The first computational step in TAL is the generation of a top theory $\top$ from a set M of mode declarations, as defined below, where *prule* and *rule* are abducible predicates.

Definition 2.2. Given a set M of mode declarations, $\top = f(M)$ is constructed as follows:

- For each head declaration $modeh(s)$, with unique identifier id_m, the following clause is in $\top$

$$s^* \leftarrow \mathtt{type}(\mathtt{s}^*), prule(RId, [id_m, \mathtt{ground}(\mathtt{s}^*), []]), rule\_id(RId), \\ body(RId, \mathtt{inputs}(\mathtt{s}^*), [id_m, \mathtt{ground}(\mathtt{s}^*), []]) \tag{2.1}$$

- The following clause is in $\top$

$$body(RId, \_, Rule) \leftarrow rule(RId, Rule) \tag{2.2}$$

- For each body declaration $modeb(s)$, with identifier id_m the following clause is in $\top$

$$body(RId, Inputs, Rule) \leftarrow append(Rule, [(id_m, \texttt{ground}(\texttt{s}^*), Links)], NRule), \\ prule(RId, NRule), link\_variables(inputs(s^*), Inputs, Links), s^*, \\ \texttt{type}(\texttt{s}^*), append(Inputs, \texttt{outputs}(\texttt{s}^*), Outputs), body(RId, Outputs, NRule) \tag{2.3}$$

As previously defined, s^* contains newly defined variables instead of placemarkers in the schema s. Since the abductive derivation is instantiated on $B \cup \top$, for all predicates appearing in head mode declarations the procedure can both use the partial definition in B or learn a new clause, unifying the current goal with s^*. s^* can also be a negative condition corresponding to a body mode declaration whose schema is a negative literal. $rule\_id(RId)$ is true whenever $1 \leq RId \leq MNR$.

$body(r, i, c)$ keeps track of the rules that are built and provides the choice of extending a partial rule (rule (2.3)) or delivering a rule as final (rule (2.2)). The first argument is the rule identifier; the second is the list of the outputs collected from the literals already added to the rule; the third is the list representing a partial rule. $link\_variables([a_1, ..., a_m], [b_1, ..., b_n], [o_1, ..., o_m])$ succeeds if for each element in the first list a_i, there exist an element in the second list b_j such that a_i unifies with b_j and $o_i = j$. $append(l_1, l_2, l_3)$ has the standard Prolog definition [Sha94]. The abducible *prule* is used to control the search through integrity constraints. For example, if we are not interested in searching for rules in which two mode declarations i and j appear together in the body, then an integrity constraint of the type $\leftarrow prule(\_, R1), member((i, \_, \_), R1), member((j, \_, \_), R1)$ can be added to I to prune such solutions in the abductive search.

In order to maintain the well-formedness of our rule's encoding and avoid trivial states of the search, a set of fixed integrity constraints I_f (omitted for brevity) is used in the abductive search.

2.3. Learning

Definition 2.3. Given an ILP task $\langle E, B, M \rangle$, $H = {T_M}^{-1}(\Delta)$ is an inductive solution derived by TAL iff $\Delta \in ALP\langle g, B \cup \top, I, A \rangle$ where $\top = f(M)$, g is the negated conjunction of the literals in E, I is a set of integrity constraints that includes I_f and $A = \{rule/2, prule/2\}$

Procedurally, an initial translation produces the ALP task introducing the new theory $\top$. An abductive proof procedure derives the abductive hypothesis Δ that is then transformed into the final inductive solution.

Theorem 2.4. *Let us consider a theory B, a set M of mode declarations, a conjunction of (possibly negated) literals E, and a set H of rules, such that $H \subseteq s(M)$. Then $B \cup H \vdash_{SLDNF} E$ iff $B \cup \top \cup \Delta \vdash_{SLDNF} E$, where $\top = f(M)$ and $\Delta = T_M(H)$*

As corollaries of Theorem (2.4), it is possible to establish soundness and completeness of our TAL system based on the properties of the underlying abductive system and on the soundness and completeness of SLDNF w.r.t. the semantics.

3. Example

The following is a modified version of the well-known example proposed in [Yam97], where even and odd numbers are both target concepts and learnt from three examples of odd numbers. The *even* predicate is partially defined, i.e. base case $even(0)$.

$even(X) \leftarrow prule(RId, [(m1, [], []]),$
$\quad nat(X), rule\_id(RId),$
$\quad body(RId, [X], [(m1, [], []])$

$odd(X) \leftarrow prule(RId, [(m2, [], []]),$
$\quad nat(X), rule\_id(RId),$
$\quad body(RId, [X], [(m2, [], []])$

$body(RId, \_, Rule) \leftarrow rule(RId, Rule)$

$body(RId, Inputs, Rule) \leftarrow$
$\quad append(Rule, [(m3, [], Links)], NRule),$
$\quad prule(RId, NRule),$
$\quad link\_variables(X, Inputs, Links),$
$\quad even(X), nat(X), body(RId, Inputs, NRule)$

$body(RId, Inputs, Rule) \leftarrow$
$\quad append(Rule, [(m4, [], Links)], NRule),$
$\quad prule(RId, NRule),$
$\quad link\_variables(X, Inputs, Links),$
$\quad odd(X), nat(X), body(RId, Inputs, NRule)$

$body(RId, Inputs, Rule) \leftarrow$
$\quad append(Rule, [(m5, [], Links)], NRule),$
$\quad prule(RId, NRule),$
$\quad link\_variables(X, Inputs, Links), X = s(Y),$
$\quad nat(X), nat(Y), append(Inputs, [Y], Outputs),$
$\quad body(RId, Outputs, NRule)$

Figure 1: Top theory $\top$ for the even-odd example.

$$B = \left\{ \begin{array}{l} even(0) \\ nat(0) \\ nat(s(X)) \leftarrow nat(X) \end{array} \right. \qquad M = \left\{ \begin{array}{l} m1 : modeh(even(+nat)) \\ m2 : modeh(odd(+nat)) \\ m3 : modeb(even(+nat)) \\ m4 : modeb(odd(+nat)) \\ m5 : modeb(+nat = s(-nat)) \end{array} \right. \qquad E = \left\{ \begin{array}{l} odd(s(s(s(s(s(0)))))) \\ not\ odd(s(s(0))) \\ not\ odd(s(s(s(s(0))))) \end{array} \right.$$

We assume the set I' of integrity constraints to restrict the language bias, which establishes that rules whose head is $even(X)$ or $odd(X)$ cannot have in the body $even(X)$ or $odd(X)$ literals. The final I is the union of I' and I_f. The set M of mode declarations is transformed into the top theory $\top$ given in Figure 1. The instantiated abductive task $\langle E, B \cup \top, I, \{rule/2, prule/2\}\rangle$ accepts then as a possible solution the set Δ translated into the inductive hypothesis H as follows:[1]

$$\Delta = \left\{ \begin{array}{l} rule(1, [(m2, [], []), (m5, [], [1]), (m3, [], [2])]) \\ rule(2, [(m1, [], []), (m5, [], [1]), (m4, [], [2])]) \end{array} \right. \qquad H = \left\{ \begin{array}{l} odd(X) \leftarrow X = s(Y), even(Y) \\ even(X) \leftarrow X = s(Y), odd(Y) \end{array} \right.$$

In the abductive search, the standard Prolog selection rule is adopted that selects clauses in order of appearance in the program. Since no head of clause in B unifies with the positive examples, the derivation uses one of the rules defined in $\top$. The selection of the *body* literal from the rule results in four derivation branches in the search tree, one for each of the four "body" clauses whose head unifies with it. A partial abductive hypothesis is generated, equivalent to the rule $odd(X) \leftarrow X = s(Y), even(Y)$. At this point, the condition $even(s(s(s(s(0)))))$, part of the current goal, is not entailed by B so one of the rules in $\top$ is used. It can be seen as an "artificial" example conditional to the partial hypothesis. The derivation results in the creation of an additional rule in the final hypothesis that defines the predicate *even*. The computation continues, thus excluding inconsistent hypotheses and those that entail also negative examples resulting in the final Δ. Partial rules are derived and used throughout the search so they can be referenced to define concepts that depend on them. It is also interesting to observe that the search is guided by the examples and thus only significant solutions are explored. The inductive solution H for this inductive problem is either not found by other ILP systems like PROGOL, or derived after a "blind"

[1] *prule* abducibles are omitted for brevity.

search as discussed in Sec. 5. The learning is non-observational (i.e. the *even* predicate is not observed in the examples). TAL is also able to learn the base case of the recursion. If the fact $even(0)$ is deleted from B and the mode declaration $modeh(even(\#nat))$ is added to M, TAL returns three solutions with the same set of examples: the first has the same definition of odd as in H and defines $even(s(s(s(s(s(0))))))$ as base case, the second and the third are the same as in H with $even(s(s(0)))$ and $even(0)$ respectively as base cases.

4. A case study

We employ a case study to compare TAL with the only other system capable of solving the same class of ILP problems XHAIL[2]. The following case study, taken from [Ray09a], represents a simple model of metabolic regulation for the bacterium E. coli and includes a formulation of the Event Calculus [Sha99] formalism. The target predicate *happens* is used to characterise the bacterium feeding mechanism based on the availability of sugar. See [Ray09a] for a more extensive explanation of the example.

$$B = \left\{ \begin{array}{l} \text{[Type definitions and Event Calculus axioms]} \\ initiates(add(G), available(G), T) \leftarrow sugar(G), timex(T) \\ terminates(use(G), available(G), T) \leftarrow sugar(G), timex(T) \\ happens(add(lactose), 0) \\ happens(add(glucose), 0) \end{array} \right. \quad I = \left\{ \begin{array}{l} \leftarrow happens(use(G), T), \\ not\ holdsAt(available(G), T) \end{array} \right.$$

$$E = \left\{ \begin{array}{l} holdsAt(available(lactose), 1) \\ holdsAt(available(lactose), 2) \\ not\ holdsAt(available(lactose), 3) \end{array} \right. \qquad M = \left\{ \begin{array}{l} m1 : modeh(happens(use(\#sugar), +timex)) \\ m2 : modeb(holdsAt(\#fluent, +timex)) \\ m3 : modeb(not\ holdsAt(\#fluent, +timex)) \end{array} \right.$$

The transformations in Definition 2.2 are applied to the given ILP instance. The abductive solution for the corresponding ALP problem is:

$$\Delta = \left\{ \begin{array}{l} rule(1, [(m1, [glucose], []), (m2, [available(glucose)], [1])]) \\ rule(2, [(m1, [lactose], []), (m2, [available(lactose)], [1]), (m3, [available(glucose)], [1])]) \end{array} \right.$$

equivalent to the inductive hypothesis:

$$H = \left\{ \begin{array}{l} happens(use(glucose), T) \leftarrow holdsAt(available(glucose), T) \\ happens(use(lactose), T) \leftarrow holdsAt(available(lactose), T), not\ holdsAt(available(glucose), T) \end{array} \right.$$

As discussed in [Ray09a], XHAIL generates *Kernel Sets* that serve as lower bound for the final hypothesis, through iterations of increasing in size abductive explanations until a satisfactory solution is found. Intuitively, a Kernel Set is computed in two phases. A first abductive phase finds the set Δ of the head of the rules in the Kernel Set and a second deductive phase constructs the body of the rules by computing all the ground instantiations of the body mode declarations that are implied by $B \cup \Delta$. Kernel Sets are generalised in a final inductive phase. Instead, TAL explores candidate solutions in a top-down manner, backtracking whenever the current solution leads to failure in the abductive derivation. The

[2]Relying on the results reported in [Ray09a], the computation time for this study appears to differ by one order of magnitude. [Ray09a] reports that a prototype XHAIL implementation took a couple of seconds to compute H on a 1.66 GHz Centrino Duo Laptop PC with 1 GB of RAM, while TAL took 30 ms to find H and 180 ms to explore the whole space of hypotheses limited to two clauses with at most two conditions on a 2.8 GHz Intel Core 2 Duo iMac with 2 GB of RAM. Unfortunately, XHAIL is not publicly available so we are not able to perform an empirical comparison of the performance of the two systems.

partial hypotheses are already in their final form and are implicitly tested for correctness whenever a new example is selected in the abductive derivation.

5. Discussion and related work

We have implemented TAL in YAP Prolog [Cos08] using a customised implementation of the ASYSTEM [Kak01] that integrates the SLDNFA proof procedure with constraint solving techniques. TAL has been tested on various non-monotonic ILP problems like the examples proposed in [Kim09], [Alr09] and [Ray07]. It has also been used to perform Theory Revision [Wro96], i.e. to change, according to some notion of minimality, an existing theory [Cor09]. This work is is motivated by the project [Ban08] that seeks to exploit the proposed approach in the context of learning privacy policies from usage logs. We performed preliminary experiments in this direction applying an extended version of TAL to the Reality Mining dataset [Eag06] where examples of refused calls were used to learn general rules. A score based on accuracy and complexity of the rules was employed to prune the search space.

The idea of a top theory as bias for the learning has been initially introduced in TOPLOG [Mug08], which performs deductive reasoning on the background knowledge extended with the top theory. Candidate hypotheses are derived from single positive examples and then the best ones are selected after a hill climbing search. SPECTRE [Bos94] also requires a user-provided overly general theory that is specialised by unfolding clauses until no negative examples are covered. HYPER [Bra99], specialises an overly general theory, deriving rules that are subsumed by those in the theory. Thus the number of rules in the final hypotheses cannot increase. FOIL and related approaches like [Coh94] perform an informed hill climbing search. These systems are not fully non-monotonic since they disallow negation in the background knowledge or in the hypotheses. In contrast to TOPLOG, TAL generates candidate hypotheses also considering negative examples, excluding *a priori* solutions that entail some of the negative examples. In general, we regard TAL a generalisation of other top-down ILP systems. Constraining it to consider only positive examples, the background theory and mode declarations being definite, would result in the same rule generation mechanism as TOPLOG. Different search strategies can be easily implemented by modifying the abductive procedure. Partial solutions can be associated with a score with respect to the examples (e.g. the sum of entailed examples over the total). This would enable the use of informed search techniques and strategies like, for instance, hill climbing or beam search that can be used to prune the space, exploring the most promising solutions. Similarly to TOPLOG [Mug08], our approach can also be applied directly to a grammar based language bias specification, instead of generating the top theory from mode declarations. Systems based on Inverse Entailment (IE), compute a bottom clause, or set of clauses (e.g. the Kernel Set) that constrains the search space from the bottom of the generality lattice. For problems dealing with definite theories our system manages to solve a wider class of problems than PROGOL, since one single example can generate more than one rule in H. IMPARO [Kim09] solves a class of problems whose solutions are not found by other IE based systems, namely connected theories where body conditions are abductively proved from the background theory. These problems, that are solved in Imparo by applying *Induction on Failure* (IoF), can also be solved by TAL, as shown in the example given in this paper. The IoF mechanism is in our system embedded in the abductive search that always includes in the search space the generation of a new rule whenever a condition is not entailed by the current theory. XHAIL can find hypotheses computed under IoF by exploring non-minimal

abductive explanations but the search is not guided by the background theory and a partial hypothesis[3]. This highlights another advantage of TAL: the computation of clause heads in the hypothesis is naturally interleaved with the generation of the body and it does not take place in a separate phase as in IMPARO and XHAIL. Moreover, all rules are constructed concurrently and their partial definitions can be used. [Kak00] propose a system for inductive learning of logic programs that compared to TAL is limited to observational predicate learning. Finally, [Adé95] introduces induction in the abductive SLDNFA procedure, defining an extended proof procedure called SLDNFAI. In contrast, TAL defines a general methodology and does not commit to a particular proof procedure. Moreover SLDNFAI does not allow a fine specification of the language bias, makes no use of constraints on the generated hypotheses and is limited to function-free definite clauses.

6. Conclusions and further work

We have presented a novel approach to non-monotonic ILP that relies on the transformation of an ILP task into an equivalent ALP task. We showed through an example how the approach is able to perform non-observational and multi-predicate learning of normal logic programs by means of a top-down search guided by the examples and abductive integrity constraints where a partial hypothesis is used in the derivation of new rules. In contrast, techniques based on IE perform a blind search or are not able to derive a solution. The mapping into ALP offers several advantages. Soundness and completeness can be established on the basis of the abductive proof procedure employed. Constraint solving techniques and optimised ALP implementations can be used and abductive integrity constraints on the structure of the rule can be employed. Furthermore, the search space makes use of partial hypotheses that allows the use of informed search techniques, thus providing a general framework that can scale to learning problems with large datasets and theories. We obtained promising result in this direction and we are currently evaluating the use of heuristics and informed search techniques. We plan to investigate the properties of the mapping and the relationships with the search space of other ILP techniques.

Acknowledgements

The authors are grateful to Krysia Broda, Robert Craven, Tim Kimber, Jiefei Ma, Oliver Ray and Daniel Sykes for their useful discussions. This work is funded by the UK EPSRC (EP/F023294/1) and supported by IBM Research as part of their Open Collaborative Research (OCR) initiative.

References

[Adé95] Hilde Adé and Marc Denecker. Ailp: Abductive inductive logic programming. In *IJCAI*, pp. 1201–1209. 1995.

[Alr09] Dalal Alrajeh, Jeff Kramer, Alessandra Russo, and Sebastián Uchitel. Learning operational requirements from goal models. In *ICSE*, pp. 265–275. 2009.

[Ban08] A. K. Bandara, B. A. Nuseibeh, and B. A. Price et al. Privacy rights management for mobile applications. In *4th Int. Symp. on Usable Privacy and Security*. Pittsburgh, 2008.

[3]In the "odd/even" example the only way for XHAIL to find the correct solution is to extend the minimal abductive solution $odd(s(s(s(s(s(0))))))$ with $even(s(s(s(s(0)))))$ and $even(s(s(s(s(s(s(0)))))))$ that have to be somehow chosen from a set of infinite candidates.

[Bos94] H. Boström and P. Idestam-Almquist. Specialization of logic programs by pruning SLD-trees. *GMD-Studien*, vol. 237. Gesellschaft für Mathematik und Datenverarbeitung MBH, 1994.

[Bra99] Ivan Bratko. Refining complete hypotheses in ILP. In *ILP '99: 9th Workshop on Inductive Logic Programming*, pp. 44–55. Springer-Verlag, London, UK, 1999.

[Cla77] Keith L. Clark. Negation as failure. In *Logic and Data Bases*, pp. 293–322. 1977.

[Coh94] William W. Cohen. Grammatically biased learning: Learning logic programs using an explicit antecedent description language. *Artif. Intell.*, 68(2):303–366, 1994.

[Cor09] D. Corapi, O. Ray, A. Russo, A.K. Bandara, and E.C. Lupu. Learning rules from user behaviour. In *5th Artif. Intell. Applications and Innovations (AIAI 2009)*. Thessaloniki, Greece, 2009.

[Cos08] Vítor Santos Costa, Luís Damas, Rogério Reis, and Rúben Azevedo. Yap user's manual, 2008.

[DŎ0] Sašo Džroski. Relational data mining applications: an overview. pp. 339–360, 2000.

[Eag06] Nathan Eagle and Alex Pentland. Reality mining: sensing complex social systems. *Personal and Ubiquitous Computing*, 10(4):255–268, 2006.

[Esp00] Floriana Esposito, Giovanni Semeraro, Nicola Fanizzi, and Stefano Ferilli. Multistrategy theory revision: Induction and Abduction in INTHELEX. *Mach. Learn.*, 38(1-2):133–156, 2000.

[Kak92] Antonis C. Kakas, Robert A. Kowalski, and Francesca Toni. Abductive logic programming. *J. Log. Comput.*, 2(6):719–770, 1992.

[Kak00] Antonis C. Kakas and Fabrizio Riguzzi. Abductive concept learning. *New Generation Comput.*, 18(3):243–, 2000.

[Kak01] C. Kakas, Antonis, Bert Van Nuffelen, and Marc Denecker. A-system : Problem solving through abduction. In *17th International Joint Conference on Artif. Intell.*, vol. 1, pp. 591–596. IJCAI, inc and AAAI, Morgan Kaufmann Publishers, Inc, 2001.

[Kim09] Tim Kimber, Krysia Broda, and Alessandra Russo. Induction on failure: Learning connected horn theories. In *LPNMR*, pp. 169–181. 2009.

[Lav94] N. Lavrač and S. Džeroski. *Inductive Logic Programming: Techniques and Applications*. 1994.

[Llo87] John W. Lloyd. *Foundations of Logic Programming, 2nd Edition*. Springer, 1987.

[Mal98] D. Malerba, F. Esposito, and F. A. Lisi. Learning recursive theories with ATRE. In H. Prade (ed.), *Proc. of the 13th European Conference on Artif. Intell.*, pp. 435–439. John Wiley & Sons, 1998.

[Moy03] S Moyle. *An investigation into theory completion techniques in inductive logic*. Ph.D. thesis, University of Oxford, 2003.

[Mug95] S. Muggleton. Inverse entailment and Progol. *New Generation Computing J.*, 13:245–286, 1995.

[Mug08] Stephen Muggleton, José Carlos Almeida Santos, and Alireza Tamaddoni-Nezhad. Toplog: Ilp using a logic program declarative bias. In Maria Garcia de la Banda and Enrico Pontelli (eds.), *ICLP*, *LCNS*, vol. 5366, pp. 687–692. Springer, 2008.

[Qui96] J. Ross Quinlan. Learning first-order definitions of functions. *J. Artif. Intell. Res. (JAIR)*, 5:139–161, 1996.

[Ray07] Oliver Ray. Inferring process models from temporal data with abduction and induction. In *1st Workshop on the Induction of Process Models*. 2007.

[Ray08] Oliver Ray and Chris Bryant. Inferring the function of genes from synthetic lethal mutations. In *2nd Int. Conf. on Complex, Intelligent and Software Intensive Systems*, pp. 667–671. IEEE Computer Society, 2008.

[Ray09a] Oliver Ray. Nonmonotonic abductive inductive learning. In *Journal of Applied Logic*, vol. 7, pp. 329–340. 2009.

[Ray09b] Oliver Ray, Ken Whelan, and Ross King. A nonmonotonic logical approach for modelling and revising metabolic networks. In *3rd Int. Conf. on Complex, Intelligent and Software Intensive Systems*. IEEE Computer Society, 2009.

[Ric95] Bradley L. Richards and Raymond J. Mooney. Automated refinement of first-order horn-clause domain theories. *Machine Learning*, 19(2):95–131, 1995.

[Sha94] Leon Shapiro and Ehud Y. Sterling. *The Art of PROLOG: Advanced Programming Techniques*. The MIT Press, 1994.

[Sha99] Murray Shanahan. The event calculus explained. *LNCS*, 1600, 1999.

[Wro96] Stefan Wrobel. First order theory refinement. In Luc De Raedt (ed.), *Advances in Inductive Logic Programming*, pp. 14 – 33. IOS Press, 1996.

[Yam97] Akihiro Yamamoto. Which hypotheses can be found with inverse entailment? In *ILP*, pp. 296–308. 1997.

Technical Communications of the International Conference on Logic Programming, 2010 (Edinburgh), pp. 64–73
http://www.floc-conference.org/ICLP-home.html

EFFICIENT SOLVING OF TIME-DEPENDENT ANSWER SET PROGRAMS

TIMUR FAYRUZOV [1] AND JEROEN JANSSEN [2] AND DIRK VERMEIR [2] AND CHRIS CORNELIS [1] AND MARTINE DE COCK [1,3]

[1] Dept. of Appl. Math. and Comp. Sc., Ghent University, Krijgslaan 281 (S9), 9000 Gent, Belgium
E-mail address: {timur.fayruzov,chris.cornelis}@ugent.be

[2] Dept. of Computer Science, Vrije Universiteit Brussel, Pleinlaan 2, 1050 Brussels, Belgium
E-mail address: {jeroen.janssen,dvermeir}@vub.ac.be

[3] Institute of Technology, University of Washington, 1900 Commerce St., Tacoma, WA-98402, USA
E-mail address: mdecock@u.washington.edu

Abstract. Answer set programs with time predicates are useful to model systems whose properties depend on time, like for example gene regulatory networks. A state of such a system at time point t then corresponds to the literals of an answer set that are grounded with time constant t. An important task when modelling time-dependent systems is to find steady states from which the system's behaviour does not change anymore. This task is complicated by the fact that it is typically not known in advance at what time steps these steady states occur. A brute force approach of estimating a time upper bound t_{max} and grounding and solving the program w.r.t. that upper bound leads to a suboptimal solving time when the estimate is too low or too high. In this paper we propose a more efficient algorithm for solving Markovian programs, which are time-dependent programs for which the next state depends only on the previous state. Instead of solving these Markovian programs for a long time interval $\{0, \ldots, t_{max}\}$, we successively find answer sets of parts of the grounded program. Our approach guarantees the discovery of all steady states and cycles while avoiding unnecessary extra work.

1. Introduction

Answer Set Programming (ASP) is a form of non-monotonic reasoning based on the stable-model semantics [Gel88]. The number of ASP application domains is growing fast (see e.g. [Dwo08, Tra06, Sch09]). Some of these require an adaptation of the general-purpose solving process to their specific needs to allow for faster answer set computation. One broad domain of ASP applications uses programs that depend on a parameter that bounds the size of a solution. Consider e.g. the following time-dependent answer set program, for which the

1998 ACM Subject Classification: I.2.4 [Artificial Intelligence] Relation systems; J.3 [Computer Applications]: Biology and genetics.

Key words and phrases: answer set programming, time-dependent programs, gene regulation networks.

Jeroen Janssen is funded by a joint Research Foundation-Flanders (FWO) project. Chris Cornelis is a postdoctoral fellow of the Research Foundation-Flanders (FWO).

Technical Communications of the 26th International Conference on Logic Programming, Edinburgh, July, 2010
Editors: Manuel Hermenegildo, Torsten Schaub
LIPIcs - Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/LIPIcs.ICLP.2010.64

grounding size depends on the parameter t_{max}. This program uses a non-standard notation of the form $p@T$ which is equivalent to $p(T)$. The purpose of this notation is described in Section 3.

Example 1.1. Program P consists of the following rules:

$$
\begin{array}{lll}
time(0 \dots t_{max}). & & \\
q@T & \leftarrow & p@(T-1), time(T), time(T-1). \\
v@T & \leftarrow & q@(T-1), not\ w@T, time(T), time(T-1). \\
w@T & \leftarrow & q@(T-1), not\ v@T, r(X), time(T), time(T-1). \\
p@T & \leftarrow & time(T). \\
r(str). & &
\end{array}
$$

where $time(0 \dots t_{max})$ is a shorthand for the facts $time@0, time@1, \dots, time@t_{max}$ and T is a time-bound variable. This program describes the behaviour of a system whose properties depend on time. The answer sets for this program change as the time boundary t_{max} increases. When $t_{max} = 0$ there is only one answer set[1], namely $A = \{r(str), time(0), p(0)\}$. The unique answer set for $t_{max} = 1$ is $B = A \cup \{time(1), p(1), q(1)\}$, which is twice the size of the previous one. For $t_{max} = 2$, negation as failure comes into play, resulting in two different answer sets $C = B \cup \{time(2), p(2), q(2), v(2)\}$ and $D = B \cup \{time(2), p(2), q(2), w(2)\}$. For $t_{max} = 3$ there are already four different answer sets:

$$
\begin{array}{lllllll}
E & = & C \cup \{time(3), p(3), q(3), v(3)\} & \qquad & G & = & D \cup \{time(3), p(3), q(3), w(3)\} \\
F & = & C \cup \{time(3), p(3), q(3), w(3)\} & & H & = & D \cup \{time(3), p(3), q(3), v(3)\}
\end{array}
$$

As this example illustrates, the number of answer sets as well as the size of the answer sets of a time-dependent program can increase exponentially in the time boundary.

An important task when modelling and simulating a time-dependent system is to find its steady states. Answer set E in Example 1.1 contains one steady state of the system described by program P, as $time, p, q$, and v (and no other time-dependent predicates) belong to E both for time step 2 and 3.

The main problem in finding these steady states is that it is typically not known in advance at what time steps they occur. Furthermore a system may converge to several steady states (e.g. E and F in Example 1.1) or may even oscillate among several states repeatedly (e.g a cycle between v and w in Example 1.1 that manifests itself in some of the answer sets of P for $t_{max} = 4$), and we may want to find them all. A brute force approach of estimating a time upper bound and grounding and solving the program w.r.t. that upper bound may lead to a suboptimal solving time: if the upper bound is estimated too high, the grounded program is larger than necessary to find the steady states, hence requiring unnecessary work, and if it is estimated too low, not all steady states are found, meaning the process needs to be redone for a larger estimate.

In this paper we propose a technique that allows to find all steady states and cycles efficiently. To this end, we define the notion of Markovian programs, which can be grounded for one time step at a time. We introduce a way of solving these programs by solving one-step grounded versions. These programs can be used to model protein interaction networks as described e.g. in [Fay09]. We proceed by recalling ASP notions in Section 2, formally defining time-dependent programs and Markovian programs in Section 3, and proposing a method to solve these programs efficiently in Section 4. We explain the difference with other approaches in Section 5 and finally conclude in Section 6.

[1]See Section 2 for preliminaries w.r.t. answer set programming.

2. Preliminaries

Answer set programs are built from a signature $\sigma = \langle \gamma, \upsilon, \pi \rangle$, where γ is a set of *constant symbols*, υ is a set of *variable symbols*, and $\pi = \bigcup_{i=1}^{m} \pi_i (m \in \mathbb{N})$ is the union of sets π_i of i-ary *predicate symbols*. We define a set of *variable expressions* ϵ containing expressions of the form $t' \pm t''$ where $t' \in \upsilon$ and $t'' \in \gamma$. An *atom* over σ is an object of the form $p(t_1, \ldots, t_n)$, where $p \in \pi_n$ and $t_i \in \gamma \cup \upsilon \cup \epsilon$ for each $i \in 1 \ldots n$. We implicitly assume that if t_i is a symbol starting with a capital, it denotes a variable; otherwise it is a constant. A *literal* over σ is either an atom, or an atom preceded by $\neg$, which denotes *classical negation*. *Naf-literals* over σ, denoting negation-as-failure, are of the form $\mathbf{not}\, l$, where l is a literal over σ. For a set of literals X we introduce the notation $\mathbf{not}\, X = \{\mathbf{not}\, l | l \in X\}$. For a literal or a naf-literal l, we use $\mathbf{vars}(l)$ to denote the set of variables contained in l. If $\mathbf{vars}(l) = \emptyset$ then l is called *ground*.

A *rule* r over σ is an object of the form $l_0 \leftarrow l_1, \ldots, l_m, \mathbf{not}\, l_{m+1}, \ldots, \mathbf{not}\, l_n$, where l_i, for $i \in 0 \ldots n$, are literals over σ. If each l_i is ground, it is called a *ground rule*. We refer to l_0 as the *head* of r, denoted as $head(r)$, to the set $\{l_1, \ldots, l_m, \mathbf{not}\, l_{m+1}, \ldots, \mathbf{not}\, l_n\}$ as the body of r, denoted as $body(r)$, to $\{l_1, \ldots, l_m\}$ as the positive part of the body, denoted as $pos(r)$, and to the set $\{l_{m+1}, \ldots, l_n\}$ as the negative part of the body, denoted as $neg(r)$. We denote $Lit(r) = pos(r) \cup neg(r)$. If the head of the rule is empty, the rule is called a *constraint*[2]; if the body of the rule is empty, the rule is called a *fact*.

An *answer set program* P over a signature σ is a finite set of rules over σ. If all rules in P are ground, it is called a *ground program*. The process of *grounding* constructs a ground program $Gnd(P)$ from an answer set program P over a signature σ by replacing each rule r by the set of rules obtained from r by all possible substitutions of the constants of σ for the variables in r. If any of the predicate arguments takes on a composite form $t' \pm t''$ with t', t'' grounded as numbers, they are substituted with the resulting value. A rule r that does not contain negation-as-failure, i.e. $neg(r) = \emptyset$, is called a *simple rule*. A program that contains only simple rules is called a simple program.

Turning to the semantics, a set of ground literals I over a signature σ is called an *interpretation* if it is *consistent*, i.e. there is no literal l such that both $l \in I$ and $\neg l \in I$. An interpretation I is a model of a simple rule r iff $pos(r) \not\subseteq I \vee head(r) \in I$. An interpretation that is a model of all rules of a simple program P is called a *model* of P. The minimal model of a simple program P is called an *answer set* of P. If P contains negation-as-failure, then an interpretation I of P is called an *answer set* of P iff I is the answer set of the *reduct program* P^I, where $P^I = \{head(r) \leftarrow pos(r) \mid r \in P, I \cap neg(r) = \emptyset\}$. The set of all answer sets of a program P is denoted as $AS(P)$.

3. Theoretical underpinnings

3.1. Time-dependent programs

In the remainder of this paper, we designate certain predicates as *time-dependent predicates* and denote atoms built with these predicates as $p(t_1, \ldots, t_{n-1})@\theta$ where θ is called a *time argument*. This is a convenience notation that allows to separate the (semantic)

[2]We do not consider constraints in our formal language, since a constraint $\leftarrow \beta$ can be simulated by a rule $l \leftarrow \mathbf{not}\, l, \beta$, where l is a literal not occurring in the program.

notion of time from the underlying syntactic representation. This notation is translated to a conventional atom of the form $p(t_1, \ldots, t_{n-1}, \theta)$ at grounding time.

Definition 3.1 (Time-dependent program)**.** A *time-dependent program* $\mathbf{P}$ is a tuple $\langle P, \tau \rangle$ over a signature $\sigma = \langle \gamma, \upsilon, \pi \rangle$, such that P is an answer set program over σ and $\tau \subseteq \pi$ is a set of *time-dependent predicates*. We denote the set of n-ary time-dependent predicates as τ_n. We define the set of *free time-dependent literals*

$$\mathcal{F}_{\mathbf{P}} = \bigcup \left\{ \{p(t_1, \ldots, t_{n-1})@\theta, \neg p(t_1, \ldots, t_{n-1})@\theta\} \;\middle|\; \begin{array}{l} t_1, \ldots, t_{n-1} \in \gamma \cup \upsilon \cup \epsilon, \\ \theta \in \upsilon \cup \epsilon, p \in \tau_n, 1 \leq n \leq m \end{array} \right\}$$

and the set of *bound time-dependent literals*

$$\mathcal{B}_{\mathbf{P}} = \bigcup \left\{ \{p(t_1, \ldots, t_{n-1})@\theta, \neg p(t_1, \ldots, t_{n-1})@\theta\} \;\middle|\; \begin{array}{l} t_1, \ldots, t_{n-1} \in \gamma \cup \upsilon \cup \epsilon, \\ \theta \in \gamma, p \in \tau_n, 1 \leq n \leq m \end{array} \right\}.$$

The literals from $\mathcal{F}_{\mathbf{P}}$ contain a variable or a variable expression as the time argument, while the literals from $\mathcal{B}_{\mathbf{P}}$ contain a constant as the time argument. The set of *time-dependent literals* of a time-dependent program $\mathbf{P}$ is defined as $Lit(\mathbf{P})^\tau = \mathcal{F}_{\mathbf{P}} \cup \mathcal{B}_{\mathbf{P}}$. It is a subset of the set of all literals of $\mathbf{P}$, which is defined as $Lit(\mathbf{P}) = \bigcup_{r \in P} Lit(r)$. Furthermore, for $l \in Lit(\mathbf{P})^\tau$, we use $t_{arg}(l)$ to refer to the time argument θ of l. A time-dependent program $\mathbf{P}$ is called *well-typed* iff

$$\forall r \in P \cdot (Lit(\mathbf{P})^\tau \cap (pos(r) \cup neg(r)) \neq \emptyset) \Rightarrow (head(r) \in Lit(\mathbf{P})^\tau)$$

Intuitively, if a rule in a well-typed time-dependent program contains a time-dependent literal in its body, it should contain a time-dependent literal in its head. In the remainder we will only consider well-typed time-dependent programs.

Definition 3.2 (t-grounding of a time-dependent literal)**.** Let $\mathbf{P} = \langle P, \tau \rangle$ be a time-dependent program and $t \in \mathbb{N}$. The t-grounding of a literal $l \in Lit(\mathbf{P})$, denoted as $Gnd(l)_t$, is obtained as follows: 1) if $l \in Lit(\mathbf{P}) \setminus \mathcal{F}_{\mathbf{P}}$ then $Gnd(l)_t = l$; 2) if $l \in \mathcal{F}_{\mathbf{P}}$ then the variable in $t_{arg}(l)$ is replaced by t, and in case of a variable expression the resulting value is calculated. In all cases, the obtained literal $Gnd(l)_t$ is subsequently translated to the conventional ASP notation. For a set of literals L, we define the t-grounding of this set as $Gnd(L)_t = \bigcup_{l \in L} Gnd(l)_t$, i.e. we take the pointwise t-grounding of its elements.

Example 3.3. The 2-grounding of literal $l = p(X, a)@(T + 1)$ is $Gnd(l)_2 = p(X, a, 3)$.

Definition 3.4 (t-grounding of a rule)**.** Let $\mathbf{P} = \langle P, \tau \rangle$ be a time-dependent program and $t \in \mathbb{N}$. The t-grounding of a rule $r \in P$ is defined as

$$Gnd(r)_t = Gnd(head(r))_t \leftarrow Gnd(pos(r))_t, \mathbf{not}\, Gnd(neg(r))_t$$

Definition 3.5 (t-grounding of a time-dependent program)**.** Let $\mathbf{P} = \langle P, \tau \rangle$ be a time-dependent program and $t_{max} \in \mathbb{N}$. The t_{max}-grounding of $\mathbf{P}$ is defined as

$$Gnd(\mathbf{P})_{t_{max}} = Gnd(\{Gnd(r)_{t'} \mid r \in P, t' \in \mathbb{N}, t' \leq t_{max}\})$$

Intuitively, to obtain $Gnd(\mathbf{P})_{t_{max}}$ we instantiate all time-dependent literals with a set of time points $\{t' | 0 \leq t' \leq t_{max}\}$ and then ground the resulting program in the conventional way.

Example 3.6. Let $\mathbf{P} = \langle P, \{v, w, q, p, time\} \rangle$ where P is the program from Example 1.1. Its 2-grounding $Gnd(\mathbf{P})_2$ is obtained by setting $t_{max} = 2$ and is defined as

$$
\begin{array}{lllll}
1: & time(0). & & & \\
2: & time(1). & & & \\
3: & time(2). & & & \\
4: & q(0) & \leftarrow & p(-1), time(0), time(-1). & \\
5: & q(1) & \leftarrow & p(0), time(1), time(0). & \\
6: & q(2) & \leftarrow & p(1), time(2), time(1). & \\
7: & v(0) & \leftarrow & q(-1), not\ w(0), time(0), time(-1). & \\
8: & v(1) & \leftarrow & q(0), not\ w(1), time(1), time(0). & \\
9: & v(2) & \leftarrow & q(1), not\ w(2), time(2), time(1). & \\
10: & w(0) & \leftarrow & q(-1), not\ v(0), r(str), time(0), time(-1). & \\
11: & w(1) & \leftarrow & q(0), not\ v(1), r(str), time(1), time(0). & \\
12: & w(2) & \leftarrow & q(1), not\ v(2), r(str), time(2), time(1). & \\
13: & p(0) & \leftarrow & time(0). & \\
14: & p(1) & \leftarrow & time(1). & \\
15: & p(2) & \leftarrow & time(2). & \\
16: & r(str). & & &
\end{array}
$$

Definition 3.7 (State of an answer set)**.** Let $\mathbf{P} = \langle P, \tau \rangle$ be a time-dependent program and I be an answer set of the t_{max}-grounding $Gnd(\mathbf{P})_{t_{max}}$ for $t_{max} \in \mathbb{N}$. Furthermore let $t \in \mathbb{N}$ with $t \leq t_{max}$. The *state* of I at time point t is defined as

$$I^t = \{l \mid l \in I, t_{arg}(l) = t\}$$

Intuitively, the state of answer set I of $Gnd(\mathbf{P})_{t_{max}}$ at time point t is the set of ground time-dependent literals in I that were grounded with t in the time argument. Two states are called equivalent if the only difference between literals in these states is in the values of the time points (see Example 3.12). We denote state equivalence as $I^{t'} =_{time} I^{t''}$.

Example 3.8. Consider the program $Gnd(\mathbf{P})_2$ from Example 3.6. The answer sets of this program are C and D as defined in Example 1.1. The states of answer set C at time points 0, 1 and 2 are $C^0 = \{time(0), p(0)\}$, $C^1 = \{time(1), p(1), q(1)\}$ and $C^2 = \{time(2), p(2), q(2), v(2)\}$.

Definition 3.9 (Trajectory of an answer set)**.** Let $\mathbf{P} = \langle P, \tau \rangle$ be a time-dependent program and I an answer set of the t_{max}-grounding $Gnd(\mathbf{P})_{t_{max}}$ for $t_{max} \in \mathbb{N}$. The *trajectory* of I is defined as

$$T^I = \langle I^0 \dots I^{t_{max}} \rangle$$

Example 3.10. The trajectory of answer set C of program $Gnd(\mathbf{P})_2$ from Example 3.6 is

$$T^C = \langle \{time(0), p(0)\}, \{time(1), p(1), q(1)\}, \{time(2), p(2), q(2), v(2)\} \rangle$$

Definition 3.11 (Steady state, steady cycle)**.** Let $\mathbf{P} = \langle P, \tau \rangle$ be a time-dependent program and I be an answer set of the t_{max}-grounding $Gnd(\mathbf{P})_{t_{max}}$ for $t_{max} \in \mathbb{N}$. The state of I at time point t, with $t < t_{max}$, is called a steady state iff $I^t =_{time} I^{t+1}$. The sequence $\langle I^k \rangle_{t_1 \leq k \leq t_2}$, with $t_1 \in \mathbb{N}$, $t_2 \in \mathbb{N}$ and $t_1 < t_2 \leq t_{max}$, is called a *steady cycle* iff $I^{t_1} =_{time} I^{t_2}$.

Note that to define whether a state is a steady state it is enough to check the next state, because if it does not change in the next step it will not change in the following steps as well due to the deterministic nature of the model.

Example 3.12. The 3-grounding $Gnd(\mathbf{P})_3$ of $\mathbf{P} = \langle P, \{v, w, q, p, time\} \rangle$ where P is the program from Example 1.1, has answer sets E, F, G, and H as defined in Example 1.1. The states of answer set E are $E^0 = \{time(0), p(0)\}$, $E^1 = \{time(1), p(1), q(1)\}$, $E^2 = \{time(2), p(2), q(2),\ v(2)\}$, and $E^3 = \{time(3), p(3), q(3), v(3)\}$. E^2 is a steady state, as $E^2 =_{time} E^3$.

When solving time-dependent programs, one is usually interested in finding steady states, steady cycles and trajectories leading to these states, as they can help to verify the model's correctness and/or provide new hypotheses about the behaviour of the underlying system. An important problem is that it is in general impossible to accurately estimate an upper time bound t_{max} that suffices to find all steady states. Thus, one should manually

adjust the bound and recompute answer sets over and over, which is very inefficient. In the following section we narrow down time-dependent programs to Markovian programs and propose an approach that does not require a time bound estimation for trajectory computation.

3.2. Markovian programs

In this section we define a subclass of time-dependent programs, called Markovian programs. This type of time-dependent programs is defined in such a way that every next state directly depends only on the previous state, and does not depend on any of the future states (hence the name Markovian). This is a reasonable assumption as real-world models are normally unaware of any future events and make their decisions based on the information directly available.

Recall that steady states and steady cycles for a time-dependent program $\mathbf{P}$ can be found by grounding the program for a manually chosen time upper bound t_{max} (see Definition 3.5), solving the resulting ground program $Gnd(\mathbf{P})_{t_{max}}$ to obtain its answer sets, and verifying whether the answer sets reveal steady states or cycles (see Definition 3.11). The Achilles' heel in this procedure is in the manual choice of t_{max}. Iteratively incrementing it and repeating the above process until reaching a time point t_{max} at which a steady state or cycle is encountered is inefficient, because that would require solving $Gnd(\mathbf{P})_0$, $Gnd(\mathbf{P})_1$, $Gnd(\mathbf{P})_2$, ..., $Gnd(\mathbf{P})_{t_{max}}$, or, in other words, grounded versions of the original time-dependent program for time intervals $\{0,1\}$,$\{0,1,2\}$,...,$\{0,\ldots,t_{max}\}$. Instead, we propose to consecutively solve smaller programs for intervals $\{0,1\}$, $\{1,2\}$,..., $\{t_{max}-1,t_{max}\}$. This approach is more efficient because we ground only for one time step at a time and solve smaller programs in every iteration. Further in this section we show that by doing so we obtain the same answer sets as by solving the initial program for interval $\{0,\ldots,t_{max}\}$.

Definition 3.13 (Markovian program)**.** A time-dependent program $\mathbf{P} = \langle P, \tau \rangle$ is called *Markovian* iff it satisfies the following conditions for every $r \in P$ with $head(r) \in Lit(\mathbf{P})^\tau$ and $t \in \mathbb{N}$:

(1) $t_{arg}(head(r)) \in \gamma \cup \upsilon$
(2) $t_{arg}(Gnd(head(r))_t) = t_{arg}(Gnd(l)_t)$ or $t_{arg}(Gnd(head(r))_t) = t_{arg}(Gnd(l)_t) + 1$ for all $l \in Lit(r) \cap Lit(\mathbf{P})^\tau$

Rules in a Markovian program $\mathbf{P}$ can be divided into two subsets: a program that describes temporal relationships $P^\tau = \{r | r \in P, (head(r) \cup Lit(r)) \cap Lit(\mathbf{P})^\tau \neq \emptyset\}$ and a program that describes the rest of the relationships $P^e = P \setminus P^\tau$. Program P^e can be interpreted as environmental conditions that are invariant over time. By definition, P^e is independent from the program's temporal part, thus it can be solved separately to obtain its answer sets that represent the values of these conditions. Note that if P^e does not have an answer set, then for any $t_{max} \in \mathbb{N}$, $Gnd(\mathbf{P})_{t_{max}}$ does not have an answer set either.

Example 3.14. Consider Markovian program $\mathbf{P}$ and its 2-grounding $Gnd(\mathbf{P})_2$ as defined in Example 3.6. Here the program P^τ contains rules 1-15, while the program P^e contains rule 16. The program $Gnd(\mathbf{P})_2$ has two answer sets, namely C and D as defined in Example 1.1. The unique answer set of P^e is $\{r(str)\}$.

Definition 3.15 (Partial temporal grounding)**.** Let $\mathbf{P} = \langle P, \tau \rangle$ be a Markovian program and $t \in \mathbb{N}$. The *partial temporal grounding* of $\mathbf{P}$ for time point t is defined as

$$P_t = \{Gnd(r)_t | r \in P, head(r) \in Lit(\mathbf{P})^\tau, t_{arg}(Gnd(head(r))_t) = t\}$$

In other words, a partial temporal grounding for a time point t is the set of t-grounded rules whose head depends on time point t.

Example 3.16. The partial temporal grounding of $\mathbf{P}$ built in Example 3.6 for time point 2 is the program P_2 that is defined as follows

$$\begin{array}{rlll} 3: & time(2). & & \\ 6: & q(2) & \leftarrow & p(1), time(2), time(1). \\ 9: & v(2) & \leftarrow & q(1), not\ w(2), time(2), time(1). \\ 12: & w(2) & \leftarrow & q(1), not\ v(2), r(X), time(2), time(1). \\ 15: & p(2) & \leftarrow & time(2). \end{array}$$

Assume that the t_{max}-grounding $Gnd(\mathbf{P})_{t_{max}}$ of a Markovian program $\mathbf{P}$ has an answer set A. Once A is known, using Definition 3.7, we can straightforwardly find its states at time points 0, 1, ..., t_{max}, i.e., A^0, A^1, ..., $A^{t_{max}}$. Below we show that it is also possible to find states of an answer set without prior knowledge of an answer set itself. In particular, the state A^t of an (unknown) answer set of $Gnd(\mathbf{P})_{t_{max}}$ at time point t can be computed based on knowledge of the state A^{t-1} at time point $t-1$, as well as knowledge of an answer set A^{-1} of P^e. This means that from the answer sets of P^e, the set of states at time point 0 can be found, and from this the set of states at time point 1, etc. This is done by building $P'_t = P_t \cup \{l \leftarrow . | l \in A^{t-1} \cup A^{-1}\}$ and then grounding P'_t and transforming it by replacing literals from $A^{t-1} \cup A^{-1}$ with true values, which is formally defined in Definition 3.17. Solving the resulting (small) program yields as answer sets the possible states at time point t given the state A^{t-1} and the environmental conditions A^{-1}.

Definition 3.17 (Partial reduct)**.** Let P be a ground program, I an interpretation of P and $P_I = \{l \leftarrow . | l \in I\}$, such that $P_I \subseteq P$ and $head(P \setminus P_I) \cap I = \emptyset$. The *partial reduct* of P w.r.t. I is the program $R^I(P)$ defined as

$$R^I(P) = \{head(r) \leftarrow (pos(r) \setminus I), \mathbf{not}\, neg(r). | r \in P \setminus P_I, neg(r) \cap I = \emptyset\}$$

Example 3.18. Assume we known that $\{time(1), p(1), q(1)\}$ is the state at time point 1 of a (possibly unknown) answer set of program $Gnd(\mathbf{P})_2$ from Example 3.6. We also know an answer set of P^e, namely $\{r(str)\}$. Let P_2 be the partial temporal grounding of $\mathbf{P}$ for time point 2 as described in Example 3.16. We construct the set $I = \{time(1), p(1), q(1)\} \cup \{r(str)\}$ and the program $P'_2 = Gnd(P_2 \cup \{l \leftarrow . | l \in I\})$ as follows:

$$\begin{array}{llll} time(2). & & & time(1). \\ q(2) & \leftarrow & p(1), time(2), time(1). & p(1). \\ v(2) & \leftarrow & q(1), not\ w(2), time(2), time(1). & q(1). \\ w(2) & \leftarrow & q(1), not\ v(2), r(str), time(2), time(1). & r(str). \\ p(2) & \leftarrow & time(2). & \end{array}$$

The partial reduct $R^I(P'_2)$ is then defined as

$$\begin{array}{llllll} time(2). & & & & & \\ q(2) & \leftarrow & time(2). & w(2) & \leftarrow & not\ v(2), time(2). \\ v(2) & \leftarrow & not\ w(2), time(2). & p(2) & \leftarrow & time(2). \end{array}$$

By applying the partial reduct we remove the literals from I that appear positively in rule bodies as well as the facts that appear as literals in I. The answer sets of the resulting program are $\{time(2), p(2), q(2), v(2)\}$ and $\{time(2), p(2), q(2), w(2)\}$ which correspond to C^2 and D^2 with C and D as in Example 1.1.

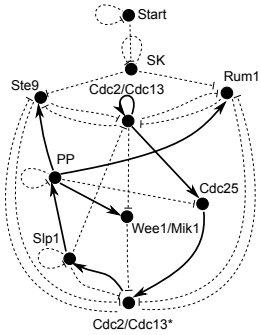


Figure 1: The gene regulatory network of Fission Yeast that can be modelled using Markovian programs (illustration from [Dav08]).

The theorem below states that instead of computing answer sets of a Markovian program $Gnd(\mathbf{P})_{t_{max}}$ directly, we can compute answer sets of smaller programs for every time step $0 \leq t' \leq t_{max}$ consecutively and obtain the same result. This fact has as an important implication that we can arrive at answer sets of a Markovian program without considering t_{max} at all, which allows to impose another stopping condition. This is the technique that is used to implement an algorithm for computing steady states explained in the following section.

Theorem 3.19. *Let* $\mathbf{P} = \langle P, \tau \rangle$ *be a Markovian program and* $Gnd(\mathbf{P})_{t_{max}}$ *be a* t_{max}*-grounding of* $\mathbf{P}$ *for* $t_{max} \in \mathbb{N}$*, then*

$$AS(Gnd(\mathbf{P})_{t_{max}}) = \left\{ B^{-1} \cup \ldots \cup B^{t_{max}} \;\middle|\; \begin{array}{l} B^{-1} \in AS(P^e), , B^t \in AS(R^{B^{t-1} \cup B^{-1}}(P'_t)), \\ t \in 0 \ldots t_{max} \end{array} \right\}$$

with $P'_t = Gnd(P_t \cup \{l \leftarrow . | l \in B^{t-1} \cup B^{-1}\})$.

4. Practical application

The results from the previous section give rise to an algorithm for finding all steady states and cycles of Markovian programs. It can be summarized as:

(1) Solve program P^e with the environmental conditions and initialize $t = 0$.
(2) Obtain the partial temporal grounding for t and find the system's states at time t.
(3) Update the list of trajectories with the new states found in step (2).
(4) Increment t.
(5) If any of the trajectories did not reach steady state or cycle, go to step (2).

This algorithm can be applied to model gene regulatory networks. An example regulatory network of Fission Yeast is presented in Figure 1, where nodes stand for genes and proteins, pointed edges define the activation of one node by another and blunt edges define the inhibition of one node by another. The semantics of the network can be expressed as a program P, while the actual network structure can be defined independently in a separate program P' as described in [Fay09]. The resulting program is a Markovian program $\mathbf{P} = \langle P \cup P', \tau \rangle$. A trajectory in the network is found w.r.t. an initial state of the network. The state of the network is defined as a combination of states of its nodes, where the state

of a node is defined as $active(a, T)$ or $inhibited(a, T)$ where a is a protein and T is a time variable, i.e. $active, inhibited \in \tau$. Looking at the network it is not possible to estimate how many time steps it would take to find all network steady states, and setting the time boundary too high would result in significant computation overheads, while the approach we propose does not involve an explicit time boundary and thus avoids these overheads. Solving the program with the algorithm outlined above allows to obtain trajectories and all steady states and cycles of the network that are reported in [Dav08].

5. Related work

In Section 4 we proposed a method to find all steady states of a Markovian program efficiently. However, our approach is not the only way to deal with the problem. Gebser et al. have recently proposed an incremental program solving approach and a specially constructed solver *iclingo* that allows for solving incremental programs [Geb08]. Even though this solver, when used for Markovian programs, terminates as soon as the first steady state is encountered, and hence unlike our approach does not find all steady states, Gebser et al.'s proposal is relevant to our work. An incremental program includes a special incremental parameter k and consists of three parts (base, cumulative and volatile) that allow to reduce the efforts required for solving this type of programs. Due to the space limitation we refer to [Geb08] for details. The advantage of this approach compared to the usual solving process is that it reduces the effort of computing the answer set for unknown k.

If we regard the incremental parameter k as time, we can simulate a Markovian program $\mathbf{P} = \langle P, \tau \rangle$ by putting P^e in the base part and P^τ in the cumulative part. However, implementing the volatile part is not straightforward. Given the set τ of time-dependent predicates we can write rules to capture steady states or cycles and define a constraint over the occurrence of such a state or cycle in the volatile part, as illustrated below.

Example 5.1. Let $\mathbf{P} = \langle P, \tau \rangle$ be a Markovian program over a signature $\sigma = \langle \gamma, \upsilon, \pi \rangle$ and $\tau = \{u, v\}$ where $u, v \in \pi$ are unary time-dependent predicates. We define an incremental program P' from $\mathbf{P}$ as explained above, i.e. by putting P^e in the base part of P' and P^τ in the cumulative part of P'. The exact contents of P^e and P^τ do not matter for the sake of this example. Next, we add the following set of rules to the cumulative part of P':

$$
\begin{array}{lcl}
int(0..k-1). & & \\
h(k) & \leftarrow & not\ u(k), not\ u(k-T_1), not\ v(k), not\ v(k-T_1), int(T_1). \\
h(k) & \leftarrow & u(k), u(k-T_1), not\ v(k), not\ v(k-T_1), int(T_1). \\
h(k) & \leftarrow & v(k), v(k-T_1), not\ u(k), not\ u(k-T_1), int(T_1). \\
h(k) & \leftarrow & v(k), v(k-T_1), u(k), u(k-T_1), int(T_1).
\end{array}
$$

Finally, we initialize the volatile part of P' with the rule $\leftarrow \mathbf{not}\, h(k)$. Intuitively, the appearance of $h(k)$ in an answer set of P' indicates that a steady state or cycle is found. The constraint in the volatile part only allows answer sets that contain $h(k)$.

However, there are two pitfalls associated with the above encoding. First, the number of rules that needs to be added to the cumulative part grows exponentially with the number and the arity of time-dependent predicates; recall that we do not only need all combinations of time-dependent predicates, but also all their possible groundings. Secondly, the solver terminates as soon as the first steady state is encountered, and hence does not generate all steady states of the program. It is not obvious how to encode the program in order to deal with these problems. For these reasons, the approach we propose in this paper is a

more suitable candidate to tackle the steady state search problem in Markovian programs. Applying a meta-procedure similar to the algorithm from Section 4 is not a solution as the incremental program still cannot find all steady states that stem from the same initial state without adjusting the termination condition $h(k)$.

Action languages [Gel92], another set of formalisms applicable to solve time-dependent programs, provide a high-level description language that can be adopted to model time-dependent systems. However, they suffer from the same drawback as incremental programs: it is not possible to define a set of constraints that allows to find all steady states and cycles.

6. Conclusions

In this paper, we introduced time-dependent answer set programs, which are useful to model systems like gene regulatory networks whose behaviour depends on time. An important task when modelling such systems is to find their steady states and cycles. Unfortunately, it is typically not known in advance at what time steps these steady states manifest themselves. A brute force approach of estimating a time upper bound and grounding and solving the program w.r.t. that upper bound may lead to a bad solving time: if the upper bound's estimate is too high, the grounded program is larger than necessary to find the steady states, hence requiring unnecessary work, and if it is too low, not all steady states (if any) are found and the process needs to be redone for a larger estimate.

We proposed an efficient algorithm for solving Markovian programs, i.e. time-dependent programs for which the next state of the program depends only on the previous state of the program. This is a reasonable assumption as real-world models are normally unaware of any future events and make their decisions based on the information directly available. Instead of solving Markovian programs for some long time interval $\{0, \ldots, t_{max}\}$ we consecutively solve smaller programs for intervals $\{0, 1\}$, $\{1, 2\}$,..., $\{t_{max} - 1, t_{max}\}$, which can be done more efficiently. We showed that by doing so we obtain the same answer sets as by solving the initial program for interval $\{0, \ldots, t_{max}\}$. We successfully applied our algorithm to find the steady states of a gene regulatory network for fission yeast.

References

[Dav08] Maria I. Davidich and Stefan Bornholdt. Boolean network model predicts cell cycle sequence of fission yeast. *PLoS ONE*, 3(2), 2008.

[Dwo08] Steve Dworschak, Susanne Grell, Victoria J. Nikiforova, Torsten Schaub, and Joachim Selbig. Modeling biological networks by action languages via answer set programming. *Constraints*, 13(1-2):21–65, 2008.

[Fay09] Timur Fayruzov, Martine De Cock, Chris Cornelis, and Dirk Vermeir. Modeling protein interaction networks with answer set programming. In *BIBM*, pp. 99–104. 2009.

[Geb08] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Max Ostrowski, Torsten Schaub, and Sven Thiele. Engineering an incremental asp solver. In *ICLP*, pp. 190–205. 2008.

[Gel88] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. pp. 1070–1080. MIT Press, 1988.

[Gel92] Michael Gelfond and Vladimir Lifschitz. Representing actions in extended logic programming. In *JICSLP*, pp. 559–573. 1992.

[Sch09] Torsten Schaub and Sven Thiele. Metabolic network expansion with answer set programming. In *ICLP*, pp. 312–326. 2009.

[Tra06] Nam Tran. *Reasoning and hypothesizing about signaling networks*. Ph.D. thesis, Arizona State University, 2006.

Technical Communications of the International Conference on Logic Programming, 2010 (Edinburgh), pp. 74–83
http://www.floc-conference.org/ICLP-home.html

IMPROVING THE EFFICIENCY OF GIBBS SAMPLING FOR PROBABILISTIC LOGICAL MODELS BY MEANS OF PROGRAM SPECIALIZATION

DAAN FIERENS

Katholieke Universiteit Leuven, Department of Computer Science, Celestijnenlaan 200A, 3001 Heverlee, Belgium
E-mail address: Daan.Fierens@cs.kuleuven.be

Abstract. There is currently a large interest in probabilistic logical models. A popular algorithm for approximate probabilistic inference with such models is Gibbs sampling. From a computational perspective, Gibbs sampling boils down to repeatedly executing certain queries on a knowledge base composed of a static part and a dynamic part. The larger the static part, the more redundancy there is in these repeated calls. This is problematic since inefficient Gibbs sampling yields poor approximations.
We show how to apply program specialization to make Gibbs sampling more efficient. Concretely, we develop an algorithm that specializes the definitions of the query-predicates with respect to the static part of the knowledge base. In experiments on real-world benchmarks we obtain speedups of up to an order of magnitude.

1. Introduction

In the field of artificial intelligence there is a large interest in *probabilistic logical models* (probabilistic extensions of logic programs and first-order logical extensions of probabilistic models such as Bayesian networks) [3, 10, 5]. *Probabilistic inference* with such a model is the task of answering various questions about the probability distribution specified by the model, usually conditioned on certain observations (the *evidence*). A variety of inference algorithms is being used. A popular algorithm for approximate probabilistic inference is *Gibbs sampling* [2, 11]. Gibbs sampling works by drawing samples from the considered probability distribution conditioned on the evidence. These samples can be used to compute an approximate answer to the probabilistic questions of interest. It is important that the process of drawing samples is efficient because the more samples can be drawn per time-unit, the more accurate the answers will be (i.e., the closer to the correct answer).

Computationally, Gibbs sampling boils down to repeatedly executing the same queries on a knowledge base composed of a static part (the evidence and background knowledge) and a highly dynamic part that changes at runtime because of the sampling. The more evidence, the larger the static part of the knowledge base, so the more redundancy there

1998 ACM Subject Classification: I.2.2, G.3, D.1.6.

Key words and phrases: probabilistic logical models, probabilistic logic programming, program specialization, Gibbs sampling.

This research is supported by Research Foundation-Flanders (FWO Vlaanderen), GOA/08/008 'Probabilistic Logic Learning' and Research Fund K.U.Leuven.

Technical Communications of the 26th International Conference on Logic Programming, Edinburgh, July, 2010
Editors: Manuel Hermenegildo, Torsten Schaub
LIPIcs - Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/LIPIcs.ICLP.2010.74

is in these repeated calls. Since it is important that the sampling process is efficient, this redundancy needs to be reduced as much as possible. In this paper we show how to do this by applying *program specialization* to the definitions of the query-predicates: we specialize these definitions with respect to the static part of the knowledge base. While a lot of work about logic program specialization is about exploiting static information about the input arguments of queries (partial deduction [6]), we instead exploit static information about the knowledge base on which the queries are executed.

While the above applies to all kinds of probabilistic logical models and programs, we will focus on models that are first-order logical or "relational" extensions of Bayesian networks [3, 5]. Concretely, we use the general framework of *parameterized Bayesian networks* [10].

The *contributions of this paper* are the following. First, we show how to represent parameterized Bayesian networks in Prolog (Section 3). Second, we show how to implement Gibbs sampling in Prolog and show that doing this efficiently poses challenges from the logic programming point of view (Section 4). Third, we develop an algorithm for specializing the considered logic programs with respect to the evidence (Section 5). Fourth, we perform experiments on real-world benchmarks to investigate the influence of specialization on the efficiency of Gibbs sampling. Our results show that specialization yields speedups of up to an order of magnitude and that these speedups grow with the data-size (Section 6). The latter two are the main contributions of this paper, the first two are minor contributions.

We first give some background on probability theory and Bayesian networks.

2. Preliminaries: Probability Theory and Bayesian Networks

In probability theory [8] one models the world in terms of *random variables (RVs)*. Each state of the world corresponds to a joint state of all considered RVs. We use upper case letters to denote single RVs and boldface upper case letters to denote sets of RVs. We refer to the set of possible states of an RV X (i.e. the set of values that X can take) as the *range* of X, denoted $range(X)$. We consider only *discrete RVs*, i.e. RVs with a finite range.

A *probability distribution* on a finite set S is a function that maps each $x \in S$ to a number $P(x) \in [0, 1]$ such that $\sum_{x \in S} P(x) = 1$. A probability distribution for an RV X is a probability distribution on the set $range(X)$. A *conditional probability distribution (CPD)* for an RV X conditioned on a set of other RVs $\mathbf{Y}$ is a function that maps each possible joint state of $\mathbf{Y}$ to a probability distribution for X.

Syntactically, a *Bayesian network* [8] for a set of RVs $\mathbf{X}$ is a set of CPDs: for each $X \in \mathbf{X}$ there is one CPD for X conditioned on a (possibly empty) set of RVs called the *parents* of X. Intuitively, the CPD for X specifies the direct probabilistic influence of X's parents on X. The probability distribution for X conditioned on its parents $\mathbf{pa}(X)$, as determined by the CPD for X, is denoted $P(X \mid \mathbf{pa}(X))$.

Semantically, a Bayesian network represents a probability distribution $P(\mathbf{X})$ on the set of all possible joint states of $\mathbf{X}$. Concretely, $P(\mathbf{X})$ is the product of all the CPDs in the Bayesian network: $P(\mathbf{X}) = \prod_{X \in \mathbf{X}} P(X \mid \mathbf{pa}(X))$. It can be shown that $P(\mathbf{X})$ is a proper probability distribution provided that the parent relation is acyclic (the parent relation is often visualized as a directed acyclic graph but given the CPDs this graph is redundant).

3. Parameterized Bayesian Networks

Bayesian networks essentially use a propositional representation. Several ways of lifting them to a first-order representation have been proposed [3, Ch.6,7,13] [5]. There also exist several probabilistic extensions of logic programming, such as PRISM, Independent Choice Logic and ProbLog [3, Ch.5,8]. Both kinds of probabilistic logical models (probabilistic logic programs and the extensions of Bayesian networks) essentially serve the same purpose. In this paper we focus on the Bayesian network approach. Our main motivation for this choice is that this paper is about Gibbs sampling and this has been well-studied in the context of Bayesian networks. There are many different representation languages for first-order logical or "relational" extensions of Bayesian networks. We use the general framework of *parameterized Bayesian networks* [10]. While this framework is perhaps not a full-fledged knowledge representation language, it does offer a representation that is suited to implement probabilistic inference algorithms on.

We now briefly introduce parameterized Bayesian networks. Like Bayesian networks use RVs, parameterized Bayesian networks use so-called *parameterized RVs* [10]. Parameterized RVs have a number of typed parameters ranging over certain populations. When each parameter in a parameterized RV is instantiated or "grounded" to a particular element of its population we obtain a regular or *"concrete"* RV. To each parameterized RV we associate a *parameterized CPD* (see below) with the same parameters as the parameterized RV.

Syntactically, a parameterized Bayesian network is a set of parameterized CPDs, one for each parameterized RV. Semantically, a parameterized Bayesian network $\mathcal{B}$, in combination with a given population for each type, specifies a probability distribution. Let $\mathbf{X}$ denote the set of all concrete RVs obtained by grounding all parameterized RVs in $\mathcal{B}$ with respect to their populations. The probability distribution specified by $\mathcal{B}$ is the following distribution on the set of all possible joint states of $\mathbf{X}$: $P(\mathbf{X}) = \prod_{X \in \mathbf{X}} P(X \mid \mathbf{pa}(X))$, where $P(X \mid \mathbf{pa}(X))$ denotes the probability distribution for X as determined by its parameterized CPD.

Rather than providing a formal discussion of parameterized Bayesian networks we show how they can be represented in Prolog (as far as we know this has not been done before).

To deal with parameterized RVs in Prolog we associate to each of them a unique predicate: for a parameterized RV with n parameters we use a $(n+1)$-ary predicate, the first n arguments correspond to the parameters, the last argument represents the state of the RV. We refer to these predicates as *state predicates*.

Syntactically a parameterized Bayesian network is a set of parameterized CPDs. To deal with parameterized CPDs we also associate to each of them a unique predicate, the last argument now represents a probability distribution on the range of the associated RV. We refer to these predicates as *CPD-predicates*. In this paper we assume that each CPD-predicate is defined by a decision list. A *decision list* is an ordered set of rules such that there is always at least one rule that applies, and of all rules that apply only the first one fires (in Prolog this is achieved by putting a cut at the end of each body and having a last clause with *true* as the body).

Example 3.1. Consider a university domain. Suppose that we use the following parameterized RVs: level (with a parameter from the population of courses), iq and graduates (each with a student parameter) and grade (with a student parameter and a course parameter). To represent the state of the RVs we use the state predicates $level/2$, $iq/2$, $graduates/2$ and $grade/3$. The meaning of for instance $level/2$ is that the atom $level(C, L)$ is true if the parameterized RV *level* for the course C is in state L .

To represent parameterized CPDs we use CPD-predicates *cpd_level*/2, *cpd_iq*/2, *cpd_grade*/3 and *cpd_graduates*/2. If the *level* RVs for instance do not have parents, their parameterized CPD could be defined as follows.

```
cpd_level(_C,[intro:0.4,advanced:0.6]).
```

Note that we use lists like `[intro:0.4,advanced:0.6]` to represent probability distributions. The other parameterized CPDs could for instance be defined as follows.

```
cpd_iq(_S,[high:0.5,low:0.5]).

cpd_grade(S,C,[a:0.7,b:0.2,c:0.1]) :- iq(S,high), level(C,intro), !.
cpd_grade(S,C,[a:0.2,b:0.2,c:0.6]) :- iq(S,low), level(C,advanced), !.
cpd_grade(S,C,[a:0.3,b:0.4,c:0.3]).

cpd_graduates(S,[yes:0.2,no:0.8]) :- grade(S,_C,c), !.
cpd_graduates(S,[yes:0.5,no:0.5]) :- findall(C,grade(S,C,a),L),
                                                    length(L,N), N<2, !.
cpd_graduates(S,[yes:0.9,no:0.1]).
```

In the bodies of the clauses defining the CPD-predicates we allow the use of state predicates (e.g. *iq*/2 and *level*/2 in the clauses for *cpd_grade*/3) and of background predicates, but not of CPD-predicates. With *background predicates* we mean auxiliary predicates that do not depend on the state of RVs (this includes built-ins such as *length*/2). We assume that the definitions of the background predicates are available in a *background knowledge base*. We also allow the use of meta-predicates (such as *findall*/3) but not of predicates with side-effects (such as *assert/1*).

When we know the population for each type (e.g. we know the set of students and the set of courses) we also know the set of concrete RVs $\mathbf{X}$. Suppose that in addition we also know the state of these concrete RVs because we are given a knowledge base with facts defining the state predicates (e.g. a fact $grade(s1, c1, a)$ indicates that student $s1$ has grade 'a' for course $c1$). We can then obtain the probability distribution for a concrete RV conditioned on its parents by simply calling the associated CPD-predicate on this knowledge base. For instance, we obtain the probability that the student $s1$ will graduate conditioned on her grades by calling $cpd\_graduates(s1, Distribution)$. We refer to this as *calling the CPD for that concrete RV*. Since we represent each parameterized CPD as a decision list it is guaranteed that this always returns exactly one probability distribution.[1]

As we explain in the next section, calling a CPD is an operation that needs to be performed frequently during probabilistic inference. Another such operation is *setting a concrete RV to a given state*. This is done by modifying the corresponding fact in the knowledge base (e.g. the fact $grade(s1, c1, a)$ is turned into $grade(s1, c1, b)$ [4]).

4. Probabilistic Inference with Parameterized Bayesian Networks

Given the population for each type, a parameterized Bayesian network defines a probability distribution $P(\mathbf{X})$ on the set of all possible joint states of the concrete RVs $\mathbf{X}$. In a typical inference scenario, the state of a subset of all these RVs is observed. This information is called the *evidence*. *Probabilistic inference* is the task of answering certain questions

[1] Some CPD-predicates are defined by non-ground facts (e.g. *cpd_level*/2). This does not cause problems because we always call CPD-predicates with all arguments except the last instantiated.

about the probability distribution $P(\mathbf{X})$ conditioned on the evidence. The most common inference task is to compute marginal probabilities. A *marginal probability* is the probability that a particular RV is in a particular state. For instance, given the level of all courses and the grades of all students for all courses (the evidence), we might want to compute for each student the probability that she has a high IQ. In theory such probabilities can be computed by performing a series of sum and product operations on the probability distributions specified by the parameterized CPDs. Unfortunately, for real-world population sizes this is computationally intractable (inference with Bayesian networks is NP-hard [8]). Hence, one often uses *approximate probabilistic inference* instead. An important class of approximate inference algorithms are *Monte Carlo algorithms* that draw samples from the given distribution conditioned on the evidence. Various algorithms are being used, a very popular one is *Gibbs sampling* [2, 11].

Let $\mathbf{O}$ denote the set of all observed concrete RVs (i.e. the RVs for which we have evidence), and $\mathbf{U}$ the set of all unobserved ones ($\mathbf{U} = \mathbf{X} \setminus \mathbf{O}$). Below we assume that we need to compute marginal probabilities for all unobserved RVs. Pseudocode for the Gibbs sampling algorithm is shown in Figure 1. We now explain this further.

```
procedure GIBBS_SAMPLING(O, U)
1 for each O ∈ O
2     set O to its known state
3 for each U ∈ U
4     set U to random state ∈ range(U)
5     initialize all counters for U
6 repeat until enough samples
7     for each U ∈ U
8         RESAMPLE(U)
9 compute estimates from counters
```

```
procedure RESAMPLE(U)
1 call the CPD for U
2 for each u ∈ range(U)
3     set U to state u
4     for each child X of U
5         call the CPD for X
6 calculate P_resample(U)
7 sample u_new from P_resample(U)
8 set U to u_new
9 increment counter for (U, u_new)
```

Figure 1: The Gibbs sampling algorithm (left) and its RESAMPLE procedure (right).

Before the start of the sampling process all observed RVs are instantiated to their known state and all unobserved RVs are instantiated to a random state. In terms of our implementation in Prolog, this is done by creating a knowledge base defining all the state predicates: for each RV $\in \mathbf{O} \cup \mathbf{U}$ there is one fact for the corresponding state predicate. Before we start sampling, we also create a number of counters: for each $U \in \mathbf{U}$ and each $u \in range(U)$ we create a counter to store the number of samples in which U is in state u. All counters are initialized to zero.

Let us now consider the sampling process itself. To create one sample, we visit (in an arbitrary but fixed order) all unobserved RVs. When we visit an RV U, we "resample" it. The idea is to sample the new state from the probability distribution for U conditioned on the current state of *all* other RVs. For details on how to construct this distribution $P_{resample}(U)$ we refer to Bidyuk and Dechter [1], here we focus on the main computations that this requires (see the RESAMPLE procedure in Figure 1): first we need to call the CPD for U, then we loop over all possible states of U and for each state u we set U to u and call the CPDs of each of the children of U.Based on the information returned by all these CPD-calls it is straightforward to construct the distribution $P_{resample}(U)$. We then randomly sample a state from this distribution, set U to this new state and increment the appropriate counter for U.

The above is done for all unobserved RVs, yielding one sample.[2] Note that observed RVs are clamped to their known state, hence the generated sample is guaranteed to be consistent with the evidence. This entire procedure is repeated N times, yielding N samples. It is then straightforward to construct an estimate of all required marginal probabilities based on the computed counts. For instance, the estimated probability that student $s1$ has a high IQ conditioned on the evidence is the number of samples in which the RV iq for $s1$ was in the state 'high', divided by N.

The higher the number of samples N, the closer the estimated marginal probabilities will be to their correct values [1, 4]. Gibbs sampling is often used by giving the sampling process a fixed time to run before computing the estimates. In this case, the less time it takes to draw a single sample, the more samples can be drawn in the given time, so the higher the accuracy of the estimates. In other words: any gain in efficiency of the sampling process might lead to a gain in accuracy of the estimates. Hence it is crucial to implement the sampling process as efficiently as possible.

The Gibbs sampling algorithm uses several operations, but there is one operation that we clearly found to be the computational bottleneck, namely *calling the CPDs*. This operation occurs inside several nested loops (see line 5 of the RESAMPLE procedure in Figure 1) and is hence performed many times. The knowledge base on which these CPD-queries are called is highly dynamic: the state of the unobserved RVs changes continuously because they are being resampled. This is only one part of the knowledge base, however. The part that is about the observed RVs (the evidence) stays constant during the entire sampling process. This static part of the knowledge base causes redundancy in the repeated calls of the CPD-queries since part of the computations are performed over and over again. The more evidence we have, the larger the redundancy. In many practical cases, the amount of evidence is considerable and hence the redundancy can be large. Since we want the sampling process to be as efficient as possible, this redundancy needs to be removed. In the next section we show how this can be done by means of program specialization.

5. Applying Logic Program Specialization to Parameterized CPDs

The idea is to *specialize the definitions of the CPD-predicates with respect to the static part of the knowledge base*. Recall that we define each CPD-predicate in Prolog by means of a decision list (Example 3.1). Our specialization approach is a source-to-source transformation that takes three inputs: 1) the decision lists for all the CPD-predicates, 2) the evidence (i.e. the observed RVs with their observed states), and 3) the background knowledge base. The output of the transformation is a specialized version of the decision lists. The transformation is such that Gibbs sampling produces exactly the same sequence of samples with the specialized decision lists as with the original ones (but in a more efficient way).

We use the term *CPD-query* to refer to any atom for a CPD-predicate with the last argument uninstantiated and all other arguments instantiated to elements of the proper populations. For instance, $cpd\_grade(s, c, Distribution)$ is a CPD-query if s is in the considered population of students and c in the population of courses. All calls to CPD-predicates that occur during Gibbs sampling are calls of CPD-queries. Moreover, there is only a fixed set of CPD-queries that are ever called during Gibbs sampling: by examining the RESAMPLE procedure (Figure 1) one can see that the only CPD-queries that are ever called are those

[2] In practice we use a slight variation of this procedure which includes a number of common optimizations (such as making use of the 'support network' [3, Ch.7]).

associated to an unobserved RV (line 1 of RESAMPLE) or to an RV with an unobserved parent (line 5). As long as the specialized decision lists that we construct behave exactly the same with respect to this fixed set of CPD-queries as the original decision lists do, Gibbs sampling will indeed produce exactly the same samples with specialization as without.

There is a lot of existing work on transformation or specialization of logic programs that has the same end-goal as our work, namely transforming a given program to an "equivalent" but more efficient program [9]. However, we are not aware of any work that considers the same setting as we do, namely that of executing a fixed set of queries on a knowledge base with a static and a dynamic part, and specializing with respect to the static part. In particular, this setting makes our work different from the work on *partial deduction* for logic programs [6, 7]. In our setting, we know all input arguments of the queries but we know only part of the knowledge base on which they will be executed. In contrast, in the partial deduction setting, one knows only some of the input arguments of the queries but one knows the entire knowledge base. Hence, existing off-the-shelve systems for partial deduction (see e.g. Leuschel et al. [7]) are, as far as we see, not optimal for our setting.

Our specialization algorithm is shown in Figure 2. The main idea is the following. The CPD-predicates are defined in terms of the state predicates. The evidence is a partial interpretation of these state predicates (specifying the known state for a subset $\mathbf{O}$ of all concrete RVs). We want to specialize the definitions of the CPD-predicates with respect to this evidence. Since the evidence is defined at the ground level but the definitions of the CPD-predicates are at the non-ground level, we first (partially) ground these definitions before we specialize them. We now explain this further.

```
procedure SPECIALIZE(U, O, o)
1  for each CPD-predicate p
2      let D be the decision list for p
3      for each q ∈ AllQueries(p, U, O)
4          SPEC_DECISION_LIST(D, q, U, O, o)

procedure SPEC_DECISION_LIST(D, q, U, O, o)
1  if D is non-empty
2      let C be the first clause in D
       and D_rest be the other clauses in D
3      C_q = GROUND_HEAD(C, q)
4      let Head be the head and B_q the body of C_q
5      Body=SPECIALIZE_BODY(B_q, U, O, o)
6      if Body = true
7          ASSERT_FACT(Head)
8      else
9          if Body ≠ false
10             ASSERT_CLAUSE(Head, Body)
11         SPEC_DECISION_LIST(D_rest, q, U, O, o)
```

Figure 2: The specialization algorithm for the decision lists that define the CPD-predicates ($\mathbf{U}$ are the unobserved RVs, $\mathbf{O}$ the observed RVs and $\mathbf{o}$ their observed values).

The outer-loop of our algorithm (line 1 of the SPECIALIZE procedure in Figure 2) is over all the CPD-predicates: we specialize each CPD-predicate p in turn. To do so, we first collect all CPD-queries for p. As explained before, the only CPD-queries that we need are the ones associated to an RV that is unobserved or has an unobserved parent. The set of all such CPD-queries is denoted $AllQueries(p, \mathbf{U}, \mathbf{O})$ (line 3 of the SPECIALIZE procedure). We then loop over this set: for each CPD-query q we apply the SPEC_DECISION_LIST procedure. We explain this procedure by means of an example.

Example 5.1. Let p be *cpd_graduates*/2, let the decision list D that defines p be the same as given earlier in Example 3.1, and let the CPD-query q be $cpd\_graduates(s1, Distr)$. The SPEC_DECISION_LIST procedure starts by processing the first clause C in D:

```
cpd_graduates(S,[yes:0.2,no:0.8]) :- grade(S,_C,c), !.
```

First we ground the head variables of C with respect to q (line 3 of SPEC_DECISION_LIST) yielding the clause C_q:

```
cpd_graduates(s1,[yes:0.2,no:0.8]) :- grade(s1,_C,c), !.
```

Next, we apply the function SPECIALIZE_BODY to the body of C_q (line 5), yielding *Body* (see Example 5.2). There are three possible cases.

- If *Body* equals *true*, we assert a fact `cpd_graduates(s1,[yes:0.2,no:0.8])` (line 7). We can discard the remaining clauses in D with respect to q (these clauses will never be reached for q since only the first applicable clause in a decision list fires).
- If *Body* equals $false$, we discard C_q and continue with the next clause in D (line 11).
- Otherwise, we assert a clause of the form
  `cpd_graduates(s1,[yes:0.2,no:0.8]) :- Body, !.`
  (line 10) and we again continue with the next clause in D (line 11).

The function SPECIALIZE_BODY (Figure 2) is rather involved. For details we refer to the full paper [4]. We now give a very simple example.

Example 5.2. Let B_q, the body to be specialized, be `grade(s1,C,c)` (this is the situation of our previous example). First we ground the free variable C, yielding a disjunction B_1, namely `grade(s1,c1,c) ; ... ; grade(s1,cn,c)`. Then we specialize each of the literals in B_1 with respect to the evidence. Consider the first literal, `grade(s1,c1,c)`. If we have evidence that $s1$ obtained grade 'c' for course $c1$ then we replace the literal by *true*, if we have different evidence we replace it by $false$, if we have no evidence we leave it unchanged. Doing this for each literal yields a specialized disjunction B_2. Finally, we simplify B_2 using logical propagation rules (e.g. a disjunction is true if one if its disjuncts is true).

From the perspective of efficiency of the specialization process (time needed for specializing) our algorithm is not optimal: the specialization time can easily be reduced, for instance by more closely integrating the different steps of SPECIALIZE_BODY. However, in our experiments we observed that the specialization time is negligible as compared to the runtime of Gibbs sampling with the specialized decision lists (see the full paper [4]). Hence, we keep our specialization algorithm as simple as possible, rather than complicating it in order to reduce specialization time. This also makes it easier to see that specialization indeed preserves the semantics of the CPD-predicates (and hence that Gibbs sampling produces the same sequence of samples as without specialization).

6. Experiments

We now experimentally analyze the influence of specializing the definitions of the CPD-predicates on the efficiency of the Gibbs sampling algorithm.

We test our algorithms on three common real-world datasets: IMDB, UWCSE and WebKB. We obtained a parameterized Bayesian network for each dataset by means of machine learning. We use two inference scenarios. The first is *'prediction'*: there is one parameterized RV that we want to predict, all concrete RVs associated to that parameterized RV are unobserved, all others are observed. For each dataset we do multiple experiments,

each time with a different parameterized RV as the prediction target. The second scenario is *'missing data'*: a random fraction f of all concrete RVs is unobserved ('missing'), the others are observed. We use several values of f, ranging from 5% to 50%. For each value we repeat each experiment 5 times, each time with different unobserved RVs. We report the mean and standard deviation of the runtime across these 5 repetitions. More details about our experimental setup are given in the full paper [4].

We report the runtime of our Gibbs sampling algorithm in minutes. The *runtime without specialization* is the runtime of Gibbs sampling with parameterized CPDs that have not been grounded or specialized. The *runtime with specialization* is the sum of the specialization time and the runtime of Gibbs sampling with the specialized CPDs. Recall that both settings produce exactly the same sequence of samples.

The results for the 'missing data' scenario are shown in Figure 3. Using specialization always yields a speedup. The magnitude of the speedup of course greatly depends on the amount of evidence. On WebKB, the dataset that is by far the most computationally demanding, we get a speedup of an order of magnitude when there are 5% unobserved RVs. On the smaller datasets (IMDB and UWCSE), the speedups are more modest.

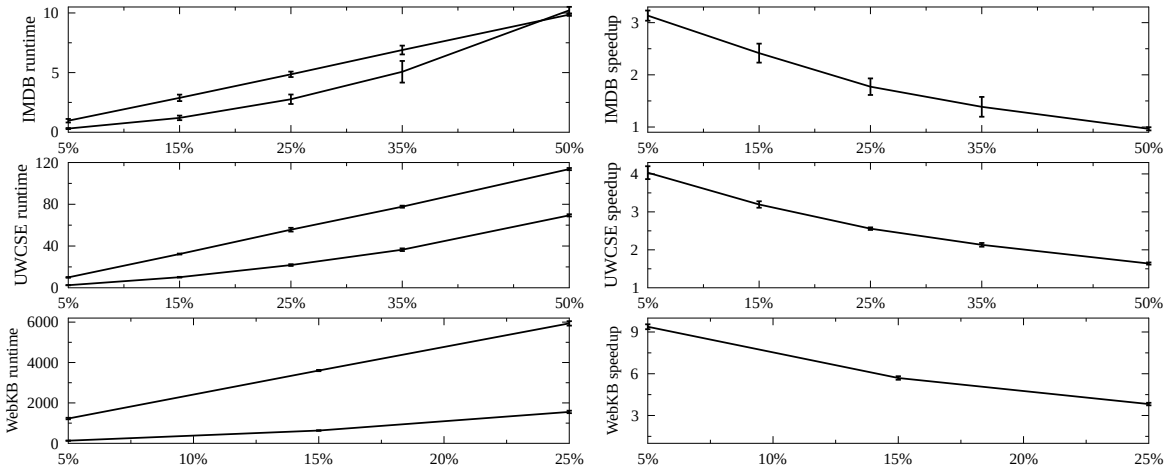


Figure 3: Results for the 'missing data' scenario. Left subgraphs show the runtime without (upper line) and with specialization (lower line); right subgraphs show the corresponding speedup-factor achieved due to specialization. Error bars indicate the standard deviation.

The results for the 'prediction' scenario are shown in Table 1. For half of the prediction targets, specialization yields significant speedups of a factor 4 to 7. For the other targets, the speedup is small to negligible (≤ 1.5). These are mostly cases where the state predicate that forms the computational bottleneck (e.g. because it is involved in a *findall*) is unobserved and hence cannot be specialized on.

In the above results (especially for the 'missing data' scenario), the speedups are the lowest on the smallest dataset (IMDB) and the highest on the largest one (WebKB). This suggest a correlation between the speedup due to specialization and the data-size. To investigate this, we performed additional experiments in which we varied the size of the datasets (see the full paper [4]). We found a clear trend: the larger the dataset, the higher

Table 1: Results for the 'prediction' scenario: runtime without specialization, runtime with specialization and speedup-factor achieved due to specialization.

Data/Target	No spec.	Spec.	Speedup	Data/Target	No spec.	Spec.	Speedup
IMDB/1	16.1	14.9	1.08	UWCSE/3	12.2	2.1	5.87
IMDB/2	2.6	1.7	1.51	UWCSE/4	71.8	15.8	4.55
UWCSE/1	75.1	17.4	4.31	WebKB/1	2628	406	6.48
UWCSE/2	10.9	10.4	1.05				

the speedup. This is a positive result: speedups are more necessary on large datasets than on small ones.

7. Conclusions

We considered the task of performing approximate probabilistic inference with probabilistic logical models by means of Gibbs sampling. We used the general framework of parameterized Bayesian networks. We showed how to represent the considered models and how to implement a Gibbs sampling algorithm for such models in Prolog. We argued that program specialization is suited to make this algorithm more efficient (which can in turn make the obtained inference answers more accurate) and introduced a concrete specialization algorithm. We experimentally investigated the influence of specialization on the efficiency of Gibbs sampling. Our results show that specialization yields speedups of up to an order of magnitude and that these speedups grow with the data-size.

References

[1] B. Bidyuk and R. Dechter. Cutset sampling for Bayesian networks. *Journal of Artificial Intelligence Research*, 28:1–48, 2007.

[2] C. M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[3] L. De Raedt, P. Frasconi, K. Kersting, and S. Muggleton. *Probabilistic Inductive Logic Programming*. Springer, 2008.

[4] D. Fierens. Improving the efficiency of Gibbs sampling for probabilistic logical models by means of program specialization. Technical Report CW-581, Department of Computer Science, Katholieke Universiteit Leuven, 2010. http://www.cs.kuleuven.be/publicaties/rapporten/cw/CW581.abs.html.

[5] D. Fierens, J. Ramon, M. Bruynooghe, and H. Blockeel. Learning directed probabilistic logical models: Ordering-search versus structure-search. *Annals of Mathematics and Artificial Intelligence*, 54(1):99–133, 2008.

[6] M. Leuschel and M. Bruynooghe. Logic program specialisation through partial deduction: Control issues. *Theory and Practice of Logic Programming*, 2(4-5):461–515, 2002.

[7] M. Leuschel, S.J. Craig, M. Bruynooghe, and W. Vanhoof. Specialising interpreters using offline partial deduction. In *Program Development in Computational Logic*, volume 3094 of *Lecture Notes in Computer Science*, pages 340–375. Springer, 2004.

[8] R.E. Neapolitan. *Learning Bayesian Networks*. Prentice Hall, New Jersey, 2003.

[9] A. Pettorossi and M. Proietti. Transformation of logic programs: Foundations and techniques. *Journal of Logic Programming*, 19-20:261–320, 1994.

[10] D. Poole. First-order probabilistic inference. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI 1997)*, pages 985–991. Morgan Kaufmann, 2003.

[11] V. Santos Costa. On the implementation of the CLP(BN) language. In *Proceedings of the 12th International Symposium on Practical Aspects of Declarative Languages (PADL 2010)*, volume 5937 of *Lecture Notes in Artificial Intelligence*, pages 234–248. Springer, 2010.

Technical Communications of the International Conference on Logic Programming, 2010 (Edinburgh), pp. 84–93
http://www.floc-conference.org/ICLP-home.html

FOCUSED PROOF SEARCH FOR LINEAR LOGIC IN THE CALCULUS OF STRUCTURES

NICOLAS GUENOT [1]

[1] Laboratoire d'Informatique (LIX), École Polytechnique
rue de Saclay, 91128 Palaiseau cedex, France
E-mail address: nguenot@lix.polytechnique.fr

Abstract. The proof-theoretic approach to logic programming has benefited from the introduction of *focused* proof systems, through the non-determinism reduction and control they provide when searching for proofs in the sequent calculus. However, this technique was not available in the *calculus of structures*, known for inducing even more non-determinism than other logical formalisms. This work in progress aims at translating the notion of *focusing* into the presentation of *linear logic* in this setting, and use some of its specific features, such as deep application of rules and fine granularity, in order to improve proof search procedures. The starting point for this research line is the multiplicative fragment of linear logic, for which a simple focused proof system can be built.

1. Introduction

The foundational principle of logic programming is the description of computation as a proof search process in some logical deductive system. It is therefore natural to implement logic programming languages in the framework of the sequent calculus, where the theory of proof objects is well-developped and analytic systems are available. This approach has been successful in extending logic programming to richer fragments of logic than Horn clauses, such as hereditary Harrop formulas [Mil91]. However, this required the definition of normal forms for proofs, reducing the search space and providing more structure, first with uniform proofs and then with focused proof systems [And92], initially designed for linear logic and later extended to both intuitionistic and classical logics [Lia09]. The focusing result is now considered an important part of proof theory, and is very much related to the broader and very active study of polarities in linear logic as well as other logics.

Proof Search in the Calculus of Structures. The deep inference methodology has been introduced to overcome the intrisic limitations of the sequent calculus, and design logical formalisms with nice symmetry properties. The most important by-product of this research line is the calculus of structures [Gug07], which generalises the sequent calculus by allowing inference rules to be applied deep inside formulas, as illustrated below:

$$\frac{\dfrac{(A \otimes (1 \otimes C)) \parr D}{(A \otimes ((B \parr B^{\perp}) \otimes C)) \parr D}}{(A \otimes (B \parr (B^{\perp} \otimes C))) \parr D}$$

Key words and phrases: proof theory, focusing, proof search, deep inference, linear logic.

Technical Communications of the 26th International Conference on Logic Programming, Edinburgh, July, 2010
Editors: Manuel Hermenegildo, Torsten Schaub
LIPIcs - Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/LIPIcs.ICLP.2010.84

There are several benefits to use this formalism, including the ability to produce proofs exponentially shorter than in the sequent calculus [Bru09], and also a large variety of design choices which allow the definition of several systems for the same logic easily, emphasizing different viewpoints.

However, the freedom given by the calculus of structures in the way of building proofs and using inference rules has an important drawback: the non-determinism involved in the process of proof construction is even greater than it is in the sequent calculus, mostly because of the possible choices regarding the depth of the next inference rule application. Thus, we cannot use directly such a system, and proof search requires the use of specific techniques [Kah06]. The goal of our project is then to get control over non-determinism using the focusing technique, which has to be adapted to this new setting.

Translating the Notion of Focusing. There is no such thing as a formal definition for the notion of focusing. The idea of this technique is mostly induced by the definition of focused sequent calculi, and their proof of completeness, although there is an intuition behind the methodology: invertible inference rules should be applied first, and a sequence of inference rules of the same polarity should always be applied as a group. In order to devise a focused proof system in the calculus of structures, we have to conform to this intuition, so that the result offers *proportionately* the same features as focusing does in the sequent calculus.

Outline of the Paper. We start with a quick survey of the usual focused sequent calculus $\mathsf{MLL}^{\mathsf{F}}$ for multiplicative linear logic, and then present the main contribution of this paper, the focused $\mathsf{MLS}^{\mathsf{F}}$ system in the calculus of structures, that we prove sound and complete. Finally, we discuss the design of this system, other possible choices as well as extensions to larger fragments of linear logic.

2. The $\mathsf{MLL}^{\mathsf{F}}$ Sequent Calculus

We work in the setting of linear logic [Gir87], and more precisely in the multiplicative fragment of this logic. This is a very small and simple logic, where no contraction nor weakening can happen, with two connectives : a conjunction $\otimes$ and a disjunction $\parr$, dual to each other. Negation can be pushed to the atoms using De Morgan's laws, and we use two sets to represent atoms and their negation, with a bijection $a \mapsto a^{\perp}$ between them.

The focused sequent calculus $\mathsf{MLL}^{\mathsf{F}}$ presented in Figure 1 is a simple restriction of the usual system [And92], where no particular mechanism is required to handle non-linear parts of the context. In this system, simple sequents are replaced with two kinds of sequents, distinguished by the different arrows used to annotate them. Then, inference rules require the definition of three classes of formulas:

$$A_+ ::= A \otimes A \mid 1 \mid a \qquad A_- ::= A \parr A \mid \perp \mid a^{\perp} \qquad A_* ::= A_+ \mid a^{\perp}$$

Remark 2.1. We consider atoms to be positive and their negations to be negative. This is arbitrary, since any *atomic bias* respecting the duality of negation can be used [Mil07].

Annotations are used to enforce that all available $\parr$ formulas are decomposed before any $\otimes$ formula, and that nested $\otimes$ are treated hereditarily. Another consequence of the syntactic restrictions is that whenever a positive atom is encoutered during a synchronous phase, the identity rule must close the branch. Finally, the management rules of *decision*, *exclusion* and *release* are meant to control annotations, and require to assume that the conclusive sequent of the proof is of the shape $\vdash \Uparrow \Gamma$.

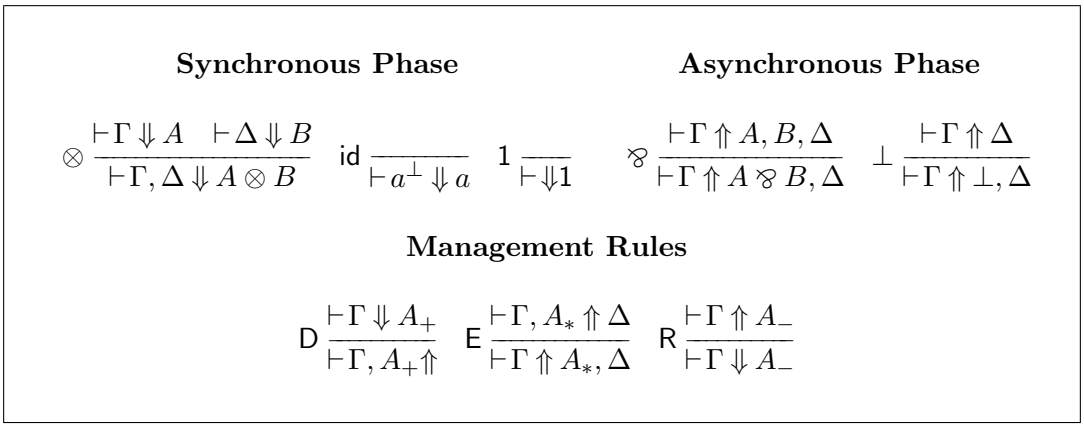


Figure 1: Inference rules for system $\mathsf{MLL}^{\mathsf{F}}$

Once this system has been defined, it is necessary to prove that it can be used to replace the usual systems. This is the actual focusing result, which was originally established in the broader case of full linear logic.

Theorem 2.2 (Andreoli, 1992)**.** *The* $\mathsf{MLL}^{\mathsf{F}}$ *system is sound and complete with respect to multiplicative linear logic.*

Searching for proofs in this calculus is much simpler than in unfocused systems, since important points of non-determinism are isolated : context splitting induced by the $\otimes$ rule is a possible point of backtracking, while asynchronous rules cannot induce any backtracking since they are invertible. Moreover, the choice of the next formula to be treated is also controlled using the decision rule, which is the most important source of non-determinism in practice, since context splitting can be implemented in a *lazy* way [Cer00]. The grouping of choices in the synchronous phase reduces the search space, so that the proof construction process is still efficient, while following a uniform strategy.

3. The $\mathsf{MLS}^{\mathsf{F}}$ Focused Calculus of Structures

The starting point for the definition of a focused system in the calculus of structures is the existing system LS for linear logic [Str03], where the multiplicative fragment MLS is very small, using only two inference rules : the *identity* rule $\mathsf{i}{\downarrow}$ and the *switch* rule s. There is no rule for the $\parr$, since there is no *meta-level* connective such as the comma, but we make the treatment of units explicit here, although it is implicit in LS, because we are concerned with proof search. The equational theory only treats associativity and commutativity. Then, a complete proof in LS is a derivation that ends with a 1 alone as a premise.

Adapting the focusing technique to the calculus of structures requires to define a syntax that enforces the correct shape for proofs through annotations added to the usual system. The resulting system is called $\mathsf{MLS}^{\mathsf{F}}$, and its equational theory and inference rules are given in Figure 2. Beyond annotations, it uses the same classes of formulas as $\mathsf{MLL}^{\mathsf{F}}$. It also uses a global flag, that can be either 0 or 1 to indicate that we are in a *decision* or *focusing* phase respectively. Indeed, the structure of phases is different from the sequent calculus: the asynchronous and synchronous phases are merged into the focusing phase, and the decision phase corresponds to the application of the decision rule.

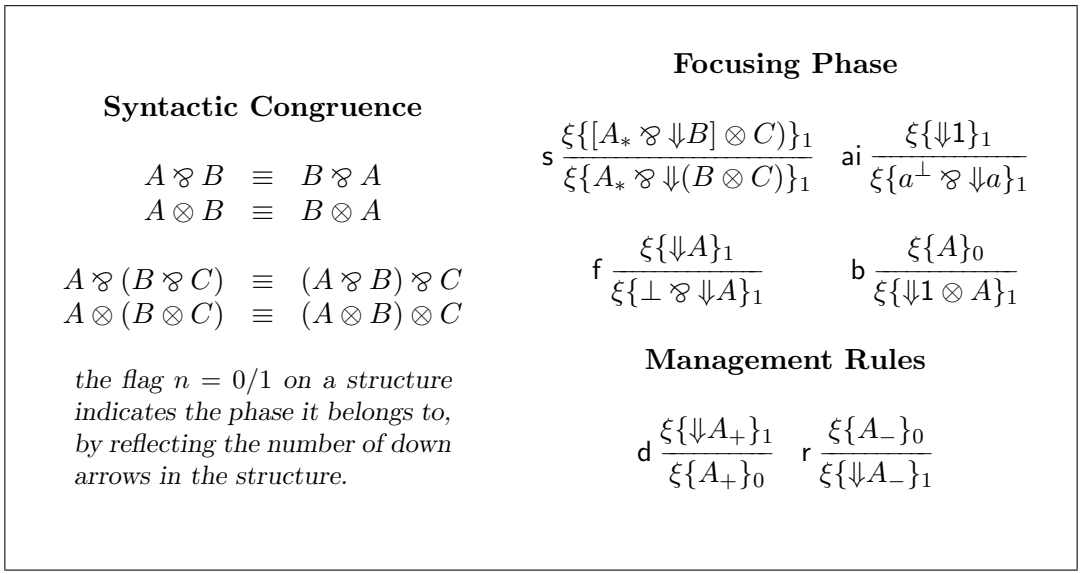


Figure 2: Congruence and inference rules for system MLS^F

Although it is built on the same idea as focused proof systems in the sequent calculus, there are several important differences, due to the deep inference setting, that should be explained concerning MLS^F and its proof construction dynamics:

- The proof search process is a sequence of focusing phases, delimited by applications of the decision rule d. A focusing phase represents the interaction between a positive formula and another formula in its surrounding context. As would be done during an asynchronous phase, some $\perp$ formulas can be erased in such a focusing phase.
- The *switch* rule s implements lazy splitting within the proof-theoretic framework, so that branching is decomposed into the stepwise distribution of formulas in the context to different sides of a $\otimes$ connective. Thus, formulas are moved hereditarily within a compound $\otimes$ structure, and we obtain a *uniform translation* of the focusing idea into the calculus of structures, in the sense that it provides proportionately the same restriction as in the sequent calculus, although it is a weaker normal form.
- An important consequence of the focusing result is the natural definition of *synthetic connectives* [Zei08]. Here, the definition of positive synthetic connectives is clear, but there is no equivalent on the negative side. However, this corresponds to the asymmetry of focusing, that emphasizes the importance of the positives. Indeed, the reason we decompose negatives first is simply that a part of them should be treated before the next positive connective to be chosen. It is even impossible to observe negative synthetic connectives in the conclusive sequent of a proof, since they are built during the proof construction process. The MLS^F system acknowledges this fact by orienting proof search with the help of positives only, and defining deduction steps in terms of interaction between a positive formula and its context.

Soundness is a trivial result for such a system, since all of its rules are sound once the annotations have been removed. Then, completeness can be proved by using a translation from proofs in MLL^F to proofs in our system in the calculus of structures. This requires to define a translation from both kinds of sequents to structures with annotations.

Definition 3.1. The translation $[\![\cdot]\!]_{\mathsf{S}}$ from $\mathsf{MLL}^{\mathsf{F}}$ sequents to $\mathsf{MLS}^{\mathsf{F}}$ structures is defined using the translation $[\![\cdot]\!]_{\mathsf{F}}$ from multisets of formulas to structures, as follows:

$$[\![\vdash \Gamma \Uparrow \Delta]\!]_{\mathsf{S}} = [\![\Gamma]\!]_{\mathsf{F}} \parr [\![\Delta]\!]_{\mathsf{F}} \qquad [\![A_1, \cdots, A_n]\!]_{\mathsf{F}} = A_1 \parr \cdots \parr A_n$$
$$[\![\vdash \Gamma \Downarrow A]\!]_{\mathsf{S}} = [\![\Gamma]\!]_{\mathsf{F}} \parr \Downarrow [\![A]\!]_{\mathsf{F}}$$

The translation of sequent calculus focused proofs into our system is not as easy as the translation of formulas. Indeed, the differences in the way phases are organised and the decomposition of $\otimes$ splitting induce a different shape of proofs. Therefore, we have to use an intermediate system, called $\mathsf{MLS}^{\mathsf{F}}_{\mathsf{s}}$, where the s and f rules are replaced with the following variants, closer to the sequent calculus:

$$\mathsf{f}_{\mathsf{s}} \frac{\xi\{A\}_0}{\xi\{\bot \parr A\}_0} \qquad \mathsf{s}_{\mathsf{s}} \frac{\xi\{[A_\bullet \parr \Downarrow B] \otimes C)\}_1}{\xi\{A_\bullet \parr \Downarrow (B \otimes C)\}_1} \qquad \textit{where} \quad A_\bullet ::= A_* \mid A_* \parr A_\bullet$$

This system relaxes the restriction on the f rule, so that $\bot$ can be erased anywhere, between two focusing phases, and the switch rule is more general. We now prove completeness for this system and we will then show how to translate these proofs into $\mathsf{MLS}^{\mathsf{F}}$ proofs.

Lemma 3.2. *For any proof a sequent $\vdash \mathcal{M}$ in $\mathsf{MLL}^{\mathsf{F}}$, there is a proof of $[\![\vdash \mathcal{M}]\!]_{\mathsf{S}}$ in $\mathsf{MLS}^{\mathsf{F}}_{\mathsf{s}}$.*

Proof. By translation of a given proof Π in $\mathsf{MLL}^{\mathsf{F}}$ to a proof in the focused system $\mathsf{MLS}^{\mathsf{F}}_{\mathsf{s}}$, using a case analysis and an induction on the height of this proof tree. The base case happens when translating axiomatic rule instances, for which the resulting derivation should end with the premise $\Downarrow 1$.

(i) The case of a 1 rule is immediate, since the conclusion is $\Downarrow 1$.

(ii) An identity rule id is directly translated as an identity ai rule:

$$\mathsf{id} \frac{}{\vdash a^\bot \Downarrow a} \quad \longrightarrow \quad \mathsf{ai} \frac{(\Downarrow 1)_1}{[a^\bot \parr \Downarrow a]_1}$$

The inductive cases use the proofs produced by applying the induction hypothesis, which can be plugged into the derivation being build, because of the deep inference setting — and this is done in accordance to the global flag. In the following, we make explicit the use of the syntactic congruence by using a fake $\equiv$ rule whenever a structure is rewritten into another. Moreover, we must be cautious when translating the $\otimes$ rule since there are two different cases, depending on the polarity of the formulas on both sides of the $\otimes$ connective being decomposed.

(iii) If there is at least one positive structure on one side of the $\otimes$, we can use it to apply a decision d rule in the end of the proof:

$$\otimes \frac{\overset{\Pi_A}{\vdash \Gamma \Downarrow A} \qquad \overset{\Pi_B}{\vdash \Delta \Downarrow B_+}}{\vdash \Gamma, \Delta \Downarrow A \otimes B_+} \quad \longrightarrow \quad \begin{array}{c} \mathsf{b} \dfrac{\overset{\Lambda_B}{\Big\Vert} \; \mathsf{d} \dfrac{[\Delta \parr \Downarrow B_+]_1}{[\Delta \parr B_+]_0}}{[\Delta \parr (\Downarrow 1 \otimes B_+)]_1} \\ \mathsf{s}_{\mathsf{s}} \dfrac{\overset{\Lambda_A}{\Big\Vert}}{[\Gamma \parr \Delta \parr \Downarrow (A \otimes B_+)]_1} \end{array}$$

(iv) In the other case, there are negative structures on both sides of the $\otimes$, and focusing enforces the use of the release rule:

$$\dfrac{\dfrac{\dfrac{\Pi_A}{\vdash \Gamma \Uparrow A_-}}{\vdash \Gamma \Downarrow A_-}\,\mathsf{R} \quad \dfrac{\dfrac{\Pi_B}{\vdash \Delta \Uparrow B_-}}{\vdash \Delta \Downarrow B_-}\,\mathsf{R}}{\vdash \Gamma, \Delta \Downarrow A_- \otimes B_-}\,\otimes \quad \longrightarrow \quad \dfrac{\dfrac{\dfrac{\dfrac{\Lambda_B}{[\Delta \parr B_-]_0}}{[\Delta \parr (\Downarrow \mathsf{1} \otimes B_-)]_1}\,\mathsf{b}}{[\Delta \parr ([\Gamma \parr A_-] \otimes B_-)]_0}\,\mathsf{r}}{[\Gamma \parr \Delta \parr \Downarrow (A_- \otimes B_-)]_1}\,\mathsf{s_s}$$

(v) The $\parr$ rule is not really translated since it has no effect on the translation:

$$\dfrac{\dfrac{\Pi'}{\vdash \Gamma \Uparrow A, B, \Delta}}{\vdash \Gamma \Uparrow A \parr B, \Delta}\,\parr \quad \longrightarrow \quad \equiv \dfrac{\dfrac{\Lambda'}{[\Gamma \parr A \parr B \parr \Delta]_0}}{[\Gamma \parr [A \parr B] \parr \Delta]_0}$$

(vi) The $\bot$ rule is simply translated as a $\mathsf{f_s}$ rule:

$$\dfrac{\dfrac{\Pi'}{\vdash \Gamma \Uparrow \Delta}}{\vdash \Gamma \Uparrow \bot, \Delta}\,\parr \quad \longrightarrow \quad \dfrac{\dfrac{\Lambda'}{[\Gamma \parr \Delta]_0}}{[\Gamma \parr \bot \parr \Delta]_0}\,\mathsf{f_s}$$

(vii) The decision rules D and d are used the same way in both systems:

$$\dfrac{\dfrac{\Pi'}{\vdash \Gamma \Downarrow A_+}}{\vdash \Gamma, A_+ \Uparrow}\,\mathsf{D} \quad \longrightarrow \quad \dfrac{\dfrac{\Lambda'}{[\Gamma \parr \Downarrow A_+]_1}}{[\Gamma \parr A_+]_0}\,\mathsf{d}$$

(viii) The exclusion rule E is not translated at all, since not required:

$$\dfrac{\dfrac{\Pi'}{\vdash \Gamma, A_* \Uparrow \Delta}}{\vdash \Gamma \Uparrow A_*, \Delta}\,\mathsf{E} \quad \longrightarrow \quad \dfrac{\Lambda'}{[\Gamma \parr A_* \parr \Delta]_0}$$

(ix) The release rules R and r are also used the same way in both systems:

$$\dfrac{\dfrac{\Pi'}{\vdash \Gamma \Uparrow A_-}}{\vdash \Gamma \Downarrow A_-}\,\mathsf{R} \quad \longrightarrow \quad \dfrac{\dfrac{\Lambda'}{[\Gamma \parr A_-]_0}}{[\Gamma \parr \Downarrow A_-]_1}\,\mathsf{r}$$

■

Finally, we can establish the focusing result, which states that the $\mathsf{MLS}^{\mathsf{F}}$ proof system is sound and complete, by using our translations. Unfortunately, this result does not allow a comparison with the sequent calculus on the level of focusing, since it is difficult to give a formal account of the reduction of the proof search space induced by both systems. The benefits of our focusing restriction are yet to be further studied.

Theorem 3.3. $\mathsf{MLS}^{\mathsf{F}}$ *is sound and complete with respect to multiplicative linear logic.*

Proof. Soundness is immediately obtained by removing arrows from inference rules in $\mathsf{MLS}^{\mathsf{F}}$, leaving us with the rules of MLS, and trivialised management rules with the same structure as premise and conclusion. Completeness is obtained in two steps. First, given a sequent calculus proof Π of $\vdash \mathcal{M}$ we produce a proof Λ of $[\![\vdash \mathcal{M}]\!]_{\mathsf{S}}$ in $\mathsf{MLS}^{\mathsf{F}}_{\mathsf{s}}$ using Lemma 3.2. Then we have to produce a proof of $[\![\vdash \mathcal{M}]\!]_{\mathsf{S}}$ in $\mathsf{MLS}^{\mathsf{F}}$, by modifying Λ.

Because of the shape of sequent calculus proofs, all instances of f_{s} in Λ are located below an instance of d in the surrounding context. It is thus possible to permute all instances of f_{s} above the associated decision rule, turning them into instances of f, as follows:

$$\mathsf{f}_{\mathsf{s}}\dfrac{\mathsf{d}\dfrac{\xi\{\Gamma \parr \Downarrow A_+ \parr \Delta\}_1}{\xi\{\Gamma \parr A_+ \parr \Delta\}_0}}{\xi\{\Gamma \parr \bot \parr A_+ \parr \Delta\}_0} \quad\longrightarrow\quad \mathsf{d}\dfrac{\mathsf{f}\dfrac{\xi\{\Gamma \parr \Downarrow A_+ \parr \Delta\}_1}{\xi\{\Gamma \parr \bot \parr \Downarrow A_+ \parr \Delta\}_1}}{\xi\{\Gamma \parr \bot \parr A_+ \parr \Delta\}_0}$$

Then, we have to reorganise sequences of s_{s} instances into sequences of s instances. Again, because of the shape of sequent calculus proofs, we know that instances of s_{s}, if they cannot be read as simple instances of s, follow a precise scheme where the reaction rule is applied above a sequence of s_{s} instances. We can thus rewrite the derivation as follows:

$$\mathsf{s}^*_{\mathsf{s}}\dfrac{\mathsf{r}\dfrac{\xi\{\zeta\{A \parr B_* \parr C_-\}\}_1}{\xi\{\zeta\{A \parr B_* \parr \Downarrow C_-\}\}_1}}{\xi\{[A \parr B_*] \parr \Downarrow\zeta\{C_-\}\}_1} \quad\longrightarrow\quad \mathsf{s}^*\dfrac{\mathsf{d}\dfrac{\mathsf{r}\dfrac{\xi\{\zeta\{A \parr B_* \parr C_-\}\}_0}{\xi\{\zeta\{A \parr \Downarrow[B_* \parr C_-]\}\}_1}}{\xi\{A \parr \Downarrow\zeta\{B_* \parr C_-\}\}_1}}{\xi\{[A \parr B_*] \parr \Downarrow\zeta\{C_-\}\}_1}\mathsf{r}\dfrac{\xi\{A \parr \zeta\{B_* \parr C_-\}\}_0}{\xi\{A \parr \zeta\{B_* \parr \Downarrow C_-\}\}_1}$$

By a simple induction on the size of the structure being moved deep inside the context $\zeta\{\cdot\}$, we can replace all instances of s_{s} with instances of s. Therefore, Λ can be turned into a proof Λ' of $[\![\vdash \mathcal{M}]\!]_{\mathsf{S}}$ in $\mathsf{MLS}^{\mathsf{F}}$. ∎

4. Variations and Extensions

The system we devised corresponds to one possible design choice among many others. It has been chosen because of its simplicity, and its use of the important feature of lazy splitting introduced by the deep inference methodology. This is probably an important feature from the viewpoint of logic programming, since it allows for clean proof-theoretic foundations for the context sharing implementation technique. However, it is unclear which are the features that could further improve proof search procedures with respect to this programming consideration, and if they would be compatible with the design choices we made so far. We discuss now some variations of the $\mathsf{MLS}^{\mathsf{F}}$ system, as well as its extension to larger fragments of the logic and stronger normal forms.

Complete $\otimes$ Decomposition. An alternative to the current design of the switch rule s is the use of a maximality condition, imposing that all elements of the immediate surrounding context are moved inside the $\otimes$ structure, along with a one-step splitting variation of the switch rule, as follows:

$$\mathsf{s}\,\frac{\xi\{[A \parr \Downarrow C] \otimes [B \parr \Downarrow D]\}_2}{\xi\{[A \parr B] \parr \Downarrow(C \otimes D)\}_1}$$

However, this would require to extend the global flag to a general counter, which would track the number of down arrows in the structure. More importantly, this would betray the idea of small-step rules decomposition that comes as a benefit when using the calculus of structures. Such a complete decomposition does not fit the deep inference methodology, since it is impossible to obtain negative synthetic connectives using this approach, unless we mimick sequent calculus proofs.

Separate Asynchronous Phase. It is possible to use another annotation to control the asynchronous rules, thus splitting the focusing phase into asynchronous and synchronous phases, as it is done in the sequent calculus. However, the asynchronous phase would still be directed by the future synchronous phase, since we must ensure that all required negative structures have been treated before we start treating a positive. Such a design would require to relate these two new phases through the interaction of annotated formulas, as can be done with the following alternative rules:

$$\mathsf{s}\,\frac{\xi\{[\Uparrow A_* \parr \Downarrow B] \otimes C)\}}{\xi\{\Uparrow A_* \parr \Downarrow(B \otimes C)\}} \qquad \mathsf{ai}\,\frac{\xi\{\Downarrow 1\}}{\xi\{\Uparrow a^\perp \parr \Downarrow a\}} \qquad \mathsf{f}\,\frac{\xi\{A\}}{\xi\{\Uparrow \perp \parr A\}}$$

With such an extension of the annotations, we have to extend the global flag, so that the presence of up arrows is tracked too. This complicates the managment rules, and does not provide any benefit compared to MLS^F if we only allow for one up arrow in a structure. Then, multiple up arrows would allow for a compromise between small-step and big-step splitting of the $\otimes$, but it requires to use extend the use of down arrows too. In order to allow medium-step splitting, it seems simpler to use the $\mathsf{MLS}^\mathsf{F}_\mathsf{s}$ intermediate system used in the completeness proof, where a group of structures can be moved inside a positive.

Extension to the Additives. Using the additive connectives $\&$ and $\oplus$ in the setting of our focused calculus of structures yields new design choices that should be studied carefully. An intuitive way of extending the system would be to add the two following rules:

$$\mathsf{a}\,\frac{\xi\{[\Downarrow A \parr B] \,\&\, [\Downarrow A \parr C]\}_2}{\xi\{\Downarrow A \parr (B \,\&\, C)\}_1} \qquad \mathsf{o}\,\frac{\xi\{\Downarrow A\}_1}{\xi\{\Downarrow[A \oplus B]\}_1}$$

The focused rule for the $\oplus$ is perfectly fine, but the rule for $\&$ is more problematic. Indeed, the duplication of the down arrow implies that the global flag has to count these arrows, as considered for the alternate switch rule. Moreover, the treatment of the left and right branches of the $\&$ are now interleaved, although a unique active positive structure provides us with a stronger normal form. Therefore, the following rule should be considered:

$$\mathsf{a}\,\frac{\xi\{[\Downarrow A \parr B] \,\&\, [A \parr C]\}_1}{\xi\{\Downarrow A \parr (B \,\&\, C)\}_1}$$

This rule corresponds to the viewpoint of *slices* on the additives, each slice being treated separately. Moreover, this fits the idea that a focusing phase represents the interaction between a positive structure and only one of the element of its context — its context being more structured here than a simple multiset created by $\parr$.

Multi-focusing. The usual notion of focusing can be extended to provide a stronger normal form, by handling multiple focused formulas [Cha08]. This idea seems quite natural in the setting of $\mathsf{MLS}^{\mathsf{F}}$, where the global flag can be extended into a counter, as mentioned above. There are three different ways of using multiple arrows in such a system:

- *Parallel focusing:* when arrows are located in different subformulas of a conjunction connective, they represent the parallel focusing of different branches in the sequent calculus. It is easy to handle multiple arrows in this case, but it is unclear how this could yield a stronger normal form.
- *Multi-focusing:* the use of multiple focusing arrows with the same $\parr$ context is more tricky. This is the case corresponding to the notion of multi-focusing in the sequent calculus, and it is difficult to know if two given positive structures can be focused at the same time — naïve maximal multi-focusing is not complete.
- *Nested focusing:* it seems easy to handle cases where an arrow annotates a positive structure located deep inside another focused structure. Moreover, this suggests an extension of synthetic connectives to disconnected layers of positive connectives, which could be interesting in the search for a stronger normal form.

5. Conclusion and Future Work

The $\mathsf{MLS}^{\mathsf{F}}$ system presented here is a first step in the larger project of extracting the focusing notion out of its sequent calculus roots. This is a prototype that should be further studied and improved, since although we know there are some benefits in using the calculus of structures as an host formalism for proof search and logic programming — for instance, lazy splitting and shorter proofs —, it is yet unclear how to gain control over its overwhelming non-determinism, especially for larger logics, such as full linear logic or even classical logic [Brü03].

The next step in this research project is to design a focused proof system for the multiplicative-additive fragment of linear logic in the calculus of structures, and solve there the problem of controlling duplications of focused formulas. Then, it might not be more difficult to accomodate the exponentials, since they do not quite fit the focusing categories in the sequent calculus — they are located at the interface between polarity groups.

In order to assess the benefits of the deep inference methodology in the definition of proof search procedures, it also seems necessary to experiment with our system. Thus, a prototype implementation of such a focused system should be carried out, so that it can be compared to other proof search software developed on top of the focusing approach [Bae07]. Choosing among the many possible designs requires to get experience with proof search in the calculus of structures, and it might even be needed to develop specific variants of the usual focusing, so that the dynamics of deep inference proof search can be used with full power — indeed, it would be useless to try to mimick the proof search behaviour of the sequent calculus, while the benefits are in the proofs that cannot be written in a shallow way. Finally, the study of deep inference proof methods raises the question of its status: is it only a way to implement more efficient proof search procedures with clean semantics, or will it yield new features to be used in logic programming languages ?

Acknowledgements

This work has benefited from useful comments and suggestions of the anonymous reviewers, and fruitful discussions with Kaustuv Chaudhuri about the design of focused systems.

References

[And92] J-M. Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3):297–347, 1992.

[Bae07] D. Baelde and D. Miller. Least and greatest fixed points in linear logic. In N. Dershowitz and A. Voronkov (eds.), *LPAR'07*, *LNCS*, vol. 4790, pp. 92–106. 2007.

[Brü03] K. Brünnler. *Deep Inference and Symmetry in Classical Proofs*. Ph.D. thesis, Technische Universität Dresden, 2003.

[Bru09] P. Bruscoli and A. Guglielmi. On the proof complexity of deep inference. *ACM Transactions on Computational Logic*, 10(2):1–34, 2009.

[Cer00] I. Cervesato, J. S. Hodas, and F. Pfenning. Efficient resource management for linear logic proof search. *Theoretical Computer Science*, 232(1-2):133–163, 2000.

[Cha08] K. Chaudhuri, D. Miller, and A. Saurin. Canonical sequent proofs via multi-focusing. In G. Ausiello, J. Karhumäki, G. Mauri, and L. Ong (eds.), *Fifth IFIP International Conference on Theoretical Computer Science*, vol. 273, pp. 383–396. 2008.

[Gir87] J-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[Gug07] A. Guglielmi. A system of interaction and structure. *ACM Transactions on Computational Logic*, 8(1):1–64, 2007.

[Kah06] O. Kahramanoğulları. *Nondeterminism and Language Design in Deep Inference*. Ph.D. thesis, Technische Universität Dresden, 2006.

[Lia09] C. Liang and D. Miller. Focusing and polarization in linear, intuitionistic, and classical logics. *Theoretical Computer Science*, 410(46):4747–4768, 2009.

[Mil91] D. Miller, G. Nadathur, F. Pfenning, and A. Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

[Mil07] D. Miller and A. Saurin. From proofs to focused proofs : a modular proof of focalization in linear logic. In J. Duparc and T. A. Henzinger (eds.), *CSL'07*, *LNCS*, vol. 4646, pp. 405–419. 2007.

[Str03] L. Straßburger. *Linear Logic and Noncommutativity in the Calculus of Structures*. Ph.D. thesis, Technische Universität Dresden, 2003.

[Zei08] N. Zeilberger. Focusing and higher-order abstract syntax. In G. Necula and P. Wadler (eds.), *POPL'08*, pp. 359–369. 2008.

Technical Communications of the International Conference on Logic Programming, 2010 (Edinburgh), pp. 94–103
http://www.floc-conference.org/ICLP-home.html

SAMPLER PROGRAMS: THE STABLE MODEL SEMANTICS OF ABSTRACT CONSTRAINT PROGRAMS REVISITED

TOMI JANHUNEN

Aalto University School of Science and Technology
Department of Information and Computer Science
PO Box 15400, FI-00076 Aalto, Finland
E-mail address: Tomi.Janhunen@tkk.fi

Abstract. Abstract constraint atoms provide a general framework for the study of aggregates utilized in answer set programming. Such primitives suitably increase the expressive power of rules and enable more concise representation of various domains as answer set programs. However, it is non-trivial to generalize the stable model semantics for programs involving arbitrary abstract constraint atoms. For instance, a nondeterministic variant of the immediate consequence operator is needed, or the definition of stable models cannot be stated directly using primitives of logic programs. In this paper, we propose sampler programs as a relaxation of abstract constraint programs that better lend themselves to the program transformation involved in the definition of stable models. Consequently, the declarative nature of stable models can be restored for sampler programs and abstract constraint programs are also covered if decomposed into sampler programs. Moreover, we study the relationships of the classes of programs involved and provide a characterization in terms of abstract but essentially deterministic computations. This result indicates that all nondeterminism related with abstract constraint atoms can be resolved at the level of program reduct when sampler programs are used as the intermediate representation.

1. Introduction

The stable model semantics [Gel88] of logic programs, also known as the answer set semantics, constitutes the semantical cornerstone of *answer set programming* (ASP). Undoubtedly, the simple and intuitive definition of stable models [Lif08] has played a major role in the success of ASP during the past two decades. Applications that emerged in the meantime demonstrate that knowledge engineers have easily grasped the essentials of rules subject to stable models. Nevertheless, the practise of ASP has led to a rich body of extensions to the basic syntax of *normal logic programs* such as strong negation [Gel90], disjunctions [Gel91], and various kinds of *aggregates*, which have also appeared in other similar disciplines. Extensions in the last category typically enable concise expression of a particular combinatorial condition involving a set of atoms or objects. Examples of aggregates supported by contemporary ASP solvers include the *cardinality* and *weight constraints* [Sim99] and the *sum*, *count*, and *max* aggregates [Del03]. To get a concrete idea of their power,

1998 ACM Subject Classification: I.2.4, F.4.1.
Key words and phrases: stable models, abstract constraints, program reduction, translation, choice rules.
This research has been partially funded by the Academy of Finland under project #122399.

Technical Communications of the 26th International Conference on Logic Programming, Edinburgh, July, 2010
Editors: Manuel Hermenegildo, Torsten Schaub
LIPIcs - Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/LIPIcs.ICLP.2010.94

consider a sum aggregate $500 \leq \mathsf{sum}\{Y : \mathsf{capacity}(X,Y) : \mathsf{in}(X) : \mathsf{disk}(X)\}$ formalizing the sufficiency of disk space selected for a particular PC configuration. Such an aggregate requires no updates when the number of disks for configuring PCs is changed.

The study of aggregates has recently lifted to the level of *abstract constraint atoms* which nicely capture a variety of important aggregates. However, it is non-trivial to generalize stable models for arbitrary abstract constraint atoms as witnessed by the number of proposals in this respect. Only the interpretation of special cases, viz. *monotone* [Mar08] and *convex* abstract constraints [Liu06], is unanimous. As regards the general case, abstract *computations*, i.e., sequences of interpretations associated with a program, have been proposed as a semantical basis [Liu10]. A key justification is that *persistent* deterministic computations essentially capture stable models of *normal logic programs*. This interconnection suggests an alternative way of defining the semantics of abstract constraint programs, but computations bring along nondeterminism and other degrees of freedom that pre-empt a conclusive semantical definition. Besides, the declarative nature of stable models is jeopardized because the outcome of a computation potentially depends on the entire sequence.

Our hypothesis is that abstract constraint programs lack a natural counterpart of the Gelfond-Lifschitz reduct [Gel88] which plays a key role in the definition of stable models. For instance, in case of a normal logic program P, a stable model M is defined as the *least model* of P^M, i.e., the program P reduced with respect to M. Attempts to generalize this idea for abstract constraint programs become intricate because the reduced program cannot be directly represented as an abstract constraint program. For instance, the representation proposed in [She07] requires new atoms and it effectively produces a positive *normal program*. Alternatively, propositional (default) logic has been used to formalize the reduct [She09a].

In this paper, we address the aforementioned deficiency related to reducts by proposing a completely new class of programs, viz. *sampler programs*. They form a relaxation of abstract constraint programs so that a reasonable notion of a reduct can be established within the class of sampler programs. This class of programs is introduced as follows. First, in Section 2, we recall a much simpler class of *choice logic programs* [Soi99] designed for modelling *product configurations*. The syntax is based on a slight extension of normal rules—enabling a straightforward generalization of stable models. Nevertheless, it provides us with insights into how stable models can be lifted to the case of sampler programs as carried out in Section 3. The relationship of sampler programs and choice logic programs is then explored in terms of translations in Section 4. The translations presented in this case indicate that choice logic programs and sampler programs are equally expressive [Jan06].

In the second part of this paper, we apply the theory developed for sampler programs to abstract constraint programs, which are first recalled in Section 5. The idea is to decompose abstract constraint atoms into sets of samplers systematically. In this way, we are able to define stable models in the context of abstract constraint programs in a traditional way [Gel88] using the notions of a program reduct and the least model. This aspect restores the declarative nature of stable models and makes our approach original due to its simplicity. The semantics obtained in this way coincides with the one proposed in [She07, She09a, She09b]. Moreover, motivated by the computation-based approach [Liu10], we propose a notion of *canonical computations* for abstract constraint programs in Section 6. The novelty is that all nondeterminism can be handled globally via the definition of stable models, and resorting to nondeterministic variants of the immediate consequence operator or *conditional satisfaction* [Son07b] can be avoided altogether. A comparison with related work is carried out in Section 7. Finally, we present our conclusions in Section 8.

2. Choice Logic Programs (CLPs)

In this section, we introduce the class of *choice logic programs* [Soi99] that suit well for nondeterministic specifications. A choice logic program (CLP) P is a set of *choice rules*

$$a_1 \mid \ldots \mid a_l \leftarrow b_1, \ldots, b_m, {\sim}c_1, \ldots, {\sim}c_n \tag{2.1}$$

where $l \geq 0$, $m \geq 0$, $n \geq 0$, and $a_1, \ldots, a_l$, $b_1, \ldots, b_m$, and $c_1, \ldots, c_n$ are *propositional atoms*, or just *atoms* for short. A rule is *normal*, iff $l = 1$, and an *integrity constraint* (IC), iff $l = 0$. A *fact* is a normal rule with $l = 1$, $m = 0$, and $n = 0$, and thus written briefly as $a_1 \leftarrow$.

The *signature* of a CLP P, denoted by $\mathrm{At}(P)$, is the set of atoms appearing in its rules. An *interpretation* $I \subseteq \mathrm{At}(P)$ of P determines which atoms of $\mathrm{At}(P)$ are *true* ($a \in I$) and which *false* ($a \notin I$). A *positive literal* b is satisfied in an interpretation, denoted $I \models b$, iff $b \in I$. A *negative literal* ${\sim}c$ is satisfied in I, denoted $I \models {\sim}c$, iff $c \notin I$. A conjunction $l_1, \ldots, l_n$ of literals is satisfied in I, denoted $I \models l_1, \ldots, l_n$, iff $I \models l_1$, $\ldots$, and $I \models l_n$. A disjunction $a_1 \mid \ldots \mid a_l$ of atoms is satisfied in I, denoted $I \models a_1 \mid \ldots \mid a_l$, iff $I \models a_i$ holds for some $i \in \{1, \ldots, l\}$. A choice rule of the form (2.1) is satisfied in I iff $I \models b_1, \ldots, b_m$ and $I \models {\sim}c_1, \ldots, {\sim}c_n$ imply $I \models a_1 \mid \ldots \mid a_l$. An interpretation $M \subseteq \mathrm{At}(P)$ for a CLP P is called a *model* of P, denoted by $M \models P$, iff every choice rule (2.1) of P is satisfied by M.

Definition 2.1 (Reduct [Soi99])**.** The reduct P^M of a CLP P with respect to an interpretation $M \subseteq \mathrm{At}(P)$ contains a positive rule $a \leftarrow b_1, \ldots, b_m$ for each choice rule (2.1) such that (i) $a \in \{a_1, \ldots, a_l\}$, (ii) $M \models a$, and (iii) $M \models {\sim}c_1, \ldots, {\sim}c_n$.

The reduced program P^M is a *positive* normal program having rules of the form $a \leftarrow b_1, \ldots, b_m$ where $m \geq 0$. Such a program P has a unique $\subseteq$-minimal model, also known as the *least model* of P hereafter denoted by $\mathrm{LM}(P)$. Stable models are defined as follows.

Definition 2.2 (Stable Model [Soi99])**.** An interpretation $M \subseteq \mathrm{At}(P)$ is a stable model of a CLP P iff $M \models P$ and $M = \mathrm{LM}(P^M)$. The set of stable models of P is denoted by $\mathrm{SM}(P)$.

This definition coincides with [Gel88] when $l = 1$ for every rule (2.1). Moreover, any ICs contained in P do not contribute to P^M and their satisfaction is enforced by the condition $M \models P$ above. In fact, this condition implies $M \models P^M$, and thus also $\mathrm{LM}(P^M) \subseteq M$, but the converse does not hold in general. Consider, e.g., the CLP $P = \{\leftarrow b, {\sim}c\}$ and $M = \{b\}$.

Example 2.3. We note that $P = \{a \mid b \mid c \leftarrow {\sim}d\}$ has seven stable models $\{a\}$, $\{b\}$, $\{c\}$, $\{a, b\}$, $\{a, c\}$, $\{b, c\}$, $\{a, b, c\}$. To verify the last but one model, i.e., $M = \{b, c\}$, we observe that $M \models P$ and $P^M = \{b \leftarrow;\ \ c \leftarrow\}$[1] so that $\mathrm{LM}(P^M) = \{b, c\}$ coincides with M.

Let us stress that a *choice rule* $\{a_1, \ldots, a_l\} \leftarrow b_1, \ldots, b_m, {\sim}c_1, \ldots, {\sim}c_n$ in the style of SMODELS [Sim02] can be captured with $a_1 \mid \ldots \mid a_l \mid e \leftarrow b_1, \ldots, b_m, {\sim}c_1, \ldots, {\sim}c_n$ and $e \leftarrow$.

3. Sampler Programs (SPs)

Our next objective is to develop the theory of *sampling atoms*, or *samplers* for short, and to propose a completely new class of logic programs based on them.

Definition 3.1 (Sampler)**.** A *sampling atom*, or a *sampler* for short, π is a triple $\langle D, L, G \rangle$ where the *domain* $\pi_{\mathbf{D}} = D$ of π is a *finite* set of atoms and $L \subseteq G \subseteq D$. The sets L and G are the *least* and the *greatest satisfier* of π, also denoted by $\pi_{\mathbf{L}}$ and $\pi_{\mathbf{G}}$, respectively.

[1] For clarity, semicolons are used to separate rules in programs.

The basic intuition behind a sampler π is that it provides a compact representation for the set of literals $\{a \mid a \in \pi_{\mathbf{L}}\} \cup \{\sim a \mid a \in \pi_{\mathbf{D}} \setminus \pi_{\mathbf{G}}\}$. Therefore, we define that π is *satisfied* in an interpretation I, denoted by $I \models \pi$, iff $\pi_{\mathbf{L}} \subseteq I \cap \pi_{\mathbf{D}} \subseteq \pi_{\mathbf{G}}$. This definition justifies the name of the new primitive: the projection of I with respect to $\pi_{\mathbf{D}}$ can be viewed as a *sample* of the interpretation I. In order to satisfy π, the sample must be in the *range* determined by $\pi_{\mathbf{L}}$ and $\pi_{\mathbf{G}}$, i.e., a superset of $\pi_{\mathbf{L}}$ and a subset of $\pi_{\mathbf{G}}$. The set of *satisfiers* of a sampler π, denoted $\pi_{\mathbf{S}}$, is $\{S \subseteq \pi_{\mathbf{D}} \mid \pi_{\mathbf{L}} \subseteq S \subseteq \pi_{\mathbf{G}}\}$. A sampler π is called *exact*, if $\pi_{\mathbf{L}} = \pi_{\mathbf{G}}$, and then abbreviated as a pair $\langle D, S\rangle$ where $S = L = G$. Thus positive and negative literals based on an atom a are captured by *primitive* exact samplers of the forms $\langle\{a\}, \{a\}\rangle$ and $\langle\{a\}, \emptyset\rangle$.

We assign a *disjunctive* interpretation to any set of samplers $\Pi = \{\pi_1, \ldots, \pi_k\}$, i.e., $I \models \Pi$ iff $I \models \pi_j$ for some $1 \leq j \leq k$. A *sampler program* (SP) P is a set of *sampling rules* of the form $\Pi \leftarrow \Pi_1, \ldots, \Pi_n$ where Π and each Π_i with $1 \leq i \leq n$ is such a set. In this notation, a singleton $\{\pi\}$ can be abbreviated by π whereas primitive exact samplers $\langle\{a\}, \{a\}\rangle$ and $\langle\{a\}, \emptyset\rangle$ are abbreviated by a and $\sim a$, respectively. The set of *head samplers* that appear in some rule head Π of P is denoted by $\mathrm{HeadS}(P)$. Likewise, we define the *set* $\mathrm{BodySS}(P)$ of sampler sets Π that occur in the rule bodies of P. A sampling rule $\Pi \leftarrow \Pi_1, \ldots, \Pi_n$ is satisfied in an interpretation I iff $I \models \Pi_1, \ldots, I \models \Pi_n$ imply $I \models \Pi$. Intuitively speaking, the body conditions $\Pi_1, \ldots, \Pi_n$ correspond to sets of samples taken of I. If at least one sample in each set Π_i produces the expected outcome, the same must hold for the head Π.

Example 3.2. Consider an infinite SP P having rules $\{p_0, q_0\} \leftarrow$ and $\{p_{i+1}, q_{i+1}\} \leftarrow \{p_i, q_i\}$ for all $i \geq 0$. Here each atom a denotes a primitive exact sampler $\langle\{a\}, \{a\}\rangle$. The latter rules correspond to choice rules $p_{i+1} | q_{i+1} \leftarrow p_i$ and $p_{i+1} | q_{i+1} \leftarrow q_i$ for each $i \geq 0$. Thus the "alternating" interpretation $M = \{p_0, q_1, p_2, q_3, \ldots\}$ is a model of P among others.

Definition 3.3 (Reduct)**.** For an SP P and an interpretation $M \subseteq \mathrm{At}(P)$, the reduct of

(1) a sampler $\pi = \langle D, L, G\rangle$, denoted π^M, is the sampler $\langle G, L, G\rangle$, if $M \models \pi$,
(2) a set Π of samplers, denoted Π^M, is the sampler set $\{\pi^M \mid \pi \in \Pi \text{ and } M \models \pi\}$, and
(3) a sampler program P, denoted by P^M, contains for all $\Pi \leftarrow \Pi_1, \ldots, \Pi_n \in P$ such that $M \models \Pi_1, \ldots, M \models \Pi_n$, and for all $\pi \in \Pi$ such that $M \models \pi$, a reduced sampling rule $\langle S, S\rangle \leftarrow \Pi_1^M, \ldots, \Pi_n^M$ where the exact satisfier $S = M \cap \pi_{\mathbf{G}}$ belongs to $\pi_{\mathbf{S}}$.

The goal of the definition of π^M is to partially evaluate negative default literals in $L = \{\sim a \mid a \in D \setminus G\}$ with respect to $M \models \pi$. For the same reason, we also have $\pi_{\mathbf{L}} \subseteq S \subseteq \pi_{\mathbf{G}}$ for the satisfier S in the last item. Thus $M \models P$ implies $M \models P^M$. The rules of a reduced SP P^M are all *positive* in the following sense: their heads comprise of *single* sampling atoms π satisfying $\pi_{\mathbf{D}} = \pi_{\mathbf{L}} = \pi_{\mathbf{G}}$ and their bodies involve only sampling atoms π with $\pi_{\mathbf{D}} = \pi_{\mathbf{G}}$. Positive SPs share a number of properties with their counterparts amongst normal programs.

Proposition 3.4 (Properties of Positive SPs)**.** *Let P and Q be two positive SPs.*

(1) *If $M_1 \models P$ and $M_2 \models P$ are two models of P, then also $M_1 \cap M_2 \models P$.*
(2) *The program P has a unique $\subseteq$-minimal model, i.e., the* least model $\mathrm{LM}(P)$ *of P which coincides with $\bigcap\{M \subseteq \mathrm{At}(P) \mid M \models P\}$.*
(3) *The least model $\mathrm{LM}(P)$ is the least fixed point $\mathrm{lfp}(\mathbf{T}_P)$ of the immediate consequence operator $\mathbf{T}_P$ defined for any $I \subseteq \mathrm{At}(P)$ by $\mathbf{T}_P(I) =$*

$$\bigcup\{S \mid \pi \leftarrow \Pi_1, \ldots, \Pi_n \in P, \pi_{\mathbf{S}} = \{S\}, \text{ and } I \models \Pi_1, \ldots, I \models \Pi_n\}.$$

Example 3.5. Consider a positive SP P with one sampling rule $\langle\{a\}, \{a\}\rangle \leftarrow \langle\{a\}, \emptyset, \{a\}\rangle$. The interpretation $M_1 = \emptyset$ is not a model of P but $M_2 = \{a\}$ is the least one.

We conclude that SPs provide a reasonable generalization of normal programs and CLPs. Accordingly, the definition of stable models (Definition 2.2) is applicable to SPs as such.

Example 3.6. For the sampler program P and interpretation M from Example 3.2, the reduct P^M is the positive SP $\{p_0 \leftarrow;\ \ q_1 \leftarrow p_0;\ \ p_2 \leftarrow q_1;\ \ q_3 \leftarrow p_2;\ \ \ldots\}$. Thus M is stable as $M \models P$ and $\mathrm{LM}(P^M) = M$. In Example 3.5, $M_2 = \{a\}$ is uniquely stable as $P^{M_2} = P$.

4. Relationship of CLPs and SPs

Let us begin by explaining how choice programs can be viewed as a special case of sampler programs. In this respect, we can fully exploit the conciseness of samplers and abbreviations introduced so far. A choice rule r of the form (2.1) can be rewritten as

$$\mathrm{Tr}_{\mathrm{SP}}(r) \quad = \quad \{a_1, \ldots, a_l\} \leftarrow \langle\{b_1, \ldots, b_m\}, \{b_1, \ldots, b_m\}\rangle, \langle\{c_1, \ldots, c_n\}, \emptyset\rangle. \tag{4.1}$$

In particular, the head of (4.1) is a shorthand for $\{\langle\{a_1\}, \{a_1\}\rangle, \ldots, \langle\{a_l\}, \{a_l\}\rangle\}$ by the notational conventions introduced above—not to be confused with the head of an SMODELS choice rule. The correctness of the program level transformation $\mathrm{Tr}_{\mathrm{SP}}(P) = \bigcup_{r \in P} \mathrm{Tr}_{\mathrm{SP}}(r)$ is formulated below. We omit the proof of this and subsequent theorems for space reasons.

Theorem 4.1 (Correctness of $\mathrm{Tr}_{\mathrm{SP}}$)**.** *For any CLP P,* $\mathrm{SM}(P) = \mathrm{SM}(\mathrm{Tr}_{\mathrm{SP}}(P))$.

Transforming SPs into CLPs is of equal interest. Due to the disjunctive interpretation of sets of sampling atoms, a set of choice rules is required to represent a sampling rule $r = \Pi \leftarrow \Pi_1, \ldots, \Pi_n$ in general. The length of the resulting CLP, denoted by $\mathrm{Tr}_{\mathrm{CLP}}(r)$ in the sequel, can be kept polynomial with respect to $\|r\|$ using new atoms.

Definition 4.2. A sampling rule $\Pi \leftarrow \Pi_1, \ldots, \Pi_n$ with $\Pi = \{\pi_1, \ldots, \pi_l\}$ is translated into

(1) a normal rule $s_i \leftarrow \pi_{\mathbf{L}}, {\sim}(\pi_{\mathbf{D}} \setminus \pi_{\mathbf{G}})$ for each $\pi \in \Pi_i$;
(2) a choice rule $h_1 | \ldots | h_l \leftarrow s_1, \ldots, s_n$;
(3) for each $\pi_i \in \Pi$, an integrity constraint $\leftarrow (\pi_i)_{\mathbf{L}}, s_1, \ldots, s_n, {\sim}h_i, {\sim}((\pi_i)_{\mathbf{D}} \setminus (\pi_i)_{\mathbf{G}})$;
(4) a normal rule $a \leftarrow h_i$ for each $\pi_i \in \Pi$ and $a \in (\pi_i)_{\mathbf{L}}$;
(5) a choice rule $c_1 \mid \ldots \mid c_k \mid e \leftarrow h_i$ with $\{c_1, \ldots, c_k\} = (\pi_i)_{\mathbf{G}} \setminus (\pi_i)_{\mathbf{L}}$ for each $\pi_i \in \Pi$;
(6) and an integrity constraint $\leftarrow h_i, b$ for each $\pi_i \in \Pi$ and $b \in (\pi_i)_{\mathbf{D}} \setminus (\pi_i)_{\mathbf{G}}$.

In the above, $h_1, \ldots, h_l$ and $s_1, \ldots, s_n$ are new atoms corresponding to samplers $\pi_1, \ldots, \pi_l$ in the head Π and the sets $\Pi_1, \ldots, \Pi_n$ in the body, respectively. The atom e in (5) is new.

The rules of Item 1 evaluate sampler sets $\Pi_1, \ldots, \Pi_n$ in the body. The rule of Item 2 is a skeleton of the original sampling rule. The application of head samplers is enforced by the integrity constraints of Item 3 (cf. Definition 3.3). The rules in Items 4–6 enforce the satisfaction of a single head sampler $\pi_i \in \Pi$ once applied. The translation of an entire SP P is $\mathrm{Tr}_{\mathrm{CLP}}(P) = (\bigcup_{r \in P} \mathrm{Tr}_{\mathrm{CLP}}(r)) \cup \{e \leftarrow\}$. To formulate the correctness of $\mathrm{Tr}_{\mathrm{CLP}}$, we need to map any interpretation $M \subseteq \mathrm{At}(P)$ to an interpretation $\mathrm{Ext}_P(M) \subseteq \mathrm{At}(\mathrm{Tr}_{\mathrm{CLP}}(P))$ which includes (i) M as such, (ii) the atom s associated with $\Pi \in \mathrm{BodySS}(P)$ iff $M \models \Pi$, (iii) the atom h associated with $\pi \in \mathrm{HeadS}(P)$ iff $M \models \pi$ and $M \models \Pi_1, \ldots, M \models \Pi_n$, and (iv) e.

Theorem 4.3 (Faithfulness of $\mathrm{Tr}_{\mathrm{CLP}}$)**.** *Let P be any SP and $\mathrm{Tr}_{\mathrm{CLP}}(P)$ its translation into a CLP. (i) If $M \in \mathrm{SM}(P)$, then $N = \mathrm{Ext}_P(M) \in \mathrm{SM}(\mathrm{Tr}_{\mathrm{CLP}}(P))$. (ii) If $N \in \mathrm{SM}(\mathrm{Tr}_{\mathrm{CLP}}(P))$, then its projection $M = N \cap \mathrm{At}(P)$ belongs to $\mathrm{SM}(P)$ and $N = \mathrm{Ext}_P(M)$.*

Due to new atoms, any SP P and $\mathrm{Tr}_{\mathrm{CLP}}(P)$ are *visibly equivalent* [Jan06] but not *strongly equivalent* [Lif01]. To conclude, SPs may provide more compact representations than CLPs.

5. Abstract Constraint Programs (ACPs)

The objective of this section is to show how SPs can be exploited to define the semantics of *abstract constraint programs* in the general case [Bla08]. Our strategy is to extend the stable model semantics by decomposing *abstract constraint atoms* into sets of samplers.

Definition 5.1. An abstract constraint atom π, or an *ac-atom* for short, has the form $\langle D, \{S_1, \ldots, S_k\}\rangle$ where the domain $\pi_{\mathbf{D}} = D$ is a finite set propositional atoms and each set $S_j \subseteq D$ where $1 \leq j \leq k$ is a satisfier in the set $\pi_{\mathbf{S}} = \{S_1, \ldots, S_k\}$ of satisfiers.

The idea is that an interpretation M satisfies an abstract constraint atom π iff the projection $M \cap \pi_{\mathbf{D}} \in \pi_{\mathbf{S}}$. An abstract constraint program (ACP) consists of rules of the form $\pi \leftarrow \pi_1, \ldots, \pi_n$ where π and each π_i is an abstract constraint atom. Certain subclasses have been identified: An ac-atom is *monotone* [Mar08] iff $S_1 \in \pi_{\mathbf{S}}$ and $S_1 \subseteq S_2 \subseteq \pi_{\mathbf{D}}$ imply $S_2 \in \pi_{\mathbf{S}}$. Furthermore, an ac-atom is *convex* [Liu06] iff $S_1 \in \pi_{\mathbf{S}}$, $S_1 \subseteq S_2 \subseteq S_3$, and $S_3 \in \pi_{\mathbf{S}}$ imply $S_2 \in \pi_{\mathbf{S}}$. The rules of *monotone* ACPs and *convex* ACPs solely consist of monotone and convex ac-atoms, respectively. It is clear that monotone ACPs specialize convex ones.

We are now ready to address the semantics of ACPs from the perspective of SPs. Consider two samplers π and π' such that $\pi_{\mathbf{D}} = (\pi')_{\mathbf{D}}$. We say that π *extends* π' iff $(\pi')_{\mathbf{S}} \subseteq \pi_{\mathbf{S}}$, i.e., $\pi_{\mathbf{L}} \subseteq (\pi')_{\mathbf{L}}$ and $(\pi')_{\mathbf{G}} \subseteq \pi_{\mathbf{G}}$. Intuitively speaking, the range of π is greater than or equal to that of π', denoted $\pi' \leq \pi$. Samplers which are $\leq$-maximal provide a basis for the decomposition of ac-atoms and they also guarantee the uniqueness of decompositions.

Definition 5.2 (Decomposition)**.** An ac-atom $\pi = \langle D, \{S_1, \ldots, S_k\}\rangle$ is decomposed into a set of samplers $\mathrm{DS}(\pi) = \{\pi_1, \ldots, \pi_m\}$ such that $(\pi_j)_{\mathbf{D}} = D$, $(\pi_j)_{\mathbf{L}} \in \pi_{\mathbf{S}}$, and $(\pi_j)_{\mathbf{G}} \in \pi_{\mathbf{S}}$ for each $1 \leq j \leq m$, $\bigcup_{i=1}^{m} (\pi_j)_{\mathbf{S}} = \pi_{\mathbf{S}}$, and each $\pi_j \in \mathrm{DS}(\pi)$ is $\leq$-maximal within $\mathrm{DS}(\pi)$.

We observe that $1 \leq m \leq k$ holds for the cardinality m of $\mathrm{DS}(\pi)$. If $m = 1$, then $k = 2^{|G \setminus L|}$, which shows that $\mathrm{DS}(\pi)$ can provide exponentially more succinct representation of π. If $m = k$, then each S_i, $1 \leq i \leq k$, corresponds to an exact sampler $\langle D, S_i\rangle$ of its own. The decomposition of ac-atoms preserves satisfaction under classical semantics, i.e., $I \models \pi$ iff $I \models \mathrm{DS}(\pi)$ holds for any ac-atom π and any interpretation I of π. If an ac-atom π is monotone, then $\mathrm{DS}(\pi)$ includes a sampler $\pi' = \langle D, S_i, D\rangle$ for the domain $D = \pi_{\mathbf{D}}$ and each $\subseteq$-minimal satisfier $S_i \in \pi_{\mathbf{S}}$. Thus we obtain $\mathrm{DS}(\langle\{a\}, \{\emptyset, \{a\}\}\rangle) = \{\langle\{a\}, \emptyset, \{a\}\rangle\}$. On the other hand, if π is convex, then $\mathrm{DS}(\pi)$ contains a sampler $\pi' = \langle D, S_i, S_j\rangle$ for each pair of a $\subseteq$-minimal satisfier $S_i \in \pi_{\mathbf{S}}$ and a $\subseteq$-maximal satisfier $S_j \in \pi_{\mathbf{S}}$ such that $S_i \subseteq S_j$. Note that for monotone ac-atoms π, the domain $D = \pi_{\mathbf{D}}$ is the unique $\subseteq$-maximal element in $\pi_{\mathbf{S}}$.

As regards a rule $\pi_0 \leftarrow \pi_1, \ldots, \pi_n$ involving ac-atoms, it can be modularly decomposed into a sampling rule $\mathrm{DS}(\pi_0) \leftarrow \mathrm{DS}(\pi_1), \ldots, \mathrm{DS}(\pi_n)$. The respective decomposition of an entire ACP P is denoted by $\mathrm{DS}(P)$. Stable models generalize for ACPs via Definition 2.2.

Definition 5.3 (Stable Models of ACPs)**.** Given an ACP P, an interpretation $M \subseteq \mathrm{At}(P)$ of P is a stable model of P iff $M \models P$ and $M = \mathrm{LM}(\mathrm{DS}(P)^M)$.

The reduct $\mathrm{DS}(P)^M$ is a positive SP for which the least model is well-defined according to Proposition 3.4. In addition, the condition $M \models P$ is equivalent to $M \models \mathrm{DS}(P)$ as classical models are preserved by decomposition. Hence we have $\mathrm{SM}(P) = \mathrm{SM}(\mathrm{DS}(P))$ for any ACP P in general. It is also possible to combine Definitions 2.1 and 3.3 in order to generalize the Gelfond-Lifschitz reduct for ACPs. For an entire ACP P, we can define P^M as $\mathrm{DS}(P)^M$ so that Definition 2.2 becomes directly applicable to ACPs. For an individual rule $\pi_0 \leftarrow \pi_1, \ldots, \pi_n \in P$ such that $M \models \pi_1, \ldots, M \models \pi_n$ and $M \models \pi_0$ under the assumption

that $M \models P$, the reduct $\mathrm{DS}(P)^M$ contains a reduced rule $\langle S, S\rangle \leftarrow \mathrm{DS}(\pi_1)^M, \ldots, \mathrm{DS}(\pi_n)^M$ with $S = M \cap \pi_{\mathbf{G}}$ for every $\leq$-maximal sampler $\pi \in \mathrm{DS}(\pi_0)$ such that $M \models \pi$. When the reduction takes place, an ac-atom π_i is mapped into $\mathrm{DS}(\pi_i)^M$ which is not generally representable as an ac-atom due to fixed domains. A convex ACP is illustrated below.

Example 5.4. Consider an ACP P with the following rules:

$$\langle\{a\},\{\emptyset,\{a\}\}\rangle \leftarrow;\ \langle\{b\},\{\emptyset,\{b\}\}\rangle \leftarrow;\ \langle\{c\},\{\emptyset,\{c\}\}\rangle \leftarrow;$$
$$\langle\emptyset,\emptyset\rangle \leftarrow \langle\{a,b,c\},\{\{a\},\{b\},\{c\},\{a,b\},\{a,c\},\{b,c\}\}\rangle.$$

The first monotone rule expresses the free choice of a and it decomposes into $\langle\{a\},\emptyset,\{a\}\rangle \leftarrow$. The rules for b and c are analogous. The last rule captures a *cardinality constraint* [Sim99] $\leftarrow 1\{a,b,c\}2$ with a convex ac-atom. If decomposed, 6 samplers $\langle\{a,b,c\},L,G\rangle$ where $L \subseteq G$, $L \in \{\{a\},\{b\},\{c\}\}$ and $G \in \{\{a,b\},\{a,c\},\{b,c\}\}$ result. The models of P are $M_1 = \emptyset$ and $M_2 = \{a,b,c\}$. For M_1, we obtain only $\langle\emptyset,\emptyset\rangle \leftarrow$ to P^{M_1} so that $M_1 \in \mathrm{SM}(P)$. The reduct P^{M_2} contains rules $\langle\{a\},\{a\}\rangle \leftarrow$, $\langle\{b\},\{b\}\rangle \leftarrow$, and $\langle\{c\},\{c\}\rangle \leftarrow$. Thus $M_2 \in \mathrm{SM}(P)$.

6. Characterization Based on Computations

The stable models of ACPs have been characterized in terms of *abstract computations*, e.g., in the monotone case [Mar08, Liu06]. In what follows, we review the definition of computations for arbitrary ACPs [Liu10] but using ordinals as indices. Given an interpretation $I \subseteq \mathrm{At}(P)$ of an ACP P, the set $P(I)$ of *supporting rules* of P is $\{\pi \leftarrow \pi_1, \ldots, \pi_n \in P \mid I \models \pi_1, \ldots, I \models \pi_n\}$. Moreover, the set $\mathrm{HAt}(P)$ of *head atoms* of P is $\bigcup\{\pi_{\mathbf{D}} \mid \pi \leftarrow \pi_1, \ldots, \pi_n \in P\}$. Computations associated with P are *sequences of interpretations* $\langle I_\alpha\rangle$ indexed by ordinals α. Their properties are formalized using a *nondeterministic* immediate consequence operator $\mathbf{T}_P^{\mathrm{nd}}$ that assigns to any interpretation $I \subseteq \mathrm{At}(P)$ a *set* of interpretations $J \subseteq \mathrm{HAt}(P(I))$ such that $J \models \mathrm{Heads}(P(I))$ where $\mathrm{Heads}(P(I))$ is the set of *heads* of the rules in $P(I)$. *Persistent computations* $\langle I_\alpha\rangle$ meet the following criteria [Liu10]:

- (**R**) *Revision:* For every ordinal α, the interpretation $I_{\alpha+1}$ is grounded in I_α and P, i.e., $I_{\alpha+1} \in \mathbf{T}_Q^{\mathrm{nd}}(I_\alpha)$ for some program $Q \subseteq P(I_\alpha)$.
- (**P**) *Persistence of beliefs:* The sequence $\langle I_\alpha\rangle$ starts from $I_0 = \emptyset$ and it is monotonically increasing, i.e., $I_\alpha \subseteq I_{\alpha+1}$ for all ordinals α, and $I_\beta = \bigcup_{\alpha<\beta} I_\alpha$ for *limit* ordinals β.
- (**C**) *Convergence:* The limit I_∞ that defines the result of the computation $\langle I_\alpha\rangle$ is a *supported model* of P, i.e., it satisfies the fixed-point condition $I_\infty \in \mathbf{T}_P^{\mathrm{nd}}(I_\infty)$.
- (**Pr**) *Persistence of reasons:* There is a sequence $\langle P_\alpha\rangle$ of programs such that for all ordinals α, the program $P_\alpha \subseteq P(I_\alpha)$, $P_\alpha \subseteq P_{\alpha+1}$, and $I_{\alpha+1} \in \mathbf{T}_{P_\alpha}^{\mathrm{nd}}(I_\alpha)$.

Item (**P**) and Knaster-Tarski lemma guarantee that the limit $I_\infty = \bigcup_\alpha I_\alpha$ exists. It is defined as a stable model of P in [Liu10]. Definition 5.3 leads to another class of computations.

Definition 6.1 (Canonical Computations for ACPs)**.** Given an ACP P and an interpretation $M \subseteq \mathrm{At}(P)$, the canonical M-computation for P is a sequence $\langle I_\alpha\rangle$ such that (i) $I_0 = \emptyset$, (ii) $I_{\alpha+1} = \mathbf{T}_{\mathrm{DS}(P)^M}(I_\alpha)$ for each ordinal α, and (iii) $I_\beta = \bigcup_{\alpha<\beta} I_\alpha$ for limit ordinals β.

The operator $\mathbf{T}_{\mathrm{DS}(P)^M}$ is monotone and compact since the rules of $\mathrm{DS}(P)^M$ have the form $\langle S, S\rangle \leftarrow \mathrm{DS}(\pi_1)^M, \ldots, \mathrm{DS}(\pi_n)^M$ and the samplers involved have finite domains. Thus, given a canonical M-computation $\langle I_\alpha\rangle$ for an ACP P and $M \subseteq \mathrm{At}(P)$, we know that (i) $\langle I_\alpha\rangle$ is monotonically increasing, (ii) the limit $I_\omega = \mathrm{lfp}(\mathbf{T}_{\mathrm{DS}(P)^M})$, and (iii) $I_\omega = \mathbf{T}_{\mathrm{DS}(P)^M}(I_\omega)$.

Corollary 6.2 (Characterization). *For an ACP P, an interpretation $M \subseteq \mathrm{At}(P)$ is a stable model of P iff $M \models P$ and $M = I_\omega$ for the result I_ω of the canonical M-computation $\langle I_\alpha \rangle$.*

Theorem 6.3 (Properties of Canonical Computations). *Let P be an ACP and $M \subseteq \mathrm{At}(P)$ a model of P such that $I_\omega = M$ for the limit I_ω of the canonical M-computation $\langle I_\alpha \rangle$. Then $\langle I_\alpha \rangle$ satisfies (**R**), (**P**), (**C**), and (**Pr**) when P_α and Q in (**Pr**) and (**R**), respectively, are substituted by $P_\alpha(M) = \{\pi \leftarrow \pi_1, \ldots, \pi_n \in P(M) \mid I_\alpha \models \mathrm{DS}(\pi_1)^M, \ldots, I_\alpha \models \mathrm{DS}(\pi_n)^M\}$.*

The characterization above is limited to the "successful cases", i.e., when the result turns out to be a stable model. The properties of canonical M-computations are not semantically important when $M \not\models P$ or $I_\omega \neq M$. In both cases, the interpretation M is disqualified as a stable model. It is nevertheless clear that (**P**) holds even for failing computations. Finally, we note that the semantics based on Definition 5.3 can be stricter than the one based on abstract computations. As shown in [She09a], there is a persistent computation and a stable model $M = \{p(-1), p(1), p(2)\}$ for an ACP with $p(1) \leftarrow$; $p(-1) \leftarrow p(2)$; and $p(2) \leftarrow \pi$ where the ac-atom π corresponds to a sum aggregate $\mathrm{Sum}(\{X \mid p(X)\}) \geq 1$ based on the domain $\pi_{\mathbf{D}} = D = \{p(-1), p(1), p(2)\}$. In our approach, the reduct P^M consists of $p(1) \leftarrow$, $p(-1) \leftarrow p(2)$, and $p(2) \leftarrow \langle D, \{p(2)\}, D\rangle$ which indicate the instability of M.

7. Comparison with Previous Approaches

This research was initially motivated by the notions of computations proposed for ACPs. In view of the results presented in [Liu10], we have lifted the notion of M-computations, originally proposed for normal programs, to the case of ACPs. One of our key design decisions was to push all nondeterminism involved in abstract computations to the notion of a program reduct—much in the spirit of choice logic programs [Soi99] covered by Definition 2.1. Corollary 6.2 indicates that each stable model M of an ACP P is generated by a unique computation satisfying the criteria of [Liu10] by Theorem 6.3. As noted above, those criteria lead to a weaker notion of stability if directly generalized for ACPs. Stability notions based on additional criteria [Liu10] depart from the traditional fixed-point definition [Gel88].

The alternative interpretation of ac-atoms as sampler sets led us to an approach which is closely related to the one presented in [She07]. In this work, the counterpart of a sampler $\pi = \langle D, L, G\rangle$ is an *abstract L-prefixed power set* of the form $L \uplus (G \setminus L)$, i.e., the set of sets $\{L \cup K \mid K \subseteq G \setminus L\}$ which coincides with $\pi_{\mathbf{S}}$. These structures are merely used as a compact representation of ac-atoms rather than new primitives for logic programs. Moreover, the generalization of the Gelfond-Lifschitz reduct for an ACP P takes place at a lower level of abstraction: the Shen-You reduct P_M [She07] is formulated as a positive normal logic program and new atoms become a necessity. The way in which ac-atoms are decomposed as sets of samplers (cf. Definition 5.2) pave the way for a tight interconnection.

Theorem 7.1. *For an ACP P and an interpretation $M \subseteq \mathrm{At}(P)$, $M \in \mathrm{SM}(P)$ iff there is a unique minimal model N of the Shen-You reduct P_M such that $M = N \cap \mathrm{At}(P)$.*

This result covers also rules of the form $\pi \leftarrow \pi_1, \ldots, \pi_n$ which have arbitrary ac-atoms in their heads. Further interconnections can be reported from [She09a] for ACPs confining to a limited syntax, i.e., rules of the form $a \leftarrow \pi_1, \ldots, \pi_n$. Given this restriction, stable models of [Den01, Son07b] coincide with models obtained as minimal models of P_M. By Theorem 7.1 the same observation can be made for stable models conforming to Definition 5.3. The iterative construction of stable models in [Son07b] is analogous to canonical M-computations

introduced by us. A difference is that using SPs, a fixed reduct $P^M = \mathrm{DS}(P)^M$ of an ACP P can be formalized and there is no need to parameterize the construction of M otherwise. In this respect, the approaches in [Son07b, Son07a] resort to *conditional satisfaction*. There are further consequences of the relationships pointed above. First of all, the semantics of monotone ACPs is captured in the standard way [Mar08] in our approach. The case of convex ACPs [Liu06] is also covered as illustrated by Example 5.4. We also observe that cardinality and weight rules of the SMODELS system [Sim99] essentially lead to convex constraints. Finally, the notions based on minimal models [Del03, Fab04, Fer05] are prone to self-supporting stable models as shown in [She09a]. We avoid such models by Theorem 7.1.

As the last comparison, we address the program transformation $trm(\cdot)$ from [Pel03]. The idea is to map sets of classical literals $F^D_{\langle L,G\rangle} = \{b \mid b \in L\} \cup \{\neg c \mid c \in D \setminus G\}$ which satisfy an ac-atom $\pi = \langle D, \{S_1, \ldots, S_k\}\rangle$ and are $\subseteq$-minimal in this respect. Given a rule $a \leftarrow \pi$, each set $F^D_{\langle L,G\rangle}$ gives rise to one normal rule $a \leftarrow L, {\sim}(D \setminus G)$ in the translation. This is analogous to translating $\leq$-maximal samplers $\pi' = \langle D, L, G\rangle \in \mathrm{DS}(\pi)$ using $\mathrm{Tr}_{\mathrm{CLP}}(\cdot)$ from Definition 4.2. However, since new atoms are not introduced, the translation $trm(\cdot)$ may create an exponential number of normal rules already for rules of the form $a \leftarrow \pi_1, \ldots, \pi_n$ with several ac-atoms in the rule body. A further aspect is that rules $\pi \leftarrow \pi_1, \ldots, \pi_n$ involving proper ac-atoms π in their heads are not covered at all.

8. Conclusions

In this paper, we propose samplers as new building blocks of logic programs. The respective class of sampler programs (SPs) is designed using a conceptually simpler class of choice logic programs (CLPs) as a starting point. Based on the intuitions provided by CLPs, the stable model semantics (Definition 2.2) extends for SPs in a natural way. As witnessed by Definition 3.3, the notion of a program reduct [Gel88, Soi99] carries over for SPs so that, in particular, the resulting program is a (positive) SP. The availability of *polynomial*, *faithful*, and *modular* (PFM) translation functions $\mathrm{Tr}_{\mathrm{SP}}$ and $\mathrm{Tr}_{\mathrm{CLP}}$ suggests that CLPs and SPs have the same expressive power in the sense of [Jan06] but SPs may provide more concise representation due to samplers and the disjunctive interpretation of sampler sets.

The second main theme of this paper is the application of SPs in order to define the semantics of abstract constraint programs (ACPs). Notably, this class of programs lacks a natural counterpart of Gelfond-Lifschitz reduction [Gel88] which would yield ACPs as its outcome. In pursuit of a simple semantical definition, we propose an approach in which ACPs are interpreted as SPs using the decomposition of ac-atoms as sampler sets as basis. The decomposition method $\mathrm{DS}(\cdot)$ is highly modular since each ac-atom π can be locally decomposed into $\mathrm{DS}(\pi)$ independently of other ac-atoms in the program. The combination of Definitions 2.2, 3.3, and 5.2 enables the definition of stable models for ACPs as given in Definition 5.3: $M \in \mathrm{SM}(P)$ iff $M \models P$ and $M = \mathrm{LM}(\mathrm{DS}(P)^M)$. The new logic programming primitives proposed in this paper, samplers, play a key role in this streamlined definition. The definition is stated without a reference to other logics such as propositional or default logics in contrast with [She07, She09a]. Nevertheless, the semantics of ACPs originally proposed in [She07] is supported by our results (Theorem 7.1). In view of computations, we established in Corollary 6.2 that each $M \in \mathrm{SM}(P)$ has a unique *deterministic* computation, i.e., the canonical M-computation, associated with it. By these observations, we conclude that SPs form an interesting class of logic programs between CLPs and ACPs.

Acknowledgement

The author would like to thank Martin Gebser for his comments on a draft of this paper.

References

[Bla08] H. Blair, V. Marek, and J. Remmel. Set based logic programming. *Annals of Mathematics and Artificial Intelligence*, 52(1):81–105, 2008.

[Del03] T. Dell'Armi, W. Faber, G. Ielpa, N. Leone, and G. Pfeifer. Aggregate functions in disjunctive logic programming: Semantics, complexity, and implementation in dlv. In *Proc. IJCAI-03*, pp. 847–852. Morgan Kaufmann, 2003.

[Den01] M. Denecker, N. Pelov, and M. Bruynooghe. Ultimate well-founded and stable semantics for logic programs with aggregates. In *Proc. ICLP'01*, *LNCS*, vol. 2237, pp. 212–226. Springer, 2001.

[Fab04] W. Faber, N. Leone, and G. Pfeifer. Recursive aggregates in disjunctive logic programs: Semantics and complexity. In *Proc. JELIA'04*, *LNCS*, vol. 3229, pp. 200–212. Springer, 2004.

[Fer05] P. Ferraris. Answer sets for propositional theories. In *Proc. LPNMR'05*, *LNCS*, vol. 3662, pp. 119–131. Springer, 2005.

[Gel88] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proc. ICLP'88*, pp. 1070–1080. 1988.

[Gel90] M. Gelfond and V. Lifschitz. Logic programs with classical negation. In *Proc. ICLP'90*, pp. 579–597. The MIT Press, 1990.

[Gel91] M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9(3/4):365–385, 1991.

[Jan06] T. Janhunen. Some (in)translatability results for normal logic programs and propositional theories. *Journal of Applied Non-Classical Logics*, 16(1–2):35–86, 2006.

[Lif01] V. Lifschitz, D. Pearce, and A. Valverde. Strongly equivalent logic programs. *ACM Transactions on Computational Logic*, 2(4):526–541, 2001.

[Lif08] Vladimir Lifschitz. Twelve definitions of a stable model. In *Proc. ICLP'08*, *LNCS*, vol. 5366, pp. 37–51. Springer, 2008.

[Liu06] L. Liu and M. Truszczynski. Properties and applications of programs with monotone and convex constraints. *Journal of Artificial Intelliegence Research*, 27:299–334, 2006.

[Liu10] L. Liu, E. Pontelli, T. C. Son, and M. Truszczynski. Logic programs with abstract constraint atoms: The role of computations. *Artificial Intelligence*, 174:295–315, 2010.

[Mar08] V. Marek, I. Niemelä, and M. Truszczynski. Logic programs with monotone abstract constraint atoms. *Theory and Practice of Logic Programming*, 8(2):167–199, 2008.

[Pel03] N. Pelov, M. Denecker, and M. Bruynooghe. Translation of aggregate programs to normal logic programs. In *Proc. ASP'03*, *CEUR Workshop Proceedings*, vol. 78. 2003.

[She07] Y.-D. Shen and J.-H. You. A generalized Gelfond-Lifschitz transformation for logic programs with abstract constraints. In *Proc. AAAI'07*, pp. 483–488. AAAI Press, 2007.

[She09a] Y.-D. Shen and J.-H. You. A default approach to semantics of logic programs with constraint atoms. In *Proc. LPNMR'09*, *LNCS*, vol. 5753, pp. 277–289. Springer, 2009.

[She09b] Y.-D. Shen, J.-H. You, and L.-Y. Yuan. Characterizations of stable model semantics for logic programs with arbitrary constraint atoms. *Theory and Practice of Logic Programming*, 9(4):529–564, 2009.

[Sim99] P. Simons. Extending the stable model semantics with more expressive rules. In *Proc. LPNMR'99*, *LNCS*, vol. 1730, pp. 305–316. Springer, 1999.

[Sim02] P. Simons, I. Niemelä, and T. Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1–2):181–234, 2002.

[Soi99] T. Soininen and I. Niemelä. Developing a declarative rule language for applications in product configuration. In *Proc. PADL'99*, *LNCS*, vol. 1551, pp. 305–319. Springer, 1999.

[Son07a] T. C. Son and E. Pontelli. A constructive semantic characterization of aggregates in answer set programming. *Theory and Practice of Logic Programming*, 7(3):355–375, 2007.

[Son07b] T. C. Son, E. Pontelli, and P. H. Tu. Answer sets for logic programs with arbitrary abstract constraint atoms. *Journal of Artificial Intelligence Research*, 29:353–389, 2007.

Technical Communications of the International Conference on Logic Programming, 2010 (Edinburgh), pp. 104–113
http://www.floc-conference.org/ICLP-home.html

A Framework for Verification and Debugging of Resource Usage Properties
RESOURCE USAGE VERIFICATION

PEDRO LOPEZ-GARCIA [1,2] AND LUTHFI DARMAWAN [3] AND FRANCISCO BUENO [3]

E-mail address: pedro.lopez@imdea.org,luthfi@clip.dia.fi.upm.es,bueno@fi.upm.es

[1] IMDEA Software, Madrid, Spain

[2] Spanish Research Council (CSIC), Spain

[3] Technical University of Madrid (UPM), Spain

Abstract. We present a framework for (static) verification of general resource usage program properties. The framework extends the criteria of correctness as the conformance of a program to a specification expressing non-functional global properties, such as upper and lower bounds on execution time, memory, energy, or user defined resources, given as functions on input data sizes. A given specification can include both lower and upper bound resource usage functions, i.e., it can express intervals where the resource usage is supposed to be included in. We have defined an abstract semantics for resource usage properties and operations to compare the (approximated) intended semantics of a program (i.e., the specification) with approximated semantics inferred by static analysis. These operations include the comparison of arithmetic functions (e.g., polynomial, exponential or logarithmic functions). A novel aspect of our framework is that the static checking of assertions generates answers that include conditions under which a given specification can be proved or disproved. For example, these conditions can express intervals of input data sizes such that a given specification can be proved for some intervals but disproved for others. We have implemented our techniques within the Ciao/CiaoPP system in a natural way, so that the novel resource usage verification blends in with the CiaoPP framework that unifies static verification and static debugging (as well as run-time verification and unit testing).

1. Introduction and Motivation

The conventional understanding of software correctness is absence of errors or bugs, expressed in terms of conformance of all possible executions of the program with a functional specification (like type correctness) or behavioral specification (like termination or possible sequences of actions). However, in an increasing number of computing applications additional observables play an essential role. For example, embedded systems must control

1998 ACM Subject Classification: D.1.6 [**Programming Techniques**]: Logic Programming; D.2.4 [**Software Engineering**]: Software/Program Verification—Assertion Checkers, Formal Methods; D.2.5 [**Software Engineering**]: Testing and Debugging. General Terms: Performance, Verification.

Key words and phrases: Program Verification and Debugging, Cost Analysis, Resource Usage Analysis, Complexity Analysis.

This research has been partially funded by the EU 7th. FP NoE *S-Cube* 215483, FET IST-231620 *HATS*, MICINN TIN-2008-05624 *DOVES* and CM project P2009/TIC/1465 *PROMETIDOS*.

Technical Communications of the 26th International Conference on Logic Programming, Edinburgh, July, 2010
Editors: Manuel Hermenegildo, Torsten Schaub
LIPIcs - Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/LIPIcs.ICLP.2010.104

and react to the environment, which also establishes constraints about the system's behavior such as resource usage and reaction times. Therefore, it is necessary for these systems to extend the criteria for correctness with new aspects which include non-functional global properties such as maximum execution time and usage of memory, energy, or other types of resources.

In this paper we propose techniques that extend the capacity of debugging and verification systems based on static analysis [3, 2, 6], when dealing with a quite general class of properties related to resource usage, including upper and lower bounds on execution time, memory, energy, and user-defined resources (the latter in the sense of [8]). Such bounds are given as functions on input data sizes (see [8] for the different metrics that can be used to measure data sizes, such as list length, term depth, or term size). The proposed extension has been implemented in the CiaoPP framework, that unifies static verification and static debugging (as well as run-time verification and unit testing). For example, For example, it extends the capacity of CiaoPP to certify programs with resource consumption assurances and also to efficiently check such certificates.

We define an abstract semantics for resource usage properties and operations to compare the (approximated) intended semantics of a program (i.e., the specification, given as assertions in the program) with approximated semantics inferred by static analysis. These operations include the comparison of arithmetic functions (e.g., polynomial, exponential or logarithmic functions). In traditional static checking, for each property of (part of) an assertion, the possible outcomes are *true* (property proved to hold), *false* (property proved not to hold), and *unknown* (the analysis cannot prove true or false). However, it is very common that cost functions have intersections, so that for a given interval of input data sizes, one of them is smaller than the other one, but for another interval it is the other way around. Thus, a novel aspect of the *resource verification and debugging* approach that we propose is that the answers of the checking process go beyond these classical outcomes and typically include conditions under which the truth or falsity of the property can be proved. Such conditions can be parameterized by attributes of inputs, such as input data size or value ranges. For example, it may be possible to say that the outcome is true if the input data size is in a given range and false if it is in another one.

Example 1.1. Consider an assertion which declares an upper bound (`ub`) on the resource usage, in terms of resolution steps, of the classical `fibonacci` program such as:

```
:- check comp fib(N,F): (int(N), var(F)) + steps_ub( exp(2, int(N))-1000 ).
```

meaning that the computation of any call to `fib(N,F)` with the first argument bound to an integer and the second one a free variable should take at most $2^x - 1000$ resolution steps, x being the size of the first argument (i.e., the actual value of `N`, since it has to be an integer number). This is true only for $x \geq 10$, and maybe programmers have tried the program only with big numbers, and then generalized their observations in the above assertion. We will see how the CiaoPP system, with our approach, is able to inform the programmer that this idea is wrong. Indeed, as we will see, the output of our assertion checking implementation within the CiaoPP system is:

```
:- false comp fib(N,F): (int(N), var(F)) + steps_ub( exp(2,int(N))-1000 ).
   in interval [0, 10] for int(N).
:- true comp fib(N,F): (int(N), var(F)) + steps_ub( exp(2,int(N))-1000 ).
   in interval [11, +inf] for int(N).
```

meaning that the system has proved that the assertion is false for values of the input argument `N` in the interval $[0, 10]$, and true for `N` in the interval $[11, \infty)$. This is because in the interval $[0, 10]$, the *lower bound* on resolution steps inferred by the analysis is greater than the upper bound expressed in the assertion, and in the interval $[11, \infty)$, the *upper bound* inferred by the analysis is less than the upper bound in the assertion.

In our approach, user specifications (i.e., assertions) can include for example lower and upper bounds, and even asymptotic values of the resource usage of the computation (given as functions on input data sizes). Moreover, a given specification can include both lower and upper bound resource usage functions, i.e., it can express intervals where the resource usage is supposed to be included in.

The most related work we are aware of presents a method for comparison of cost functions inferred by the COSTA system for Java bytecode [1]. The method proves whether a cost function is smaller than another one *for all the values* of a given initial set of input data sizes. The result of this comparison is a boolean value. However, as mentioned before, in our approach the result is in general a set of subsets (intervals) in which the initial set of input data sizes is partitioned, so that the result of the comparison is different for each subset. The method in [1] also differs from ours in that comparison is syntactic, using a method similar to what was already being done in the CiaoPP system: performing a function normalization and then using some syntactic comparison rules. However, in this work we go beyond these syntactic comparison rules. Moreover, we present an application for which cost function comparison is instrumental and which is not covered in the cited work: verification of resource usage properties. This implies extending the criteria of correctness and defining a resource usage (abstract) semantics and conditions under which a program is correct or incorrect with respect to an (approximated) intended semantics.

In the following, we describe, in Section 2, how to extend and use the CiaoPP verification framework, that we take as starting point, for the verification of general resource usage program properties. In Section 3 we explain the technique that we have developed for resource usage function comparison. Section 4 summarizes our conclusions.

2. A Framework for Verification of Resource Usage Properties

The verification and debugging framework of CiaoPP [6] uses abstract interpretation-based analyses, which are provably correct and also practical, in order to statically compute semantic approximations of programs. These semantic approximations are compared with (partial) specifications, in the form of assertions that are written by the programmer, in order to detect inconsistencies or to prove such assertions.

Both program verification and debugging compare the *actual semantics* $[\![P]\!]$ of a program P with an *intended semantics* for the same program, which we will denote by I. In the framework, both semantics are given in the form of (*safe*) approximations. The abstract approximation $[\![P]\!]_\alpha$ of the concrete semantics $[\![P]\!]$ of the program is actually computed and compared directly to the (also approximate) specification, which is safely assumed to be also given as an abstract value I_α. Program verification is then performed by comparing I_α and $[\![P]\!]_\alpha$. We refer the reader to [3, 5, 6] for a detailed description of the foundations, such as conditions for safely prove partial correctness or incorrectness, and implementation issues of the framework. In this paper we concentrate on defining the main elements of the framework required for its application to resource usage properties.

Resource usage semantics. Given a program p, let C_p be the set of all calls to p. The concrete resource usage semantics of a program p, for a particular resource of interest, $[\![P]\!]$, is a set of pairs $(p(\bar{t}), r)$ such that $\bar{t}$ is a tuple of terms, $p(\bar{t}) \in C_p$ is a call to predicate p with actual parameters $\bar{t}$, and r is a number expressing the amount of resource usage of the computation of the call $p(\bar{t})$. Such a semantic object can be computed by a suitable operational semantics, such as SLD-resolution, adorned with the computation of the resource usage. We abstract away such computation, since it will in general be dependent on the particular resource r refers to. The concrete resource usage semantics can be defined as a function $[\![P]\!] : C_p \mapsto R$ where R is the set of real numbers (note that depending on the type of resource we can take other set of numbers, e.g., the set of natural numbers).

The abstract resource usage semantics is a set of 4-tuples:

$$(p(\bar{v}) : c(\bar{v}), \Phi, input_p, size_p)$$

where $p(\bar{v}) : c(\bar{v})$ is an abstraction of a set of calls. $\bar{v}$ is a tuple of variables and $c(\bar{v})$ is an abstraction representing a set of tuples of terms which are instances of $\bar{v}$. $c(\bar{v})$ is an element of some abstract domain expressing instantiation states. Φ is an abstraction of the resource usage of the calls represented by $p(\bar{v}) : c(\bar{v})$. We refer to it as a *resource usage interval function* for p, defined as follows:

- A *resource usage bound function* for p is a monotonic arithmetic function, $\Psi : S \mapsto R_\infty$, for a given subset $S \subseteq R^k$, where R is the set of real numbers, k is the number of input arguments to predicate p and R_∞ is the set of real numbers augmented with the special symbols ∞ and $-\infty$. We use such functions to express lower and upper bounds on the resource usage of predicate p depending on input data sizes.
- A *resource usage interval function* for p is an arithmetic function, $\Phi : S \mapsto RI$, where S is defined as before and RI is the set of intervals of real numbers, such that $\Phi(\bar{n}) = [\Phi^l(\bar{n}), \Phi^u(\bar{n})]$ for all $\bar{n} \in S$, where $\Phi^l(\bar{n})$ and $\Phi^u(\bar{n})$ are *resource usage bound functions* that denote the lower and upper endpoints of the interval $\Phi(\bar{n})$ respectively for the tuple of input data sizes $\bar{n}$. Although $\bar{n}$ is typically a tuple of natural numbers, we do not want to restrict our framework. We require that Φ be well defined so that $\forall \bar{n}\ (\Phi^l(\bar{n}) \leq \Phi^u(\bar{n}))$.

$input_p$ is a function that takes a tuple of terms $\bar{t}$ and returns a tuple with the input arguments to p. This function can be inferred by using existing mode analysis or can be given by the user by means of assertions. $size_p(\bar{t})$ is a function that takes a tuple of terms $\bar{t}$ and returns a tuple with the sizes of those terms under a given metric. The metric used for measuring the size of each argument of p can be automatically inferred (based on type analysis information) or can be given by the user by means of assertions [8].

Example 2.1. Consider for example the naive reverse program in Figure 1, with the classical definition of predicate `append`. The first argument of `nrev` is declared input, and the two first arguments of `append` are consequently inferred to be also input. The size measure for all of them is inferred to be *list-length*. Then, we have that:
$input_{nrev}((x, y)) = (x)$, $input_{app}((x, y, z)) = (x, y)$,
$size_{nrev}((x)) = (length(x))$ and $size_{app}((x, y)) = (length(x), length(y))$.

In order to make the presentation simpler, we will omit the $input_p$ and $size_p$ functions in abstract tuples, with the understanding that they are present in all such tuples.

```
:- module(reverse, [nrev/2], [assertions]).
:- use_module(library('assertions/native_props')).
:- entry nrev(A,B) : ( ground(A), list(A), var(B) ).
nrev([],[]).
nrev([H|L],R) :- nrev(L,R1), append(R1,[H],R).
```

Figure 1: A module for naive reverse.

Intended meaning. The intended approximated meaning I_α of a program is an abstract semantic object with the same kind of tuples: $(p(\bar{v}) : c(\bar{v}), \Phi, input_p, size_p)$, which are given in the form of assertions. The basic form of resource usage assertions is:[1]

`:- comp` *Pred* [`:` *Precond*] `+` *ResUsage*`.`

which expresses that for any call to *Pred*, if *Precond* is satisfied in the calling state, then *ResUsage* should also be satisfied for the computation of *Pred*. *ResUsage* defines in general an interval of numbers for the particular resource usage of the computation of the call to *Pred* (i.e., *ResUsage* is satisfied by the computation of the call to *Pred* if the resource usage of such computation is in the defined interval).

Example 2.2. In the program of Figure 1 one could use the assertion:

```
:- comp nrev(A,B): ( ground(A), list(A), var(B) )
                 +  resource(ub, steps, 1+exp(length(A), 2)).
```

to express that for any call to `nrev(A,B)` with the first argument bound to a ground list and the second one a free variable, an upper bound (`ub`) on the number of resolution `steps` performed by the computation is $1 + n^2$, where $n = length(A)$. In this case, the interval approximating the number of resolution steps is $[0, 1 + n^2]$. Since the number of resolution steps cannot be negative, the minimum of the interval is zero. If we assume that the resource usage can be negative, the interval would be $(-\infty, 1 + n^2]$. If we had a lower bound (`lb`) instead of an upper bound in the assertion, the interval would be $[1 + n^2, \infty)$.

Such an assertion describes a tuple in I_α which is given by $(p(\bar{v}) : c(\bar{v}), \Phi, input_p, size_p)$, where $p(\bar{v}) : c(\bar{v})$ is defined by *Pred* and *Precond*, and Φ is defined by *ResUsage*. The information about $input_p$ and $size_p$ is implicit in *ResUsage*. The concretization of I_α, $\gamma(I_\alpha)$, is the set of all pairs $(p(\bar{t}), r)$ such that $\bar{t}$ is a tuple of terms and $p(\bar{t})$ is an instance of *Pred* that meets precondition *Precond*, and r is a number that meets the condition expressed by *ResUsage* (i.e., r lies in the interval defined by *ResUsage*) for some assertion.

Example 2.3. The assertion in Example 2.2 captures the following concrete semantic tuples:

```
( nrev([a,b,c,d,e,f,g],X), 35 )        ( nrev([],Y), 1 )
```

but it does not capture the following ones:

```
( nrev([A,B,C,D,E,F,G],X), 35 )        ( nrev(W,Y), 1 )
( nrev([a,b,c,d,e,f,g],X), 53 )        ( nrev([],Y), 11 )
```

those in the first line above because they correspond to calls which are outside the scope of the assertion (i.e., they do not meet the precondition *Precond*); those on the second line

[1] Assertions may be prefixed with a *status* indicating that it is to be checked, or that it has been already checked, detected to be false or detected to be true. Omitting this prefix means "to be checked" [9].

(which will never occur on execution) because they violate the assertion (i.e., they meet the precondition *Precond*, but do not meet the condition expressed by *ResUsage*).

Partial correctness: comparing the abstract semantics. During verification / debugging within our framework, we need to compare an abstract semantics inferred by analysis with an intended abstract semantics. We give here some ideas about how to do it, and refer the reader to [7] for a complete formalization of the abstract semantics and comparison operations.

Given a program p and an intended resource usage semantics I, where $I : C_p \mapsto R$, we say that p is partially correct w.r.t. I if for all $p(\bar{t}) \in C_p$ we have that $(p(\bar{t}), r) \in I$, where r is precisely the amount of resource usage of the computation of the call $p(\bar{t})$. We say that p is partially correct with respect to a tuple of the form $(p(\bar{v}) : c_I(\bar{v}), \Phi_I)$ if for all $p(\bar{t}) \in C_p$ such that r is the amount of resource usage of the computation of the call $p(\bar{t})$, it holds that: if $p(\bar{t}) \in \gamma(p(\bar{v}) : c_I(\bar{v}))$ then $r \in \Phi_I(\bar{s})$, where $\bar{s} = size_p(input_p(\bar{t}))$. Finally, we say that p is partially correct with respect to I_α if:

- For all $p(\bar{t}) \in C_p$, there is a tuple $(p(\bar{v}) : c_I(\bar{v}), \Phi_I)$ in I_α such that $p(\bar{t}) \in \gamma(p(\bar{v}) : c_I(\bar{v}))$, and
- p is partially correct with respect to every tuple in I_α.

Let $(p(\bar{v}) : c(\bar{v}), \Phi)$ and $(p(\bar{v}) : c_I(\bar{v}), \Phi_I)$ be tuples expressing an abstract semantics $[\![P]\!]_\alpha$ inferred by analysis and an intended abstract semantics I_α, respectively, such that $c_I(\bar{v}) \sqsubseteq c(\bar{v})$,[2] and for all $\bar{n} \in S$ ($S \subseteq R^k$), $\Phi(\bar{n}) = [\Phi^l(\bar{n}), \Phi^u(\bar{n})]$ and $\Phi_I(\bar{n}) = [\Phi^l_I(\bar{n}), \Phi^u_I(\bar{n})]$. We have that:

(1) If for all $\bar{n} \in S$, $\Phi^l_I(\bar{n}) \leq \Phi^l(\bar{n})$ and $\Phi^u(\bar{n}) \leq \Phi^u_I(\bar{n})$, then p is partially correct with respect to $(p(\bar{v}) : c_I(\bar{v}), \Phi_I)$.
(2) If for all $\bar{n} \in S$ $\Phi^u(\bar{n}) < \Phi^l_I(\bar{n})$ or $\Phi^u_I(\bar{n}) < \Phi^l(\bar{n})$, then p is incorrect with respect to $(p(\bar{v}) : c_I(\bar{v}), \Phi_I)$.

However, for simplicity, in this paper we assume that one of the endpoints of the interval is always the maximum (resp., minimum) of the possible values, i.e., $\forall \bar{n}$ $(\Phi^u_I(\bar{n}) = \infty)$ (resp., $\Phi^l_I(\bar{n}) = -\infty$ or $\Phi^l_I(\bar{n}) = 0$, depending on the resource). Thus, one of the resource usage bound function comparisons in each of the two cases above is always trivial. Therefore, we will be faced with only one such comparison, between two resource usage bound functions, each denoting either a lower bound (l) or an upper bound (u).

For the particular case where resource usage bound functions depend on one argument, the result of the resource usage bound function comparison in our approach is in general a set of intervals of input data sizes for which a function is less, equal, or greater than another. This allows us to give intervals of input data sizes for which a program p is partially correct (or incorrect).

3. Resource Usage Bound Function Comparison

Given two resource usage bound functions (one of them inferred by the static analysis and the other one given in an assertion/specification present in the program), $\Psi_1(n)$ and $\Psi_2(n)$, $n \in R$ the objective of this operation is to determine intervals for n in which $\Psi_1(n) > \Psi_2(n)$, $\Psi_1(n) = \Psi_2(n)$, or $\Psi_1(n) < \Psi_2(n)$.

[2]Note that the condition $c_I(\bar{v}) \sqsubseteq c(\bar{v})$ can be checked using the CiaoPP capabilities for comparing program state properties such as types.

Our approach consists in defining $f(n) = \Psi_1(n) - \Psi_2(n)$ and finding the roots of the equation $f(n) = 0$. Assume that the equation has m roots, $n_1, \ldots, n_m$. These roots are intersection points of $\Psi_1(n)$ and $\Psi_2(n)$. We consider the intervals $S_1 = [0, n_1)$, $S_2 = (n_1, n_2)$, $S_m = \ldots (n_{m-1}, n_m)$, $S_{m+1} = (n_m, \infty)$. For each interval S_i, $1 \leq i \leq m$, we select a value v_i in the interval. If $f(v_i) > 0$ (respectively $f(v_i) < 0$), then $\Psi_1(n) > \Psi_2(n)$ (respectively $\Psi_1(n) < \Psi_2(n)$) for all $n \in S_i$.

Since our resource analysis is able to infer different types of functions (e.g., polynomial, exponential and logarithmic), it is also desirable to be able to compare all of these functions. For polynomial functions there exist powerful algorithms for obtaining roots. For the other functions, we have to approximate them using polynomials. In this case, we should guarantee that the error falls in the safe side when comparing the corresponding resource usage bound functions.

3.1. Comparing Polynomial Functions

There are general methods for finding roots of polynomial equations. Root equation finding of polynomial functions can be done analytically until polynomial order four. For higher order polynomial functions, numerical methods must be used. According to the fundamental theorem of algebra, a polynomial equation of order m has m roots, whether real or complex numbers. Numerical methods exist that allow computing all these roots (although the complex numbers are not needed in our approach).

For this purpose in our implementation we have used the GNU Scientific Library [4], which offers a specific polynomial function library that uses analytical methods for finding roots of polynomials up to order four, and uses numerical methods for higher order polynomials.

3.2. Approximation of Non-Polynomial Functions

There are two non-polynomial resource usage functions that the CiaoPP analyses can infer: exponential and logarithmic. For approximating these functions we use Taylor series.

Exponential function approximation using polynomials. This approximation is carried out using these formulae:

$$e^x \approx \Sigma_{n=0}^{\infty} \frac{x^n}{n!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots \qquad for\ all\ x$$

$$a^x = e^{x\ ln\ a} \approx 1 + x\ ln\ a + \frac{(x\ ln\ a)^2}{2!} + \frac{(x\ ln\ a)^3}{3!} + \ldots$$

In our implementation these series are limited up to order 8. This decision has been taken based on experiments we have carried out that show that higher orders do not bring a significant difference in practice. Also, in our implementation, the computation of the factorials is done separately and the results are kept in a table in order to reuse them.

Logarithmic function approximation using polynomials. Unfortunately this approximation cannot be done in a straightforward way as previously. A Taylor series for this function for whole interval does not exist, the series only holds for interval $-1 < x < 1$. One possibility to work within this restriction is using range reduction [10].

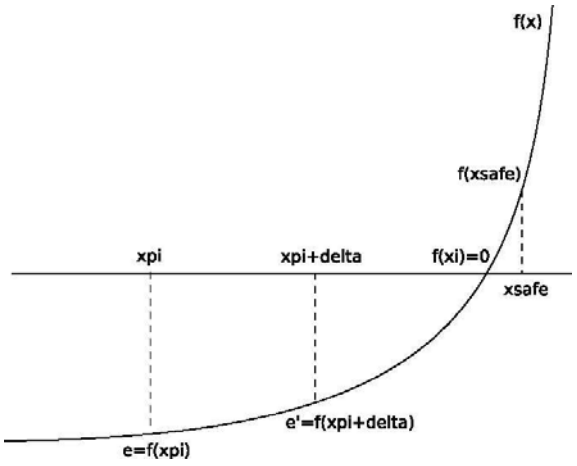


Figure 2: Case 1. $x_i > xp_i$ (since $e' > e$). A safe approximate root found is x_{safe}.

3.3. Safety of the Approximation

Since the roots obtained for function comparison are in some cases approximations of the real roots, we must guarantee that their values are safe, i.e., that they can be used for verification purposes, in particular, for safely checking the conditions in (1) and (2) (page 109.) Assume for example that we are going to safely check whether $\Phi^u(x) \leq \Phi^u_I(x)$ (where Φ^u and Φ^u_I are resource usage bound functions, the former is a result of program analysis and the latter an assertion declared in the program). In this case, we define $f(x) = \Phi^u_I(x) - \Phi^u(x)$, so that we can safely say that if $f(x) > 0$, then $\Phi^u(x) \leq \Phi^u_I(x)$. Assume also that Φ^l_I is not given in the assertion, meaning that the specification does not state any lower bound for the resource usage (i.e., the lower endpoint of any resource usage interval is $-\infty$, which means that $\Phi^l_I(x) \leq \Phi^l(x)$ is always true). We can then safely state that the assertion is true for all x such that $f(x) > 0$. In the same way, if we define $f(x) = \Phi^l(x) - \Phi^u_I(x)$ we can safely say that if $f(x) > 0$ then, $\Phi^u_I(x) < \Phi^l(x)$, proving that the assertion is false for all x such that $f(x) > 0$. We can reason similarly in the comparisons involving a lower bound in the assertion (Φ^l_I). Thus, we focus exclusively on safely determining values for x such that $f(x) > 0$, where $f(x)$ is conveniently defined in each case. Let us see how it can be performed.

In general, we approximate $f(x)$ using a polynomial $P(x)$, so that $f(x) = P(x) + -e$, with e being an approximation error. Let the roots of equation $f(x) = 0$ be $x_0, ..., x_n$. Using a root finding algorithm on equation $P(x) = 0$, we obtain the roots $xp_0, ..., xp_n$, so that we have $P(xp_i) = 0$, and therefore $f(xp_i) \in [-e, +e]$. Then, we have to determine, for each approximated root xp_i, $1 \leq i \leq n$, a value ε such that $f(xp_i+\varepsilon) > 0$ and $x_i \in [xp_i-\varepsilon, xp_i+\varepsilon]$. We do this by first determining the relative position of xp_i and x_i (i.e., whether xp_i is "to the right" or "to the left" of x_i) and then starting an iterative process that increments (or decrements) xp_i by some δ until we have that, after m iterations, $f(xp_i + m\ \delta) > 0$.

Determining the relative position of the exact root. To determine the relative position of the exact root and its approximated value we use the gradient of $f(x)$ around $x = xp_i$. For determining the gradient we use the values of $e = f(xp_i)$ and $e' = f(xp_i + \delta')$,

with $\delta' > 0$ a relatively small number. Whether the approximated root is greater or less than the exact root depends on the following conditions:

(1) if $e < 0$ and $e' > e$ then $x_i > xp_i$
(2) if $e > 0$ and $e' > e$ then $x_i < xp_i$
(3) if $e > 0$ and $e' < e$ then $x_i > xp_i$
(4) if $e < 0$ and $e' < e$ then $x_i < xp_i$

From Figure 2 we can see the rationale behind the first case (the other cases follow an analogous reasoning). If $e' > e$ then $f(x)$ is increasing, but, since $e < 0$, then $f(x) > 0$ can only occur for values of x greater than xp_i. Therefore, $x_i > xp_i$. In such cases we use a positive value of δ for the iterative process. When $x_i < xp_i$ we use a negative value of δ.

Iterative process for computing the safe root. Once we have determined the relative position of the exact root and its approximated value, we first set up the appropriate sign for δ, where $|\delta|$ is a relatively small number: $\delta < 0$ if the iteration should go to the left ($x_i < xp_i$), or $\delta > 0$ if it should go to the right ($x_i > xp_i$). Then we iterate on the addition $xp_i = xp_i + \delta$ until $f(xp_i) > 0$ (if $e < 0$) or $f(xp_i) < 0$ (if $e > 0$). Such an iteration is apparent in the following pseudo-code:

```
1: xsafe ← xp_i
2: if f(xp_i) < 0 then
3:    while f(xsafe) < 0 do xsafe ← xsafe + δ
4:    end while
5: else (f(xp_i) > 0)
6:    while f(xsafe) > 0 do xsafe ← xsafe + δ
7:    end while
8: end if
9: return xsafe
```

Example 3.1. Consider again the assertion in Example 1.1 in Section 1, which declares an upper bound on resource usage given by function $\Phi^u_I(x) = 2^x - 1000$. Let the analysis infer a lower bound $\Phi^l(x) = 1.45 \times 1.62^x - 1$. Their intersection occurs at $x \approx 10.22$. However, the root obtained by our root finding algorithm is $x \approx 10.89$. By doing an iterative approximation from 10.89 to the left, we finally obtain a safe approximate root of $x \approx 10.18$.

Note that usually (as in the above example), resource usage functions are on variables which range on natural numbers. Because of this, the iterative approximation process for safe roots can be substituted by simply taking the closest natural number to the left or right of the approximated root (depending on the gradient) to obtain a safe value. In the previous example, we will simply take 10, without any iteration.

It turns out that the analysis also infers an upper bound given by function $\Phi^u(x) = 1.45 \times 1.62^x - 1$. Thus, the output of our assertion checking for the `fibonacci` program will be that of Example 1.1, showing extra conditions (an interval of integers) on which the assertion can be proved false, on one hand, and another condition (the rest of the range of the positive integer numbers) on which it can be proved true, on the other hand.

4. Conclusions

We have proposed a method for extending how a framework for verification/debugging (implemented in the CiaoPP system) deals with specifications about the resource usage of

programs. We have provided a formalization which blends in with the previous framework for verification of functional or program state properties. A key aspect of the framework is to be able to compare mathematical functions. We have proposed a method which is safe, in the sense that the results of verification/debugging cannot go wrong. In the case where the resource usage functions being compared depend on one variable (which represents some input argument size) our method reveals particular numerical intervals for such variable, if they exist, which might result in different answers to the verification problem: a given specification might be proved for some intervals but disproved for others. Our current method computes such intervals with precision for polynomial and exponential resource usage functions, and in general for functions that can be approximated by polynomials. Moreover, we have proposed an iterative post-process to safely tune up the interval bounds by taking as starting values the previously computed roots of the polynomials.

References

[1] E. Albert, P. Arenas, S. Genaim, I. Herraiz, and G. Puebla. Comparing cost functions in resource analysis. In *1st International Workshop on Foundational and Practical Aspects of Resource Analysis (FOPARA'09)*, Lecture Notes in Computer Science. Springer, 2009. To appear.

[2] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A Static Analyzer for Large Safety-Critical Software. In *Proc. of PLDI'03*. ACM Press, 2003.

[3] F. Bueno, P. Deransart, W. Drabent, G. Ferrand, M. Hermenegildo, J. Maluszynski, and G. Puebla. On the Role of Semantic Approximations in Validation and Diagnosis of Constraint Logic Programs. In *Proc. of the 3rd. Int'l WS on Automated Debugging–AADEBUG*, pages 155–170. U. Linköping Press, May 1997.

[4] M. Galassi, J. Davies, J. Theiler, B. Gough, G. Jungman, P. Alken, M. Booth, and F. Rossi. *GNU Scientific Library Reference Manual*. Network Theory Ltd, 2009. Library and Manual also available at `http://www.gnu.org/software/gsl/`.

[5] M. Hermenegildo, G. Puebla, and F. Bueno. Using Global Analysis, Partial Specifications, and an Extensible Assertion Language for Program Validation and Debugging. In *The Logic Programming Paradigm: a 25–Year Perspective*, pages 161–192. Springer-Verlag, 1999.

[6] M. Hermenegildo, G. Puebla, F. Bueno, and P. López García. Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). *Science of Comp. Progr.*, 58(1–2), 2005.

[7] P. López-García, L. Darmawan, F. Bueno, and M. Hermenegildo. Towards a Framework for Resource Usage Verification and Debugging in the CiaoPP System. Technical Report CLIP1/2010.0, Technical University of Madrid (UPM), School of Computer Science, UPM, February 2010. Available at `http://cliplab.org/papers/resource-verif-10-tr.pdf`.

[8] J. Navas, E. Mera, P. López-García, and M. Hermenegildo. User-Definable Resource Bounds Analysis for Logic Programs. In *ICLP'07*, number 4670 in LNCS, pages 348–363, 2007.

[9] G. Puebla, F. Bueno, and M. Hermenegildo. An Assertion Language for Constraint Logic Programs. In P. Deransart, M. Hermenegildo, and J. Maluszynski, editors, *Analysis and Visualization Tools for Constraint Programming*, number 1870 in LNCS, pages 23–61. Springer-Verlag, September 2000.

[10] Jyri Ylostalo. Function approximation using polynomials. *Signal Processing Magazine*, 23:99–102, September 2006.

Technical Communications of the International Conference on Logic Programming, 2010 (Edinburgh), pp. 114–123
http://www.floc-conference.org/ICLP-home.html

CONTRACTIBILITY AND CONTRACTIBLE APPROXIMATIONS OF SOFT GLOBAL CONSTRAINTS

MICHAEL J. MAHER [1,2]

[1] NICTA, Locked Bag 6016, The University of New South Wales, Sydney, NSW 1466, Australia

[2] School of Computer Science and Engineering, The University of New South Wales, Sydney, NSW 2052, Australia
E-mail address: michael.maher@nicta.com.au

Abstract. We study contractibility and its approximation for two very general classes of soft global constraints. We introduce a general formulation of decomposition-based soft constraints and provide a sufficient condition for contractibility and an approach to approximation. For edit-based soft constraints, we establish that the tightest contractible approximation cannot be expressed in edit-based terms, in general.

1. Introduction

Soft constraints are useful for addressing problems that might be overconstrained. Open global constraints [1, 11, 14] allow variables to be added to the global constraint during execution, which is vital when we want to interleave problem construction and problem solving. Contractible approximations of global constraints are a necessary part of implementing open versions of the constraint. In this paper we investigate contractibility [13] and contractible approximations [14] of soft constraints in the sense of Petit *et al* [18] for the purpose of implementing versions that are dynamic, or open, in the sense of Barták [1]. Contractibility of soft constraints was studied in [15] but approximations have not been investigated.

We investigate two general classes of soft constraints based, respectively, on decompositions and edit-distance. We improve on several results of [15] and establish new results for these classes. While hard constraints seem to be amenable to tight contractible approximation, at least in the cases studied so far [14], we show that soft constraints are much less so.

Section 2 provides some preliminaries on open global constraints and contractibility. In section 3 we introduce a very general class of decomposition-based soft constraints, repeat the definition of edit-based soft constraints from [15] and explore some consequences of

1998 ACM Subject Classification: D.1.6 Logic Programming; D.3.2 Language Classifications, Constraint and logic languages; D.3.3 Language Constructs and Features, Constraints .

Key words and phrases: constraint logic programming, global constraints, open constraints, soft constraints.

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.

Technical Communications of the 26th International Conference on Logic Programming, Edinburgh, July, 2010
Editors: Manuel Hermenegildo, Torsten Schaub
LIPIcs - Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/LIPIcs.ICLP.2010.114

these definitions. Section 4 investigates contractible approximations for soft constraints in these two classes.

2. Background

The reader is assumed to have a basic knowledge of constraint logic programming, CSPs, global constraints, and filtering, as might be found in [9, 19, 2].

For the purposes of this paper, a global constraint is a relation over a single sequence of variables. Other arguments of a constraint are considered parameters and are assumed to be fixed before execution. A sequence of variables will be denoted by $\vec{X}$ or $[X_1, \ldots, X_n]$. We make no *a priori* restriction on the variables that may participate in the sequence except that, in common with most work on global constraints, we assume that no variable appears more than once in a single constraint.

We assume that each use of a global constraint has a static type T that assigns, for every position i of its argument, a set of values. Thus every variable X_i in $\vec{X}$ has a static type $T(X_i)$ of values that it may take. This is distinct from the *domain* $D(X)$ of a variable X, which changes during execution. We always have $D(X) \subseteq T(X)$.

We formalize the semantics of a global constraint C as a formal language L_C. A word $d_1 d_2 \ldots d_n$ appears in L_C iff the constraint $C([X_1, X_2, \ldots, X_n])$ has a solution $X_1 = d_1, \ldots X_n = d_n$. Thus, for example, the semantics of AllDifferent is $\{a_1 \ldots a_n \mid \forall i, j\ i \neq j \rightarrow a_i \neq a_j, n \in \mathbb{N}\}$ and the semantics of Regular$(\mathcal{A}, \vec{X})$ is $L(\mathcal{A})$, the language accepted by $\mathcal{A}$. When it is convenient, we will describe languages with Kleene regular expressions.

We will need the following definitions later. Let $P(L) = \{w \mid \exists u\ wu \in L\}$ denote the set of prefixes of a language L, called the prefix-closure of L. We say L is *prefix-closed* if $P(L) = L$.

Constraint logic programming supports the generation of new variables and new constraints during execution. Use of open constraints pre-supposes additional capabilities to add variables to an open constraint and to close an open constraint. In this paper we will abstract away the details of the language mechanisms that provide these capabilities so that we can focus on the filtering/propagation for open constraints.

There are three models of open constraint that have been proposed [1, 11, 14]. In this paper we follow the model of Barták [1] as refined in [13]: the collection of variables forms a sequence, to which variables may be added at the right-hand end only.

We take *filtering* to refer to any function f that reduces domains, that is, $\forall X\ f(D)(X) \subseteq D(X)$. A filtering algorithm f for a constraint C is *sound* if every solution of C in D also appears in $f(D)$. We say a domain D *defines an assignment* if $\forall X\ |D(X)| = 1$; in that case the assignment maps each X to the element of $D(X)$. We say filtering performs *complete checking* if, whenever D defines an assignment, the result of filtering with a constraint C is D iff the assignment satisfies C. Soundness and complete checking can be considered minimal requirements for filtering methods [20].

Consistency conditions like domain consistency are inappropriate for open constraints because some of the variables in an open constraint will be unspecified during part of the execution. The counterpart of domain consistency is open D-consistency, defined in [14].

During execution, we may extend an open constraint $C(\vec{X})$ with an extra variable Y to $C(\vec{X}Y)$. We would like to do filtering on the smaller constraint without knowing whether it will be extended by Y, or further, and without creating a choicepoint. When we can do

this, we have a kind of monotonicity property of C, called contractibility [13], which can be characterized as follows.

Definition 2.1. We say a constraint $C(\vec{X})$ is *contractible* iff L_C is prefix-closed.

Any filtering method satisfying the minimal requirements discussed above requires contractibility to guarantee that closed filtering is sound for an open constraint. This refines a result of [13].

Theorem 2.2. *Let C be a constraint, and consider a sound filtering method that performs complete checking. It is always sound to interleave filtering and the addition of new variables iff C is contractible.*

Consequently, for contractible constraints, filtering does not need to be undone if the list is lengthened. That is, algorithms for filtering a closed contractible constraint are valid also for the corresponding open constraint.

Conversely, any constraint C that is not contractible might need to undo the effects of filtering if the list is lengthened. Barták [1] proposed an implementation approach to avoid this effect. Essentially, a propagator for a contractible approximation of C is used until C is closed, when a propagator for C is used. The importance of contractible approximations lies in this ability to provide filtering for open uncontractible constraints. Tight approximations can yield best-possible filtering [14].

Theorem 2.3. *Let C_{app} be the tightest contractible approximation to C, and suppose we have closed propagators for C_{app} and C that maintain domain consistency wrt $\vec{X}$. Then Barták's proposal maintains open D-consistency for C.*

3. Contractibility of Soft Constraints

We consider "soft" global constraints in the style of [18]. In such constraints there is a violation measure, which measures the degree to which an assignment to the variables violates the associated "hard" constraint, and solutions are assignments that satisfy an upper bound on the violation measure. Thus such soft constraints have the form $m(\vec{X}) \leq Z$, where m is the violation measure[1]. We refer to the hard constraint as $C(\vec{X})$ and the corresponding soft constraint as $C_s(\vec{X}, Z)$.

We say that a function f is an *accumulation function* if it maps sequences of values to a single value. We say f is *non-decreasing* if for every sequence of values $\vec{X}$ and value Y, $f(\vec{X}) \leq f(\vec{X}Y)$. Addressing the contractability of such constraints is made easier by the following characterization [13].

Proposition 3.1. *A soft constraint $m(\vec{X}) \leq Z$ is contractible iff m is a non-decreasing accumulation function.*

Consequently, we will also call such functions m contractible. Thus, to evaluate whether or not soft constraints are contractible we must consider the form of the violation measure, and whether it forms a contractible function.

[1] Also called violation cost [18].

Definition 3.2. A *violation measure* for a sublanguage L of a language L' is a function m which maps L' to the non-negative real numbers, such that if $w \in L$ then $m(w) = 0$. m is *proper* for L if for all words $w \in L'$, $m(w) = 0$ iff $w \in L$. A violation measure for a constraint $C(\vec{X})$ is a violation measure for L_C as a sublanguage of the static type $T(\vec{X})$.

For example, a use of AllDifferent might give the set $\mathbb{Z}$ of integers as the static type of each variable. A violation measure might then be the number of disequalities $X_i \neq X_j, i \neq j$ violated by a valuation for $\vec{X}$, or the number of variables equal to another variable, or the minimum absolute value of the sum over i of values c_i such that $\forall j\ j \neq i \rightarrow X_i + c_i \neq X_j$. It is easy to see that each of these defines a violation measure. The third is not a proper violation measure because, for example, the word 11233 gives rise to values of c_i of $-1, -1, 0, 1, 1$. Thus $m(11233) = 0$ but $11233 \notin L_C$.

Proper violation measures for a language L are a refinement of the characteristic function of L, and can be considered more intuitive than non-proper measures. Most violation measures in the literature are proper for their intended language. Although any function from words to non-negative reals can be considered a proper violation measure by appropriate choice of language L, in practice the hard constraint determines L and the violation measure is then designed to be proper. A non-proper measure can be considered misleading because a word w that violates the language L can have a violation measure of 0. We admit non-proper violation measures mainly because contractible approximations considered in Section 4 can be non-proper.

There are three broad classes of violation measures [15]: those based on constraint decomposition, edit distance, and graph properties. We address the first two classes in the following subsections. The richness of the graph property framework [3] makes it difficult to obtain general results on contractibility.

3.1. Decomposition-based Violation Measures

Many hard constraints can be decomposed into elementary constraints, whether naturally (such as the decomposition of AllDifferent into disequalities) or by design, as in [5]. Violation measures can be constructed by combining the violations of each elementary constraint. We define a general class of decomposition-based violation measures that includes as special cases: primal graph based violation costs [18], decomposition-based violation measures of [10], the value-based violation measure for GCC [18, 10], the measures used for the soft Sequence constraint [16] and the soft Cumulative constraint [17], and the class of decomposition-based measures discussed in [15]. We begin with several definitions.

A *weighted set* is a pair (S, w) where S is a set and w is a function mapping each element of S to a non-negative real number or ∞. Values not in S have weight 0. If these are the only values of weight 0 we say (S, w) is *proper*. A weighted set is a minor generalization of a multiset. A weighted set (S_1, w_1) is a *sub-weighted set* of weighted set (S_2, w_2) if, for every element $s \in S_1$, $w_1(s) \leq w_2(s)$. Union of weighted sets is defined by $(S_1, w_1) \cup (S_2, w_2) = (S_1 \cup S_2, w_1 + w_2)$. When a weighted set contains expressions with variables that are subject to substitution, the application of a substitution might unify elements of the set. Hence, $(S, w)\theta$ denotes $(S\theta, w')$ where $w'(s)$ is the sum of $w_1(s')$ over all $s' \in S$ such that $s'\theta \equiv s$.

We need to carefully formalize the notion of decomposition. Decomposition is a function that maps a constraint C with a given type T and a sequence of variables $\vec{X}$ to a tuple

$(\vec{X}, \vec{U}, T', S, w)$ where T' is an extension of T, $\vec{U}$ is a collection of new variables, $T(\vec{U})$ is their corresponding types, and (S, w) is a proper weighted set of elementary constraints over $\vec{X}\vec{U}$ such that $C(\vec{X}) \leftrightarrow \exists \vec{U}\ T'(\vec{U}) \wedge \bigwedge_{s \in S} s$. The weights are used only to emphasize some constraints in a decomposition over others; in particular, the infinite weight allows us to specify elementary constraints that must not be violated. An *unweighted decomposition* is one where all constraints in S have the same, non-zero weight. In that case, we may omit w. We write DECOMP$(C(\vec{X}))$ to express the weighted set (S, w), or simply S when the decomposition is unweighted. This definition of decomposition is very broad, perhaps too broad, since it allows the set of elementary constraints to vary radically as the length of $\vec{X}$ changes.

An *error function* e maps an elementary constraint and a valuation to a non-negative real number, representing the amount of error (or violation) of the constraint by the valuation. We require that $e(v, c) = 0$ iff c is satisfied by v. We extend e to weighted sets of constraints by defining $e(v, (S, w)) = (S', w')$ where $S' = \{e(v, s) \mid s \in S\}$ and $w'(x) = \sum_{v(s)=x} w(s)$.

A *combining function* maps a set of numbers and a weighting function to a single number. A combining function *comb* is *monotonic* if, whenever (S_1, w_1) is a sub-weighted set of (S_2, w_2), $comb(S_1, w_1) \leq comb(S_2, w_2)$. The function *comb* is *disjunctive* if for all weighted sets of reals (S, w), $comb(S, w) = 0$ iff $S = \{0\}$. Counting non-zero values, summation, sum of squares, and maximization are examples of monotonic, disjunctive combining functions; product and minimization are neither monotonic nor disjunctive.

Definition 3.3. A *decomposition-based violation measure* m for a constraint $C(\vec{X})$ with type T is based on a decomposition $(\vec{X}, \vec{U}, T', S, w)$ of $C(\vec{X})$, an error function e, and a combining function *comb* and is defined by, for each valuation v of $\vec{X}$,

$$m(C(v(\vec{X}))) = \min_{v'} comb(e(v', \text{DECOMP}(C(\vec{X}))))$$

where we minimize over all extensions v' of v to $\vec{U}$ that satisfy T'.

This definition was inspired by the formulation of hierarchical constraints in [6, 7]. The decomposition measures of [18, 10] can be obtained when the error function $e(v, c)$ returns 0 if v satisfies c and 1 otherwise, and the combining function is summation. The value-based measures of [18, 10, 16, 17] also use summation as the combining function, but use an error function that returns the amount by which the constraint c is violated by the valuation v. If we use maximization or the sum of squares in place of summation we have new violation measures similar to the *worst-case-better* and *least-squares-better* comparators of [6, 7]. Clearly many violation measures are available for a constraint by making different choices for the decomposition and the error and combining functions.

There is a powerful sufficient condition for a decomposition-based violation measure to be proper.

Proposition 3.4. *Let m be a decomposition-based violation measure for a constraint C. If comb is disjunctive then m is proper for L_C.*

We say that one formula $(\vec{X}, \vec{U}, T_1, S_1, w_1)$ is *covered* by another formula $(\vec{W}, \vec{V}, T_2, S_2, w_2)$ if there is a substitution θ that maps $\vec{X}$ into $\vec{W}$ and $\vec{U}$ into $\vec{V} \cup \vec{W} \cup \Sigma$, where Σ is a set of constants, such that $T_1(\vec{X}) = T_2(\vec{X}\theta)$, $(S_1, w_1)\theta$ is a sub-weighted set of (S_2, w_2) and $T_2(\vec{U}\theta) \subseteq T_1(\vec{U})$.

Example 3.5. The decomposition of $\textsc{AllDifferent}(\vec{X})$ into a set of disequalities is formalized as $(\vec{X}, \emptyset, T, S, w)$ where S is the set of disequalities and w gives every disequality a weight of 1. It is clear that the decomposition of $\textsc{AllDifferent}(\vec{X})$ is covered by that of $\textsc{AllDifferent}(\vec{X}Y)$ where the substitution is the identity.

$\textsc{Contiguity}$ is implemented in [12] essentially by the decomposition

$$\textsc{Contiguity}(\vec{X}) \leftrightarrow \exists \vec{L}, \vec{R}, X_0, X_{n+1} \bigwedge_{i=1}^{n} C'(X_{i-1}, R_{i-1}, L_i, X_i, R_i, L_{i+1}, X_{i+1})$$

for a constraint C'. This decomposition is formalized as $(\vec{X}, \vec{L}\vec{R}X_0X_{n+1}, T, S, w)$ where T gives all variables a type of $\{0, 1\}$, S is the set of C' constraints, and w gives every constraint a weight of 1. Alternatively, if contiguity is more important for variables nearer the right end of the sequence $\vec{X}$, we might weight each C' constraint by the largest index of a variable appearing in it. The decomposition of $\textsc{Contiguity}(\vec{X}Y)$ covers that of $\textsc{Contiguity}(\vec{X})$ where the substitution is the identity on $\vec{X}$, $\vec{L}$, and $\vec{R}$.

We can now provide a sufficient condition for a soft constraint with a decomposition-based violation measure to be contractible.

Proposition 3.6. *Let C_s be a soft constraint with a decomposition-based violation measure defined using a monotonic combining function. Let $(\vec{X}, \vec{U}, T_1, S_1, w_1)$ be the decomposition of $C(\vec{X})$ and $(\vec{X}Y, \vec{V}, T_2, S_2, w_2)$ be the decomposition of $C(\vec{X}Y)$. If $(\vec{X}, \vec{U}, T_1, S_1, w_1)$ is covered by $(\vec{X}Y, \vec{V}, T_2, S_2, w_2)$ via a substitution that is the identity on $\vec{X}$ then C_s is contractible.*

It follows that the constraints in Example 3.5 are contractible. Covering is only a sufficient condition for contractibility, as the following example demonstrates.

Example 3.7. Consider the definition of a rising sawtooth relation rs on variables $\vec{X}$. In such a relation, the subsequence of values in even numbered positions forms a non-decreasing sequence, and every value in odd numbered positions is greater than its immediately adjacent neighbours. This relation can be decomposed into elementary constraints as follows. The decomposition is defined recursively, but notably requires two recursive cases, corresponding to the distinction between odd and even length sequences.

$$\begin{aligned}
&\textsc{decomp}(rs([])) &&= \mathit{true}\\
&\textsc{decomp}(rs([X_1])) &&= \mathit{true}\\
&\textsc{decomp}(rs([X_1, X_2])) &&= X_1 \geq X_2\\
&\textsc{decomp}(rs([X_1, \ldots, X_{2n}, X_{2n+1}])) &&=\\
&\qquad \textsc{decomp}(rs([X_1, \ldots, X_{2n}])) \wedge X_{2n+1} \geq X_{2n}\\
&\textsc{decomp}(rs([X_1, \ldots, X_{2n}, X_{2n+1}, X_{2n+2}])) &&=\\
&\qquad \textsc{decomp}(rs([X_1, \ldots, X_{2n}])) \wedge X_{2n+1} \geq X_{2n+2} \wedge X_{2n+2} \geq X_{2n}
\end{aligned}$$

Consider the soft constraint derived from this decomposition by counting the number of violations. It is clear that the sufficient condition of Proposition 3.6 does not apply because there is no covering. Nevertheless, we can verify that a decomposition-based soft rs constraint is contractible. Note first that when $\vec{X}$ has even length $\textsc{decomp}(rs(\vec{X})) \subseteq \textsc{decomp}(rs(\vec{X}Y))$ and consequently the violation measure is non-decreasing in this case. When $\vec{X}$ has odd length the relationship is less obvious. However, we know that

$$\neg(X_{2n+1} \geq X_{2n}) \rightarrow \neg(X_{2n+1} \geq X_{2n+2}) \vee \neg(X_{2n+2} \geq X_{2n})$$

Hence, any valuation for the variables that gives rise to a violation of $X_{2n+1} \geq X_{2n}$ will also give rise to a violation of $X_{2n+1} \geq X_{2n+2}$, or $X_{2n+2} \geq X_{2n}$, or both. Thus the violation measure is non-decreasing in this case also. Since the violation measure is non-decreasing, the decomposition-based soft rs constraint is contractible.

Similarly, the violation measures derived from summing the amount of violation or taking the maximum amount of violation of any elementary constraint lead to contractible soft rs constraints.

This example demonstrates a major limitation of the sufficient condition in Proposition 3.6: it addresses only the syntactic structure of the decomposition. However some constraints, such as rs, require reasoning about the semantics of the elementary constraints in order to recognise that the decomposition-based soft constraint is contractible. (For rs we exploited the knowledge that $\geq$ forms a total order.)

3.2. Edit-based Violation Measures

The *edit-based* violation measures use a notion of edit distance, which is the minimum number of edit operations required to transform a word into a word of L_C. There are many possible edit operations but the common ones are: to substitute one letter for another, to insert a letter, to delete a letter, and to transpose two adjacent letters[2]. This class includes the *variable-based* violation measures [18, 10], the *object-based* measures of [3], and edit-based measures from [10].

Let $\alpha, \beta, \gamma, \delta$ be non-negative weights for the edit operations substitution, insertion, deletion and transposition, respectively, and let n_s, n_i, n_d, n_t be the number of the respective operations used in an edit. Then, for any language L, we define $m_L(w) = \min_{edits} \alpha n_s + \beta n_i + \gamma n_d + \delta n_t$ to be the minimum, over all edits that transform the word w to an element of L, of the weighted sum of the edit operations.

Definition 3.8. An *open edit-based violation measure* for a sublanguage L of L' is a weighted edit distance $m_{P(L)}$ for $P(L)$. An open edit-based violation measure m for L is *proper* if $m(w) = 0$ iff $w \in P(L)$ for every $w \in L'$.

We can characterize when an open edit-based violation measure is proper. Roughly, m is improper iff some edits have zero cost and these are able to edit some $w \in L' \backslash P(L)$ to $w' \in P(L)$.

Proposition 3.9. *Let m be an open edit-based violation measure for L where $P(L)$ is a sublanguage of L', with weights α, β, γ and δ.*

m is proper iff one of the following conditions holds:

- $\min\{\alpha, \beta, \gamma, \delta\} > 0$
- $\alpha = 0$, $\min\{\beta, \gamma\} > 0$ *and* $L' \cap SameLength(P(L)) \subseteq P(L)$
- $\beta = 0$, $\min\{\alpha, \gamma, \delta\} > 0$ *and* $L' \cap SubSeq(P(L)) \subseteq P(L)$
- $\gamma = 0$ *and* $L' \subseteq P(L)$
- $\delta = 0$, $\min\{\alpha, \beta, \gamma\} > 0$ *and* $L' \cap Perm(P(L)) \subseteq P(L)$
- $\alpha = \beta = 0$, $\gamma > 0$ *and* $L' \subseteq Shorter(P(L))$
- $\beta = \delta = 0$, $\min\{\alpha, \gamma\} > 0$ *and* $L' \cap Subset(P(L)) \subseteq P(L)$

[2]Edit distance based on these operations is known as Damerau-Levenshtein distance.

where, for any language L,
$SameLength(L)$ is the set of all words of the same length as a word of L,
$Shorter(L)$ is the set of all words the same length or shorter than a word of L,
$Perm(L)$ is the set of all permutations of words of L,
$SubSeq(L)$ is the set of all subsequences of a word of L, and
$Subset(L)$ is set of all words whose letters form a submultiset of the letters of a word of L.

In many cases, edit-based violation measures are contractible. This is a slight strengthening of a theorem of [15].

Theorem 3.10. *Let C_s be a soft constraint with an open edit-based violation measure. Suppose* $\min\{\alpha, \beta, \gamma\} \leq \delta$.

Then C_s is contractible.

An example from [15] shows that if $\delta < \min\{\alpha, \beta, \gamma\}$ then an edit-based soft constraint might be uncontractible. Thus Theorem 3.10 cannot be strengthened further without imposing extra conditions on C_s.

4. Contractible Approximations of Soft Constraints

As with hard constraints, when a soft constraint is uncontractible we can use a contractible approximation while the constraint is open.

We reformulate the notion of tight approximation for soft constraints of the form $m(\vec{X}) \leq Z$ as follows. A violation measure m_1 is an *approximation* of the violation measure m if, for all words $\vec{a}$, $m_1(\vec{a}) \leq m(\vec{a})$. We order violation measures with the pointwise extension of the ordering on the reals: $m_1 \leq m_2$ iff $\forall \vec{a}\ m_1(\vec{a}) \leq m_2(\vec{a})$. A contractible approximation m_1 to a violation measure m is *tight* if, for all contractible functions m_2, if $m_1 \leq m_2 \leq m$ then $m_2 = m_1$. Given two contractible approximations m_1 and m_2 to a violation measure m, we say m_2 is *tighter* than m_1 if $m_1 \leq m_2$. We write m^* to denote the tightest contractible approximation of m.

We can characterize the tightest contractible approximation of a violation measure, independent of how the violation measure is formulated.

Proposition 4.1. *Let m be a violation measure. The tightest contractible approximation to m is characterized by $m^*(\vec{a}) = \inf_{\vec{b}} m(\vec{a}\vec{b})$, where the infimum is taken over all finite sequences $\vec{b}$.*

4.1. Decomposition-based Violation Measures

One way to obtain a contractible approximation is to ignore parts of a decomposition that cause incontractibility. A *weakening* of a decomposition of a constraint $C(\vec{X})$ is a function that, for every sequence $\vec{X}$, maps the decomposition $(\vec{X}, \vec{U}, T', S, w)$ to $(\vec{X}, \vec{U}, T', S', w')$ where (S', w') is a sub-weighted set of (S, w). For this weakened decomposition we can apply the sufficient condition of Proposition 3.6.

Proposition 4.2. *Consider a decomposition-based violation measure m for a constraint $C(\vec{X})$ and a weakening W of the decomposition. Suppose m is defined via a monotonic combining function. If, for every sequence $\vec{X}$, the weakening of the decomposition of $C(\vec{X})$ is covered by the weakening of the decomposition of $C(\vec{X}Y)$ via a substitution that is the*

identity on $\vec{X}$ then the measure m' defined by using the weakened decompositions is a contractible approximation of m.

This result shows an approach to finding a contractible approximation to $C(\vec{X})$, but it provides no guarantee of finding a useful approximation. Like Proposition 3.6, it follows a syntactic approach and has the same inherent weakness.

4.2. Edit-based Violation Measures

Recall that an edit-based violation measure m is contractible if $\delta \geq \min\{\alpha, \beta, \gamma\}$ (Theorem 3.10). If $\delta < \min\{\alpha, \beta, \gamma\}$ then m might be uncontractible and we must consider contractible approximations. We can provide generic contractible approximations for edit-based soft constraints by modifying the weights to accord with the sufficient condition of Theorem 3.10.

Proposition 4.3. *Let m be an open edit-based violation measure for a constraint C with weights $\alpha, \beta, \gamma, \delta$ where $\delta < \min\{\alpha, \beta, \gamma\}$. Then the following violation measures are contractible approximations of m for C.*

(1) *m_1 based on weights $\delta, \beta, \gamma, \delta$*
(2) *m_2 based on weights $\alpha, \delta, \gamma, \delta$*
(3) *m_3 based on weights $\alpha, \beta, \delta, \delta$*
(4) *m_4 defined by $m_4(w) = \max\{m_1(w), m_2(w), m_3(w)\}$*

Clearly m_4 is the tightest of these approximations, and might be sufficient in practice. However, in general, this approximation is not tight, as the following example shows.

Example 4.4. Let $L = (abc)^*$, so that $P(L) = L \cup La \cup Lab$. Let $\alpha = \beta = \gamma = 4$ and $\delta = 1$. Consider $w = bbb(abc)^3ca$. Two kinds of edits are needed, addressing the initial b's and the trailing ca. Then $m(w) = 12$ from substituting for the first and third b, and deleting the last c. $m(wb) = 10$ using the same substitutions and two transpositions on c. Thus m is not contractible.

The tightest approximation to m has $m^*(w) = 10$. Now consider the approximations in Proposition 4.3. If we reduce α to 1 then $m_1(w) = 4$ by applying four substitutions. If we reduce β to 1 then $m_2(w) = 8$ by inserting a and c around each initial b and inserting ab before the last c. If we reduce γ to 1 then $m_3(w) = 4$ by deleting the three b's and the last c. Thus $m_4(w) = 8$.

This shows that m_4 is not the tightest contractible approximation to m, since $m_4(w) \neq m^*(w)$.

The question now arises: how to express m^* in edit-based terms so that a closed propagator for $m(\vec{X}) \leq Z$ might be adapted to implement $m^*(\vec{X}) \leq Z$. Disappointingly, this turns out to be impossible, in general.

Theorem 4.5. *There is an open edit-based violation measure m for a language L such that its tightest contractible approximation cannot be expressed as a proper (not necessarily open) edit-based violation measure on any language.*

The language and violation measure demonstrating this claim are those from Example 4.4. Given that the language is so simple, we can expect that many uncontractible edit-based violation measures cannot be tightly approximated by a contractible edit-based violation measure. This contrasts markedly with work on hard constraints, where tight contractible

approximations of several uncontractible hard constraints can be formulated in terms of the original hard constraint [14, 15]. It suggests that the edit-based implementation of the closed constraint is not a suitable basis for implementing the tight approximation.

5. Conclusions

We have investigated violation measures for soft constraints based on decomposition and edit-distance. We defined a class of decomposition-based violation measures that generalizes several previous works. For both forms of violation measures we identified sufficient conditions for constraints to be proper and strengthened results of [15] on when they are contractible. Finally, we found that the edit-based framework is not expressive enough to represent tight contractible approximations, in general.

Acknowledgements

Thanks to the referees for their comments, which helped improve this paper.

References

[1] R. Barták, Dynamic Global Constraints in Backtracking Based Environments, *Annals of Operations Research* 118, 101–119, 2003.
[2] N. Beldiceanu, M. Carlsson and J.-X. Rampon, Global Constraint Catalog, SICS Technical Report T2005:08. Current version available at `http://www.emn.fr/x-info/sdemasse/gccat/`
[3] N. Beldiceanu and T. Petit, Cost Evaluation of Soft Global Constraints, CPAIOR, 80–95, 2004.
[4] C. Bessière, E. Hebrard, B. Hnich, Z. Kiziltan and T. Walsh, SLIDE: a useful special case of the CardPath constraint, ECAI 2008, 475–479, 2008.
[5] C. Bessière, G. Katsirelos, N. Narodytska, C.-G. Quimper and T. Walsh, Decompositions of All Different, Global Cardinality and Related Constraints, IJCAI 2009: 419–424.
[6] A. Borning, B. Freeman-Benson and M. Wilson, Constraint Hierarchies, *Lisp and Symbolic Computation* 5, 3, 223–270, 1992.
[7] A. Borning, M.J. Maher, A. Martindale and M. Wilson, Constraint Hierarchies and Logic Programming, ICLP, 149–164, 1989.
[8] S. Brand, N. Narodytska, C.-G. Quimper, P.J. Stuckey and T. Walsh, Encodings of the Sequence Constraint, CP 2007, LNCS 4741, Springer, 210–224.
[9] R. Dechter, *Constraint Processing*, Morgan Kaufmann, 2003.
[10] W-J. van Hoeve, G. Pesant and L-M. Rousseau, On Global Warming: Flow-Based Soft Global Constraints, *Journal of Heuristics*, 12(4-5), 347–373, 2006.
[11] W-J. van Hoeve and J-C. Régin, Open Constraints in a Closed World, CPAIOR'06, LNCS 3990, Springer, 244–257, 2006.
[12] M.J. Maher, Analysis of a Global Contiguity Constraint, Proc. Workshop on Rule-Based Constraint Reasoning and Programming, 2002.
[13] M.J. Maher, Open Contractible Global Constraints, IJCAI 2009, 578–583.
[14] M.J. Maher, Open Constraints in a Boundable World, CPAIOR 2009, LNCS 5547, 163–177.
[15] M.J. Maher, SOGgy Constraints: Soft Open Global Constraints, CP 2009, LNCS 5732, 584–591, 2009.
[16] M. Maher, N. Narodytska, C.-G. Quimper and T. Walsh, Flow-based propagators for the SEQUENCE and related global constraints, CP 2008, LNCS 5202, 159–174.
[17] T. Petit and E. Poder, The Soft Cumulative Constraint, Research Report TR09/06/info, Ecole des Mines de Nantes, 2009.
[18] T. Petit, J-C. Régin and C. Bessière, Specific Filtering Algorithms for Over-constrained Problems, CP 2001, LNCS 2239, Springer, 451–463.
[19] F. Rossi, P. van Beek and T. Walsh (Eds), *Handbook of Constraint Programming*, Elsevier, 2006.
[20] C. Schulte and G. Tack, Weakly Monotonic Propagators, CP 2009, LNCS 5732, 723–730, 2009.

Technical Communications of the International Conference on Logic Programming, 2010 (Edinburgh), pp. 124–133
http://www.floc-conference.org/ICLP-home.html

DEDICATED TABLING FOR A PROBABILISTIC SETTING

THEOFRASTOS MANTADELIS [1] AND GERDA JANSSENS [1]

[1] Departement Computerwetenschappen, Katholieke Universiteit Leuven
Celestijnenlaan 200A - bus 2402, 3001 Heverlee, Belgium
E-mail address: {Theofrastos.Mantadelis,Gerda.Janssens}@cs.kuleuven.be

Abstract. ProbLog is a probabilistic framework that extends Prolog with probabilistic facts. To compute the probability of a query, the complete SLD proof tree of the query is collected as a sum of products. ProbLog applies advanced techniques to make this feasible and to assess the correct probability. Tabling is a well-known technique to avoid repeated subcomputations and to terminate loops. We investigate how tabling can be used in ProbLog. The challenge is that we have to reconcile tabling with the advanced ProbLog techniques. While standard tabling collects only the answers for the calls, we do need the SLD proof tree. Finally we discuss how to deal with loops in our probabilistic framework. By avoiding repeated subcomputations, our tabling approach not only improves the execution time of ProbLog programs, but also decreases accordingly the memory consumption. We obtain promising results for ProbLog programs using exact probability inference.

1. Introduction

ProbLog [4] is a probabilistic framework that extends Prolog with probabilistic facts and answers several kinds of probabilistic queries. While the framework includes different inference methods, we focus for this paper on the exact probability inference method.

The implementation of ProbLog [8] is based on the use of tries [5] and reduced ordered binary decision diagrams (ROBDDs) [1, 2]. The execution of ProbLog programs uses SLD-resolution to collect all the proofs for a query. ProbLog gathers for each successful proof of the query the list of probabilistic facts the proof uses and compactly represents all proofs in a trie. Such a trie is then considered to be a sum of products (a disjunction of conjunctions of probabilistic facts). ROBDDs are used to solve the disjoint sum problem and to obtain the correct probability of the query.

The challenge is to find out how tabling can be combined with the ProbLog execution mechanism. Tabling mechanisms are available in XSB [11], YAP [12] and other Prolog systems. The basic idea is to collect the answers of a tabled subgoal in a table and, when the subgoal is re-encountered, to reuse the tabled answers instead of computing them. As a consequence of this memoization, tabling ensures termination of programs with the bounded term-size property, i.e. programs where the size of subgoals and answers produced during an evaluation is less than some fixed number. In the case of ProbLog, tabling answers is not sufficient, as the proofs are needed too. Also loops have to be dealt with correctly.

1998 ACM Subject Classification: I.2.2, I.2.8, D.1.6, G.3.
Key words and phrases: Tabling, Loop Detection, Probabilistic Logical Programming, ProbLog.

Technical Communications of the 26th International Conference on Logic Programming, Edinburgh, July, 2010
Editors: Manuel Hermenegildo, Torsten Schaub
LIPIcs - Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/LIPIcs.ICLP.2010.124

The PRISM [14] assumptions such as exclusiveness and no loops imply that PRISM computations are simpler, in the sense that they do not have to deal with the disjoint sum problem. PRISM contains a linear tabling system [15]. Only when PRISM is executing its learning algorithms, is its tabling extended to do something special, namely to build support graphs which represent the shared structure of explanations for an observed goal. The support graphs play a central role in efficient EM learning of PRISM programs. The proofs ProbLog needs to compute are somewhat similar to these explanations.

This paper extends our early work [9]. The contributions of this paper are the identification of the necessary tabling mechanism for ProbLog programs and their implementation while respecting the current ProbLog optimizations, such as the exploitation of tries and the optimized translation of tries into ROBDDs.

ProbLog's motivating link discovery applications and other typical ProbLog programs have only ground goals. In order to table them, we represent the SLD proof tree as **nested tries**. We implemented a light-weight dedicated tabling for ground goals that supports nested tries and obtained impressive time improvements for some classes of programs. By the virtue of the nested tries, we also realize suffix sharing and thus a substantial memory compaction. By adding **loop detection**, path-finding programs, typical for link discovery, also benefit from the memoization.

The paper is structured as follows. First, we briefly introduce ProbLog and its relevant implementation details in Section 2. We present the nested tries and identify the necessary tabling support in Section 3. Transforming the nested tries to a sum of products efficiently is in Section 4. Section 5 contains the experimental evaluation. Related work and conclusions are in Sections 6 and 7 respectively.

2. ProbLog

ProbLog [4] is essentially an extension of Prolog where facts are labeled with probabilities[1] that they belong to a randomly sampled program. As such, a ProbLog program specifies a probability distribution over all its possible non-probabilistic subprograms. The success probability of a query is defined as the probability that it succeeds in such a random subprogram. ProbLog follows the distribution semantics [13]. We use the ProbLog program from Example 2.1 to describe how ProbLog calculates the exact probability of a query. The graph in Figure 1 is represented by the probabilistic facts `edge/2`. The rest of the program is normal Prolog code for finding a path in a graph. Consider the query `path(1,4)`.

Example 2.1 (Program path/2)**.**

```
path(X, Y):- path(X, Y, [X]).
path(X, Y, P):- edge(X, Z), Y \== Z, \+ member(Z, P), path(Z, Y, [Z|P]).
path(X, Y, _):- edge(X, Y).
```

ProbLog first uses SLD-resolution to collect all the proofs of a query. Actually, ProbLog gathers for each successful proof of the query the list of probabilistic facts the proof uses. For the `path(1,4)` query, ProbLog collects eight successful proofs and for each proof a list of probabilistic `edge/2` facts as shown in Figure 2. ProbLog uses a trie to represent such a set of lists. The trie of a query is built during SLD-resolution: as soon as a successful proof is found, it is added to the trie. We use **proofs** for SLD-refutations as well as for lists of probabilistic facts. The trie for our example is given in Figure 2.

[1]Probabilistic facts are mutually independent random variables.

```
0.9::edge(1, 2).  0.9::edge(2, 1).  0.2::edge(5, 4).
0.4::edge(6, 5).  0.4::edge(5, 6).  0.2::edge(4, 5).
0.8::edge(2, 3).  0.8::edge(3, 2).  0.7::edge(1, 6).
0.5::edge(2, 6).  0.5::edge(6, 2).  0.7::edge(6, 1).
0.7::edge(5, 3).  0.7::edge(3, 5).
0.6::edge(3, 4).  0.6::edge(4, 3).
```

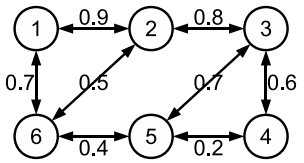


Figure 1: Graph of Example 2.1.

```
edge(1,2),edge(2,3),edge(3,5),edge(5,4)
edge(1,2),edge(2,3),edge(3,4)
edge(1,2),edge(2,6),edge(6,5),edge(5,3),edge(3,4)
edge(1,2),edge(2,6),edge(6,5),edge(5,4)
edge(1,6),edge(6,2),edge(2,3),edge(3,5),edge(5,4)
edge(1,6),edge(6,2),edge(2,3),edge(3,4)
edge(1,6),edge(6,5),edge(5,3),edge(3,4)
edge(1,6),edge(6,5),edge(5,4)
```

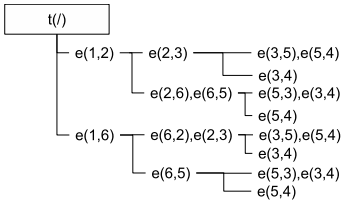


Figure 2: The successful proofs and the respective trie for *path(1,4)* of Example 2.1.

Now, think of the `edge/2` facts as Boolean random variables indicating whether the facts are in the logic program. As a consequence, each proof corresponds to a conjunction of random variables and as a whole the trie represents a disjunction of conjunctions of probabilistic facts, also known as a sum of products. ProbLog needs to assess the probability of the sum of products, which has been shown to be #P-Hard [17]. ProbLog transforms the trie into a ROBDD in two steps. First, starting from the trie, a so-called ROBDD script is generated. Secondly, the ROBDD script is executed by a ROBDD package. From the ROBDD, ProbLog calculates the success probability (0.53864 for our example) of the query by a bottom-up dynamic programming algorithm over the ROBDD structure [8].

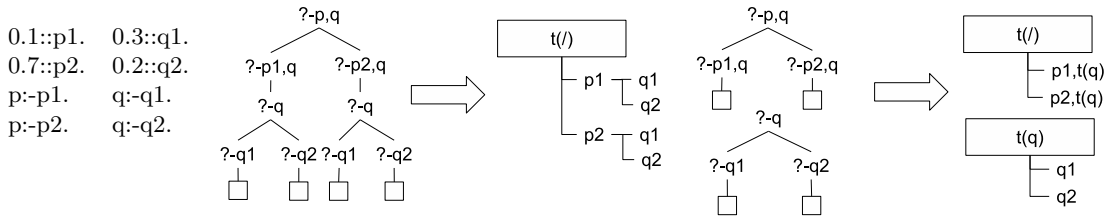
As argued in [8], representing the proofs compactly in a trie establishes prefix sharing and this turns out to be indispensable for the typical ProbLog mining applications. This prefix sharing can be clearly seen in Figure 2. In this paper we show how our tabling also establishes suffix sharing and thus further reduces the memory consumption.

3. Memoization by Tabling

Typical ProbLog goals are ground; indeed, in a probabilistic framework, one is interested in the probability that a goal can be proven, rather than in what the answers are. While our work can be generalized for non-ground goals, we focus on ground goals for this paper.

ProbLog collects all the SLD-refutations of a query as lists of probabilistic facts in a trie. Our tabling builds a forest of SLG-trees [3], one for the original query and one for each tabled subgoal. For each tabled goal, we need to memoize its contribution to the trie: we break up a single trie into a set of nested tries. The **nested** trie of a goal represents the successful proofs of the goal just as a normal trie does, but the parts of the trie that are contributed by other tabled subgoals are replaced by a reference to the trie of that subgoal.

Consider the query `?-p,q.` for the program of Figure 3a. The SLD-tree and the corresponding trie are in Figure 3b. Tabling the predicate `q/0` avoids recomputation during resolution and results in the forest of SLG-trees and the nested tries shown in Figure 3c. Note that the nested trie for the subgoal `q` is denoted by `t(q)` and `t(/)` denotes the trie of the topquery.



(a) Tabling q/0. (b) SLD-tree and trie. (c) SLG-trees and the nested tries.

Figure 3: Example of SLD-resolution and SLG-resolution trees and tries.

When ProbLog uses tabling while proving the topquery, it constructs a set of nested tries. This set of nested tries is equivalent with the trie that a non-tabled program would generate: it contains all the information about the complete successful proofs, the SLD-refutations, of the topquery. In the non-tabled evaluation, repeated subcomputations give rise to tries with repeated suffixes. The use of nested tries such as `t(q)` establishes suffix-sharing which reduces substantially the memory consumption of ProbLog.

Tabling uses a suspension/resumption mechanism to built a forest of SLG-trees. Tabling keeps in its tables an entry for each tabled subgoal that contains the nested trie of the subgoal. When ProbLog encounters the first instance of a tabled subgoal, the **generator** subgoal, tabling suspends the resolution of the parent goal, creates an entry for the subgoal with an empty nested trie, and starts proving it. As soon as a successful proof for the subgoal is found, it is added to its nested trie. Note that if the subgoal fails, no proof will be added and the nested trie will remain empty.

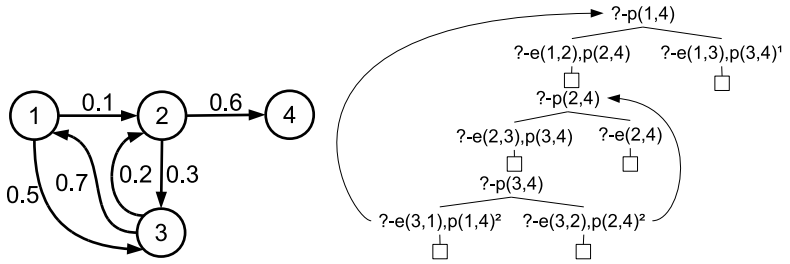
We eagerly collect the proofs for the generator goals. For programs without loops, tabling deals completely with a tabled subgoal before the parent goal is resumed and it is known whether the subgoal failed or succeeded. If a tabled subgoal fails, resumption of the parent goal fails its current proof. If the tabled subgoal succeeds then, on resumption of the parent goal, a reference to the nested trie of the tabled subgoal is added to the current proof of the parent goal. For subsequent occurrences of tabled subgoals, the **consumer** subgoals, tabling avoids recomputation by adding a reference to the appropriate nested trie in the current proof.

An attentive reader might indeed have noticed that the path program of Example 2.1 is a version that encodes loop detection explicitly by using `absent/2`. That version does not benefit from tabling as all the calls to `path` are different. The `path/2` program in Example 3.1 has no code to detect loops. We use this program and the graph of Figure 4a to explain how tabling should support loop handling in a probabilistic framework.

Example 3.1 (Program path/2 without loop detection)**.**

```
0.1::edge(1, 2).    0.5::edge(1, 3).    0.7::edge(3, 1).
0.3::edge(2, 3).    0.2::edge(3, 2).    0.6::edge(2, 4).
path(X, Y):- edge(X, Z), Y \== Z, path(Z, Y).
path(X, Y):- edge(X, Y).
```

For the query `path(1,4)` on this graph with the program of Example 2.1, we collect the following two proofs: `edge(1,2), edge(2,4)` and `edge(1,3), edge(3,2), edge(2,4)`. While proving the query `path(1,4)` with the program of Example 3.1, one will enter in infinite loops, such as the one between nodes 2 and 3 due to `edge(2,3)` and `edge(3,2)`. The



(a) Graph of Example 3.1. (b) SLG-trees with goal suspension.

Figure 4: Graph and SLG-trees of Example 3.1.

SLD-tree for `path(1,4)` is indeed infinite. The SLG-trees are in Figure 4b with backward arcs indicating the detected loops.

In the presence of loops, tabling normally uses a completion mechanism to ensure that all answers are returned to all consumers. For our ground ProbLog goals, this boils down to returning information about failure or success and in case of success the nested trie. A simple way to achieve this for consumer goals that give rise to a loop, is to assume that the generator goal will succeed and add the reference to the partially completed nested trie to the parent goal proof. Although a nested trie represents all the proofs for the subgoal, we do not need its final value to re-use it, as we put a reference to it in the other tries.

Now, our nested tries no longer only contain successful proofs. We optimistically assumed success for consumer goals giving rise to a loop. If none of the goals involved in the loop has a finite successful proof, they all fail. In this case, the nested tries contain references to failed subgoals. We deal with this during the ROBDD script generation step.

We have built a light-weight tabling prototype which allow us to experiment without having to change the ProbLog implementation. A program transformation is used to add tabling specific predicate calls that manipulate the extra tabling data structures. In this transformation we use findall to implement the eager proof collection for generator goals.

We implemented SLG tabling with suspension and resumption. As we have ground goals, we do not collect answers for goals, but their success or failure. Due to the probabilistic context, we need a special mechanism to construct a nested trie for each tabled subgoal, as we do not want to lose the prefix sharing required by ProbLog. Also note that ProbLog needs all the successful proofs, whereas clever normal tabling would stop after one successful proof for a ground goal.

For consumer goals that give rise to a loop, our optimistic approach assumes success of the goal, but in the end the goal might fail. Our nested tries then contain references to failed subgoals. This failure needs to be detected during the BDD script generation step.

4. Extracting the Boolean Formula from the Set of Nested Tries

Tabling computes for the original query a set of nested tries, which contain all the proofs of the query. As mentioned in Section 2, the ROBDD script generation step of ProbLog extracts all collected proofs from the Trie and generates a Boolean formula that expresses the proofs as a sum of products. In this section we are going to describe how to generate the Boolean formula from the set of nested tries. A naive implementation would have a performance cost similar to that of the non-tabled SLD resolution. We use dynamic

programming to implement a top down traversal of the nested tries with loop detection. While generating the formula, we want to preserve the prefix sharing present in the tries and to exploit the suffix sharing. Dynamic programming enables the re-usage of completely unfolded tries, while re-usage of partially unfolded tries can be realized by introducing an ancestor subset check.

Starting from the nested trie of the original query, we reconstruct its proofs as a normal trie by unfolding the nested tries, but we also establish suffix sharing. To unfold a reference to another nested trie $t(g)$, we replace in the partial trie the reference by the nested trie itself. For each branch of the partial trie, we keep an **ancestor list**: initially it is empty and each time we unfold a reference to a $t(g)$, the g is added. In programs without loops, a nested trie $t(g)$ is unfolded only once, dynamic programming makes the other occurrences of references to $t(g)$ reuse the first unfolding and as such we now have tries with suffix sharing.

For programs with loops, we start from a finite representation of the infinite SLD-tree. The ancestor list enables us to detect loops during unfolding. Loops typically give rise to proofs that do not contribute to the final probability.

When unfolding detects a loop, we can prune the current proof: either because is a failing proof or because it gives rise to non-minimal proofs. If pruning results in an empty trie, which encodes failure, we can prune the parent branch. Figure 5 shows the nested tries, that contain infinite loops, for the query of Example 3.1.

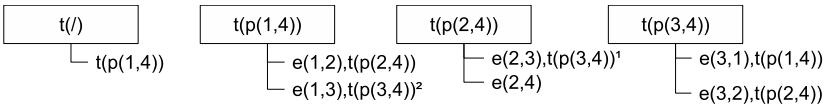


Figure 5: Nested tries of Example 3.1.

We have to be careful not to miss proofs because of the dynamic programming memoization. During unfolding, a nested trie typically occurs in different proofs. Its position in the SLD tree determines whether it gives rise to a loop or not. Consider $t(p(3,4))$ of Figure 5, suppose we first encounter its occurrence annotated with ^{1}. The two proofs this occurrence gives rise to, have loops and thus they are pruned. But, the second occurrence annotated with ^{2} belongs to a proof without a loop.

The $t(p(3,4))$ example shows that re-using results for nested tries that introduced loops, is not safe as different occurrences might give rise to different proofs. Actually, it depends on the context of the occurrence of the nested trie, in particular on the ancestor list of the occurrence. In order to improve suffix sharing, we start from the following observation. When two occurrences of a reference to a nested trie have exactly the same ancestor list, then obviously the unfolding of the references will introduce exactly the same loops. Generally, the occurrence of a goal will introduce at least the same loops as a previous occurrence, when the ancestor list of the previous occurrence is a subset of the current ancestor list.

5. Experiments

Our tabling directly interacts with the first two execution steps of ProbLog, SLD-resolution and ROBDD script generation. The third step is affected indirectly. It is well-known that ROBDD packages use heuristics while constructing a ROBDD and that their behaviour depends on the input, i.e. different inputs describing the same Boolean formula

	Memory (Bytes)		SLD/SLG resolution		ROBDD Script generation		ROBDD Script execution	
Day	non-tab	tab	non-tab	tab	non-tab	tab	non-tab	tab
1	352	912	0	0	0	0	5	5
2	1048	2148	0	0	0	0	5	5
13	184941472	15744	115400	2	2584	4	802	37
14	554824408	16980	380011	3	7938	4	2380	36
167	-	206088	-	25	-	89	-	9728
1600	-	1977276	-	262	-	3587	-	-

Table 1: Results for the weather program.

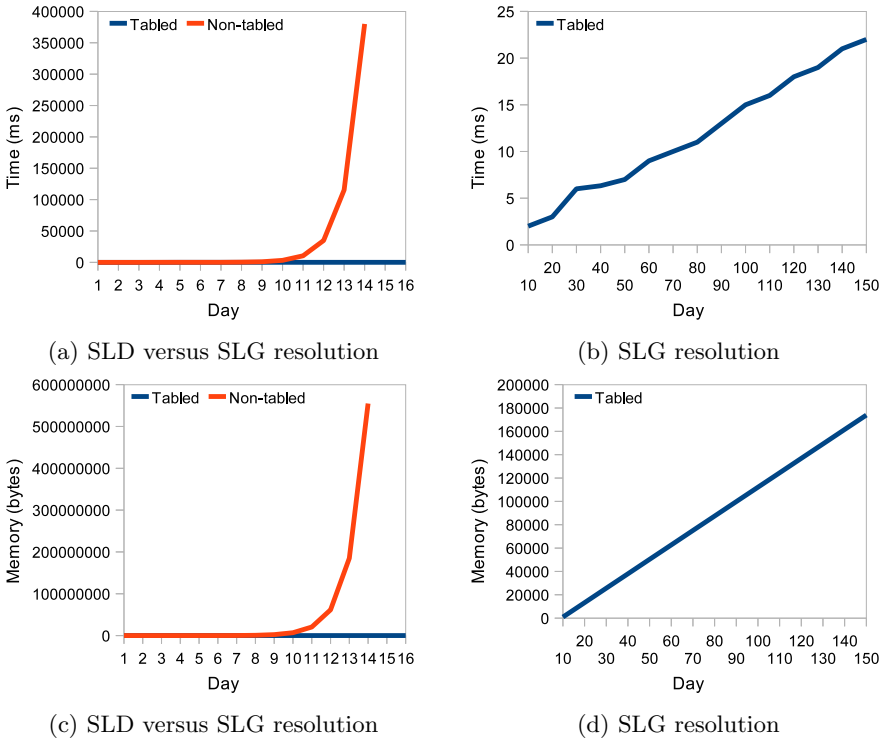
can give rise to different results and/or execution times. We address the following questions: (1) How does our tabling implementation perform both in time and in space for the SLD-resolution? (2) How do the nested tries compare with their flatten equivalents during the ROBDD script generation and the ROBDD script execution? (3) How do the nested tries with loops perform and what are the effects of the ancestor subset check?

Our benchmarks represent two typical categories of problems. Our **weather** benchmark is an example of a {Hidden} Markov Model (HMM). The value of the current time state depends on the value of the previous time state. It is well known that time series problems, when naively implemented, are of exponential complexity. Using tabling for this type of problems we expect significant improvement as memoization reduces the complexity of the problem's SLD-resolution to linear. The size of the weather problem is determined by the "Day" argument.

From the link discovery [4] applications, we took a **graph** benchmark, namely a number of graphs from the biomine database [16]. This benchmark expresses connections between various types of objects such as genes, proteins, tissues, etc and predicts relationships among them. We use the program of Example 2.1 for the non-tabled version and the program of Example 3.1 for the tabled version. For our experiments we used the first sample of graphs and the queries of [4]. The size is determined by the number of edges of the graph and the interconnectivity between the nodes. As these graphs are cyclic, loop handling is necessary.

All the experiments are done on an Intel^R Core^{TM}2 Duo CPU at 3.00GHz with 2GB of RAM memory running Ubuntu 8.04.2 Linux under a usual load. The reported times are the averages of five runs from which we dropped the best and worst, and all times are in milliseconds. For the SLD resolution and the ROBDD script generation, we used a time-out of 1 hour, while for the ROBDD script execution we used a 1 minute time-out (for our benchmarks most ROBDDs either will be built within one minute or run out of memory).

For **weather**, figure 6a shows that the SLD-resolution execution times of the non-tabled version are exponential with respect to the "Day" argument, while the tabled version is linear. Figure 6b shows the scaling of the SLD-resolution for queries that the non-tabled version fails to compute, as it exceeds the available memory. In Table 1, we see that the tabled version manages to compute "Day 167", while the non-tabled version stops at "Day 14". The tabled version is limited by the ROBDD script execution step that generates the ROBDD using a state-of-the-art ROBDD tool. For weather, the tabled version outperforms the non-tabled version in all stages including the ROBDD script execution. The memory usage to represent the proofs goes from exponential to linear for the tabled version as shown in Figure 6c. In Figure 6d we see the gain in memory is similar to the gain in time. This can be explained by the suffix sharing in the nested tries.



(a) SLD versus SLG resolution

(b) SLG resolution

(c) SLD versus SLG resolution

(d) SLG resolution

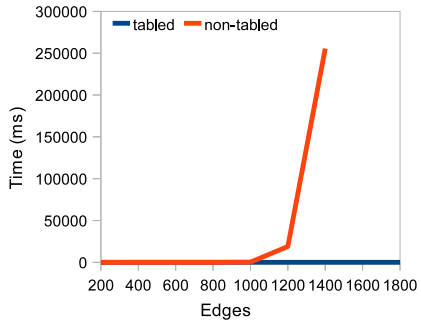
Figure 6: SLD/SLG-resolution times and memory consumption for the weather program.

In the graph benchmark, we study the benefits of tabling in combination with loop handling. As shown in Figure 7a, the tabled version has a significant performance improvement for the SLD-resolution. Figure 7b shows that increasing the number of edges in the graph affects the SLG-resolution linearly. Table 2 displays the results of the graph benchmark. The effect of tabling on the memory usage is a bit different now. Using nested tries for tabling favours suffix sharing rather than prefix sharing. It seems that in some graphs prefix sharing is more important for memory compaction than suffix sharing. However, in bigger graphs, the nested tries are again improving significantly memory consumption. That the tabled version requires to construct the nested tries even for goals that fail or succeed without probabilistic facts, introduces a significant minimum memory cost.

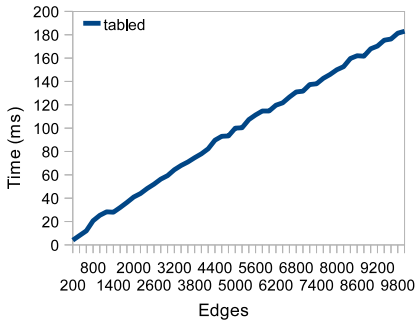
Unfortunately, we notice in Figure 7c that all the versions behave exponentially when summing up the SLD/SLG-resolution times and the ROBDD script generation times. Figure 7d presents the times for generating the ROBDD scripts from the nested tries with loops and the performance gain of the ancestor subset check. While the tabled version without the ancestor subset check underperformed the non-tabled in total time, the version with the ancestor subset check has significant performance gains in all cases.

6. Related Work

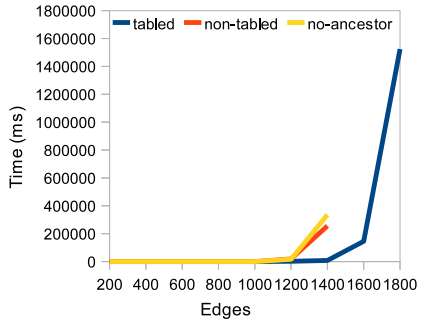
This paper investigates a dedicated form of tabling: memoization of all the proofs for ground goals in nested tries. Our work is similar to the PRISM [15, 14] tabling mechanism, as both mechanisms are restricted to grounded goals and both are memorising all the proofs.



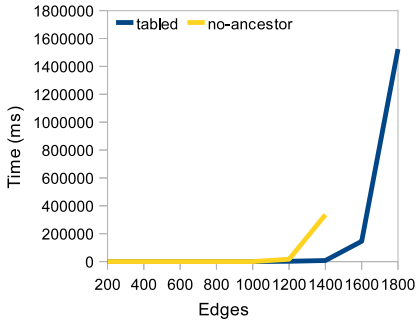
(a) SLD versus SLG resolution



(b) SLG resolution



(c) SLD/SLG-resolution & ROBBD script generation



(d) ROBBD script generation

Figure 7: Graph program results.

Edges	Memory (Kilobytes)		SLD/SLG resolution		ROBBD Script generation			ROBBD Script execution		
	non-tab	*tab*	*non-tab*	*tab*	*non-tab*	*tab*	*no-anc*	*non-tab*	*tab*	*no-anc*
800	< 1	192	19	17	0	22	37	3	3	4
1000	32	239	245	20	1	107	258	36	33	37
1200	3303	286	18928	25	28	1982	16844	119	214	196
1400	39020	333	255546	28	473	7832	335853	455	454	278
1600	-	380	-	33	-	145669	-	-	26789	-
1800	-	426	-	37	-	1523948	-	-	-	-

Table 2: Results for the graph program.

One could compare ProbLog's nested tries with PRISM support graphs. The differences are that PRISM assumes the exclusiveness condition for the proofs, while ProbLog does not, and that ProbLog requires the handling of loops as it is intended for link discovery in graphs. Another example of tabling that needs the memoization of proofs, is in the scope of justification [6]. Note that tabling in justification keeps only one proof [10] instead of all the proofs, and that it requires tabling of non-grounded goals. Finally, [7] proposes tabling for another ProbLog inference method, namely Monte Carlo sampling.

7. Conclusions and Future Work

We successfully identified the requirements for a tabling mechanism for ProbLog and we realized a light-weight implementation of such a mechanism. Our experiments have shown that tabling is definitely beneficial for the SLD-resolution step, where the memory and time consumption can go from exponential to linear. Nested tries establish prefix and suffix sharing. We presented how loop handling can be performed using the nested tries. Tabling also affects the next ROBDD related steps of ProbLog. For benchmarks without loops, tabling further reduces the execution times, as these steps also benefit from the compaction by the sharing. In the graph benchmark, we see that tabling improves the overall performance of the system. While the improvement for SLD resolution is remarkable, the work is partly transferred to the ROBDD script generation step. In the presence of loops, the ancestor subset check is indispensable. We want to extend tabling for non-ground goals and use tabling for approximate inference methods.

Acknowledgements: This research is supported by GOA/08/008 "Probabilistic Logic Learning".

References

[1] Sheldon B. Akers. Binary decision diagrams. *IEEE Trans. Computers*, 27(6):509–516, 1978.
[2] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986.
[3] Weidong Chen and David S. Warren. Tabled evaluation with delaying for general logic programs. *Journal of the ACM*, 43(1):20–74, 1996.
[4] Luc De Raedt, Angelika Kimmig, and Hannu Toivonen. ProbLog: A probabilistic prolog and its application in link discovery. In *Proceedings of IJCAI*, pages 2462–2467, 2007.
[5] Edward Fredkin. Trie Memory. *Communications of the ACM*, 3:490–499, 1962.
[6] Hai-Feng Guo, C. R. Ramakrishnan, and I. V. Ramakrishnan. Speculative beats conservative justification. In *Proceedings of ICLP*, pages 150–165, 2001.
[7] Angelika Kimmig, Bernd Gutmann, and Vítor Santos Costa. Trading memory for answers: Towards tabling ProbLog. In *Proceedings of SRL*, 2009.
[8] Angelika Kimmig, Vítor Santos Costa, Ricardo Rocha, Bart Demoen, and Luc De Raedt. On the efficient execution of ProbLog programs. In *Proceedings of ICLP*, pages 175–189, 2008.
[9] Theofrastos Mantadelis and Gerda Janssens. Tabling relevant parts of SLD proofs for ground goals in a probabilistic setting. In *Proceedings of CICLOPS*, 2009.
[10] Giridhar Pemmasani, Hai-Feng Guo, Yifei Dong, C. R. Ramakrishnan, and I. V. Ramakrishnan. Online justification for tabled logic programs. In *LNCS: Logic Programming*, pages 500–501, 2003.
[11] Prasad Rao, Konstantinos F. Sagonas, Terrance Swift, David S. Warren, and Juliana Freire. XSB: A system for effciently computing wfs. In *Proceedings of LPNMR*, pages 431–441, 1997.
[12] Ricardo Rocha, Fernando Silva, and Vítor Santos Costa. A Tabling Engine for the Yap Prolog System. In *Proceedings of AGP*, 2000.
[13] Taisuke Sato. A statistical learning method for logic programs with distribution semantics. In *Proceedings of ICLP*, pages 715–729. MIT Press, 1995.
[14] Taisuke Sato and Yoshitaka Kameya. Parameter learning of logic programs for symbolic-statistical modeling. *JAIR*, 15:391–454, 2001.
[15] Taisuke Sato and Yoshitaka Kameya. Statistical abduction with tabulation. In *Computational Logic: Logic Programming and Beyond, Essays in Honour of Robert A. Kowalski, Part II*, pages 567–587, 2002.
[16] Petteri Sevon, Lauri Eronen, Petteri Hintsanen, Kimmo Kulovesi, and Hannu Toivonen. Link discovery in graphs derived from biological databases. In *Proceedings of DILS*, pages 35–49, 2006.
[17] Leslie G. Valiant. The complexity of enumeration and reliability problems. *SIAM Journal on Computing*, 8(3):410–421, 1979.

Technical Communications of the International Conference on Logic Programming, 2010 (Edinburgh), pp. 134–143
http://www.floc-conference.org/ICLP-home.html

TIGHT SEMANTICS FOR LOGIC PROGRAMS

LUÍS MONIZ PEREIRA [1] AND ALEXANDRE MIGUEL PINTO [1]

[1] Centro de Inteligência Artificial (CENTRIA)
Departamento de Informática, Faculdade de Ciências e Tecnologia,
Universidade Nova de Lisboa, 2829-516 Caparica, Portugal
E-mail address: lmp@di.fct.unl.pt

E-mail address: amp@di.fct.unl.pt

Abstract. We define the Tight Semantics (TS), a new semantics for *all* NLPs complying with the requirements of: 2-valued semantics; preserving the models of SM; guarantee of model existence, even in face of Odd Loops Over Negation (OLONs) or infinite chains; relevance; cumulativity; and compliance with the Well-Founded Model.

When complete models are unnecessary, and top-down querying (*à la* Prolog) is desired, TS provides the 2-valued option that guarantees model existence, as a result of its relevance property. Top-down querying with abduction by need is rendered available too by TS. The user need not pay the price of computing whole models, nor that of generating all possible abductions, only to filter irrelevant ones subsequently.

A TS model of a NLP P is any minimal model (MM) M of P that further satisfies $\widehat{P}$—the program remainder of P—in that each loop in $\widehat{P}$ has a MM contained in M, whilst respecting the constraints imposed by the MMs of the other loops so-constrained too.

The applications afforded by TS are all those of Stable Models, which it generalizes, plus those permitting to solve OLONs for model existence, plus those employing OLONs for productively obtaining problem solutions, not just filtering them (like Integrity Constraints).

1. Introduction and Motivation

The semantics of Stable Models (SM) [Gel88] is a cornerstone for some of the most important results in logic programming of the past three decades, providing increased logic programming declarativity and a new paradigm for program evaluation. When needing to know the 2-valued truth-value of all literals in a normal logic program (NLP) for the problem being solved, the solution is to produce complete models. In such cases, tools like *SModels* [Syr01] or *DLV* [Leo02] may be adequate enough, as they can indeed compute finite complete models according to the SM semantics and its extensions to Answer Sets [Lif92] and Disjunction. However, lack of some important properties of the base SM semantics, like relevance, cumulativity and guarantee of model existence—enjoyed by, say, Well-Founded Semantics [Gel91] (WFS)—somewhat reduces its applicability and practical ease of use when complete models are unnecessary, and top-down querying (*à la* Prolog) would be sufficient. In addition, abduction by need top-down querying is not an option with SM,

Key words and phrases: Normal Logic Programs, Relevance, Cumulativity, Stable Models, Well-Founded Semantics, Program Remainder.

Technical Communications of the 26th International Conference on Logic Programming, Edinburgh, July, 2010
Editors: Manuel Hermenegildo, Torsten Schaub
LIPIcs - Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/LIPIcs.ICLP.2010.134

creating encumbrance in required pre- and post-processing, because needless full abductive models are generated. The user should not pay the price of computing whole models, nor that of generating all possible abductions and then filtering irrelevant ones, when not needed. Finally, one would like to have available a semantics for that provides a model for every NLP.

WFS in turn does not produce 2-valued models though these are often desired, nor does it guarantee 2-valued model existence.

To overcome these limitations, we present the Tight Semantics (TS), a new 2-valued semantics for NLPs which guarantees model existence; preserves the models of SM; enjoys relevance and cumulativity; and complies with the WFM. TS also deals with infinite chains [Fag94], proffering an alternative to SM-based Answer-Set Programming.

TS supersedes our previous RSM semantics [Per05], which we have recently found wanting in capturing our intuitively desired models in some examples, and because TS relies on a clearer, simpler way of tackling the difficult problem of assigning a semantics to every NLP while affording the aforementioned properties, via adapting better known formal LP methods than RSM's *reductio ad absurdum* stance.

A TM of an NLP P is any minimal model (MM) M of P that further satisfies $\widehat{P}$—the program remainder of P—in that each loop in $\widehat{P}$ has a MM contained in M, whilst respecting the constraints imposed by the MMs of the other loops so-constrained too.

A couple of examples bring out the need for a semantics supplying all NLPs with models, and permitting models otherwise eliminated by Odd Loops Over default Negation (OLONs):

Example 1.1. Jurisprudential reasoning. A murder suspect not preventively detained is likely to destroy evidence, and in that case the suspect shall be preventively detained:

$$\begin{array}{lcl} likely\_destroy\_evidence(suspect) & \leftarrow & not\ preventive\_detain(suspect) \\ preventive\_detain(suspect) & \leftarrow & likely\_destroy\_evidence(suspect) \end{array}$$

There is no SM, and a single $TM = \{preventive\_detain(suspect)\}$. This jurisprudential reasoning is carried out without need for a *suspect* to exist now. Should we wish, TS's cumulativity allows adding the model literal as a fact.

Example 1.2. A joint vacation problem. Three friends are planning a joint vacation. First friend says "I want to go to the mountains, but if that's not possible then I'd rather go to the beach". The second friend says "I want to go traveling, but if that's not possible then I'd rather go to the mountains". The third friend says "I want to go to the beach, but if that's not possible then I'd rather go traveling". However, traveling is only possible if the passports are OK. They are OK if they are not expired, and they are expired if they are not OK. We code this information as the NLP:

$$\begin{array}{lcl} beach & \leftarrow & not\ mountain \\ mountain & \leftarrow & not\ travel \\ travel & \leftarrow & not\ beach, passport\_ok \\ passport\_ok & \leftarrow & not\ expired\_passport \\ expired\_passport & \leftarrow & not\ passport\_ok \end{array}$$

The first three rules contain an odd loop over default negation through *beach*, *mountain*, and *travel*; and the rules for *passport_ok* and *expired_passport* form an even loop over default negation. Henceforth we will abbreviate the atoms' names. This program has a single SM: $\{e\_p, m\}$. But looking at the rules relevant for p_ok we find no irrefutable reason to assume e_p to be true. TS allows p_ok to be true, yielding three other models besides the SM: $TM_1 = \{b, m, p\_ok\}$, $TM_2 = \{b, t, p\_ok\}$, and $TM_3 = \{t, m, p\_ok\}$.

The even loop has two minimal models: $\{p\_ok\}$ and $\{e\_p\}$. Assuming the first MM, the odd loop has three MMs corresponding to TM_1, TM_2, and TM_3 above. Assuming the second MM (where e_p is true), the OLON has only one MM: the SM mentioned above $\{e\_p, m\}$, also a TM.

The applications afforded by TS are all those of SM, plus those requiring solving OLONs for model existence, and those where OLONs are employed for the production of solutions, not just used as Integrity Constraints (ICs). Model existence is essential in applications where knowledge sources are diverse (like in the semantic web), and where the bringing together of such knowledge (automatically or not) can give rise to OLONs that would otherwise prevent the resulting program from having a semantics, thereby brusquely terminating the application. A similar situation can be brought about by self-, mutual- and external updating of programs, where unforeseen OLONs would stop short an ongoing process. Coding of ICs via odd loops, commonly found in the literature, can readily be transposed to IC coding in TS, as explained in the sequel.

Paper structure. After background notation and definitions, we usher in the desiderata for TS, and only then formally define TS, exhibit examples, and prove its properties. Conclusions, and future work close the paper.

2. Background Notation and Definitions

Definition 2.1. Normal Logic Program. A Normal Logic Program (NLP) P is a (possibly infinite) set of logic rules, each of the form $H \leftarrow B_1, \ldots, B_n, not\ C_1, \ldots, not\ C_m$ where H, the B_i and the C_j are atoms, and each rule stands for all its ground instances. H is the head of the rule, denoted by $head(r)$, and $body(r)$ denotes set $\{B_1, \ldots, B_n, not\ C_1, \ldots, not\ C_m\}$ of all the literals in the body of r. $heads(P)$ denotes $\{head(r) : r \in P\}$. Throughout, '*not* ' signals default negation. Abusing notation, we write $not\ S$ to denote $\{not\ s : s \in S\}$. If the body of a rule is empty, we say its head is a fact and may write the rule just as H.

Throughout too, we consider MMs of programs, and write $MM_P(M)$ to denote M is a minimal model of P. When both $MM_P(M)$ and $M \subseteq heads(P)$ hold, then M_N denotes the union of M with the negations of heads of P absent in M; i.e., $M_N = M \cup not\ (heads(P) \backslash M)$. We dub M_N a *completed minimal model* of P.

Definition 2.2. Rule dependencies. Given an NLP P build a dependency graph $G(P)$ such that the rules of P are the nodes of $G(P)$, and there is an arc, labeled "positive", from a node r_2 to a node r_1 if $head(r_2)$ appears in the body of r_1; or labeled "negative" if $not\ head(r_2)$ appears in the body of r_1.

We say a rule r_1 directly depends on r_2 (written as $r_1 \leftarrow r_2$) iff there is a direct arc in $G(P)$ from r_2 to r_1. By transitive closure we say r_1 depends on r_2 ($r_1 \twoheadleftarrow r_2$) iff there is a path in $G(P)$ from r_2 to r_1.

Dependencies through default negation play a major role in the sequel and so we also need to define the following: we say a rule r_1 directly depends negatively on r_2 (written as $r_1 \leftarrow -r_2$) iff $not\ head(r_2)$ appears in the body of r_1. By transitive closure we say r_1 depends negatively on r_2 ($r_1 \twoheadleftarrow -r_2$) iff r_1 directly depends negatively on r_2 or r_1 depends on some r_3 which directly depends negatively on r_2.

Definition 2.3. $Rel_P(a)$ — **Relevant part of NLP** P **for the positive literal** a**.** The Relevant part of a NLP P for some positive literal a, $Rel_P(a)$ is defined as

$$Rel_P(a) = \bigcup\{r, r' \in P : r \twoheadleftarrow r' \wedge head(r) = a\}$$

Intuitively, $Rel_P(a)$ is just the set of rules with head a and the rules in the call-graph for a.

Definition 2.4. Loop in P**.** We say a subset P_L of rules of P is a *loop* iff for every two rules r_1 and r_2 in P_L there is a path from r_1 to r_2 in $G(P)$ and vice-versa. I.e., $\forall_{r_1,r_2 \in P_L} r_1 \twoheadleftarrow r_2 \wedge r_2 \twoheadleftarrow r_1$. We write $Loop(P_L)$ to denote that P_L is a *Loop*.

Definition 2.5. Program Remainder [Bra01]**.** The program remainder $\widehat{P}$ is guaranteed to exist for every NLP, and is computed by applying to P the *positive reduction* (which deletes the *not* b from the bodies of rules where b has no rules), the *negative reduction* (which deletes rules that depend on *not* a where a is a fact), the *success* (which deletes facts from the bodies of rules), and the *failure* (which deletes rules that depend on atoms without rules) transformations, and then eliminating also the *unfounded sets* [Gel91] via a *loop detection* transformation. The *loop detection* is computationally equivalent to finding the strongly connected components [Tar72] in the $G(P)$ graph, as per definition 2.2, and is known to be of polynomial time complexity.

Definition 2.6. Program Division. Let P be an NLP and $I \subseteq heads(P) \cup not\ heads(P)$ a consistent interpretation of P. $P : I$ denotes the subset of P remaining after performing this sequence of steps:

(1) delete rules with *not* a in the body where $a \in I$ – similar to *negative reduction*
(2) delete all a in the bodies of rules where $a \in I$ – similar to *success*
(3) delete all *not* a in the bodies of rules where *not* $a \in I$

The rationale behind program division is to obtain the subset of P remaining after considering all literals in I true. Step 1 eliminates the rules of P which are already satisfied (in a classical way) by the literals in I. Step 2 is similar to *success* but deletes all positive literals a from the bodies of rules where $a \in I$. Step 3 is a negative counterpart of step 2; one could dub it *negative success*. Thus, steps 2 and 3 are slightly more credulous that the original *success*.

3. Desiderata

Intuitively desired semantics. Usually, both the default negation *not* and the $\leftarrow$ in rules of Logic Programs reflect some asymmetry in the intended MMs, e.g., in a program with just the rule $a \leftarrow not\ b$, although it has two MMs: $\{a\}$, and $\{b\}$, the only intended one is $\{a\}$. This is afforded by the syntactic asymmetry of the rule, reflected in the one-way direction of the $\leftarrow$, coupled with the intended semantics of default negation. Thus, a fair principle underlying the rationale of a reasonable semantics would be to accept an atom in a model only if there exist rules in a program, at least one, with it as head. This principle rejects $\{b\}$ as a model of program $a \leftarrow not\ b$.

When rules form loops, the syntactic asymmetry disappears and, as far as the loop only is concerned, MMs can reflect the intended semantics of the loop. That is the case, e.g., when we have just the rules $a \leftarrow not\ b$ and $b \leftarrow not\ a$; both $\{a\}$ and $\{b\}$ are the intended

models. However, loops may also depend on other literals with which they form no loop. Those asymmetric dependencies should have the same semantics as the single $a \leftarrow not\ b$ rule case described previously.

So, on the one side, asymmetric dependencies should have the semantics of a single $a \leftarrow not\ b$ rule; and the symmetric dependencies (of *any* loop) should subscribe to the same MMs semantics as the $a \leftarrow not\ b$ and $b \leftarrow not\ a$ set of rules. Intuitively, a good semantics should cater for both the symmetric and asymmetric dependencies as described.

Desirable formal properties. By design, our TS benefits from number of desirable properties of LP semantics [Dix95], namely: guarantee of model existence; relevance; and cumulativity. We recapitulate them here for self-containment. Guarantee of model existence ensures all programs have a semantics. Relevance permits simple (object-level) top-down querying about truth of a query in some model (like in Prolog) without requiring production of a whole model, just the part of it supporting the call-graph rooted on the query. Formally:

Definition 3.1. Relevance. A semantics Sem for logic programs is said Relevant iff for every program P, $a \in Sem(P) \Leftrightarrow a \in Sem(Rel_P(a))$.

Relevance ensures any partial model supporting the query's truth can always be extended to a complete model; relevance is of the essence to abduction by need, in that only abducibles in the call-graph need be considered for abduction.

Cumulativity signifies atoms true in the semantics can be added as facts without thereby changing it; thus, lemmas can be stored. Formally:

Definition 3.2. Cumulativity. A semantics Sem is Cumulative iff the semantics of P remains unchanged when any atom *true* in the semantics is added to P as a fact:

$Cumulative(Sem) \Leftrightarrow \forall_P \forall_{a,b} \_ a \in Sem(P) \wedge b \in Sem(P) \Rightarrow a \in Sem(P \cup \{b\})$

Neither of these three properties are enjoyed by SMs, the *de facto* standard semantics for NLPs. The core reason SM semantics fails to guarantee model existence for every NLP is that the stability condition it imposes on models is impossible to be complied with by OLONs.

Example 3.3. Stable Models semantics misses Relevance and Cumulativity.

$$\begin{array}{ll} c \leftarrow not\ c & c \leftarrow not\ a \\ a \leftarrow not\ b & b \leftarrow not\ a \end{array}$$

This program's unique SM is $\{b, c\}$. However, $P \cup \{c\}$ has two SMs $\{a, c\}$, and $\{b, c\}$ rendering b no longer *true* in the SM semantics, which is the intersection of its models. SM semantics lacks Cumulativity. Also, though b is *true* in P according to SM semantics, b is not *true* in $Rel_P(b) = \{a \leftarrow not\ b;\ \ b \leftarrow not\ a\}$, shows SM semantics lacks Relevance.

In fact, the ASP community uses the SM semantics inability to assign a model to OLONs as a means to impose ICs, such as $a \leftarrow not\ a, X$, where the OLON over a prevents X from being *true* in any model.

TS goes beyond the SM standard, not just because in complying with all the above 3 properties, but also in being a model conservative extension of the SMs semantics, in this sense: A semantics is a model conservative extension of another when it provides at least the same models as the latter, for programs where the latter's are defined, and further provides semantics to programs for which the latter's are not defined. Another way of couching this is: new desired models are provided which the semantics being extended

was failing to produce, but all the latter's produced ones are nevertheless provided by the model-conservative extension.

While encompassing the above properties, TS still respects the Well-Founded Model (WFM) like SM does: every TS model complies with the true and the false atoms in the WFM of a program. Formally:

Definition 3.4. Well-Founded Model of a Normal Logic Program P. Following [Bra01], the *true* atoms of the WFM of P (the irrefutably *true* atoms of P) are the facts of $\widehat{P}$, the *remainder* of P (their definition 5.17). Moreover, the *true* or *undefined* literals of P are just the heads of rules of $\widehat{P}$; and the computation of $\widehat{P}$ can be done in polynomial time. Thus, we shall write $WFM^{+}(P)$ to denote the set of facts of $\widehat{P}$, and $WFM^{+u}(P)$ to denote the set of heads of rules of $\widehat{P}$. Also, since the *false* atoms in the WFM of P are just the atoms of P with no rules in $\widehat{P}$, we write $WFM^{-}(P)$ to denote those false atoms.

Definition 3.5. Interpretation M of P respects the WFM of P. An interpretation M respects the WFM of P iff M contains the set of all the *true* atoms of the WFM of P, and it contains no *false* atoms of the WFM of P. Formally:

$$Respect WFM_P(M) \Leftrightarrow WFM^{+}(P) \subseteq M \subseteq WFM^{+u}(P)$$

TS's WFM compliance, besides keeping with SM's compliance (i.e. the WFM approximates the SM), is important to TS for a specific implementation reason too. Since WFS enjoys relevance and polynomial complexity, one can use it to obtain top-down—in present day tabled implementations—the residual or remainder program that expresses the WFM, and then apply TS to garner its 2-valued models, foregoing the need to generate complete models.

For program $a \leftarrow not\ a$, the only Tight Model (TM) is $\{a\}$. In the TS, OLONs are not ICs. ICs are enforced employing rules for the special atom $falsum$, of the form $falsum \leftarrow X$, where X is the body of the IC one wishes to prevent being true. This does not preclude $falsum$ from figuring in some models. From a theoretical standpoint it means the TS semantics does not *a priori* include a built-in IC compliance mechanism. ICs can be dealt with in two ways, either by (1) a syntactic post-processing step, as a model "test stage" after their "generate stage"; or by (2) embedding IC compliance in the query-driven computation, whereby the user conjoins query goals with $not\ falsum$. If inconsistency examination is desired, like in case (1), models including $falsum$ can be discarded *a posteriori*. Thus, TS clearly separates OLON semantics from IC compliance, and frees OLONs for a wider knowledge representation usage.

4. Tight Semantics

The rationale behind tightness follows the intuitively desired semantics principles described in section 3. On the one side any TM M is necessarily an MM of $\widehat{P}$ which guarantees that no atoms with no rules are in M. This is in accordance with the principle of intuitively desired semantics for asymmetric dependencies, and is also what guarantees that TMs respect the WFM, as proved in the sequel. On the other side, implementing the intuitively desired semantics for symmetric dependencies, the TS imposes each TM to have the internal loop congruency of tightness: the semantics for each loop is its MMs, as long as a chosen MM for it is compatible (via program division) with the model for the whole program, while the rest $M_{\overline{L}}$ of the original model M is itself Tight.

Definition 4.1. Tight Model. Let P be an NLP, and M a minimal model of $\widehat{P}$, such that M_N is a completed minimal model of P. Let $\widehat{P}_L$ denote a $Loop(\widehat{P}_L)$ strictly contained in $\widehat{P}$; and given $MM_{\widehat{P}_L}(M_L)$, let $M_{\overline{L}}$ denote $(M \setminus M_L) \cup \{M_L \cap heads(P : M_{L_N})\}$. We say M is tight in P — $Tight_P(M)$ — iff

$$\exists \widehat{P}_L \Rightarrow \exists M_L : M_{L_N} \subseteq M_N \wedge Tight_{P:M_{L_N}}(M_{\overline{L}})$$

The Tight Semantics of P — $TS(P)$ — is the intersection of all its Tight Models.

Example 4.2. Mixed loops 1. Let P be

$$\begin{array}{lll} a \leftarrow k & k \leftarrow not\ t & t \leftarrow a, b \\ a \leftarrow not\ b & b \leftarrow not\ a & \end{array}$$

$\widehat{P}$ coincides with P, so its MMs are $M_1 = \{a, k\}$ with $M_{1_N} = \{a, not\ b, k, not\ t\}$; $M_2 = \{a, t\}$ with $M_{2_N} = \{a, not\ b, not\ k, t\}$; and $M_3 = \{b, t\}$ with $M_{3_N} = \{not\ a, b, not\ k, t\}$. Of these, only M_{1_N} and M_{3_N} are Tight. M_1 in particular is also an SM. To see that M_{2_N} is not Tight notice that there are three loops in P: $P_{L_1} = \{a \leftarrow not\ b; b \leftarrow not\ a\}$, $P_{L_2} = \{a \leftarrow k; k \leftarrow not\ t; t \leftarrow a, b\}$, $P_{L_3} = \{a \leftarrow k; k \leftarrow not\ t; t \leftarrow a, b; b \leftarrow not\ a\}$. The MMs of P_{L1} are $M_{L1_1} = \{a\}$ with $M_{L1_{1N}} = \{a, not\ b\}$, and $M_{L1_2} = \{b\}$ with $M_{L1_{2N}} = \{not\ a, b\}$. Dividing P by $M_{L1_{1N}}$ we get $P : M_{L1_{1N}} = \{a \leftarrow k; k \leftarrow not\ t; t \leftarrow b\}$. $M_{\overline{L1}_1}$ is now $(\{a, t\} \setminus \{a\}) \cup \{\{a\} \cap heads(\{a \leftarrow k; k \leftarrow not\ t; t \leftarrow b\})\} = \{t\} \cup \{a\} = \{a, t\}$. But $\{a, t\}$ is not Tight in $P : M_{L1_{1N}}$ since it is not even an MM of it.

Example 4.3. Difference between TS and RSM semantics. Let P be

$$\begin{array}{l} a \leftarrow not\ b, c \\ b \leftarrow not\ c, not\ a \\ c \leftarrow not\ a, b \end{array}$$

TS accepts both $M_1 = \{a\}$ and $M_2 = \{b, c\}$ as TMs, whereas the RSM semantics [Per05] only accepts M_1. Neither are SMs.

Example 4.4. Mixed loops 2. Let P be

$$\begin{array}{l} a \leftarrow not\ b \\ b \leftarrow not\ c, e \\ c \leftarrow not\ a \\ e \leftarrow not\ e, a \end{array}$$

In this case, TS, like the RSM semantics, accepts all minimal models: $M_1 = \{a, b, e\}$, $M_2 = \{a, c, e\}$, and $M_3 = \{b, c\}$.

Example 4.5. Quasi-Stratified Program. Let P be

$$\begin{array}{l} d \leftarrow not\ c \\ c \leftarrow not\ b \\ b \leftarrow not\ a \\ a \leftarrow not\ a \end{array}$$

The unique TS is $\{a, c\}$, and there are no SMs. In this case it is quite easy to see how the Tightness works: $\{a\}$ is necessarily the unique MM of $a \leftarrow not\ a$. Dividing the whole program by $\{a\}$ we get $\{d \leftarrow not\ c; c \leftarrow not\ b\}$. Its unique TM is $\{c\}$ providing the global model $\{a, c\}$ together with the $\{a\}$ model for $a \leftarrow not\ a$.

5. Properties of the Tight Semantics

Forthwith, we prove some properties of TS, namely: guarantee of model existence, relevance, cumulativity, model-conservative extension of SMs, and respect for the Well-Founded Model. The definitions involved are to be found in section 3.

Theorem 5.1. ***Existence.*** *Every Normal Logic Program has a Tight Model.*

Proof. Let P be an NLP. $\widehat{P}$ is guaranteed to exist. So are MMs of any given NLP, in particular, for $\widehat{P}$ too. If $\widehat{P}$ has no *loops*, then every MM of $\widehat{P}$ is trivially Tight. In particular, if $\widehat{P}$ has no *loops* it means $\widehat{P}$ is stratified and the unique TM is its unique MM.

Consider now $\widehat{P}$ has *loops*, and that P_L is any such loop in $\widehat{P}$. Assume $\widehat{P}$ has no TMs. In this case, for every P_L there is no $M_L \subseteq M_{L_N}$ such that $Tight_{P:M_{L_N}}(M_{\overline{L}})$ holds. Since for every P_L it is always possible to compute an M_L and its respective M_{L_N}, the tightness condition must fail because $Tight_{P:M_{L_N}}(M_{\overline{L}})$ fails. But any P_L which does not depend on any other rule outside P_L is unaffected by any program division $P : M_{L'_N}$ where $M_{L'}$ is an MM of some other $P_{L'}$. Hence the hypothetical failure of tightness in holding of $M_{\overline{L}}$ in $P : M_{L_N}$ must be because all M_Ls of all P_L are not Tight in some $P_{L'''} = P_{L''} \cup P_L$ such that P_L depends on $P_{L''}$ and vice-versa. I.e., for all M_L of P_L, $Tight_{P_{L'''}:M_{L_N}}(M_{\overline{L}})$ must not hold. Since it is always possible to compute $M_{\overline{L}} = (M \setminus M_L) \cup \{M_L \cap heads(P_{L'''} : M_{L_N})\}$ it must be the case that for every $M_{L''}$ of each $P_{L''}$, $M_{L''} \cup M_L$ is not a consistent MM of $P_{L''} \cup P_L$, which is an absurdity because consistent MMs of any given program are always guaranteed to exist. ∎

Theorem 5.2. ***Relevance of Tight Semantics.*** *The Tight Semantics is relevant.*

Proof. According to definition 3.1 a semantics Sem is relevant iff $a \in Sem(P) \Leftrightarrow a \in Sem(Rel_P(a))$ for all atoms a. Since the TS of a program P — $TS(P)$ — is the intersection of all its TMs, relevance becomes $a \in TS(P) \Leftrightarrow a \in TS(Rel_P(a))$ for TS.

$\Rightarrow$: We assume $a \in TS(P)$, so we can take any M such that $TM_P(M)$ holds, and conclude $a \in M$. Assuming, by contradiction, that $a \notin TS(Rel_P(a))$ then there is at least one TM of $Rel_P(a)$ where a is *false*. Let us write M_a to denote such TM of $Rel_P(a)$ where $a \notin M_a$. Since all TMs of P are MMs of $\widehat{P}$ we have two possibilities: 1) a is a fact in $\widehat{P}$ — in this case there is a rule (a fact) for a and hence this fact rule is in $Rel_P(a)$ forcing $a \in M_a$; 2) a is not a fact in $\widehat{P}$ — by definition of TM a can be in M only if a is the head of a rule and there is some MM $M_L \subseteq M$ of a loop $P_L \subseteq P$ such that $a \in M_L$. Since a must be the head of a rule in loop, that loop is, by definition, in $Rel_P(a)$. Since M is Tight in P, by definition so must be each and every of its subset MMs of loops; i.e., $a \in M_a$.

$\Leftarrow$: Assume $a \in TS(Rel_P(a))$. Take the whole $P \supseteq Rel_P(a)$. Again, a will be in every TM of P because a is in all TMs of $Rel_P(a)$, and, by definition, every TM of P always contains one TM of $Rel_P(a)$. ∎

Theorem 5.3. ***Cumulativity of Tight Semantics.*** *The Tight Semantics is cumulative.*

Proof. By definition 4.1, the semantics of a program P is the intersection of its TMs. So, $a \in TS(P) \Leftrightarrow \forall_{TM_P(M)} a \in M$. For the TS semantics cumulativity becomes expressed by $\forall_{a,b}(a \in TS(P) \wedge b \in TS(P)) \Rightarrow a \in TS(P \cup \{b\})$

Let us assume $a \in TS(P) \wedge b \in TS(P)$. Since both $a \in TS(P)$ and $b \in TS(P)$, we know that whichever TM M and $M_L \subseteq M$ such that $a \in M_L$, $b \in TS(P : M_{L_N})$ holds; and in that case $P : M_{L_N} = (P \cup \{a\}) : M_{L_N}$. Hence, $b \in TS(P \cup \{a\})$. ∎

Theorem 5.4. ***Stable Models Extension.*** *Any Stable Model is a TM of P.*

Proof. Assume M is a SM of P. Then $M = least(P/M)$ where the division P/M deletes all rules with *not* a in the body where $a \in M$, and then deletes all remaining *not* b from the bodies of rules. The program division $P : M$ performs exactly the same step as the P/M one, but then only deletes the *not* b such that *not* $b \in M_N$. Moreover, the P/M division is performed using the whole M at once, whilst the $P : M$ considers not the whole M but only partial M_{L_N}s of M. Tightness requires consistency amongst the several individual M_{L_N}s whilst the $M = least(P/M)$ stability condition requires consistency throughout the whole M. We can thus say the division P/M performs all the steps the $P : M$ division does, and then some. In this sense the $M = least(P/M)$ stability condition demands from M all that Tightness does and even more. Hence, a model passing the stability condition is bound to be also a TM. ∎

Theorem 5.5. ***Tight Semantics respects the Well-Founded Model.*** *Every Tight Model of P respects the Well-Founded Model of P — $\forall_{M:TM_P(M)} RespectWFM_P(M)$.*

Proof. Take any TM M of P. Since all TMs are MMs of $\widehat{P}$, M must contain all the facts of $\widehat{P}$, i.e., $M \supseteq WFM^{+}(P)$. Also MMs of $\widehat{P}$ are bound to be a subset of the heads of rules of $\widehat{P}$, hence $M \subseteq WFM^{+u}(P)$. ∎

Due to lack of space, the complexity analysis of this semantics is left out of this paper. Nonetheless, a brief note is due. Tight Model existence is guaranteed for every NLP, whereas finding if there are any SMs for an NLP is NP-complete. Since TS enjoys relevance, the computational scope of Brave Reasoning can be restricted to $Rel_P(a)$ only, instead of the whole P. Nonetheless, we conjecture that Brave reasoning — finding if there is any model of the program where some atom a is true — is a Σ^2_P-hard task. This is so because each relevant branch in the call-graph can be a loop. Traversing the entire call-graph is in itself an NP-complete task. For each loop, the TS requires the computation of a minimal model — another NP-complete task. Hence the conjectured Σ^2_P-hardness. Still, from a practical standpoint, having to traverse only the relevant call-graph for brave reasoning, instead of considering the whole program, can have a significant impact in the performance of concrete applications. By the same token, cautious reasoning (finding out if some atom a is in all models) in the TS should have the complementary complexity of brave reasoning: co-Σ^2_P-complete.

One common objection to these kind of semantics concerns the notion of support. Tight models are not supported, considering the classical notion of support. However, we abide by a more general notion of support: an atom is supported iff there is at least one rule with it as head and all the literals in the body of the rule which do not depend on the head are also true. This *loop support* is a generalization of the classical *support*. This ensures the truth assignment to atoms in, say, a loop L_2 which depends asymmetrically on loop L_1, is consistent with the truth assignments in loop L_1 and that these take precedence over L_2 in their truth labeling. As a consequence of the loop support requirement, Tight model comply with the WFM of the loops they asymmetrically depend on.

6. Conclusions, Future and Ongoing Topics, and Similar Work

Having defined a more general 2-valued semantics for LPs much remains in store, and to be explored and reported, in the way of properties, complexity, comparisons, implementation, and applications. We hope the concepts and techniques newly introduced here might be adopted by other logic programming semantics approaches and systems.

We defined TS, a semantics for *all* NLPs complying with the express requirements of: 2-valued semantics, preserving the models of SM, guarantee of model existence (even in face of odd loops over negation or infinite chains), relevance, cumulativity, and WFM respect.

Relevancy condones top-down querying and avoids the need to compute whole models. It also permits abduction by need, avoiding much useless consideration of irrelevant abducibles.

That TS includes the SM semantics and that it always exists and admits top-down querying is a novelty making us look anew at 2-valued semantics use in KRR, contrasting its use to other semantics employed heretofore for KRR, even though SM has already been compared often enough [Bar03].

A current avenue of further work already being taken follows the line of thought we laid out in [Per09] by partitioning an NLP into layers, a generalization of strata, to further segment the program and thus reduce the combinatorics of the Tightness test. Although not reducing the theoretical complexity class of the Tightness test, in practical implementations the syntactical partitioning of layering can have a substantial impact on performance.

References

[Bar03] C. Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2003.

[Bra01] S. Brass, J. Dix, B. Freitag, and U. Zukowski. Transformation-based bottom-up computation of the well-founded model. *TPLP*, 1(5):497–538, 2001.

[Dix95] J. Dix. A Classification-Theory of Semantics of Normal Logic Programs: I, II. *Fundamenta Informaticae*, XXII(3):227–255, 257–288, 1995.

[Fag94] F. Fages. Consistency of Clark's completion and existence of stable models. *Methods of Logic in Computer Science*, 1:51–60, 1994.

[Gel88] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *ICLP/SLP*, pp. 1070–1080. MIT Press, 1988.

[Gel91] A. Van Gelder, K. A. Ross, and J. S. Schlipf. The well-founded semantics for general logic programs. *J. of ACM*, 38(3):620–650, 1991.

[Leo02] 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.

[Lif92] Vladimir Lifschitz and Thomas Y. C. Woo. Answer sets in general nonmonotonic reasoning (preliminary report). In *KR*, pp. 603–614. 1992.

[Per05] L. M. Pereira and A. M. Pinto. Revised stable models - a semantics for logic programs. In G. Dias et al. (ed.), *Progress in AI*, *LNCS*, vol. 3808, pp. 29–42. Springer, 2005.

[Per09] L. M. Pereira and A. M. Pinto. Layer supported models of logic programs. In E. Erdem, F. Lin, and T. Schaub (eds.), *Procs. 10th LPNMR*, *LNAI*, vol. 5753, pp. 450–456. Springer, 2009.
URL `http://centria.di.fct.unl.pt/$\sim$lmp/publications/online-papers/LSMs.pdf(longversion)`

[Syr01] Tommi Syrjänen and Ilkka Niemelä. The smodels system. In T. Eiter et al. (ed.), *LPNMR 2001*, *LNAI*, vol. 2173. Springer-Verlag, 2001.

[Tar72] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Computing*, 1(2):146–160, 1972.

Technical Communications of the International Conference on Logic Programming, 2010 (Edinburgh), pp. 144–153
http://www.floc-conference.org/ICLP-home.html

FROM RELATIONAL SPECIFICATIONS TO LOGIC PROGRAMS

JOSEPH P. NEAR [1]

[1] Computer Science and Artificial Intelligence Lab
Massachusetts Institute of Technology
32 Vassar St. Cambridge, MA, USA

Abstract. This paper presents a compiler from expressive, relational specifications to logic programs. Specifically, the compiler translates the Imperative Alloy specification language to Prolog. Imperative Alloy is a declarative, relational specification language based on first-order logic and extended with imperative constructs; Alloy specifications are traditionally not executable. In spite of this theoretical limitation, the compiler produces useful prototype implementations for many specifications.

1. Introduction

This paper presents a compiler from declarative, relational specifications to Prolog programs, eliminating the need for manual implementation. I express specifications in Imperative Alloy [28], a language based on the combination of first-order logic with transitive closure and the standard imperative programming constructs. My compiler transforms these specifications to Prolog for execution. Prolog represents an appropriate target language, since it supports nondeterminism and provides a database for storing global relations; the compiler uses these features to simulate Alloy's relational operators, quantifiers, and classical negation.

The existing Alloy Analyzer is designed for the *verification* and *animation* of specifications. My compiler is intended to complement the Analyzer by *executing* specifications. Animators perform their analyses within a fixed universe of predetermined size, while execution engines allow the creation of new objects. In practice, animators typically deal with models containing tens of objects, while execution engines must handle hundreds or thousands. In this case, this increased scalability comes at the cost of analysis: the Alloy Analyzer is designed to check all cases within a small bound, while my compiler executes a single, potentially large, case. In exchange, the compiler provides efficiency: most specifications can be executed fast enough to serve as prototype implementations.

Along with the Alloy Analyzer, my compiler provides end-to-end support for specifying and implementing programs. The Alloy language provides the expressive logic and relational constructs needed to express complex properties of programs and data; the Analyzer supports the animation and verification of program specifications; and the compiler presented in this paper allows for the efficient execution of those specifications.

Key words and phrases: logic programming, specification languages, executable specifications.

Technical Communications of the 26th International Conference on Logic Programming, Edinburgh, July, 2010
Editors: Manuel Hermenegildo, Torsten Schaub
LIPIcs - Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/LIPIcs.ICLP.2010.144

2. The Alloy Language

Alloy [16] is a modeling language based on first-order relational logic with transitive closure. It is designed to be simple but expressive, and to be amenable to automatic analysis. Imperative Alloy [28] adds imperative constructs, including assignment to global relations, sequential composition, and loops. The Alloy Analyzer is a tool for automatic analysis of Alloy models. While this analysis is bounded, it does allow for incremental, agile development of models; and the *small-scope hypothesis* [4]—which claims that most inconsistent models have counterexamples within small bounds—means that modelers may have high confidence in the results. The sacrifice of completeness in favor of automation is in line with the *lightweight formal methods* philosophy [17].

Alloy's universe is made up of uninterpreted atoms, each of which belongs to one of the disjoint sets defined using *signatures*. Signatures also may define global relations in the form of fields. As an example, consider a filesystem made up of file and directory nodes.

```
sig Data {}
abstract sig INode {}
sig DirNode extends INode { files: Name →INode }
sig FileNode extends INode { data: dynamic Data }
```

Directory and file nodes extend nodes, which are abstract, meaning that the file and directory nodes exhaustively partition the set of nodes. The " files " relation contains 3-tuples of type DirNode→Name→INode, while "data" is a mutable relation of type FileNode→Data. A representation of path names as linked lists of names can be defined similarly.

```
sig Name {}
abstract sig FilePath { name: Name }
sig DirName extends FilePath { dnext: FilePath }
sig FileName extends FilePath {}
```

Given a path name and a filesystem, a logical operation is to navigate through the filesystem to the node corresponding to the path name. I define a single step of this operation as an action in Imperative Alloy, using a singleton signature with mutable fields to hold pointers into the path name and the filesystem, as well as some temporary data.

```
one sig MVar {
    path: dynamic FilePath, current: dynamic INode, mdata: dynamic Data
}
action navigate {
    MVar.path := MVar.path.dnext;
    MVar.current := (MVar.path.name).(MVar.current.files) }
```

In defining navigation, I have used both imperative (field update and sequencing) and declarative (Alloy's generalized relational join) features of the language. Now, I can define reading from and writing to the filesystem by repeating "navigate" until the file node is reached and then either reading to or writing from the temporary storage.

```
action read {
    loop { navigate[] } && after MVar.current in FileNode;
    MVar.mdata := MVar.current.data
}
```

```
action write {
    loop { navigate[] } && after MVar.current in FileNode;
    let  file  = MVar.current | file.data := MVar.mdata
}
```

Given both actions, the user might wish to verify that writing to the filesystem and then reading from it produces the written data. I can define this property as an assertion to be checked by the Alloy Analyzer. In addition to the first-order quantifiers "**some**" and "**no**," I use the temporal quantifier "**always**" to indicate that the property must hold in all possible executions of the action. I use "**before**" and "**after**" to represent pre- and post-conditions on the action. The overall property states that for all starting nodes in the filesystem, it is always the case that if I begin at that node and write to the filesystem, remember the written data, reset the current node, and read from the filesystem, then the read data will match the remembered data.

```
one sig Temp { tdata: dynamic Data }
assert readMatchesPriorWrite {
  all n: INode |
    always |
      before (MVar.current = n && no f: FileNode | f.data = MVar.mdata) &&
      write; Temp.tdata := MVar.mdata, MVar.current := n;
      read ⇒ after (Temp.tdata = MVar.mdata) }
```

For more information on Alloy, refer to [16]; for more on Imperative Alloy, see [28].

3. Compiling Alloy Specifications

Any execution strategy for the Alloy language must allow relations as first-class values, nondeterminism, imperative constructs, and both relational and logical operators. I begin with Alloy's global relations, whose representation as dynamic predicates in Prolog I demonstrate using the filesystem from Section 2. For each signature, I generate a unary predicate representing membership in the signature and a predicate for each relation. I represent the existence of a single atom in each singleton signature by generating a fact.

```
:- dynamic sigName/1, sigFilePath/1, sigDirName/1, sigFileName/1,
           sigINode/1, sigDirNode/1, sigRootNode/1, sigFileNode/1,
           sigData/1, sigMVar/1, data/3, path/3, current/3,
           mdata/3, name/2, dnext/2, files/3.
sigRootNode(gensym62).
sigMVar(gensym63).
```

Relational values in Alloy may be thought of as sets of tuples. In Prolog, I can represent each tuple using a term; to represent the set of tuples in a relation, an expression may yield multiple instantiations of that term—one for each tuple in the Alloy relation. For example, I compile the expression representing the next element in a path as follows.

MVar.path.dnext → `sigMVar(MVar), path(T0, MVar, Path), dnext(Path, 0)`

The Prolog expression yields values by instantiating a member of the "MVar" signature, looking up a `Path` in the field of that "MVar," and then instantiating the free variable `0`

based on the next element of that path. The other relational operators can similarly be compiled into expressions involving Prolog's logical connectives.

I compile formulas involving relations to comparisons between their possible instantiations: $r_1 \subseteq r_2$, for example, assuming that r_1 and r_2 are binary relations, becomes `forall(r1(A,B), r2(A,B))`. Another option is to enumerate each relation's tuples explicitly (*e.g.* in a list or in the global database); this strategy may make lookup faster, but it forces the enumeration of the relational value of each subexpression, making the complex use of relational operators expensive.

Imperative Alloy also differs from Prolog in its imperative constructs: field update, sequential composition, and loops are notions built into the language. Sequencing and looping are easy to simulate in Prolog, and side effects can be expressed using `assert` and `retract`. Since Imperative Alloy's semantics call for side effects to interact well with nondeterminism, I have defined `assertl` to assert a list of terms, and then retract them upon backtracking, allowing side effects to be undone.

The combination of lazy evaluation and side effects means that a relation's value may depend on the value another relation had in the past. My prototype implementation therefore keeps track of the history of each global relation by adding an argument to each relation representing a *time-step*; the compiler passes the current time-step to called relations to instantiate the call's other arguments with the relation's value at that time-step. The "addr" relation, for example, has the type Time→Name→Addr, so I compile a reference to it to `addr(T, N, A)`, placing the time argument in the first position because most Prolog systems index on that argument. Parts of a relation's history upon which no "current" values depend may be eliminated in a process analogous to garbage collection.

This infrastructure makes compiling field assignments straightforward. I translate an update of the form $o.f := e$ by compiling e at the current time-step, then using `assertl` to update the global relation f. The first two arguments to f in the update are the next time-step and o; the remaining arguments are the free variables of the result of compiling e, and the body is the expression to which e compiles. I compile the full action for navigating the filesystem, for example, into a Prolog predicate as follows.

action navigate {
  MVar.path := MVar.path.dnext;
  MVar.current := (MVar.path.name).(MVar.current.files)
}
→

```
navigate(T0, T1) :-
  T2 is T0 + 1, sigMVar(Mv),
  assertl([((path(T2, Mv, Path) :- sigMVar(Mv2), path(T0, Mv2, Var3), dnext(Var3, Path))),
           ((data(T2, Var6, Var7) :- data(T0, Var6, Var7))),
           ((current(T2, Var8, Var9) :- current(T0, Var8, Var9))),
           ((mdata(T2, Var10, Var11) :- mdata(T0, Var10, Var11)))]),
  T1 is T2 + 1, sigMVar(Mv3),
  assertl([((current(T1, Mv3, Var22) :-
              sigMVar(Mv4), path(T2, Mv4, Var15), name(Var15, Var21),
              sigMVar(Mv5), current(T2, Mv5, Var20), files(Var20, Var21, Var22))),
           ((data(T1, Var24, Var25) :- data(T2, Var24, Var25))),
           ((path(T1, Var26, Var27) :- path(T2, Var26, Var27))),
           ((mdata(T1, Var28, Var29) :- mdata(T2, Var28, Var29)))]).
```

I compile each update in the navigation action to a call to `assertl` in Prolog and sequence them using conjunction. In both cases, the update itself is the first element in the list passed to `assertl`, and I form it by compiling the expression on the update action's right-hand side, then placing the result in the body of a rule defining the relation specified on the assignment's left-hand side. I also increment the current time-step so that the updated rule will be the sole definition of the relation at that time-step. The other elements passed to `assertl` represent the frame condition: in this new time-step, the other mutable relations do not change, so I generate rules to delegate these relations to their previous definitions.

I compile the action for reading from the filesystem in a similar way, except for its loop and declarative post-condition. I compile loops to nondeterministic repetition of an action, which I implement using the `loop` predicate. The post-condition checks that the current node is a file using the subset operator; in Prolog, this requires checking that the right-hand side of the "**in**" formula succeeds for every possible instantiation of the left-hand side.

action read {
  loop { navigate[] } && **after** MVar.current **in** FileNode;
  MVar.mdata := MVar.current.data
}
→

```
read(T3, T4) :-
  loop(T3, T5, navigate, []),
  forall((sigMVar(Mv), current(T5, Mv, Var33)), sigFileNode(Var33)),
  T4 is T5 + 1, sigMVar(Var39),
  assertl([((mdata(T4, Var39, Var38) :-
              sigMVar(Var35), current(T5, Var35, Var37), data(T5, Var37, Var38))),
           ((data(T4, Var40, Var41) :- data(T5, Var40, Var41))),
           ((path(T4, Var42, Var43) :- path(T5, Var42, Var43))),
           ((current(T4, Var44, Var45) :- current(T5, Var44, Var45)))]).
```

Finally, I compile the action for writing to the filesystem. Except for the "**let**" formula, it is nearly identical to that for reading.

action write {
  loop { navigate[] } && **after** MVar.current **in** FileNode;
  let file = MVar.current | file.data := MVar.mdata
}
→

```
write(T6, T7) :-
  loop(T6, T8, navigate, []),
  forall((sigMVar(Var47), current(T8, Var47, Var49)), sigFileNode(Var49)),
  sigMVar(Var51), current(T8, Var51, File), T7 is T8 + 1,
  assertl([((data(T7, File, Var55) :- sigMVar(Var54), mdata(T8, Var54, Var55))),
           ((path(T7, Var56, Var57) :- path(T8, Var56, Var57))),
           ((current(T7, Var58, Var59) :- current(T8, Var58, Var59))),
           ((mdata(T7, Var60, Var61) :- mdata(T8, Var60, Var61)))]).
```

This collection of predicates represents a simplified model of a filesystem that can be executed by a Prolog system; combined with a tool like FUSE (Filesystem in User Space), it can be used as a prototype implementation to store real data and be tested on an actual system. The user may add features slowly, using the Alloy Analyzer to verify their correctness.

C_E	::	expression $\rightarrow$ time $\rightarrow$ (Prolog expression, [variable])
$C_E(a,t)$ $(a \in vars)$	$\hat{=}$	$(\emptyset, [A])$
$C_E(f,t)$ $(f \in r)$	$\hat{=}$	$(f(A_1, A_2, ..., A_n), [A_1, A_2, ..., A_n])$ where f has arity n; $A_1, ..., A_n$ are fresh variables
$C_E(f,t)$ $(f \in r_d)$	$\hat{=}$	$(f(A_1, A_2, ..., A_n, t), [A_1, A_2, ..., A_n])$ where f has arity n; $A_1, ..., A_n$ are fresh variables
$C_E(e_1 \rightarrow e_2, t)$	$\hat{=}$	$((E_1, E_2), [A_1, ..., A_n, B_1, ..., B_n])$
$C_E(e_1.e_2, t)$	$\hat{=}$	$((A_n = B_1, E_1, E_2), [A_1, ..., A_{n-1}, B_2, ..., B_n])$ where $C_E(e_1, t) = (E_1, [A_1, ..., A_n])$ and $C_E(e_2, t) = (E_2, [B_1, ..., B_n])$
$C_E(e_1 + e_2, t)$	$\hat{=}$	$((A_1 = B_1, ..., A_n = B_n, E_1; A_1 = C_1, ..., A_n = C_n, E_2), [A_1, ..., A_n])$
$C_E(e_1 - e_2, t)$	$\hat{=}$	$((A_1 = B_1, ..., A_n = B_n, E_1, A_1 = C_1, ..., A_n = C_n, \backslash\texttt{+}E_2), [A_1, ..., A_n])$
$C_E(e_1 \& e_2, t)$	$\hat{=}$	$((A_1 = B_1, ..., A_n = B_n, E_1, A_1 = C_1, ..., A_n = C_n, E_2), [A_1, ..., A_n])$ where $C_E(e_1, t) = (E_1, [B_1, ..., B_n])$ and $C_E(e_2, t) = (E_2, [C_1, ..., C_n])$

Figure 1: Rules for Compiling Alloy Expressions into Prolog

4. Implementing the Compiler

My compiler transforms a complete Alloy specification into a Prolog program. In Alloy, sets are represented as unary relations; scalars, then, are singleton sets. For an Alloy expression whose value is an n-ary relation, my compiler produces a Prolog expression with n free variables; each possible instantiation of those free variables represents one tuple of the original relation. My compiler therefore produces a 2-tuple (e, v) containing the compiled Prolog expression e and a list of free variables v. I present the set of compilation rules for expressions in Figure 1; r represents the set of global relations in the original Alloy model, while r_d is the set of dynamic relations.

Two issues make compiling expressions tricky. First, the translation requires a representation of the time-step at which the expression is being evaluated. My implementation represents time-steps using integers; each global relation accepts one of these time-steps as its first argument and instantiates its other arguments to the values of the relation at that time-step. Second, some relational operators (*e.g.* difference) require the use of the cut or negation-as-failure. These impure elements restrict the contexts in which the compilation produces useful programs: the Alloy expression !(i $<$ j), for example, produces the Prolog expression `\+ (I<J)`, which will not correctly instantiate `I` or `J`.

Compiling Alloy formulas is straightforward, since Alloy's logical connectives map directly to those of Prolog. The equality and subset operators are the most interesting: since expressions evaluate to relations, both logical operators must examine *all* instantiations of the expressions' free variables generated by the resulting Prolog expressions. Figure 2 contains the rules for compiling formulas; again, the rules require the time at which the formula is being evaluated. Actions are compiled into formulas sequenced using conjunction. The

C_M	::	formula $\rightarrow$ time $\rightarrow$ Prolog expression
$C_M(e_1 \in e_2, t)$	$\hat{=}$	`forall(`$(E_1), (B_1 = C_1, ..., B_n = C_n, E_2)$`)`

$C_M(e_1 = e_2, t)$	$\hat{=}$	`forall(`$(E_1), (B_1 = C_1, ..., B_n = C_n, E_2)$`),`
		`forall(`$(E_2), (B_1 = C_1, ..., B_n = C_n, E_1)$`)`
		where $C_E(e_1, t) = (E_1, [B_1, ..., B_n])$
		and $C_E(e_2, t) = (E_2, [C_1, ..., C_n])$

$C_M(f_1 \&\& f_2, t)$	$\hat{=}$	$C_M(f_1, t)$ `,` $C_M(f_2, t)$
$C_M(f_1 \,\|\|\, f_2, t)$	$\hat{=}$	$C_M(f_1, t)$ `;` $C_M(f_2, t)$
$C_M(!f, t)$	$\hat{=}$	`\+` $C_M(f, t)$
$C_M(\textbf{all}\ x{:}\text{ite} \mid f, t)$	$\hat{=}$	`forall((`$x = A$`,`E`),` $C_M(f, t)$`)`
$C_M(\textbf{some}\ x{:}\text{ite} \mid f, t)$	$\hat{=}$	$x = A$`,`E`,`$C_M(f, t)$ where $C_E(e, t) = (E, [A])$

Figure 2: Rules for Compiling Alloy Formulas into Prolog

C_A	::	action $\rightarrow$ time $\rightarrow$ time $\rightarrow$ Prolog expression

$C_A(o.f{:=}e, t, t')$	$\hat{=}$	t' `is` $t + 1$`,`
		`assertl((`$f(o, A_1, ..., A_n, t')$ `:-` E`)),`
		`assertl((`$f(O, B_1, ..., B_n, t')$ `:-`
		`dif(`O, o`),` $f(O, B_1, ..., B_n, t)$`))),`
		$\bar{\forall} r : \textit{relations} \mid$ `assertl((`$r(O', C_1, ..., C_k, t')$ `:-`
		$r(O', C_1, ..., C_k, t)$`).`
		where $C_E(e, t) = (E, [A_1, ..., A_n])$

$C_A(a_1;\ a_2, t, t')$	$\hat{=}$	$C_A(a_1, t, T'')$`,` $C_A(a_2, T'', t')$ where T'' is a fresh variable
$C_A(a_1 \&\& a_2, t, t')$	$\hat{=}$	$C_A(a_1, t, t')$ `,` $C_A(a_2, t, t')$
$C_A(a_1 \,\|\|\, a_2, t, t')$	$\hat{=}$	$C_A(a_1, t, t')$ `;` $C_A(a_2, t, t')$
$C_A(action[e_1, ..., e_n], t, t')$	$\hat{=}$	`action(`$e_1, ..., a_n, t, t'$`)`

$C_A(\textbf{loop}\ \{act[e_1,...,e_n]\}, t, t')$	$\hat{=}$	E_1`,...,`E_n`, loop(`$t, t', act, [V_1, ..., V_n]$`)`
		where $C_E(e_i, t) = (E_i, [V_i])$ and
		`loop(T, T, F, Args).`
		`loop(T, Tp, F, Args) :-`
		`append(Args, [T, T1], A),`
		`apply(F, A), loop(T1, Tp, F, Args).`

Figure 3: Rules for Compiling Alloy Actions into Prolog

rule for field assignment updates the global relation f at the object o and time step $t + 1$ with the results of the right-hand side expression e. The next two lines of the rule express the frame condition: first, that the values of the relation f at objects other than o do not change, and second, that the values of the relations not being updated do not change. I use the "meta" quantifier $\bar{\forall}$ to represent quantification over the (static) set of relations in a given specification. Figure 3 contains the complete set of rules for compiling actions.

5. Related Work

As a notation, Imperative Alloy is similar to existing specification languages that support modeling dynamic systems [27, 23, 1, 5, 31, 20, 11, 9]; some of these notations also have associated analysis and animation tools. The novelty of this work is in the combination, using a single notation, of analysis and execution.

Most similar to my own work is an approach that translates Z specifications into Prolog [10], but produces relatively inefficient programs. Similarly, PVS has been translated to LISP [8] for execution, but the translation places restrictions on the PVS language. Squander [29] animates Alloy specifications embedded in Java programs, but provides the same level of performance as the Alloy Analyzer. My technique, on the other hand, produces programs that are fast enough to serve as prototype implementations. Notations for Abstract State Machines [22], rule-based transition systems [32, 30], concurrent object interactions [24], and the Maude language [7] have been translated to Prolog, but these are less expressive than Imperative Alloy. A Prolog translation also exists [25] from description logic, which has expressive power similar to that of Alloy. Many non-automated approaches [14, 21, 12] have been proposed, but the possibility of introducing errors during manual translation makes these unattractive, and most still require further refinement—even after a manual translation effort—for efficient execution.

Animation of expressive specifications is a well-studied topic. The toolkit supporting the B method [2] and tools for JML [6] and Z [19] all support animating specifications. However, many of these tools do not allow animation of the most expressive parts of the language, require a concrete instantiation of the initial state, and use constraint-solving approaches that do not scale as well as Prolog's search. By separating verification and animation from execution, my approach can provide both analysis and verification (using the Alloy Analyzer) and efficient execution (using my Prolog compiler).

The addition of constraints [18], more expressive logics [26, 15], more efficient execution strategies [3], and classical negation to logic programming [13] has made logic programming much more expressive; these advances may present an alternate approach to achieving the goal of a single language for specification and implementation.

6. Conclusions & Future Work

I have presented a compiler from the expressive, relational, first-order specification language Alloy to Prolog, making the process of implementing a specification automatic. Together with the Alloy Analyzer, this compiler represents a complete end-to-end solution for specifying and implementing programs. The Analyzer provides for the animation and verification of specifications, and the compiler I have presented in this paper allows for their execution.

My experience with this toolchain has identified two key areas for future work. First, the compiler does not yet identify parts of input specifications that may be troublesome to execute. Some Alloy constructs, such as negation and quantification, can make the resulting Prolog program impossible to execute. Specifications that make extensive use of these constructs do not seem to occur often in practice, but a warning message from the compiler would be useful to the user in case they do.

Second, performance of the compiled specifications is not yet optimal. With better knowledge of the strengths and weaknesses of the particular Prolog implementation the

compiler targets, I should be able to generate more appropriate code. Moreover, the compiler itself may be able to detect code that will perform poorly and signal a warning. I have already encountered cases requiring refinement of the specification in order to obtain efficient code; a profiling tool for a future version of the compiler might be able to suggest refinement in these cases.

Even without these improvements, my compiler produces useful prototype implementations for most specifications. My larger goal is to enable a single language to express systems at all levels of abstraction, from high-level requirements to low-level implementation. An implementation of such a language would naturally need to target several execution engines; more expressive features, such as first-order quantifiers, can be handled by the Alloy Analyzer, while low-level language features run at full speed. Of primary importance is the middle ground between these: the largest benefit to programmers comes from the use of expressive constructs in ways that can be identified and optimized by the compiler. By compiling Alloy to Prolog, I have shown that this middle ground is achievable: even the most expressive language constructs can be used in executable programs.

Acknowledgements

I am deeply grateful to Daniel Jackson, without whose guidance this work would not have been possible; to Eunsuk Kang, Rishabh Singh, and Jean Yang, who provided thoughtful comments on an early draft of this paper; and to the anonymous reviewers, who aided in the clarification of many points. This research was funded in part by the National Science Foundation under grants 0541183 (Deep and Scalable Analysis of Software), and 0707612 (CRI: CRD – Development of Alloy Tools, Technology and Materials), and by the Northrop Grumman Cybersecurity Research Consortium under the Secure and Dependable Systems by Design project.

References

[1] J.R. Abrial. *The B-book: assigning programs to meanings*. Cambridge Univ Pr, 1996.

[2] J.R. Abrial, M.K.O. Lee, D. Neilson, PN Scharbach, and I. Sørensen. The B-method. In *Proceedings of the 4th International Symposium of VDM Europe on Formal Software Development*, volume 2, pages 398–405. Springer, 1991.

[3] H. Ait-Kaci. *Warren's abstract machine: a tutorial reconstruction*. Citeseer, 1991.

[4] Alexandr Andoni, Dumitru Daniliuc, Sarfraz Khurshid, and Darko Marinov. Evaluating the "small scope hypothesis". In *In Popl '02: Proceedings Of The 29th Acm Symposium On The Principles Of Programming Languages*, 2002.

[5] E. Börger and R.F. Stärk. *Abstract state machines: a method for high-level system design and analysis*. Springer Verlag, 2003.

[6] F. Bouquet, F. Dadeau, B. Legeard, and M. Utting. Symbolic animation of JML specifications. *Lecture notes in computer science*, 3582:75, 2005.

[7] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and JF Quesada. Maude: specification and programming in rewriting logic. *Theoretical Computer Science*, 285(2):187–243, 2002.

[8] J. Crow, S. Owre, J. Rushby, N. Shankar, and D. Stringer-Calvert. Evaluating, testing, and animating PVS specifications. *Computer Science Laboratory, SRI International, Menlo Park, CA, Tech. Rep., Mar*, 2001.

[9] G. Dennis, F.S.H. Chang, and D. Jackson. Modular verification of code with SAT. In *Proceedings of the 2006 international symposium on Software testing and analysis*, page 120. ACM, 2006.

[10] AJJ Dick, PJ Krause, and J. Cozens. Computer aided transformation of Z into Prolog. In *Z User Workshop: proceedings of the Fourth Annual Z User Meeting, Oxford, 15 December 1989*, page 71. Springer Verlag, 1990.
[11] MR Frias, JP Galeotti, CGL Pombo, and NM Aguirre. DynAlloy: upgrading alloy with actions. In *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*, pages 442–450, 2005.
[12] N.E. Fuchs. Specifications are (preferably) executable. *Software Engineering Journal*, 7(5):323–334, 1992.
[13] M. Gelfond and V. Lifschitz. Logic programs with classical negation. In *Logic programming*, page 597. MIT Press, 1990.
[14] A. M. Gravell and P. Henderson. Why execute formal specifications? pages 165–184, 1991.
[15] J.S. Hodas and D. Miller. Logic programming in a fragment of intuitionistic linear logic. *Information and Computation*, 110(2):327–365, 1994.
[16] D. Jackson. *Software Abstractions: logic, language, and analysis*. The MIT Press, 2006.
[17] D. Jackson and J. Wing. Lightweight formal methods. *Lecture Notes in Computer Science*, 2021:1–1, 2001.
[18] J. Jaffar and J.L. Lassez. Constraint logic programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 111–119. ACM New York, NY, USA, 1987.
[19] X. Jia. An approach to animating Z specifications. *COMPSAC-NEW YORK-*, pages 108–108, 1995.
[20] C.B. Jones. *Systematic software development using VDM*. Prentice Hall New York, 1990.
[21] A. Kans and C. Hayton. Using ABC to prototype VDM specifications. *ACM SigPLAN Notices*, 29(1):27–36, 1994.
[22] A. Kappel. Executable specifications based on dynamic algebras. In *Logic Programming and Automated Reasoning*, pages 229–240. Springer.
[23] Butler Lampson. 6.826 class notes, 2009. (`http://web.mit.edu/6.826/www/notes/`).
[24] P. Letelier, P. Sánchez, and I. Ramos. Prototyping a requirements specification through an automatically generated concurrent logic program. *Practical Aspects of Declarative Languages*, pages 31–45.
[25] G. Lukácsy and P. Szeredi. Efficient description logic reasoning in prolog: The dlog system. *Theory and Practice of Logic Programming*, 9(03):343–414, 2009.
[26] D. Miller, G. Nadathur, F. Pfenning, and A. Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51(1-2):125–157, 1991.
[27] C. Morgan. The specification statement. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 10(3):403–419, 1988.
[28] J. Near and D. Jackson. An Imperative Extension to Alloy. *Abstract State Machines, Alloy, B and Z*, pages 118–131, 2010.
[29] D. Rayside, A. Milicevic, K. Yessenov, G. Dennis, and D. Jackson. Agile specifications. In *Proceeding of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 999–1006. ACM, 2009.
[30] B. Schätz. Formalization and Rule-Based Transformation of EMF Ecore-Based Models. *Software Language Engineering*, pages 227–244, 2009.
[31] JM Spivey. *The Z notation: a reference manual*. 1992.
[32] Daniel Varro and Dniel Varr. Automated program generation for and by model transformation systems. In *Applied Graph Transformation (AGT'02), pages 161 - 174.*, 2002.

Technical Communications of the International Conference on Logic Programming, 2010 (Edinburgh), pp. 154–161
http://www.floc-conference.org/ICLP-home.html

METHODS AND METHODOLOGIES FOR DEVELOPING ANSWER-SET PROGRAMS—PROJECT DESCRIPTION

JOHANNES OETSCH, JÖRG PÜHRER, AND HANS TOMPITS

Technische Universität Wien,
Institut für Informationssysteme 184/3,
Favoritenstraße 9–11, A–1040 Vienna, Austria
E-mail address: {oetsch,puehrer,tompits}@kr.tuwien.ac.at

Abstract. Answer-set programming (ASP) is a well-known formalism for declarative problem solving, enjoying a continuously increasing number of diverse applications. However, arguably one of the main challenges for a wider acceptance of ASP is the need of tools, methods, and methodologies that support the actual programming process. In this paper, we review the main goals of a project, funded by the Austrian Science Fund (FWF), which aims to address this aspect in a systematic manner. The project is planned for a duration of three years and started in September 2009. Generally, the focus of research will be on methodologies for *systematic program development*, *program testing*, and *debugging*. In particular, in working on these areas, special emphasis shall be given to the ability of the developed techniques to respect the declarative nature of ASP. To support a sufficient level of usability, solutions are planned to be compatible not only for the core language of ASP but also for important extensions thereof that are commonly used and realised in various answer-set solvers. Ultimately, the methods resulting from the project shall form the basis of an *integrated development environment* (IDE) for ASP that is envisaged to combine straightforward as well as advanced techniques, realising a convenient tool for developing answer-set programs.

1. Introduction

Answer-set programming (ASP), in its most important incarnation as *logic programming under the answer-set semantics* [Gel88, Gel91a], is a well-known paradigm for declarative problem solving whose underlying idea is that problems are solved by encoding them in terms of programs such that the solutions of a given problem are determined by the models of the associated program. Hence, the models of the latter provide the "answers" of the input problems—this stands in contrast to traditional logic-based knowledge representation where *proofs* constitute an answer to a problem. The development of sophisticated answer-set solvers (see, e.g., Denecker *et al.* [Den09] for a recent overview), led to successful applications in diverse fields like planning [Gel02], diagnosis [Eit99, Gel01], symbolic model checking [Hel03], bioinformatics [Pal09, Erd09b, Erd09a], e-tourism [Iel09], patient monitoring [Mil09], music composition [Boe08, Boe09], and many others.

Key words and phrases: answer-set programming, program development, testing, debugging.
This work was partially supported by the Austrian Science Fund (FWF) under grant P21698.

Technical Communications of the 26th International Conference on Logic Programming, Edinburgh, July, 2010
Editors: Manuel Hermenegildo, Torsten Schaub
LIPIcs - Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/LIPIcs.ICLP.2010.154

Although ASP is regarded as a programming paradigm, it currently offers only limited support for *developing programs*, compared to other programming languages for which a large number of tools and development methodologies exists. Indeed, ASP research so far concentrated, by and large, on (i) formal properties of the answer-set semantics, (ii) issues related to using it for knowledge representation and reasoning, and (iii) the development of ASP solvers. These endeavours led to the acceptance of ASP as a viable approach for declarative knowledge representation, but in order to achieve a wider acceptance of ASP among practising software and knowledge engineers, *tools and methods for supporting the programmer in developing programs are needed*. Although already more than a decade ago, De Schreye and Denecker [DS99] identified this need as being vital towards practicability of computational logic formalisms in general, only recently increased efforts in this direction have started in the ASP community—particularly on debugging and modularity aspects [Bra05, Pon09, Syr06, Bra07, Mik07, Geb08]. The awareness of these engineering-oriented requirements is also reflected by the launch of the SEA workshop series on *Software Engineering for Answer-Set Programming* in 2007.[1]

In this paper, we review the main goals of the project "Methods and Methodologies for Developing Answer-Set Programs" which we started in September 2009. It aims to address the above mentioned engineering requirements for ASP in a systematic manner and is hosted by the Knowledge-Based Systems Group of the Institute of Information Systems at the Vienna University of Technology. Funding is provided by the Austrian Science Fund (FWF) and the planned duration of the project is three years. It comprises two researcher positions and is lead by Hans Tompits.

Generally, the focus of research of the project will be on methodologies for *systematic program development*, *program testing*, and *debugging*. We want to study both *abstract concepts* that underlie the tasks emerging when programming in ASP as well as to *develop concrete tools* realising the researched techniques. In particular, special emphasis shall be given to the ability of the developed techniques to respect the declarative nature of ASP. Furthermore, the theoretical line of research will also involve aspects of decidability and complexity, and, concerning the development of tools, we plan to incorporate proof-of-concept implementations of the theoretical formalisms and approaches that will emerge from our research into an *integrated development environment* (IDE). The evolution of this system shall be subject to continuous evaluation in the context of laboratory courses on logic programming (such courses are held each semester at our institute).

We also plan research cooperations with different ASP groups, in particular with those at the University of Potsdam (Germany), Aalto University (Finland; incorporating the former Helsinki University of Technology), the University of Bath (U.K.), and the University of Calabria (Italy).

The next section provides a more detailed discussion of our research agenda.

2. Project objectives

2.1. Systematic program development

We want to investigate methods which support the systematic development of programs. In standard programming languages, *incremental* and *iterative program development* are

[1] See `sea07.cs.bath.ac.uk` and `sea09.cs.bath.ac.uk`.

well-known techniques. In the first method, a program is divided into subparts by functionality and developed one by one, whilst in the second method, a complex program is developed by iteratively refining the functionalities of less complex versions of it. We plan to study incremental and iterative development methods in the context of ASP. Here, the notion of a *conservative extension* [Gel91b, Gel96], which was introduced as a theoretical basis for the enhancement of a logic program by adding further rules, can be abstracted by considering program extensions in terms of *operators* prescribing how a program can be enhanced along with a binary *correspondence relation* between a program and its enhancement. The role of such a correspondence relation is to reflect the property to be preserved when enhancing the original program. The issue of program correspondence has been the subject of extensive research in the recent past [Lif01, Ino04, Eit05, Oik06, Oet07, Eit07, Fin08, Oet08, Tru09] and a project, also funded by the FWF, was conducted at our research group on this topic and successfully finished in 2008.[2]

A crucial aspect for the issue of incremental program development, where larger programs are composed from smaller program parts, is the question of *program modularity* (cf., e.g., the work of Brogi *et al.* [Bro94] about logic programs without negation). We want to analyse how different notions of a program module can be used to compose complex programs or how such notions can be extended or refined. An interesting concept in this regard is the notion of a *DLP-function* [Jan09], having its roots in early work on *lp-functions* [Gel96], which, roughly speaking, is a logic program together with an interface specifying the input, output, and local atoms of that program. DLP-functions are assigned with an extension of the answer-set semantics admitting a compositional semantics, i.e., the semantics of a modularised program is given as the union (suitably defined) of the semantics of its modules. For the software support methods that will be developed in this project, we will investigate how they can be refined to module-based versions.

We also want to develop techniques that support developers during the *intermediate coding process*. In particular, we are interested in methods that provide the user with *suggestions on how to proceed* when writing a program, which are computed from the current state of the program and the desired behaviour that is captured by the specification of a program. A potential instance of a tool providing suggestions is an intelligent *code-completion technique* that offers a selection of potential endings for a rule that is currently edited. From a methodological point of view, suggestion techniques seem especially valuable in combination with a *test-driven development approach* [Bec02]. We want to analyse the suitability of development approaches where test cases are formulated in advance and then used to directly support the coding process.

2.2. Testing methodologies

Though crucial for the quality of conventional software, there is little work on *testing methodologies* for logic programs. We want to address this issue by analysing how prominent methods from software testing [Mye79, Het91], like *black-box testing* or *white-box testing*, can be adapted to ASP. We recall that the idea of black-box testing is to derive test cases from the specification of a component but to abstract from the internal structure of the component—indeed, as the name suggests, it is only used as a "black box". Important in this context is how test cases can be derived from specifications and how to rate the *quality* of a suite of test cases with respect to their ability to detect unimplemented or

[2]See `http://www.kr.tuwien.ac.at/research/projects/eq/` for details about this project.

faultily implemented parts of a specification. Complementary to black-box testing, white-box testing is based on the *structure* of a component: For conventional languages, white-box testing aims at deriving test cases that cover possible paths through a software component. As one of the first results within our project, we developed, jointly with Ilkka Niemelä and Tomi Janhunen from Aalto University, different coverage metrics for ASP, along with their mutual relationships, and laid down basic techniques for test automation using ASP itself [Nie10].

2.3. Debugging

Complementing the development and testing aspects, we want to study methods to *debug answer-set programs*, i.e., techniques supporting the programmer in *localising and fixing program errors*. Initial research in this direction is already available [Bra05, Pon09, Syr06, Bra07, Geb08] but further investigation is needed.[3] Especially, as we want to develop debugging techniques for ASP that are applicable in real-world scenarios, debugging methodologies for non-ground programs are needed—most debugging methods studied so far concentrate on propositional programs only, however. To address this issue, we extended the meta-programming technique of Gebser *et al.* [Geb08] to the non-ground case [Oet10]. Moreover, we intend to consider not only the core language of answer-set programs but also cover important language extensions like *weak constraints*, *aggregates*, or *choice rules*.

Analogous to techniques for providing suggestions during the immediate coding process, we want to investigate similar features for program debugging as well. Advice on how to fix a program can be of different nature, e.g., proposals to remove specific constraints that eliminate desired answer sets.

Another important issue we want to address is *local debugging* in connection with modular-programming concepts. We want to clarify how the search for errors can be restricted to suspicious program components and how program parts can be individually debugged, yielding correctness of the overall program.

2.4. Specifications for answer-set programs

Most of the methods we are aiming at require information about the *intended behaviour* of a program under consideration. For instance, for localising a bug in an erroneous program, a debugging system needs to be aware of what the correct semantics of the program is, in order to classify wrong behaviour. Thus, *respective methods for specifying program properties are needed*. This point is important despite the widely held view that, because of the declarative nature of ASP, logic programs can be seen as specifications themselves, which, in turn, would eliminate the ubiquitous gap between specification and programming, as argued by Baral [Bar03]. However, since the process of developing answer-set programs is not as straightforward as the latter point of view might suggest, it may prove helpful to describe the desired behaviour of a program in a way that is easier to achieve than the program itself. This may be done, e.g., in the form of sample test cases, or formally defined program properties, that only describe *certain aspects of the intended semantics*, vis-a-vis a full specification as represented by a complete program. In any case, there is certainly some

[3] It is also worthwhile to mention here the work of Wittocx *et al.* [Wit09] on debugging ID-logic theories, i.e., first-order theories with inductive definitions.

subtlety in using declarative descriptions for a declarative language which requests paying close attention to the practicability of studied specification formalisms.

2.5. Implementation

Complementing our theoretical investigations, we also plan to develop prototype implementations of our techniques. Towards providing effective support for developers, we want to incorporate these into an *integrated development environment* (IDE). A few systems for developing answer-set programs have been introduced so far [Per07, Sur07], but, despite providing useful utilities, they are still in an early state of development and leave much room for improvement. Besides the envisaged functionalities for specification, testing, and debugging, our IDE should also allow customary functions like providing a code editor, syntax checking, and syntax highlighting, which are rather trivial from a theoretical point of view but handy in use. Moreover, we want to guarantee interoperability with most of the available popular solvers. The IDE will be subject to continuous evaluation within the laboratory courses on logic programming we teach regularly at our university, and it will be made publicly available for, e.g., other universities and their teaching needs.

3. Conclusion

Providing intelligent development methodologies and tools constitutes a natural next step in the evolution of ASP and will hopefully have a positive impact on this field as a whole. Furthermore, with such techniques at hand, both expert as well as novice programmers will have an enhanced access to powerful declarative problem-solving machineries.

For further information about the project, see

`http://www.kr.tuwien.ac.at/research/projects/mmdasp/`.

References

[Bar03] Chitta Baral. *Knowledge Representation, Reasoning, and Declarative Problem Solving*. Cambridge University Press, Cambridge, England, UK, 2003.

[Bec02] Kent Beck. *Test Driven Development: By Example*. Addison-Wesley Professional, Boston, MA, USA, 2002.

[Boe08] Georg Boenn, Martin Brain, Marina De Vos, and John Fitch. Automatic composition of melodic and harmonic music by answer set programming. In Maria Garcia de la Banda and Enrico Pontelli (eds.), *Proceedings of the 24th International Conference on Logic Programming* (*ICLP'08*), *Udine, Italy, December 9-13, 2008*, *Lecture Notes in Computer Science*, vol. 5366, pp. 160–174. Springer, Berlin-Heidelberg, Germany, 2008.

[Boe09] Georg Boenn, Martin Brain, Marina De Vos, and John Fitch. ANTON: Composing logic and logic composing. In Esra Erdem, Fangzhen Lin, and Torsten Schaub (eds.), *Proceedings of the 10th International Conference on Logic Programming and Nonmonotonic Reasoning* (*LPNMR'09*), *Potsdam, Germany, September 14-18, 2009*, *Lecture Notes in Computer Science*, vol. 5753, pp. 542–547. Springer, Berlin-Heidelberg, Germany, 2009.

[Bra05] Martin Brain and Marina De Vos. Debugging logic programs under the answer-set semantics. In *Proceedings of the 3rd Workshop on Answer Set Programming: Advances in Theory and Implementation* (*ASP'05*), *Bath, UK, July 27-29, 2005*, *CEUR Workshop Proceedings*, vol. 142. CEUR-WS.org, Aachen, Germany, 2005.

[Bra07] Martin Brain, Martin Gebser, Jörg Pührer, Torsten Schaub, Hans Tompits, and Stefan Woltran. Debugging ASP programs by means of ASP. In Chitta Baral, Gerhard Brewka, and John S. Schlipf (eds.), *Proceedings of the 9th International Conference on Logic Programming and Nonmonotonic Reasoning* (*LPNMR'07*), *Tempe, AZ, USA, May 15-17, 2007*, *Lecture Notes in Computer Science*, vol. 4483, pp. 31–43. Springer, Berlin-Heidelberg, Germany, 2007.

[Bro94] Antonio Brogi, Paolo Mancarella, Dino Pedreschi, and Franco Turini. Modular logic programming. *ACM Transactions on Programming Languages and Systems*, 16(4):1361–1398, 1994.

[Den09] Marc Denecker, Joost Vennekens, Stephen Bond, Martin Gebser, and Miroslaw Truszczynski. The second answer set programming competition. In Esra Erdem, Fangzhen Lin, and Torsten Schaub (eds.), *Proceedings of the 10th International Conference on Logic Programming and Nonmonotonic Reasoning* (*LPNMR'09*), *Potsdam, Germany, September 14-18, 2009*, *Lecture Notes in Computer Science*, vol. 5753, pp. 637–654. Springer, Berlin-Heidelberg, Germany, 2009.

[DS99] Daniel De Schreye and Marc Denecker. Assessment of some issues in CL-theory and program development. In Krzysztof R. Apt, Victor Marek, Miroslaw Truszczynski, and David S. Warren (eds.), *The Logic Programming Paradigm: A 25 Years Perspective*, Artificial Intelligence Series, pp. 195–208. Springer, Berlin-Heidelberg, Germany, 1999.

[Eit99] Thomas Eiter, Wolfgang Faber, Nicola Leone, and Gerald Pfeifer. The diagnosis frontend of the `dlv` system. *AI Communications*, 12(1–2):99–111, 1999.

[Eit05] Thomas Eiter, Hans Tompits, and Stefan Woltran. On solution correspondences in answer-set programming. In Leslie Pack Kaelbling and Alessandro Saffiotti (eds.), *Proceedings of the 19th International Joint Conference on Artificial Intelligence* (*IJCAI'05*), *Edinburgh, Scotland, UK, July 30-August 5, 2005*, pp. 97–102. Professional Book Center, Denver, CO, USA, 2005.

[Eit07] Thomas Eiter, Michael Fink, and Stefan Woltran. Semantical characterizations and complexity of equivalences in answer set programming. *ACM Transactions on Computational Logic*, 8(3):17, 2007.

[Erd09a] Esra Erdem. PHYLO-ASP: Phylogenetic systematics with answer set programming. In Esra Erdem, Fangzhen Lin, and Torsten Schaub (eds.), *Proceedings of the 10th International Conference on Logic Programming and Nonmonotonic Reasoning* (*LPNMR'09*), *Potsdam, Germany, September 14-18, 2009*, *Lecture Notes in Computer Science*, vol. 5753, pp. 567–572. Springer, Berlin-Heidelberg, Germany, 2009.

[Erd09b] Esra Erdem, Ozan Erdem, and Ferhan Türe. HAPLO-ASP: Haplotype inference using answer set programming. In Esra Erdem, Fangzhen Lin, and Torsten Schaub (eds.), *Proceedings of the 10th International Conference on Logic Programming and Nonmonotonic Reasoning* (*LPNMR'09*), *Potsdam, Germany, September 14-18, 2009*, *Lecture Notes in Computer Science*, vol. 5753, pp. 573–578. Springer, Berlin-Heidelberg, Germany, 2009.

[Fin08] Michael Fink. Equivalences in answer-set programming by countermodels in the logic of here-and-there. In Maria Garcia de la Banda and Enrico Pontelli (eds.), *Proceedings of the 24th International Conference on Logic Programming* (*ICLP'08*), *Udine, Italy, December 9-13, 2008*, *Lecture Notes in Computer Science*, vol. 5366, pp. 99–113. Springer, Berlin-Heidelberg, Germany, 2008.

[Geb08] Martin Gebser, Jörg Pührer, Torsten Schaub, and Hans Tompits. A meta-programming technique for debugging answer-set programs. In Dieter Fox and Carla P. Gomes (eds.), *Proceedings of the 23rd AAAI Conference on Artificial Intelligence* (*AAAI'08*), *Chicago, IL, USA, July 13-17, 2008*, pp. 448–453. AAAI Press, Menlo Park, CA, USA, 2008.

[Gel88] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *Proceedings of the 5th International Conference on Logic Programming* (*ICLP'88*), *Seattle, WA, USA, August 15-19, 1988*, pp. 1070–1080. The MIT Press, 1988.

[Gel91a] Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9(3/4):365–386, 1991.

[Gel91b] Michael Gelfond and Halina Przymusińska. Definitions in epistemic specifications. In Wiktor Marek, Anil Nerode, and V. S. Subrahmanian (eds.), *Proceedings of the 1st International Workshop on Logic Programming and Non-monotonic Reasoning* (*LPNMR'91*), *Washington, D.C., USA, July 23, 1991*, pp. 245–259. MIT Press, Cambridge, MA, USA, 1991.

[Gel96] Michael Gelfond and Halina Przymusińska. Towards a theory of elaboration tolerance: Logic programming approach. *Journal on Software and Knowledge Engineering*, 6(1):89–112, 1996.

[Gel01] Michael Gelfond, Marcello Balduccini, and Joel Galloway. Diagnosing physical systems in A-prolog. In Thomas Eiter, Wolfgang Faber, and Mirosław Truszczyński (eds.), *Proceedings of the 6th International Conference on Logic Programming and Nonmonotonic Reasoning* (*LPNMR'01*), *Vienna, Austria, September 17-19, 2001*, *Lecture Notes in Computer Science*, vol. 2173, pp. 213–225. Springer, Berlin-Heidelberg, Germany, 2001.

[Gel02] Michael Gelfond. The USA-Advisor: A case study in answer set programming. In Sergio Flesca, Sergio Greco, Nicola Leone, and Giovambattista Ianni (eds.), *Proceedings of the 8th European Conference on Logics in Artificial Intelligence* (*JELIA'02*), *Cosenza, Italy, September, 23-26, 2002*, *Lecture Notes in Computer Science*, vol. 2424, pp. 566–568. Springer, Berlin-Heidelberg, Germany, 2002.

[Hel03] Keijo Heljanko and Ilkka Niemelä. Bounded LTL model checking with stable models. *Theory and Practice of Logic Programming*, 3(4-5):519–550, 2003.

[Het91] William C. Hetzel and Bill Hetzel. *The Complete Guide to Software Testing*. John Wiley & Sons, Inc., New York, NY, USA, 1991.

[Iel09] Salvatore Maria Ielpa, Salvatore Iiritano, Nicola Leone, and Francesco Ricca. An ASP-based system for e-tourism. In Esra Erdem, Fangzhen Lin, and Torsten Schaub (eds.), *Proceedings of the 10th International Conference on Logic Programming and Nonmonotonic Reasoning* (*LPNMR'09*), *Potsdam, Germany, September 14-18, 2009*, *Lecture Notes in Computer Science*, vol. 5753, pp. 368–381. Springer, Berlin-Heidelberg, Germany, 2009.

[Ino04] Katsumi Inoue and Chiaki Sakama. Equivalence of logic programs under updates. In José Júlio Alferes and João Alexandre Leite (eds.), *Proceedings of the 9th European Conference on Logics in Artificial Intelligence* (*JELIA'04*), *Lisbon, Portugal, September 27-30, 2004*, *Lecture Notes in Computer Science*, vol. 3229, pp. 174–186. Springer, Berlin-Heidelberg, Germany, 2004.

[Jan09] Tomi Janhunen, Emilia Oikarinen, Hans Tompits, and Stefan Woltran. Modularity aspects of disjunctive stable models. *Journal of Artificial Intelligence Research*, 35:813–857, 2009.

[Lif01] Vladimir Lifschitz, David Pearce, and Agustín Valverde. Strongly equivalent logic programs. *ACM Transactions on Computational Logic*, 2(4):526–541, 2001.

[Mik07] Artur Mikitiuk, Eric Moseley, and Mirosław Truszczyński. Towards debugging of answer-set programs in the language PSpb. In *Proceedings of the 2007 International Conference on Artificial Intelligence* (*ICAI'07*), *Las Vegas, NV, USA, June 25-28, 2007*, vol. II, pp. 635–640. CSREA Press, 2007.

[Mil09] Alessandra Mileo, Davide Merico, and Roberto Bisiani. Non-monotonic reasoning supporting wireless sensor networks for intelligent monitoring: The SINDI system. In Esra Erdem, Fangzhen Lin, and Torsten Schaub (eds.), *Proceedings of the 10th International Conference on Logic Programming and Nonmonotonic Reasoning* (*LPNMR'09*), *Potsdam, Germany, September 14-18, 2009*, *Lecture Notes in Computer Science*, vol. 5753, pp. 585–590. Springer, Berlin-Heidelberg, Germany, 2009.

[Mye79] Glenford J. Myers. *The Art of Software Testing*. John Wiley & Sons, New York, NY, USA, 1979.

[Nie10] Ilkka Niemelä, Tomi Janhunen, Johannes Oetsch, Jörg Pührer, and Hans Tompits. On testing answer-set programs. In *Proceedings of the 19th European Conference on Artificial Intelligence* (*ECAI'10*), *Lisbon, Portugal, August 16-20*. 2010. To appear.

[Oet07] Johannes Oetsch, Hans Tompits, and Stefan Woltran. Facts do not cease to exist because they are ignored: Relativised uniform equivalence with answer-set projection. In *Proceedings of the 22nd Conference on Artificial Intelligence* (*AAAI'07*), *Vancouver, BC, Canada, July 22-26, 2007*, pp. 458–464. AAAI Press, Menlo Park, CA, USA, 2007.

[Oet08] Johannes Oetsch and Hans Tompits. Program correspondence under the answer-set semantics: The non-ground case. In Maria Garcia de la Banda and Enrico Pontelli (eds.), *Proceedings of the 24th International Conference on Logic Programming* (*ICLP'08*), *Udine, Italy, December 9-13, 2008*, *Lecture Notes in Computer Science*, vol. 5366, pp. 591–605. Springer, Berlin-Heidelberg, Germany, 2008.

[Oet10] Johannes Oetsch, Jörg Pührer, and Hans Tompits. Catching the Ouroboros: Towards debugging non-ground answer-set programs. *Theory and Practice of Logic Programming. Special Issue on the 2010 International Conference on Logic Programming*, 2010.

[Oik06] Emilia Oikarinen and Tomi Janhunen. Modular equivalence for normal logic programs. In Gerhard Brewka, Silvia Coradeschi, Anna Perini, and Paolo Traverso (eds.), *Proceedings of the 17th European Conference on Artificial Intelligence* (*ECAI'06*), *Riva del Garda, Italy, August 28-September 1, 2006*, pp. 412–416. IOS Press, Amsterdam, The Netherlands, 2006.

[Pal09] Alessandro Dal Palù, Agostino Dovier, and Enrico Pontelli. Logic programming techniques in protein structure determination: Methodologies and results. In Esra Erdem, Fangzhen Lin, and Torsten Schaub (eds.), *Proceedings of the 10th International Conference on Logic Programming and Nonmonotonic Reasoning* (*LPNMR'09*), *Potsdam, Germany, September 14-18, 2009*, *Lecture Notes in Computer Science*, vol. 5753, pp. 560–566. Springer, Berlin-Heidelberg, Germany, 2009.

[Per07] Simona Perri, Francesco Ricca, Giorgio Terracina, Daniela Cianni, and Pierfrancesco Veltri. An integrated graphic tool for developing and testing DLV programs. In Marina De Vos and Torsten Schaub (eds.), *Proceedings of the 1st International Workshop on Software Engineering for Answer Set Programming* (*SEA'07*), *Tempe, AZ, USA, May 14, 2007*, *CEUR Workshop Proceedings*, vol. 281, pp. 71–85. CEUR-WS.org, Aachen, Germany, 2007.

[Pon09] Enrico Pontelli, Tran Cao Son, and Omar El-Khatib. Justifications for logic programs under answer set semantics. *Theory and Practice of Logic Programming*, 9(1):1–56, 2009.

[Sur07] Adrian Sureshkumar, Marina De Vos, Martin Brain, and John Fitch. APE: An AnsProlog* environment. In Marina De Vos and Torsten Schaub (eds.), *Proceedings of the 1st International Workshop on Software Engineering for Answer Set Programming* (*SEA'07*), *Tempe, AZ, USA, May 14, 2007*, *CEUR Workshop Proceedings*, vol. 281, pp. 71–85. CEUR-WS.org, Aachen, Germany, 2007.

[Syr06] Tommi Syrjänen. Debugging inconsistent answer-set programs. In *Proceedings of the 11th International Workshop on Nonmonotonic Reasoning* (*NMR'06*), *Lake District, U.K., May 30-June 1, 2006*, *IfI Technical Report Series*, vol. IfI-06-04, pp. 77–83. Institut für Informatik, Technische Universität Clausthal, Clausthal-Zellerfeld, Germany, 2006.

[Tru09] Mirosław Truszczyński and Stefan Woltran. Relativized hyperequivalence of logic programs for modular programming. *Theory and Practice of Logic Programming*, 9(6):781–819, 2009.

[Wit09] Johan Wittocx, Hanne Vlaeminck, and Marc Denecker. Debugging for model expansion. In Patricia M. Hill and David Scott Warren (eds.), *Proceedings of the 25th International Conference on Logic Programming* (*ICLP'09*), *Pasadena, CA, USA, July 14-17, 2009*, *Lecture Notes in Computer Science*, vol. 5649, pp. 296–311. Springer, Berlin-Heidelberg, Germany, 2009.

Technical Communications of the International Conference on Logic Programming, 2010 (Edinburgh), pp. 162–171
http://www.floc-conference.org/ICLP-home.html

TABLING AND ANSWER SUBSUMPTION FOR REASONING ON LOGIC PROGRAMS WITH ANNOTATED DISJUNCTIONS

FABRIZIO RIGUZZI [1] AND TERRANCE SWIFT [2]

[1] ENDIF – Università di Ferrara, Via Saragat 1, Ferrara, Italy
E-mail address: fabrizio.riguzzi@unife.it

[2] CENTRIA – Universidade Nova de Lisboa, Quinta da Torre 2829-516, Caparica, Portugal
E-mail address: tswift@cs.suysb.edu

Abstract. The paper presents the algorithm "Probabilistic Inference with Tabling and Answer subsumption" (PITA) for computing the probability of queries from Logic Programs with Annotated Disjunctions. PITA is based on a program transformation techniques that adds an extra argument to every atom. PITA uses tabling for saving intermediate results and answer subsumption for combining different answers for the same subgoal. PITA has been implemented in XSB and compared with the ProbLog, `cplint` and CVE systems. The results show that in almost all cases, PITA is able to solve larger problems and is faster than competing algorithms.

Introduction

Languages that are able to represent probabilistic information have a long tradition in Logic Programming, dating back to [Sha83, van86]. With these languages, it is possible to model domains which contain uncertainty, situation often appearing in the real world. Recently, efficient systems have started to appear for performing reasoning with these languages [DR07, Kim08]

Logic Programs with Annotated Disjunction (LPADs) [Ven04] are a particularly interesting formalism because of the simplicity of their syntax and semantics, along with their ability to model causation [Ven09]. LPADs share with many other languages a distribution semantics [Sat95]: a theory defines a probability distribution over logic programs and the probability of a query is given by the sum of the probabilities of the programs where the query is true. In LPADs the distribution over logic programs is defined by means of disjunctive clauses in which the atoms in the head are annotated with a probability.

Various approaches have appeared for performing inference on LPADs. [Rig07] proposed `cplint` that first finds all the possible explanations for a query and then makes them mutually exclusive by using Binary Decision Diagrams (BDDs), similarly to what has been proposed for the ProbLog language [DR07]. [Rig08] presented SLGAD resolution that extends SLG resolution by repeatedly branching on disjunctive clauses. [Mee09] discusses the CVE algorithm that first transforms an LPAD into an equivalent Bayesian network and then performs inference on the network using the variable elimination algorithm.

Key words and phrases: Probabilistic Logic Programming, Tabling, Answer Subsumption, Logic Programs with Annotated Disjunction, Program Transformation.

Technical Communications of the 26th International Conference on Logic Programming, Edinburgh, July, 2010
Editors: Manuel Hermenegildo, Torsten Schaub
LIPIcs - Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/LIPIcs.ICLP.2010.162

In this paper, we present the algorithm "Probabilistic Inference with Tabling and Answer subsumption" (PITA) for computing the probability of queries from LPADs. PITA builds explanations for every subgoal encountered during a derivation of the query. The explanations are compactly represented using BDDs that also allow an efficient computation of the probability. Since all the explanations for a subgoal must be found, it is very useful to store such information so that it can be reused when the subgoal is encountered again. We thus propose to use tabling, which has recently been shown useful for probabilistic logic programming in [Kam00, Rig08, Kim09, Man09]. This is achieved by transforming the input LPAD into a normal logic program in which the subgoals have an extra argument storing a BDD that represents the explanations for its answers. Moreover, we also exploit answer subsumption to combine different explanations for the same answer. PITA is tested on a number of datasets and compared with `cplint`, CVE and ProbLog [Kim08]. The algorithm was able to successfully solve more complex queries than the other algorithms in most cases, and it was also almost always faster.

The paper is organized as follows. Section 1 briefly recalls tabling and answer subsumption. Section 2 illustrates syntax, semantics and inference for LPADs. Section 3 presents PITA, Section 4 describes the experiments and Section 5 concludes the paper.

1. Tabling and Answer Subsumption

The idea behind tabling is to maintain in a table both subgoals encountered in a query evaluation and answers to these subgoals. If a subgoal is encountered more than once, the evaluation reuses information from the table rather than re-performing resolution against program clauses. Although the idea is simple, it has important consequences. First, tabling ensures termination of programs with the *bounded term size property*. A program P has the bounded term size property if there is a finite function $f : N \to N$ such that if a query term Q to P has size $size(Q)$, then no term used in the derivation of Q has size greater than $f(size(Q))$. This makes it easier to reason about termination than in basic Prolog. Second, tabling can be used to evaluate programs with negation according to the Well-Founded Semantics (WFS) [van91]. Third, for queries to wide classes of programs, such as datalog programs with negation, tabling can achieve the optimal complexity for query evaluation. And finally, tabling integrates closely with Prolog, so that Prolog's familiar programming environment can be used, and no other language is required to build complete systems. As a result, a number of Prologs now support tabling, including XSB, YAP, B-Prolog, ALS, and Ciao. In these systems, a predicate p/n is evaluated using SLDNF by default: the predicate is made to use tabling by a declaration such as *table p/n* that is added by the user or compiler.

This paper makes use of a tabling feature called *answer subsumption*. Most formulations of tabling add an answer A to a table for a subgoal S only if A is a not a variant (as a term) of any other answer for S. However, in many applications it may be useful to order answers according to a partial order or (upper semi-)lattice. In the case of a lattice, answer subsumption may be specified by means of a declaration such as *table p(_,or/3 - zero/1)).* where a lattice is defined on the second argument by providing a bottom element (returned by $zero/1$) and a join operation ($or/3$). With the previous declaration, if a table contains an answer $p(a, E_1)$ and a new answer $p(a, E_2)$ were derived, the answer $p(a, E_1)$ is replaced by $p(a, E_3)$, where E_3 is obtained by $or(E_1, E_2, E_3)$. Answer subsumption over arbitrary

upper semi-lattices is implemented in XSB for stratified programs [Swi99]; in addition, the mode-directed tabling of B-Prolog can also be seen as a form of answer subsumption.

2. Logic Programs with Annotated Disjunctions

A *Logic Program with Annotated Disjunctions* [Ven04] consists of a finite set of annotated disjunctive clauses of the form $h_1 : \alpha_1 \ ; \ \ldots \ ; \ h_n : \alpha_n \leftarrow b_1, \ldots, b_m$. In such a clause $h_1, \ldots h_n$ are logical atoms and $b_1, \ldots, b_m$ are logical literals, $\{\alpha_1, \ldots, \alpha_n\}$ are real numbers in the interval $[0, 1]$ such that $\sum_{j=1}^{n} \alpha_j \leq 1$. $h_1 : \alpha_1 \ ; \ \ldots \ ; \ h_n : \alpha_n$ is called the *head* and $b_1, \ldots, b_m$ is called the *body*. Note that if $n = 1$ and $\alpha_1 = 1$ a clause corresponds to a normal program clause, sometimes called a *non-disjunctive* clause. If $\sum_{j=1}^{n} \alpha_j < 1$, the head of the annotated disjunctive clause implicitly contains an extra atom *null* that does not appear in the body of any clause and whose annotation is $1 - \sum_{j=1}^{n} \alpha_j$. For a clause C of the form above, we define $head(C)$ as $\{(h_i : \alpha_i) | 1 \leq i \leq n\}$ if $\sum_{i=1}^{n} \alpha_i = 1$ and as $\{(h_i : \alpha_i) | 1 \leq i \leq n\} \cup \{(null : 1 - \sum_{i=1}^{n} \alpha_i)\}$ otherwise. Moreover, we define $body(C)$ as $\{b_i | 1 \leq i \leq m\}$, $h_i(C)$ as h_i and $\alpha_i(C)$ as α_i.

If LPAD T is ground, a clause represents a probabilistic choice between the non-disjunctive clauses obtained by selecting only one atom in the head. If T is not ground, it can be assigned a meaning by computing its grounding, $ground(T)$. The semantics of LPADs, given in [Ven04], requires the ground program to be finite, so the program must not contain function symbols if it contains variables.

By choosing a head atom for each ground clause of an LPAD we get a normal logic program called a *possible world* of the LPAD (*instance* in [Ven04]). A probability distribution is defined over the space of possible worlds by assuming independence between the choices made for each clause.

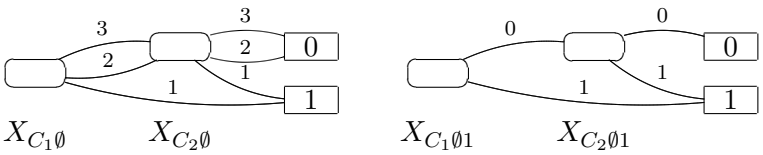
More specifically, an *atomic choice* is a triple (C, θ, i) where $C \in T$, θ is a substitution that grounds C and $i \in \{1, \ldots, |head(C)|\}$. (C, θ, i) means that, for ground clause $C\theta$, the head $h_i(C)$ was chosen. A set of atomic choices κ is *consistent* if $(C, \theta, i) \in \kappa, (C, \theta, j) \in \kappa \Rightarrow i = j$, i.e., only one head is selected for a ground clause. A *composite choice* κ is a consistent set of atomic choices. The *probability* $P(\kappa)$ *of a composite choice* κ is the product of the probabilities of the individual atomic choices, i.e. $P(\kappa) = \prod_{(C,\theta,i) \in \kappa} \alpha_i(C)$.

A *selection* σ is a composite choice that, for each clause $C\theta$ in $ground(T)$, contains an atomic choice (C, θ, i) in σ. We denote the set of all selections σ of a program T by $\mathcal{S}_T$. A selection σ identifies a normal logic program w_σ defined as follows: $w_\sigma = \{(h_i(C) \leftarrow body(C))\theta | (C, \theta, i) \in \sigma\}$. w_σ is called a *possible world* (or simply *world*) of T. Since selections are composite choices, we can assign a probability to possible worlds: $P(w_\sigma) = P(\sigma) = \prod_{(C,\theta,i) \in \sigma} \alpha_i(C)$.

We consider only *sound* LPADs in which every possible world has a total well-founded model. In this way, the uncertainty is modeled only by means of the disjunctions in the head and not by the features of the semantics. In the following we write $w_\sigma \models \phi$ to mean that the closed formula ϕ is true in the well-founded model of the program w_σ.

The probability of a closed formula ϕ according to an LPAD T is given by the sum of the probabilities of the possible worlds where the formula is true according to the WFS: $P(\phi) = \sum_{\sigma \in \mathcal{S}_T, w_\sigma \models \phi} P(\sigma)$. It is easy to see that P satisfies the axioms of probability.

Example 2.1. The following LPAD encodes the dependency of a person's sneezing on his having the flu or hay fever:



(a) MDD. (b) BDD.

Figure 1: Decision diagrams for Example 2.1.

$C_1 = \; strong\_sneezing(X) : 0.3 \; ; \; moderate\_sneezing(X) : 0.5 \; \leftarrow \; flu(X).$
$C_2 = \; strong\_sneezing(X) : 0.2 \; ; \; moderate\_sneezing(X) : 0.6 \; \leftarrow \; hay\_fever(X).$
$C_3 = \; flu(david).$
$C_4 = \; hay\_fever(david).$

If the LPAD contains function symbols, its semantics can be given by following the approach proposed in [Poo00] for assigning a semantics to ICL programs with function symbols. A similar result can be obtained using the approach of [Sat95]. In a forthcoming extended version of this paper we discuss how this can be done.

In order to compute the probability of a query, we can first find a *covering set of explanations* and then compute the probability from them. A composite choice κ identifies a set of possible worlds ω_κ that contains all the worlds relative to a selection that is a superset of κ, i.e., $\omega_\kappa = \{w_\sigma | \sigma \in \mathcal{S}_T, \sigma \supseteq \kappa\}$. Similarly we can define the set of possible worlds associated to a set of composite choices K: $\omega_K = \bigcup_{\kappa \in K} \omega_\kappa$. Given a closed formula ϕ, we define the notion of explanation and of covering set of composite choices. A finite composite choice κ is an *explanation* for ϕ if ϕ is true in every world of ω_κ. In Example 2.1, the composite choice $\{(C_1, \{X/david\}, 1)\}$ is an explanation for $strong\_sneezing(david)$. A set of choices K is *covering* with respect to ϕ if every world w_σ in which ϕ is true is such that $w_\sigma \in \omega_K$. In Example 2.1, the set of composite choices $L_1 = \{\{(C_1, \{X/david\}, 1)\}, \{(C_2, \{X/david\}, 1)\}\}$ is covering for $strong\_sneezing(david)$. Moreover, both elements of L_1 are explanations, so L_1 is a covering set of explanations for the query $strong\_sneezing(david)$.

We associate to each ground clause $C\theta$ appearing in a covering set of explanations a multivalued variable $X_{C\theta}$ with values $\{1, \ldots, head(C)\}$. Each atomic choice (C, θ, i) can then be represented by the propositional equation $X_{C\theta} = i$. If we conjoin equations for a single explanation and disjoin expressions for the different explanations we obtain a Boolean function that assumes value 1 if the values assumed by the multivalued variables correspond to an explanation for the goal. Thus, if K is a covering set of explanations for a query ϕ, the probability of the Boolean formula $f(\mathbf{X}) = \bigvee_{\kappa \in K} \bigwedge_{(C,\theta,i) \in \kappa} X_{C\theta} = i$ taking value 1 is the probability of the query, where $\mathbf{X}$ is the set of all ground clause variables.

For example, the covering set of explanations L_1 translates into the function $f(\mathbf{X}) = (X_{C_1\emptyset} = 1) \vee (X_{C_2\emptyset} = 1)$. Computing the probability of $f(\mathbf{X})$ taking value 1 is equivalent to computing the probability of a DNF formula which is an NP-hard problem. In order to solve it as efficiently as possible we use Decision Diagrams, as proposed by [DR07].

A Multivalued Decision Diagram (MDD) [Tha78] represents a function $f(\mathbf{X})$ taking Boolean values on a set of multivalued variables $\mathbf{X}$ by means of a rooted graph that has one level for each variable. Each node has one child for each possible value of the multivalued variable associated to the level of the node. The leaves store either 0 or 1. For example, the MDD corresponding to the function for L_1 is shown in Figure 1(a). MDDs represent a Boolean function $f(\mathbf{X})$ by means of a sum of disjoint terms, thus the probability of $f(\mathbf{X})$

can be computed by means of a dynamic programming algorithm that traverses the MDD and sums up probabilities.

Decision diagrams can be built with various software packages that provide highly efficient implementation of Boolean operations. However, most packages are restricted to work on Binary Decision Diagram (BDD), i.e., decision diagrams where all the variables are Boolean [Bry86]. To work on MDD with a BDD package, we must represent multivalued variables by means of binary variables. Various options are possible, we found that the following, proposed in [DR08], gives the best performance. For a variable X having n values, we use $n-1$ Boolean variables $X_1, \ldots, X_{n-1}$ and we represent the equation $X = i$ for $i = 1, \ldots n-1$ by means of the conjunction $\overline{X_1} \wedge \overline{X_2} \wedge \ldots \wedge \overline{X_{i-1}} \wedge X_i$, and the equation $X = n$ by means of the conjunction $\overline{X_1} \wedge \overline{X_2} \wedge \ldots \wedge \overline{X_{n-1}}$. The BDD representation of the function for L_1 is given in Figure 1(b). The Boolean variables are associated with the following parameters: $P(X_1) = P(X = 1), \ldots, P(X_i) = \frac{P(X=i)}{\prod_{j=1}^{i-1}(1-P(X_j))}$.

3. Program Transformation

The first step of the PITA algorithm is to apply a program transformation to an LPAD to create a normal program that contains calls for manipulating BDDs. In our implementation, these calls provide a Prolog interface to the CUDD[1] C library and use the following predicates[2]

- *init, end*: for the allocation and deallocation of a BDD manager, a data structure used to keep track of the memory for storing BDD nodes;
- *zero(-BDD), one(-BDD), and(+BDD1, +BDD2, -BDDO), or(+BDD1, +BDD2, -BDDO), not(+BDDI, -BDDO)*: Boolean operations between BDDs;
- *add_var(+N_Val, +Probs, -Var)*: addition of a new multi-valued variable with *N_Val* values and parameters *Probs*;
- *equality(+Var, +Value, -BDD)*: *BDD* represents *Var=Value*, i.e. that the variable *Var* is assigned *Value* in the BDD;
- *ret_prob(+BDD, -P)*: returns the probability of the formula encoded by *BDD*.

add_var(+N_Val,+Probs,-Var) adds a new random variable associated to a new instantiation of a rule with *N_Val* head atoms and parameters list *Probs*. The auxiliary predicate *get_var_n/4* is used to wrap *add_var/3* and avoid adding a new variable when one already exists for an instantiation. As shown below, a new fact *var(R,S,Var)* is asserted each time a new random variable is created, where R is an identifier for the LPAD clause, S is a list of constants, one for each variable of the clause, and *Var* is an integer that identifies the random variable associated with clause R under grounding S. The auxiliary predicates has the following definition

$get\_var\_n(R, S, Probs, Var) \leftarrow$
  $(var(R, S, Var) \rightarrow true$;
  $length(Probs, L), add\_var(L, Probs, Var), assert(var(R, S, Var)))$.

where R, S and *Probs* are input arguments while *Var* is an output argument.

The PITA transformation applies to clauses, literals and atoms.

- If h is an atom, $PITA_h(h)$ is h with the variable BDD added as the last argument.
- If b_j is an atom, $PITA_b(b_j)$ is b_j with the variable B_j added as the last argument.

[1] `http://vlsi.colorado.edu/~fabio/`

[2] BDDs are represented in CUDD as pointers to their root node.

In either case for an atom a, $BDD(\text{PITA}(a))$ is the value of the last argument of $PITA(a)$,

- If b_j is negative literal $\neg a_j$, $PITA_b(b_j)$ is the conditional $(PITA'_b(a_j) \rightarrow not(BN_j, B_j); one(B_j))$, where $PITA'_b(a_j)$ is a_j with the variable BN_j added as the last argument.

In other words, the BDD BN_j for a is negated if it exists (i.e. $PITA'_b(a_j)$ succeeds); otherwise the BDD for the constant function 1 is returned.

A non-disjunctive fact $C_r = h$ is transformed into the clause
$PITA(C_r) = PITA_h(h) \leftarrow one(BDD)$.

A disjunctive fact $C_r = h_1 : \alpha_1 \ ; \ \ldots \ ; \ h_n : \alpha_n.$ where the parameters sum to 1, is transformed into the set of clauses $PITA(C_r)$

$$\begin{aligned} PITA(C_r,1) = PITA_h(h_1) \leftarrow \ & get\_var\_n(r,[],[\alpha_1,\ldots,\alpha_n],Var), \\ & equality(Var,1,BDD). \\ \ldots & \\ PITA(C_r,n) = PITA_h(h_n) \leftarrow \ & get\_var\_n(r,[],[\alpha_1,\ldots,\alpha_n],Var), \\ & equality(Var,n,BDD). \end{aligned}$$

In the case where the parameters do not sum to one, the clause is first transformed into $h_1 : \alpha_1 \ ; \ \ldots \ ; \ h_n : \alpha_n \ ; \ null : 1 - \sum_1^n \alpha_i.$ and then into the clauses above, where the list of parameters is $[\alpha_1, \ldots, \alpha_n, 1 - \sum_1^n \alpha_i]$ but the $(n+1)$-th clause (the one for $null$) is not generated.

The definite clause $C_r = h \leftarrow b_1, b_2, \ldots, b_m.$ is transformed into the clause

$$\begin{aligned} PITA(C_r) = PITA_h(h) \leftarrow \ & PITA_b(b_1), PITA_b(b_2), and(B_1, B_2, BB_2), \ldots, \\ & PITA_b(b_m), and(BB_{m-1}, B_m, BDD). \end{aligned}$$

The disjunctive clause

$C_r = h_1 : \alpha_1 \ ; \ \ldots \ ; \ h_n : \alpha_n \leftarrow b_1, b_2, \ldots, b_m.$

where the parameters sum to 1, is transformed into the set of clauses $PITA(C_r)$

$$\begin{aligned} PITA(C_r,1) = PITA_h(h_1) \leftarrow \ & PITA_b(b_1), PITA_b(b_2), and(B_1, B_2, BB_2), \ldots, \\ & PITA_b(b_m), and(BB_{m-1}, B_m, BB_m), \\ & get\_var\_n(r, VC, [\alpha_1,\ldots,\alpha_n], Var), \\ & equality(Var,1,B), and(BB_m, B, BDD). \\ \ldots & \\ PITA(C_r,n) = PITA_h(h_n) \leftarrow \ & PITA_b(b_1), PITA_b(b_2), and(B_1, B_2, BB_2), \ldots, \\ & PITA_b(b_m), and(BB_{m-1}, B_m, BB_m), \\ & get\_var\_n(r, VC, [\alpha_1,\ldots,\alpha_n], Var), \\ & equality(Var,n,B), and(BB_m, B, BDD). \end{aligned}$$

where VC is a list containing each variable appearing in C_r. If the parameters do not sum to 1, the same technique used for disjunctive facts can be applied.

Example 3.1. Clause C_1 from the LPAD of Example 2.1 is translated into

$$\begin{aligned} strong\_sneezing(X, BDD) \quad & \leftarrow flu(X, B_1), get\_var\_n(1, [X], [0.3, 0.5, 0.2], Var), \\ & equality(Var, 1, B), and(B_1, B, BDD). \\ moderate\_sneezing(X, BDD) \quad & \leftarrow flu(X, B_1), get\_var\_n(1, [X], [0.3, 0.5, 0.2], Var), \\ & equality(Var, 2, B), and(B_1, B, BDD). \end{aligned}$$

while clause C_3 is translated into

$flu(david, BDD) \quad \leftarrow one(BDD).$

In order to answer queries, the goal *solve(Goal,P)* is used, which is defined by

$$
\begin{array}{ll}
solve(Goal, P) & \leftarrow init, retractall(var(\_,\_,\_)), \\
& add\_bdd\_arg(Goal, BDD, GoalBDD), \\
& (call(GoalBDD) \rightarrow ret\_prob(BDD, P)\ ;\ P = 0.0), \\
& end.
\end{array}
$$

Moreover, various predicates of the LPAD should be declared as tabled. For a predicate p/n, the declaration is *table p(_1,...,_n,or/3-zero/1)*, which indicates that answer subsumption is used to form the disjunct of multiple explanations: At a minimum, the predicate of the goal should be tabled; as in normal programs, tabling may also be used for to ensure termination of recursive predicates, or to reduce the complexity of evaluations.

PITA is correct for range restricted, bounded term-size and fixed-order dynamically stratified LPADs. A formal presentation with all proofs and supporting definitions will be reported in a forthcoming extended version of this paper.

4. Experiments

PITA was tested on programs encoding biological networks from [DR07], a game of dice from [Ven04] and the four testbeds of [Mee09]. PITA was compared with the exact version of ProbLog [DR07] available in the git version of Yap as of 19/12/2009, with the version of `cplint` [Rig07] available in Yap 6.0 and with the version of CVE [Mee09] available in ACE-ilProlog 1.2.20. All experiments were performed on Linux machines with an Intel Core 2 Duo E6550 processor (2333 MHz) and 4 GB of RAM.

The biological network problems compute the probability of a path in a large graph in which the nodes encode biological entities and the links represents conceptual relations among them. Each programs in this dataset contains a deterministic definition of path plus a number of links represented by probabilistic facts. The programs have been sampled from a very large graph and contain 200, 400, ..., 5000 edges. Sampling has been repeated ten times, so overall we have 10 series of programs of increasing size. In each test we queried the probability that the two genes HGNC_620 and HGNC_983 are related.

We used the definition of path of [Kim08] that performs loop checking explicitly by keeping the list of visited nodes:

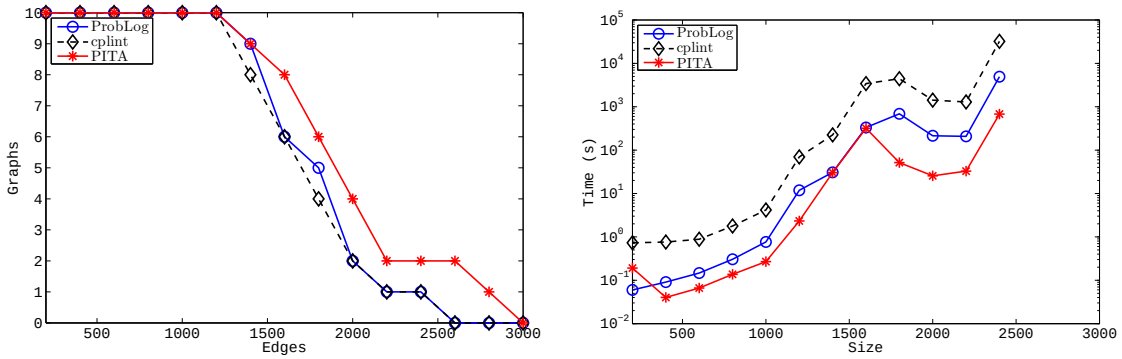
$$
\begin{array}{lll}
path(X, Y) & \leftarrow & path(X, Y, [X], Z). \\
path(X, Y, V, [Y|V]) & \leftarrow & edge(X, Y). \\
path(X, Y, V0, V1) & \leftarrow & edge(X, Z), append(V0, \_S, V1), \\
& & \neg member(Z, V0), path(Z, Y, [Z|V0], V1).
\end{array}
$$

This definition gave better results than the one without explicit loop checking. We are currently investigating the reasons for this unexpected behavior.

We ran PITA, ProbLog and `cplint` on the graphs in sequence starting from the smallest program and in each case we stopped after one day or at the first graph for which the program ended for lack of memory[3]. In PITA, we used the group sift method for automatic reordering of BDDs variables[4]. Figure 2(a) shows the number of subgraphs for which each algorithm was able to answer the query as a function of the size of the subgraphs, while Figure 2(b) shows the execution time averaged over all and only the subgraphs for which all the algorithms succeeded. PITA was able to solve more subgraphs and in a shorter time

[3]CVE was not applied to this dataset because the current version can not handle graph cycles.

[4]For each experiment, we used either group sift automatic reordering or no reordering of BDDs variables depending on which gave the best results.



(a) Number of successes. (b) Execution times.

Figure 2: Biological graph experiments.

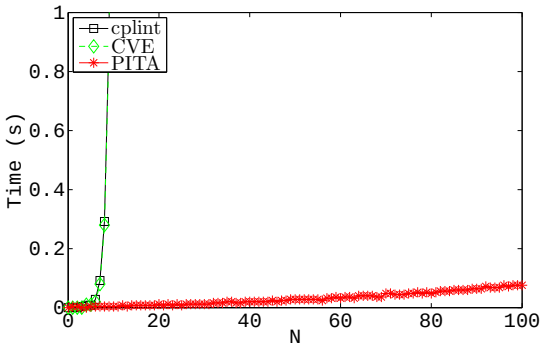


Figure 3: Three sided die.

than `cplint` and ProbLog. For PITA the vast majority of time for larger graphs was spent on BDD maintenance.

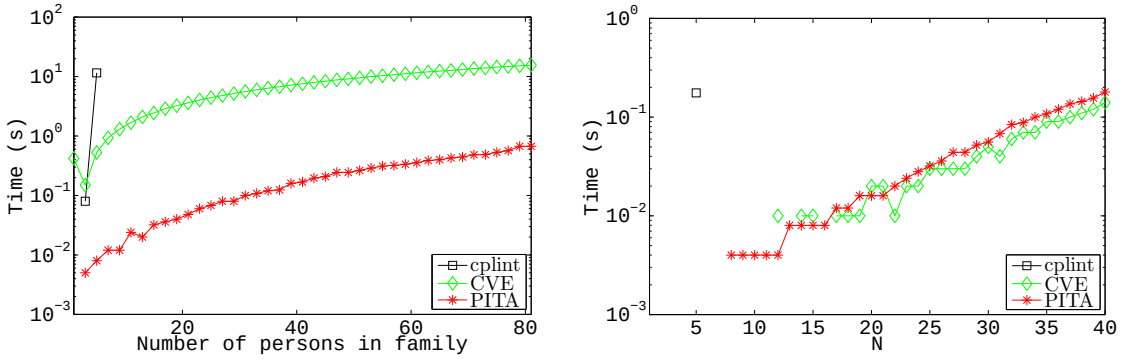
The second problem models a game in which a die with three faces is repeatedly thrown until a 3 is obtained. This problem is encoded by the program

$$on(0,1):1/3\ ;\ on(0,2):1/3\ ;\ on(0,3):1/3.$$
$$on(N,1):1/3\ ;\ on(N,2):1/3\ ;\ on(N,3):1/3 \leftarrow$$
$$N1 \text{ is } N-1, N1 \geq 0, on(N1,F), \neg on(N1,3).$$

Form the above program, we query the probability of *on(N,1)* for increasing values of N. Note that this problem can also be seen as computing the probability that a Hidden Markov Model (HMM) is in state 1 at time N, where the HMM has three states of which 3 is an end state.

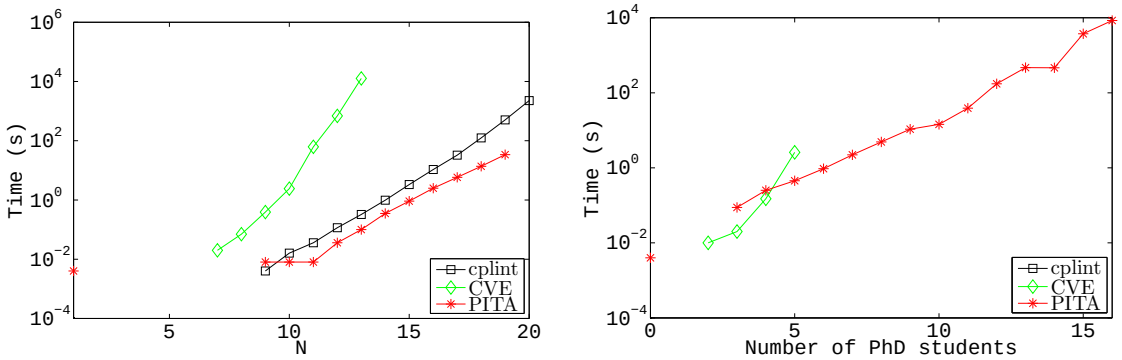
In PITA, we disabled automatic variable reordering. The execution times of PITA, CVE and `cplint` are shown in Figure 3. In this problem, tabling provides an impressive speedup, since computations can be reused often.

The four datasets of [Mee09], containing programs of increasing size. served as a final suite of benchmarks. `bloodtype` encodes genetic inheritance of blood type, `growingbody` and `growinghead` contains programs with growing bodies and heads respectively, and `uwcse` encodes a university domain. In PITA we disabled automatic reordering of BDDs variables



(a) `bloodtype`. (b) `growingbody`.

Figure 4: Datasets from [Mee09].



(a) `growinghead`. (b) `uwcse`.

Figure 5: Datasets from [Mee09].

for all datasets except for `uwcse` where we used group sift. The execution times of `cplint`, CVE and PITA are shown respectively in Figures 4(a), 4(b), 5(a) and 5(b)[5]. PITA was faster than `cplint` in all domains and faster than CVE in all domains except `growingbody`.

5. Conclusion and Future Works

This paper presents the algorithm PITA for computing the probability of queries from an LPAD. PITA is based on a program transformation approach in which LPAD disjunctive clauses are translated into normal program clauses.

The experiments substantiate the PITA approach which uses BDDs together with tabling with answer subsumption. PITA outperformed `cplint`, CVE and ProbLog in scalability or speed in almost all domains considered.

[5]For the missing points at the beginning of the lines a time smaller than 10^{-6} was recorded. For the missing points at the end of the lines the algorithm exhausted the available memory.

References

[Bry86] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. on Comput.*, 35(8):677–691, 1986.

[DR07] L. De Raedt, A. Kimmig, and H. Toivonen. ProbLog: A probabilistic Prolog and its application in link discovery. In *International Joint Conference on Artificial Intelligence*, pp. 2462–2467. 2007.

[DR08] L. De Raedt, B. Demoen, D. Fierens, B. Gutmann, G. Janssens, A. Kimmig, N. Landwehr, T. Mantadelis, W. Meert, R. Rocha, V. Santos Costa, I. Thon, and J. Vennekens. Towards digesting the alphabet-soup of statistical relational learning. In *NIPS*2008 Workshop on Probabilistic Programming*. 2008.

[Kam00] Y. Kameya and T. Sato. Efficient EM learning with tabulation for parameterized logic programs. In *Computational Logic*, *LNCS*, vol. 1861, pp. 269–284. Springer, 2000.

[Kim08] A. Kimmig, V. Santos Costa, R. Rocha, B. Demoen, and L. De Raedt. On the efficient execution of ProbLog programs. In *International Conference on Logic Programming*, *LNCS*, vol. 5366, pp. 175–189. Springer, 2008.

[Kim09] A. Kimmig, B. Gutmann, and V. Santos Costa. Trading memory for answers: Towards tabling ProbLog. In *International Workshop on Statistical Relational Learning*. KU Leuven, Leuven, Belgium, 2009.

[Man09] T. Mantadelis and G. Janssens. Tabling relevant parts of SLD proofs for ground goals in a probabilistic setting. In *Colloquium on Implementation of Constraint and Logic Programming Systems*. 2009.

[Mee09] W. Meert, J. Struyf, and H. Blockeel. CP-Logic theory inference with contextual variable elimination and comparison to BDD based inference methods. In *International Conference on Inductive Logic Programming*. KU LEuven, Leuven, Belgium, 2009.

[Poo00] D. Poole. Abducing through negation as failure: stable models within the independent choice logic. *J. Log. Program.*, 44(1-3):5–35, 2000.

[Rig07] F. Riguzzi. A top down interpreter for LPAD and CP-logic. In *Congress of the Italian Association for Artificial Intelligence*, *LNAI*, vol. 4733, pp. 109–120. Springer, 2007.

[Rig08] F. Riguzzi. Inference with logic programs with annotated disjunctions under the well founded semantics. In *International Conference on Logic Programming*, *LNCS*, vol. 5366, pp. 667–771. Springer, 2008.

[Sat95] T. Sato. A statistical learning method for logic programs with distribution semantics. In *International Conference on Logic Programming*, pp. 715–729. 1995.

[Sha83] E. Y. Shapiro. Logic programs with uncertainties: a tool for implementing rule-based systems. In *International Joint conference on Artificial intelligence*, pp. 529–532. Morgan Kaufmann Publishers Inc., 1983.

[Swi99] T. Swift. Tabling for non-monotonic programming. *Ann. Math. Artif. Intell.*, 25(3-4):201–240, 1999.

[Tha78] A. Thayse, M. Davio, and J. P. Deschamps. Optimization of multivalued decision algorithms. In *International Symposium on Multiple-Valued Logic*, pp. 171–178. IEEE Computer Society Press, Los Alamitos, CA, USA, 1978.

[van86] M H van Emden. Quantitative deduction and its fixpoint theory. *J. Log. Program.*, 30(1):37–53, 1986.

[van91] A. van Gelder, K.A. Ross, and J.S. Schlipf. Unfounded sets and well-founded semantics for general logic programs. *J. ACM*, 38(3):620–650, 1991.

[Ven04] J. Vennekens, S. Verbaeten, and M. Bruynooghe. Logic programs with annotated disjunctions. In *International Conference on Logic Programming*, *LNCS*, vol. 3131, pp. 195–209. Springer, 2004.

[Ven09] J. Vennekens, M. Denecker, and M. Bruynooghe. CP-logic: A language of causal probabilistic events and its relation to logic programming. *Theory Pract. Log. Program.*, 9(3):245–308, 2009.

Technical Communications of the International Conference on Logic Programming, 2010 (Edinburgh), pp. 172–181
http://www.floc-conference.org/ICLP-home.html

SUBSUMER: A PROLOG θ-SUBSUMPTION ENGINE

JOSÉ SANTOS [1] AND STEPHEN MUGGLETON [1]

[1] Department of Computing, Imperial College London
E-mail address: {jcs06,shm}@doc.ic.ac.uk

Abstract. State-of-the-art θ-subsumption engines like Django (C) and Resumer2 (Java) are implemented in imperative languages. Since θ-subsumption is inherently a logic problem, in this paper we explore how to efficiently implement it in Prolog.

θ-subsumption is an important problem in computational logic and particularly relevant to the Inductive Logic Programming (ILP) community as it is at the core of the hypotheses coverage test which is often the bottleneck of an ILP system. Also, since most of those systems are implemented in Prolog, they can immediately take advantage of a Prolog based θ-subsumption engine.

We present a relatively simple ($\approx$ 1000 lines in Prolog) but efficient and general θ-subsumption engine, Subsumer. Crucial to Subsumer's performance is the dynamic and recursive decomposition of a clause in sets of independent components. Also important are ideas borrowed from constraint programming that empower Subsumer to efficiently work on clauses with up to several thousand literals and several dozen distinct variables.

Using the notoriously challenging Phase Transition dataset we show that, cputime wise, Subsumer clearly outperforms the Django subsumption engine and is competitive with the more sophisticated, state-of-the-art, Resumer2. Furthermore, Subsumer's memory requirements are only a small fraction of those engines and it can handle arbitrary Prolog clauses whereas Django and Resumer2 can only handle Datalog clauses.

1. Introduction and motivation

Current state-of-the-art ILP systems are usually developed in Prolog, e.g. Aleph [Sri07] and ProGolem [Mug09], mainly because many of the algorithms needed for an ILP system are already built-in in a Prolog engine (e.g. unification, backtracking, SLD-resolution).

However, for complex learning problems where predicates are highly non-determinate and the target concept size is large ($>$ 10 literals), the Prolog's built-in SLD-resolution is inadequate. In these situations there is a combinatorial explosion of alternative variable bindings and consequently it will often take too long for the Prolog engine to decide whether the given goal succeeds. This is unacceptable for an ILP system as there will be, typically, tenths to hundredths of thousands such complex goals (i.e. putative hypothesis) that need to be evaluated before a final theory is proposed.

The subsumption problem at the culprit of the ILP bottleneck has not received much attention because, for many ILP applications, Prolog's built-in resolution seems to suffice. However, due to the non-determinism explosion highlighted above, ILP researchers often

1998 ACM Subject Classification: I.2.3 Deduction and Theorem Proving.
Key words and phrases: Theta-subsumption, Prolog, Inductive Logic Programming.

Technical Communications of the 26th International Conference on Logic Programming, Edinburgh, July, 2010
Editors: Manuel Hermenegildo, Torsten Schaub
LIPIcs - Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/LIPIcs.ICLP.2010.172

have to bound the maximum hypotheses length and recall (i.e. number of solutions per predicate) to relatively small values, which may be preventing better theories to be found.

In the last few years two efficient subsumption engines, Django [Mal04] and Resumer2 [Kuz08], were developed. However these are complex engines, around 10.000 lines of source code each, implemented in C and Java respectively, making them unpractical to use within a Prolog based ILP system. More importantly, both those engines require substantial amounts of memory, sometimes 10x more memory than the ILP system itself for the same data. This limits considerably their applicability given that, for challenging problems, the ILP system already consumes a sizeable portion of the system's resources.

The motivation for Subsumer was to develop a simple, lightweight, fully general Prolog subsumption engine that could be easily integrated from any Prolog application and, in particular, Prolog implementations of ILP systems.

2. The θ-subsumption problem

θ-subsumption [Rob65] is an approximation to logical implication. While implication is undecidable in general θ-subsumption is a NP-complete problem [Kap86]. A clause $C\theta$-subsumes a clause D ($C \vdash_\theta D$) if and only if there exists a substitution θ such that $C\theta \subseteq D$.

Example 2.1 (θ-subsumption)**.**
$C : h(X_0) \leftarrow l1(X_0, X_1), l1(X_0, X_2), l1(X_0, X_3), l2(X_1, X_2), l2(X_1, X_3)$
$D : h(c_0) \leftarrow l1(c_0, c_1), l1(c_0, c_2), l2(c_1, c_2)$
$C\theta$ subsumes D with $\theta = \{X_0/c_0, X_1/c_1, X_2/c_2, X_3/c_2\}$.

The θ-subsumption problem is thus, given two clauses, C and D, find a substitution θ such that all literals of C can be mapped into a subset of the literals of D.

The standard algorithm for θ-subsumption is based on Prolog's SLD-resolution [Kow71]. Within SLD-resolution all mappings from the literals in C onto the literals in D (for the same predicate symbol) are constructed left-to-right in a depth-first search manner. Note that the order of the literals in C has a significant impact on SLD-resolution (in)efficiency.

2.1. θ-subsumption time complexity

Let N and M be the lengths of clauses C and D. The standard θ-subsumption algorithm has complexity $O(M^N)$ as we need to map each literal of C (ranging from $1..N$) to a literal in D (ranging from $1..M$).

In practice, since SLD-resolution tests the consistency of the matching while constructing the substitution (thus bounding other variables) and not just at the end, for clauses C with too many literals (i.e. $M \approx N$) the subsumption problem may become overconstrained and thus be easier than when M is a fraction of N.

Let V be the set of distinct variables in C, and T the set of distinct terms in D. The θ-subsumption problem is then equivalent to do a mapping from V to T. This approach has complexity $O(|T|^{|V|})$ which is generally better than $O(M^N)$ since usually the clauses we are interested have $|T| << M$ and $|V| << N$. Django, Resumer2 and Subsumer all use this latter mapping.

```
1.  solve_component(VarsInComp, VarsConstr)
2.    if VarsInComp is empty then
3.      return true
4.    Let V = most_promising_free_variable(VsInComp, VsConstr)
5.    Let SubVsInComps = decomp_comp(VsInComp, VsConstr, V)
6.    Let V_Neighbours = free vars sharing a literal with V
7.    for each value Val in V's domain
8.    do
9.      V=Val;
9.      Let NVsConstr = update_vars_domains(V_Neighbours, VsConstr)
10.     for each component VComp in SubVsInComps
11.     do
12.       solve_component(VComp, NVsConstr)
13.     done
14.   done
15.   return false
16. end solve
```

Figure 1: Pseudo-code for Subsumer's main algorithm

3. Subsumer: A Prolog θ-subsumption engine

Subsumer is a publicly available (`http://ilp.doc.ic.ac.uk/Subsumer`), simple ($\approx$ 1000 lines of Prolog) fully general θ-subsumption engine with the expected behaviour from a Prolog implementation as it does not need to keep state. The Subsumer library exports a predicate, $theta\_subsumes(+subsumer, +subsumee)$, that either fails or succeeds. In case of success the variables in the subsumer clause are bound with the corresponding terms/variables of the subsumee and all possible solutions are returned by backtracking.

3.1. Main algorithm

Subsumer's main algorithm (Fig 1) works by at each iteration finding the most "promising" free (i.e still unbound) variable, V, to bound from the current component. Note that a component is defined solely by the variables appearing on it. The current heuristic is to pick the variable with smallest domain. Then the current component is decomposed assuming V has been bound (line 5). The components are returned in increasing order of their number of variables. In that way smaller components, which in principle are easier to test, are evaluated before longer ones. This can speed up the overall subsumption test significantly in case no solution is found for those smaller components.

In line 7 we iterate over the possible values for V's domain and in line 9 update its neighbour variables domain. This neighbour variable domain update is the most expensive part of Subsumer's algorithm but, due to space restrictions, we will be not be able to go into detail here. Essentially, it is implemented with a sophisticated indexing and back-indexing datastructure, that allows efficient assignment of a value to a variable and respective propagation of its new value to its direct interacting variables.

Each time the domain for a neighbour of V becomes inconsistent we have to backtrack and assign a different value to V. Although this can be particularly lengthy and get to several levels of deep recursion before a backtracking occurs, it works well in practice.

Also note that this algorithm is natural to parallelize. The natural place is the "for each loop" in line 8 where we could evaluate several components in parallel. This type of

parallelization has the peculiar property of possibly achieving superlinear (in the number of cores) speedups in case the subsumption test fails. This is because if a thread evaluating a component fails, all the other component evaluation threads running in parallel can stop immediately as there will be no solution for the whole clause. Unfortunately, however, implementing this parallel algorithm is not easy with current Prolog compilers [1].

3.2. Datastructures

The subsumer clause, $C = h \leftarrow b_1, .., .., b_n$ is represented as a list of literals. The hypothesis is preprocessed to gather all the distinct (upon variable renaming) calling patterns for the existing predicate symbols. E.g. $l1(X_0), X_1$ and $l1(X_1, X_2)$ have the same calling pattern but $l1(X_0, X_1)$ and $l1(X_0, X_0)$ are distinct.

The subsumee clause, $D = e \leftarrow g_1, .., .., g_n$ is given as a list of ground literals representing everything known to be true about e (it is the ground bottom clause of e with recall set to infinity). The example is preprocessed so that we just keep for each distinct predicate symbol $p_{s/a}$ (i.e. PredicateName/Arity) its available list of values $Val(p_{s/a})$, that is the predicate symbol domain. For instance, we would compactly represent clause D in Example 2.1, as $\{l1/2 : [\langle c_0, c_1\rangle, \langle c_0, c_2\rangle], l2/2 : [\langle c_0, c_2\rangle]\}$.

The space needed to store clause's D related information is thus: $O(\sum_1^N Val(p_{s/a_i}))$ where N is the number of distinct predicate symbols in D. A necessary condition for subsumption is that all distinct predicate symbols in C also exist in D.

The variables are extracted from C and their initial domain is computed. The initial domain for a variable is the intersection of its individual domains in each of the unique calling patterns it occurs. For instance, we the initial domains for clause C when subsuming clause D in Example 2.1, is $X_0 \in \{c_0\}, X_1 \in \{c_1\}, X_2 \in \{c_2\}, X_3 \in \{c_2\}$.

All direct pairwise variable interactions are also stored. A variable v_1 directly interacts with another variable v_2 iff they share the same literal in C. For instance, we have the following variable interactions for clause C in Example 2.1: $X_0 : \{X_1, X_2, X_3\}, X_1 : \{X_0, X_2, X_3\}, X_2 : \{X_0, X_1\}, X_3 : \{X_0, X_3\}$.

We also have a datastructure that, for each variable, holds the indexs of the literals where the variable occurs in the clause (clause's head being index 1). For the same clause C from Example 2.1 we then have $X_0 : [1, 2, 3, 4], X_1 : [2, 5, 6], X_2 : [3, 5], X_3 : [4, 6]$.

3.3. Clause decomposition

The dominant factor for reduced time complexity in Subsumer is clause decomposition. Let $H = h \leftarrow b_1, .., b_i, .., b_N$ and suppose literal b_i succeeds $k_i > 0$ times. The worst case number of predicate calls is $\prod_1^N k_i$ which, assuming an average branching factor, b, of solutions per literal leads to a $O(b^N)$ time complexity. For non-determinate clauses (i.e. clauses having literals with $b > 1$) this becomes untractable for relatively small N.

However, when the clause is decomposable in K groups of independent literals the complexity drops from $O(b^N)$ to $\sum_1^K O(b^{N_{g_i}})$, which is $O(b^{maxN_{g_i}})$. The worst case is now only exponential in the size of the longest group rather than the whole clause size.

[1] There are two problems: efficiency and transparency. From our experience, managing the threads explicitly in YAP is inefficient and also obfuscates the structure of the algorithm underneath. The ideal situation would be for Prolog compilers to have native parallel versions of list processing libraries (predicate checklist/2 in library(apply_macros) is the relevant one here).

The reasoning is then applied recursively to the newly found subcomponents. This idea, named once-transformation, was initially presented in [Cos03]. In Subsumer we implement a variant of it with several important differences. In the once-transformation the clause was transformed and independent literals were embedded in once/1 calls. The transformed clause was then called by the Prolog engine. In our approach, the clause is not transformed and our unit of evaluation are the distinct logical variables in a component, not a literal.

Two clause components are independent if, and only if, they do not share any (free) variable. Note that a clause is only satisfiable if all its components are. Thus if one component has no solutions then there is no solution for the whole clause. Equally importantly, the different solutions (θ-substitutions) of a component do not impact the solutions of the remaining components meaning that we can safely skip to the next component as soon as a solution for the current component has been found.

Example 3.1. $h(X) \leftarrow a(X,Y), b(X,Z), c(Y,A), d(Y,B), e(Z,C), f(Z,D)$

In Example 3.1 all variables are connected and thus the whole clause is a single component. However, when variable X becomes bound, literals $a(x,Y), c(Y,A), d(Y,B)$ belong to one component and literals $b(x,Z), e(Z,C), f(Z,D)$ to another. They are independent of each other as they do not share any common variable. This type of decomposition, when the head variables are assumed ground, is called the cut-transformation in [Cos03]. Resumer2 does this level of decomposition whereas Django does not do any form of clause decomposition.

In Subsumer this decomposition is applied recursively. If variable Y becomes bound next, then component $a(x,y), c(y,A), d(y,B)$ can be further divided into two components $c(y,A)$ and $d(y,B)$. Literal $a(x,y)$ no longer appears as it is now fully ground and thus no longer belongs to a component.

Also significantly, in Subsumer the independent components are created dynamically rather than statically at the beginning of clause evaluation. Although this has an overhead, it allows to choose the variable with the smallest domain (or another promising heuristic) as the splitting variable rather than, as in the once-transformation, an arbitrary variable where no information about its goodness exist. The costs of doing the decomposition dynamically should be more than offset by minimizing early the domain of the variable used.

3.4. Related engines

There are only two other subsumption engines comparable with Subsumer in terms of the complexity of clauses they can handle: Django [Mal04] and Resumer2 [Kuz08].

Common to the three engines are algorithms inspired by the constraint satisfaction framework. All do some custom form of arc-consistency and propagate constraints. Django and Resumer2 require particularly large quantities of memory as they perform determinate matching between the literals in the subsumer clause and the literals in the subsumee prior of starting its normal non-determinate matching.

Determinate matching is an idea originally presented in [Kie94], where signatures (fingerprints) of a literal are computed taking into account its neighbours (i.e. variables and literals it interacts). If the same unique fingerprint exists on both clauses for a given pair of literals these can be safely matched. Django computes these signatures with second level neighbours whereas Resumer2 uses only first level neighbours. This explains partially why

Django requires even more memory than Resumer2. Subsumer does not perform any form of determinate matching.

Django default variable ordering heuristic is the minimal variable domain divided by the number of variable interactions. In Resumer2 each variable is assigned a weight equal to its number of interactions divided by its domain size and then variables are selected with probability proportional to their weight. Subsumer uses simply minimal variable domain. Django also has a meta layer where it tries to adapt its heuristics to the underlying dataset. Resumer2 main novelty on the other side is a randomized restart mechanism inspired by SAT solvers, where if it finds itself stuck for a long time in a subsumption test, it restarts subsumption with a different variable ordering. This is an interesting idea whose impact we will investigate in the next section.

Finally, Subsumer can deal with arbitrary Prolog clauses whereas both Resumer2 and Django can handle only Datalog clauses (i.e. Prolog clauses with no function symbols).

4. Empirical evaluation

In this section we extensively compare Django, Resumer2 and Subsumer. The goal is to compare running times and memory requirements for the three engines on a very challenging benchmark for θ-subsumption engines. In the sections below when we refer to examples we mean the subsumee clauses and by hypotheses we mean the subsumer clauses. This analogy is due to the direct translation of clauses' roles to an ILP system.

All the datasets, subsumption engines and scripts to replicate these experiments can be found at `http://ilp.doc.ic.ac.uk/Subsumer`.

4.1. Datasets

The datasets selected to compare the subsumption engines are instances of the Phase Transition (PT) problem [Gio00]. This artificial problem was originally developed to be a challenge for relational learners like ILP systems. In an ILP system the task is to induce a theory (i.e. target concept) that, together with provided background knowledge, entails a set of positive examples (of the target concept) but no negative examples.

The PT problem is a collection of noise free datasets of varying difficulty each characterized by two parameters, the target concept size, $M \in [5..30]$, and the distinct number of terms, $L \in [12..38]$, present in a subsumee clause. Furthermore each instance is highly non-determinate with 100 solutions per distinct predicate symbol/arity. For each instance there are 200 positive and 200 negative examples evenly divided between train and test and there exists at least one single clause (the target concept) that perfectly discriminates between the positive and negative examples (i.e. has 100% accuracy).

The instances belong to three major regions: Yes, No and Phase Transition. In the Yes region the probability that a randomly generated clause will cover an arbitrary example is close to 1, in the No region is close to 0 and in the narrow Phase Transition (PT) region the probability drops abruptly from 1 to 0.

We selected 43 datasets from the set of 702 possible PT instances ($range(M) * range(L) = 26 * 27 = 702$) as they are good representatives of the three regions. 12 instances are from the Yes region, 15 from the No region and 16 from the PT region. These are the same instances that were used in [Bot03] but there to highlight the difficulty of learning concepts from the PT and No regions for a relational learning system.

4.2. Subsumees/Examples

Each example is a single (saturated) clause with all facts known to be true about it.

All the 400 examples per dataset instance were used. From the subsumption engine perspective all examples are equal, there is no distinction between positive or negative examples. However, since our hypotheses are biased to cover positive examples, it is a better challenge if subsumee clauses that are less likely to be covered are also included.

Due to the nature of the PT dataset all the examples for a particular instance have the same size (i.e. number of literals) and the number of distinct predicate symbols is equal to the concept size, M. The number of distinct terms in an example is L. The arity of all predicate symbols is three with the first argument being always the term from the head. All terms in the examples are constants with no function symbols.

Below is a small excerpt of an example for dataset id=3 (m=18,l=16). The full example has 801 literals.

$$p(d0) \leftarrow br0(d0, d0\_9, d0\_5), br0(d0, d0\_9, d0\_3), br0(d0, d0\_9, d0\_2), \ldots,$$
$$br3(d0, d0\_0, d0\_11), br3(d0, d0\_0, d0\_1), br4(d0, d0\_9, d0\_6), \ldots,$$
$$br7(d0, d0\_0, d0\_3), br7(d0, d0\_0, d0\_15), br7(d0, d0\_0, d0\_13).$$

The examples length range from 501 literals (m=5, l=15) to 2921 literals (m=29, l=24). These instances are from the Yes and No region respectively.

4.3. Subsumers/Hypotheses

The clauses used as subsumers (i.e. hypotheses) were generated using the concept of assymmetric relative minimal generalizations (ARMG) [Mug09]. Essentially the ARMG algorithm receives a clause C and an example e as input and returns a reduced clause R_c, where all literals from C responsible for not entailing e are pruned.

The hypotheses generation algorithm employed receives a list of positive examples and computes the iterative ARMG of all of them. The iterative ARMG of a list of examples is found by computing the (variablized) bottom clause for the first example and then, using it as the start clause, iteratively apply the ARMG algorithm to the remaining examples.

The more examples used to construct an ARMG the smaller (and more general) it will be. Furthermore an ARMG will at least entail all the examples used in its construction.

In order to create the ARMGs we used 10 randomly selected lists of 6, 7, 8, 9 and 10 positive only examples [2] , yielding 50 varying length hypotheses (10 hypotheses are ARMGs with 6 positive examples, ..., 10 hypotheses are ARMGs with 10 positives).

Below is a small excerpt of an hypothesis, an ARMG of 6 positive examples, for dataset id=3 (m=8,l=16). The full hypothesis has 59 literals.

$$p(A) \leftarrow br0(A, B, C), br0(A, B, D), br0(A, E, F), br0(A, E, G), br0(A, E, H), \ldots,$$
$$br1(A, E, O), br1(A, E, N), br1(A, E, L), br1(A, E, J), br2(A, J, K), \ldots,$$
$$br4(A, H, Q), br4(A, D, F), br4(A, D, C), br5(A, I, N), br6(A, J, Q).$$

[2]We did not want to mix positive and negative examples in the ARMG. The reason is that we know this dataset is noise free and since an ARMG, by construction, covers the examples used to create it, the resulting clauses would be shorter and thus less difficult to test for subsumption.

Note that, since our hypotheses are not random -they are biased towards covering positive examples- in the Yes, No and Phase transition regions the probabilities for subsumption are not necessarily close to 1, 0 and 0.5. Nevertheless, it is still relevant to divide the dataset in these three regions as the subsumption tests have a region related difficulty (e.g. longer clauses with more terms in examples and variables in hypotheses).

The hypotheses length vary significantly within each instance (e.g. from 59 to 121 literals for m=17,l=14) but the extremes are 29 literals (m=14, l=24) and 626 literals (m=26, l=12). These instances are both from the PT region. The length of the examples and hypotheses is just a rough indication of the subsumption problem difficulty. Other important factors are: ratio between those lengths, distinct terms in the examples, distinct variables in the hypotheses, distinct predicate symbols.

4.4. Subsumption engines

We used Subsumer, Django [Mal04] and Resumer2 [Kuz08]. Older subsumption engines based on determinate matching [Kie94] and maximal clique search [Sch96] were not tested as we could no longer find them publicly available. However, in [Mal04] they were tested against Django and it clearly outperformed those older engines by several orders of magnitude (speedups between 150x to 1200x).

As for Resumer2, we will also test a variant, which we name Resumer1, that has randomized restarts turned off. This experiment is interesting because it directly tests the importance of randomized restarts in this benchmark. Furthermore, by comparing the relative performance of Resumer1 to Resumer2, we can roughly estimate the gains we would obtain if we were to implement randomized restarts in top of Subsumer.

We compiled Django with gcc 4.1.2, Resumer2 (and Resumer1) with Sun's Java 1.6 and Subsumer with YAP6 Prolog [dS06], all with full optimizations enabled. All experiments were performed in a Athlon Opteron processor 1222 running at 3.0 GHz with 4 GB RAM and a 64 bit build of Linux.

4.5. Results and discussion

A first point to mention is that the four subsumption engines returned the same list of subsumed examples for each instance. This was expected as otherwise there would be at least one faulty implementation. Nevertheless, this is strong evidence that all engines correctly implement θ-subsumption. Notice that each instance consists of 50 (hypotheses) * 400 (examples) = 20.000 subsumption tests.

Analyzing Table 1[3] the first conclusion is that Django consumes too much memory. It consumes so much memory that in only 14 of the 43 datasets it did not crash for exceeding the 4Gb memory limit. It could not solve a single dataset from the No region, the most difficult one. Also, from a CPU time perspective, Django is clearly behind Resumer1/2 and Subsumer by up to 2 orders of magnitude for the few datasets it managed to finish.

The interesting comparison is between Resumer1/2 and Subsumer. Resumer1 is faster than Subsumer but the difference is merely, on average, 5%. Also relevant, standard deviation in Subsumer's running times are about half of Resumer1's. More importantly, Subsumer's memory requirements are only a small fraction (1/8 to 1/10) of either Resumer.

[3]Note that the PT and Overall columns favor Django as, naturally, we can just take into account the datasets where Django successfully finished.

Table 1: Average CPU times (seconds) and memory (megabytes) for problems in each region of the Phase Transition dataset

	Phase Transition dataset region							
	Yes		No		PT		Overall	
Engine	CPU	RAM	CPU	RAM	CPU	RAM	CPU	RAM
Django	4,404	2,248	N/A	N/A	78,736	3,037	15,023	2,361
Resumer1	99	608	544	1,167	225	749	301	855
Resumer2	75	578	154	1,136	120	875	119	883
Subsumer	190	75	442	141	292	92	316	105

Resumer2 is clearly best on all regions. It is followed by Resumer1 though Subsumer manages to outperform Resumer1 in the No region by 23%. Notice that randomized restarts are particularly helpful in this region and are solely responsible for the almost 4 times speedup that Resumer2 has over Resumer1. In the easiest Yes region, Subsumer is about 2 times slower than either Resumer and randomized restarts have almost no impact. In the PT region Resumer1 outperforms Subsumer by 30% and Resumer2 outperforms both by about 2 times showing that, again, randomized restarts are important. Randomized restarts are more helpful as the difficulty of the subsumption test increases. This is as expected as, for simple instances, randomized restarts do not have time to occur.

Overall, Resumer2 clearly outperforms Resumer1 being, on average, 2.5 times faster than it. Also the standard deviation for a subsumption test in Resumer2 decreased considerably comparing with Resumer1. Notably, this is achieved without increasing the memory footprint. This result is further evidence to Resumer2's authors claim in [Kuz08] that randomized restarts are helpful to reduce expected subsumption time.

We did a further experiment to test to which extent dynamic clause decomposition is important to Subsumer. We disabled it and analyzed how Subsumer performed using only the cut transformation. Although for the Yes and PT regions dynamic clause decomposition turned out to be mainly overhead (10%-25% slower), for all the problems in the No region it proved essential. Without it Subsumer would get stuck. It is in the No region that Subsumer outperforms Resumer1 and this is due to dynamic clause decomposition.

To test the importance of the particular compilers used, in a separate experiment we compiled Resumer1 with GNU Java compiler (gcj 4.3.3) also with full optimizations enabled. This gcj version of Resumer1 took 2.5 times longer and required 25% more memory than Resumer1 compiled with Sun's JVM. Subsumer compiled with SWI-Prolog (5.6.59) takes 5.5 times longer than with YAP6. Compilers significantly influence running times.

5. Conclusions and future directions

Our subsumption engine comparison on the challenging PT problem showed that Subsumer clearly outperformed Django both in time and memory and that it is competitive with Resumer2 without randomized restarts. Furthermore, Subsumer requires only $\approx 1/8$ of Resumer2's memory and can handle arbitrary Prolog clauses. We also confirmed the importance of randomized restarts as previously pointed out in [Kuz08]. This is incentive to implement a randomized restart strategy in a future version of Subsumer.

Also worth investigating is the impact of our θ-subsumption engine embedded in a Prolog based ILP system. Could ILP systems then tackle problems they cannot now?

Besides the ILP community, we expect that other related research communities, e.g automated theorem proving, can also profit from our Prolog subsumption engine.

From a strict performance perspective, there would be gains in relaxing Subsumer's auto-imposed constraint of having no state. Namely, often hypotheses are related and have many identical literals, much of the datastructures could be computed once and, at the expense of some memory, running times could be significantly improved.

As for the θ-subsumption problem itself, it is worth verifying if it could be entirely mapped to a constraint satisfaction problem or a sub-graph isomorphism matching problem. If so, one can then use existing state-of-the-art solvers for those problems and check whether they are any better than custom engines like Resumer2 or Subsumer.

Acknowledgments

We thank Ondrej Kuzelka and Filip Zelezný for kindly providing Resumer2 and Django and, specially, for fruitful discussions that improved both Resumer2 and Subsumer. The first author thanks Wellcome Trust for his Ph.D. scholarship. The second author thanks the Royal Academy of Engineering and Microsoft for funding his present 5 year Research Chair. We are also indebted to three anonymous referees for valuable comments.

References

[Bot03] Marco Botta, Attilio Giordana, Lorenza Saitta, and Michèle Sebag. Relational learning as search in a critical region. *Journal of Machine Learning Research*, 4:431–463, 2003.

[Cos03] Vítor Santos Costa, Ashwin Srinivasan, Rui Camacho, Hendrik Blockeel, Bart Demoen, Gerda Janssens, Jan Struyf, Henk Vandecasteele, and Wim Van Laer. Query transformations for improving the efficiency of ILP systems. *Journal of Machine Learning Research*, 4:465–491, 2003.

[dS06] Anderson Faustino da Silva and Vítor Santos Costa. The design and implementation of the YAP compiler: An optimizing compiler for logic programming languages. In Sandro Etalle and Miroslaw Truszczynski (eds.), *ICLP*, *LNCS*, vol. 4079, pp. 461–462. Springer, 2006.

[Gio00] Attilio Giordana and Lorenza Saitta. Phase transitions in relational learning. *Machine Learning*, 41(2):217–251, 2000.

[Kap86] Deepak Kapur and Paliath Narendran. NP-completeness of the set unification and matching problems. In Jörg H. Siekmann (ed.), *CADE*, *LNCS*, vol. 230, pp. 489–495. Springer, 1986.

[Kie94] Jrg-Uwe Kietz and Marcus Lbbe. An efficient subsumption algorithm for Inductive Logic Programming. *Proc. 11th Int. Conf. on Machine Learning*, pp. 130–138. Morgan Kaufmann, 1994.

[Kow71] Robert A. Kowalski and Donald Kuehner. Linear resolution with selection function. *Artif. Intell.*, 2(3/4):227–260, 1971.

[Kuz08] Ondrej Kuzelka and Filip Zelezný. A restarted strategy for efficient subsumption testing. *Fundam. Inform.*, 89(1):95–109, 2008.

[Mal04] Jérôme Maloberti and Michèle Sebag. Fast theta-subsumption with constraint satisfaction algorithms. *Machine Learning*, 55(2):137–174, 2004.

[Mug09] Stephen Muggleton, Jose Santos, and Alireza Tamaddoni-Nezhad. Progolem: a system based on relative minimal generalisation. In Luc De Raedt (ed.), *Proceedings of the 19th International Conference on ILP*, *LNCS*, vol. 5989, pp. 131–148. Springer, 2009.

[Rob65] John Alan Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, 1965.

[Sch96] Tobias Scheffer, Ralf Herbrich, and Fritz Wysotzki. Efficient theta-subsumption based on graph algorithms. In S. Muggleton (ed.), *ILP workshop*, *LNCS*, vol. 1314, pp. 212–228. Springer, 1996.

[Sri07] Ashwin Srinivasan. *The Aleph Manual*. University of Oxford, 2007.

Technical Communications of the International Conference on Logic Programming, 2010 (Edinburgh), pp. 182–191
http://www.floc-conference.org/ICLP-home.html

USING GENERALIZED ANNOTATED PROGRAMS TO SOLVE SOCIAL NETWORK OPTIMIZATION PROBLEMS

PAULO SHAKARIAN [1] AND V.S. SUBRAHMANIAN [1] AND MARIA LUISA SAPINO [2]

[1] Univestiy of Maryland
College Park, MD
E-mail address: {pshak,vs}@cs.umd.edu

[2] Università di Torino
Torino, Italy
E-mail address: mlsapino@di.unito.it

Abstract. Reasoning about social networks (labeled, directed, weighted graphs) is becoming increasingly important and there are now models of how certain phenomena (e.g. adoption of products/services by consumers, spread of a given disease) "diffuse" through the network. Some of these diffusion models can be expressed via generalized annotated programs (GAPs). In this paper, we consider the following problem: suppose we have a given goal to achieve (e.g. maximize the expected number of adoptees of a product or minimize the spread of a disease) and suppose we have limited resources to use in trying to achieve the goal (e.g. give out a few free plans, provide medication to key people in the SN) - how should these resources be used so that we optimize a given objective function related to the goal? We define a class of *social network optimization problems* (SNOPs) that supports this type of reasoning. We formalize and study the complexity of SNOPs and show how they can be used in conjunction with existing economic and disease diffusion models.

1. Introduction

There is a rapid proliferation of different types of *graph data* in the world today. These include social network data (FaceBook, Flickr, YouTube, etc.), cell phone network data [NE08] collected by virtually all cell phone vendors, email network data (such as those derived from the Enron corpus or Gmail logs), as well as information on disease networks [FC08, And79]. In addition, the World Wide Consortium's RDF standard is also a graph-based standard for encoding semantic information contained in web pages. There has been years of work on analyzing how various properties of nodes in such networks "diffuse" through the network - different techniques have been invented in different academic disciplines including economics [Jac05, Sch78], infectious diseases [FC08], sociology [Gra78] and computer science [Kem03].

1998 ACM Subject Classification: I.2.4 Knowledge Representation Formalisms and Methods.
Key words and phrases: annotated logic programming, optimization queries, social networks.
Some of the authors of this paper were funded in part by AFOSR grant FA95500610405, ARO grant W911NF0910206 and ONR grant N000140910685.

Technical Communications of the 26th International Conference on Logic Programming, Edinburgh, July, 2010
Editors: Manuel Hermenegildo, Torsten Schaub
LIPIcs - Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/LIPIcs.ICLP.2010.182

Many of these methods focus on modeling a *specific* type of diffusion in an SN and often, they only rely on the network topology [Wat99, Cow04, Ryc08], rather than on properties of vertices, and the nature of the relationships between vertices. In this paper, we first argue that Generalized Annotated Programs (GAPs) [Kif92b, Kif92a, Thi93] and their variants [Ven04, Kra04, Lu96, Lu93, Dam99] form a convenient method to express many diffusion models. Next, unlike most existing work in social networks which focus on learning diffusion models, we focus on reasoning with previously learned diffusion models (expressed via GAPs). In particular, if we wish to achieve certain *goals* based on a social network, how best can we achieve these goals? Two examples are given below.

- **(Q1) Cell phone plans.** A cell phone company is promoting a new cell phone plan - as a promotion, it is giving away k free plans to existing customers. Which k people should they pick so as to maximize the (expected) number of plan adoptees predicted by a cell phone plan adoption diffusion model they have learned from their past promotions?
- **(Q2) Medication distribution plan.** A government combating a disease spread by physical contact has limited stocks of free medication to give away. Based on a diffusion model of how the disease spreads (e.g. kids might be more susceptible than adults, those previously inoculated against the disease are safe, etc.), they want to find the k people who maximally spread the disease (so that they can provide immediate treatment to these k people in an attempt to halt the disease's spread).

Both the above problems are instances of a class of queries that we call SNOP queries. They differ from queries studied in the past in quantitative (both probabilistic and annotated) logic programming in two fundamental ways: (i) They are specialized to operate on graph data, (ii) They optimize complex kinds of objective functions. Neither of these has been studied before by any kind of quantitative logic programming framework, though work on optimizing objective functions in the context of different types of semantics (minimal model and stable model semantics) has been studied before[Leo04]. And of course, constraint logic programming[Apt03] has also extensively studied optimization issues as well in logic programming - however, here, optimization and constraint solving is embedded in the constraint logic program, whereas in our case, they are part of the query over an annotated logic program.

This paper is organized as follows. In Section 2, we provide an overview of GAPs (past work), define a social network, and explain how GAPs can represent some types of diffusion in SNs. Section 3 formally defines different types of social network optimization problems and provides results on their computational complexity. Finally, section 4 shows how our framework can represent several existing diffusion models for social networks including one each from economics, epidemiology, and computer science.

2. Technical Preliminaries

In this section, we first formalize social networks, then briefly overview generalized annotated logic programs (GAPs)[Kif92b] and then describe how GAPs can be used to represent concepts related to diffusion in SNs. Throughout this paper, we assume the existence of two arbitrary but fixed disjoint sets VP, EP of *vertex* and *edge predicate symbols* respectively. Each vertex predicate symbol has arity 1 and each edge predicate symbol has arity 2.

Definition 2.1. A **social network** ($\mathcal{S}$) is a 5-tuple $(\mathbf{V}, \mathbf{E}, \ell_{vert}, \ell_{edge}, w)$ where:

(1) $\mathbf{V}$ is a set whose elements are called vertices.
(2) $\mathbf{E} \subseteq \mathbf{V} \times \mathbf{V}$ is a multi-set whose elements are called edges.
(3) $\ell_{vert} : \mathbf{V} \rightarrow 2^{\mathsf{VP}}$ is a function, called *vertex labeling function.*
(4) $\ell_{edge} : \mathbf{E} \rightarrow \mathsf{EP}$ is a function, called *edge labeling function.* [1]
(5) $w : \mathbf{E} \times \mathsf{EP} \rightarrow [0, 1]$ is a function, called *weight function.*

We now present a brief example of an SN that will be used throughout this paper.

Example 2.2. Let us return to the cell phone example (query **(Q1)**). Figure 1 shows a toy SN the cell phone company might use. Here, we might have $\mathsf{VP} = \{male, female, adopters, temp\_adopter, non\_adptr\}$ denoting the sex and past adoption behavior of each vertex; EP might be the set $\{phone, email, IM\}$ denoting the types of interactions between vertices. $w(v_1, v_2, ep)$ denotes the percentage of communications of type $ep \in \mathsf{EP}$ initiated by v_1 that were with v_2 (measured either w.r.t. time or bytes). The function ℓ_{vert} is shown in the figure by the shape (denoting past adoption status) and shading (male/female). The type of edges (bold for phone, dashed for email, dotted for IM) is used to illustrate ℓ_{edge}.

It is important to note that our definition of social networks is much broader than that used by several researchers[And79, FC08, Jac05, Kem03] who often do not consider either ℓ_{edge} or ℓ_{vert} — these can have a significant impact on what we do with such networks.

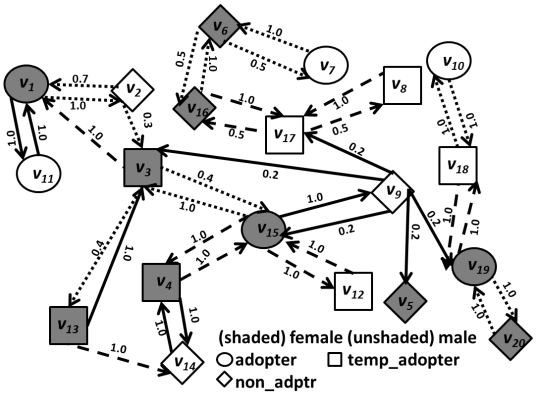


Figure 1: Example cellular network.

Note. We note that each social network must satisfy various integrity constraints. In Example 2.2, it is clear that $\ell_{vert}(V)$ should include at most one of $male, female$ and at most one of $adopters$, $temp\_adopter$,non_adptr. *We assume the existence of some integrity constraints to ensure this kind of semantic integrity* – they can be written in any reasonable syntax to express ICs – in the rest of this paper, we assume that social networks have associated ICs and that they satisfy them. In our example, we will assume ICs ensuring that a vertex can be marked with at most one of $male/female$ and at most one of $adopters, temp\_adopter, non\_adptr$.

We now recapitulate the definition of generalized annotated logic programs from [Kif92b]. We assume the existence of a set AVar of variable symbols ranging over the unit real interval $[0, 1]$ and a set $\mathcal{F}$ of function symbols each of which has an associated arity. We start by defining annotations.

Definition 2.3 (annotation term)**.** (i) Any member of $[0, 1] \cup \mathsf{AVar}$ is an annotation.
(ii) If f is an n-ary function symbol over $[0, 1]$ and $t_1, \ldots, t_n$ are annotations, then so is $f(t_1, \ldots, t_n)$.

We define a separate logical language whose constants are members of $\mathbf{V}$ and whose predicate symbols consist of $\mathsf{VP} \cup \mathsf{EP}$. We also assume the existence of a set $\mathcal{V}$ of variable

[1] Each edge $e \in \mathbf{E}$ is labeled by exactly one predicate symbol from EP. However, there can be multiple edges between two vertices labeled with different predicate symbols.

symbols ranging over the constants (vertices). No function symbols are present. Terms and atoms are defined in the usual way (cf. [Llo87]). If $A = p(t_1, \ldots, t_n)$ is an atom and $p \in \mathsf{VP}$ (resp. $p \in \mathsf{EP}$), then A is called a *vertex* (resp. *edge*) atom.

Definition 2.4 (annotated atom/GAP-rule/GAP)**.** If A is an atom and μ is an annotation, then $A : \mu$ is an *annotated atom*. If $A_0 : \mu_0, A_1 : \mu_1, \ldots, A_n : \mu_n$ are annotated atoms, then

$$A_0 : \mu_0 \quad \leftarrow \quad A_1 : \mu_1 \wedge \ldots \wedge A_n : \mu_n$$

is called a *GAP rule*. When $n = 0$, the above GAP-rule is called a *fact*. A generalized annotated program Π is a finite set of GAP rules.

Every social network $SN = (\mathbf{V}, \mathbf{E}, \ell_{vert}, \ell_{edge}, w)$ can be represented by the GAP $\Pi_{SN} = \{q(v) : 1 \leftarrow \mid v \in \mathbf{V} \wedge q \in \ell_{vert}(v)\} \cup \{ep(V_1, V_2) : w(V_1, V_2, ep) \leftarrow \mid (V_1, V_2) \in \mathbf{E} \wedge \ell_{edge}(V_1, V_2) = ep\}$.

Definition 2.5 (embedded social network)**.** A social network SN is said to be *embedded* in a GAP Π iff $\Pi_{SN} \subseteq \Pi$.

We see immediately from the definition of Π_{SN} that all social networks can be represented as GAPs. When we augment Π_{SN} with other rules — such as rules describing how certain properties diffuse through the social network, we get a GAP $\Pi \supseteq \Pi_{SN}$ that captures both the structure of the SN and the diffusion principles. Here is a small example of such a GAP.

Example 2.6. The GAP Π_{cell} might consist of Π_{SN} using the social network of Figure 1 plus the GAP-rules:

(1) $will\_adopt(V) : 0.8 \times X + 0.2 \leftarrow adopter(V) : 1 \wedge male(V) : 1 \wedge IM(V, V') : 0.3 \wedge female(V') \wedge will\_adopt(V') : X$.
(2) $will\_adopt(V) : 0.9 \times X + 0.1 \leftarrow adopter(V) : 1 \wedge male(V) : 1 \wedge IM(V, V') : 0.3 \wedge male(V') \wedge will\_adopt(V') : X$.
(3) $will\_adopt(V) : 1 \leftarrow temp\_adopter(V) : 1 \wedge male(V) : 1 \wedge email(V', V) : 1 \wedge female(V') : 1 \wedge will\_adopt(V') : 1$.

Rule (1) says that if V is a male adopter and V' is female and the weight of V's instant messages to V' is 0.3 or more, and we previously thought that V would be an adopter with confidence X, then we can infer that V will adopt the new plan with confidence $0.8 \times X + 0.2$. The other rules may be similarly read.

GAPs have a formal semantics that can be immediately used. An interpretation I is any mapping from the set of all grounds atoms to $[0, 1]$. The set $\mathcal{I}$ of all interpretations can be partially ordered via the ordering: $I_1 \preceq I_2$ iff for all ground atoms A, $I_1(A) \leq I_2(A)$. $\mathcal{I}$ forms a complete lattice under the $\preceq$ ordering.

Definition 2.7 (satisfaction/entailment)**.** An interpretation I *satisfies* a ground annotated atom $A : \mu$, denoted $I \models A : \mu$, iff $I(A) \geq \mu$. I satisfies the ground GAP-rule $AA_0 \leftarrow AA_1 \wedge \ldots \wedge AA_n$ (denoted $I \models AA_0 \leftarrow AA_1 \wedge \ldots \wedge AA_n$) iff either (i) I satisfies AA_0 or (ii) there exists an $1 \leq i \leq n$ such that I does not satisfy AA_i. I satisfies a non-ground atom (rule) iff I satisfies all ground instances of it. GAP Π *entails* AA, denoted $\Pi \models AA$, iff every interpretation I that satisfies all rules in Π also satisfies AA.

As shown by [Kif92b], we can associate a fixpoint operator with any GAP Π that maps interpretations to interpretations.

Definition 2.8. Suppose Π is any GAP and I an interpretation. The mapping $\mathbf{T}_\Pi$ that maps interpretations to interpretations is defined as $\mathbf{T}_\Pi(I)(A) = \mathbf{sup}\{\mu \mid A : \mu \leftarrow AA_1 \wedge \ldots \wedge AA_n$ is a ground instance of a rule in Π and for all $1 \leq i \leq n$, $I \models AA_i\}$.

[Kif92b] show that $\mathbf{T}_\Pi$ is monotonic and has a least fixpoint $lfp(\mathbf{T}_\Pi)$. Moreover, they show that Π entails $A : \mu$ iff $\mu \leq lfp(\mathbf{T}_\Pi)(A)$ and hence $lfp(\mathbf{T}_\Pi)$ precisely captures the ground atomic logical consequences of Π.

Thus, we see that any social network $\mathcal{S}$ can be represented as a GAP $\Pi_\mathcal{S}$. We will show (in Section 4) that many existing diffusion models of $\Pi_\mathcal{S}$ can be expressed as a GAP $\Pi \supseteq \Pi_\mathcal{S}$ by adding some GAP-rules describing the diffusion process to $\Pi_\mathcal{S}$.

3. Social Network Optimization (SNOP) Queries

In this section, we develop a formal syntax and semantics for optimization in social networks, taking advantage of the above embedding of SNs into GAPs. We see from queries **(Q1),(Q2)** that a SNOP-query looks for a set $\mathbf{V'}$ of vertices and has the following components: (i) an aggregate operator, (ii) an integer $k \geq 0$, (iii) a set of conditions that each vertex in $\mathbf{V'}$ must satisfy, and (iv) a *goal* atom $g(V)$ where g is a vertex predicate and V is a variable.

Aggregates. It is clear that in order to express queries like **(Q1),(Q2)**, we need aggregate operators which are mappings $agg : \mathsf{FM}([0,1]) \rightarrow \mathbb{R}$ ($\mathbb{R}$ is the set of reals) where $\mathsf{FM}(X)$ denotes the set of all finite multisets that are subsets of X. Relational DB aggregates like SUM,COUNT,AVG,MIN,MAX are all aggregate operators which can take a finite multiset of reals as input and return a single real.

Aggregates may be monotonic or not. We first define a partial ordering $\sqsubseteq$ on multi-sets of numbers as follows. $X_1 \sqsubseteq X_2$ iff there exists an *injective* mapping $\beta : X_1 \rightarrow X_2$ such that $(\forall x_1 \in X_1) x_1 \leq \beta(x_1)$. The aggregate agg is *monotonic (resp. anti-monotonic)* iff whenever $X_1 \sqsubseteq X_2$, it is the case that $agg(X_1) \leq agg(X_2)$ (resp. $agg(X_2) \leq agg(X_1)$).

Vertex condition. A vertex condition is a conjunction VC of annotated vertex atoms containing at most one variable.

Thus, in our example, $male(V) : 1 \wedge adopter(V) : 1$ is a conjunctive vertex condition, but $male(V) : 1 \wedge email(V, V') : 1$ is not. We are now ready to define a SNOP-query.

Definition 3.1 (SNOP-query)**.** A *SNOP-query* is a 4-tuple $(agg, VC, k, g(V))$ where agg is an aggregate, VC is a vertex condition, $k \geq 0$ is an integer, and $g(V)$ is a goal atom.

If we return to our cell phone example, we can set $agg = \mathsf{SUM}$, $k = 3$ (for example), $VC = true$ and the goal to be $adopter(V)$. Here, the goal is to find a set X of annotated ground atoms of the form $adopter(v) : \mu$ such that X's cardinality is 3 or less and such that $\mathsf{SUM}\{\mu \mid adopter(v) : \mu \in X\}$ is maximized. Here, the SUM is applied to a multiset rather than a set.

Definition 3.2 (pre-answer/value)**.** Suppose an SN $\mathcal{S} = (\mathbf{V}, \mathbf{E}, \ell_{vert}, \ell_{edge}, w)$ is embedded in a GAP Π. A *pre-answer* to the SNOP query $Q = (agg, VC, k, g(V))$ w.r.t. Π is any set $\mathbf{V'} \subseteq \mathbf{V}$ such that: (i) $|\mathbf{V'}| \leq k$, (ii) for all vertices $v' \in \mathbf{V'}$, $lfp(\mathbf{T}_{\{\Pi \cup \{g(v'):1\leftarrow \mid v' \in \mathbf{V'}\}}) \models VC[V/v']$. We use $\mathsf{pre\_ans}(Q)$ to denote the set of all pre-answers to query Q.

The *value*, $value(\mathbf{V'})$, of a pre-answer $\mathbf{V'}$ is $agg(\{lfp(\mathbf{T}_{\Pi \cup \{g(v'):1\leftarrow \mid v' \in \mathbf{V'}\}})(g(V)) \mid V \in \mathbf{V}\})$ — here, the aggregate is applied to a multi-set rather than a set. We also note that we

can define *value* as a mapping from interpretations to reals based on a SNOP query. We say $value(I) = agg(\{I(g(v)) \mid v \in \mathbf{V}\})$.

If we return to our cell phone example, $\mathbf{V'}$ is the set of vertices to which the company is considering giving free plans. The value of this set ($value(\mathbf{V'})$) is computed as follows. Find the least fixpoint of $T_{\Pi'}$ where Π' is Π expanded with annotated atoms of the form $adopter(V') : 1$ for each vertex $V' \in \mathbf{V'}$. For each vertex $V \in \mathbf{V}$ (the entire set of vertices, not just $\mathbf{V'}$ now), we now find the confidence assigned by the least fixpoint. Summing up these confidences gives us a measure of the expected number of plan adoptees.

Definition 3.3 (answer)**.** Suppose an SN $\mathcal{S} = (\mathbf{V}, \mathbf{E}, \ell_{vert}, \ell_{edge}, w)$ is embedded in a GAP Π and $Q = (agg, VC, k, g(V))$ is a SNOP-query. A pre-answer $\mathbf{V'}$ is an *answer* to the SNOP-query Q iff the SNOP-query has no other pre-answer $\mathbf{V''}$ such that $value(\mathbf{V''}) > value(\mathbf{V'})$.[2]

The *answer set*, $\mathsf{ans}(Q)$, to the SNOP-query $Q = (agg, VC, k, g(V))$ w.r.t. Π is the set of all answers to Q.

Example 3.4. Consider the GAP Π_{cell} with the social network from Figure 1 embedded and the SNOP-query $Q_{cell} = (SUM, true, 3, will\_adopt)$. The sets $\mathbf{V}'_1 = \{v_{15}, v_{19}, v_6\}$ and $\mathbf{V}'_2 = \{v_{15}, v_{18}, v_6\}$ are both *pre-answers*. In the case of $\mathbf{V}'_1$, two applications of the $\mathbf{T}_\Pi$ operator yield a fixpoint where the vertex atoms formed with *will_adopt* in set $\{v_{15}, v_{19}, v_6, v_{12}, v_{18}, v_7, v_{10}\}$ are annotated with 1. For $\mathbf{V}_2$, only one application of $\mathbf{T}_\Pi$ is required to reach a fixpoint, and the corresponding set of vertices (where the vertex atom formed with *will_adopt* is annotated with 1) is $\{v_{15}, v_6, v_{12}, v_{18}, v_7, v_{10}\}$. As these are the only vertex atoms formed with *will_adopt* that have a non-zero annotation after reaching the fixed point, we know that $value(\mathbf{V}'_1) = 7$ and $value(\mathbf{V}'_2) = 6$. As $value(\mathbf{V}'_1) > value(\mathbf{V}'_2)$, it is easy to see that $\mathbf{V}'_1$ is an answer to this SNOP-query.

Theorem 3.5. *Answering SNOP-queries is NP-Hard.*[3]

Under some reasonable conditions, the problem of answering SNOP-queries is also in NP.

Theorem 3.6. *If both the aggregate function agg and the functions in $\mathcal{F}$ are polynomially computable, then the problem of finding an answer to a SNOP-query is in NP*[4]*.*

Most common aggregate functions like SUM, AVERAGE, Weighted average, MIN, MAX, COUNT are all polynomially computable. Moreover, the assumption that the functions in $\mathcal{F}$ are polynomially computable is also reasonable. The counting problem version of SNOP-query answering seeks to find the number of answers to a SNOP query. Unfortunately, this problem is $\#P$-complete under the same assumptions.

[2]Throughout this paper, we only treat maximization problems - minimizing an objective function f is the same as maximizing $-f$.

[3]**Proof Sketch:** Due to space constraints, we only explain the hardness result by reducing SET COVER to the problem of answering SNOP queries. Given a SET COVER problem instance consisting of a set S, a family $\mathcal{H} = \{H_1, \ldots, H_{max}\}$ of subsets of S, and a positive integer K, we can reduce this problem instance to a SNOP query by polynomially constructing a graph whose vertices correspond to the members of S and to the H_i's - there is an edge from an $s \in S$ to H_i iff $s \in H_i$. All edges have a weight of 1. Every vertex $v \in S$ has an associated propositional symbol *marked* set to "true." There is only one label and all edges are labeled with it and there are no integrity constraints. We have a GAP consisting of one rule $marked(v) : 1 \leftarrow marked(v') : 1 \wedge (v', v, label) : 1$. If we now consider the SNOP-query $(SUM, true, K, marked(v))$, we see that solutions to the SNOP-query (which cause certain H_i's to get marked) correspond precisely to a solution of the SET COVER problem.

[4]By abuse of notation, we refer to the obvious decision problem associated with answering SNOP-queries.

Theorem 3.7. *The counting version of the SNOP query answering problem is #P-complete.*

Although the counting version of the query is $\#P$-hard, finding the *union* of all answers to a SNOP query is no harder than a SNOP query. We shall refer to this problem as *SNOP-ALL* - and it reduces both to and from a regular SNOP query[5].

4. Applying SNOPs to Real Diffusion Problems

In this section, we briefly show how SNOPs may be used to solve two diffusion problems - one each in economics and disease spread.

The Jackson-Yariv Diffusion Model [Jac05]**.** In this framework, a set of agents is associated with each vertex in an undirected graph $\mathbf{G'} = (\mathbf{V'}, \mathbf{E'})$. Each agent has a default behavior (A) and a new behavior (B). Suppose d_i denotes the degree of a vertex v_i. [Jac05] use a function $g : \{0, \ldots, |\mathbf{V}| - 1\} \rightarrow [0, 1]$ to describe how the number of neighbors of v affects the benefits to v for adopting behavior B. For instance, $g(3)$ specifies the benefits (in adopting behavior B) that accrue to an arbitrary vertex $v \in \mathbf{V'}$ that has three neighbors. Let π_i denote the fraction of neighbors of v_i that have adopted behavior B; Let constants b_i and c_i be the benefit and cost for vertex v_i to adopt behavior B, respectively. [Jac05] state that node v_i switches to behavior B iff $\frac{b_i}{c_i} \cdot g(d_i) \cdot \pi_i \geq 1$.

Returning to our cell-phone example, one could potentially use this model to describe the spread of the new plan. In this case, behavior B would be the use of the new plan. The associated SNOP-query would ask to simply find the nodes given a free plan that would maximize use of the plan in the network. Cost and benefit could be computed from factors such as income, time invested in switching plans, etc.

Given a Jackson-Yariv model consisting of $\mathbf{G'} = (\mathbf{V'}, \mathbf{E'})$ and g, we can set up an SN $(\mathbf{V'}, \mathbf{E''}, \ell_{vert}, \ell_{edge}, w)$ as follows. We define $\mathbf{E''} = \{(x, y), (y, x) \mid (x, y) \in \mathbf{E'}\}$. We have a single edge predicate symbol *edge* and ℓ_{edge} assigns 1 to all edges in $\mathbf{E''}$. Our associated GAP Π_{JY} now consists of Π_{SN} plus the single rule:

$$B(V_i) : \lfloor \frac{b_i}{c_i} \cdot g(\sum_j E_j) \cdot \frac{\sum_j X_j}{\sum_j E_j} \rfloor \leftarrow \bigwedge_{V_j | (V_j, V_i) \in \mathbf{E''}} (edge(V_j, V_i) : E_j \wedge B(V_j) : X_j)$$

It is easy to see that this rule (when applied in conjunction with Π_{SN} for a social network SN) precisely encodes the Jackson-Yariv semantics.

The Kempe-Kleinberg-Tardos Framework.[Kem03] If we take the above construction, and for each v_i replace the $\frac{\sum_j X_j}{\sum_j E_j}$ in the head with a monotone *threshold function*, f_i, we have embedded the general framework of [Kem03], of which the [Jac05] model is a special case. It is important to note that the framework of [Kem03] captures a wide variety of diffusion models seen in social sciences and interacting particle systems. These include the "linear threshold model" - which is based on models in social science made popular by [Sch78] and [Gra78] and the "independent cascade model," introduced in [JG01]. However, this work provides a further generalization, as we allow for multiple properties to be "activated" on the vertices, permit labeled edges signifying different relationships, and provide a rule-based

[5]Our proofs of this statement rely on two constructions. First, a regular SNOP query, where the answer must be of size k, can be solved with k successive SNOP-ALL queries. Likewise, a SNOP-ALL query can be answered by solving $|\mathbf{V}|$ SNOP queries. Details are omitted due to lack of space.

framework which can allow for learned diffusion models. Additionally, [Kem03] does not solve SNOP queries with complex aggregates.

The SIR Model of Disease Spread. The SIR (*susceptible, infectious, removed*) model of disease spread [And79] is a classic disease model which labels each vertex in a graph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ (of humans) with *susceptible* if it has not had the disease but can receive it from one of its neighbors, *infectious* if it has caught the disease and t_{rec} units of time have not expired, and *removed* where the vertex can no longer catch or transmit the disease. The SIR model assumes that a vertex v that is infected can transmit the disease to any of its neighbors v' with a probability $p_{v,v'}$ for t_{rec} units of time. We would like to "find k vertices that would maximize the expected number of vertices that become infected". These are obviously good candidates to treat with appropriate medications.

Let $\mathcal{S} = (\mathbf{V}, \mathbf{E}, \ell_{vert}, \ell_{edge}, w)$ be an SN where each edge is labeled with the predicate symbol e and $w(v, v', e) = p_{v,v'}$. We use the predicate *inf* to designate the initially infected vertices. In order to create a GAP Π_{SIR} capturing the SIR model of disease spread, we first define t_{rec} predicate symbols $rec_1, \ldots, rec_{t_{rec}}$ where $rec_i(v)$ intuitively means that node v was infected i days ago. Hence, $rec_{t_{rec}}(v)$ means that v is "removed." We embed $\mathcal{S}$ into GAP Π_{SIR} by adding the following diffusion rules. If $t_{rec} > 1$, we add a non-ground rule for each $i = \{2, \ldots, t_{rec}\}$ - starting with t_{rec}:

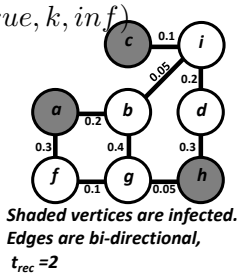
$$\begin{aligned} rec_i(V) : R &\leftarrow rec_{i-1}(V) : R \\ rec_1(V) : R &\leftarrow inf(V) : R \\ inf(V) : (1 - R) \cdot P_{V',V} \cdot (P_{V'} - R') &\leftarrow rec_{t_{rec}}(V) : R \wedge\ e(V', V) : P_{V',V} \ \wedge \\ &\quad\ inf(V') : P_{V'} \ \wedge\ rec_{t_{rec}}(V') : R'. \end{aligned}$$

The first rule says that if a vertex is in its $(i-1)$'th day of recovery with certainty R in the j'th iteration of the $\mathbf{T}_{\Pi_{SIR}}$ operator, then the vertex is i days into recovery (with the same certainty) in the $j+1$'th iteration of the operator. Likewise, second rule intuitively encodes the fact that if a vertex became infected with certainty R in the j'th iteration of the $\mathbf{T}_{\Pi_{SIR}}$ operator, then the vertex is one day into recovery in the $j+1$'th iteration of the operator with the same certainty. The last rule says that if a vertex V' has been infected with probability $P_{V'}$ and there is an edge from V' to V in the social network (weighted with probability $P_{V',V}$), and the vertex V' has recovered with certainty R', given the probability $1-R$ that V is not already recovered, (and hence, cannot be re-infected)[6], then the certainty that the vertex V gets infected is $(1 - R) \cdot P_{V',V} \cdot (P_{V'} - R')$. Here, $P_{V'} - R'$ is one way of measuring the certainty that V' has recovered (difference of the probability that it was infected and the probability it has recovered) and $P_{V',V}$ is the probability of infecting the neighbor.

To see how this GAP works, we execute a few iterations of the $\mathbf{T}_{\Pi_{SIR}}$ operator and show the fixpoint that it reaches on a toy sample graph shown in Figure 2. In this graph, the initial infected vertices are those shown in a shaded circle. The transmission probabilities weight the edges in the graph.

[6]Note that the SIS (Susceptible-Infectious-Susceptible) model [Het76], where an individual becomes susceptible to disease after recovering (as opposed to SIR, where an individual acquires immunity) can be easily represented by a modification to the described construction. Simply change the annotation function in the head of the third rule to $P_{V',V} \cdot (P_{V'} - R')$. In this way, we do not consider the probability that vertex V is immune.

The SNOP-query is $(SUM, true, k, inf)$ to count the number of infected vertices in the least fixpoint of T_Π. This query says "find the k vertices in the social network which, if infected, would cause the maximal number of vertices to become infected in the future." However, the above set of rules can be easily used to express other things. For instance, an epidemiologist may not be satisfied with only one set of k vertices that can cause the disease to spread to the maximum extent - as there may be another, disjoint set of k vertices that could cause the same effect. Note that a single set of vertices may still be sufficient for other applications, such as viral marketing. The epidemiologist may want to find all members of the population, that if in a group of size k could spread the disease to a maximum extent. This can be answered using a *SNOP-ALL* query, described in Section 3.



1	inf(a):1, inf(c):1, inf(d):1
2	rec_1(a):1, rec_1(c):1, rec_1(d):1, inf(b):0.2, inf(d):0.3, inf(f):0.3, inf(g):0.05, inf(i):0.1
3	rec_2(a):1, rec_2(c):1, rec_2(d):1, rec_1(b):0.2, rec_1(d):0.3, rec_1(f):0.3, rec_1(g):0.05, rec_1(i):0.1 inf(g):0.08
4	rec_2(b):0.2, rec_2(d):0.3, rec_2(f):0.3, rec_2(g):0.05, rec_2(i):0.1, rec_1(g):0.08
5	rec_2(g):0.08

Figure 2: Left: Sample network for disease spread. Right: annotated atoms entailed after each application of $\mathbf{T}_{\Pi_{SIR}}$ (maximum, non-zero annotations only).

5. Conclusion

In this paper, we described how General Annotated Logic Programs can be used to represent a variety of diffusion models in social networks. Based on this formulation, we presented the social network optimization problem (SNOP) - which queries the GAP for a set of nodes that cause a given phenomenon to spread through the social network to a maximum extent - shown here to be NP-Complete. We also showed how several well-known diffusion models can be represented in our framework. In future work, we intend to explore heuristic approaches for large sub-classes of SNOPs to answer queries on real-world datasets.

6. Acknowledgments

Some of the authors of this paper were funded in part by AFOSR grant FA95500610405, ARO grant W911NF0910206 and ONR grant N000140910685.

References

[And79] Roy M. Anderson and Robert M. May. Population biology of infectious diseases: Part i. *Nature*, 280(5721):361, 1979.

[Apt03] K. Apt. *Principles of constraint programming*. Cambridge University Press, 2003.

[Cow04] Robin Cowan and Nicolas Jonard. Network structure and the diffusion of knowledge. *Journal of Economic Dynamics and Control*, 28(8):1557 – 1575, 2004. doi:DOI:10.1016/j.jedc.2003.04.002. URL http://www.sciencedirect.com/science/article/B6V85-4B3K2R3-2/2/16e1c8fae6818c11bb45325c9b3ea4a1

[Dam99] C. Damasio, L. Pereira, and T. Swift. Coherent well-founded annotated logic programs. In *Proc. Intl. Conf. on Logic Programming and Non-Monotonic Reasoning*, pp. 262–276. Springer Lecture Notes in Computer Science Vol. 1730, 1999.

[FC08] H. Cruz F.C. Coelho, C. Codeco. Epigrass: A tool to study disease spread in complex networks. *Source Code for Biology and Medicin*, 3(3), 2008.

[Gra78] Mark Granovetter. Threshold models of collective behavior. *The American Journal of Sociology*, 83(6):1420–1443, 1978. doi:10.2307/2778111.
URL `http://dx.doi.org/10.2307/2778111`

[Het76] Herbert W. Hethcote. Qualitative analyses of communicable disease models. *Mathematical Biosciences*, 28(3-4):335 – 356, 1976. doi:DOI:10.1016/0025-5564(76)90132-2.
URL `http://www.sciencedirect.com/science/article/B6VHX-4771JSV-9/2/701b5ad988380270c29b4ab5dcd67bbe`

[Jac05] M. Jackson and L. Yariv. Diffusion on social networks. In *Economie Publique*, vol. 16, pp. 69–82. 2005.

[JG01] E. Muller J. Goldenberg, B. Libai. Talk of the network: A complex systems look at the underlying process of word-of-mouth. *Marketing Letters*, 12(3):211, 2001.

[Kem03] David Kempe, Jon Kleinberg, and Éva Tardos. Maximizing the spread of influence through a social network. In *KDD '03: Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 137–146. ACM, New York, NY, USA, 2003. doi:http://doi.acm.org/10.1145/956750.956769.

[Kif92a] M. Kifer and E. L. Lozinskii. A logic for reasoning with inconsistency. *J. Autom. Reasoning*, 9(2):179–215, 1992.

[Kif92b] Michael Kifer and V.S. Subrahmanian. Theory of generalized annotated logic programming and its applications. *J. Log. Program.*, 12(3&4):335–367, 1992.

[Kra04] Stanislav Krajci, Rastislav Lencses, and Peter Vojts. A comparison of fuzzy and annotated logic programming. *Fuzzy Sets and Systems*, 144(1):173 – 192, 2004. doi:DOI:10.1016/j.fss.2003.10.019.
URL `http://www.sciencedirect.com/science/article/B6V05-49YH3XJ-2/2/1a631467fa197cb0a6f6ae93c1db1a59`

[Leo04] Nicola Leone, Francesco Scarcello, and V.S. Subrahmanian. Optimal models of disjunctive logic programs: Semantics, complexity, and computation. *IEEE Transactions on Knowledge and Data Engineering*, 16:487–503, 2004. doi:http://doi.ieeecomputersociety.org/10.1109/TKDE.2004.1269672.

[Llo87] John W. Lloyd. *Foundations of logic programming*. Springer-Verlag New York, Inc., 1987.

[Lu93] J.J. Lu, N.V. Murray, and E. Rosenthal. Signed formulas and annotated logics. In *Multiple-Valued Logic, 1993., Proceedings of The Twenty-Third International Symposium on*, pp. 48–53. 1993. doi:10.1109/ISMVL.1993.289582.

[Lu96] J. Lu. Logic programs with signs and annotations. *Journal of Logic and Computation*, 6(6):755–778, 1996.

[NE08] A. Pentland N. Eagle and D. Lazer. Mobile phone data for inferring social network structure. In *Proc. 2008 Intl. Conference on Social and Behavioral Computing*, pp. 79–88. Springer Verlag, 2008.

[Ryc08] Jan Rychtář and Brian Stadler. Evolutionary dynamics on small-world networks. *International Journal of Computational and Mathematical Sciences*, 2(1), 2008.
URL `www.waset.org`

[Sch78] Thomas C. Schelling. *Micromotives and Macrobehavior*. W.W. Norton and Co., 1978.

[Thi93] K. Thirunarayan and M. Kifer. A theory of nonmonotonic inheritance based on annotated logic. *Artificial Intelligence*, 60(1):23–50, 1993.

[Ven04] J. Venneksn, S. Verbaeten, and M. Bruynooghe. Logic programs with annotated disjunctions. In *Proc. Intl. Conf. on Logic Programming*, pp. 431–445. Springer Lecture Notes in Computer Science Vol. 3132, 2004.

[Wat99] Duncan J. Watts. Networks, dynamics, and the small-world phenomenon. *The American Journal of Sociology*, 105(2):493–527, 1999.
URL `http://www.jstor.org/stable/2991086`

Technical Communications of the International Conference on Logic Programming, 2010 (Edinburgh), pp. 192–201
http://www.floc-conference.org/ICLP-home.html

ABDUCTIVE INFERENCE IN PROBABILISTIC LOGIC PROGRAMS

GERARDO I. SIMARI AND V.S. SUBRAHMANIAN

Department of Computer Science and UMIACS
University of Maryland College Park
College Park, MD 20742, USA
E-mail address, Gerardo Simari: gisimari@cs.umd.edu
E-mail address, V.S. Subrahmanian: vs@cs.umd.edu

ABSTRACT. Action-probabilistic logic programs (*ap*-programs) are a class of probabilistic logic programs that have been extensively used during the last few years for modeling behaviors of entities. Rules in *ap*-programs have the form "If the environment in which entity E operates satisfies certain conditions, then the probability that E will take some action A is between L and U". Given an *ap*-program, we are interested in trying to change the environment, subject to some constraints, so that the probability that entity E takes some action (or combination of actions) is maximized. This is called the Basic Probabilistic Logic Abduction Problem (Basic PLAP). We first formally define and study the complexity of Basic PLAP and then provide an exact (exponential) algorithm to solve it, followed by more efficient algorithms for specific subclasses of the problem. We also develop appropriate heuristics to solve Basic PLAP efficiently.

1. Introduction

Action probabilistic logic programs (*ap*-programs for short) [Khu07] are a class of the extensively studied family of probabilistic logic programs (PLPs) [Ng92, Ng93, KI04]. *ap*-programs have been used extensively to model and reason about the behavior of groups and an application for reasoning about terror groups based on *ap*-programs has users from over 12 US government entities [Gil08]. *ap*-programs use a two sorted logic where there are "state" predicate symbols and "action" predicate symbols[1] and can be used to represent behaviors of arbitrary entities (ranging from users of web sites to institutional investors in the finance sector) because they consist of rules of the form "if a conjunction C of atoms is true in a given state S, then entity E (the entity whose behavior is being modeled) will take action A with a probability in the interval $[L, U]$."

In this kind of application, it is essential to avoid making probabilistic independence assumptions, since the approach involves *finding out* what probabilistic relationships exist and then exploit these findings in the forecasting effort. For instance, Figure 1 shows a small set of rules automatically extracted from data [Asa08] about Hezbollah's past. Rule 1 says that Hezbollah uses kidnappings as an organizational strategy with probability between 0.5 and 0.56 whenever no political support was provided to it by a foreign state (`forstpolsup`), and the severity of inter-organizational conflict involving it (`intersev1`) is at level "*c*". Rules 2 and 3, also about kidnappings, state that

1998 ACM Subject Classification: I.2.3: Logic Programming, Probabilistic Reasoning.
Key words and phrases: Probabilistic Logic Programming, Imprecise Probabilities, Abductive Inference.

[1]Action atoms only represent the fact that an action is taken, and not the action itself; they are therefore quite different from actions in domains such as AI planning or reasoning about actions, in which effects, preconditions, and postconditions are part of the specification. We assume that effects and preconditions are generally not known.

Technical Communications of the 26th International Conference on Logic Programming, Edinburgh, July, 2010
Editors: Manuel Hermenegildo, Torsten Schaub
LIPIcs - Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/LIPIcs.ICLP.2010.192

r_1. kidnap$(1) : [0.50, 0.56] \leftarrow$ forstpolsup$(0) \wedge$ intersev1(c).
r_2. kidnap$(1) : [0.80, 0.86] \leftarrow$ extsup$(1) \wedge$ demorg(0).
r_3. kidnap$(1) : [0.80, 0.86] \leftarrow$ extsup$(1) \wedge$ elecpol(0).
r_4. tlethciv$(1) : [0.49, 0.55] \leftarrow$ demorg(1).
r_5. tlethciv$(1) : [0.71, 0.77] \leftarrow$ elecpol$(1) \wedge$ intersev2(c).

Figure 1: A small set of rules modeling Hezbollah.

this action will be performed with probability between 0.8 and 0.86 when no external support is solicited by the organization (extsup) and either the organization does not advocate democratic practices (demorg) or electoral politics is not used as a strategy (elecpol). Similarly, Rules 4 and 5 refer to the action "civilian targets chosen based on ethnicity" (tlethciv). The first one states that this action will be taken with probability 0.49 to 0.55 whenever the organization advocates democratic practices, while the second states that the probability rises to between 0.71 and 0.77 when electoral politics are used as a strategy and the severity of inter-organizational conflict (with the organization with which the second highest level of conflict occurred) was not negligible" (intersev2). *ap*-programs have been used extensively by terrorism analysts to make predictions about terror group actions [Gil08, Man08].

Suppose, rather than predicting what action(s) a group would take in a given situation or environment, we want to determine what *we can do* in order to induce a given behavior by the group. For example, a policy maker might want to understand what we can do so that a given goal (*e.g.*, the probability of Hezbollah using kidnappings as a strategy is below some percentage) is achieved, given some constraints on what is feasible. The *probabilistic logic abduction problem* (PLAP) deals with finding how to *reach* a new (feasible) state from the current state such that the *ap*-program associated with the group and the new state jointly entail that the goal will be true within a given probability interval.

In this paper, we first briefly recall *ap*-programs and then formulate PLAP theoretically. We then develop a host of complexity results for PLAP under varying assumptions. We then describe both exact and heuristic algorithms to solve the PLAP problem. We briefly describe a prototype implementation and experiments showing that our algorithm is feasible to use even when the *ap*-program contains hundreds of rules. A brief note on related work before we begin; almost all past work on abduction in such settings have been devised under various independence assumptions [Poo97, Poo93]. We are aware of no work to date on abduction in possible worlds-based probabilistic logic systems such as those of [Hai84], [Nil86], and [Fag90] where independence assumptions are not made.

2. Preliminaries

We now overview the syntax and semantics of *ap*-programs from [Khu07]. We assume the existence of a logical alphabet that consists of a finite set $\mathcal{L}_{cons}$ of constant symbols, a finite set $\mathcal{L}_{pred}$ of predicate symbols (each with an associated arity) and an infinite set $\mathcal{L}_{var}$ of variable symbols; function symbols are not allowed. Terms, atoms, and literals are defined in the usual way [Llo87]. We assume that $\mathcal{L}_{pred}$ is partitioned into disjoint sets: $\mathcal{L}_{act}$ of *action symbols* and $\mathcal{L}_{sta}$ of *state* symbols. Thus, if $t_1, \ldots, t_n$ are terms, and p is an n-ary action (resp. state) symbol, then $p(t_1, \ldots, t_n)$, is called an *action (resp. state) atom*.

Definition 2.1. A (ground) action formula is defined as: *(i)* a (ground) action atom is a (ground) action formula; *(ii)* if F and G are (ground) action formulas, then $\neg F$, $F \wedge G$, and $F \vee G$ are also (ground) action formulas.

The set of all possible action formulas is denoted by *formulas*$(B_{\mathcal{L}_{act}})$, where $B_{\mathcal{L}_{act}}$ is the Herbrand base associated with $\mathcal{L}_{act}$, $\mathcal{L}_{cons}$, and $\mathcal{L}_{var}$.

Definition 2.2. If F is an action formula and $\mu = [\alpha, \beta] \subseteq [0, 1]$, then $F : \mu$ is called an *annotated action formula* (or *ap*-formula), and μ is called the *ap*-annotation of F.

Definition 2.3. A *world* is any finite set of ground action atoms. A *state* is any finite set of ground state atoms.

It is assumed that all actions in the world are carried out more or less in parallel and at once, given the temporal granularity adopted along with the model. Contrary to (related but essentially different) approaches such as stochastic planning, we are not concerned here with reasoning about the effects of actions. We now define *ap*-rules.

Definition 2.4. If F is an action formula, $B_1, \ldots, B_n$ are state atoms, and μ is an *ap*-annotation, then $F : \mu \leftarrow B_1 \ \wedge \ \ldots \wedge \ B_m$ is called an ap-*rule*. If this rule is named r, then *Head*(r) denotes $F : \mu$ and *Body*(r) denotes $B_1 \ \wedge \ \ldots \wedge \ B_n$.

Intuitively, the rule specified above says that if $B_1, \ldots, B_m$ are all true in a given state, then there is a probability in the interval μ that the action combination F is performed by the entity modeled by the *ap*-rule.

Definition 2.5. An *action probabilistic logic program* (*ap*-program for short) is a finite set of *ap*-rules. An *ap*-program Π' such that $\Pi' \subseteq \Pi$ is called a *subprogram* of Π.

Figure 1 shows a small portion of an *ap*-program we derived automatically to model Hezbollah's actions. Henceforth, we use *Heads*(Π) to denote the set of all annotated formulas appearing in the head of some rule in Π. Given a ground *ap*-program Π, we will use *sta*(Π) (resp., *act*(Π)) to denote the set of all state (resp., action) atoms that appear in Π.

Example 2.6. Coming back to the *ap*-program in Figure 1, the following are examples of worlds:

$$\{\texttt{kidnap}(1)\}, \{\texttt{kidnap}(1), \texttt{tlethciv}(1)\}, \{\}$$

The following are examples of states:

$$\{\texttt{forstpolsup}(0), \texttt{elecpol}(0)\}, \{\texttt{extsup}(1), \texttt{elecpol}(1)\}, \{\texttt{demorg}(1)\}.$$

We use $\mathcal{W}$ to denote the set of all possible worlds, and $\mathcal{S}$ to denote the set of all possible states. It is clear what it means for a state to satisfy the body of a rule [Llo87].

Definition 2.7. Let Π be an *ap*-program and s a state. We say that s *satisfies* the body of a rule $F : \mu \leftarrow B_1 \ \wedge \ \ldots \wedge \ B_m$ if and only if $\{B_1, \ldots, B_M\} \subseteq s$.

Similarly, we define what it means for a world to satisfy a ground action formula:

Definition 2.8. Let F, F_1, F_2 be ground action formulas and w a world. We say that w *satisfies* F if and only if:
– if $F \equiv a$, for some atom $a \in B_{\mathcal{L}_{act}}$, then $a \in w$;
– if $F \equiv F_1 \wedge F_2$, then w satisfies F_1 and w satisfies F_2;
– if $F \equiv F_1 \vee F_2$, then w satisfies F_1 or w satisfies F_2;
– if $F \equiv \neg F'$, for some action formula $F' \in$ *formulas*$(B_{\mathcal{L}_{act}})$, then w does not satisfy F'.

Finally, we will use the concept of *reduction* of an *ap*-program w.r.t. a state:

Definition 2.9. Let Π be an *ap*-program and s a state. The *reduction of* Π *w.r.t.* s, denoted Π_s, is the set $\{F : \mu \mid s \text{ satisfies } \textit{Body} \text{ and } F : \mu \leftarrow \textit{Body} \text{ is a ground instance of a rule in } \Pi\}$. Rules in this set are said to be *relevant* in state s.

The semantics of *ap*-programs uses possible worlds in the spirit of [Hai84, Nil86, Fag90]. Given an *ap*-program Π and a state s, we can define a set $LC(\Pi, s)$ of linear constraints associated with s. Each world w_i expressible in the language $\mathcal{L}_{act}$ has an associated variable v_i denoting the probability that it will actually occur. $LC(\Pi, s)$ consists of the following constraints.
(1) For each $\textit{Head}(r) \in \Pi_s$ of the form $F : [\ell, u]$, we have constraint $\ell \leq \sum_{w_i \in \mathcal{W} \wedge w_i \models F} v_i \leq u$.
(2) $LC(\Pi, s)$ contains the constraint $\sum_{w_i \in \mathcal{W}} v_i = 1$.
(3) All variables are non-negative.
(4) $LC(\Pi, s)$ contains only the constraints described in $1 - 3$.
While [Khu07] provides a more formal model theory for *ap*-programs, we merely provide the definition below. Π_s is *consistent* iff $LC(\Pi, s)$ is solvable over $\mathbb{R}$.

Definition 2.10. Let Π be an *ap*-program, s a state, and $F : [\ell, u]$ a ground action formula. Π_s *entails* $F : [\ell, u]$, denoted $\Pi_s \models F : [\ell, u]$ iff $[\ell', u'] \subseteq [\ell, u]$ where:
$\ell' =$ **minimize** $\sum_{w_i \in \mathcal{W} \wedge w_i \models F} v_i$ **subject to** $LC(\Pi, s)$.
$u' =$ **maximize** $\sum_{w_i \in \mathcal{W} \wedge w_i \models F} v_i$ **subject to** $LC(\Pi, s)$.

The following is an example of $LC(\Pi, s)$ and entailment of an *ap*-formula.

Example 2.11. Consider *ap*-program Π from Figure 1 and state s_2 from Figure 2. The set of possible worlds is as follows: $w_0 = \{\}$, $w_1 = \{\texttt{kidnap}(1)\}$, $w_2 = \{\texttt{tlethciv}(1)\}$, and $w_3 = \{\texttt{kidnap}(1), \texttt{tlethciv}(1)\}$. Suppose we use p_i to denote the variable associated with the probability of world w_i; $LC(\Pi, s_2)$ then consists of the following constraints:

$$0.5 \leq p_1 + p_3 \leq 0.56$$
$$0.49 \leq p_2 + p_3 \leq 0.55$$
$$p_0 + p_1 + p_2 + p_3 = 1$$

One possible solution to this set of constraints is $p_0 = 0$, $p_1 = 0.51$, $p_2 = 0.05$, and $p_3 = 0.44$; another possible distribution is $p_0 = 0.5$, $p_1 = 0$, $p_2 = 0$, and $p_3 = 0.5$; yet another one is $p_0 = 0$, $p_1 = 0.45$, $p_2 = 0.11$, and $p_3 = 0.44$. Finally, formula $\texttt{kidnap}(1) \wedge \texttt{tlethciv}(1)$ (satisfied only by world w_3) is entailed with probability in the interval $[0, 0.55]$, meaning that one cannot assign a probability greater than 0.55 to this formula (this example shows that, contrary to what one might think, the interval $[0, 1]$ is not necessarily a solution).

Note that representing a set of distributions is not possible in many other approaches to probabilistic reasoning, such as Bayesian networks [Pea88], Poole's Independent Choice Logic [Poo97] and related formalisms such as [Poo93]. However, this is a key capability for our approach, as we specifically require a formalism that is not forced to make assumptions about the probabilistic dependence (or independence) of the events we are reasoning about. On the other hand, it is certainly possible to extend our approach in such a way that the key aspects of Bayesian networks and related formalisms are directly expressible, as was shown in [Ng93] when probabilistic logic programs were introduced.

$s_1 = \{\texttt{forstpolsup}(0), \texttt{intersev1}(c), \texttt{intersev2}(0), \texttt{elecpol}(1), \texttt{extsup}(0), \texttt{demorg}(0)\}$
$s_2 = \{\texttt{forstpolsup}(0), \texttt{intersev1}(c), \texttt{intersev2}(0), \texttt{elecpol}(0), \texttt{extsup}(0), \texttt{demorg}(1)\}$
$s_3 = \{\texttt{forstpolsup}(0), \texttt{intersev1}(c), \texttt{intersev2}(0), \texttt{elecpol}(0), \texttt{extsup}(0), \texttt{demorg}(0)\}$
$s_4 = \{\texttt{forstpolsup}(1), \texttt{intersev1}(c), \texttt{intersev2}(c), \texttt{elecpol}(1), \texttt{extsup}(1), \texttt{demorg}(0)\}$
$s_5 = \{\texttt{forstpolsup}(0), \texttt{intersev1}(c), \texttt{intersev2}(c), \texttt{elecpol}(0), \texttt{extsup}(1), \texttt{demorg}(0)\}$

Figure 2: A small set of possible states

3. The Probabilistic Logic Abduction Problem

Suppose s is a state (the current state), G is a goal (an action formula), and $[\ell, u] \subseteq [0, 1]$ is a probability interval. The basic PLAP problem requires finding a new state s' such that $\Pi_{s'}$ entails $G : [\ell, u]$. However, s' must be *reachable* from s. For this, we merely assume the existence of a reachability predicate $reach$ specifying *direct* reachability from one state to another. $reach^*$ is the reflexive transitive closure of $reach$ and $unReach$ is its complement. We will investigate, in Section 4.2 below, one way in which $reach$ can be specified; when available, knowledge of action effects and preconditions can be encoded into this predicate.

Example 3.1. Suppose, for simplicity, that the only state predicate symbols are those that appear in the rules of Figure 1, and consider the set of states in Figure 2. Then, some examples of reachability are the following: $reach(s_1, s_2)$, $reach(s_1, s_3)$, $reach(s_2, s_1)$, $reach(s_4, s_1)$, $\neg reach(s_2, s_5)$, and $\neg reach(s_3, s_5)$. Note that, if state s_5 is reachable, then the *ap*-program is inconsistent, since both rules 1 and 2 are relevant in that state.

We can now state the Basic PLAP problem formally:
Basic PLAP Problem.
Input: An *ap*-program Π, a state s, a reachability predicate $reach$ and a ground *ap*-formula $G : [\ell, u]$.
Output: "Yes" if there exists a state s' such that $reach^*(s, s')$ and $\Pi_{s'} \models G : [\ell, u]$, and "No" otherwise.

Example 3.2. Consider once again the program in the running example and the set of states from Figure 2. If the goal is $kidnap(1) : [0, 0.6]$ (we want the probability of Hezbollah using kidnappings to be at most 0.6) and the current state is s_4, then the problem is solvable because Example 3.1 shows that state s_1 can be reached from s_4, and $\Pi_{s_1} \models kidnap(1) : [0, 0.6]$.

The following result shows the intractability of Basic PLAP in the general case.

Proposition 3.3. *Basic PLAP is EXPTIME-complete.*

Moreover, this problem is likely to be intractable even under simplifying assumptions.

Proposition 3.4. *Let $\mathcal{L}_{sta}$ be such that $|\mathcal{L}_{sta}| \leq c'$ for some constant $c' \in \mathbb{N}$; the Basic PLAP problem under this assumption is NP-hard.*

Proposition 3.5. *Let $\mathcal{L}_{act}$ be such that $|\mathcal{L}_{act}| \leq c'$ for some constant $c' \in \mathbb{N}$; the Basic PLAP problem under this assumption is NP-hard.*

The above results reveal that the complexity of PLAP is caused by two factors. (P1) We need to find a subprogram Π' of Π such that when the body of all rules in that subprogram is deleted, the resulting subprogram entails the goal, and (P2) Decide if there exists a state s' such that $\Pi' = \Pi_s$ and s is reachable from the initial state.

4. Algorithms for PLAP

In this section, we leverage the above intuition to develop an algorithm for PLAP under the assumption that all goals are of the form $F : [0, u]$ (ensure that F's probability is less than or equal to u) or $F : [\ell, 1]$ (ensure that F's probability is at least ℓ). Finally, we develop a heuristic algorithm.

4.1. Answering Threshold Goals

A *threshold goal* is an annotated action formula of the form $F : [0, u]$ or $F : [\ell, 1]$. In this section, we study how we can devise a better algorithm for Basic PLAP when only threshold goals are considered. This is a reasonable approach, since threshold goals can be used to express the desire that certain formulas (actions) should only be entailed with a certain maximum probability (upper bound) or should be entailed with at least a certain minimum probability (lower bound). The tradeoff lies in the fact that we lose the capacity to express both desires at once. We start by inducing equivalence classes on subprograms that limit the search space, helping address problem P1.

Definition 4.1. Let Π be a ground *ap*-program and F be a ground action formula. We say that subprograms $\Pi_1, \Pi_2 \subseteq \Pi$ are *equivalent* given F, written $\Pi_1 \sim_F \Pi_2$, iff $\Pi_1 \models F : [\ell, u] \Leftrightarrow \Pi_2 \models F : [\ell, u]$ for any $\ell, u \in [0, 1]$. Furthermore, states s_1 and s_2 are *equivalent* given F, written $s_1 \sim_F s_2$, iff $reach(s_1, s_2)$, $reach(s_2, s_1)$, and $\Pi_{s_1} \sim_F \Pi_{s_2}$.

Example 4.2. Let Π be the *ap*-program from Figure 1, formula $F = \texttt{kidnap}(1)$, $\Pi_1 = \{r_1\}$, $\Pi_2 = \{r_2, r_3\}$ $\Pi_3 = \{r_1, r_4\}$ $\Pi_4 = \{r_1, r_5\}$, and $\Pi_5 = \{r_2, r_3, r_5\}$. Here, $\Pi_1 \sim_F \Pi_3$, $\Pi_1 \sim_F \Pi_4$, $\Pi_3 \sim_F \Pi_4$, and $\Pi_2 \sim_F \Pi_5$. For instance, we can see that $\Pi_1 \sim_F \Pi_3$ because the probability with which $\texttt{kidnap}(1)$ is entailed is given by rule r_1; rule r_4 is immaterial in this case. Clearly, $\Pi_1 \not\sim_F \Pi_2$ since F is entailed with different probabilities in each case.

Next, consider the states from Figure 2 and the reachability predicate from Example 3.1. Since we have that $reach(s_1, s_2)$, $reach(s_2, s_1)$, Π_1 is relevant in s_1, and Π_3 is relevant in s_2, we can conclude that $s_1 \sim_F s_2$.

Relation $\sim$, both between states and between subprograms, is clearly an equivalence relation. The following lemma specifies a way to construct equivalence classes.

Lemma 4.3. *Let Π be an* ap*-program and G be an action formula. Consider two subprograms $\Pi', \Pi'' \subseteq \Pi$ such that $\Pi' = \Pi_a \cup \Pi'_p$ (resp., $\Pi'' = \Pi_a \cup \Pi''_p$), where Π_a is a set of rules whose heads have formulas F such that $F \wedge G \not\models \bot$ and Π'_p (resp., Π''_p) contains rules whose heads have formulas H such that $H \wedge G \models \bot$. Then, $\Pi' \sim_G \Pi''$.*

Lemma 4.4. *Let Π be a consistent* ap*-program and $G : [\ell_G, u_G]$ be a threshold goal. If there exists a rule $r \in \Pi$ such that Head$(r) = F : [\ell_F, u_F]$ and: either (1) if $u_G = 1$, $F \models G$, and $\ell_G \leq \ell_F$; or (2) if $\ell_G = 0$, $G \models F$, and $u_G \geq u_F$; then, $\Pi \models G : [\ell_G, u_G]$.*

The algorithm in Figure 3 first tries to leverage Lemma 4.4 and only proceeds if this is not possible. The way in which the algorithm partitions Π is partly based on Lemma 4.3.

Proposition 4.5. *Given an* ap*-program Π, a state $s \in \mathcal{S}$, and an annotated action formula $G : [\ell, u]$, Algorithm simpleAnnPLAP correctly computes a solution to Basic PLAP. Its worst case running time is in $O\left(2^{|\Pi|} + 2^{|\mathcal{L}_{sta}|} + 2^{|\mathcal{L}_{act}|}\right)$.*

We now present an example of how this algorithm works.

Algorithm 1: simpleAnnPLAP($\Pi, s, G : [\ell_G, u_G]$)

(1) Select rules of the form $r : F : [\ell_r, u_r] \leftarrow s_1 \wedge \ldots \wedge s_n$ such that $F \wedge G \not\models \bot$; call all such rules *active rules*, and the complement set *passive rules*, denoted *active*$(\Pi, G : [\ell_G, u_G])$ and *passive*$(\Pi, G : [\ell_G, u_G])$.
(2) If Lemma 4.4 is applicable, return *true* if there exists a consistent $\Pi' \subseteq$ *candAct*$(\Pi, G : [\ell_G, u_G]) \cup$ *passive*$(\Pi, G : [\ell_G, u_G])$ such that:
  (a) if $u_G = 1$, then at least one rule $r \in \Pi'$ must have head $F : [\ell_F, u_F]$ such that $F \models G$ and $\ell_G \leq \ell_F$; if $\ell_G = 0$), at least one rule $r \in \Pi'$ must have head $F : [\ell_F, u_F]$ such that $G \models F$ and $u_G \geq u_F$;
  (b) state s' for which $\Pi_{s'} = \Pi'$ is such that $reach^*(s, s')$.
(3) Otherwise, for each rule $r_i : F : [\ell_r, u_r] \leftarrow s_1 \wedge \ldots \wedge s_n$ do:
  (a) If $\ell_G = 0$, $F \models G$, and $\ell_r > u_G$ then add r_i to set *conf*$(\Pi, G : [\ell_G, u_G])$
  (b) Otherwise (*i.e.*, $u_G = 1$), if $G \models F$ and $u_r < \ell_G$ then add r_i to set *conf*$(\Pi, G : [\ell_G, u_G])$.
(4) Let *candAct*$(\Pi, G : [\ell_G, u_G])$ = *active*$(\Pi, G : [\ell_G, u_G]) \setminus$ *conf*$(\Pi, G : [\ell_G, u_G])$;
(5) Consider the set *candAct*$(\Pi, G : [\ell_G, u_G]) \cup$ *passive*$(\Pi, G : [\ell_G, u_G])$ and, for each pair of rules of the form $r_i : F_i : [\ell_{r_i}, u_{r_i}] \leftarrow s_1^i \wedge \ldots \wedge s_n^i$ and $r_j : F_j : [\ell_{r_j}, u_{r_j}] \leftarrow s_1^j \wedge \ldots \wedge s_m^j$ such that $F_i : [\ell_{r_i}, u_{r_i}]$ and $F_j : [\ell_{r_j}, u_{r_j}]$ are mutually inconsistent, add the pair (r_i, r_j) to a set called *inc*(Π).
(6) Return *true* if there exists a set of rules $\Pi' \subseteq$ *candAct*$(\Pi, G : [\ell_G, u_G]) \cup$ *passive*$(\Pi, G : [\ell_G, u_G])$ such that $\Pi' \cap$ *candAct*$(\Pi, G : [\ell_G, u_G]) \neq \emptyset$, no pair $\{r_1, r_2\} \subseteq \Pi'$ belongs to *inc*(Π), and:
  (a) $\Pi' \models G : [\ell_G, u_G]$;
  (b) $\exists$ state s' for which $\Pi_{s'} = \Pi'$ such that $reach^*(s, s')$;
(7) If Step 6 is not possible, return *false*;

Figure 3: An algorithm to solve Basic PLAP assuming a threshold goal.

Example 4.6. Suppose Π is the *ap*-program of Figure 1, the goal is $\texttt{kidnap}(1) : [0, 0.6]$ (abbreviated with $G : [0, 0.6]$ from now on) and the state is that of Example 3.2, $s_{curr} = \{\texttt{forstpolsup}(1), \texttt{intersev1}(c), \texttt{intersev2}(c), \texttt{elecpol}(1), \texttt{extsup}(1), \texttt{demorg}(0)\}$; note that $\Pi_{s_{curr}} = \{r_2, r_5\}$ and that clearly $\Pi_{s_{curr}} \not\models \texttt{kidnap}(1) : [0, 0.6]$. Step 1 of simpleAnnPLAP is simple in this case, since all the heads of rules in Π are atomic – therefore *passive*$(\Pi_{s_{curr}}, G : [0, 0.6]) = \emptyset$, and the set of active rules contains all the rules in Π. The following step checks for the applicability of Lemma 4.4; clearly rule r_1 satisfies the conditions and we only need to verify that some subprogram containing it is reachable. Assuming the same reachability predicate outlined in Example 3.1, $s_1 = \{\texttt{forstpolsup}(0), \texttt{intersev1}(c), \texttt{intersev2}(0), \texttt{elecpol}(1), \texttt{extsup}(0), \texttt{demorg}(0)\}$ is reachable from s_{curr}; this corresponds to choosing subprogram $\Pi' = \{r_1\}$. The only other possibilities are to make both r_1 and r_4, or r_1 and r_5 relevant.

4.2. An Improved PLAP Algorithm

In this section, we show that if we assume reachability/unreachability is specified in a syntactic manner rather than in a very general manner as presented earlier, we can come up with some good heuristics to solve Basic PLAP.

Definition 4.7. Let F and G be first-order formulas over $\mathcal{L}_{sta}$ and $\mathcal{L}_{var}$ with connectives $\wedge$, $\vee$, and $\neg$, and such that the set of variables over F is equal to those over G; all variables are assumed to be universally quantified with scope over both F and G. A *reachability constraint* is of the form $F \not\rightsquigarrow G$; we call F the antecedent and G the consequent of the constraint, and its semantics is:

$$unReach(s_1, s_2) \Leftrightarrow s_1 \models F \text{ and } s_2 \models G$$

where s_1 and s_2 are states in $\mathcal{S}$.

Algorithm 2: simpleAnnPLAP-Heur-RC($\Pi, s, G : [\ell_G, u_G], RC$)

(1) Execute Steps 1, 3, 4, and 5 of *simpleAnnPLAP*
(2) Let *goalState*, *goalStateAct*, *goalStateConf*, and *goalStateInf* be logical formulas over $\mathcal{L}_{sta}$ and $\mathcal{L}_{var}$;
(3) Initialize *goalState* to null, *goalStateAct* to $\bot$, and *goalStateConf*, *goalStateInc* to $\top$;
(4) for each rule $r_i \in$ *candAct*$(\Pi, G : [\ell_G, u_G])$ with *Head*$(r_i) = F : [\ell_F, u_F]$ do
  if $\big[(u_G = 1)$ and $(F \models G$ and $\ell_G \leq \ell_F)\big]$ or $\big[(\ell_G = 0)$ and $(G \models F$ and $u_G \geq u_F)\big]$
    then set *goalStateAct* := *goalStateAct* $\vee$ *getStateFormula*(r_i);
(5) for each rule $r_i \in$ *conf*$(\Pi, G : [\ell_G, u_G])$ do
  set *goalStateConf* := *goalStateConf* $\wedge \neg$*getStateFormula*(r_i);
(6) for each pair of rules $(r_i, r_j) \in$ *inc*(Π) do
  set *goalStateInc* := *goalStateInc* $\wedge \neg$(*getStateFormula*$(r_i) \wedge$ *getStateFormula*(r_j));
(7) set *goalState* := *goalStateAct* $\wedge$ *goalStateConf* $\wedge$ *goalStateInc*;
  // *goalState* describes the states that satisfy the goal
(8) return *decideReachability*$(s,$ *goalState*$, RC)$;

Figure 4: A heuristic algorithm based on Lemma 4.4 to solve Basic PLAP assuming threshold goals and that state reachability is expressed as a set RC of reachability constraints.

Reachability constraints simply state that if the antecedent is satisfied in a certain state, then no states that satisfy the consequent are reachable from it. We now present an example of a set of reachability constraints.

Example 4.8. Consider again the setting and *ap*-program from Figure 1. The following are examples of reachability constraints[2]:

$$\texttt{forstpolsup}(1) \not\rightarrow \texttt{intersev1}(c)$$
$$\texttt{intersev1}(c) \vee \texttt{intersev2}(c)) \wedge \texttt{demorg}(0) \not\rightarrow \texttt{demorg}(1)$$

Algorithm *simpleAnnPLAP-Heur-RC* (Figure 4) takes advantage of the structure added by the presence of reachability constraints. The algorithm starts out by executing the steps of *simpleAnnPLAP* that compute the sets *active*, *passive*, *candAct*, *conf*, and *inc*. It then builds formulas generated by reachability constraints that any solution state must satisfy; the algorithm uses a subroutine *formula*(s) which returns a formula that is a conjunction of all the atoms in state s and the negations of those not in s. In Step 4, the formula describes the fact that at least one of the states that make relevant "candidate active" rules (as described in Algorithm *simpleAnnPLAP*) must be part of the solution; similarly, Step 5 builds a formula ensuring that none of the conflicting active rules can be relevant if the problem is to have a solution. Finally, Step 6 describes the constraints associated with making relevant rules that are probabilistically inconsistent. Noticeably absent are the "passive" rules from the previous algorithm; such rules impose no constraints on the solution. The last two steps put subformulas together into a conjunction of constraints, and the algorithm must decide if there exist any states that model formula *goalState* and are eventually reachable from s. Eventual reachability can be decided by means of a *SAT-based* method as follows: if the current state does not satisfy *goalState*, it starts by initializing formula *Reachable* which will be used to represent the set of eventually reachable states at each step. The initial formula describes state s, and the algorithm then proceeds to select all the constraints whose antecedents are entailed by *Reachable*. Once we have this set, *Reachable* is updated to the conjunction of the negations of all the consequents of constraints in the set. We are done whenever either *Reachable* models *goalState*, or the old version of *Reachable* is modeled by the new one (no new reachable states exist).

[2]If available, knowledge of action effects and preconditions can be represented with similar constraints.

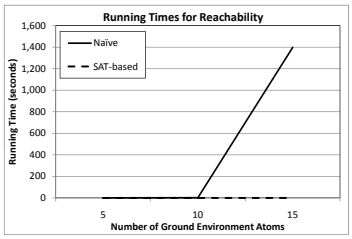


Figure 5: 5 rules, 25 ground action atoms, 5 reachability constraints, and atomic queries.

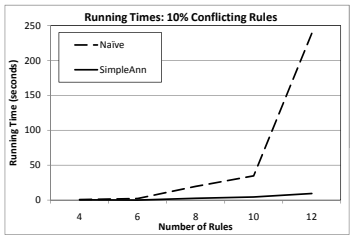


Figure 6: 10% goal-conf. rules, 25 action atoms, 5 state atoms (ground), and atomic queries.

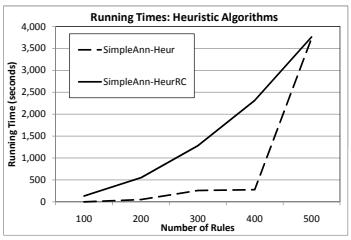


Figure 7: Larger *ap*-programs; 25 action atoms, 5 state atoms (ground), and 5 reach. constr.

5. Experimental Results

We conducted experiments using a prototype JAVA implementation consisting of roughly 2,500 lines of code. All experiments were run on multiple multi-core Intel Xeon E5345 processors at 2.33GHz, 8GB of memory, running the Scientific Linux distribution of the GNU/Linux operating system, kernel version 2.6.9-55.0.2.ELsmp.[3] Numbers reported are averages over at least 20 runs.

No. of State Atoms. In Figure 5 we show the running times of the different approaches to deciding reachability; the naive approach becomes intractable very quickly, while the (still exact) SAT-based algorithm approach has negligible cost for these runs.

No. of Rules. Figure 6 reports the running times of the SimpleAnn rule selection algorithm vs. the naive approach for programs in which 10% of the rules were forced to be in probabilistic conflict with the goal. This experiment shows how SimpleAnn leverages the presence of these rules, greatly reducing its running time w.r.t. that of the naive algorithm.

Finally, Figure 7 shows the running times for the SimpleAnn heuristic step (that is, assuming the algorithm only tries to apply the heuristic and pessimistically returns *false* otherwise) and the SimpleAnn-HeurRC algorithm for larger programs. It is interesting to see the different shapes of the curves: as programs get larger, the SAT formulas associated with SimpleAnn-HeurRC become larger as well, leading to the gradual increase in the running time; on the other hand, we can see that the strategy of only focusing on certain "heuristic rules" pays off for the SimpleAnn heuristic step, but there is a spike in running time when the size grows from 400 to 500 rules. This is likely due to the appearance of more such rules, which means that the algorithm has many more subprograms to verify for entailment of the goal.

6. Related Work and Conclusions

Abduction has been extensively studied in diagnosis [Con91], reasoning with non-monotonic logics [Eit95], probabilistic reasoning [Pea91, Poo97], argumentation [Koh02], planning [Esh88], and temporal reasoning [Esh88]; furthermore, it has been combined quite naturally with different variants of logic programs [Den02]. David Poole *et al*. combined probabilistic and non-monotonic reasoning, leading to the development of the Independent Choice Logic [Poo97]. Though this model is related to our work, it makes general assumptions of pairwise independence of probabilties of events; other related models are based on the class of graphical models including Bayesian Networks

[3]We note that this implementation makes use of only one processor and one core.

(BNs). The main difference between graphical model-based work and our work is that we *make no assumptions on the dependence or independence of probabilities of events*.

While AI planning may seem relevant, there are several differences. First, we are not assuming knowledge of the effects of actions; second, we assume the existence of a probabilistic model underlying the behavior of the entity being modeled. In this framework, we want to find a state such that *when the atoms in the state are added to the* ap*-program, the resulting combination* entails *the desired goal with a given probability.* While the italicized component of the previous sentence can be achieved within planning, it would require a state space that is exponentially larger than the one we use (the search space would be the set of all sets of atoms that are jointly entailed by any subprogram of the *ap*-program and any state).

To the best of our knowledge, this is the first paper that tackles the problem of abductive reasoning in probabilistic logic programming under no independence assumptions, in the tradition of [Ng92] for probabilistic logic programming, and [Hai84, Nil86, Fag90] for probabilistic logic.

Acknowledgements. The authors were funded in part by AFOSR grant FA95500610405 and ARO grant W911NF0910206.

References

[Asa08] V Asal, J Carter, and J Wilkenfeld. Ethnopolitical violence and terrorism in the middle east. In J Hewitt, J Wilkenfeld, and T Gurr (eds.), *Peace and Conflict 2008*. Paradigm, Boulder, CO, 2008.

[Con91] Luca Console and Pietro Torasso. A spectrum of logical definitions of model-based diagnosis. *Comput. Intell.*, 7(3):133–141, 1991.

[Den02] Marc Denecker and Antonis C. Kakas. Abduction in logic programming. In *Computational Logic: Logic Programming and Beyond, Part I*, pp. 402–436. Springer-Verlag, London, UK, 2002.

[Eit95] Thomas Eiter and Georg Gottlob. The complexity of logic-based abduction. *J. ACM*, 42(1):3–42, 1995.

[Esh88] Kave Eshghi. Abductive planning with event calculus. In *ICLP/SLP*, pp. 562–579. 1988.

[Fag90] Ronald Fagin, Joseph Y. Halpern, and Nimrod Megiddo. A logic for reasoning about probabilities. *Information and Computation*, 87(1/2):78–128, 1990.

[Gil08] Jim Giles. Can conflict forecasts predict violence hotspots? *New Scientist*, (2647), 2008.

[Hai84] T. Hailperin. Probability logic. *Notre Dame Journal of Formal Logic*, 25 (3):198–212, 1984.

[Khu07] Samir Khuller, Maria Vanina Martinez, Dana S. Nau, Amy Sliva, Gerardo I. Simari, and V. S. Subrahmanian. Computing most probable worlds of action probabilistic logic programs: scalable estimation for 10^30,000 worlds. *AMAI*, 51(2-4):295–331, 2007.

[KI04] Gabriele Kern-Isberner and Thomas Lukasiewicz. Combining probabilistic logic programming with the power of maximum entropy. *Artif. Intell.*, 157(1-2):139–202, 2004.

[Koh02] J. Kohlas, D. Berzati, and R. Haenni. Probabilistic argumentation systems and abduction. *AMAI*, 34(1-3):177–195, 2002.

[Llo87] J. W. Lloyd. *Foundations of Logic Programming, Second Edition*. Springer-Verlag, 1987.

[Man08] A. Mannes, M. Michael, A. Pate, A. Sliva, V. S. Subrahmanian, and J. Wilkenfeld. Stochastic opponent modelling agents: A case study with Hezbollah. In Huan Liu and John Salerno (eds.), *Proc. of IWSCBMP*. 2008.

[Ng92] Raymond T. Ng and V. S. Subrahmanian. Probabilistic logic programming. *Information and Computation*, 101(2):150–201, 1992.

[Ng93] Raymond T. Ng and V. S. Subrahmanian. A semantical framework for supporting subjective and conditional probabilities in deductive databases. *J. Autom. Reasoning*, 10(2):191–235, 1993.

[Nil86] Nils Nilsson. Probabilistic logic. *Artificial Intelligence*, 28:71–87, 1986.

[Pea88] Judea Pearl. *Probabilistic reasoning in intelligent systems: networks of plausible inference*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1988.

[Pea91] Judea Pearl. Probabilistic and qualitative abduction. In *AAAI Spring Symposium on Abduction*, pp. 155–158. AAAI Press, Stanford, CA, 1991.

[Poo93] David Poole. Probabilistic horn abduction and bayesian networks. *Artif. Intell.*, 64(1):81–129, 1993.

[Poo97] David Poole. The independent choice logic for modelling multiple agents under uncertainty. *Artif. Intell.*, 94(1-2):7–56, 1997.

Technical Communications of the International Conference on Logic Programming, 2010 (Edinburgh), pp. 202–211
http://www.floc-conference.org/ICLP-home.html

CIRCUMSCRIPTION AND PROJECTION AS PRIMITIVES OF LOGIC PROGRAMMING

CHRISTOPH WERNHARD

Technische Universität Dresden
E-mail address: christoph.wernhard@tu-dresden.de

Abstract. We pursue a representation of logic programs as classical first-order sentences. Different semantics for logic programs can then be expressed by the way in which they are wrapped into – semantically defined – operators for circumscription and projection. (Projection is a generalization of second-order quantification.) We demonstrate this for the stable model semantics, Clark's completion and a three-valued semantics based on the Fitting operator. To represent the latter, we utilize the polarity sensitiveness of projection, in contrast to second-order quantification, and a variant of circumscription that allows to express predicate minimization in parallel with maximization. In accord with the aim of an integrated view on different logic-based representation techniques, the material is worked out on the basis of first-order logic with a Herbrand semantics.

Introduction

The multitude of semantics for logic programs is traditionally specified by a multitude of techniques: different rule languages, consequence operators, syntactic transformations like *reduct* and *completion*, and notions of model, two- and three-valued, for example. This makes it difficult to uncover relationships and transfer results between the semantics. It lets the long-term goal of a single logic-based system in which a variety of logic programming methods is simultaneously available appear quite fanciful. This work aims towards a unified and integrated view on different semantics for logic programs. We show a framework in which a logic program is represented by a classical first-order sentence, and several semantics for logic programs can be characterized by applying two further logic operators that are *defined in terms of classical semantics*: circumscription and projection.

A key observation is that semantics for logic programs involve circumscription in a way such that only certain *occurrences* of a predicate are affected, while others – basically those in the scope of negation as failure – stay unminimized. Indeed, as shown in [Lin91] and described in [Lif08], the stable models semantics can be characterized accordingly in terms of circumscription. From this point of view, the purpose of a rule syntax is just to indicate which occurrences are to be circumscribed. The alternative pursued here is to replace each "original" predicate symbol by two replicas, one of them used in occurrences where circumscription should take effect. The formula then is classical, permitting for example simplifications that preserve classical equivalence.

Projection, a generalization of second-order quantification, can be used to control the interaction between the replicas. In general, projection is applied in the context of this work to express operations in a semantic way that are typically specified in syntactical terms, like

Technical Communications of the 26th International Conference on Logic Programming, Edinburgh, July, 2010
Editors: Manuel Hermenegildo, Torsten Schaub
LIPIcs - Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/LIPIcs.ICLP.2010.202

systematic renaming of predicate symbols and completion construction, where we refine a semantic characterization in [Lee06].

We apply our framework to the stable model semantics, Clark's completion and a three-valued semantics based on the Fitting operator. The first two are distinguished just by the choice of circumscribed predicate occurrences, reflecting the characterization of Clark's completion in terms of stable models with negation as failure in the head described in [Ino98]. The independence of syntactic constructions lets our framework quite naturally cover extensions of normal logic programs, including disjunctive heads and negation as failure in the head. In accord with the long-term goal of a unified logic-based knowledge processing system, the material in the paper is worked out for first-order logic with a Herbrand semantics, extended by circumscription and projection.

The paper is structured as follows: After notation and the used classical semantics have been specified in Sect. 1, projection and circumscription are introduced in Sect. 2. A view of logic programs as classical first-order sentences is described in Sect. 3. On this basis, it is shown in Sect. 4 how semantics for logic programs are expressed in terms of circumscription and projection. Specifically, the stable model semantics, Clark's completion, and a three-valued semantics based on the Fitting operator are considered. In Sect. 5, the new characterization of the latter is related to the traditional definition, and a similar characterization of partial stable models is sketched. In the conclusion, further potential applications of this framework and a view on computational aspects are indicated.

Please note that this paper is a short version of [Wer10a], which includes additional technical material such as proofs showing correspondence of the discussed and traditional characterizations, and a further notational variant of the described framework that might facilitate its application to prove properties of semantics for logic programs.

1. Notation and Preliminaries

Symbolic Notation. We use the following symbols, also with sub- and superscripts, to stand for items of types as indicated in the following list (precise definitions of the types are given later on), considered implicitly as universally quantified in definition, proposition and theorem statements: F, G, H – Formula; A – Atom; L – Literal; S – Set of ground literals (also called *literal scope*); M – Consistent set of ground literals; I, J, K – Structure; β – Variable assignment. We write the positive (negative) *literal* with atom A as $+A$ ($-A$). The *complement* of literal L is written $\widetilde{L}$. The *set of complements* of a given set S of literals (i.e. $\{\widetilde{L}|L \in S\}$) is written $\widetilde{S}$. We assume a fixed first-order signature with at least one constant symbol. The sets of all ground terms, all ground literals, all positive ground literals, and all negative ground literals – with respect to this signature – are denoted by TERMS, ALL, POS, NEG, respectively. *Variables* are x, y, z, also with subscripts. The sequence $x_1, \ldots, x_n$, where n is the arity of predicate symbol p, is abbreviated by $\overline{x}_p$.

Formulas. We assume that a *formula* is constructed from first-order literals and the logic operators shown in the left column of Tab. 1. That is, we consider formulas of first-order logic, extended by an operator for syntactic equality ($\doteq$) and the two operators project and raise, discussed in Sect. 2. As meta-level notation with respect to this syntax, we use versions of the binary connectives with arbitrary integers ≥ 0 as arity, sequences of

Table 1: The Satisfaction Relation

$\langle I,\beta\rangle \models L$	iff_{def}	$L\beta \in I$
$\langle I,\beta\rangle \models \top$		
$\langle I,\beta\rangle \not\models \bot$		
$\langle I,\beta\rangle \models \neg F$	iff_{def}	$\langle I,\beta\rangle \not\models F$
$\langle I,\beta\rangle \models F_1 \wedge F_2$	iff_{def}	$\langle I,\beta\rangle \models F_1$ and $\langle I,\beta\rangle \models F_2$
$\langle I,\beta\rangle \models F_1 \vee F_2$	iff_{def}	$\langle I,\beta\rangle \models F_1$ or $\langle I,\beta\rangle \models F_2$
$\langle I,\beta\rangle \models \forall x\ F$	iff_{def}	for all $t \in \mathsf{TERMS}$ it holds that $\langle I,\beta\frac{t}{x}\rangle \models F$
$\langle I,\beta\rangle \models \exists x\ F$	iff_{def}	there exists a $t \in \mathsf{TERMS}$ such that $\langle I,\beta\frac{t}{x}\rangle \models F$
$\langle I,\beta\rangle \models t_1 \doteq t_2$	iff_{def}	$t_1\beta = t_2\beta$
$\langle I,\beta\rangle \models \mathsf{project}_S(F)$	iff_{def}	there exists a J such that $\langle J,\beta\rangle \models F$ and $J \cap S \subseteq I$
$\langle I,\beta\rangle \models \mathsf{raise}_S(F)$	iff_{def}	there exists a J such that $\langle J,\beta\rangle \models F$ and $J \cap S \subset I \cap S$

variables as quantifier arguments, and omitting of universal quantifiers. A *sentence* is a formula without free variables. A *clausal sentence* is a sentence $\forall x_1 \dots x_n\ F$, where F is a conjunction with arbitrary arity of disjunctions (*clauses*) with arbitrary arity of literals.

Classical Semantics. We use a notational variant of the framework of Herbrand interpretations: An *interpretation* is a pair $\langle I,\beta\rangle$, where I is a *structure*, that is, a set of ground literals that contains for all ground atoms A exactly one of $+A$ or $-A$, and β is a *variable assignment*, that is, a mapping of the set of variables into TERMS. Formula F with all free variables replaced by their image in β is denoted by $F\beta$; the variable assignment that maps x to ground term t and all other variables to the same values as β is denoted by $\beta\frac{t}{x}$. As explicated in [Wer08], the structure component I of an interpretation $\langle I,\beta\rangle$ represents a *structure* in the conventional sense used in model theory, and, moreover, an interpretation represents a *second-order interpretation* [Ebb84], if predicate variables are considered as distinguished predicate symbols. The satisfaction relation between interpretations and formulas is defined by the clauses in Tab. 1. Entailment and equivalence are straightforwardly defined in terms of it. Entailment: $F_1 \models F_2$ holds if and only if for all $\langle I,\beta\rangle$ such that $\langle I,\beta\rangle \models F_1$ it holds that $\langle I,\beta\rangle \models F_2$. Equivalence: $F_1 \equiv F_2$ if and only if $F_1 \models F_2$ and $F_2 \models F_1$.

2. Projection, Literal Scopes and Circumscription

The project operator, defined semantically in Tab. 1, is applied in the context of this paper to provide *semantic* characterizations of operations and properties that are typically defined in syntactic terms: Clark's completion, extracting the subformula with the "converse rules" from Clark's completion, systematic renaming of predicate symbols, and independence of a formula from given predicate symbols. The formula $\mathsf{project}_S(F)$ is called the *projection* of formula F onto literal scope S. The *forgetting* in F about S is a variant of projection, where the scope is considered complementary:

Definition 1 (Forgetting)**.** $\mathsf{forget}_S(F) \stackrel{\text{def}}{=} \mathsf{project}_{\mathsf{ALL}-S}(F)$.

We call a set of ground literals in the role as argument to projection a *literal scope*. When specifying literal scopes, we let a set of predicate symbols stand for the set of all ground instances of literals whose predicate symbol is in the set.

As an intuitive special case of projection, consider a literal scope S that contains the same atoms in positive as well as negative literals. The condition $J \cap S \subseteq I$ in the definition of $\mathsf{project}$ is then equivalent to $J \cap S = I \cap S$, that is, structures I and J are required to be equal as far as members of S are considered, but unrelated otherwise. Projection is a generalization of second-order quantification: if S is the set of all ground literals with a predicate symbol other than p, then $\mathsf{project}_S(F)$ (or equivalently $\mathsf{forget}_{\{\mathsf{p}\}}(F)$) can be expressed by the second-order formula $\exists \mathsf{p}\, F$.

Beyond second-order quantification, the condition $J \cap S \subseteq I$ in the definition of $\mathsf{project}$ encodes a different effect on literals depending on whether they are positive or negative (w.r.t. to formulas that do not contain $\neg$). Hence, this variant of projection is also termed *literal projection*. Consider for example, $\mathsf{forget}_{\{+\mathsf{q},-\mathsf{q}\}}((+\mathsf{p} \vee -\mathsf{q}) \wedge (+\mathsf{q} \vee -\mathsf{r}))$ which is equivalent to $(+\mathsf{p} \vee -\mathsf{r})$, and, in contrast, $\mathsf{forget}_{\{+\mathsf{q}\}}((+\mathsf{p} \vee -\mathsf{q}) \wedge (+\mathsf{q} \vee -\mathsf{r}))$ which is equivalent to $((+\mathsf{p} \vee -\mathsf{q}) \wedge (+\mathsf{p} \vee -\mathsf{r}))$, where $-\mathsf{q}$ is retained. In the context of this paper, these effects are applied to specify a three-valued semantics for logic programs. Further material on projection can be found in [Wer08]. The other "nonstandard" operator defined in Tab. 1 is raise, which we apply to define *scope-determined circumscription* [Wer10b], a generalization of predicate circumscription [McC80]:

Definition 2 (Scope-Determined Circumscription)**.** $\mathsf{circ}_S(F) \stackrel{\text{def}}{=} F \wedge \neg \mathsf{raise}_S(F)$.

The argument S is also a literal scope, which then provides a uniform interface for expressions combining projection and circumscription. Superficially, raise is very similar to $\mathsf{project}$: Consider Tab. 1. The definition of $\mathsf{project}$ is equivalent to $J \cap S \subseteq I \cap S$. Just by replacing the subset relation ($\subseteq$) with strict subset ($\subset$), the definition of raise is obtained. If F is a sentence over disjoint sets of predicate symbols P, Q and Z, then the *parallel predicate circumscription of P in F with fixed Q and varied Z* [Lif94], traditionally written $\mathrm{CIRC}[F; P; Z]$, is expressed as $\mathsf{circ}_{(P \cap \mathsf{POS}) \cup Q}(F)$. Recall that in specifications of literal scopes, we let a set of predicate symbols stand for the set of all ground instances of literals whose predicate symbol is in the set. The scope $(P \cap \mathsf{POS}) \cup Q$ thus is the set of all *positive* ground literals with a circumscribed predicate symbol, and *all* ground literals with a fixed predicate symbol. While circumscription traditionally just allows to express predicate *minimization*, scope-determined circumscription symmetrically permits to express *maximization* by scopes containing just *negative* ground literals with predicate symbols to be maximized. In the context of this paper, parallel minimization and maximization is applied to specify a three-valued semantics for logic programs.

3. Logic Programs as Classical Sentences

A logic program is typically understood as a set of rules of the form:

$$A_1 \mid \ldots \mid A_k \mid \mathsf{not}\, A_{k+1} \mid \ldots \mid \mathsf{not}\, A_l \leftarrow A_{l+1}, \ldots, A_m, \mathsf{not}\, A_{m+1}, \ldots, \mathsf{not}\, A_n. \tag{3.1}$$

This involves logic operators which do not belong to classical first-order logic. To represent a logic program as a classical first-order sentence, we assume that the set of all predicate symbols can be partitioned into *predicate groups*, that is, disjoint sets of equal cardinality. The idea is that each "original predicate symbol" is replicated once in each group. The respective copy of the "original symbol" p in predicate group P is then written p^P. If P and Q are two predicate groups, we say that p^P and p^Q are *corresponding* predicate symbols, assuming that they have the same arity, which we also call *arity of p*. We transfer

the notation p^P to atoms and literals: A^P (L^P) stands for an atom (literal) whose predicate symbol is in predicate group P. Formally, the partitioning into predicate groups can be modeled by means of a total ordering $<_{\mathsf{pred}}$ on predicate symbols such that p denotes the position of p^P within predicate group P sorted according to $<_{\mathsf{pred}}$. Corresponding predicate symbols then have the same positions within their respective group. The set of all such positions p is written PREDS.

Definition 3 (Predicate Groups $\mathcal{C}, \mathcal{F}, \mathcal{O}$)**.** The symbols $\mathcal{C}, \mathcal{F}, \mathcal{O}$ denote three different predicate groups.

Predicate groups $\mathcal{C}, \mathcal{F}, \mathcal{O}$ are used to express logic programs. Roughly, the group indicates whether a predicate occurrence should be *circumscribed* (group $\mathcal{C}$), should be *fixed* with respect to circumscription (group $\mathcal{F}$), or is yet *open* (group $\mathcal{O}$), that is, further operations are applied that place it into group $\mathcal{C}$ or $\mathcal{F}$ at a later stage.

Definition 4 (Rule Clause, Raw Rule Clause)**.** (i) A *rule clause* is a clause of the form

$$+A_1^{\mathcal{C}} \vee \ldots \vee +A_k^{\mathcal{C}} \vee -A_{k+1}^{\mathcal{F}} \vee \ldots \vee -A_l^{\mathcal{F}} \vee -A_{l+1}^{\mathcal{C}} \vee \ldots \vee -A_m^{\mathcal{C}} \vee +A_{m+1}^{\mathcal{F}} \vee \ldots \vee +A_n^{\mathcal{F}},$$

where $n \geq m \geq l \geq k \geq 0$.
(ii) A *raw rule clause* is like a rule clause, except that the literals with indexes from $l+1$ to m are from predicate group $\mathcal{O}$ instead of $\mathcal{C}$.

Based on Def. 4, a logic program can be understood as a clausal sentence with rule clauses or raw rule clauses. In both cases, a logic program is then just a *classical first-order sentence* that meets certain restrictions. ([Raw] rule clauses can contain universal variables.) When we say that a [raw] rule clause *corresponds* to a rule of the form (3.1), we assume that the [raw] rule clause has predicate symbols from groups as indicated by matching (3.1) with Def. 4.

The *head* of a [raw] rule clause is the disjunction of those of its literals whose index is less or equal to l, its *body* is the conjunction of the complements of its literals with index greater than l. A [raw] rule clause can express a *normal rule* (if $k = l = 1$), *integrity constraint* (if $k = l = 0$), *disjunctive rule* (if $k = l > 1$) and a *rule with negation as failure in the head* (if $l > k$). The class of rules in *general extended disjunctive programs (GEDP)* considered in [Ino98] is however strictly more general: In rules of the form (3.1), GEDP would allow also *negated* atoms in place of the atoms A_i, for $i \in \{1, \ldots, n\}$.

Predicate Renaming. Definition 6 below gives a semantic account of systematically replacing predicate symbols from one group P by their correspondents from another group Q. First we define of shorthands for formulas that will be used at several places in the sequel.

Definition 5 (Predicate Inclusion)**.** Let P, Q be predicate groups. (i) $P \leq Q \stackrel{\text{def}}{=} \forall \overline{x} \bigwedge_{p \in \mathsf{PREDS}} (-p^P(\overline{x}_p) \vee +p^Q(\overline{x}_p))$; (ii) $P = Q \stackrel{\text{def}}{=} (P \leq Q) \wedge (Q \leq P)$; (iii) $P < Q \stackrel{\text{def}}{=} (P \leq Q) \wedge \neg(Q \leq P)$.

Definition 6 (Predicate Renaming in Terms of Projection)**.** Let P, Q, P be predicate groups. Then $\mathsf{rename}_{P \setminus Q}(F) \stackrel{\text{def}}{=} \mathsf{forget}_P(F \wedge P{=}Q)$. Notation $\mathsf{rename}_{[P_1 \setminus P_2, \ldots, P_{n-1} \setminus P_n]}(F)$ is a shorthand for $\mathsf{rename}_{P_{n-1} \setminus P_n}(\ldots(\mathsf{rename}_{P_1 \setminus P_2}(F))\ldots)$.

The formula $\mathsf{rename}_{P \setminus Q}(F)$ is equivalent to F with all occurrences of predicate symbols from P replaced by their respective corresponding predicate symbols from Q.

4. Semantics for Logic Programs via Circumscription and Projection

Based on the representation of a logic program as a clausal first-order sentence with raw rule clauses, three well-known semantics for logic programs – the stable model semantics, the classical models of Clark's completion, and the three-valued minimal models obtained with the Fitting operator – can be characterized in terms of circumscription and projection:

Definition 7 (Semantics For Logic Programs)**.** Let F be a formula over $\mathcal{C} \cup \mathcal{F} \cup \mathcal{O}$.

(i) $\mathsf{ans\text{-}stable}(F) \stackrel{\text{def}}{=} \mathsf{rename}_{\mathcal{F}\backslash\mathcal{C}}(\mathsf{circ}_{(\mathcal{C}\cap\mathsf{POS})\cup\mathcal{F}}(\mathsf{rename}_{\mathcal{O}\backslash\mathcal{C}}(F)))$.

(ii) $\mathsf{ans\text{-}completion}(F) \stackrel{\text{def}}{=} \mathsf{rename}_{\mathcal{F}\backslash\mathcal{C}}(\mathsf{circ}_{(\mathcal{C}\cap\mathsf{POS})\cup\mathcal{F}}(\mathsf{rename}_{\mathcal{O}\backslash\mathcal{F}}(F)))$.

(iii) $\mathsf{ans\text{-}fitting}(F) \stackrel{\text{def}}{=} \mathsf{circ}_{(\mathcal{C}\cap\mathsf{POS})\cup(\mathcal{F}\cap\mathsf{NEG})}(\mathcal{C} \leq \mathcal{F} \wedge \mathsf{rename}_{\mathcal{O}\backslash\mathcal{C}}(F) \wedge F^*)$,

where $F^* = \mathsf{rename}_{[\mathcal{C}\backslash\mathcal{O},\, \mathcal{F}\backslash\mathcal{C},\, \mathcal{O}\backslash\mathcal{F}]}(\mathsf{forget}_{\mathcal{C}\cap\mathsf{POS}}(\mathsf{circ}_{(\mathcal{C}\cap\mathsf{POS})\cup\mathcal{O}\cup\mathcal{F}}(F)))$.

The definientia are formulas of first-order logic extended with project (recall that rename is a shorthand for a formula with project) and circ as additional operators. For ans-stable and ans-completion, the involved projection could also be expressed as second-order quantification, as indicated in Sect. 2, and the involved scope-determined circumscription corresponds to parallel predicate circumscription of $\mathcal{C}$ with fixed $\mathcal{F}$. For ans-fitting, in contrast, proper generalizations of second-order quantification and parallel predicate circumscription are utilized: The scope $\mathcal{C} \cap \mathsf{POS}$ of the forgetting is just about *positive literals* with a predicate from $\mathcal{C}$. The scope $(\mathcal{C}\cap\mathsf{POS})\cup(\mathcal{F}\cap\mathsf{NEG})$ of the outer circumscription expresses minimization of $\mathcal{C}$ in parallel with *maximization* of $\mathcal{F}$.

Semantics for logic programs are usually specified in terms of sets of atoms (*answer sets*), or "partial interpretations", that is, consistent sets of literals, representing a three-valued assignment of atoms to truth values: *true* (*false*) for the atoms of positive (negative) literals in the set, and *undefined* for the remaining atoms. In contrast, semantics for logic programs are specified in Def. 7 as classical models. For ans-stable, such a classical model $\langle I, \beta\rangle$ corresponds to the answer set $\{A \mid +A^{\mathcal{C}} \in I\}$. Predicates from $\mathcal{F}$ are not considered for the answer set, reflecting that $\mathcal{F}$ is forgotten by the outer rename. For ans-fitting, $\langle I, \beta\rangle$ corresponds to the partial interpretation $\{+A \mid +A^{\mathcal{C}} \in I\} \cup \{-A | -A^{\mathcal{F}} \in I\}$.

The characterization of stable models in terms of circumscription (Def. 7.i) originates from [Lin91] and is described as *"definition F"* in [Lif08] for logic programs over $\mathcal{C} \cup \mathcal{F}$ (in our notation). We use the third group $\mathcal{O}$ for mapping to other semantics. In [Fer07] a characterization of stable models in terms of a formula translation that is *similar* to predicate circumscription has been presented. Roughly, it differs from circumscription in that only certain *occurrences* of predicates are circumscribed. In this respect it is like the approach pursued here. However, in [Fer07] these occurrences are identified by their syntactic position within formulas from a fragment of classical propositional logic – to the effect, that classically equivalent programs might not be equivalent when considered as logic programs. For a formal proof of the correspondence of ans-stable to the original characterization of stable models [Gel88] and variants of it see [Wer10a].

Equivalence of ans-completion to the syntactically defined Clark's completion [Cla78] is shown in [Wer10a] along the approach of [Lee06], but generalized to first-order logic and refined by utilizing predicate groups: Head literals are distinguished by placing them in $\mathcal{C}$, which allows to prove *equivalence* of semantic and syntactic characterizations, whereas the related Proposition 4 in [Lee06] just makes the weaker statement that the semantically defined completion of F_1 is equivalent to the syntactically defined Clark's completion of F_2 for *some* F_2 *that is equivalent to* F_1.

The formula F in Def. 7 is over $\mathcal{C} \cup \mathcal{F} \cup \mathcal{O}$. In ans-stable and ans-completion, it is subjected to renaming the predicate symbols from $\mathcal{O}$ to either $\mathcal{C}$ or $\mathcal{F}$, respectively, which is actually the only difference between these semantics. For F that are just over $\mathcal{C} \cup \mathcal{F}$ both semantics are identical. The characterization of Clark's completion in terms of stable models of programs with negation as failure in the head, described by means of a program transformation in [Ino98], thus can be rendered by the following equivalence:

$$\text{If } F \text{ is over } \mathcal{C} \cup \mathcal{F} \cup \mathcal{O}, \text{ then } \mathsf{ans\text{-}completion}(F) \equiv \mathsf{ans\text{-}stable}(\mathsf{rename}_{\mathcal{O} \setminus \mathcal{F}}(F)). \tag{4.1}$$

Based on a fixed-point characterization of the models of Clark's completion as so-called *supported models* [Apt88], it has been shown in [Mar92] that a stable model of a *normal* logic program (i.e. with rules of the form (3.1) where $k = l = 1$) is also a minimal model of its Clark completion. For more general classes of logic programs, analogous properties can be proven on the basis of Def. 7: If F is over $\mathcal{C} \cup \mathcal{F} \cup \mathcal{O}$ and $F \equiv \mathsf{forget}(F, \mathcal{O} \cap \mathsf{POS})$, then $\mathsf{ans\text{-}stable}(F) \models \mathsf{ans\text{-}completion}(F)$ and, if in addition, $F \equiv \mathsf{forget}(F, \mathcal{F} \cap \mathsf{NEG})$, then $\mathsf{ans\text{-}stable}(F) \models \mathsf{circ}_{\mathcal{C} \cap \mathsf{POS}}(\mathsf{ans\text{-}completion}(F))$ [Wer10a].

5. Three-Valued Semantics Based on the Fitting Operator

In [Fit85] a consequence operator Φ (*Fitting operator*) is introduced which is applied to construct three-valued interpretations M, represented by consistent sets of ground literals. For a ground program F with rules of the form (3.1), constrained by $k = l = 1$ (i.e. normal rules), the value of the Fitting operator can be described as follows: The body of a rule is *true* with respect to M, if and only if each of its literals is contained in M. It is *false* with respect to M if and only if the complement of at least one of its literals is in M. For a given M, the Fitting operator yields the union of (1.) the set of all positive literals $+A$ such that there exists a rule of F with head $+A$ whose body is true with respect to M, and (2.) the set of all negative literals $-A$ such all rules of F with head $+A$ have a body that is false with respect to M. The minimal fixed point (minimal w.r.t. set inclusion of the consistent literal sets M) of the Fitting operator then represents a (partial) model, the result of the program, and thus might be called "answer set" according to "Fitting's semantics".

To show that ans-fitting (Def. 7.iii) corresponds to this semantics, we reconstruct it in our framework. We use interpretations over the union of the two predicate groups $\mathcal{C}$ and $\mathcal{F}$ to represent the consistent literal sets expressing three-valued or interpretations. Structures I such that

$$\langle I, \beta \rangle \models \mathcal{C} \leq \mathcal{F} \tag{5.1}$$

(assignment β is irrelevant for (5.1) since $\mathcal{C} \leq \mathcal{F}$ does not contain free variables) are mapped with the following one-to-one correspondence to such literal sets M: $\mathsf{litset}(I) \stackrel{\text{def}}{=} \{+A \mid +A^{\mathcal{C}} \in I\} \cup \{-A \mid -A^{\mathcal{F}} \in I\}$; and $\mathsf{litset}^{-1}(M) \stackrel{\text{def}}{=} \{+A^{\mathcal{C}} \mid +A \in M\} \cup \{-A^{\mathcal{C}} \mid +A \notin M\} \cup \{+A^{\mathcal{F}} \mid -A \notin M\} \cup \{-A^{\mathcal{F}} \mid -A \in M\}$. Minimization with respect to set inclusion of the literal sets M can be expressed by scope-determined circumscription with scope $S = (\mathcal{C} \cap \mathsf{POS}) \cup (\mathcal{F} \cap \mathsf{NEG})$, since $\mathsf{litset}(I) \subseteq \mathsf{litset}(J)$ if and only if $I \cap S \subseteq J$. The scope S effects that predicates from $\mathcal{C}$ are minimized, and, in parallel, predicates from $\mathcal{F}$ are *maximized*, which can not be directly expressed by conventional predicate circumscription.

The Fitting operator is – like the original form of Clark's completion – applied to normal logic programs, that is, sets of rules of the form (3.1) where $k = l = 1$. Such a program corresponds to a clausal sentence with rule clauses that are over $\mathcal{F}$ except possibly for a single positive literal over $\mathcal{C}$. For Clark's completion, in a first "preprocessing" step, such a

sentence is transformed to an equivalent, possibly nonclausal, sentence of a second particular form, which is then the basis for the proper completion transformation. A suitable such second form will be specified in Def. 8 below. We call it *normal completion input sentence*, since any clausal sentence with rule clauses constrained by $k = l = 1$ is equivalent to such a sentence, obtainable by straightforward rewriting with equivalences, including

$$+p(t_1, \ldots, t_n) \vee G \;\equiv\; \forall x_1 \ldots x_n \, +p(x_1, \ldots, x_n) \vee \neg x_1 \doteq t_1 \vee \ldots \vee \neg x_n \doteq t_n \vee G, \qquad (5.2)$$

where $x_1, \ldots, x_n$ are variables not occurring in $t_1, \ldots, t_n, G$.

Definition 8 (Normal Completion Input Sentence)**.** A sentence F is called a *normal completion input sentence* if it is over $\mathcal{C} \cup P$, with P being a set of predicate symbols not in $\mathcal{C}$, and is of the form $\forall \overline{x} \; (\bigwedge_{p \in \mathsf{PREDS}} (+p^{\mathcal{C}}(\overline{x}_p) \vee G_p(\overline{x}_p)))$, where (1.) $\overline{x}$ is $x_1, \ldots, x_k$, with k being the maximal arity of all members of PREDS, (2.) $G_p(\overline{x}_p)$ are formulas whose free variables are in $\overline{x}_p$, (3.) $G_p(\overline{x}_p)$ does not contain predicate symbols from $\mathcal{C}$.

In traditional terminology, a subformula $+p^{\mathcal{C}}(\overline{x}_p)$ of a normal completion input sentence corresponds to a head, and $G_p(\overline{x}_p)$ to the negated disjunction of all bodies of clauses with head $+p^{\mathcal{C}}(\overline{x}_p)$. For a normal completion input sentence F over $\mathcal{C} \cup \mathcal{F}$, Clark's completion of F can then be defined as $\mathsf{rename}_{\mathcal{F} \setminus \mathcal{C}}(F \wedge F^*)$, where F^* is the syntactic completion addendum of F, defined as follows:

Definition 9 (Syntactic Completion Addendum)**.** Let F be a normal completion input sentence with syntactic constituents as specified in Def. 8. The following sentence is called the *syntactic completion addendum* of F: $\forall \overline{x} \; (\bigwedge_{p \in \mathsf{PREDS}} (-p^{\mathcal{C}}(\overline{x}_p) \vee \neg G_p(\overline{x}_p)))$.

Let F be a normal completion input sentence over $\mathcal{C} \cup \mathcal{F} \cup \mathcal{O}$. Recall the definition of ans-fitting (Def. 7.iii):

$$\mathsf{ans\text{-}fitting}(F) \;\stackrel{\text{def}}{=}\; \mathsf{circ}_{(\mathcal{C} \cap \mathsf{POS}) \cup (\mathcal{F} \cap \mathsf{NEG})}(\mathcal{C} \leq \mathcal{F} \wedge \mathsf{rename}_{\mathcal{O} \setminus \mathcal{C}}(F) \wedge F^*),$$

where $F^* = \mathsf{rename}_{[\mathcal{C} \setminus \mathcal{O}, \, \mathcal{F} \setminus \mathcal{C}, \, \mathcal{O} \setminus \mathcal{F}]}(\mathsf{forget}_{\mathcal{C} \cap \mathsf{POS}}(\mathsf{circ}_{(\mathcal{C} \cap \mathsf{POS}) \cup \mathcal{O} \cup \mathcal{F}}(F)))$. The outer circumscription has the scope S discussed above in this section, and thus effects minimization to the smallest models with respect to the three-valued view of interpretations. The argument formula of this circumscription consists of three conjuncts. The first one is (5.1) which excludes interpretations without a consistent three-valued correspondence. The other ones correspond to the positive and negative consequences, respectively, of the Fitting operator.

Assume that the normal completion input sentence F has been obtained in a "preprocessing" step, as outlined above, from an equivalent clausal sentence with raw clauses, representing a conjunction F_0 of rules of the form (3.1), constrained by $k = l = 1$, and such that all heads have just mutually distinct variables as argument terms (in presence of equivalence (5.2) the last condition is w.l.o.g.). Let p be some member of PREDS. The formula $G_p(\overline{x}_p)$ is then a constituent of F as specified in Def. 8. The second conjunct $\mathsf{rename}_{\mathcal{O} \setminus \mathcal{C}}(F)$ is the original logic program, with $\mathcal{O}$ renamed to $\mathcal{C}$, as in ans-stable. It can be shown that $\langle I, \beta \rangle \models \mathcal{C} \leq \mathcal{F} \wedge \mathsf{rename}_{\mathcal{O} \setminus \mathcal{C}}(\neg G_p(\overline{x}_p))$ if and only if there is a rule R with head predicate p in F_0 such that the body of its ground instance $R\beta$ is *true* with respect to $\mathsf{litset}(I)$. The subformulas $(+p^{\mathcal{C}}(\overline{x}_p) \vee \mathsf{rename}_{\mathcal{O} \setminus \mathcal{C}}(\neg G_p(\overline{x}_p)))$ in $\mathsf{rename}_{\mathcal{O} \setminus \mathcal{C}}(F)$ then allow to infer positive literals with predicate $p^{\mathcal{C}}$, corresponding to positive consequences of the Fitting operator.

Analogously, $\langle I, \beta \rangle \models \mathcal{C} \leq \mathcal{F} \wedge \mathsf{rename}_{[\mathcal{F} \setminus \mathcal{C}, \, \mathcal{O} \setminus \mathcal{F}]}(G_p(\overline{x}_p))$ if and only if for all rules R with head predicate p in F_0 it holds that the body of the ground instance $R\beta$ is *false* with

respect to $\mathsf{litset}(I)$. Subformulas of the form $(-p^{\mathcal{F}}(\overline{x}_p) \vee \mathsf{rename}_{[\mathcal{F}\setminus\mathcal{C},\, \mathcal{O}\setminus\mathcal{F}]}(G_p(\overline{x}_p)))$ then allow to infer negative literals with predicate $p^{\mathcal{F}}$, corresponding to negative consequences of the Fitting operator. Sentence F^* is the universally quantified conjunction of these subformulas, for each predicate p from PREDS. It is equivalent to the syntactic completion addendum of F (Def. 9), subjected to switching group assignments $\mathcal{C}$ and $\mathcal{F}$, and renaming $\mathcal{O}$ to $\mathcal{F}$. This switching and renaming is expressed by rename applied to $[\mathcal{C}\setminus\mathcal{O},\ \mathcal{F}\setminus\mathcal{C},\ \mathcal{O}\setminus\mathcal{F}]$. As shown in [Wer10a], the circumscription in F^* is equivalent to Clark's completion of F. The forgetting about $\mathcal{C} \cap \mathsf{POS}$ serves to extract an equivalent to the *syntactic completion addendum* from the completion, according to the following theorem proven in [Wer10a]:

Theorem 5.1 (Semantic Extraction of the Completion Addendum)**.** *Let F be a completion input sentence and F^* its completion. Then $\mathsf{forget}_{\mathcal{C}\cap\mathsf{POS}}(F^*)$ is equivalent to the syntactic completion addendum of F.*

Literal projection is utilized there to preserve the negative literals from $\mathcal{C}$ in the addendum, but forget about the positive literals from $\mathcal{C}$ in the original formula, and with them the whole original formula.

A further prominent semantics for logic programs with three-valued models is the *partial stable model semantics*. In [Jan06] a characterization of partial stable models as stable models of a translated program is given (tracing back to earlier work [Sch95]). Based on a reconstruction of the syntactic transformation Tr(P) of [Jan06] in terms of rename, and on the characterization of stable models by $\mathsf{ans\text{-}stable}$, partial stable models can be characterized in our framework as shown in the following definition. The three-valued (i.e. partial) interpretations are represented there in the same way as shown above for the Fitting semantics.

Definition 10. Let F be a formula over $\mathcal{C} \cup \mathcal{F} \cup \mathcal{O}$. Let $\mathcal{C}'$ and $\mathcal{F}'$ be two additional predicate groups, different from each other and from $\mathcal{C}, \mathcal{F}, \mathcal{O}$.

$$\mathsf{ans\text{-}partial\text{-}stable}(F) \stackrel{\text{def}}{=} \mathsf{rename}_{[\mathcal{C}'\setminus\mathcal{C},\mathcal{F}'\setminus\mathcal{F}]}(\mathsf{circ}_{((\mathcal{C}\cup\mathcal{F})\cap\mathsf{POS})\cup\mathcal{C}'\cup\mathcal{F}'}(\mathcal{C} \leq \mathcal{F} \wedge F_1 \wedge F_2)),$$

where $F_1 = \mathsf{rename}_{[\mathcal{O}\setminus\mathcal{C},\mathcal{F}\setminus\mathcal{F}']}(F)$ and $F_2 = \mathsf{rename}_{[\mathcal{O}\setminus\mathcal{C},\mathcal{F}\setminus\mathcal{C}',\mathcal{C}\setminus\mathcal{F}]}(F)$.

6. Conclusion

We investigated a representation of logic programs as classical first-order sentences that are wrapped into the semantically defined additional operators circumscription and projection, in different ways, rendering different established semantics of logic programs. The generality of our framework indicates interesting spaces that have yet to be explored: Our characterizations of semantics for logic programs apply to broad formula classes. The scopes of circumscription and projection in the characterizations of semantics could be modified, or additional applications of projection could be merged in, to express, for example, models that are "stable only with respect to some atoms", and to restrict answer sets to atoms that are relevant for the user [Eit08, Geb09].

A computational approach to the processing of operators for circumscription and projection is "elimination", analogous to second-order quantifier elimination: Computing for a given formula that involves the operator (second-order quantifier, resp.) an equivalent formula without the operator (second-order quantifier, resp.). Indeed, methods for the computation of circumscription and projection can essentially be considered as methods for

second-order quantifier elimination [Gab08, Wer08, Wer09]. Our framework thus indicates that methods for processing logic programs could be seen in this context: On one hand, established methods for second-order quantifier elimination might be applied to process logic programs, which might be especially interesting for nonground programs. On the other hand, known efficient techniques for processing logic programs with specific semantics get embedded in a wider context when seen as particular efficient second-order quantifier elimination methods for constrained inputs.

Acknowledgements. I am obliged to anonymous referees of an earlier version for suggestions to improve the presentation and bringing important related works [Mar92, Sch95, Jan06] to attention.

References

[Apt88] K. R. Apt, H. A. Blair, and A. Walker. Towards a theory of declarative knowledge. In Jack Minker (ed.), *Foundations of deductive databases and logic programming*, pp. 89–148. Morgan Kaufmann, San Francisco, 1988.

[Cla78] K. L. Clark. Negation as failure. In H. Gallaire and J. Minker (eds.), *Logic and Databases*, pp. 292–322. Plenum Press, New York, 1978.

[Ebb84] H.-D. Ebbinghaus, J. Flum, and W. Thomas. *Mathematical Logic*. Springer, New York, 1984.

[Eit08] T. Eiter and K. Wang. Semantic forgetting in answer set programming. *Artif. Int.*, 172:1644–1672, 2008.

[Fer07] P. Ferraris, J. Lee, and V. Lifschitz. A new perspective on stable models. In *IJCAI-07*, pp. 372–379. 2007.

[Fit85] M. Fitting. A Kripke-Kleene semantics for logic programs. *J. of Logic Prog.*, 2(4):295–312, 1985.

[Gab08] D. M. Gabbay, R. A. Schmidt, and A. Szałas. *Second-Order Quantifier Elimination: Foundations, Computational Aspects and Applications*. College Publications, London, 2008.

[Geb09] M. Gebser, B. Kaufmann, and T. Schaub. Solution enumeration for projected Boolean search problems. In *CPAIOR 2009*, pp. 71–86. 2009.

[Gel88] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *ICLP/SLP 1988*, pp. 1070–1080. 1988.

[Ino98] K. Inoue and Chiaki Sakama. Negation as failure in the head. *J. of Logic Prog.*, 35(1):39–78, 1998.

[Jan06] T. Janhunen, I. Niemelä, D. Seipel, Patrik Simons, and Jia-Huai You. Unfolding partiality and disjunctions in stable model semantics. *ACM Trans. on Comput. Logic*, 7(1):1–37, 2006.

[Lee06] J. Lee and F. Lin. Loop formulas for circumscription. *Artificial Intelligence*, 170:160–185, 2006.

[Lif94] V. Lifschitz. Circumscription. In *Handbook of Logic in AI and Logic Programming*, vol. 3, pp. 298–352. Oxford University Press, Oxford, 1994.

[Lif08] V. Lifschitz. Twelve definitions of a stable model. In *ICLP 2008*, pp. 37–51. 2008.

[Lin91] F. Lin. *A Study of Nonmonotonic Reasoning*. Ph.D. thesis, Stanford University, 1991.

[Mar92] W. Markek and V. S. Subrahmanian. The relationship between stable, supported, default and autoepistemic semantics for general logic programs. *Theor. Computer Science*, 103:365–386, 1992.

[McC80] John McCarthy. Circumscription – a form of non-monotonic reasoning. *Artif. Int.*, 13:27–39, 1980.

[Sch95] J. S. Schlipf. The expressive powers of the logic programming semantics. *J. of Computer and System Sciences*, 51(1):64–86, 1995.

[Wer08] C. Wernhard. Literal projection for first-order logic. In *JELIA 08*, pp. 389–402. 2008.

[Wer09] C. Wernhard. Tableaux for projection computation and knowledge compilation. In *TABLEAUX 2009*, pp. 325–340. 2009.

[Wer10a] C. Wernhard. Circumscription and projection as primitives of logic programming – extended version. Tech. rep., TU Dresden, 2010. Available from http://cs.christophwernhard.com/papers/logprog2010extended.pdf.

[Wer10b] C. Wernhard. Literal projection and circumscription. In *FTP'09*, *CEUR Proc.*, vol. 556. 2010.

Technical Communications of the International Conference on Logic Programming, 2010 (Edinburgh), pp. 212–221
http://www.floc-conference.org/ICLP-home.html

TIMED DEFINITE CLAUSE ω-GRAMMARS

NEDA SAEEDLOEI [1] AND GOPAL GUPTA [1]

[1] Department of Computer Science
University of Texas at Dallas, Richardson, TX 75080, USA
E-mail address: {nxs048000,gupta@utdallas.edu}

Abstract. We propose timed context-free grammars (TCFGs) and show how parsers for such grammars can be developed using *definite clause grammars (DCGs)* coupled with *constraints over reals (CLP(R))*. Timed context-free grammars describe timed context-free languages (TCFLs). We next extend timed context-free grammars to timed context-free ω-grammars (ω-TCFGs for brevity) and incorporate *co-inductive logic programming* in DCGs to obtain parsers for them. Timed context-free ω-grammars describe timed context-free languages containing infinite-sized words, and are a generalization of timed ω-regular languages recognized by *timed automata*. We show a practical application of ω-TCFGs to the well-known *generalized railroad crossing problem*.

Introduction

Using timed automata is a popular approach to designing, specifying and verifying real-time systems [Alu90, Alu94]. Timed and hybrid automata provide the foundational basis for cyber-physical systems (CPS) that are currently receiving a lot of attention [Lee08, Gup06]. Timed automata are ω-automata [Tho90] extended with clocks (or stop-watches). Transitions from one state to another are made not only on the alphabet symbols of the language, but also on constraints imposed on clocks (e.g., at least 2 units of time must have elapsed). A timed automaton recognizes a sequence of timed words, where a timed word is made of symbols from the alphabet of the language the automaton accepts (a regular language), paired with a time-stamp indicating the time that symbol was seen. Since finite automata are equivalent to regular languages it seems natural to think of timed automata as being equivalent to timed regular languages. However, regular expressions are unsuitable for many complex (and useful) applications; in many situations one needs context-free languages. For real-time systems this means that timed regular languages may not be powerful enough, and one has to resort to TCFLs.

In this paper we propose timed grammars as a simple and natural method for describing timed languages. Timed grammars describe words that have real-time constraints placed on the times at which the words' symbols appear. Note that previous approaches to dealing with time typically have discretized time, which resulted in frameworks that cannot model problems faithfully. Lack of a framework that models real-time systems and other continuous physical quantities has been perceived as a problem by the research community [Lee08, Gup06].

Key words and phrases: Constraint Logic Programming over reals, Co-induction, Context-Free Grammars, ω-Grammars.

Technical Communications of the 26th International Conference on Logic Programming, Edinburgh, July, 2010
Editors: Manuel Hermenegildo, Torsten Schaub
LIPIcs - Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/LIPIcs.ICLP.2010.212

We extend the concept of context-free grammars to timed context-free grammars and timed context-free ω-grammars. Informally, a timed context-free grammar is obtained by associating clock constraints with terminal and non-terminal symbols of the productions of a CFG. The timed language accepted by a timed CFG contains those strings that are accepted by the underlying untimed grammar but which also satisfy the timing constraints imposed by the associated clock constraints. The language accepted by a timed ω-CFG (ω-TCFG) contains timed strings that are infinite in size. Such languages are useful for modeling complex CPS including real-time systems [Sae] that run forever.

In this paper we describe timed grammars, and show how DCGs together with CLP(R) and co-induction can be used to develop an effective and elegant method for parsing them. We also illustrate timed grammars through examples, in particular we show how the *controller* component of the *generalized railroad crossing problem* of Lynch and Heitmeyer [Hei94] can be specified naturally using timed grammars. Introducing ω-TCFGs and their logic programming realization is the main contribution of this paper. Note that pushdown timed automata (PTA) have been introduced in the past; however, to the best of our knowledge, ours is the first attempt to (i) develop the notion of timed grammars, and (ii) develop practical methods for parsing these timed grammars. We assume that the reader is familiar with *constraint logic programming* over reals *(CLP(R))* [Jaf94]. We next present an overview of *co-inductive logic programming (co-LP)*.

1. Co-inductive Logic Programming

The traditional declarative and operational semantics for logic programming (LP) is inadequate for various programming practices such as programming with infinite data structures and *corecursion* [Bar96]. Recently *co-induction* [Bar96] has been introduced into logic programming by Simon et al [Sim06a, Sim06b] to overcome this problem. Co-inductive LP can be used for reasoning about unfounded sets, behavioral properties of (interactive) programs, elegantly proving liveness properties in model checking, type inference in functional programming, representing and verifying properties of Kripke structures and ω-automata, etc. [Gup07, Sim07].

Co-induction is the dual of induction and corresponds to the greatest fixed-point (*gfp*) semantics. Simon et al's work gives an operational semantics—similar to SLD resolution—for computing the greatest fixed-point of a logic program. This operational semantics (called co-SLD resolution) relies on the *co-inductive hypothesis rule* and systematically computes elements of the *gfp* of a program (w.r.t. a query) via backtracking. It is briefly described below. The semantics is limited only to *regular proofs*, i.e., those cases where the infinite behavior is obtained by infinite repetition of a finite number of finite behaviors.

In the co-inductive LP (co-LP) paradigm the declarative semantics of the predicate is given in terms of *infinitary Herbrand (or co-Herbrand) universe, infinitary Herbrand (or co-Herbrand) base* [Llo87], and *maximal models* (computed using *greatest fixed-points)* [Sim06a]. The operational semantics under co-induction is identical to Prolog's operational semantics, except for the following addition [Sim06a]: a predicate call `p(`$\bar{t}$`)` succeeds if it unifies with one of its ancestor calls. Thus, every time a call is made, it has to be remembered. This set of ancestor calls constitutes the *co-inductive hypothesis set*. Under co-LP, infinite *rational* answers can be computed, and infinite rational terms are allowed as arguments of predicates. Infinite terms are represented as solutions to unification equations, and the occurs check is omitted during the unification process: for example, `X = [1 | X]` represents the binding

of `X` to an infinite list of `1`'s. Thus, in co-SLD resolution, given a single clause (note the absence of a base case)

```
p([1 | X]) :- p(X).
```

the query `?- p(A).` succeeds in two resolution steps with the (infinite) answer: `A = [1 | A]` which is a finite representation of the infinite answer: `A = [1, 1, 1, ....]`. Now for a slightly more non-trivial example, consider the program `P1` below:

```
bit(0).                          stream([]).
bit(1).                          stream([H| T]) :- bit(H), stream(T).
```

A call to `?- stream(X).` in a standard LP system will systematically generate all finite bit streams one by one starting from the `[]` stream. Suppose now we remove the base case and obtain the program `P2`:

```
bit(0).
bit(1).                          stream([H| T]) :- bit(H), stream(T).
```

Under co-inductive semantics, posing the query `?- stream(X).` will produce infinite sized streams as answers, e.g., `X = [0, 0, 0 | X]`, `X = [1, 1, 1 | X]`, `X = [0, 1, 0 | X]`, etc.

2. Timed Context-free ω-Grammars

A timed grammar is a grammar extended with real-valued variables.[1] These variables model the clocks in the grammar. All clocks advance at the same rate (identical to the rate at which a wall clock advances). Clock constraints are used to restrict the kind of strings generated by the underlying untimed grammar. Clocks may be reset to zero when a particular symbol of the language is seen. Informally, timed grammars are collections of production rules in which clock expressions (clock constraints and resets) can appear after both the terminal and non-terminal symbols on the right hand side. The timed language accepted by a timed grammar consists of strings that can be derived from rules of the grammar and that satisfy the clock constraints appearing in that grammar. Since CFGs are suitable for describing a large class of complex applications, we consider timed CFGs only (though, in general, one could have timed versions of context-sensitive grammars, as well as timed Turing machines). We use a timed CFG to describe a timed language by generating each string of that language in a manner similar to a CFG. Informally, during the derivation, the right hand side of a rule can be substituted for a non-terminal symbol only if the timing constraints accompanying that non-terminal symbol are satisfied. Similarly, a terminal symbol cannot be generated if its accompanying clock constraint is violated. Timed context-free grammars extend CFGs with:

- A fixed number of *clocks*, which may be reset to zero. Clock names are global to all the production rules, i.e., all occurrences of a clock name c refer to the same clock.
- *Clock resets*, which are written within curly braces and can appear after a terminal or a non-terminal symbol on the right hand side of a production rule. Resetting the clock after a terminal symbol a, denoted $a\{c := 0\}$ where c is the clock, is used to remember the time at which a has been seen; while resetting the clock after a non-terminal symbol B, is used to remember the time at which the *last* terminal symbol in the string that is reduced to B has been seen.

[1]Grammars extended with real-valued variables can be used to model other cyber-physical phenomenon, however, in this paper our focus is on real-time systems.

- *Clock constraints*, which are put in the timed grammar in exactly the same manner as *clock resets*, i.e., within curly braces; however, they are used to indicate the timing constraints between the time stamps of the various symbols that appear in an accepted string. For example $a\{c < 2\}$ indicates that the symbol a must appear within two units of time since the clock c was reset.

Each *terminal* as well as *non-terminal* symbol appearing on the right hand side of a production rule maybe followed by curly braces that enclose a non-empty sequence of *clock resets* and *clock constraints*. Before we formally define timed grammars, let us consider some examples.

Example 2.1. Consider a language in which each sequence of a's is followed by a sequence of an equal number of b's, with each accepted string having at least two a's and two b's. For each pair of equinumerous sequences of a's and b's, the *first* symbol b must appear within 5 units of time from the *first* symbol a and the *final* symbol b must appear within 20 units of time from the *first* symbol a. The grammar annotated with clock expressions is shown below: c is a clock which is reset when the first symbol a is seen.

1. $S \to R\ \ S$
2. $R \to a\ \ \{c := 0\}\ \ T\ \ b\ \ \{c < 20\}$
3. $T \to a\ \ T\ \ b$
4. $T \to a\ \ b\ \ \{c < 5\}$

Note that, for example, in the first production ($S \to R\ S$), the sets of clock constraints associated with R and S are empty, and are therefore omitted. Note also that in the above grammar, the first rule is co-inductive (i.e., a recursive rule with no base case). Thus, this grammar is an ω-grammar.

Example 2.2. The following grammar describes a language in which sequences of a's are followed by a final symbol b, which must appear within 5 units of time from the *last* symbol a.

$S \to a\ \ \{c := 0\}\ \ S$
$S \to b\ \ \{c < 5\}$

Example 2.3. With a slight change in the timed grammar in the previous example we can capture a language in which sequences of a's are followed by a final symbol b that appears within 5 units of time from the *first* symbol a. The timed grammar for this timed language is as follows.

$S \to a\ \ \{c := 0\}\ \ R$
$R \to a\ \ R$
$R \to b\ \ \{c < 5\}$

Note the difference in how the clocks are reset in the last two examples. The clock c in Example 2.2 is reset on every occurrence of the symbol a; while in Example 2.3 the clock c is reset only on the first occurrence of symbol a.

Definition 2.4. A *timed context-free grammar* is a 6-tuple $G = \langle V, T, C, E, R, S\rangle$, where

- V is a finite set of non-terminal symbols;
- T is a finite set of terminal symbols, disjoint from V, which is the alphabet of the language defined by the grammar;
- C is a finite set of clock identifiers;

- E is a set of clock expressions over C (clock constraints and clock resets);
- R is a finite set of productions of the form $A \to (a\delta)^*$, where $A \in V$, $a \in (V \cup T)$, and δ denotes a (possibly empty) collection of clock expressions from E (* denotes Kleene closure);
- $S \in V$ is a special symbol called the *start* symbol.

The set E of clock expressions is limited to expressions of the form $\{c := 0\}$ and $\{c \sim x\}$, where c is a free variable, x is a constant, and $\sim \in \{=, <, \leq, >, \geq\}$.

Definition 2.5. A *timed context-free language (TCFL)* is a set of timed strings. A timed string is a sequence of pairs of the form $(a,\ t_a)$ where a is a symbol from the alphabet, and t_a is the time at which the symbol a was seen (by time we mean wall clock or physical time). If $(a,\ t_a)$ is immediately followed by $(b,\ t_b)$ in a timed string, then $t_b > t_a$, i.e., two or more symbols cannot appear at the same instant.

We now formally define the language generated by a timed context-free grammar $G = \langle V, T, C, E, R, S\rangle$. Note that because time flows linearly we only consider left to right *derivations*. Note also that because clocks may reset during a derivation, the reset times have to be recorded as part of the state. We could carry this state locally with each step of the derivation; however, to keep the exposition below simple we maintain this state as a global entity, which is accessed during each step of the derivation. For each clock c, $reset(c)$ denotes the wall clock time at which clock c was last reset. The wall clock is treated as a global variable whose value can be read at any time. We consider two cases, one where we reduce using a production that has a terminal symbol occurring in the leftmost position on its RHS, and the other where this production has a non-terminal symbol as the leftmost symbol in the RHS.

Case I: If $A \to a\sigma B$ is a production of R, where $a \in T$, σ is set of clock expressions (possibly empty) from E, and B is in $(V \cup T \cup E)^*$, then $A\gamma F$ *yields* $(a, t_a)B\gamma F$, written $A\gamma F \underset{G}{\Rightarrow} (a, t_a)B\gamma F$, where F is in $(V \cup T \cup E)^*$, γ is set of clock expressions (possibly empty) from E associated with non-terminal symbol A, and for each clock expression $k \in \sigma$:

- if $k = \{c := 0\}$, then $reset(c) = t_a$, i.e., we record that the clock c was reset at wall clock time t_a.
- if $k = \{c \sim x\}$, then $t_a - reset(c) \sim x$ must hold. If this clock constraint does not hold, then the derivation fails.

Case II: If $A \to D\sigma B$ is a production of R, where $D \in V, \sigma$ is set of clock expressions (possibly empty) from E, and B is in $(V \cup T \cup E)^*$, then $A\gamma F \underset{G}{\Rightarrow} D\sigma B\gamma F$, where F is in $(V \cup T \cup E)^*$, and γ is set of clock expressions (possibly empty) from E associated with non-terminal symbol A.

If two sets of clock expressions, γ and σ, appear next to each other during a derivation, they are replaced by $\gamma \cup \sigma$.

Definition 2.6. Suppose that $\alpha_1, \alpha_2, \ldots, \alpha_m$ are strings in $(V \cup T \cup E \cup (T \times \mathbf{R}^+))^*$, $m \geq 1$, and

$$\alpha_1 \underset{G}{\Rightarrow} \alpha_2,\ \alpha_2 \underset{G}{\Rightarrow} \alpha_3,\ \ldots,\ \alpha_{m-1} \underset{G}{\Rightarrow} \alpha_m.$$

Then we say that $\alpha_1 \overset{*}{\underset{G}{\Rightarrow}} \alpha_m$, or α_1 *derives* α_m in grammar G. Alternatively, $\alpha \overset{*}{\underset{G}{\Rightarrow}} \beta$ if β follows from α by application of zero or more productions of R. Note that $\alpha \overset{*}{\underset{G}{\Rightarrow}} \alpha$ for each string α. If it is clear which grammar G is involved, we use $\Rightarrow$ for $\underset{G}{\Rightarrow}$ and $\overset{*}{\Rightarrow}$ for $\overset{*}{\underset{G}{\Rightarrow}}$.

Definition 2.7. The language generated by G [denoted L(G)] is

$$\{w \mid w = (w_1, t_1), (w_2, t_2), \ldots, (w_n, t_n) \textit{ where } w_i \in T, t_i \in \mathbf{R}^+, t_1 < t_2 \cdots < t_n, \textit{and } S \overset{*}{\underset{G}{\Rightarrow}} w\}.$$

That is, a timed string is in $L(G)$ if:

(1) The timed string (timed word) consists of a sequence of pairs; the first element of each pair is a terminal symbol and the second is a real number.
(2) The timed string can be derived from S and satisfies the time constraints imposed by the grammar.

We call L a *timed context-free language (TCFL)* for some timed CFG G, if $L = L(G)$.

As mentioned before, timed context-free ω-grammars (ω-TCFG) are CFGs with co-recursive grammar rules (i.e., recursive rules with no base cases). ω-TCFGs generate TCFLs with infinite sized words.

Adding clocks to a context-free grammar results in a grammar which is not context-free any more, rather it is context-sensitive. Intuitively, this is easy to see, as whether a non-terminal symbol can be reduced depends not only on if a matching production exists but also that the clock constraints associated with the non-terminal are consistent with the past clock resets. In other words a pushdown timed automaton would not be able to recognize a TCFG, because an extra component is needed to save the information about various clocks which are used in the TCFG. Thus, a TCFL cannot be recognized by a push down automaton. We conjecture that TCFGs are equivalent to *linear bounded automata* (LBA). A linear bounded automaton is a nondeterministic Turing machine which, instead of having potentially infinite tape for storage, is restricted to the portion of the tape containing the input x plus the two tape squares holding the end-markers.

Example 2.8. Consider the timed ω-grammar of Example 2.1. Consider a timed word

$$(a, 2), (a, 4), (b, 5), (b, 10), \ldots$$

Below we give the initial segment of the derivation of this timed word. The global state will record the time at which clock c is reset; c will be set to the value 2 when the first symbol a is seen.

$$\begin{aligned}
S &\Rightarrow R\ S \\
&\Rightarrow a\ \{c := 0\}\ T\ b\ \{c < 20\}\ S \\
&\Rightarrow (a, 2)\ T\ b\ \{c < 20\}\ S \\
&\Rightarrow (a, 2)\ a\ b\ \{c < 5\}\ b\ \{c < 20\}\ S \\
&\Rightarrow (a, 2)\ (a, 4)\ b\ \{c < 5\}\ b\ \{c < 20\}\ S \\
&\Rightarrow (a, 2)\ (a, 4)\ (b, 5)\ b\ \{c < 20\}\ S \\
&\Rightarrow (a, 2)\ (a, 4)\ (b, 5)\ (b, 10)\ S \ldots
\end{aligned}$$

3. Modeling ω-TCFGs with Co-inductive CLP(R)

To model and reason about ω-TCFGs we should be able to handle the fact that: (i) the underlying language is context-free, (ii) accepted strings are infinite, and (iii) clock constraints are posed over continuously flowing time. All three aspects can be elegantly handled within LP. It is well known that the definite clause grammar (DCG) facility of Prolog allows one to obtain parsers for context-free grammars (and even for context-sensitive grammars) with minimal effort. By extending LP with co-induction, one can develop language processors that recognize infinite strings. DCGs extended with co-induction can act as recognizers for ω-grammars. Further, incorporation of co-induction and CLP(R) into the DCG allows modeling of time aspects of ω-TCFGs. Once an ω-TCFG is modeled as a co-inductive CLP(R) program, it can be used to (i) check whether a particular timed string will be accepted or not; and, (ii) systematically generate all possible timed strings that can be accepted (note that a CLP(R) system will represent time-stamps of terminal symbols as variables in the output, with constraints imposed on them). The LP realization of the system based on co-induction and CLP(R) can also be used to verify system properties by posing appropriate queries.

We have developed a system that takes an ω-TCFG and converts it into a DCG augmented with co-induction and CLP(R). The resulting co-inductive constraint logic program acts as a parser for the ω-TCFL recognized by the ω-TCFG. The general method of this system is outlined next (the system is shown in `http://www.utdallas.edu/~nxs048000/tGrammar.yap`). The method takes an ω-TCFG as input and generates a parser as a collection of DCG rules (one rule per production in the ω-TCFG), where each rule is extended with clock expressions. For simplicity of presentation the method is explained for a timed grammar with one clock, but it can handle any number of clocks in a similar manner. We assume c is the clock; Ti, and To are used to remember the last wall clock time this clock was reset, and to pass on this clock's value to the next step in the derivation respectively. T, and Tn are used to represent the wall clock time, and the new wall clock time after each step in the derivation respectively. The method replaces every component of the timed grammar with its corresponding timed component as follows.

- A *non-terminal* symbol s is replaced with predicate $s(T, Ti, Tn, To)$;
- A *terminal* symbol a is replaced with $[(a, t_a)]$, where t_a is the wall clock time at which the symbol a appeared;
- A *clock constraint* of the form $\{c \sim x\}$, where $\sim \in \{=, <, \leq, >, \geq\}$ is replaced with $\{\{T - Ti \sim x\}\}$;
- A *clock reset* of the form $\{c := 0\}$ is replaced with $\{\{Ti = T\}\}$ or $\{\{To = T\}\}$ (depending on where it appears in the production).

Note that everything within curly braces in DCG's is treated as a standard Prolog code, i.e., curly braces would be simply dropped and the code within them would be executed. Constraint solving is done using the CLP(R) system (after dropping a pair of braces; since it is the convention in most CLP(R) systems to put constraints inside a pair of curly braces).

For each production in the timed grammar, the method replaces each component (starting with the left hand side) with its corresponding timed component as explained above, advances the clock if necessary, passes on the wall clock time and last reset time to the next component, and repeats this step until the end of production rule is reached. If T represents the wall clock time, then time is advanced by posting the constraint $T' > T$ where T' represents the current wall clock time. Advancing the clock is necessary since the

underlying assumption is that multiple symbols of the timed string are not seen at the same instant. if (a, t_1) is immediately followed by (b, t_2) in a timed string, then $t_2 > t_1$. Thus the wall clock should be advanced after each symbol in the timed string.

To illustrate the process, we describe the logic programming rendering of the timed ω-grammar shown in Example 2.1 in section 2.

```
:- coinductive(s/6).
s(T, Ti, Tn, To) --> r(T, Ti, T1, To1), {{T2 > T1}}, s(T2, To1, Tn, To).
r(T, Ti, Tn, To) --> [(a, T)],    {{Ti = T, T1 > T}}, t(T1, Ti, T2, To),
                     {{Tn > T2}}, [(b, Tn)],  {{Tn - To < 20}}.
t(T, Ti, Tn, To) --> [(a, T)],    {{T1 > T}}, t(T1, Ti, T2, To), {{Tn > T2}},
                     [(b, Tn)].
t(T, Ti, Tn, To) --> [(a, T)], {{Tn > T}}, [(b, Tn)], {{Tn - Ti < 5, To = Ti}}.
```

Note that the predicates s, r, and t are defined as DCG rules; therefore, two arguments (for the difference lists) will be added to each of them by a Prolog compiler. The first explicit argument of these predicates is the wall clock time; Tn is the new wall clock time after each predicate call. The pair of arguments, Ti and To, represent the clock c of the timed grammar. In fact, a pair of arguments have to be added for each clock that is used in the grammar (in the current example there is only one clock). The first argument of this pair is used to remember the last wall clock time this clock was reset, while the second one is used to pass on this clock's value to the next predicate call. Given this program one can pose queries to it to check if a timed string satisfies the timing constraints imposed in the timed grammar. Alternatively, one can generate possible legal timed strings. Finally, one can verify properties of this timed language (e.g., checking the simple property that all the a's are generated within 5 units of time, in any timed string that is accepted).

The co-inductive termination of predicate $s/6$ will depend only on the two arguments (for the difference lists) that are added by the Prolog compiler, i.e., the wall-clock time and other arguments will be ignored when checking if the $s/6$ predicate is cyclical. In the Co-LP system we have used for our work [2], one can declare the arguments w.r.t. which a predicate should behave co-inductively. Only those arguments will be employed by the system for determining co-inductive termination for that predicate. For truly co-inductive termination, the constraints induced in a given cycle in the grammar should also be taken into account: one must ensure that if a cycle `P` is part of an accepting string, then the constraints generated in one traversal of cycle `P` must be entailed by those generated in the next traversal of `P`. This is indeed the case for practical timed systems, where all clocks involved are reset in every accepting cycle. Due to this resetting, the same constraints are repeated in every such cycle and therefore co-inductive termination is justified w.r.t. constraints also.

Next we illustrate applications of our co-inductive CLP(R) realization of ω-TCFGs by using it for the *controller* component of the GRC problem with two tracks.

4. Timed ω-Grammar for GRC

Informally, the GRC problem [Hei94] describes a railroad crossing system with several tracks and an unspecified number of trains traveling through the tracks in both directions. There is a gate at the railroad crossing that should be operated in a way that guarantees the *safety* and *utility* properties. The safety property stipulates that the gate must be down

[2]The interpreter for Co-LP that we have used is based on YAP, and can be found in `http://www.utdallas.edu/~nxs048000/co-lp.yap`

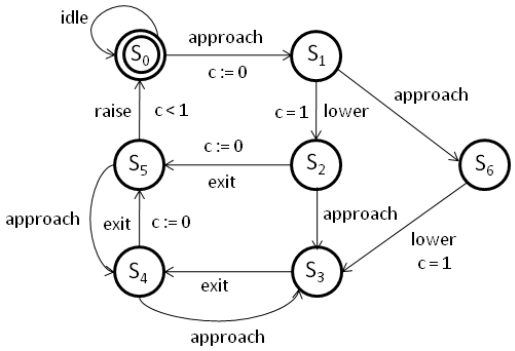


Figure 1: The controller for the GRC with two tracks

while one or more trains are in the crossing. The utility property states that the gate must be up when there is no train in the crossing. The system is composed of three subsystems: a gate, a set of tracks and an overall controller. We show a timed context-free ω-grammar for the *controller* component of the GRC system with two tracks. The behavior of the controller can be expressed graphically via a timed push down automaton (Figure 1), and described using the ω-TCFG shown below:

$C \rightarrow approach\ \{c := 0\}\ L\ exit\ \{c := 0\}\ raise\ \{c < 1\}\ C$
$C \rightarrow approach\ \{c := 0\}\ L\ N\ exit\ \{c := 0\}\ raise\ \{c < 1\}\ C$
$L \rightarrow lower\ \{c < 1\}$
$L \rightarrow approach\ lower\ \{c < 1\}\ exit$
$N \rightarrow approach\ exit$
$N \rightarrow approach\ exit\ N$
$N \rightarrow exit\ approach$
$N \rightarrow exit\ approach\ N$

The logic programming rendering of this ω-TCFG is presented below.

```
c(T, Ti, Tn, To) --> [(approach, T)], {{Ti = T, T1 > T}},    l(T1, Ti, T2, To1),
                     {{T3 > T2}}, [(exit, T3)], {{To2 = T3, T4 > T3}},
                     [(raise, T4)], {{T4 - To2 <1, T5 > T4}}, c(T5, T5, Tn, To).
c(T, Ti, Tn, To) --> [(approach, T)], {{Ti = T, T1 > T}},    l(T1, Ti, T2, To1),
                     {{T3 > T2}}, n(T3, To1, T4, To2), {{T5 > T4}},
                     [(exit, T5)], {{To3 = T5, T6 > T5}}, [(raise, T6)],
                     {{T6 - To3 < 1, T7 > T6}}, c(T7, T7, Tn, To).
l(T, Ti, Tn, To) --> [(lower, T)],    {{T - Ti < 1, To = Ti, Tn = T}}.
l(T, Ti, Tn, To) --> [(approach, T)], {{T1 > T}}, [(lower, T1)],
                     {{T1 - Ti < 1, Tn > T1}},    [(exit, Tn)], {{To = Ti}}.
n(T, Ti, Tn, To) --> [(approach, T)], {{Tn > T}}, [(exit, Tn)], {{To = Ti}}.
n(T, Ti, Tn, To) --> [(approach, T)], {{T1 > T}}, [(exit, T1)], {{T2 > T1}},
                     n(T2, Ti, Tn, To).
n(T, Ti, Tn, To) --> [(exit, T)], {{Tn > T}}, [(approach, Tn)], {{To = Ti}}.
n(T, Ti, Tn, To) --> [(exit, T)], {{T1 > T}}, [(approach, T1)], {{T2 > T1}},
                     n(T2, Ti, Tn, To).
```

The ω-TCFGs for *gate* and *track* components of the GRC problem along with their corresponding logic programs can be generated in a similar manner.

5. Conclusions and Future Work

In this paper we have extended context-free grammars with clocks and clock expressions. The resulting grammars, called timed CFGs, are means of describing complex languages consisting of timed words, where a timed word is a symbol from the alphabet of the language the grammar generates, paired with the time that symbol was seen. As a result, timed CFGs are suitable for specifying systems whose behavior can be described by recursive structures with time constraints. We have developed a system for generating parsers for timed context-free ω-grammars using the DCG facility of logic programming coupled with CLP(R) and co-induction. We have applied our method of generating parsers for ω-TCFGs to the GRC problem with two tracks, and presented a simple timed context-free ω-grammar for the *controller* component of this problem.

To conclude, a combination of constraint over reals, co-induction, and the language processing capabilities of logic programming provides an elegant and expressive formalism for describing real-time, hybrid, and cyber-physical systems. In fact, our framework is a general framework that can be used for handling not only time but other continuous physical quantities as well.

Acknowledgment

The authors would like to thank D. T. Huynh, Kevin Hamlen, and Feliks Kluźniak for discussions.

References

[Alu90] Rajeev Alur and David L. Dill. Automata for modeling real-time systems. In *ICALP*, *Lecture Notes in Computer Science*, vol. 443, pp. 322–335. Springer, 1990.

[Alu94] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[Bar96] Jon Barwise and Lawrence Moss. *Vicious circles: on the mathematics of non-wellfounded phenomena*. Center for the Study of Language and Information, Stanford, CA, USA, 1996.

[Gup06] Rajesh Gupta. Programming models and methods for spatiotemporal actions and reasoning in cyber-physical systems. In *NSF Workshop on CPS*. 2006.

[Gup07] Gopal Gupta, Ajay Bansal, Richard Min, Luke Simon, and Ajay Mallya. Coinductive logic programming and its applications. In *ICLP*, pp. 27–44. Springer, 2007.

[Hei94] Constance L. Heitmeyer and Nancy A. Lynch. The generalized railroad crossing: A case study in formal verification of real-time systems. In *IEEE RTSS*, pp. 120–131. 1994.

[Jaf94] Joxan Jaffar and Michael J. Maher. Constraint logic programming: A survey. *J. Log. Program.*, 19/20:503–581, 1994.

[Lee08] Edward A. Lee. Cyber-physical systems: Design challenges. In *ISORC*. 2008.

[Llo87] J. W. Lloyd. *Foundations of logic programming / J.W. Lloyd*. Springer-Verlag, Berlin, New York, 2nd edn., 1987.

[Sae] Neda Saeedloei and Gopal Gupta. Logic programming foundations of cyber-physical systems. In Preparation.

[Sim06a] L. Simon. *Coinductive Logic Programming*. Ph.D. thesis, University of Texas at Dallas, Richardson, Texas, 2006.

[Sim06b] Luke Simon, Ajay Mallya, Ajay Bansal, and Gopal Gupta. Coinductive logic programming. In *ICLP*, pp. 330–345. 2006.

[Sim07] Luke Simon, Ajay Bansal, Ajay Mallya, and Gopal Gupta. Co-logic programming: Extending logic programming with coinduction. In *ICALP*, pp. 472–483. 2007.

[Tho90] Wolfgang Thomas. Automata on infinite objects. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pp. 133–192. MIT Press, 1990.

Technical Communications of the International Conference on Logic Programming, 2010 (Edinburgh), pp. 222–225
http://www.floc-conference.org/ICLP-home.html

TOWARDS A PARALLEL VIRTUAL MACHINE FOR FUNCTIONAL LOGIC PROGRAMMING

ABDULLA ALQADDOUMI

New Mexico State University, Computer Science Department,
P.O. Box 30001, MSC CS, Las Cruces, NM 88003, USA
E-mail address: aalqaddo@cs.nmsu.edu

Abstract. Functional logic programming is a multi-paradigm programming that combines the best features of functional programming and logic programming. Functional programming provides mechanisms for demand-driven evaluation, higher order functions and polymorphic typing. Logic programming deals with non-determinism, partial information and constraints. Both programming paradigms fall under the umbrella of declarative programming. For the most part, the current implementations of functional logic languages belong to one of two categories: (1) Implementations that include the logic programming features in a functional language. (2) Implementations that extend logic languages with functional programming features. In this paper we describe the undergoing research efforts to build a parallel virtual machine that performs functional logic computations. The virtual machine will tackle several issues that other implementations do not tackle: (1) Sharing of sub-terms among different terms especially when such sub-terms are evaluated to more than one value (non-determinism). (2) Exploitation of all forms of parallelism present in computations. The evaluation strategy used to evaluate functional logic terms is needed narrowing, which is a complete and sound strategy.

1. Introduction

Functional logic programming is a multi-paradigm programming that combines the best features of functional programming and logic programming. Functional programming provides mechanisms for demand-driven evaluation, higher order functions and polymorphic typing. Logic programming deals with non-determinism, partial information and constraints. Both programming paradigms fall under the umbrella of declarative programming.

For the most part, the current implementations of functional logic languages belong to one of two categories: (1) Implementations that include the logic programming features in a functional language. (2) Implementations that extend logic languages with functional programming features. Interested readers are referred to [Han07] for a survey on such languages and implementations.

In this paper we describe the undergoing research efforts to build a parallel virtual machine that performs functional logic computations. The virtual machine will tackle several issues that other implementations do not tackle: (1) Sharing of sub-terms among different terms especially when such sub-terms are evaluated to more than one value (non--determinism). (2) Exploitation of all forms of parallelism present in computations. The

Key words and phrases: functional logic programming; term rewriting system; non-determinism; needed narrowing; and-parallelism; or-parallelism.

Technical Communications of the 26th International Conference on Logic Programming, Edinburgh, July, 2010
Editors: Manuel Hermenegildo, Torsten Schaub
LIPIcs - Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/LIPIcs.ICLP.2010.222

evaluation strategy used to evaluate functional logic terms is needed narrowing [Ant00], which is a complete and sound strategy.

2. Progress and Research Objectives

2.1. Types of Parallelism

Functional logic computations can exploit and-parallelism, or-parallelism or both. And-parallelism arises when two or more sub-terms need computation in order to compute their ancestor term. Computing such sub-terms simultaneously will decrease the waiting time of their parent. And-parallelism is usually independent, but when the computation of such sub-terms simultaneously involves a shared variable, it becomes dependent. In case of dependencies, synchronization of terms is required. Or-parallelism arises when a choice operator "?" or a variable need computation. The choice operator, "?", and logic or extra variables represent non-determinisim in functional logic computations. Equation (1) below is an example of non-deterministic operation using the choice operator. In this case, "coin" can be replaced by either 0 or 1. The computation of a choice operator in parallel can be done by invoking as many processes as there are arguments and locally for each process, its argument will rewrite the choice with its selected argument. The same thing is true for logic or extra variables.

```
coin = 0 ? 1                (1)
```

2.2. Research Objectives

The goal of my research is to implement a parallel virtual machine that will execute functional logic programs efficiently, specifically for the functional logic language, Curry [Han06]. Curry is a functional logic language that seamlessly integrates the logic and functional features. The evaluation strategy used in the virtual machine will be based on needed narrowing. Needed narrowing is based on the concept that only needed arguments of a term must be selected and evaluated to rewrite the term. Needed arguments can be decided by using definitional trees. For more details about needed narrowing, please refer to [Ant00]. For more details about definitional trees, please refer to [Ant92]. Computing normal forms of terms may involve computation of several sub-terms at the same time (and-parallelism). This is called don't-know non-determinism, because the strategy does not know which sub-term to execute first. A sequential evaluation strategy will compute the sub-terms one after the other. When choices or variables are encountered during the computation of some term, the system can try all such alternatives that will replace the choice or logic variable in order to compute the term in hand (or-parallelism). This is called don't-care non-determinism, because the system will not care which alternative of the choice or variable was chosen to compute the term. Our virtual machine will exploit both (and-parallelism and or-parallelism).

2.3. Progress

The system is implemented for the time being in Ruby. The current implementation includes basic libraries and parsing of expressions with sharing. Ruby supports the use of threads. The evaluation strategy will select needed sub-terms after each rewriting step and different threads will compute those needed sub-terms simultaneously. The first stage of the implementation that is done in Ruby will be used as proof of correctness and the final version of the virtual machine will be developed in C or Java to improve its efficiency.

3. Future Work

Computation of choice operations or logic variables as mentioned earlier leads to non-determinism and having more than one term to rewrite the current choice or variable. Such terms could possibly be shared between more than one parent-term. Therefore, the information about the option that rewrites the term must be global in the whole graph. This will ensure all solutions obtained are correct. We will present several solutions that can ensure the correctness of the solutions obtained from terms where choices or variables are shared within them. All the solutions are based on needed narrowing as a strategy for instantiating variables.

```
double x = x + x          (2)
double coin               (3)
```

In equation (2) the operation "double" will be rewritten as a plus operation. The argument of "double" must be represented as one shared argument between the first and second arguments of plus. In (1) we defined the non-deterministic operation coin that can be rewritten as 0 or 1. In the evaluation of (3), the correct computation must yield the values 0 or 2. If there was no implementation of sharing, equation (3) would be rewritten as "coin + coin". Such term would evaluate to any of the four values, 0, 1, 1, or 2.

3.1. Stack Copying

This evaluation strategy will perform computations of deterministic terms normally. When a non-deterministic term is encountered (choice or variable), separate contexts of the graph will be managed by different processes. There will be as many environments as the number of the arguments of the choice or the variable. This solution is inspired from strategies of implementing or-trees in logic programming [Gup01]. This approach will be very expensive in terms of memory but optimizations can be done. The other main drawback is the re-computation of terms. In the evaluation of (3), two separate environments will be created to accommodate each argument of the operation "coin": "double 0" and "double 1". Note that in case non-deterministic terms appear on both sides of the tree, re-computations of such non-deterministic terms may not be avoidable.

3.2. Fingerprinting

This evaluation strategy is built on replacing choices or variables with sets. These sets will contain all the arguments of the choice or values of the variable's domain. Each element of the set will be tagged with a unique fingerprint that will identify the origin of the element. The fingerprint consists of two parts, parent and position tags. Whenever a set contributes in other computations, compatibility of fingerprints will be checked to avoid

inconsistency. Two terms are said to be incompatible if they have the same parent tag but different position tag. The drawback of such approach is the overhead of bookkeeping of fingerprints and the compatibility tests performed.

In the evaluation of (3), the following set will be created to replace the non–deterministic operation "coin": $\{0^{A1}, 1^{A2}\}$. The first element of the set, "0", will be tagged with the fingerprint A1, and the second element of the set, "1", will be tagged with the fingerprint A2. The "plus" operation will perform the compatibility check between the two sets: $\{0^{A1}, 1^{A2}\}$ and $\{0^{A1}, 1^{A2}\}$. The resulting set will become $\{0^{A1} + 0^{A1}$, $\cancel{0^{A1} + 1^{A2}}$, $\cancel{1^{A2} + 0^{A1}}$, $1^{A2} + 1^{A2}\}$ ==> $\{0^{A1} + 0^{A1}, 1^{A2} + 1^{A2}\}$ ==> $\{0^{A1}, 2^{A2}\}$. The two stroked options refer to incompatibility between the elements and hence removed. Note that the compatibility test is performed before the operation is applied to avoid any unnecessary computations.

3.3. Promotion of Choice or Variable (Bubbling)

This evaluation strategy will promote non-deterministic operations. When a choice or a variable is encountered, the choice or variable will be promoted to take the place of its parent(s) and make as many copies of the parent(s) as the number of the arguments of the choice or variable. In case a choice has more than one parent, a nearest common ancestor between such parents will be computed and the spine that connects both parents with the common ancestor will be copied. All copies of the spine will be identical in their arguments except for the argument that had the choice which will be replaced by one of the choice arguments. Eventually the choice will reach the root of the graph. A rewriting of (3) in this case will be "double coin ==> (double 0) ? (double 1) ==> (0 + 0) ? (1 + 1) ==> 0 ? 2". Use of Promotion is inspired from [Ant07, Jan94, Lop97].

References

[Ant92] Sergio Antoy. Definitional trees. In *In Proc. of the 3rd International Conference on Algebraic and Logic Programming*, pp. 143–157. Springer LNCS, 1992.

[Ant00] Sergio Antoy, Rachid Echahed, and Michael Hanus. A needed narrowing strategy. *J. ACM*, 47(4):776–822, 2000.

[Ant07] Sergio Antoy, Daniel W. Brown, and Su-Hui Chiang. Lazy context cloning for non-deterministic graph rewriting. *Electron. Notes Theor. Comput. Sci.*, 176(1):3–23, 2007.

[Gup01] Gopal Gupta, Enrico Pontelli, Khayri A.M. Ali, Mats Carlsson, and Manuel V. Hermenegildo. Parallel execution of prolog programs: a survey. *ACM Trans. Program. Lang. Syst.*, 23(4):472–602, 2001.

[Han06] M. Hanus (ed.). *Curry: An Integrated Functional Logic Language (Vers. 0.8.2)*. 2006. Available at `http://www.informatik.uni-kiel.de/~curry`.

[Han07] M. Hanus. Multi-paradigm declarative languages. In *Proceedings of the International Conference on Logic Programming (ICLP 2007)*, pp. 45–75. Springer LNCS 4670, 2007.

[Jan94] Sverker Janson. *AKL - A Multiparadigm Programming Language*. Ph.D. thesis, Uppsala University, SICS, 1994.

[Lop97] Ricardo Lopes and Vitor Santos Costa. The BEAM: Towards a first EAM implementation, 1997.

Technical Communications of the International Conference on Logic Programming, 2010 (Edinburgh), pp. 226–235
http://www.floc-conference.org/ICLP-home.html

DYNAMIC MAGIC SETS FOR DISJUNCTIVE DATALOG PROGRAMS

MARIO ALVIANO

Department of Mathematics, University of Calabria — 87036 Rende (CS), Italy
E-mail address: alviano@mat.unical.it

Abstract. Answer set programming (ASP) is a powerful formalism for knowledge representation and common sense reasoning that allows disjunction in rule heads and nonmonotonic negation in bodies. Magic Sets are a technique for optimizing query answering over logic programs and have been originally defined for standard Datalog, that is, ASP without disjunction and negation. Essentially, the input program is rewritten in order to identify a subset of the program instantiation which is sufficient for answering the query.

Dynamic Magic Sets (DMS) are an extension of this technique to ASP. The optimization provided by DMS can be exploited also during the nondeterministic phase of ASP systems. In particular, after some assumptions have been made during the computation, parts of the program may become irrelevant to a query (because of these assumptions). This allows for dynamic pruning of the search space, which may result in exponential performance gains.

DMS has been implemented in the DLV system and experimental results confirm the effectiveness of the technique.

Introduction

Answer set programming (ASP) is a powerful formalism for knowledge representation and common sense reasoning [Bar03]. Allowing disjunction in rule heads and nonmonotonic negation in bodies, ASP can express every query belonging to the complexity class Σ_2^{P} ($\mathrm{NP}^{\mathrm{NP}}$); the same expressive power is preserved even if negation is restricted to be used in a stratified way [Eit94].

Magic Sets are a technique for optimizing query answering over logic programs. ASP computations are typically characterized by two phases, namely *program instantiation* and *answer set search*. Program instantiation is deterministic and transforms the input program into an equivalent one with no variables. Answer set search is nondeterministic in general and works on the instantiated program.

Magic Sets have been originally defined for standard Datalog, that is, ASP without disjunction and negation. Essentially, the input program is rewritten in order to identify a subset of the program instantiation which is sufficient for answering the query. The

1998 ACM Subject Classification: Logic and constraint programming.

Key words and phrases: answer set programming, decidability, magic sets, disjunctive logic programs.

Thanks: The author is grateful to Wolfgang Faber, Gianluigi Greco and Nicola Leone for the fundamental contribution in achieving the results summarized in this article and reported in [Alv09]. This research has been partly supported by Regione Calabria and EU under POR Calabria FESR 2007-2013 within the PIA project of DLVSYSTEM s.r.l., and by MIUR under the PRIN project LoDeN.

Technical Communications of the 26th International Conference on Logic Programming, Edinburgh, July, 2010
Editors: Manuel Hermenegildo, Torsten Schaub
LIPIcs - Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/LIPIcs.ICLP.2010.226

restriction of the instantiation is obtained by means of additional "magic" predicates, whose extensions represent relevant atoms w.r.t. the query.

An attempt to extend the method to (disjunctive) ASP has been done in [Gre03]. Magic set predicates of [Gre03] have a deterministic definition and, consequently, have the same extension in each answer set. Actually, this extension can always be computed during program instantiation, and so we call the technique of [Gre03] Static Magic Sets (SMS).

In the context of (disjunctive) ASP, there is no reason for having a deterministic definition of magic predicates. Indeed, while Datalog programs admit exactly one answer set, ASP programs can have several answer sets, each one representing a different, plausible scenario. Since atoms relevant in one scenario could be irrelevant in another (or also in each other), one expects that Magic Sets should capture this aspect and provide a *dynamic* optimization to the answer set search.

Our principal contributions concerning Magic Sets for ASP are stated below.

- We have defined Dynamic Magic Sets (DMS). With DMS, ASP computations can exploit the information provided by magic set predicates also during the nondeterministic answer set search, allowing for potentially exponential performance gains w.r.t. SMS. Indeed, the definition of our magic set predicates depends on the assumptions made during the computation, identifying the atoms that are relevant in the current (partial) scenario.
- We have established the correctness of DMS by proving that the transformed program is query-equivalent to the original program and we have highlighted a strong relationship between magic sets and unfounded sets: The atoms that are relevant w.r.t. an answer set are either true or form an unfounded set.
- We have implemented DMS in the DLV system and compared the performance of DLV with no magic sets, with SMS, and with DMS. The experimental results show that in many cases DMS yields a significant performance benefit. The system is available at `http://www.dlvsystem.com/magic/`.

The remainder of the paper is structured as follows. In Section 1, syntax and semantics of ASP are briefly mentioned. Dynamic Magic Sets for stratified ASP programs are introduced in Section 2. In Section 3, the implemented prototype system is briefly presented, while experimental result are discussed in Section 4. Finally, in Section 5, we draw our conclusion and discuss about future work we intend to address.

1. Answer Set Programming

In this section, we recall syntax and semantics of disjunctive ASP with stratified negation, the language for which we will introduce Dynamic Magic Sets in Section 2.2.

1.1. Syntax

A *term* is either a *variable* or a *constant*. If $\mathtt{p}$ is a *predicate* of arity $k \geq 0$, and $\mathtt{t}_1, \ldots, \mathtt{t}_k$ are terms, then $\mathtt{p}(\mathtt{t}_1, \ldots, \mathtt{t}_k)$ is an *atom*[1]. A *literal* is either an atom $\mathtt{p}(\bar{\mathtt{t}})$ (a positive literal), or an atom preceded by the *negation as failure* symbol `not` $\mathtt{p}(\bar{\mathtt{t}})$ (a negative literal). A *rule* r is of the form

$$\mathtt{p}_1(\bar{\mathtt{t}}_1) \ \mathtt{v} \ \cdots \ \mathtt{v} \ \mathtt{p}_n(\bar{\mathtt{t}}_n) \ \mathtt{:-} \ \mathtt{q}_1(\bar{\mathtt{s}}_1), \ \ldots, \ \mathtt{q}_j(\bar{\mathtt{s}}_j), \ \mathtt{not} \ \mathtt{q}_{j+1}(\bar{\mathtt{s}}_{j+1}), \ \ldots, \ \mathtt{not} \ \mathtt{q}_m(\bar{\mathtt{s}}_m).$$

[1] We use the notation $\bar{\mathtt{t}}$ for a sequence of terms, for referring to atoms as $\mathtt{p}(\bar{\mathtt{t}})$.

where $\mathtt{p_1}(\bar{\mathtt{t}}_1)$, ..., $\mathtt{p_n}(\bar{\mathtt{t}}_n)$, $\mathtt{q_1}(\bar{\mathtt{s}}_1)$, ..., $\mathtt{q_m}(\bar{\mathtt{s}}_m)$ are atoms and $n \geq 1$, $m \geq j \geq 0$. The disjunction $\mathtt{p_1}(\bar{\mathtt{t}}_1)$ $\mathtt{v}$ $\cdots$ $\mathtt{v}$ $\mathtt{p_n}(\bar{\mathtt{t}}_n)$ is the *head* of r, while the conjunction $\mathtt{q_1}(\bar{\mathtt{s}}_1)$, ..., $\mathtt{q_j}(\bar{\mathtt{s}}_j)$, $\mathtt{not}\ q_{j+1}(\bar{s}_{j+1})$, ..., $\mathtt{not}\ q_m(\bar{s}_m)$ is the *body* of r. Moreover, $H(r)$ denotes the set of head atoms, while $B(r)$ denotes the set of body literals. We also use $B^+(r)$ and $B^-(r)$ for denoting the set of atoms appearing in positive and negative body literals, respectively, and $Atoms(r)$ for the set $H(r) \cup B^+(r) \cup B^-(r)$. Rules are assumed to be safe, that is, each variable appearing in a rule r also appears in $B^+(r)$. A rule r is positive (or negation-free) if $B^-(r) = \emptyset$, a *fact* if both $B(r) = \emptyset$ and $|H(r)| = 1$.

A *program* $\mathcal{P}$ is a finite set of rules; if all rules in it are positive, then $\mathcal{P}$ is a positive program. Stratified programs constitute another interesting class of programs. A predicate $\mathtt{p}$ appearing in the head of a rule r *depends* on each predicate $\mathtt{q}$ such that an atom $\mathtt{q}(\bar{\mathtt{s}})$ belongs to $B(r)$; if $\mathtt{q}(\bar{\mathtt{s}})$ belongs to $B^+(r)$, $\mathtt{p}$ depends on $\mathtt{q}$ positively, otherwise negatively. A program is *stratified* if each cycle of dependencies involves only positive dependencies.

1.2. Semantics

Given a predicate $\mathtt{p}$, a *defining rule* for $\mathtt{p}$ is a rule r such that some atom $\mathtt{p}(\bar{\mathtt{t}})$ belongs to $H(r)$. If all defining rules of a predicate $\mathtt{p}$ are facts, then $\mathtt{p}$ is an *EDB predicate*; otherwise $\mathtt{p}$ is an *IDB predicate*[2]. Given a program $\mathcal{P}$, the set of rules having some IDB predicate in head is denoted by $IDB(\mathcal{P})$, while $EDB(\mathcal{P})$ denotes the remaining rules, that is, $EDB(\mathcal{P}) = \mathcal{P} \setminus IDB(\mathcal{P})$.

The set of constants appearing in a program $\mathcal{P}$ is the *universe* of $\mathcal{P}$ and is denoted by $U_\mathcal{P}$[3], while the set of ground atoms constructible from predicates in $\mathcal{P}$ with elements of $U_\mathcal{P}$ is the *base* of $\mathcal{P}$, denoted by $B_\mathcal{P}$. We call a term (atom, rule, or program) *ground* if it does not contain any variable. A ground atom $\mathtt{p}(\bar{\mathtt{t}})$ (resp. a ground rule r_g) is an instance of an atom $\mathtt{p}(\bar{\mathtt{t}}')$ (resp. of a rule r) if there is a substitution ϑ from the variables in $\mathtt{p}(\bar{\mathtt{t}}')$ (resp. in r) to $U_\mathcal{P}$ such that $\mathtt{p}(\bar{\mathtt{t}}) = \mathtt{p}(\bar{\mathtt{t}}')\vartheta$ (resp. $r_g = r\vartheta$). Given a program $\mathcal{P}$, $Ground(\mathcal{P})$ denotes the set of all instances of the rules in $\mathcal{P}$.

An *interpretation* I for a program $\mathcal{P}$ is a subset of $B_\mathcal{P}$. A positive ground literal $\mathtt{p}(\bar{\mathtt{t}})$ is true w.r.t. an interpretation I if $\mathtt{p}(\bar{\mathtt{t}}) \in I$; otherwise, it is false. A negative ground literal $\mathtt{not}\ \mathtt{p}(\bar{\mathtt{t}})$ is true w.r.t. I if and only if $\mathtt{p}(\bar{\mathtt{t}})$ is false w.r.t. I. The body of a ground rule r_g is true w.r.t. I if and only if all the body literals of r_g are true w.r.t. I, that is, if and only if $B^+(r_g) \subseteq I$ and $B^-(r_g) \cap I = \emptyset$. An interpretation I *satisfies* a ground rule $r_g \in Ground(\mathcal{P})$ if at least one atom in $H(r_g)$ is true w.r.t. I whenever the body of r_g is true w.r.t. I. An interpretation I is a *model* of a program $\mathcal{P}$ if I satisfies all the rules in $Ground(\mathcal{P})$.

Given an interpretation I for a program $\mathcal{P}$, the reduct of $\mathcal{P}$ w.r.t. I, denoted $Ground(\mathcal{P})^I$, is obtained by deleting from $Ground(\mathcal{P})$ all the rules r_g with $B^-(r_g) \cap I = \emptyset$, and then by removing all the negative literals from the remaining rules. The semantics of a program $\mathcal{P}$ is then given by the set $\mathcal{AS}(\mathcal{P})$ of the answer sets of $\mathcal{P}$, where an interpretation M is an answer set for $\mathcal{P}$ if and only if M is a subset-minimal model of $Ground(\mathcal{P})^M$.

Given a ground atom $\mathtt{p}(\bar{\mathtt{t}})$ and a program $\mathcal{P}$, $\mathtt{p}(\bar{\mathtt{t}})$ is a cautious (resp. brave) consequence of $\mathcal{P}$, denoted by $\mathcal{P} \models_c \mathtt{p}(\bar{\mathtt{t}})$ (resp. $\mathcal{P} \models_b \mathtt{p}(\bar{\mathtt{t}})$), if $\mathtt{p}(\bar{\mathtt{t}}) \in M$ for each (resp. some) $M \in$

[2] *EDB* and *IDB* stand for Extensional Database and Intensional Database, respectively.

[3] If $\mathcal{P}$ has no constants, then an arbitrary constant is added to $U_\mathcal{P}$.

$\mathcal{AS}(\mathcal{P})$. Given a *query* $\mathcal{Q} = \mathtt{g}(\bar{\mathtt{t}})$? (an atom)[4], $Ans_c(\mathcal{Q},\mathcal{P})$ (resp. $Ans_b(\mathcal{Q},\mathcal{P})$) denotes the set of all the substitutions ϑ for the variables of $\mathtt{g}(\bar{\mathtt{t}})$ such that $\mathcal{P} \models_c \mathtt{g}(\bar{\mathtt{t}})\vartheta$ (resp. $\mathcal{P} \models_b \mathtt{g}(\bar{\mathtt{t}})\vartheta$). Two programs $\mathcal{P}$ and $\mathcal{P}'$ are cautious-equivalent (resp. brave-equivalent) w.r.t. a query $\mathcal{Q}$, denoted by $\mathcal{P} \equiv^c_{\mathcal{Q}} \mathcal{P}'$ (resp. $\mathcal{P} \equiv^b_{\mathcal{Q}} \mathcal{P}'$), if $Ans_c(\mathcal{Q},\mathcal{P}\cup\mathcal{F}) = Ans_c(\mathcal{Q},\mathcal{P}'\cup\mathcal{F})$ (resp. $Ans_b(\mathcal{Q},\mathcal{P}\cup\mathcal{F}) = Ans_b(\mathcal{Q},\mathcal{P}'\cup\mathcal{F})$) is guaranteed for each set of facts $\mathcal{F}$ defined over the EDB predicates of $\mathcal{P}$ and $\mathcal{P}'$.

2. Magic Sets Techniques

In this section, we first briefly discuss about Magic Sets in the literature; we then introduce Dynamic Magic Sets, our proposal for extending the standard technique to ASP.

2.1. Overview of the Existing Literature

The Magic Set method is a strategy for simulating the top-down evaluation of a query by modifying the original program by means of additional rules, which narrow the computation to what is relevant for answering the query.

The key idea of Magic Sets is to materialize the binding information for IDB predicates that would be propagated during a top-down computation, like for instance the one adopted by Prolog. According to this kind of evaluation, all the rules r such that $\mathtt{g}(\bar{\mathtt{t}}') \in H(r)$ (where $\mathtt{g}(\bar{\mathtt{t}}')\vartheta = \mathcal{Q}$ for some substitution ϑ) are considered in a first step. Then, the atoms in $Atoms(r\vartheta)$ different from $\mathcal{Q}$ are considered as new queries and the procedure is iterated. Note that during this process the information about *bound* (i.e. non-variable) arguments in the query is "passed" to the other atoms in the rule. Moreover, it is assumed that the rule is processed in a certain sequence, and processing an atom may bind some of its arguments for subsequently considered atoms, thus "generating" and "passing" bindings. Therefore, whenever an atom is processed, each of its argument is considered to be either *bound* (`b`) or *free* (`f`).

The specific propagation strategy adopted in a top-down evaluation scheme is called *sideways information passing strategy* (SIPS), which is just a way of formalizing a partial ordering over the atoms of each rule together with the specification of how the bindings originate and propagate [Bee91, Gre03].

The first attempt to extend Magic Sets to disjunctive Datalog programs is due to [Gre03]. Magic predicates of [Gre03] identify a sizeable superset of all the atoms relevant to answer the given query. An important observation is that this set is defined in a deterministic way, which means that assumptions during the computation cannot be exploited for restricting the relevant part of the program. In terms of bottom-up systems, this implies that the optimization affects only the grounding portion of the solver. For this reason, we refer to the method of [Gre03] as *Static Magic Sets* (SMS).

Intuitively, it would be beneficial to also have a form of conditional relevance, exploiting also relevance for assumptions.[5] In the following, we propose a novel Magic Set method that guarantees semantic equivalence and also allows for the exploitation of conditional or dynamic relevance, overcoming a major drawback of SMS.

[4]More complex queries can still be expressed using appropriate rules. We assume that each constant appearing in $\mathcal{Q}$ also appears in $\mathcal{P}$; if this is not the case, then we can add to $\mathcal{P}$ a fact $\mathtt{p}(\bar{\mathtt{t}})$ such that $\mathtt{p}$ is a predicate not occurring in $\mathcal{P}$ and $\bar{\mathtt{t}}$ are the arguments of $\mathcal{Q}$.

[5]Experimental evidence for this intuition is provided in Section 4.

2.2. Dynamic Magic Sets

Our proposal to extend Magic Sets to (disjunctive) ASP relies on the observation that atoms relevant in one answer set could be irrelevant in another (or also in each other). DMS capture this aspect, providing a *dynamic* optimization to the answer set search.

In order to properly describe the proposed Magic Set method, we need some additional definition and notation. First, we can materialize the binding information for IDB predicates by means of adorned atoms.

Definition 2.1 (Adorned atom)**.** Let $\mathtt{p}(\mathtt{t}_1, \ldots, \mathtt{t}_k)$ be an atom and $\alpha = \alpha_1 \cdots \alpha_k$ a string of the alphabet $\{\mathtt{b}, \mathtt{f}\}$. Then $\mathtt{p}^\alpha(\mathtt{t}_1, \ldots, \mathtt{t}_k)$ is the adorned version of $\mathtt{p}(\mathtt{t}_1, \ldots, \mathtt{t}_k)$ in which $\mathtt{t}_i$ is considered either *bound* if α_i is $\mathtt{b}$, or *free* if α_i is $\mathtt{f}$.

Adorned atoms are then associated with magic atoms, which will be used for identifying those atoms that are relevant for answering the input query.

Definition 2.2 (Magic atom)**.** For an adorned atom $\mathtt{p}^\alpha(\bar{\mathtt{t}})$, let $\mathtt{magic}(\mathtt{p}^\alpha(\bar{\mathtt{t}}))$ be its *magic version* defined as the atom $\mathtt{magic\_p}^\alpha(\bar{\mathtt{t}}')$, where $\bar{\mathtt{t}}'$ is obtained from $\bar{\mathtt{t}}$ by eliminating all arguments corresponding to an $\mathtt{f}$ label in α, and where $\mathtt{magic\_p}^\alpha$ is a new predicate symbol (for simplicity denoted by attaching the prefix "$\mathtt{magic\_}$" to the predicate symbol $\mathtt{p}^\alpha$).

Finally, we formally define SIPS for (disjunctive) ASP rules.

Definition 2.3 (SIPS)**.** A *SIPS* for a rule r w.r.t. a binding α for an atom $\mathtt{p}(\bar{\mathtt{t}}) \in H(r)$ is a pair $(\prec_r^{\mathtt{p}^\alpha(\bar{\mathtt{t}})}, f_r^{\mathtt{p}^\alpha(\bar{\mathtt{t}})})$, where:

(1) $\prec_r^{\mathtt{p}^\alpha(\bar{\mathtt{t}})}$ is a strict partial order over the atoms in $Atoms(r)$, such that:
  (a) $\mathtt{p}(\bar{\mathtt{t}}) \prec_r^{\mathtt{p}^\alpha(\bar{\mathtt{t}})} \mathtt{q}(\bar{\mathtt{s}})$, for all atoms $\mathtt{q}(\bar{\mathtt{s}}) \in Atoms(r)$ different from $\mathtt{p}(\bar{\mathtt{t}})$;
  (b) for each pair of atoms $\mathtt{q}(\bar{\mathtt{s}}) \in (H(r) \setminus \{\mathtt{p}(\bar{\mathtt{t}})\}) \cup B^-(r)$ and $\mathtt{b}(\bar{\mathtt{z}}) \in Atoms(r)$, $\mathtt{q}(\bar{\mathtt{s}}) \prec_r^{\mathtt{p}^\alpha(\bar{\mathtt{t}})} \mathtt{b}(\bar{\mathtt{z}})$ does not hold; and,

(2) $f_r^{\mathtt{p}^\alpha(\bar{\mathtt{t}})}$ is a function assigning to each atom $\mathtt{q}(\bar{\mathtt{s}}) \in Atoms(r)$ a subset of the variables in $\bar{\mathtt{s}}$—intuitively, those made bound when processing $\mathtt{q}(\bar{\mathtt{s}})$.

The Dynamic Magic Set method is reported in Figure 1. The algorithm exploits a set S for storing all the adorned predicates to be used for propagating the binding of the query and, after all the adorned predicates are processed, outputs a rewritten program $\mathrm{DMS}(\mathcal{Q}, \mathcal{P})$ consisting of a set of *modified* and *magic* rules, stored by means of the sets $modifiedRules_{\mathcal{Q},\mathcal{P}}$ and $magicRules_{\mathcal{Q},\mathcal{P}}$, respectively.

The computation starts by initializing S and $modifiedRules_{\mathcal{Q},\mathcal{P}}$ to the empty set (step *1*). Then, the function ***BuildQuerySeed***$(\mathcal{Q}, \mathcal{P}, S)$ is used for storing the query seed $\mathtt{magic}(\mathtt{g}^\alpha(\bar{\mathtt{t}}))$ in $magicRules_{\mathcal{Q},\mathcal{P}}$, where α is a string having a $\mathtt{b}$ in position i if $\mathtt{t}_i$ is a constant, or an $\mathtt{f}$ if $\mathtt{t}_i$ is a variable. In addition, ***BuildQuerySeed***$(\mathcal{Q}, \mathcal{P}, S)$ adds the adorned predicate $\mathtt{magic\_g}^\alpha$ into the set S.

The core of the algorithm (steps *2–9*) is repeated until the set S is empty, i.e., until there is no further adorned predicate to be propagated. In particular, an adorned predicate $\mathtt{p}^\alpha$ is removed from S (step *3*), and its binding is propagated in each rule of the form

$$r: \ \mathtt{p}(\bar{\mathtt{t}}) \ \mathtt{v} \ \mathtt{p}_1(\bar{\mathtt{t}}_1) \ \mathtt{v} \cdots \mathtt{v} \ \mathtt{p}_n(\bar{\mathtt{t}}_n) \coloneq \mathtt{q}_1(\bar{\mathtt{s}}_1), \ldots, \ \mathtt{q}_j(\bar{\mathtt{s}}_j), \ \mathtt{not} \ \mathtt{q}_{j+1}(\bar{\mathtt{s}}_{j+1}), \ldots, \ \mathtt{not} \ \mathtt{q}_m(\bar{\mathtt{s}}_m).$$

(with $n \geq 0$) having an atom $\mathtt{p}(\bar{\mathtt{t}})$ in the head (note that the rule r is processed as often as head atoms with predicate $\mathtt{p}$ occur; steps *4–8*).

Input: A stratified program $\mathcal{P}$, and a query $\mathcal{Q} = \mathtt{g}(\bar{\mathtt{t}})$?
Output: The optimized program DMS$(\mathcal{Q}, \mathcal{P})$.
var S: **set** of adorned predicates; $modifiedRules_{\mathcal{Q},\mathcal{P}}$,$magicRules_{\mathcal{Q},\mathcal{P}}$: **set** of rules;
begin
1. $S := \emptyset$; $modifiedRules_{\mathcal{Q},\mathcal{P}} := \emptyset$; $magicRules_{\mathcal{Q},\mathcal{P}} := \{\textbf{\textit{BuildQuerySeeds}}(\mathcal{Q}, \mathcal{P}, S)\}$;
2. **while** $S \neq \emptyset$ **do**
3. $\mathtt{p}^\alpha :=$ an element of S; $S := S \setminus \{\mathtt{p}^\alpha\}$;
4. **for each** rule $r \in \mathcal{P}$ and **for each** atom $\mathtt{p}(\bar{\mathtt{t}}) \in H(r)$ **do**
5. r^a:=$\textbf{\textit{Adorn}}(r, \mathtt{p}^\alpha, S)$;
6. $magicRules_{\mathcal{Q},\mathcal{P}} := magicRules_{\mathcal{Q},\mathcal{P}} \bigcup \textbf{\textit{Generate}}(r^a)$;
7. $modifiedRules_{\mathcal{Q},\mathcal{P}} := modifiedRules_{\mathcal{Q},\mathcal{P}} \bigcup \{\, \textbf{\textit{Modify}}(r^a) \,\}$;
8. **end for**
9. **end while**
10. DMS$(\mathcal{Q}, \mathcal{P})$:=$magicRules_{\mathcal{Q},\mathcal{P}} \cup modifiedRules_{\mathcal{Q},\mathcal{P}} \cup EDB(\mathcal{P})$;
11. **return** DMS$(\mathcal{Q}, \mathcal{P})$;
end.

Figure 1: Dynamic Magic Set algorithm (DMS) for stratified programs.

Adornment. The function $\textbf{\textit{Adorn}}(r, \mathtt{p}^\alpha, S)$ implements the adornment of the rule r w.r.t. an (adorned) head atom $\mathtt{p}^\alpha(\bar{\mathtt{t}})$ according to a fixed SIPS $(\prec_r^{\mathtt{p}^\alpha(\bar{\mathtt{t}})}, f_r^{\mathtt{p}^\alpha(\bar{\mathtt{t}})})$ (step *5*). In particular, a variable $\mathtt{X}$ of an IDB atom[6] $\mathtt{q}(\bar{\mathtt{s}})$ in r is bound if and only if either:

(1) $\mathtt{X} \in f_r^{\mathtt{p}^\alpha(\bar{\mathtt{t}})}(\mathtt{q}(\bar{\mathtt{s}}))$ with $\mathtt{q}(\bar{\mathtt{s}}) = \mathtt{p}(\bar{\mathtt{t}})$; or,
(2) $\mathtt{X} \in f_r^{\mathtt{p}^\alpha(\bar{\mathtt{t}})}(\mathtt{b}(\bar{\mathtt{z}}))$ for an atom $\mathtt{b}(\bar{\mathtt{z}}) \in B^+(r)$ such that $\mathtt{b}(\bar{\mathtt{z}}) \prec_r^{\mathtt{p}^\alpha(\bar{\mathtt{t}})} \mathtt{q}(\bar{\mathtt{s}})$ holds.

Therefore, $\textbf{\textit{Adorn}}(r, \mathtt{p}^\alpha, S)$ produces an adorned disjunctive rule r^a from an adorned predicate $\mathtt{p}^\alpha$ and a suitable unadorned rule r (according to the bindings defined in (1) and (2) above), by inserting all newly adorned predicates in S. Hence, the rule r^a is of the form

$$r^a\colon \mathtt{p}^\alpha(\bar{\mathtt{t}})\ \mathtt{v}\ \mathtt{p}_1^{\alpha_1}(\bar{\mathtt{t}}_1)\ \mathtt{v}\ \cdots\ \mathtt{v}\ \mathtt{p}_\mathtt{n}^{\alpha_\mathtt{n}}(\bar{\mathtt{t}}_\mathtt{n}) \mathtt{:-}\ \mathtt{q}_1^{\beta_1}(\bar{\mathtt{s}}_1), \ldots, \mathtt{q}_\mathtt{j}^{\beta_\mathtt{j}}(\bar{\mathtt{s}}_\mathtt{j}), \mathtt{not}\ \mathtt{q}_{\mathtt{j}+1}^{\beta_{\mathtt{j}+1}}(\bar{\mathtt{s}}_{\mathtt{j}+1}), \ldots, \mathtt{not}\ \mathtt{q}_\mathtt{m}^{\beta_\mathtt{m}}(\bar{\mathtt{s}}_\mathtt{m}).$$

Generation. The adorned rules are then used to generate *magic rules* defining *magic predicates*, which represent the atoms relevant for answering the input query (step *6*). The bodies of magic rules contain the atoms required for binding the arguments of some atom, following the adopted SIPS. More specifically, if $\mathtt{q}_\mathtt{i}^{\beta_\mathtt{i}}(\bar{\mathtt{s}}_\mathtt{i})$ is an adorned atom (i.e., β_i is not the empty string) in an adorned rule r^a having $\mathtt{p}^\alpha(\bar{\mathtt{t}})$ in head, $\textbf{\textit{Generate}}(r^a)$ produces a magic rule r^* such that (i) $H(r^*) = \{\mathtt{magic}(\mathtt{q}_\mathtt{i}^{\beta_\mathtt{i}}(\bar{\mathtt{s}}_\mathtt{i}))\}$ and (ii) $B(r^*)$ is the union of $\{\mathtt{magic}(\mathtt{p}^\alpha(\bar{\mathtt{t}}))\}$ and the set of all the atoms $\mathtt{q}_\mathtt{j}^{\beta_\mathtt{j}}(\bar{\mathtt{s}}_\mathtt{j}) \in Atoms(r)$ such that $\mathtt{q}_\mathtt{j}(\bar{\mathtt{s}}_\mathtt{j}) \prec_r^\alpha \mathtt{q}_\mathtt{i}(\bar{\mathtt{s}}_\mathtt{i})$.

Modification. Subsequently, magic atoms are added to the bodies of the adorned rules in order to limit the range of the head variables, thus avoiding the inference of facts which are irrelevant for the query. The resulting rules are called *modified rules* (step *7*).

A modified rule r' is obtained from an adorned rule r^a by adding to its body a magic atom $\mathtt{magic}(\mathtt{p}^\alpha(\bar{\mathtt{t}}))$ for each atom $\mathtt{p}^\alpha(\bar{\mathtt{t}}) \in H(r^a)$ and by stripping off the adornments of the original atoms. Hence, the function $\textbf{\textit{Modify}}(r^a)$ constructs a rule r' of the form

$$r'\colon\ \mathtt{p}(\bar{\mathtt{t}})\ \mathtt{v}\ \mathtt{p}_1(\bar{\mathtt{t}}_1)\ \mathtt{v}\ \cdots\ \mathtt{v}\ \mathtt{p}_\mathtt{n}(\bar{\mathtt{t}}_\mathtt{n}) \mathtt{:-}\ \mathtt{magic}(\mathtt{p}^\alpha(\bar{\mathtt{t}})),\ \mathtt{magic}(\mathtt{p}_1^{\alpha_1}(\bar{\mathtt{t}}_1)),\ \ldots,\ \mathtt{magic}(\mathtt{p}_\mathtt{n}^{\alpha_\mathtt{n}}(\bar{\mathtt{t}}_\mathtt{n})),\ \mathtt{q}_1(\bar{\mathtt{s}}_1),\ \ldots, \mathtt{q}_\mathtt{j}(\bar{\mathtt{s}}_\mathtt{j}),\ \mathtt{not}\ \mathtt{q}_{\mathtt{j}+1}(\bar{\mathtt{s}}_{\mathtt{j}+1}),\ \ldots,\ \mathtt{not}\ \mathtt{q}_\mathtt{m}(\bar{\mathtt{s}}_\mathtt{m}).$$

Finally, after all the adorned predicates have been processed, the algorithm outputs the program DMS$(\mathcal{Q}, \mathcal{P})$ (steps *10–11*).

[6]EDB atoms are always adorned with the empty string.

2.3. Query Equivalence Results

We conclude the presentation of the DMS algorithm by sketching the correctness proof presented in [Alv09], to which we refer for the details. Throughout this section, we use the well established notion of unfounded set for disjunctive programs with negation defined in [Leo97]. Since we deal with total interpretations, represented as the set of atoms interpreted as true, the definition of unfounded set can be restated as follows.

Definition 2.4 (Unfounded sets)**.** Let I be an interpretation for a program $\mathcal{P}$, and $X \subseteq B_\mathcal{P}$ be a set of ground atoms. Then X is an *unfounded set* for $\mathcal{P}$ w.r.t. I if and only if for each ground rule $r_g \in \mathit{Ground}(\mathcal{P})$ with $X \cap H(r_g) \neq \emptyset$, either $(1.a)$ $B^+(r_g) \not\subseteq I$, or $(1.b)$ $B^-(r_g) \cap I \neq \emptyset$, or (2) $B^+(r_g) \cap X \neq \emptyset$, or (3) $H(r_g) \cap (I \setminus X) \neq \emptyset$.

Intuitively, conditions $(1.a)$, $(1.b)$ and (3) check if the rule is satisfied by I regardless of the atoms in X, while condition (2) assures that the rule can be satisfied by taking the atoms in X as false. Therefore, the next theorem immediately follows from the characterization of unfounded sets in [Leo97].

Theorem 2.5. *Let I be an interpretation for a program $\mathcal{P}$. Then, for any answer set $M \supseteq I$ of $\mathcal{P}$, and for each unfounded set X of $\mathcal{P}$ w.r.t. I, $M \cap X = \emptyset$ holds.*

We now prove the correctness of the DMS strategy by showing that it is *sound* and *complete*. In both parts of the proof, we exploit the following set of atoms.

Definition 2.6 (Killed atoms)**.** Given a model M for $\mathrm{DMS}(\mathcal{Q},\mathcal{P})$, and a model $N \subseteq M$ of $\mathit{Ground}(\mathrm{DMS}(\mathcal{Q},\mathcal{P}))^M$, the set $\mathtt{killed}^M_{\mathcal{Q},\mathcal{P}}(N)$ of the *killed atoms* w.r.t. M and N is defined as:

$$\{\, \mathtt{p}(\bar{\mathtt{t}}) \in B_\mathcal{P} \setminus N \mid \text{either } \mathtt{p} \text{ is } \mathit{EDB}, \text{ or some } \mathtt{magic}(\mathtt{p}^\alpha(\bar{\mathtt{t}})) \text{ belongs to } N \,\}.$$

Thus, killed atoms are either false instances of some EDB predicate, or false atoms which are relevant for $\mathcal{Q}$ (since a magic atom exists in N). Therefore, we expect that these atoms are also false in any answer set for $\mathcal{P}$ containing $M \cap B_\mathcal{P}$.

Proposition 2.7. *Let M be a model for $\mathrm{DMS}(\mathcal{Q},\mathcal{P})$, and $N \subseteq M$ a model of the reduct $\mathit{Ground}(\mathrm{DMS}(\mathcal{Q},\mathcal{P}))^M$. Then $\mathtt{killed}^M_{\mathcal{Q},\mathcal{P}}(N)$ is an unfounded set for $\mathcal{P}$ w.r.t. $M \cap B_\mathcal{P}$.*

The soundness of the algorithm for stratified programs is proved by the next lemma.

Lemma 2.8. *Let $\mathcal{Q}$ be a query over a stratified program $\mathcal{P}$. Then, for each answer set M' of $\mathrm{DMS}(\mathcal{Q},\mathcal{P})$, there is an answer set M of $\mathcal{P}$ such that, for every substitution ϑ, $\mathcal{Q}\vartheta \in M$ if and only if $\mathcal{Q}\vartheta \in M'$.*

Proof. Consider the program $\mathcal{P} \cup (M' \cap B_\mathcal{P})$, that is, the program obtained by adding to $\mathcal{P}$ a fact for each atom in $M' \cap B_\mathcal{P}$. Since $\mathcal{P}$ is stratified, there is at least an answer set M for $\mathcal{P} \cup (M' \cap B_\mathcal{P})$. Clearly $M \supseteq M' \cap B_\mathcal{P}$; moreover, we can show that M is an answer set of $\mathcal{P}$ as well (by following [Alv09]). Thus, since $\mathcal{Q}\vartheta$ belongs either to M' or to $\mathtt{killed}^{M'}_{\mathcal{Q},\mathcal{P}}(M')$, for every substitution ϑ, the claim follows by Proposition 2.7. ∎

For proving the completeness of the algorithm we provide a construction for passing from an interpretation for $\mathcal{P}$ to one for $\mathrm{DMS}(\mathcal{Q},\mathcal{P})$.

Definition 2.9 (Magic variant)**.** Let I be an interpretation for $\mathcal{P}$. We define an interpretation $\mathtt{var}^{\infty}_{\mathcal{Q},\mathcal{P}}(I)$ for DMS$(\mathcal{Q},\mathcal{P})$, called the magic variant of I w.r.t. $\mathcal{Q}$ and $\mathcal{P}$, as the fixpoint of the following sequence:

$$\begin{array}{rcl}
\mathtt{var}^{0}_{\mathcal{Q},\mathcal{P}}(I) & = & \mathit{EDB}(\mathcal{P}) \\
\mathtt{var}^{i+1}_{\mathcal{Q},\mathcal{P}}(I) & = & \mathtt{var}^{i}_{\mathcal{Q},\mathcal{P}}(I) \cup \{\mathtt{p}(\bar{\mathtt{t}}) \in I \mid \text{some } \mathtt{magic}(\mathtt{p}^{\alpha}(\bar{\mathtt{t}})) \text{ belongs to } \mathtt{var}^{i}_{\mathcal{Q},\mathcal{P}}(I)\} \\
& \cup & \{\mathtt{magic}(\mathtt{p}^{\alpha}(\bar{\mathtt{t}})) \mid \exists\, r^{*}_{g} \in \mathit{Ground}(\mathrm{DMS}(\mathcal{Q},\mathcal{P})) \text{ such that} \\
& & \qquad \mathtt{magic}(\mathtt{p}^{\alpha}(\bar{\mathtt{t}})) \in H(r^{*}_{g}) \text{ and } B^{+}(r^{*}_{g}) \subseteq \mathtt{var}^{i}_{\mathcal{Q},\mathcal{P}}(I)\}, \quad \forall i \geq 0
\end{array}$$

By definition, for a magic variant $\mathtt{var}^{\infty}_{\mathcal{Q},\mathcal{P}}(I)$ of an interpretation I for $\mathcal{P}$, $\mathtt{var}^{\infty}_{\mathcal{Q},\mathcal{P}}(I) \cap B_{\mathcal{P}} \subseteq I$ holds. More interesting, the magic variant of an answer set for $\mathcal{P}$ is in turn an answer set for DMS$(\mathcal{Q},\mathcal{P})$ preserving the truth/falsity of $\mathcal{Q}\vartheta$, for every substitution ϑ.

Lemma 2.10. *For each answer set M of $\mathcal{P}$, there is an answer set M' of* DMS$(\mathcal{Q},\mathcal{P})$ *(which is the magic variant of M) such that, for every substitution ϑ, $\mathcal{Q}\vartheta \in M$ if and only if $\mathcal{Q}\vartheta \in M'$.*

Proof. We can show that $M' = \mathtt{var}^{\infty}_{\mathcal{Q},\mathcal{P}}(I)$ is an answer set of DMS$(\mathcal{Q},\mathcal{P})$. Thus, since $\mathcal{Q}\vartheta$ belongs either to M' or to $\mathtt{killed}^{M'}_{\mathcal{Q},\mathcal{P}}(M')$, for every substitution ϑ, the claim follows by Proposition 2.7. ∎

From the above lemma, together with Lemma 2.8, the correctness of the Magic Set method with respect to query answering directly follows.

Theorem 2.11. *Let $\mathcal{Q}$ be a query over a stratified program $\mathcal{P}$. Then both* DMS$(\mathcal{Q},\mathcal{P}) \equiv^{b}_{\mathcal{Q}} \mathcal{P}$ *and* DMS$(\mathcal{Q},\mathcal{P}) \equiv^{c}_{\mathcal{Q}} \mathcal{P}$ *hold.*

3. Implementation

DMS has been implemented and integrated into the core of the DLV [Leo04] system. The DMS algorithm is applied automatically by default when the user invokes DLV with `-FB` (brave reasoning) or `-FC` (cautious reasoning) together with a (partially) bound query. Magic Sets are not applied by default if the query does not contain any constant. The user can modify this default behaviour by specifying the command-line options `-ODMS` (for applying Magic Sets) or `-ODMS-` (for disabling magic sets). If a completely bound query is specified, DLV can print the magic variant of the answer set (not displaying magic predicates), which witnesses the truth (for brave reasoning) or the falsity (for cautious reasoning) of the query, by specifying the command-line option `--print-model`.

An executable of the DLV system supporting the Magic Set optimization is available at `http://www.dlvsystem.com/magic/`.

4. Experimental Results

In order to evaluate the impact of the proposed method, we have compared DMS both with the traditional DLV evaluation without Magic Sets and with the SMS rewriting. We considered several benchmarks, including an application scenario that has received considerable attention in recent years, the problem of answering user queries over possibly inconsistent databases originating from integration of autonomous sources. Below, we briefly discuss Conformant Plan Checking, a representative benchmark of our experimentation.

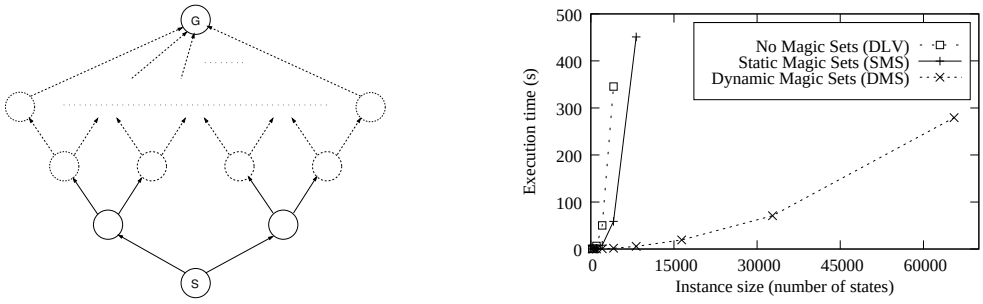


Figure 2: *Conformant Plan Checking*: instance structure and average execution time.

Definition 4.1 (Conformant Plan Checking [Gol96])**.** A state transition diagram and a plan are given. A plan is a sequence of nondeterministic actions, and is conformant if all its possible executions lead from an initial state S to a goal state G.

In our experiments, the shape of transition diagrams is as shown in Figure 2. A successor state is guessed for each state in the input diagram, so that the plan is conformant if G is reachable from S in each answer set of the program

```
trans(X,Y) v trans(X,Z) :- ptrans(X,Y,Z).
reach(X,Y) :- trans(X,Y).
reach(X,Y) :- reach(X,Z), trans(Z,Y).
```

The experiments have been performed on a 3GHz Intel® Xeon® processor system with 4GB RAM under the Debian 4.0 operating system with a GNU/Linux 2.6.23 kernel. The DLV prototype used has been compiled using GCC 4.3.3. For each instance, we have allowed a maximum running time of 600 seconds (10 minutes) and a maximum memory usage of 3GB.

As shown in Figure 2, DMS has an exponential speed-up over both DLV and SMS. In this case, the exponential computational gain of DMS over DLV and SMS is due to the dynamic optimization of the answer set search phase resulting from our magic sets definition. Indeed, DMS include nondeterministic relevance information that can be exploited also during the nondeterministic search phase of DLV, dynamically disabling parts of the ground program. In particular, after having made some choices, parts of the program may no longer be relevant to the query, but only because of these choices, and the magic atoms present in the ground program can render these parts satisfied, which means that they will no longer be considered in this part of the search.

5. Conclusion

The Magic Set method is one of the most well-known techniques for the optimization of positive recursive Datalog programs due to its efficiency and its generality. In our work, we have extended the technique to (disjunctive) ASP. The main novelty of the proposed Magic Set method is the dynamic optimization of the answer set search. Indeed, with DMS, ASP computations can exploit the information provided by magic set predicates also during the nondeterministic answer set search, allowing for potentially exponential performance gains with respect to unoptimized evaluations.

We have established the correctness of DMS for stratified ASP programs by proving that the transformed program is query-equivalent to the original program. A strong relationship between magic sets and unfounded sets has been highlighted: The atoms that are relevant w.r.t. an answer set are either true or form an unfounded set.

DMS has been implemented in the DLV system. Experiments on the implemented prototype system evidenced that our implementation can outperform the standard evaluation in general also by an exponential factor. This is mainly due to the optimization of the model generation phase, which is specific of our Magic Set technique.

Our research has been focused on ASP with stratified negation, because the concept of "relevance" can be extended quite easily for this class of programs. Conversely, if recursion over negation is allowed, an inconsistency may arise in some part of the program apparently not related to the query. A first step forward in extending DMS to programs with unstratified negation has been done in [Alv10], in which the technique presented in this paper has been proved to be correct for super-consistent ASP, a large class of programs including odd-cycle-free programs (that is, programs in which no cycle of dependencies involves an odd number of negative dependencies). Analysing the possibility to extend DMS to a larger class of unstratified ASP programs is a challenge we intend to address in the future.

Acknowledgement

The author wishes to thank Wolfgang Faber for his care in checking this work and for the fruitful discussions without which the results herein summarized could not be achieved.

References

[Alv09] Mario Alviano, Wolfgang Faber, Gianluigi Greco, and Nicola Leone. Magic Sets for Disjunctive Datalog Programs. Tech. Rep. 09/2009, Department of Mathematics, University of Calabria, Italy, 2009. `http://www.wfaber.com/research/papers/TRMAT092009.pdf`.

[Alv10] Mario Alviano and Wolfgang Faber. Dynamic Magic Sets for Super-Consistent Answer Set Programs. In *3rd Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP10)*. 2010. To appear.

[Bar03] Chitta Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2003.

[Bee91] Catriel Beeri and Raghu Ramakrishnan. On the power of magic. 10(1–4):255–259, 1991.

[Eit94] Thomas Eiter, Georg Gottlob, and Heikki Mannila. Adding Disjunction to Datalog. In *Proceedings of the Thirteenth ACM SIGACT SIGMOD-SIGART Symposium on Principles of Database Systems (PODS-94)*, pp. 267–278. ACM Press, 1994.

[Gol96] Robert P. Goldman and Mark S. Boddy. Expressive Planning and Explicit Knowledge. In *Proceedings AIPS-96*, pp. 110–117. AAAI Press, 1996.

[Gre03] Sergio Greco. Binding Propagation Techniques for the Optimization of Bound Disjunctive Queries. *IEEE Transactions on Knowledge and Data Engineering*, 15(2):368–385, 2003.

[Leo97] Nicola Leone, Pasquale Rullo, and Francesco Scarcello. Disjunctive Stable Models: Unfounded Sets, Fixpoint Semantics and Computation. 135(2):69–112, 1997.

[Leo04] Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. The DLV System for Knowledge Representation and Reasoning. 2004. To appear. Available via `http://www.arxiv.org/ps/cs.AI/0211004`.

Technical Communications of the International Conference on Logic Programming, 2010 (Edinburgh), pp. 236–240
http://www.floc-conference.org/ICLP-home.html

BISIMILARITY IN CONCURRENT CONSTRAINT PROGRAMMING

ANDRÉS A. ARISTIZÁBAL P.

CNRS, LIX École Polytechnique and INRIA Team COMÈTE
Route de Saclay 91128 Palaiseu Cedex, France.
E-mail address: andresaristi@lix.polytechnique.fr
URL: http://www.lix.polytechnique.fr/∼andresaristi/

Abstract. In this doctoral work we aim at developing a new approach to labelled semantics and equivalences for the Concurrent Constraint Programming (CCP) which will enable a broader capture of processes behavioural equivalence. Moreover, we work towards exploiting the strong connection between first order logic and CCP. Something which will allow us to represent logical formulae in terms of CCP processes and verify its logical equivalence by means of our notion of bisimilarity. Finally, following the lines of the Concurrecy Workbench we plan to implement a CCP Workbench based on our theoretical structure.

Motivations

Concurrency is concerned with the fundamental aspects of systems consisting of multiple computing agents, usually called *processes*, that interact among each other. *Bisimilarity* is a central behavioural equivalence in concurrency theory as it elegantly captures our intuitive notion of process equivalence; two processes are equivalent if they can match each other's moves. In fact, several concurrent formalisms such as CCS [Mil80] and the π-calculus [Mil99] are equipped with semantic, axiomatic, verification and, in general, reasoning techniques for bisimilarity.

Concurrent Constraint Programming (CCP) [Sar90] is a well-established *declarative* formalism for concurrency. Its basic intuitions arise mostly from *first-order logic*. In CCP processes can interact by *adding* (or *telling*) partial information in a medium, a so-called *store*. Partial information is represented by constraints (e.g., $x > 42$) on the shared variables of the system. The other way in which processes can interact is by *asking* partial information to the store. This provides the synchronization mechanism of the model; asking agents are suspended until there is enough information in the store to answer their query.

Despite the relevance of bisimilarity on the behavioural theory of processes, there have been few attempts to define a proper notion of bisimilarity equivalence for CCP. Apart from the rich reasoning techniques that are typically derived from this equivalence, the close ties between CCP and logic may provide with a novel characterization of logic equivalence in terms of bisimilarity.

1998 ACM Subject Classification: D.1.3, D.3.2, D.3.3, F.1.1, F.1.2, F.3.2, F.4.0.

Key words and phrases: Concurrent Constraint Programming, Concurrency, Behavioural Equivalence, Bisimilarity, Process Calculi, Operational Semantics, Labelled Semantics.

Technical Communications of the 26th International Conference on Logic Programming, Edinburgh, July, 2010
Editors: Manuel Hermenegildo, Torsten Schaub
LIPIcs - Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/LIPIcs.ICLP.2010.236

1. Goals

We aim to provide CCP with an appropriate notion of bisimilarity and its derived reasoning techniques. Furthermore, we plan to use the close connection between CCP and first order logic to give a characterization of logical equivalence in terms of bisimilarity. Finally, we plan to implement an automated tool for verifying bisimilarity equivalence of CCP processes along the lines of the Concurrency Workbench [Cle93] .

2. Current Work

Like in other process algebras, in CCP processes are represented as syntactic terms reflecting their structure. For example, $tell(c)$ represents the process that adds the constraint c to the store and $ask(c).P$ is a process that asks if c can be derived from the information in the store and if so, it executes the process P. The composite term $P \parallel Q$ represents the execution of the processes P and Q in parallel.

In [Sar91] the authors gave an operational semantics for CCP which we will refer to as reduction semantics. Intuitively, a reduction $\langle P, S\rangle \rightarrow \langle P', S'\rangle$ represents a one-step evolution of the process-store configuration $\langle P, S\rangle$ to $\langle P', S'\rangle$.

We use $\models$ to denote an entailment relation specifying interdependencies between constraints (e.g. $x > 10 \models x > 5$). We follow the well-established notion of barbed bisimilarity for the π-calculus [Mil99] and introduce the corresponding notion for CCP:

Definition 2.1. *(Barbed bisimilarity)* A barbed bisimulation is a symmetric relation $\mathcal{R}$ s.t., $\langle P, S_p\rangle \mathcal{R} \langle Q, S_q\rangle$ implies that:

(i) if $\langle P, S_p\rangle \rightarrow \langle P', S'_p\rangle$ then $\exists \langle Q', S'_q\rangle : \langle Q, S_q\rangle \rightarrow \langle Q', S'_q\rangle$ and $\langle P', S'_p\rangle \mathcal{R} \langle Q', S'_q\rangle$, and
(ii) $S_p \models S_q$.

We say that $\langle P, S_p\rangle$ and $\langle Q, S_q\rangle$ are barbed bisimilar, written $\langle P, S_p\rangle \dot{\sim}_B \langle Q, S_q\rangle$, if there is a barbed bisimulation $\mathcal{R}$ s.t. $\langle P, S_p\rangle \mathcal{R} \langle Q, S_q\rangle$.

Unfortunately, there are barbed bisimilar processes that when placed in a given context are not longer equivalent. Roughly, a context $C[\cdot]$ is a process term with a single hole $\cdot$ such that replacing $\cdot$ with a process gives a well-formed process. E.g., by taking $P = ask(x > 0).tell(y = 0)$ and $Q = ask(x > 10).tell(y = 1)$ and $C[\cdot] = tell(x > 5) \parallel \cdot$ we can verify that $P \dot{\sim}_B Q$ but $C[P] \not\dot{\sim}_B C[Q]$. Thus, we define:

Definition 2.2. *(Barbed Congruence)* We say that P and Q are barbed congruent, written $P \sim_B Q$, if for all contexts $C[\cdot]$, $\langle C[P], true\rangle \dot{\sim}_B \langle C[Q], true\rangle$.

The above definition is rather unsatisfactory because of the quantification over all possible contexts. To deal with this we define a labelled transition semantics. Intuitively, a transition $\langle P, S\rangle \xrightarrow{\alpha} \langle P', S'\rangle$ labelled with a constraint α, represents the minimal constraint α that needs to be added to the store S to evolve from $\langle P, S\rangle$ into $\langle P', S'\rangle$.

Our work builds on a similar CCP labelled semantics introduced in [Sar90]. The notion of bisimilarity in [Sar90] is, however, over-discriminating; e.g., it distinguishes $P = ask(x < 10).tell(y = 0) \parallel ask(x < 10).tell(y = 0)$ from $Q = ask(x < 5).tell(y = 0) \parallel ask(x < 10).tell(y = 0)$ which are clearly equivalent. Our notion of bisimilarity is defined thus:

Definition 2.3. *(Strong bisimilarity)* A strong bisimulation is a symmetric relation $\mathcal{R}$ s.t., $\langle P, S_p\rangle \mathcal{R} \langle Q, S_q\rangle$ implies that:

(i) if $\langle P, S_p\rangle \xrightarrow{\alpha} \langle P', S'_p\rangle$ then $\exists\langle Q', S'_q\rangle : \langle Q, S_q \wedge \alpha\rangle \rightarrow \langle Q', S'_q\rangle$ and $\langle P', S'_p\rangle \mathcal{R} \langle Q', S'_q\rangle$ and
(ii) $S_p \models S_q$.

We say that $\langle P, S_p\rangle$ and $\langle Q, S_q\rangle$ are strong bisimilar, written $\langle P, S_p\rangle \dot{\sim} \langle Q, S_q\rangle$, if there exists a strong bisimulation $\mathcal{R}$ such that $\langle P, S_p\rangle \mathcal{R} \langle Q, S_q\rangle$.

The main result we have obtained so far that the above notion fully captures barbed congruence but without quantification over all possible contexts: I.e., we state:

Theorem 2.4. *$\langle P, S_p\rangle \dot{\sim} \langle Q, S_q\rangle$ if and only if $\langle P, S_p\rangle \sim_B \langle Q, S_q\rangle$.*

Acknowledgement

This work is supervised by Catuscia Palamidessi and Frank Valencia in collaboration with Filippo Bonchi in the context of the INRIA project FORCES.

References

[Bon08] Filippo Bonchi. Abstract semantics by observable contexts. In *ICGT '08: Proceedings of the 4th international conference on Graph Transformations*, pp. 478–480. Springer-Verlag, Berlin, Heidelberg, 2008. doi:http://dx.doi.org/10.1007/978-3-540-87405-8_38.

[Cle93] Rance Cleaveland, Joachim Parrow, and Bernhard Steffen. The concurrency workbench: A semantics-based tool for the verification of concurrent systems. *ACM Trans. Program. Lang. Syst.*, 15(1):36–72, 1993.

[Mil80] Robin Milner. *A Calculus of Communicating Systems*, *Lecture Notes in Computer Science*, vol. 92. Springer, 1980.

[Mil99] Robin Milner. *Communicating and mobile systems: the π-calculus.* Cambridge University Press, New York, NY, USA, 1999.

[Sar90] Vijay A. Saraswat and Martin C. Rinard. Concurrent constraint programming. In *POPL*, pp. 232–245. 1990.

[Sar91] Vijay A. Saraswat, Martin C. Rinard, and Prakash Panangaden. Semantic foundations of concurrent constraint programming. In *POPL*, pp. 333–352. 1991.

Appendix A. Proof of Theroem 2.4

Firstly we show out new definitions and lemmas to proof our main theorem.

Another alternative definition for the barbed congruence is what we will name as a saturated barbed bisimilarity. This will be rather important since its definition is a bit more specific towards CCP than a barbed congruece, therefore is easier to relate with the strong bisimilarity we will define later on.

Definition A.1. *(Saturated barbed bisimilarity).* A saturated barbed bisimulation is a symmetric binary relation $\mathcal{R}$ on tuples of processes and stores satisfying the following: $\langle P, S_p\rangle \mathcal{R} \langle Q, S_q\rangle$ implies that:

(i) if $\langle P, S_p\rangle \to \langle P', S'_p\rangle$ then $\exists \langle Q', S'_q\rangle : \langle Q, S_q\rangle \to \langle Q', S'_q\rangle$ and $\langle P', S'_p\rangle \mathcal{R} \langle Q', S'_q\rangle$.
(ii) $S_p \models S_q$.
(iii) $\forall S' \langle P, S_p \wedge S'\rangle \mathcal{R} \langle Q, S_q \wedge S'\rangle$.

We say that $\langle P, S_p\rangle$ and $\langle Q, S_q\rangle$ are saturated barbed bisimilar, written $\langle P, S_p\rangle \dot{\sim}_{SB} \langle Q, S_q\rangle$, if there exists a saturated barbed bisimulation $\mathcal{R}$ such that $\langle P, S_p\rangle \mathcal{R} \langle Q, S_q\rangle$.

We did not report neither the reduction semantics nor the labelled semantics for lack of space. In order to prove our main theorem we assume that the two following lemmas hold.

Lemma A.2. *(Soundness of labelled semantics). If* $\langle P, S_p\rangle \xrightarrow{\alpha} \langle P', S'_p\rangle$*, then* $\langle P, S_p \wedge \alpha\rangle \to \langle P', S'_p\rangle$*.*

Lemma A.3. *(Completeness of labelled semantics). If* $\langle P, S_p \wedge x\rangle \to \langle P', S'_p\rangle$ *then* $\exists y, z$ *s.t.* $\langle P, S_p\rangle \xrightarrow{y} \langle P', S''_p\rangle$ *and* $(y \wedge z = x) \wedge (S''_p \wedge z = S'_p)$*.*

Corollary A.4. $\langle P, S_p\rangle \xrightarrow{true} \langle P', S'_p\rangle$ *if and only if* $\langle P, S_p\rangle \to \langle P', S'_p\rangle$

Theorem A.5. $\langle P, S_p\rangle \dot{\sim} \langle Q, S_q\rangle \Rightarrow \forall S' \langle P, S_p \wedge S'\rangle \dot{\sim} \langle Q, S_q \wedge S'\rangle$

Proof. We take a strong bisimulation $\mathcal{R} = \{(\langle P, S_p \wedge S'\rangle, \langle Q, S_q \wedge S'\rangle)$ s.t. $\langle P, S_p\rangle \dot{\sim} \langle Q, S_q\rangle\}$

(i) $\langle P, S_p \wedge S'\rangle \xrightarrow{\alpha} \langle P', S'_p\rangle$
By Lemma A.2 $\langle P, S_p \wedge S' \wedge \alpha\rangle \to \langle P', S'_p\rangle$.
By Lemma A.3 $\langle P, S_p\rangle \xrightarrow{y} \langle P', S''_p\rangle$ and $(y \wedge z = S' \wedge \alpha) \wedge (S''_p \wedge z = S'_p)$. Since $\langle P, S_p\rangle \dot{\sim} \langle Q, S_q\rangle$, then $\langle Q, S_q \wedge y\rangle \to \langle Q', S''_q\rangle$ s.t. $\langle P', S''_p\rangle \dot{\sim} \langle Q', S''_q\rangle$. Note that all reductions are preserved when adding constraints to the store, therefore from $\langle Q, S_q \wedge y\rangle \to \langle Q', S''_q\rangle$ we can derive that $\langle Q, S_q \wedge y \wedge z\rangle \to \langle Q', S''_q \wedge z\rangle$. This means that $\langle Q, S_q \wedge S' \wedge \alpha\rangle \to \langle Q', S''_q \wedge z\rangle$. Now we have that $\langle P', S'_p\rangle = \langle P', S''_p \wedge z\rangle \mathcal{R} \langle Q', S''_q \wedge z\rangle$, because $\langle P', S''_p\rangle \dot{\sim} \langle Q', S''_q\rangle$.
(ii) $S_p \wedge S' \models S_q \wedge S'$ since $S_p \models S_q$ by $\langle P, S_p\rangle \dot{\sim} \langle Q, S_q\rangle$ and $S' = S'$.

■

Now we state the lemmas which will enable us to prove our main theorem.

Lemma A.6. $\langle P, S_p\rangle \dot{\sim} \langle Q, S_q\rangle \Rightarrow \langle P, S_p\rangle \dot{\sim}_{SB} \langle Q, S_q\rangle$.

Proof. There exists a saturated barbed bisimulation $\mathcal{S}$ s.t. $\mathcal{S} = \{(\langle P, S_p\rangle, \langle Q, S_q\rangle)$ s.t. $\langle P, S_p\rangle \dot{\sim} \langle Q, S_q\rangle\}$ if the following conditions are fulfilled:

(i) if $\langle P, S_p\rangle \rightarrow \langle P', S'_p\rangle$ then $\exists\langle Q', S'_q\rangle : \langle Q, S_q\rangle \rightarrow \langle Q', S'_q\rangle$ and $\langle P', S'_p\rangle \; \mathcal{S} \; \langle Q', S'_q\rangle$. Suppose that $\langle P, S_p\rangle \rightarrow \langle P', S'_p\rangle$ then by Corollary A.4 $\langle P, S_p\rangle \xrightarrow{true} \langle P', S'_p\rangle$. Since $\langle P, S_p\rangle \dot{\sim} \langle Q, S_q\rangle$ then $\langle Q, S_q \wedge true\rangle \rightarrow \langle Q', S'_q\rangle$ then $\langle Q, S_q\rangle \rightarrow \langle Q', S'_q\rangle$ and $\langle P, S_p\rangle \dot{\sim} \langle Q, S_q\rangle$ then $\langle P, S_p\rangle \; \mathcal{S} \; \langle Q, S_q\rangle$

(ii) $S_p \models S_q$. Since $P \dot{\sim} Q$ (Condition (ii)).

(iii) $\forall S' \langle P, S_p \wedge S'\rangle \; \mathcal{R} \; \langle Q, S_q \wedge S'\rangle$. By Theorem A.5

■

Lemma A.7. $\langle P, S_p\rangle \dot{\sim}_{SB} \langle Q, S_q\rangle \Rightarrow \langle P, S_p\rangle \dot{\sim} \langle Q, S_q\rangle$.

Proof. There exists a strong bisimulation $\mathcal{R}$ s.t. $\mathcal{R} = \{(\langle P, S_p\rangle, \langle Q, S_q\rangle) \text{ s.t. } \langle P, S_p\rangle \dot{\sim}_{SB} \langle Q, S_q\rangle\}$ and if the following conditions are fulfilled:

(i) if $\langle P, S_p\rangle \xrightarrow{\alpha} \langle P', S''_p\rangle$ then $\exists\langle Q', S''_q\rangle : \langle Q, S_q \wedge \alpha\rangle \rightarrow \langle Q', S''_q\rangle$ and $\langle P', S''_p\rangle \; \mathcal{R} \; \langle Q', S''_q\rangle$. Suppose that $\langle P, S_p\rangle \xrightarrow{\alpha} \langle P', S'_p\rangle$ then by Lemma A.2 $\langle P, S_p \wedge \alpha\rangle \rightarrow \langle P', S'_p\rangle$. Since $\langle P, S_p\rangle \dot{\sim}_{SB} \langle Q, S_q\rangle$ then $\langle Q, S_q \wedge \alpha\rangle \rightarrow \langle Q', S'_q\rangle$ s.t. $\langle P', S'_p\rangle \dot{\sim}_{SB} \langle Q', S'_q\rangle$ then $\langle P', S'_p\rangle \; \mathcal{R} \; \langle Q', S'_q\rangle$

(ii) $S_p \models S_q$. Since $P \dot{\sim}_{SB} Q$ (Condition (ii)).

■

Theorem 2.4

Proof. By Lemma A.6 and Lemma A.7. ■

Technical Communications of the International Conference on Logic Programming, 2010 (Edinburgh), pp. 241–247
http://www.floc-conference.org/ICLP-home.html

PROGRAM ANALYSIS FOR CODE DUPLICATION IN LOGIC PROGRAMS

CÉLINE DANDOIS

University of Namur
Faculty of Computer Science
rue Grandgagnage 21
B-5000 Namur (Belgium)
E-mail address: cda@info.fundp.ac.be
URL: http://www.fundp.ac.be/info

Abstract. In this PhD project, we deal with the issue of code duplication in logic programs. In particular semantical duplication or redundancy is generally viewed as a possible seed of inconvenience in all phases of the program lifecycle, from development to maintenance. The core of this research is the elaboration of a theory of semantical duplication, and of an automated program analysis capable of detecting such duplication and which could steer, to some extent, automatic refactoring of program code.

1. Introduction and problem description

Program understanding or *program comprehension* refers to the process of acquiring knowledge about the structure and the functioning of a computer program [Rug96]. Such knowledge proves useful in support of a variety of software-engineering related activities including documentation, corrective and adaptive maintenance, migration and evolution of existing software systems [Sto06]. As a research area, program comprehension spans several subfields ranging from cognitive science and software psychology [Cou83, Shn93], over the development of abstract comprehension models [Bro83, Sol84, Sne98] and software visualization techniques [Bal96, JTS98] to using *program analysis* to (partially) automate program comprehension.

In this project, we will investigate program analysis techniques that allow to detect *duplication* within the source code of a given program. In its most general form, the notion of duplication refers to code fragments that are related in the sense that they subsume the same functionality. Note that this definition covers not only code fragments that are *textually* similar ("copy-paste programming") but also code fragments that are *functionally* similar but possibly implemented in a different way. Since duplication constitutes an existential and non-trivial program property, the discovery of duplication is obviously an undecidable problem that can nevertheless be approximated by using program analysis.

1998 ACM Subject Classification: D.1.6, D.2.7, F.3.2.

Key words and phrases: logic programming, program comprehension, static program analysis, code duplication, code clone, software engineering.

Technical Communications of the 26th International Conference on Logic Programming, Edinburgh, July, 2010
Editors: Manuel Hermenegildo, Torsten Schaub
LIPIcs - Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/LIPIcs.ICLP.2010.241

Detailed knowledge about duplication in a program is valuable for various reasons :

- Several studies show that software in which code duplication is present is more *error-prone* and difficult to understand and maintain than software without duplication [Kos06, Wal07a]. Hence the presence of certain forms of duplication is generally considered a bad smell [Fow99] and believed to have a negative impact on software evolution [Gei06]. Although there is some disagreement as to know whether duplication removal is always beneficial [Kap06], there seems to be a consensus that duplication should at the very least be detected [SD02, Joh94, Man06, JM96, Fow99].

- From a purely technical viewpoint, duplication reflects *redundancy* in the source code since it contains distinct code fragments that are semantically equivalent. Although removing such redundancy may offer a practical interest (e.g. *code compaction* that aims at decreasing the executable program size [Bes03]), it also raises an interesting theoretical question [Kos06] whether and to what extend program code can be *normalized* – i.e. being brought into a form such that it contains as few duplication as possible – in a sense similar to the normalization of relational databases [Kho02, Lee95].

- Identifying duplication in a program allows to steer advanced analyses and transformations on the program code such as refactoring [Fow99], cliche recognition [Rug96], aspect mining [JZ08, AK07], virus detection [Wal07b] and plagiarism detection [Lan04]. In addition, an analysis for detecting duplication could be integrated in the code development process in such a way that the creation of duplicates of a certain size is avoided from the beginning [Lag97].

2. Background and overview of the existing literature

No consensus exists in the literature about an exact definition of code duplication. One also finds the notion of *code clones*, although this notion sometimes describes textually similar fragments, sometimes refers to a more general case of duplication [Kos06, Wal07a]. The scope of the proposed definition often depends on the detection technique.

This lack of standardization about code duplication is clearly addressed in three important recent papers which constitute complete overviews of this domain: [Kos06], [Roy07] and [Roy09]. In addition to code duplication terminology, they treat the reasons for code duplication and its consequences, they give the general code duplication detection process, they describe, then evaluate and compare, the existing detection techniques and tools, they talk about visualization, removal, avoidance and management of duplication, they explain evolution and quality analyses based on duplication, they synthesize the applications and related research for code duplication detection, and finally, they expose the open problems in this research field.

Among the vast amount of research done about code duplication during the last decade, to the best of our knowledge, the problem has not received much attention in a logic programming setting. The majority of the results have indeed been reported upon in the context of *imperative* and *object-oriented languages*, as showed in the above references. In the *functional paradigm*, even if it has not really more success than the logic paradigm, some works are emerging. We can point [Li09] which presents the tool Wrangler, able of

detecting duplication, among other things, in Erlang programs. [Bro10] is another work, complementing the latter, which proposes a code duplication detection technique for Haskell, built into the framework of the Haskell Refactorer (HaRe). Some language-independent detection tools exist but, as experimented in [Roy09], such tools lose in precision what they gain with their versatility since they cannot be tuned for the target language.

Furthermore, despite the fact that useful techniques and tools have been developed, most of these code duplication detection techniques are based on the *text*, *syntax* and *structure* of a particular programming language. As such, they are capable of detecting duplicated syntactical programming constructs (real duplication in the terminology of [Roy07]) rather than duplicated *program logic* within the source code of a program [Kos06, Roy07]. Nevertheless, concentrating on the program logic or the computations performed by the program rather than its syntactical appearance is deemed essential [Kos06, Roy07] if duplication detection techniques are to become more powerful, more generally applicable and independent of particular programming language constructs.

3. Goal of the research

The cornerstone of this project is to study duplication in programs written in a logic programming language. Compared with other programming paradigms, logic programming languages have an arguably simple syntax and a small, clear and well-defined semantics. These characteristics make the development of a duplicated-code analysis both more manageable (less dependent on cumbersome syntax) and at the same time more powerful. Indeed, a logic program basically specifies a number of relations that hold between data objects rather than the algorithms to compute certain results as is the case in an imperative language. Consequently, rather than comparing syntactical algorithmic constructs, one can directly compare control- and data-flow relations, in particular when the program is augmented with mode information [Sma00].

4. Current status of the research

This project is still in its infancy and no result have been produced yet. For the moment, we focus on the dissection of the state of the art and we familiarize with some previous recent work realized inside our research group. Indeed, in [Van05], an initial study was made on the concept of code duplication in logic programs and the outline for a basic code duplication analysis of logic programs has been reported upon in [Van08]. The current project presents the natural continuation of both papers. The first ideas extracted from these works were presented at the GRASCOMP 2010 Contact Day, as a hotbed for future development.

5. Open issues and expected achievements

Although promising, devising a duplicated-code analysis for logic programs remains a daunting and non-trivial task. Finding duplication between the data-flow relations exhibited by a program is equivalent to finding isomorphic subgraphs in the data-flow graph of a program which is known to be NP-hard [Kri01, Kom01]. Consequently, sophisticated algorithms guided by heuristics will be necessary in order to make the analysis scalable to medium- and large-size programs. Topics of interest that will be studied in this project

include the following:

- Elaborate a *theory* of duplication in logic programs, including a classification of different kinds of duplication. On the one hand, this will allow to formally define what duplication is about and to state and prove certain results on analyses that try to detect duplicated code. On the other hand, these results could pave the way to develop a theory of normalization of logic programs by removing redundancies. A possible starting point for the latter topic is [Deg07] in which a first attempt was made to define a normal form in the restricted setting of Mercury [Som96] programs. However, [Deg07] does not deal with a number of important issues such as the normalization of data terms and predicate arguments.

- Based on the theory developed above, we will aim at developing an *analysis* that is able to detect duplication into a logic program up to a certain degree. Issues that need to be taken into account include *precision* (maximize the number of real duplicates while possibly minimizing the number of false positives), *granularity* (what is considered a useful duplicate), and *scalability* of the analysis. With respect to granularity, it seems that the notion of a *useful* duplicate may depend on the context of the analysis or the transformation aimed for. Hence, it seems desirable to design an analysis that can be parameterized with respect to the characteristics of the duplication it should search for.

- We will study the relation with advanced programming analysis and transformation techniques. A first topic of interest is automatically detecting opportunities for *refactoring* source code. Preliminary work on refactoring of logic programs [Ser08, Van05] has showed that a number of interesting refactorings can effectively be based on knowledge about duplication in a program. Nevertheless, the exact coupling between duplication in a program and the possibilities for automatic refactoring remains an open problem.

  Another topic of interest is the automatic detection of *cross-cutting concerns*, sometimes called *aspect-mining* [Kic96, AK07]. Basically, a cross-cutting concern is a functionality of the program that is implemented by a set of semantically similar code fragments that are scattered through the source code of a program (a typical example being the "logging" functionality within an application). Recent research has showed that duplication detection techniques can be beneficial for aspect-mining but stronger techniques capable of finding *semantically* related program code are necessary [Bru05, AK07].

Finally, since a logic program can be seen as a set of relations capturing data-flow information, we expect our research on automatically finding duplication within logic programs to be beneficial for analyses trying to find semantically related code in other programming languages and paradigms.

Acknowledgement

This PhD research is done under the supervision of Professor Wim Vanhoof.

References

[AK07] Kim Mens Andy Kellens and Paolo Tonella. A survey of automated code-level aspect mining techniques. *Transactions on Aspect-Oriented Software Development*, 4(4640):143–162, 2007.

[Bal96] Thomas Ball and Stephen G. Eick. Software visualization in the large. *IEEE Computer*, 29(4):33–43, 1996.

[Bes03] Árpád Beszédes, Rudolf Ferenc, Tibor Gyimóthy, André Dolenc, and Konsta Karsisto. Survey of code-size reduction methods. *ACM Computing Surveys*, 35(3):223–267, 2003. doi:http://doi.acm.org/10.1145/937503.937504.

[Bro83] Ruven Brooks. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, 18(6):543–554, 1983.

[Bro10] Christopher Brown and Simon Thompson. Clone detection and elimination for haskell. In *PEPM '10: Proceedings of the 2010 ACM SIGPLAN workshop on Partial evaluation and program manipulation*, pp. 111–120. ACM Press, 2010.

[Bru05] Magiel Bruntink, Arie van Deursen, Remco van Engelen, and Tom Tourwé. On the use of clone detection for identifying crosscutting concern code. *IEEE Trans. Software Eng*, 31(10):804–818, 2005. doi:http://doi.ieeecomputersociety.org/10.1109/TSE.2005.114.

[Cou83] Neal S. Coulter. Software science and cognitive psychology. *IEEE Transactions on Software Engineering*, 9(2):166–171, 1983.

[Deg07] François Degrave and Wim Vanhoof. Towards a normal form for mercury programs. In Andy King (ed.), *LOPSTR*, *Lecture Notes in Computer Science*, vol. 4915, pp. 43–58. Springer, 2007. doi:http://dx.doi.org/10.1007/978-3-540-78769-3_4.

[Fow99] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: improving the design of existing code*. Object Technology Series. Addison-Wesley, 1999.

[Gei06] Reto Geiger, Beat Fluri, Harald Gall, and Martin Pinzger. Relation of code clones and change couplings. In Luciano Baresi and Reiko Heckel (eds.), *Fundamental Approaches to Software Engineering, 9th International Conference, FASE 2006 March 27-28, 2006, Proceedings*, *Lecture Notes in Computer Science*, vol. 3922, pp. 411–425. Springer, 2006. doi:http://dx.doi.org/10.1007/11693017_31.

[JM96] Claude Leblanc Jean Mayrand and Ettore M. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *Proceedings of the 12th International Conference on Software Maintenance (ICSM '96)*, pp. 244–253. 1996.

[Joh94] John Howard Johnson. Substring matching for clone detection and change tracking. In *Proceedings of the International Conference on Software Maintenance (ICSM '94)*, pp. 120–126. 1994. doi:10.1109/ICSM.1994.336783.

[JTS98] Marc H. Brown John T. Stasko, John B. Domingue and Blaine A. Price. *Software Visualization: Programming As a Multimedia Experience*. Cambridge, Mass: MIT Press, 1998.

[JZ08] Yuehua Lin Jing Zhang, Jeff Gray and Robert Tairas. Aspect mining from a modelling perspective. *Int. J. of Computer Applications in Technology*, 31:74–82, 2008.
URL `http://www.inderscience.com/link.php?id=17720`

[Kap06] Cory Kapser and Michael W. Godfrey. Cloning considered harmful. In *Proceedings of the 13th Working Conference on Reverse Engineering (WCRE2006)*, pp. 19–28. IEEE Computer Society, 2006. doi:http://doi.ieeecomputersociety.org/10.1109/WCRE.2006.1.

[Kho02] V. V. Khodorovskii. On normalization of relations in relational databases. *Program. Comput. Softw.*, 28(1):41–52, 2002. doi:http://dx.doi.org/10.1023/A:1013759617481.

[Kic96] G. Kiczales. Aspect-oriented programming. *ACM Computing Surveys*, 28(4es), 1996.

[Kom01] Raghavan Komondoor and Susan Horwitz. Using slicing to identify duplication in source code. In *Proceedings of the 8th International Symposium on Static Analysis (SAS)*. Springer-Verlag, Paris, France, 2001.
URL `http://www.cs.wisc.edu/~raghavan/sas01.pdf`

[Kos06] Rainer Koschke. Survey of research on software clones. In Rainer Koschke, Ettore Merlo, and Andrew Walenstein (eds.), *Duplication, Redundancy, and Similarity in Software*, *Dagstuhl Seminar Proceedings*, vol. 06301. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2006.
URL `http://drops.dagstuhl.de/opus/volltexte/2007/962`

[Kri01] Jens Krinke. Identifying similar code with program dependence graphs. In *Proceedings Eigth Working Conference on Reverse Engineering (WCRE'01)*, pp. 301–309. IEEE Computer Society, 2001. doi:10.1109/WCRE.2001.957835.

[Lag97] Bruno Laguë, Daniel Proulx, Jean Mayrand, Ettore Merlo, and John P. Hudepohl. Assessing the benefits of incorporating function clone detection in a development process. In *ICSM*, pp. 314–321. 1997.

[Lan04] Thomas Lancaster and Culwin Finta. A comparison of source code plagiarism detection engines. *Computer Science Education*, 14(2):101–112, 2004.

[Lee95] Heeseok Lee. Justifying database normalization: a cost/benefit model. *Inf. Process. Manage.*, 31(1):59–67, 1995. doi:http://dx.doi.org/10.1016/0306-4573(94)E0011-P.

[Li09] Huiqing Li and Simon Thompson. Clone detection and removal for erlang/otp within a refactoring environment. In *PEPM '09: Proceedings of the 2009 ACM SIGPLAN workshop on Partial evaluation and program manipulation*, pp. 169–178. ACM Press, 2009.

[Man06] Zoltán Ádám Mann. Three public enemies: Cut, copy, and paste. *IEEE Computer*, 39(7):31–35, 2006. doi:http://doi.ieeecomputersociety.org/10.1109/MC.2006.246.

[Roy07] C. K. Roy and J. R. Cordy. A survey on software clone detection research. Tech. rep., 2007. TR 2007-541 School of Computing Queen's University at Kingston Ontario, Canada.

[Roy09] Chanchal K. Roy, James R. Cordy, and Rainer Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, 74(7):470–495, 2009.

[Rug96] Spencer Rugaber. Program understanding. *Encyclopedia of Computer Science and Technology*, 1996.

[SD02] Stéphane Ducasse Serge Demeyer and Oscar Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2002.
URL `http://www.iam.unibe.ch/~scg/OORP`

[Ser08] Alexander Serebrenik, Tom Schrijvers, and Bart Demoen. Improving prolog programs: Refactoring for prolog. *TPLP*, 8(2):201–215, 2008. doi:http://dx.doi.org/10.1017/S1471068407003134.

[Shn93] Ben Shneiderman. Software psychology: Sparks of innovation in human-computer interaction, 1993.

[Sma00] J.-G. Smaus, P. Hill, and A. King. Mode analysis domains for typed logic programs. In A. Bossi (ed.), *LOPSTR*. Springer-Verlag, 2000.
URL `http://www.cs.kent.ac.uk/pubs/2000/1011`

[Sne98] Gregor Snelting. Concept Analysis — A New Framework for Program Understanding. In *SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, pp. 1–10. ACM Press, Montreal, Canada, 1998.

[Sol84] Elliot Soloway and Kate Ehrlich. Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering*, 10(5):595–609, 1984. Special Issue on Software Reusability.

[Som96] Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury: an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29(1–3), 1996.

[Sto06] Margaret-Anne D. Storey. Theories, tools and research methods in program comprehension: past, present and future. *Software Quality Journal*, 14(3):187–208, 2006. doi:http://dx.doi.org/10.1007/s11219-006-9216-4.

[Van05] W. Vanhoof. Searching semantically equivalent code fragments in logic programs. In S. Etalle and Springer-Verlag (eds.), *Proceedings of LOPSTR 2004*, *LLNCS*, vol. 3573. 2005.

[Van08] W. Vanhoof and F. Degrave. An algorithm for sophisticated code matching in logic programs. In M. Garcia de la Banda, E. Pontelli, and Springer-Verlag (eds.), *Proceedings of ICLP 2008*, *LLNCS*, vol. 5366. 2008.

[Wal07a] Andrew Walenstein, Mohammad El-Ramly, James R. Cordy, William S. Evans, Kiarash Mahdavi, Markus Pizka, Ganesan Ramalingam, and Jürgen Wolff von Gudenberg. Similarity in programs. In Rainer Koschke, Ettore Merlo, and Andrew Walenstein (eds.), *Duplication, Redundancy, and Similarity in Software*, no. 06301 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, Dagstuhl, Germany, 2007.
URL `http://drops.dagstuhl.de/opus/volltexte/2007/968`

[Wal07b] Andrew Walenstein and Arun Lakhotia. The software similarity problem in malware analysis. In Rainer Koschke, Ettore Merlo, and Andrew Walenstein (eds.), *Duplication, Redundancy, and Similarity in Software*, no. 06301 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, Dagstuhl, Germany, 2007.
URL `http://drops.dagstuhl.de/opus/volltexte/2007/964`

Technical Communications of the International Conference on Logic Programming, 2010 (Edinburgh), pp. 248–254
http://www.floc-conference.org/ICLP-home.html

PROGRAM ANALYSIS TO SUPPORT CONCURRENT PROGRAMMING IN DECLARATIVE LANGUAGES

ROMAIN DEMEYER

University of Namur - Faculty of Computer Science
Rue Grandgagnage 21, 5000 Namur (Belgium)
E-mail address: rde@info.fundp.ac.be

Abstract. In recent years, manufacturers of processors are focusing on parallel architectures in order to increase performance. This shift in hardware evolution is provoking a fundamental turn towards concurrency in software development. Unfortunately, developing concurrent programs which are correct and efficient is hard, as the underlying programming model is much more complex than it is for simple sequential programs. The goal of this research is to study and to develop program analysis to support and improve concurrent software development in declarative languages. The characteristics of these languages offer opportunities, as they are good candidates for building concurrent applications while their simple and uniform data representation, together with a small and formally defined semantics makes them well-adapted to automatic program analysis techniques. In our work, we focus primarily on developing static analysis techniques for detecting race conditions at the application level in Mercury and Prolog programs. A further step is to derive (semi-) automatically the location and the granularity of the critical sections using a data-centric approach.

1. Introduction and Problem Description

Since the mid-70s, the power of the microprocessor, which is the basic component of the computer responsible for instruction execution and data processing, has increased constantly. For decades, we have witnessed a dramatic and continuous growth of clock speed, which is one of the main factors determining the performance of processors [Olu05]. Recently, however, this growth appears to have stabilized. Indeed, the manufacturers encounter several physical problems, notably the impossibility to dissipate the heat and a too high power consumption [Sut05]. Instead of driving clock speeds and straight-line instruction throughput ever higher, processor manufacturers are, for these reasons, turning to *hyperthreading* and *multicore* architectures, i.e. processors with multiple identical units of calculation [Her06, Sut05, Lu08].

This hardware revolution is going to change fundamentally the way people write software. Indeed, to benefit from the power of the new processors, software must be able to exploit their innate parallelism, which is not the case for traditional software which is, in

1998 ACM Subject Classification: D.1.3 [**Programming Techniques**]: Concurrent Programming; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages–*Program Analysis*; D.1.6 [**Programming Techniques**]: Logic Programming.

Key words and phrases: Program Analysis – Concurrent Programming – Logic Languages – Abstract Interpretation .

Technical Communications of the 26th International Conference on Logic Programming, Edinburgh, July, 2010
Editors: Manuel Hermenegildo, Torsten Schaub
LIPIcs - Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/LIPIcs.ICLP.2010.248

most cases, written following the sequential model of programming. In this new context, software must be designed following the *concurrency model* [Her06, Her08, Mat04, Mag99, BA90, Hug08] of programming: the application is made of a set of interacting *processes* that are executed, at least conceptually, in parallel and often in a shared memory space [Don08].

Unfortunately, developing concurrent programs that are correct and efficient is really hard, as potential bugs related to concurrent execution are difficult to detect and to isolate. Indeed, the underlying programming model is much more complex than it is for simple sequential programs [Lu08, BA90, Gro07, AZ08]. What makes concurrent programming hard in any language is that one has to deal with the interactions between processes and the nondeterministic interleaving of executions, especially if these processes handle shared memory. That is how undesirable phenomena, which are called *race conditions*, occur: two or more threads attempt to change a shared piece of data at (almost) the same time and the final value of the data depends simply in what order threads access it [Hug08]. These race conditions occur because of a bad *synchronization* between threads [Lu08, BA90].

To avoid errors, so-called *critical sections* have to be identified in the source code and *mutual exclusion* between execution of these sections must be guaranteed, using for example locks [BA90] or software transactional memories (STM) [Sha97, Lar06, Jon07, Har05, Har03, Mul06, Mik07, Kel05]. Whatever the way in which this mutual exclusion is ensured, a crucial point is to determine the location and the size of the critical sections. On the one hand, if they are too small or badly located, it can introduce race conditions at the application level. One the other hand, it is essential to keep the critical section as small as possible in order not to loose more performance than necessary and to avoid inter-blocking [Gro07]. Moreover, ensuring mutual exclusion is far from trivial. Locks are not composable [Har05] – i.e. correctly protected pieces of codes can't be simply reused to form larger correctly protected operations – and using them can lead to deadlocks, livelocks, priority inversion [BA90, Ho05, Eng03, Nai07, Bec08] or security breaks [Tip06, Che04, How09]. While STM avoid these issues, their implementation is complex and irreversible operations, like i/o operations, are traditionnaly prohibited inside the atomic blocks [Gro07, Men08, Har09, Dal09, Luc08, Boe09].

Obviously, programmers desperately need a higher-level programming model for concurrency than what languages offer today [Sut05]. Logic programming languages are known to be particularly well-adapted to parallelism [Tic91] but program analysis is needed as it can be used to detect race conditions and other bugs related to concurrency. It can also be used to (help to) determine the appropriate location and granularity of the critical sections.

2. Background and Overview of the Existing Literature

The interest of program analysis to support concurrent programming is increasingly prevalent with the actual multicore crisis. In the context of *explicit parallelism* – where the programmer decides where and how to integrate the parallelism in his program, classical analysis tools target to detect race conditions [McC06, Pra06]. Some of these tools are on the base of transformation programs methods, the goal of which is to combine conceptual advantages of STM with those of locks [McC06, Pra06, Hic06]. But these tools are only able to detect low-level race conditions – i.e. simple reading and writing of a memory space. These tools are not able to detect the errors that occur at the level of the logic of the

application. For example, bad utilisation of critical sections can lead to violate an invariant related to a data structure of the application.

Recently, the problem has caught the attention in the context of declarative languages and very recent works are targeting race conditions detection in these languages [Chr10, Cla09]. In the context of object-oriented languages, [Bec08] proposes to use typestate specifications [Del04] and linear logic [Gir87] to express invariant related to an object and the input and output conditions of its methods. The goal is to statically detect race conditions involving these objects at the application level on the basis of the specified behaviour. A related work [Har06] presents a dynamic analysis for STM in Haskell which ensures that an invariant will not be violated during an execution.

A still more ambitious but complex objective is to (semi-)automatically determine the location and the size of the critical sections. Recent work [Vaz06, McC06] has proposed a *data-centric approach* to synchronize threads which consists of a two-step procedure: first, the programmer associates synchronization constraints to the data structures that must be accessed atomically; second, these information are used to complete, through program transformation, the source code with the adequate locking mechanism where the synchronisation is necessary. One main advantage of this approach is the control of the granularity of the concurrent system.

Despite numerous works about concurrency, we are far from being able to detect all kind of errors related to concurrent programming [Vaz06, Lu08]. In most cases, analysis is able to detect synchronization mistakes related to only one variable. Moreover, it generally does not deliver pertinent informations about the way one can correct it [Lu08]. *Test case generation* to expose concurrency errors must also be considered, but has to cope with a high complexity issue: the number of possible interleaving to consider is exponential [Tay92, Yan97]. This imposes to explore subtle methods to produce pertinent tests in practice [Lu08, Qad04].

Although the primary goal of these program analysis is to assist the programmer in writing concurrent programs, they can also be useful to guide so-called *implicit parallelism* transformation that aim at the automatic parallelisation of programs [Cos08]. This field is particularly active in the context of logic languages [Bon08, Gup01, Cha08, Mou08, Cas08, Cas07]. In this kind of code analysis, the detection and the exploitation of the parallelism is made completely automatically, often at compilation time, without clear contribution of the programmer.

3. Goal of the Research

The goal of this research is to study and to develop program analysis to support and improve concurrent software development in declarative languages. In contrast with more classic imperative ones, declarative languages allow to describe the logic behind a solution instead of having to describe the step-by-step process of how this solution must be computed by the computer. Declarative languages are mostly pure – i.e. they do not allow programs to provoke side-effects [Hen96] – which is known to increase the productivity of the developers and the reliability of the programs, and makes these languages good candidates to use for building concurrent applications. Moreover, their simple and uniform data representation, together with a small and formally defined semantics, makes these languages well-adapted to automatic program analysis techniques. In our work, we focus primarily on Mercury.

Mercury [Som96] is a modern logic programming language, which is designed to develop modular and reliable large-size software applications.

4. Current Status of the Research

This research is still in its beginning. For the moment, we focus on the dissection of the state of the art and we familiarize with the very large field of concurrency by reading papers, books and by trying to make constructive contacts with other researcher working on related projects. We are trying to figure out in what directions it is the most valuable to guide the research. Since the latter is highly related to logic programming, the ICLP Doctoral Consortium would be an excellent opportunity, not only to acquire a profound complement to the state of the art, but also to get in touch with both experts and other PhD students working on related topics and to exchange point of views and opinions about my future work. It goes without doubt that the consortium would be a valuable experience from which we will be able to take full advantage in pursuing our project.

5. Open Issues and Expected Achievements

We plan to develop static analysis techniques for detecting race conditions at the application level in Mercury in first phase, Prolog in a following phase, languages that are particularly well-suited for concurrency [Bon08, Tan07, Wan08]. Such an analysis can be done based on the location of the critical sections and a abstract specification of the behaviour of the shared data. We study how a very expressive formalism, such as linear temporal logic [Pnu77], can be used for this behavioural specification.

Also in the context of declarative languages, a further step is to derive (semi-) automatically the location and the granularity of the critical section, possibly by extending a data-centric approach as suggested by recent work [Vaz06, McC06]. The particular type representation in declarative languages, like Mercury, is expected to fit well with such an analysis and to open new perspectives compared to traditional imperative languages.

Further elements of interest are the automatic generation of test cases targeted to detecting concurrency bugs and how our techniques can be used to advance research on *implicit* parallelism [Cos08] in declarative languages.

Acknowledgements

This PhD research is under the supervision of Professor Wim Vanhoof.

References

[AZ08] Abdallah Deeb I. Al Zain, Kevin Hammond, Jost Berthold, Phil Trinder, Greg Michaelson, and Mustafa Aswad. Low-pain, high-gain multicore programming in Haskell: coordinating irregular symbolic computations on multicore architectures. In *DAMP '09: Proceedings of the 4th workshop on Declarative aspects of multicore programming*, pp. 25–36. ACM, New York, NY, USA, 2008. doi:http://doi.acm.org/10.1145/1481839.1481843.

[BA90] M. Ben-Ari. *Principles of concurrent and distributed programming*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990.

[Bec08] Nels E. Beckman, Kevin Bierhoff, and Jonathan Aldrich. Verifying correct usage of atomic blocks and typestate. In Gail E. Harris (ed.), *OOPSLA*, pp. 227–244. ACM, 2008.
URL `http://doi.acm.org/10.1145/1449764.1449783`

[Boe09] Hans-J. Boehm. Transactional memory should be an implementation technique, not a programming interface. Tech. Rep. HPL-2009-45, Hewlett Packard Laboratories, 2009.
URL `http://www.hpl.hp.com/techreports/2009/HPL-2009-45.html;http://www.hpl.hp.com/techreports/2009/HPL-2009-45.pdf`

[Bon08] Paul Bone. Calculating likely parallelism within dependant conjunctions for logic programs. October, 2008.

[Cas07] Amadeo Casas, Manuel Carro, and Manuel V. Hermenegildo. Annotation algorithms for unrestricted independent and-parallelism in logic programs. In Andy King (ed.), *LOPSTR*, *Lecture Notes in Computer Science*, vol. 4915, pp. 138–153. Springer, 2007.
URL `http://dx.doi.org/10.1007/978-3-540-78769-3_10`

[Cas08] Amadeo Casas, Manuel Carro, and Manuel V. Hermenegildo. A high-level implementation of non-deterministic, unrestricted, independent and-parallelism. In Maria Garcia de la Banda and Enrico Pontelli (eds.), *ICLP*, *Lecture Notes in Computer Science*, vol. 5366, pp. 651–666. Springer, 2008.
URL `http://dx.doi.org/10.1007/978-3-540-89982-2`

[Cha08] Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, and Gabriele Keller. Partial vectorisation of Haskell programs. In M. Hermenegildo (ed.), *Workshop on Declarative Aspects of Multicore Programming*. 2008.

[Che04] Brian Chess and Gary McGraw. Static analysis for security. *IEEE Security & Privacy*, 2(6):76–79, 2004.
URL `http://doi.ieeecomputersociety.org/10.1109/MSP.2004.111`

[Chr10] Maria Christakis and Konstantinos Sagonas. Static detection of race conditions in erlang. In *Practical Aspects of Declarative Languages : PADL 2010*, no. 5937 in Lecture Notes in Computer Science, pp. 119–133. Springer-Verlag, 2010.

[Cla09] Koen Claessen, Michał Pałka, Nicholas Smallbone, John Hughes, Hans Svensson, Thomas Arts, and Ulf Wiger. Finding race conditions in erlang with quickcheck and pulse. In *ICFP '09: Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*. ACM, New York, NY, USA, 2009.

[Cos08] Vitor Santos Costa. On supporting parallelism in a logic programming system. In Manuel Hermenegildo (ed.), *Workshop on Declarative Aspects of Multicore Programming*. 2008.

[Dal09] Luke Dalessandro and Mickael L. Scott. Strong isolation is a weak idea. 2009. doi:http://transact09.cs.washington.edu/33_paper.pdf.

[Del04] Robert Deline and Manuel Fahndrich. Typestates for objects. In *In Proc. 18th ECOOP*, pp. 465–490. Springer, 2004.

[Don08] M. R. C. Van Dongen. Thread programming, 2008.

[Eng03] Dawson Engler and Ken Ashcraft. Racerx: Effective, static detection of race conditions and deadlocks, 2003.

[Gir87] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[Gro07] Dan Grossman. The transactional memory / garbage collection analogy. *ACM SIGPLAN Notices*, 42, 2007.

[Gup01] Gopal Gupta, Enrico Pontelli, Khayri A. M. Ali, Mats Carlsson, and Manuel V. Hermenegildo. Parallel execution of prolog programs: a survey. *ACM Transactions on Programming Languages and Systems*, 23(4):472–602, 2001.

[Har03] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pp. 388 – 402. 2003.

[Har05] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pp. 48–60. ACM, New York, NY, USA, 2005. doi:http://doi.acm.org/10.1145/1065944.1065952.

[Har06] Tim Harris and Simon Peyton-Jones. Transactional memory with data variants. In *First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT'06)*. Ottawa, 2006.

[Har09] Tim Harris. Language constructs for transactional memory. *SIGPLAN Notices*, 44(1):1–1, 2009. doi:http://doi.acm.org/10.1145/1594834.1480883.

[Hen96] Fergus Henderson, Thomas Conway, Zoltan Somogyi, David Jeffery, Peter Schachte, Simon Taylor, and Chris Speirs. The Mercury language reference manual. Tech. rep., 1996.

[Her06] Maurice Herlihy. The art of multiprocessor programming. In *PODC '06: Proceedings of the twenty-fifth annual ACM symposium on Principles of distributed computing*, pp. 1–2. ACM, New York, NY, USA, 2006. doi:http://doi.acm.org/10.1145/1146381.1146382.

[Her08] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008. URL `http://www.worldcat.org/isbn/0123705916`

[Hic06] Michael Hicks, Jeffrey S. Foster, and Polyvios Prattikakis. Lock inference for atomic sections. In *Proceedings of the First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*. 2006.

[Ho05] Alex Ho, Steven Smith, and Steven Hand. On deadlock, livelock, and forward progress. Tech. Rep. UCAM-CL-TR-633, University of Cambridge Computing Laboratory, 2005.

[How09] Michael Howard. Basic training: Improving software security by eliminating the CWE top 25 vulnerabilities. *IEEE Security & Privacy*, 7(3):68–71, 2009. doi:http://dx.doi.org/10.1109/MSP.2009.69.

[Hug08] Cameron Hughes and Tracey Hughes. *Professional Multicore Programming: Design and Implementation for C++ Developers*. Wrox Press Ltd., Birmingham, UK, UK, 2008.

[Jon07] Simon Peyton Jones. Beautiful concurrency. In Andy Oram and Greg Wilson (eds.), *Beautiful Code*, pp. 385–406. O'Reilly & Associates, Inc., Sebastopol, CA 95472, 2007. Ch. 24.

[Kel05] Richard Kelsey, Jonathan Rees, and Mike Sperber. The incomplete scheme 48 reference manuel release 1.3, April 2005.

[Lar06] James Larus and Ravi Rajwar. *Transactional Memory*. Morgan & Claypool, 2006.

[Lu08] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In Susan J. Eggers and James R. Larus (eds.), *ASPLOS*, pp. 329–339. ACM, 2008.
URL `http://doi.acm.org/10.1145/1346281.1346323`

[Luc08] Victor Luchangco. Against lock-based semantics for transactional memory. In Friedhelm Meyer auf der Heide and Nir Shavit (eds.), *SPAA*, pp. 98–100. ACM, 2008.
URL `http://dblp.uni-trier.de/db/conf/spaa/spaa2008.html#Luchangco08`

[Mag99] Jeff Magee and Jeff Kramer. *Concurrency: state models & Java programs*. John Wiley & Sons, Inc., New York, NY, USA, 1999.

[Mat04] Timothy Mattson, Beverly Sanders, and Berna Massingill. *Patterns for parallel programming*. Addison-Wesley Professional, 2004.

[McC06] Bill McCloskey, Feng Zhou, David Gay, and Eric Brewer. Autolocker: synchronization inference for atomic sections. *ACM SIGPLAN Notices*, 41(1):346–358, 2006.

[Men08] Vijay Menon, Steven Balensiefer, Tatiana Shpeisman, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Bratin Saha, and Adam Welc. Single global lock semantics in a weakly atomic stm. *SIGPLAN Notices*, 43(5):15–26, 2008. doi:http://doi.acm.org/10.1145/1402227.1402235.

[Mik07] Leon Mika. Software transactional memory in Mercury, October 2007.

[Mou08] Paulo Moura, Ricardo Rocha, and Sara C. Madeira. Thread-based competitive or-parallelism. In Maria Garcia de la Banda and Enrico Pontelli (eds.), *ICLP*, *Lecture Notes in Computer Science*, vol. 5366, pp. 713–717. Springer, 2008.
URL `http://dx.doi.org/10.1007/978-3-540-89982-2`

[Mul06] Ulrich Muller. Introducing the atomic keyword into c/c++ using assembler code instrumentation and software transactional memory, 2006.

[Nai07] Lee Naish. Resource-oriented deadlock analysis. In Verónica Dahl and Ilkka Niemelä (eds.), *ICLP*, *Lecture Notes in Computer Science*, vol. 4670, pp. 302–316. Springer, 2007.
URL `http://dx.doi.org/10.1007/978-3-540-74610-2_21`

[Olu05] Kunle Olukotun and Lance Hammond. The future of microprocessors. *Queue*, 3(7):26–29, 2005. doi:http://doi.acm.org/10.1145/1095408.1095418.

[Pnu77] A. Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science*. IEEE, 1977.

[Pra06] Polyvios Pratikakis, Jeffrey S. Foster, and Michael Hicks. LOCKSMITH: context-sensitive correlation analysis for race detection. *ACM SIGPLAN Notices*, 41(6):320–331, 2006.

[Qad04] Shaz Qadeer and Dinghao Wu. KISS: keep it simple and sequential. *ACM SIGPLAN Notices*, 39(6):14–24, 2004.

[Sha97] Nir Shavit and Dan Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, 1997.

[Som96] Zoltan Somogyi, Fergus Henderson, and Thomas Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language, 1996.

[Sut05] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3), 2005.

[Tan07] Jrme Tannier. Parallel Mercury. Tech. rep., Facults Universitaires Notre-Dame de la Paix, 2007. Mmoire fin d'tudes.

[Tay92] R. N. Taylor, D. L. Levine, and C. D. Kelly. Structural testing of concurrent programs. *IEEE Trans. on Softw. Eng.*, 18(3):206, 1992.

[Tic91] Evan Tick. *Parallel logic programming*. MIT Press, Cambridge, MA, USA, 1991.

[Tip06] Harold F. Tipton and Micki Krause (eds.). *Information security management handbook*. Auerbach Publications, Boca Raton, FL, USA, 5th edn., 2006.
URL `http://www.loc.gov/catdir/enhancements/fy0659/2003061151-d.html`

[Vaz06] Mandana Vaziri, Frank Tip, and Julian Dolby. Associating synchronization constraints with data in an object-oriented language. *ACM SIGPLAN Notices*, 41(1):334–345, 2006.

[Wan08] Peter Wang. Parallel Mercury. October, 2008.

[Yan97] Cheer-Sun Yang and Lori L. Pollock. The challenges in automated testing of multithreaded programs. In *In Proceedings of the 14th International Conference on Testing Computer Software*, pp. 157–166. 1997.

Technical Communications of the International Conference on Logic Programming, 2010 (Edinburgh), pp. 255–264
http://www.floc-conference.org/ICLP-home.html

CONSTRAINT ANSWER SET PROGRAMMING SYSTEMS

CHRISTIAN DRESCHER [1]

[1] Institute of Information Systems,
Vienna University of Technology, Favoritenstraße 9-11, A-1040 Vienna, Austria
E-mail address: christian.drescher@student.tuwien.ac.at

Abstract. We present an integration of answer set programming and constraint processing as an interesting approach to constraint logic programming. Although our research is in a very early stage, we motivate constraint answer set programming and report on related work, our research objectives, preliminary results we achieved, and future work.

1. Introduction

Constraint satisfaction problems (CSP) are combinatorial problems defined as a set of variables whose value must satisfy a number of limitations, and are subject to intense research. Problems that has been successfully modelled as a CSP stem from a variety of areas, for example, artificial intelligence, operations research, electrical engineering and telecommunications.

There are several approaches to representing and solving constraint satisfaction problems: constraint programming (CP; [Ros06]), answer set programming (ASP; [Bar03]), propositional satisfiability checking (SAT; [Bie09]), its extension to satisfiability modulo theories (SMT; [Nie06]), and many more. Each has its particular strengths: for example, CP systems support global constraints, ASP systems permit recursive definitions and offer default negation, whilst SAT solvers often exploit very efficient implementations. In many applications it would often be helpful to exploit the strengths of multiple approaches. Consider the problem of timetabling at an university (cf. [Jär09]). To model the problem, we need to express the mutual exclusion of events (for instance, we cannot place two events in the same room at the same time). A straightforward representation of such constraint with clauses and rules uses quadratic space. In contrast, global constraints such as *all-different* typically supported by CP systems can give a much more concise encoding. On the other hand, there are features which are hard to describe in traditional constraint programming, like the temporary unavailability of a particular room. However, this is easy to represent with non-monotonic rules such as those used in ASP. Such rules also provide a flexible mechanism for defining new relations on the basis of existing ones. This makes answer set programming an attractive approach to declarative problem solving. Indeed, ASP has been shown as the computational embodiment of non-monotonic reasoning (NMR; [Rei87]), adequate for common-sense reasoning and modelling of dynamic and incomplete knowledge.

As a primary candidate for an effective tool for knowledge representation and reasoning, ASP combines an expressive language with high-performance solving capacities. Largely

Key words and phrases: answer set programming, constraint logic programming, constraint processing.

Technical Communications of the 26th International Conference on Logic Programming, Edinburgh, July, 2010
Editors: Manuel Hermenegildo, Torsten Schaub
LIPIcs - Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/LIPIcs.ICLP.2010.255

based on SAT technology, modern ASP solvers offer an efficiency and scalability which in practice remain largely unmatched to date [Geb07a], able to encode all search problems within the first three levels of the polynomial hierarchy [Dre08]. Particularly of relevance here is the fact that clause learning is known to be more general and potentially more powerful than traditional learning in constraint solvers [Kat05]. Unlike SAT, however, ASP offers a uniform modelling language admitting variables. In fact, grounding non-propositional specifications is addressed in SAT anew for each application while ASP centralized this task in its grounders [Syr, Geb07c]. Answer set programming has been shown to be useful in numerous application scenarios, like bioinformatics [Bar04], crypto analysis [Aie01], configuration [Soi99], database integration [Leo05], diagnosis [Eit99], hardware design [Erd], model checking [Hel03], planing [Lif02], preference reasoning [Bre96], semantic web [Eit08], and – as a highlight among these applications – the high-level control of the space shuttle [Nog01].

However, as some CSP are more naturally modelled by using non-propositional constructs, like resources or functions over finite domains, in particular global constraints, the need to handle constraints beyond pure ASP is increasing. This naturally leads to the combination of ASP with constraint processing [Dec03] techniques, and is target of our research activity. A key contribution of our work is a novel approach to constraint logic programming (CLP; [Jaf94]) centred around ASP as both a declarative specification language and an efficient reasoning engine, enhanced with specialised propagators sufficient to solve interesting constraint satisfaction problems.

This research summary is organized as follows. We start by giving the necessary background and an overview of the existing literature. In turn, we formulate our objectives in Section 3. Section 4 gives a brief overview of the current status of our research and Section 5 presents some preliminary results. The last part summarises open questions which are target to our future work.

2. Background

2.1. Answer Set Programming

A *(normal) logic program* Π over a set of primitive propositions $\mathcal{A}$ is a finite set of rules of the form

$$a_0 \leftarrow a_1, \ldots, a_m, not\ a_{m+1}, \ldots, not\ a_n \tag{2.1}$$

where $0 \leq m \leq n$ and $a_i \in \mathcal{A}$ is an *atoms* for $0 \leq i \leq n$. A *literal* $\hat{a}$ is an atom a or its default negation *not* a. For a rule r, let $head(r) = a_0$ be the *head* of r and $body(r) = \{a_1, \ldots, a_m, not\ a_{m+1}, \ldots, not\ a_n\}$ the *body* of r. Furthermore, define $body(r)^+ = \{a_1, \ldots, a_m\}$ and $body(r)^- = \{a_{m+1}, \ldots, a_n\}$. The set of atoms occurring in a logic program Π is denoted by $atom(\Pi)$, and the set of bodies in Π is $body(\Pi) = \{body(r) \mid r \in \Pi\}$. For regrouping bodies sharing the same head a, define $body(a) = \{body(r) \mid r \in \Pi,\ head(r) = a\}$.

The semantics of a program is given by its answer sets. A set $X \subseteq \mathcal{A}$ is an *answer set* of a logic program Π over $\mathcal{A}$, if X is the $\subseteq$-minimal model of the *reduct* [Gel88]

$$\Pi^X = \{head(r) \leftarrow body(r)^+ \mid r \in \Pi,\ body(r)^- \cap X = \emptyset\}.$$

A rule r of the form (2.1) can be seen as a constraint on the answer sets of a program, stating that if $a_1, \ldots, a_m$ are in the answer set and none of $a_{m+1}, \ldots, a_n$ are included, then a_0 must be in the set. The answer sets are the key objects of interest in this paradigm and, hence, the task of ASP systems is to compute answer sets for programs. Such a system differs substantially from traditional logic programming systems, such as Prolog, which are goal-directed backward chaining query evaluation systems.

The semantics of important extensions to normal logic programs, such as choice rules, integrity and cardinality constraints, is given through program transformations that introduce additional propositions (cf. [Sim02]). A *choice rule* allows for the non-deterministic choice over atoms in $\{h_1, \ldots, h_k\}$ and has the following form:

$$\{h_1, \ldots, h_k\} \leftarrow a_1, \ldots, a_m, not\ a_{m+1}, \ldots, not\ a_n \tag{2.2}$$

An *integrity constraint*

$$\leftarrow a_1, \ldots, a_m, not\ a_{m+1}, \ldots, not\ a_n \tag{2.3}$$

is a short hand for a rule with an unsatisfiable head, and thus forbids its body to be satisfied in any answer set. A *cardinality constraint*

$$\leftarrow k\{a_1, \ldots, a_m, not\ a_{m+1}, \ldots, not\ a_n\} \tag{2.4}$$

is interpreted as no k literals of the set $\{a_1, \ldots, a_m, not\ a_{m+1}, \ldots, not\ a_n\}$ are included in an answer set. [Sim02] provides a transformation that needs just $\mathcal{O}(nk)$ rules. Alternatively, modern ASP solvers also incorporate specialised propagators for cardinality constraints that run in $\mathcal{O}(n)$.

Although the answer set semantics are propositional, atoms in $\mathcal{A}$ and can be constructed from a first-order signature $\Sigma_{\mathcal{A}} = (\mathcal{F}_{\mathcal{A}}, \mathcal{V}_{\mathcal{A}}, \mathcal{P}_{\mathcal{A}})$, where $\mathcal{F}_{\mathcal{A}}$ is a set of function symbols (including constant symbols), $\mathcal{V}_{\mathcal{A}}$ is a denumerable collection of (first-order) variables, and $\mathcal{P}_{\mathcal{A}}$ is a set of predicate symbols. The logic program over $\mathcal{A}$ is then obtained by a grounding process, systematically substituting all occurrences of variables $\mathcal{V}_{\mathcal{A}}$ by terms in $\mathcal{T}(\mathcal{F}_{\mathcal{A}})$, where $\mathcal{T}(\mathcal{F}_{\mathcal{A}})$ denotes the set of all ground terms over $\mathcal{F}_{\mathcal{A}}$. Atoms in $\mathcal{A}$ are formed from predicate symbols $\mathcal{P}_{\mathcal{A}}$ and terms in $\mathcal{T}(\mathcal{F}_{\mathcal{A}})$.

ASP systems usually use a *generate-and-test* [Bar03] technique to model a problem, by producing the space of solution candidates in the *generate* component and defining rules that filter invalid solutions in the *test* component. Typically, solutions are computed by applying *conflict-driven nogood learning* (CDNL; [Geb07b]). This combines search and propagation by recursively assigning the value of a proposition and using *unit-propagation* to determine logical consequences [Mit05].

2.2. Constraint Satisfaction Problem

The classic definition of a constraint satisfaction problem is as follows (cf. [Ros06]). A CSP is a triple (V, D, C) where V is set *variables* $V = \{v_1, \ldots, v_n\}$, D is an set of (finite) *domains* $D = \{D_1, \ldots, D_n\}$ such that each variable v_i has an associated domain $dom(v_i) = D_i$, and C is a set of *constraints*. A constraint c is a pair (R_S, S) where R_S is a k-ary *relation* on the variables in $S \subseteq V^k$, called the *scope* of c. In other words, R_S is a subset of the Cartesian product of the domains of the variables in S. To access the relation and the scope of c define $range(c) = R_S$ and $scope(c) = S$. For a *(constraint variable) assignment* $A : V \to \bigcup_{v \in V} dom(v)$ and a constraint $c = (R_S, S)$ with $S = (v_1, \ldots, v_k)$,

define $A(S) = (A(v_1), \ldots, A(v_k))$, and call c *satisfied* if $A(S) \in range(c)$. Given this, define the set of constraints satisfied by A as $sat_C(A) = \{c \mid A(scope(c)) \in range(c),\ c \in C\}$.

A binary constraint c has $|scope(c)| = 2$. For example, $v_1 \neq v_2$ ensures that v_1 and v_2 take different values. A global (or n-ary) constraint c has parametrized scope. For example, the *all-different* constraint ensures that a set of variables, $\{v_1, \ldots, v_n\}$ take all different values. This can be decomposed into $O(n^2)$ binary constraints, $v_i \neq v_j$ for $i < j$. However, such decomposition can hinder inference [Wal00]. An assignment A is a *solution* for a CSP iff it satisfies all constraints in C.

2.3. Constraint Programming

Constraint programming is a natural paradigm for solving constraint satisfaction problems. CP systems usually use a *constrain-and-generate* technique in which an initial deterministic phase assigns a domain to each of the constraint variables and imposes a number of constraints, then a non-deterministic phase generates and explores the solution space. Various heuristics affecting, for instance, the variable selection criteria and the ordering of the attempted values, can be used to guide the search. Each time a variable is assigned a value, a deterministic propagation stage is executed, pruning the set of values to be attempted for the other variables, i.e., enforcing a certain type of local consistency.

A binary constraint c is called *arc consistent* iff when a variable $v_1 \in scope(c)$ is assigned any value $d_1 \in dom(v_1)$, there exists a consistent value $d_2 \in dom(v_2)$ for the other variable v_2. An n-ary constraint c is *hyper-arc consistent* or *domain consistent* iff when a variable $v_i \in scope(c)$ is assigned any value $d_i \in dom(v_i)$, there exist compatible values in the domains of all the other variables $d_j \in dom(v_j)$ for all $1 \leq j \leq n,\ j \neq i$ such that $(d_1, \ldots, d_n) \in range(c)$.

The concepts of bound and range consistency are defined for constraints on ordered intervals. Let $min(D_i)$ and $max(D_i)$ be the minimum value and maximum value of the domain D_i. A constraint c is *bound consistent* iff when a variable v_i is assigned $d_i \in \{min(dom(v_i)), max(dom(v_i))\}$ (i.e. the minimum or maximum value in its domain), there exist compatible values between the minimum and maximum domain value for all the other variables in the scope of the constraint. Such an assignment is called a *bound support*. A constraint is *range consistent* iff when a variable is assigned any value in its domain, there exists a bound support. Notice that range consistency is in between domain and bound consistency.

2.4. Constraint Logic Programming

Constraint logic programming is a programming paradigm that naturally merges traditional constraint programming and logic programming. The goal is to bring advantages of logic programming based knowledge representation techniques to constraint programming.

Formally, a *constraint logic program* Π is defined as logic programs over an extended alphabet distinguishing regular and constraint atoms, denoted by $\mathcal{A}$ and $\mathcal{C}$, respectively, such that $head(r) \in \mathcal{A}$ for each $r \in \Pi$. Observe that a constraint logic program without constraints is in fact a (normal) logic program. Constraint atoms are identified with constraints via a function $\gamma : \mathcal{C} \to C$. For sets of constraints, define $\gamma(C') = \{\gamma(c) \mid c \in C'\}$ for $C' \subseteq C$. Similar to (normal) logic programs, the atoms in $\mathcal{A}$ and $\mathcal{C}$ can be constructed from a multi-sorted, first-order signature $\Sigma = (\mathcal{F}_\mathcal{A} \cup \mathcal{F}_\mathcal{C}, \mathcal{V}_\mathcal{A} \cup \mathcal{V}_\mathcal{C}, \mathcal{P}_\mathcal{A} \cup \mathcal{P}_\mathcal{C})$, where $\mathcal{F}_\mathcal{A} \cup \mathcal{F}_\mathcal{C}$ is a

finite set of function symbols (including constant symbols), $\mathcal{V}_\mathcal{A}$ is a denumerable collection of regular variable symbols, $\mathcal{V}_\mathcal{C} \subseteq \mathcal{T}(\mathcal{F}_\mathcal{A})$ is a set of constraint variable symbols, and $\mathcal{P}_\mathcal{A} \cup \mathcal{P}_\mathcal{C}$ is a finite set of predicate symbols, where $\mathcal{P}_\mathcal{A}$ and $\mathcal{P}_\mathcal{C}$ are disjoint. While the atoms in $\mathcal{A}$ are formed as discussed before, the ones in $\mathcal{C}$ are constructed from predicate symbols $\mathcal{P}_\mathcal{C}$ and $(\mathcal{F}_\mathcal{C}, \mathcal{V}_\mathcal{C})$-terms. This definition follows Gebser et. al. [Geb09c] and tolerates occurrences of similar ground terms in atoms of both $\mathcal{A}$ and $\mathcal{C}$.

An integration of constraint and logic programming has been studied mainly from the point of view of extending Prolog implementations by allowing, e.g., constraints on finite domains in the rules and by integrating the necessary constraint solvers into the logic programming system. Although a Prolog-based CLP approach follows the constrain-and-generate technique from constraint programming systems, it has many advantages, including recursive definitions.

However, this significantly differs from our approach where the rules have a declarative semantics and can be understood themselves as constraints on solutions for the program.

2.5. Constraint Answer Set Programming

We extend the answer set semantics to constraint logic programs and define the *constraint reduct* as

$$\Pi^A = \{head(r) \leftarrow body(r)|_\mathcal{A} \mid r \in \Pi, \\ \gamma(body(r)^+|_\mathcal{C}) \subseteq sat_C(A),\ \gamma(body(r)^-|_\mathcal{C}) \cap sat_C(A) = \emptyset\}.$$

Then, a set $X \subseteq \mathcal{A}$ is a *constraint answer set* of Π with respect to A, if X is an answer set of Π^A. An open question which is target to intensive research is how to efficiently incorporate answer set programming engines and constraint processing, i.e., how to generate assignments and enforce satisfaction (or violation, respectively) of constraints in $\gamma(\mathcal{C})$. We identified three different approaches: (1) translation-based techniques, (2) integration of constraint solvers, and (3) usage of additional propagators, such as aggregates.

Translation-based Techniques. Generally, in a translation-based approach all parts of the model are mapped into a single constraint language for which highly efficient off-the-shelf solvers are available. Previous work has mostly focussed on the translation of specific types of constraints to SAT. For example, pseudo-Boolean constraints (linear constraints over Boolean variables), including the special case of Boolean cardinality constraints, have been Booleanised such that a SAT solver can compete with the best existing native pseudo-Boolean solvers [Eén06, Sin05, Bai03, Bai06, Bai09]. Integer linear constraints have also been translated to SAT by transforming all constraints into primitive comparisons, of the form $v \leq c$, and encoding each of these by a different Boolean variable for each integer variable v and integer value c [Tam06].

Although efficient, these results have a number of limitations. First, the types of constraints dealt with are limited. Second, the techniques proposed are not necessarily compatible, thus making the translation of a heterogeneous constraint model difficult in both practice and theory. The latter is faced in [Hua08] presenting translation techniques to SAT at language level by systematically Booleanising a general constraint language, rather than specialised constraint types. However, this comes with the price of weaker encodings in terms of propagation power and loss of explicit domain knowledge and structure. It remains a difficult task to define universal SAT encodings that are both compact and enforce a strong type of consistency on the original model.

ASP is put forward as a constraint programming paradigm in [Nie99a], also showing that answer set programming embeds SAT but provides a more expressive framework from a knowledge representation point of view. An empirical comparison of the performance of ASP and CLP systems on solving combinatorial problems in [Dov09] proves ASP encodings to be more compact, more declarative, and highly competitive. However, techniques for translating constraint variables and constraint propagation algorithms to ASP received few attention in our context. A first study on introducing high-level statements for multi-valued propositions into the language of ASP was conducted in [Geb09a]. As we shall see, a translational approach to constraint answer set solving [Dre10b] offers an efficient way to seamlessly combine the propagators of all constraints, through the unit-propagation of an ASP solver. In particular, queueing of propagators becomes irrelevant as all constraints are always propagated at once. Another major strength is that the unified conflict resolution framework can exploit constraint interdependencies, which may lead to faster propagation between constraints.

Hybrid Approach. In a hybrid system, theory-specific solvers interact in order to compute solutions to the whole constraint model, similar to satisfiability modulo theories. Hence, the key idea of an integrative approach is to incorporate constraint predicates into propositional formulas, and extending an ASP solver's decision engine for a more high-level proof procedure. This becomes increasingly attractive in constraint answer set programming when the variables in a constraint model have significantly large domains, and therefore, computing the ground instantiation has huge memory requirements [Pal09]. Related work was conducted in [Geb09c, Bas, Mel08b, Mel08a]. While Gelfond et. al. [Bas, Mel08b, Mel08a] view ASP and CP solvers as blackboxes, Gebser et. al. [Geb09c] embed a CP solver into an ASP solver adding support for advanced backjumping and conflict-driven learning techniques. However, the computational impact compared to traditional CP is limited, first, because their methods lack support for global constraints, and second, the communication between the ASP and CP solvers with respect to learning constraint interdependencies is restricted. Balduccini [Bal09] added support for global constraints but sees constraint answer set programming largely as a CSP specification language. In particular, his approach does not allow constraint literals in the body of a rule, which does not coincide with our general notion of constraint logic programming.

Formulation of Additional Propagators. Little attention is paid to constraint answer set programming through decomposition to ASP with usage of additional propagators, such as aggregates. Aggregations and other forms of set constructions have been shown to be useful extensions to ASP [Del]. In fact, a lack of aggregation capabilities may lead to an exponential growth in the number of rules required to model a CSP [Bar03]. Therefore, it is common to most ASP solvers to incorporate specialised algorithms, for instance, the treatment of cardinality constraints (2.4), and their generalisation to weight constraints [Nie99b]. Work on a generic framework which provides an elegant treatment of such extensions was conducted in [Elk] where external constraint propagators are employed for their handling. However, it does not carry over to modern ASP solving technology based on conflict-driven learning. A first comprehensive approach to integrating specialised algorithms for weight constraint rules into CDNL is presented in [Geb09b].

3. Research Objective

We want to put forward constraint answer set programming as a novel approach to constraint (logic) programming. Therefore, we (1) investigate efficient encodings of propagation algorithms in answer set programming, (2) study the integration of techniques from constraint processing into answer set programming engines, and (3) define a modelling language for constraint logic programming under answer set semantics, that can be accepted by the community. Furthermore, we (4) want to implement our techniques in state-of-the-art systems.

4. State of the Research

Our research is in a very early stage. In a Master's project we introduced a novel, translation-based approach to constraint answer set solving [Dre10b] that allows for learning constraint interdependencies to improve propagation between constraints. As part of a Master's thesis we also started an investigation of symmetry-breaking in the context of answer set programming to eliminate symmetric parts of the search space and, thereby, simplify the solution process [Dre10a].

5. Preliminary Results

In our translational approach to constraint answer set solving, a constraint logic program is compiled into a logic program by adding an ASP decomposition of all constraints comprised in the constraint logic program. The constraint answer sets can then be obtained by applying the same algorithms as for calculating answer sets, e.g. CDNL. Since all variables will be shared between constraints, nogood learning techniques as in CDNL exploit constraint interdependencies. This can improve propagation between constraints. We identify a number of choices of how to decompose constraints on multi-valued propositions, e.g. constraint variables, in answer set programming. Namely, we propose a *direct*, *support*, *range*, and *bound* representation of constraints [Dre10b] each generically encoding a propagation algorithm in ASP (using rule types 2.1–2.4) that achieves, e.g., arc, range and bound consistency on the original constraint (or its binary decomposition, respectively), using unit-propagation. In particular, we present the following results:

Theorem 5.1. *Enforcing arc consistency on the binary decomposition of the original constraint prunes more values from the variables domain than unit-propagation on its direct encoding.*

Theorem 5.2. *Unit-propagation on the support encoding enforces arc consistency on the binary decomposition of the original constraint.*

Theorem 5.3. *Unit-propagation on the range encoding enforces range consistency on the original constraint.*

Theorem 5.4. *Unit-propagation on the bound encoding enforces bound consistency on the original constraint.*

We illustrate our approach on an encoding of the global all-different constraint enforcing, e.g., bound consistency by pruning Hall intervals [Lec96]. Surprisingly, a very simple decomposition into ASP can simulate a complex propagation algorithm like from

Leconte's [Lec96] with a similar overall complexity of reasoning. Our techniques were formulated as preprocessing and can be applied to any ASP system without changing its source code, which allows for programmers to select the solvers that best fit their needs. We have empirically evaluated their performance on benchmarks from CSP and found them outperforming integrated constraint answer set programming systems as well as pure CP solvers.

However, many CSP exhibit symmetries which can frustrate a search algorithm to fruitlessly explore independent symmetric subspaces. We have investigated symmetry-breaking in the context of answer set programming [Dre10a]. In particular, we propose a reduction from symmetry detection of disjunctive logic programs to the automorphisms of a coloured digraph. Our techniques are formulated as a completely automated flow that (1) starts with a logic program, (2) detects all of its permutational symmetries, (3) represents all symmetries implicitly and always with exponential compression, (4) adds symmetry-breaking constraints that do not affect the existence of answer sets. We have empirically evaluated symmetry-breaking on difficult CSP and got promising results. In many cases, symmetry-breaking lead to significant pruning of the search space and yield solutions to problems which are otherwise intractable. We also observe a significant compression of the solution space which makes symmetry-breaking attractive whenever all solutions have to be post-processed.

6. Future Work

Regarding symmetry-breaking answer set solving, it is often reasonable to assume that the symmetries for a problem are known. For particular symmetries, there are more efficient breaking methods, for instance, the global value precedence constraint (cf. [Wal06]).

Therefore, future work concerns, but is not limited to, the integration of further constraints useful in constraint answer set programming. In particular, we are interested in decompositions of constraints using the full range of propagators available in state-of-the-art ASP systems, and if necessary, extending ASP solving by further useful algorithms that make constraint answer set programming an efficient approach to constraint logic programming.

Acknowledgements

The author wishes to acknowledge the continuing support of Thomas Eiter, Torsten Schaub, and Toby Walsh.

References

[Aie01] L. Aiello and F. Massacci. Verifying security protocols as planning in logic programming. *ACM Transactions on Computational Logic*, 2(4):542–580, 2001.

[Bai03] O. Bailleux and Y. Boufkhad. Efficient CNF encoding of boolean cardinality constraints. In *Proceedings of CP'03*, pp. 108–122. Springer, 2003.

[Bai06] O. Bailleux, Y. Boufkhad, and O. Roussel. A translation of pseudo boolean constraints to SAT. *Journal of Satisfiability*, 2(1-4):191–200, 2006.

[Bai09] O. Bailleux, Y. Boufkhad, and O. Roussel. New encodings of pseudo-boolean constraints into CNF. In *Proceedings of SAT'09*, pp. 181–194. Springer, 2009.

[Bal09] M. Balduccini. CR-prolog as a specification language for constraint satisfaction problems. In *Proceedings of LPNMR'09*, pp. 402–408. Springer, 2009.

[Bar03] C. Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2003.

[Bar04] C. Baral, K. Chancellor, N. Tran, N. Tran, A. Joy, and M. Berens. A knowledge based approach for representing and reasoning about signaling networks. In *Proceedings of ISMB/ECCB'04*, pp. 15–22. 2004.

[Bas] S. Baselice, P. Bonatti, and M. Gelfond. Towards an integration of answer set and constraint solving. In *Proceedings of ICLP'05*, pp. 52–66. Springer.

[Bie09] A. Biere, M. Heule, H. van Maaren, and T. Walsh (eds.). *Handbook of Satisfiability*. IOS Press, 2009.

[Bre96] G. Brewka and T. Eiter. Preferred answer sets for extended logic programs. In *Proceedings of KR'96*, pp. 86–97. Morgan Kaufmann Publishers, 1996.

[Dec03] R. Dechter. *Constraint Processing*. Morgan Kaufmann Publishers, 2003.

[Del] T. Dell'Armi, W. Faber, G. Ielpa, N. Leone, and G. Pfeifer. Aggregate functions in disjunctive logic programming: Semantics, complexity, and implementation in DLV. In *Proceedings of IJCAI'93*, pp. 847–852. Morgan Kaufmann Publishers.

[Dov09] A. Dovier, A. Formisano, and E. Pontelli. An empirical study of constraint logic programming and answer set programming solutions of combinatorial problems. *Journal of Experimental and Theoretical Artificial Intelligence*, 21(2):79–121, 2009.

[Dre08] C. Drescher, M. Gebser, T. Grote, B. Kaufmann, A. König, M. Ostrowski, and T. Schaub. Conflict-driven disjunctive answer set solving. In *Proceedings of KR'08*, pp. 422–432. AAAI Press, 2008.

[Dre10a] C. Drescher, O. Tifrea, and T. Walsh. Symmetry-breaking answer set solving. In *Proceedings of ICLP'10 Workshop ASPOCP*. 2010. To appear.

[Dre10b] C. Drescher and T. Walsh. A translational approach to constraint answer set solving. In *Proceedings of ICLP'10*. Cambridge University Press, 2010. To appear.

[Eén06] N. Eén and N. Sörensson. Translating pseudo-Boolean constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation*, 2:1–26, 2006.

[Eit99] T. Eiter, W. Faber, N. Leone, and G. Pfeifer. The diagnosis frontend of the dlv system. *AI Communications*, 12(1-2):99–111, 1999.

[Eit08] T. Eiter, G. Ianni, T. Lukasiewicz, R. Schindlauer, and H. Tompits. Combining answer set programming with description logics for the semantic web. *Artificial Intelligence*, 172(12-13):1495–1539, 2008.

[Elk] I. Elkabani, E. Pontelli, and T. Son. Smodels with CLP and its applications: A simple and effective approach to aggregates in ASP. In *Proceedings of ICLP'04*, pp. 73–89. Springer.

[Erd] E. Erdem and M. Wong. Rectilinear Steiner tree construction using answer set programming. In *Proceedings of ICLP'04*, pp. 386–399. Springer.

[Geb07a] M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. clasp: A conflict-driven answer set solver. In *Proceedings of LPNMR'07*, pp. 260–265. Springer, 2007.

[Geb07b] M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. Conflict-driven answer set solving. In *Proceedings of IJCAI'07*, pp. 386–392. AAAI Press/The MIT Press, 2007.

[Geb07c] M. Gebser, T. Schaub, and S. Thiele. GrinGo: A new grounder for answer set programming. In *Proceedings of LPNMR'07*, pp. 266–271. Springer, 2007.

[Geb09a] M. Gebser, H. Hinrichs, T. Schaub, and S. Thiele. xpanda: A (simple) preprocessor for adding multi-valued propositions to ASP. In *Proceedings of WLP'09*. 2009.

[Geb09b] M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub. On the implementation of weight constraint rules in conflict-driven ASP solvers. In *Proceedings of ICLP'09*, pp. 250–264. Springer, 2009.

[Geb09c] M. Gebser, M. Ostrowski, and T. Schaub. Constraint answer set solving. In *Proceedings of ICLP'09*, pp. 235–249. Springer, 2009.

[Gel88] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proceedings of ICLP'88*, pp. 1070–1080. The MIT Press, 1988.

[Hel03] K. Heljanko and I. Niemelä. Bounded LTL model checking with stable models. *Theory and Practice of Logic Programming*, 3(4-5):519–550, 2003.

[Hua08] J. Huang. Universal booleanization of constraint models. In *Proceedings of CP'08*, pp. 144–158. Springer, 2008.

[Jaf94] J. Jaffar and M. J. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19-20:503–581, 1994.

[Jär09] M. Järvisalo, E. Oikarinen, T. Janhunen, and I. Niemelä. A module-based framework for multi-language constraint modeling. In *Proceedings of LPNMR'09*, pp. 155–169. Springer, 2009.

[Kat05] G. Katsirelos and F. Bacchus. Generalized nogoods in CSPs. In *Proceedings of AAAI'05*, pp. 390–396. AAAI Press, 2005.

[Lec96] M. Leconte. A bounds-based reduction scheme for constraints of difference. In *CP'96, Second International Workshop on Constraint-based Reasoning*. 1996.

[Leo05] N. Leone, G. Greco, G. Ianni, V. Lio, G. Terracina, T. Eiter, W. Faber, M. Fink, G. Gottlob, R. Rosati, D. Lembo, M. Lenzerini, M. Ruzzi, E. Kalka, B. Nowicki, and W. Staniszkis. The INFOMIX system for advanced integration of incomplete and inconsistent data. In *Proceedings of SIGMOD'05*, pp. 915–917. 2005.

[Lif02] V. Lifschitz. Answer set programming and plan generation. *Artificial Intelligence*, 138(1-2):39–54, 2002.

[Mel08a] V. Mellarkod and M. Gelfond. Integrating answer set reasoning with constraint solving techniques. In *Proceedings of FLOPS'08*, pp. 15–31. Springer, 2008.

[Mel08b] 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.

[Mit05] D. Mitchell. A SAT solver primer. *Bulletin of the European Association for Theoretical Computer Science*, 85:112–133, 2005.

[Nie99a] 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.

[Nie99b] I. Niemelä, P. Simons, and T. Soininen. Stable model semantics of weight constraint rules. In *Proceedings of NMR'99*, pp. 317–333. Springer, 1999.

[Nie06] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *Journal of the ACM*, 53(6):937–977, 2006.

[Nog01] M. Nogueira, M. Balduccini, M. Gelfond, R. Watson, and M. Barry. An A-prolog decision support system for the space shuttle. In *Proceedings of PADL'01*, pp. 169–183. Springer, 2001.

[Pal09] A. D. Palù, A. Dovier, E. Pontelli, and G. Rossi. Answer set programming with constraints using lazy grounding. In *Proceedings of ICLP'09*, pp. 115–129. Springer, 2009.

[Rei87] R. Reiter. Nonmonotonic reasoning. *Annual Review of Computer Science*, 2:147–187, 1987.

[Ros06] F. Rossi, P. van Beek, and T. Walsh (eds.). *Handbook of Constraint Programming*. Elsevier, 2006.

[Sim02] P. Simons, I. Niemelä, and T. Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1-2):181–234, 2002.

[Sin05] C. Sinz. Towards an optimal cnf encoding of boolean cardinality constraints. In *Proceedings of CP'05*, pp. 827–831. Springer, 2005.

[Soi99] T. Soininen and I. Niemelä. Developing a declarative rule language for applications in product configuration. In *Proceedings of PADL'99*, pp. 305–319. Springer, 1999.

[Syr] T. Syrjänen. Lparse 1.0 user's manual.

[Tam06] N. Tamura, A. Taga, S. Kitagawa, and M. Banbara. Compiling finite linear csp into sat. In *Proceedings of CP'06*, pp. 590–603. Springer, 2006.

[Wal00] T. Walsh. SAT v CSP. In *Proceedings of CP'00*, pp. 441–456. Springer, 2000.

[Wal06] T. Walsh. Symmetry breaking using value precedence. In *Proceedings of ECAI'06*, pp. 168–172. IOS Press, 2006.

Technical Communications of the International Conference on Logic Programming, 2010 (Edinburgh), pp. 265–269
http://www.floc-conference.org/ICLP-home.html

TOWARDS A GENERAL ARGUMENTATION SYSTEM BASED ON ANSWER-SET PROGRAMMING

SARAH ALICE GAGGL

Institute of Information Systems 184/2,
Vienna University of Technology,
Favoritenstrasse 9-11,
A-1040 Vienna,
Austria
E-mail address: gaggl@dbai.tuwien.ac.at
URL: http://www.dbai.tuwien.ac.at/staff/gaggl/

Abstract. Within the last years, especially since the work proposed by Dung in 1995, argumentation has emerged as a central issue in Artificial Intelligence. With the so called argumentation frameworks (AFs) it is possible to represent statements (arguments) together with a binary attack relation between them. The conflicts between the statements are solved on a semantical level by selecting acceptable sets of arguments. An increasing amount of data requires an automated computation of such solutions. Logic Programming in particular Answer-Set Programming (ASP) turned out to be adequate to solve problems associated to such AFs. In this work we use ASP to design a sophisticated system for the evaluation of several types of argumentation frameworks.

Introduction and Problem Description

Argumentation systems provide a formal way of dealing with conflicting knowledge. In particular argumentation frameworks (AFs) introduced by Dung [11] in 1995 are used to represent statements together with a relation denoting rebuttals between them, where the internal structure of the statements is of no interest for the evaluation of the framework. Several semantics have been defined to solve the inherent conflicts between the statements by selecting acceptable subsets of them. The most recognized of them are the stable, preferred and grounded semantics. The following example illustrates the definition and graphical representation of an AF.

Example 1. Let the AF $F = (A, R)$ be defined as follows, $A = \{a, b, c, d\}$ is the set of *arguments*, and $R = \{(a, b), (b, c), (b, d), (c, d)\}$ is the *attack relation* between the arguments. Let now $S = \{a, c\}$ be a set of *acceptable* arguments (also called a solution of F wrt a given semantics). Such an AF can be represented as a directed graph as shown in Figure 1.

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

Key words and phrases: Argumentation, Implementation, Answer-Set Programming.

This work was supported by the Vienna Science and Technology Fund (WWTF) under grant ICT08-028.

Technical Communications of the 26th International Conference on Logic Programming, Edinburgh, July, 2010
Editors: Manuel Hermenegildo, Torsten Schaub
LIPIcs - Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/LIPIcs.ICLP.2010.265

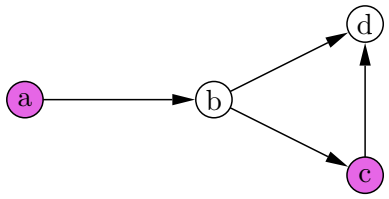


Figure 1: The argumentation framework F from Example 1.

Lets have a closer look why the set S represents a solution for our framework. The argument a is not attacked by any argument, hence it can be clearly viewed as acceptable. The argument b is only attacked by a, and as we accepted a, we can not also accept b, because two arguments attacking each other would not lead to a meaningfull solution. Whereas, we can say the argument c is defended by a against the attack from b and thus can be included into the solution. Finally, the argument d is also defended by a against the attack from b, but it is still attacked by c which already is part of the solution. Hence, d cannot be contained in S.

Within the last years, AFs became a main research area in Artificial Intelligence (AI). Recently two textbooks on argumentation [8, 22] and a special issue on argumentation in AI [7] have been published. Furthermore the Conference on Computational Models of Argument (COMMA) is held every second year.

The increasing interest in this topic resulted in the fact that Dung's approach has been extended and generalized continuously according to specific application scenarios like Multi-Agent Systems and Law Research. On the one hand, various semantics like semi-stable [9] or ideal semantics [12] have been introduced to adjust to the specific scenarios, on the other hand, the framework in itself has been adapted by modifying the notion of rebuttal [1], introducing new relations between the statements [2] or augmenting them with priorities [6].

For small instances it is quite easy to evaluate the frameworks under different semantics, but an increasing amount of data requires a sophisticated system for the evaluation. Argumentation problems are in general intractable, for instance deciding if an argument is contained in some preferred extensions is known to be *NP*-complete. Therefore, developing dedicated algorithms for the different reasoning problems is non-trivial. A promising way to implement such systems is to use a reduction method, where the given problem is translated into another language, for which sophisticated systems already exist. Logic Programming methods, in particular Answer-Set Programming (ASP) [17] turned out to be a promising direction for this aim, since it not only allows for a concise representation of concepts inherent to argumentation semantics, but also offers sophisticated off-the-shelves solvers which can be used as core computation engines (like Smodels, DLV, clasp or GnT [10]).

1. Background and Overview of the Existing Literature

Previous work has demonstrated that Logic Programming is adequate to encode argumentation problems. Dung has already mentioned in [11] the strong relation between argumentation and Logic Programming. Nieves et. al. proposed in [21] an encoding schema

to represent AFs as logic programs, and they showed how different semantics for logic programs can be used to compute different forms of extensions using this particular schema. Furthermore, Nieves et. al proposed in [21] an approach to compute preferred extensions by means of logic programs which requires a recompilation of the encoding for each particular AF. Similarly, [24] also provide ASP encodings for different semantics. In contrast to our work, their encodings for complete and stable semantics are based on labelings, whereas for grounded, preferred and semi-stable semantics they use a meta-programming technique applying additional translations for each AF into normal logic programs. One major difference of our system ASPARTIX [15] to this work is that it uses a *fixed* query for all semantics, which requires the actual instance just as an input database. For the concrete queries, we refer to [15] and for the ideal semantics to [16].

2. Goal of the Research

We want to provide a system for argumentation frameworks which is capable to deal with a broad range of argumentation semantics and generalizations of AFs. We turn our attention especially on a user-friendly implementation which does not require any background knowledge on Logic Programming or ASP. Hence, the user just needs to set up the input database, consisting of problem instance, and select the desired evaluation. We believe that this system can be useful for researchers for analysing and comparing argumentation systems, as well as a versatile decision support system. Especially, we will exploit ASP for more advanced problems. On the one hand, we plan to make use of the rich syntax of ASP (e.g., weak constraints, aggregates, weight constraints, etc.) to deal with weights on arguments or attacks [14, 19]; on the other hand, we want to combine our encodings in order to represent reasoning problems where several semantics come into play (e.g. the coherence problem [13] which decides whether for a given AF F every preferred extension of F is also a stable extension of F).

3. Current Status of the Research

In [15] we presented the first version of ASPARTIX, an ASP tool, which makes use of DLV [18]. This system was designed to compute the basic semantics defined by Dung in [11] such as admissible, complete, preferred, grounded and stable semantics. Additionally we provide encodings for semi-stable [9] and ideal semantics [12]. Furthermore, ASPARTIX can be used to evaluate Preference-based Argumentation Frameworks [1], Value-based Argumentation Frameworks [6], and Bipolar Argumentation Frameworks [2]. All necessary programs to run ASPARTIX are available at

`http://www.dbai.tuwien.ac.at/research/project/argumentation/systempage/`

Currently we are focusing on the encodings of the next generation of argumentation semantics and extensions. Recently, we incorporated the SCC-recursive *cf2* semantics [5] into ASPARTIX. Further encodings include the resolution-based semantics due to Baroni and Giacomin [4] as well as some generalizations of AFs like AFs with Recursive Attacks (AFRAs) [3], Extended AF (EAF) [20] and Dynamic AFs (DAFs) [23].

4. Preliminary Results Accomplished

With the system ASPARTIX, we provide ASP encodings for most of the semantics and frameworks proposed so far. As stated in [15], the encodings are adequate from a complexity point of view. One major advantage of ASPARTIX is that it is independent from the concrete AF to process. It serves as an interpreter which takes an AF given as input. Although there is no advantage of the interpreter approach from a theoretical point of view (as long as the reductions are polynomial-time computable), there are several practical ones. The interpreter is easier to understand, easier to debug, and easier to extend.

5. Open Issues and Expected Achievements

Future work includes a comparison between the different ASP solver and systems wrt our encodings. Especially we will perform run-time tests with the grounders Lparse and Gringo and the solvers Smodels, claspD, GnT2 as well as the system DLV [10]. Preliminary tests showed that our system is capable to deal with frameworks of more than 150 arguments.

As another direction of future work, we will offer a web application of ASPARTIX including a graphical representation of the problem instance and the solution. Hence, researchers can use our system without downloading or installation of any program or ASP solver.

References

[1] Leila Amgoud and Claudette Cayrol. A Reasoning Model Based on the Production of Acceptable Arguments. *Ann. Math. Artif. Intell.*, 34(1-3):197–215, 2002.

[2] Leila Amgoud, Claudette Cayrol, Marie-Christine Lagasquie, and Pierre Livet. On Bipolarity in Argumentation Frameworks. *International Journal of Intelligent Systems*, 23:1–32, 2008.

[3] Pietro Baroni, Federico Cerutti, Massimiliano Giacomin, and Giovanni Guida. Encompassing Attacks to Attacks in Abstract Argumentation frameworks. In Claudio Sossai and Gaetano Chemello, editors, *Proceedings of the 10th European Conference on Symbolic and Quantitative Approaches to Reasoning with Uncertainty* (*ECSQARU 2009*), volume 5590 of *Lecture Notes in Computer Science*, pages 83–94, 2009.

[4] Pietro Baroni and Massimiliano Giacomin. Resolution-Based Argumentation Semantics. In Philippe Besnard, Sylvie Doutre, and Anthony Hunter, editors, *Proceedings of the 2nd Conference on Computational Models of Argument* (*COMMA 2008*), volume 172 of *Frontiers in Artificial Intelligence and Applications*, pages 25–36. IOS Press, 2008.

[5] Pietro Baroni, Massimiliano Giacomin, and Giovanni Guida. SCC-Recursiveness: A General Schema for Argumentation Semantics. *Artif. Intell.*, 168(1-2):162–210, 2005.

[6] Trevor J. M. Bench-Capon. Persuasion in Practical Argument Using Value-based Argumentation frameworks. *J. Log. Comput.*, 13(3):429–448, 2003.

[7] Trevor J. M. Bench-Capon and Paul E. Dunne. Special Issue on Argumentation in Artificial Intelligence. *Artificial Intelligence*, 171(10-15):619–641, 2007. Argumentation in Artificial Intelligence.

[8] Philippe Besnard and Anthony Hunter. *Elements of Argumentation*. The MIT Press, 2008.

[9] Martin Caminada. Semi-Stable Semantics. In Paul E. Dunne and Trevor J. M. Bench-Capon, editors, *Proceedings of the 1st Conference on Computational Models of Argument* (*COMMA 2006*), volume 144 of *Frontiers in Artificial Intelligence and Applications*, pages 121–130. IOS Press, 2006.

[10] Marc Denecker, Joost Vennekens, Stephen Bond, Martin Gebser, and Miroslaw Truszczynski. The Second Answer Set Programming Competition. In Esra Erdem, Fangzhen Lin, and Torsten Schaub, editors, *Proceedings of the 10th International Conference on Logic Programming and Nonmonotonic Reasoning,* (*LPNMR 2009*), volume 5753 of *LNCS*, pages 637–654. Springer, 2009.

[11] Phan M. 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.

[12] Phan M. Dung, Paolo Mancarella, and Francesca Toni. Computing Ideal Sceptical Argumentation. *Artif. Intell.*, 171(10-15):642–674, 2007.

[13] Paul E. Dunne and Trevor J. M. Bench-Capon. Coherence in Finite Argument Systems. *Artif. Intell.*, 141(1/2):187–203, 2002.

[14] Paul E. Dunne, Anthony Hunter, Peter McBurney, Simon Parsons, and Michael Wooldridge. Inconsistency Tolerance in Weighted Argument Systems. In Carles Sierra, Cristiano Castelfranchi, Keith S. Decker, and Jaime Simão Sichman, editors, *Proceedings of the 8th International Joint Conference on Autonomous Agents and Multiagent Systems,* (*AAMAS 2009*), pages 851–858, 2009.

[15] Uwe Egly, Sarah A. Gaggl, and Stefan Woltran. Answer-Set Programming Encodings for Argumentation Frameworks. Accepted for publication in *Argument and Computation*, 2010; a short version appeared as Technical Report, Technische Universität Wien, DBAI-TR-2008-62.

[16] Wolfgang Faber and Stefan Woltran. Manifold Answer-Set Programs for Meta-Reasoning. In *LPNMR*, pages 115–128, 2009.

[17] Michael Gelfond and Vladimir Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Comput.*, 9(3/4):365–386, 1991.

[18] Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. The DLV System for Knowledge Representation and Reasoning. *ACM Trans. Comput. Log.*, 7(3):499–562, 2006.

[19] Diego C. Martínez, Alejandro J. García, and Guillermo R. Simari. An Abstract Argumentation Framework with Varied-strength attacks. In *Proceedings of the 11h International Conference on Principles of Knowledge Representation and Reasoning,* (*KR 2008*), pages 135–144, 2008.

[20] Sanjay Modgil. Reasoning about Preferences in Argumentation Frameworks. *Artif. Intell.*, 173(9-10):901–934, 2009.

[21] Juan C. Nieves, Mauricio Osorio, and Ulises Cortés. Preferred Extensions as Stable Models. *Theory and Practice of Logic Programming*, 8(4):527–543, 2008.

[22] Iyad Rahwan and Guillermo R. Simari. *Argumentation in Artificial Intelligence*. Springer Publishing Company, Incorporated, 2009.

[23] Nicolas D. Rotstein, Martin O. Maguillansky, Alejandro J. Garcia, and Guillermo R. Simari. An Abstract Argumentation Framework for Handling Dynamics. In *Proceedings of the 12th International Workshop on Non-Monotonic Reasoning* (*NMR'08*), pages 131–139, Sydney, Australia, September 2008.

[24] Toshiko Wakaki and Katsumi Nitta. Computing Argumentation Semantics in Answer Set Programming. In Hiromitsu Hattori, Takahiro Kawamura, Tsuyoshi Idé, Makoto Yokoo, and Yohei Murakami, editors, *New Frontiers in Artificial Intelligence* (*JSAI 2008*)*, Conference and Workshops*, volume 5447 of *LNCS*, pages 254–269. Springer, 2008.

Technical Communications of the International Conference on Logic Programming, 2010 (Edinburgh), pp. 270–276
http://www.floc-conference.org/ICLP-home.html

MODELS FOR TRUSTWORTHY SERVICE AND PROCESS ORIENTED SYSTEMS

HUGO A. LÓPEZ [1]

Programming, Logic and Semantics group, IT University.
Rued Langgaards Vej 7, 2300 Copenhagen, Denmark
E-mail address: lopez@itu.dk
URL: http://www.itu.dk/~hual/

Abstract. Service and process-oriented systems promise to provide more effective business and work processes and more flexible and adaptable enterprise IT systems. However, the technologies and standards are still young and unstable, making research in their theoretical foundations increasingly important. Our studies focus on two dichotomies: the global/local views of service interactions, and their imperative/declarative specification. A global view of service interactions describes a process as a protocol for interactions, as e.g. an UML sequence diagram or a WS-CDL choreography. A local view describes the system as a set of processes, e.g. specified as a π -calculus or WS-BPEL process, implementing each participant in the process. While the global view is what is usually provided as specification, the local view is a necessary step towards a distributed implementation. If processes are defined imperatively, the control flow is defined explicitly, e.g. as a sequence or flow graph of interactions/commands. In a declarative approach processes are described as a collection of conditions they should fulfill in order to be considered correct. The two approaches have evolved rather independently from each other. Our thesis is that we can provide a theoretical framework based on typed concurrent process and concurrent constraint calculi for the specification, analysis and verification of service and process oriented system designs which bridges the global and local view and combines the imperative and declarative specification approaches, and can be employed to increase the trust in the developed systems. This article describes our main motivations, results and future research directions.

1. Introduction

As recently pointed out by the ICT theme of EU Seventh Framework Programme (FP7), the need of trustworthy and pervasive services infrastructure is considered one of the three mayor challenges in ICT for the next ten years. The "future internet" proposes questions in terms of scalability, mobility, flexibility, security, trust and robustness to the more than thirty years old current Internet architecture. A vast landscape of application and ever-changing requirements and environments must be supported, and new ways of interaction must be devised, coping with safety and reliability in their coordination methods.

1998 ACM Subject Classification: F.3.2: Semantics of Programming Languages, F.4.3: Formal Languages.

Key words and phrases: Concurrent Constraint Calculi, Session Types, Logic, Service and Process oriented computing.

Technical Communications of the 26th International Conference on Logic Programming, Edinburgh, July, 2010
Editors: Manuel Hermenegildo, Torsten Schaub
LIPIcs - Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/LIPIcs.ICLP.2010.270

The line of research investigating such questions has been constantly expanding since the early nineties, both combining approaches from the academia and the industry. As result of such efforts, its been normally hard to differentiate between similar derived fields, like Business Processes, Workflow technologies and Service Oriented Computing. A Business Process is the set of steps executed in order to fulfill a (business) goal. Business processes have always been at the hearth of companies interests, and the obvious goal has been to develop better, cheaper, and faster processes, incrementing the profits of the company. Workflows came as an initial response for the need of proper descriptions of business processes, providing a framework for the specification and automation of processes by means of activities respecting a business logic. They aim at integrating coarse-grained components and have a single place where the business logic is specified. Furthermore, Service Oriented Computing (SOC) opens a new different horizon by distributing the places where the business logic is defined: now, small process units (services) can be shared between different organizations, so each of them can fulfill their business goals by reusing and outsourcing services.

Giving the intrinsic complexity when analyzing services in distributed environments, one normally use different abstractions to describe and analyse services. One of such abstractions deals with the the study of the *concurrent* nature of services. *Process calculi* are formal languages conceived for the description and analysis of concurrent systems. As such, the goal of a process calculus is to provide a rigorous framework where complex systems can be accurately analyzed, including reasoning techniques (type systems, specification logics) to verify essential properties of a system. The term *structured communications* [9] refers to the branch of process calculi devoted to the analysis of interactions between services. On a calculus for structured communications, one considers the computation within a service as an atomic activity, and focus the core of the analysis in the interactions between services.

One of the most important aspects when modelling services relate to the notion of *trustworthiness*, or the extent to which users believe that the systems behave correctly. A *safe* system is one in which a property considered harmful for the life of the system would never happen, like for instance the disclosure of the private credentials of the manager to a thief.

Despite of being such a young trend, different but interrelated views for the analysis of service oriented systems have been proposed. We can enclose such approaches in two dichotomies: *global/local* views of services, and *imperative/declarative* specifications. In the first dichotomy, either one describe the system as the exchange of messages between different participants, or one consider the system as the composition of the local behaviours of each participant. In this first view, known as *choreography* [4, 5], one consider the system as a whole, taking care only of the interfaces that participants use when interacting to the outside world. In the second view, known as *orchestration* [14, 5], one model the system as perceived by the eyes of each participant (so-called *end-point*), sending and receiving messages but not knowing which other actors are present in a communication. As recently presented, choreographies and orchestrations can be operationally correspondent, and one can either project a choreography to generate distributed orchestrations that implements it, or lift a process specification done in an orchestrated manner to describe its respective choreography [5].

As a simple example of such duality, take a simplified version of an online booking scenario: Here, the customer interacts with the airline company AC using its service *ob*, such interaction will be labelled by an identifier (referred here as a *session*). The customer and

AC can interact in more than one manner, requiring sessions to be unique and independent from each other. In this case, we will use sessions labelled k_1, k_2 to identify the direction of the communication: k_1 from Customer to the AC and k_2 for its dual. Once sessions are established, the customer will request the company about a flight offer with his booking data, along the session key k_1. The airline company will process the customer request and will send a reply back with an offer using the session key k_2. The customer will eventually accept the offer, sending back an acknowledgment to the airline company using k_1. The description of this protocol in a choreographic way will describe the sequence of interactions between Customer and AC, for instance: $Customer \rightarrow AC : ob(k_1, k_2)$ will create sessions k_1 and k_2 between Customer and AC, and $Customer \rightarrow AC : k_1\langle booking, x\rangle$ will describe the communication of the *booking* value using the session key k_1 from Customer to AC. The rest of the specification representing this protocol can be described as follows:

$$
\begin{aligned}
&Customer \rightarrow AC : ob(k_1, k_2).\\
&Customer \rightarrow AC : k_1\langle booking, x\rangle.\\
&AC \rightarrow Customer : k_2\langle offer, y\rangle.\\
&Customer \rightarrow AC : k_1\langle accept, z\rangle
\end{aligned}
$$

In an orchestrated version of the above example, one might consider the system as the concurrent execution of processes implementing the actions for Customer and AC. Here, processes will communicate via *session establishment* and *message passing*, among other actions. Following the notation from [9], the concurrent execution of **request** $ob(k)$ **in** P and **accept** $ob(k)$ **in** P will create a session k between P and Q, whereas $k![booking]; P$ in parallel with $k?(x)$ **in** Q will use a previously established session k to communicate the data contained in *booking* from P to Q. The specification of the example is coded below, using $\|$ to denote parallel composition of processes, $(\nu x)\ P$ as the creation of a new resource x local to P, and $\mathbf{0}$ the termination of a process:

$$
\begin{aligned}
Customer &= \textbf{request}\ ob(k_1, k_2)\ \textbf{in}\ (k_1![booking]; k_2?(y)\ \textbf{in}\ (k_1![accept]; \mathbf{0}))\\
AC &= \textbf{accept}\ ob(k_1, k_2)\ \textbf{in}\ (k_1?(x)\ \textbf{in}\ (\nu\ offer)k_2![offer]; k_1?(z)\ \textbf{in}\ \mathbf{0})\\
System &= Customer \parallel AC
\end{aligned}
$$

Here, the communication will be *structured* if we can provide guarantees about the use of sessions along the life of the protocol. For instance, considering the choreographic specification of the example given above, we can guarantee that the usage of sessions will require first an interaction using k_1, followed by k_2 and finalized by k_1. It is obvious that such guarantees become harder to express in architectures with thousands of services, which is the case of service oriented architectures.

The second dichotomy here considered refers to the approach used to construct the models. Descriptions can have imperative or declarative flavors: In an imperative approach, one explicitly define the control flow of commands. Typical representatives of this approach are based on process calculi, and come with behavioral equivalences and type disciplines as their main analytic tools [18, 10, 2, 9, 21]. On the contrary, in a declarative approach the

focus drifts to the specification of the set of constraints (causality relations, time constraints, quality of service) processes should fulfill in order to be considered correct [17, 20, 12, 15]. Even if these two trends address similar concerns, we find that they have evolved rather independently from each other. Returning to our example, we might consider the specifications above presented imperative specifications, whereas a declarative specification will let parts of the process unspecified. For instance, we could relax the specification given above by accepting any implementation of AC that complies with an ordering of actions where it first receives the booking data, and eventually (that is, immediately or in an unspecified sequence of interactions) returns a booking offer. Such a policy can be observed better on a logical formalization, as for instance a formula in Linear Temporal Logic [13].

2. A unifying framework for structured communications

This research has as a main objective to leverage the trustworthiness level of a system by providing a clear methodology of specification and verification of structured communications. Our goal is to give characterizations of services, both at the *operational* and *logical* level. This is done by relating the way services are specified, both from their global and local viewpoints. Figure 1 illustrates the approach for the specification and verification of structured communications. A specification of a choreography C can be projected to the parallel composition of end points P_i with an index i corresponding to each of the participants involved in choreography. Similarly, every choreographic specification in C corresponds to a formula in a modal logic representing the interactions between agents; such a correspondence is described in the figure as the bijection $\mathcal{GL}$ between C and ϕ_C. $\mathcal{GL}$ not only provides a logical characterization of a process; it also allows for *partial specification*: Given a logical formula, one can see if there is a process in C that can satisfy ϕ_C.

A similar reasoning is provided for orchestrations: Starting with i-indexed parallel composition implementing each participant P_i (denoted $\prod_i [\![P_i]\!]$), one is interested in describing the behaviour of its composition. Such description is embedded in the bijection $\mathcal{LL}$ between the orchestration in $\prod_i [\![P_i]\!]$ and its logical counterpart in $\bigcap_i [\![\chi_i]\!]$. Moreover, a formula representing the global behaviour of a choreography can be projected to a corresponding formula describing the behaviour of a set of orchestrations. Such a mapping can be observed in the diagram as the function LP from ϕ_C to $\bigcap_i [\![\chi_i]\!]$.

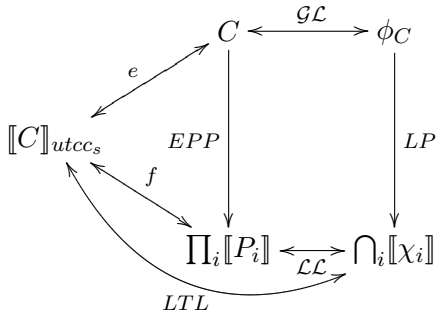


Figure 1: Methodology for the verification of structured communcations

Finally, one can observe an interesting relation when comparing languages of structured communications and other models of concurrency. CC refers to the Concurrent Constraint

family of languages [19] and its timed extensions, such as timed CC (tcc) [6] and universal tcc (utcc) [16]. We can see CC languages as part of the declarative approaches for the analysis of choreographies: First, differently from the classical approach where a value is assigned to each system variable *(store-as-valuation)*, in CC languages the store represents a *constraint* on the possible values of variables at one point in the life of the system. Second, it allows one to consider both the declarative flavour of logics and the execution of processes both in a single framework: the satisfaction of a formula allows the system to proceed, and the execution/inhibition of a process in the interaction is only defined by the amount of information available in the store. Timed extensions of the CC family refine the notion of store-as-constraint, describing the system as sequences of input-output stimuli between a set of processes and a store. These extensions give us enough modelling power to express declarative and imperative information in the same framework. The encodings between a choreography model and a timed CC specification are depicted by the function e, the corresponding mapping between a timed CC model and an orchestration model are depicted by the bijective function f; finally, the correspondence between an CC model and its logical counterpart is given by means of Linear Temporal Logic (LTL) [13].

3. Overview of Completed and Current Work

The evolution of this research project can be divided in three mayor research areas: First, we started by equipping CC languages with primitives for the analysis of structured communications, namely the treatment of mobile data and access control of information flow. A recent addition to the familiy of CC languages, known as Universal Timed CCP (`utcc`) introduces the possibility of universally quantify over predicates in the constraint store. `utcc` is presented as a candidate for representing mobility and security, both important concepts when talking about structured communications. However, the universal quantification in `utcc` is completely unrestricted. This means that in the proposed representations of link mobility and security protocols in `utcc`, every agent may guess channel names and encrypted values by universal quantification. It is thus necessary to enforce a restriction on the allowed processes to make sure that this is not possible. We proposed $\texttt{utcc}_s$, an extension of universal `tcc` with a type system for constraints used as patterns in process abstractions, which essentially allows us to distinguish between universally abstractable information and secure (non-leakable information) in predicates. We also proposed a novel notion of abstraction under local knowledge, which gives a general way to model that a process (principal) knows a key and can use it to decrypt a message encrypted with this key without revealing the key [8].

Second, we related CC and orchestration languages. We exploited `utcc` to give a declarative interpretation to the language of orchestrations at [9]. This way, services can be analyzed in a declarative framework where time is defined explicitly, and their behaviour compared to formulae in LTL. We do so by giving an operationally correspondent encoding of the language in [9] into `utcc`. Moreover, the selected language is prone to timed extensions: as we show in [11] an orchestration language can be benefitted from the inclussion of timed information on the duration of sessions, declarative preconditions within session establishment constructs, and session abortion primitives.

Finally, we filled the gap between choreographic models and logical specifications. Starting with an extension of Hennesy-Milner logic [7], we introduced $\mathcal{GL}$, a global logic for the

study of choreographies. $\mathcal{GL}$ describes properties over the transitions of a given choreography. As for structured communications, $\mathcal{GL}$ places special on the main elements of interactions in the choreography, namely the participants involved in a communication, the sessions used in an interaction and the effects on the variables by a given communication. The logic is equipped with a proof system that allows for verification of properties among participants in a choreography. With $\mathcal{GL}$, one can see the state of a choreography as a formula in the logic, and one can check for satisfaction of desirable properties by relating a logical formula wrt a choreographic specification [3].

4. Open Issues

The research being done to the moment constitutes just seminal steps on the path towards a verification framework of structured commmunications. Our main concerns relate to establishing a relation between the model of end-points and logical frameworks for the specification of sessions. In [1], Berger et al. presented proof systems characterizing May/Must testing preorders and bisimilarities over typed π -calculus processes. The connection between types and logics in such system comes in handy to restrict the shape of the processes one might be interested. In particular, being the synchronization methods in orchestration languages of similar nature as the ones present in the π calculus, one might consider such work as a suitable proof system for the calculus of end points. Our next step will focus on relating $\mathcal{GL}$ to orchestrations, both by providing a corresponding logic for the analysis of orchestration and by making the logical projection between global and local formulae. If successful, this research will contribute by providing a basis for logical specifications and model checking of structured communications. Finally, we want to continue the research on representing both global and local process views in concurrent constraint calculi, aiming at a unified representation of both views within the same formal model.

Acknowledgments

This research owes much to Thomas Hildebrandt for his indispensable guidance, and to Marco Carbone for the many insightful discussions profiling this topic of research. The research has been partially supported by the Computer Supported Mobile Adaptive Business Processes (www.CosmoBiz.org) project and the Trustworthy Pervasive Healthcare Services (www.Trustcare.dk) project, supported by the Danish Research Agency (grant no.: 274-06-0415 and grant no.: 2106-07-0019).

References

[1] M. Berger, K. Honda, and N. Yoshida. Completeness and logical full abstraction in modal logics for typed mobile processes. In L. Aceto, editor, *ICALP'08*, number 5126 in LNCS, pages 99–111. Springer-Verlag, Berlin Germany, 2008.

[2] M. Boreale, R. Bruni, L. Caires, R. De Nicola, I. Lanese, M. Loreti, F. Martins, U. Montanari, A. Ravara, and D. Sangiorgi. SCC: a service centered calculus. *Proceedings of WS-FM*, 4184:38–57, 2006.

[3] M. Carbone, T. Hildebrandt, and H. A. López. Towards a modal logic for the global calculus. In K. Honda and A. Mycroft, editors, *Programming Language Approaches to Concurrency and Communication-cEntric Software (PLACES)*, 2010.

[4] M. Carbone, K. Honda, and N. Yoshida. A calculus of global interaction based on session types. In *2nd Workshop on Developments in Computational Models (DCM)*, ENTCS, 2006.

[5] M. Carbone, K. Honda, and N. Yoshida. Structured communication-centred programming for web services. In *16th European Symposium on Programming (ESOP)*, volume 4421 of *LNCS*, pages 2–17, Braga, Portugal, March 2007. Springer, Berlin Heidelberg.

[6] F. de Boer, M. Gabbrielli, and M. Meo. A Timed Concurrent Constraint Language. *Information and Computation*, 161(1):45–83, 2000.

[7] M. Hennessy and R. Milner. On Observing Nondeterminism and Concurrency. In *Proceedings of the 7th Colloquium on Automata, Languages and Programming*, pages 299–309. Springer-Verlag London, UK, 1980.

[8] T. Hildebrandt and H. A. López. Types for Secure Pattern Matching with Local Knowledge in Universal Concurrent Constraint Programming . In *International Conference on Logic Programming (ICLP)*, volume 5649 of *Lecture Notes in Computer Science*, pages 417–431. Springer, Berlin Heidelberg, 2009.

[9] K. Honda, V. Vasconcelos, and M. Kubo. Language Primitives and Type Discipline for Structured Communication-Based Programming. In *7th European Symposium on Programming (ESOP): Programming Languages and Systems*, pages 122–138. Springer-Verlag London, UK, 1998.

[10] A. Lapadula, R. Pugliese, and F. Tiezzi. A calculus for orchestration of web services. In *Proc. of 16th European Symposium on Programming (ESOP'07)*, volume 4421 of *Lecture Notes in Computer Science*, pages 33–47. Springer, 2007.

[11] H. A. López, C. Olarte, and J. A. Pérez. Towards a Unified Framework for Declarative Structured Communications. In *Programming Language Approaches to Concurrency and Communication-cEntric Software (PLACES'2009)*, volume 17 of *EPTCS*, pages 1–15, 2010.

[12] K. M. Lyng, T. Hildebrandt, and R. R. Mukkamala. The Resultmaker Online Consultant: From Declarative Workflow Management in Practice to LTL. In *Proc. of 1st Intl. Workshop on Dynamic and Declarative Business Processes (DDBP)*, Munich, Germany, 2008.

[13] Z. Manna and A. Pnueli. *The temporal logic of reactive and concurrent systems: Specification*. Springer, 1992.

[14] J. Misra and W. R. Cook. Computation orchestration: A basis for wide-area computing. *Journal of Software and Systems Modeling*, May 2006.

[15] A. K. Nørgaard, L. Pedersen, and P. Strøiman. Method for generating a workflow on a computer, and a computer system adapted for performing the method. Patent, 05 2005. US 6895573.

[16] C. A. Olarte and F. D. Valencia. Universal concurrent constraint programming: Symbolic semantics and applications to security. In *23rd Annual ACM Symposium on Applied Computing (SAC)*, 2008.

[17] M. Pesic and W. van der Aalst. A Declarative Approach for Flexible Business Processes Management. *Lecture Notes in Computer Science*, 4103:169, 2006.

[18] F. Puhlmann and M. Weske. Using the Pi-Calculus for Formalizing Workflow Patterns. *BPM*, 3649:153–168, 2005.

[19] V. Saraswat. *Concurrent Constraint Programming*. MIT Press, 1993.

[20] W. van der Aalst and M. Pesic. DecSerFlow: Towards a Truly Declarative Service Flow Language. *Lecture Notes in Computer Science*, 4184:1, 2006.

[21] H. Vieira, L. Caires, and J. Seco. The Conversation Calculus: A Model of Service-Oriented Computation. In *Programming languages and systems: 17th European Symposium on Programming (ESOP)*, page 269, Budapest, Hungary, 2008. Springer-Verlag New York.

Technical Communications of the International Conference on Logic Programming, 2010 (Edinburgh), pp. 277–280
http://www.floc-conference.org/ICLP-home.html

DESIGN AND IMPLEMENTATION OF A CONCURRENT LOGIC PROGRAMMING LANGUAGE WITH LINEAR LOGIC CONSTRAINTS

THIERRY MARTINEZ

EPI Contraintes, INRIA Paris-Rocquencourt, BP105, 78153 Le Chesnay Cedex, France

1. Introduction

Prolog is originally rooted in logic with the elegant mapping: "programs = formulas, execution = proof search". Constraint logic programming extends Prolog to program in a richer structure than mere Herbrand terms. The underlying constraint solving engine requires either to be built-in or to add coroutines mechanisms that are not in the scope of logical reading. Implementations add other non-logical features, like assert/retract, and mutable variables, to mimic imperative programming style.

The concurrent constraint programming language enjoys logical semantics and is expressive enough to describe constraint propagators [Sar93]. Agents tell constraints as messages and are synchronised by asking whether the messages entail some constraints. The execution pursues once the guard is entailed. The suspension is therefore a transient state, captured by using linear-logic implication $\multimap$ [Fag01]. Furthermore, reading constraints as resources in linear logic allows semantics to capture the non-monotonous traits of imperative programming like mutability. LCC enjoys the mapping "programs = linear-logic formulas, execution = logical deduction": observables are the logical consequences of a program, by opposition to the logical resolution in the Prolog settings.

My thesis aims at designing a practical language as close as possible to the linear concurrent constraint (LCC) theory. The main contribution is a new operational semantics which behaves as an angelic scheduler with a tractable algorithmic complexity. This operational semantics is sound and complete with respect to the logical semantics and allows the construction of a rich language over a very simple kernel.

The second section presents the kernel, the third describes the operational semantics and the last section describes the state of the implementation work and its perspectives.

2. Kernel syntax and logical semantics

The four rules of the syntax are given below with their reading in linear logic. We recall that the ! modality introduces unlimited resources.

$$\begin{array}{lll}
[\![\texttt{forall}\ x_1 \dots x_n (x\texttt{.}p\texttt{(}x_1\texttt{,}\dots\texttt{,}x_n\texttt{)} \Rightarrow a)]\!] & = !\forall x_1 \dots x_n (p(x, x_1, \dots, x_n) \multimap [\![a]\!]) & \textit{(ask)} \\
[\![\texttt{exists}\ x\texttt{(}a\texttt{)}]\!] & = \exists x([\![a]\!]) & \textit{(hiding)} \\
[\![x\texttt{.}p\texttt{(}x_1\texttt{,}\dots\texttt{,}x_n\texttt{)}]\!] & = p(x, x_1, \dots, x_n) & \textit{(tell)} \\
[\![a\ a']\!] & = [\![a]\!] \otimes [\![a']\!] & \textit{(parallel composition)}
\end{array}$$

Technical Communications of the 26th International Conference on Logic Programming, Edinburgh, July, 2010
Editors: Manuel Hermenegildo, Torsten Schaub
LIPIcs - Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/LIPIcs.ICLP.2010.277

This syntax is a subset of the syntax of modular LCC agents [Hae07]: constraints are restricted to be single *linear tokens* (*i.e.*, linear-logic predicates without any non-logical axiom) and all asks are persistent (interpreted with the ! modality). The variable which precedes the dot in linear tokens is called the module variable. It is worth noticing that the kernel restricts all the arguments of the constraints guarding asks to be universally quantified. On the opposite, the module variable is never universally quantified.

I proved that modular LCC is as expressive as CHR as far as logical semantics and original operational semantics are concerned [Mar10]. However, CHR implementations trade completeness for committed-choice. Several refinements for controlling the scheduler in CHR have been proposed, taking into account the order of the rules in the program [Duc03], priority annotations [Kon07] or probabilities [Frü02]. None of these refinements are captured by logical semantics. The next section proposes a tractable operational semantics which is correct and complete with respect to the linear-logic reading and such that this kernel is as expressive as the whole modular LCC language.

3. Angelic operational semantics

The observable of interest is the set of all the constraints which are logical consequences of the program. This observable raises naturally in the correctness theorem of the traditional operational semantics [Fag01]. Operationally, it is the set of entailed constraints, leading to observable side-effects.

I propose the concept of derivation nets which generalises Palamidessi's SOS semantics [Bes97] for reducing scheduling non-determinism: derivations are represented as a potentially infinite multihypergraph where vertices are agents and edges are derivation steps. The derivations of the traditional LCC operational semantics correspond to interpretation as Petri-net. Strategies for reducing non-determinism are expressible as vertex sharing. There exist sharings, like the sharing of all ask instances, which allow each edge to be checked in polynomial time. Other tractable sharings should be investigated.

The angelic semantics contrasts with the usual committed-choice execution model. Concurrent languages differ by the expressive power of their guards: single channel matching for π-calculus, multiple-head matching for Join-calculus, multiple-head Prolog checking for CHR. Our CHRat [Fag08] proposition for generalising guards relies on extra-logical CHR propagations for consuming heads only once the entailment is checked. The angelic semantics overcomes this difficulty and allows the kernel to be restricted to the simplest form of guards while providing the most general expressive power: the two linear-logic formulas $\forall\vec{x}(c \otimes c' \multimap a)$ and $\forall\vec{x}(c \multimap (c' \multimap a))$ have the same constraints for consequences since the non-consumption of c' cannot be observed, and computation can be triggered after the consumption of c to check that c' is entailed.

Since the kernel forces asks to universally quantify over all the arguments appearing in guards, LCC agents of the form $x.p(v) \Rightarrow A$ (for any variables x and v and sub-agent A) should be translated in the kernel syntax. A natural translation is `forall` $v'(x.p(v') \Rightarrow$ `exists` $k(eq.\text{check}(v, v', k)\ (k.\text{true}() \Rightarrow A)))$ where the token $eq.\text{check}(v, v', k)$ refers to an agent implementing value comparison (that we suppose defined in the standard library). Since there is no observable side-effects between the consumption of $x.p(v')$ and the execution of A, angelic semantics ensures that consumptions of $x.p(v')$ for $v \neq v'$ are not observed. Therefore the proposed translation preserves the semantics. However, trying to consume all

the tokens $x.p(v')$ is inefficient compared to usual argument indexing mechanisms present in CHR implementations for example.

An illustration of the expressive power of angelism is that the indexing of linear tokens with respect to their arguments is user-implementable, as soon as tokens are indexed with respect to their module variable (which is a weaker hypothesis for the implementation since the module variable is distinguished and is never universally quantified). Suppose an agent M implementing maps (*e.g.* with hash-tables, AVL or other custom structures) associating a fresh variable x_v to each value v: for a map m, the agent M is supposed to react to the tokens $m.\text{get}(v, x)$ by unifying x with x_v. Indexing tokens $x.p(v)$ with respect to v in the map m is performed by the agent **forall** $v(x.p(v) \Rightarrow$ **exists** $x_v(m.\text{get}(v, x_v)\ x_v.\text{p}()))$. Then, agents of the form **exists** $x_v(m.\text{get}(v, x_v)\ (x_v.p() \Rightarrow A))$ have the same logical consequences as $x.p(v) \Rightarrow A$: consuming $x_v.p()$ supposes that $x.p(v)$ has been consumed, while $x.p(v)$ can still be consumed by other asks, preventing $x_v.p()$ to appear in the store.

4. The SiLCC project and perspectives

The implementation aims to build a compiler for the kernel and to reconstruct the full LCC language on top of it. A prototype has been implemented with a library for full LCC over Herbrand domain [1]. Transient asks (without the ! modality) are encoded with token consumption and complex guards are decomposed to elementary asks. *E.g.*, the ask **forall** $x(m.\text{p}(x)\ m.\text{q}(x) \to a)$ is compiled to the following kernel agent:

exists $t(t.\text{transient}()$
forall $x(m.\text{p}(x) \Rightarrow$
forall $y(m.\text{q}(y) \Rightarrow$
exists $k(eq.\text{check}(x, y, k)$
$(k.\text{true}() \Rightarrow t.\text{transient}() \Rightarrow a)))))$

where the token $t.\text{transient}()$ translates the non-persistency of the original ask. More evolved syntactic sugars have been implemented for sequentiality, conditionals, pattern-matching on Herbrand terms and records, *etc.*, so that usual programming idioms can be expressed easily on top of the mono-paradigm simple kernel.

LCC can express sequentiality and non-monotonous traits for imperative programming, closures and modules. Asks allow LCC agents to wait for some logical consequence, therefore LCC enjoys a reflexive mechanism allowing LCC agents to observe (the consequences of) their own accessible stores, since these stores are proved to be equal to the set of logical consequences by the correctness and completeness theorem of the operational semantics. However, the canonical encoding of constraint propagators as LCC agents have terminal stores for observable of interest: terminal stores cannot be reflectively observed by any agent, for the mere fact that they are terminal. We should investigate more involved encoding of constraint propagators such that relevant observables would be the accessible stores. Moreover, constraint programming involves search tree exploration: how to express search is still open. Our work on formalising search strategies as pattern-matching [Mar09] initiates a better understanding of the distinction between search trees and search heuristics in the settings of a modeling language. We still have to investigate how to encode trees and heuristics in LCC, possibly with similar control mechanism encoding as for sequentiality.

[1] The compiler together with a reasonable documentation and examples are available for download: `http://contraintes.inria.fr/~tmartine/silcc`

References

[Bes97] E. Best, F.S. de Boer, and C. Palamidessi. Partial order and SOS semantics for linear constraint programs. In *Proceedings of Coordination*, *Lecture Notes in Computer Science*, vol. 1282, pp. 256–273. Springer-Verlag, 1997.

[Duc03] Gregory J. Duck, Peter J. Stuckey, Marìa Garcìa de la Banda, and Christian Holzbaur. Extending arbitrary solvers with constraint handling rules. In *Proceedings of PPDP'03, International Conference on Principles and Practice of Declarative Programming, Uppsala, Sweden*, pp. 79–90. ACM Press, 2003.

[Fag01] François Fages, Paul Ruet, and Sylvain Soliman. Linear concurrent constraint programming: operational and phase semantics. *Information and Computation*, 165(1):14–41, 2001. doi:10.1006/inco.2000.3002.

[Fag08] François Fages, Cleyton Mario de Oliveira Rodrigues, and Thierry Martinez. Modular CHR with ask and tell. In Thom Frühwirth and Tom Schrijvers (eds.), *Proceedings of the fifth Constraint Handling Rules Workshop CHR'08*. 2008.

[Frü02] Thom Frühwirth, Alessandra Di Pierro, and Herbert Wiklicky. Probabilistic constraint handling rules. In *WFLP 2002, 11th International Workshop on Functional and (Constraint) Logic Programming, Selected Papers*, vol. 76, pp. 115–130. 2002. doi:DOI:10.1016/S1571-0661(04)80789-8.

[Hae07] Rémy Haemmerlé, François Fages, and Sylvain Soliman. Closures and modules within linear logic concurrent constraint programming. In V. Arvind and Sanjiva Prasad (eds.), *Proceedings of FSTTCS 2007, IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science*, *Lecture Notes in Computer Science*, vol. 4855, pp. 544–556. Springer-Verlag, 2007. doi:10.1007/978-3-540-77050-3_45.

[Kon07] Leslie De Koninck, Tom Schrijvers, and Bart Demoen. User-definable rule priorities for CHR. In *Proceedings of PPDP'07, International Conference on Principles and Practice of Declarative Programming, Wroclaw, Poland*, pp. 25–36. ACM Press, 2007.

[Mar09] Julien Martin, Thierry Martinez, and François Fages. On the specification of search tree heuristics by pattern-matching in a rule-based modelling language. In *Proceedings of the Eighth International Workshop on Constraint Modelling and Reformulation, associated to CP'09*, pp. 73–86. 2009.

[Mar10] Thierry Martinez. Semantics-preserving translations between linear concurrent constraint programming and constraint handling rules. In *Proceedings of PPDP'10, International Conference on Principles and Practice of Declarative Programming, Edinburgh, UK (to appear)*. 2010.

[Sar93] Vijay A. Saraswat. *Concurrent constraint programming*. ACM Doctoral Dissertation Awards. MIT Press, 1993.

Technical Communications of the International Conference on Logic Programming, 2010 (Edinburgh), pp. 281–285
http://www.floc-conference.org/ICLP-home.html

HIGHER-ORDER LOGIC LEARNING AND λPROGOL

NIELS PAHLAVI [1]

[1] Department of Computing, Imperial College London
180 Queen's Gate, London, United Kingdom
E-mail address: niels.pahlavi@imperial.ac.uk
URL: http://www.doc.ic.ac.uk/~namdp05/

Abstract. We present our research produced about Higher-order Logic Learning (HOLL), which consists of adapting First-order Logic Learning (FOLL), like Inductive Logic Programming (ILP), within a Higher-order Logic (HOL) context. We describe a first working implementation of λProgol, a HOLL system adapting the ILP system Progol and the HOL formalism λProlog. We compare λProgol and Progol on the learning of recursive theories showing that HOLL can, in these cases, outperform FOLL.

Introduction, Problem Description and Background

Much of logic-based Machine Learning research is based on First-order Logic (FOL) and Prolog, including Inductive Logic Programming (ILP). As such, learning higher-order theories is not possible for such a system, and even some first-order tasks are not handled well, like "learning first-order recursive theories" which "is a difficult learning task" in a normal ILP setting [Mal03]. Yet, [Far08] describes HOL as "a natural extension of first-order logic (FOL) which is simple, elegant, highly expressive, and practical" and recommends its use as an "attractive alternative to first-order logic". HOL, which allows for quantification over predicates and functions, is intrinsically more expressive than FOL, would give sounder logical foundations, and "has generally been under-exploited" [Llo03] in logic-based Machine Learning. According to [Llo03], "the logic programming community needs to make greater use of the power of higher-order features and the related type systems and the use of HOL in Computational Logic is illustrated: functional languages, like Haskell98; Higher-order programming introduced with λProlog [Mil98]; integrated functional logic programming languages like Curry or Escher; or the higher-order logic interactive theorem proving environment "HOL". It is also used in IBAL and for Deep Transfer Learning.

The use of HOL in ILP would allow to consider the learning of higher-order predicates; but it would also make the learning of first-order learning theories sounder, more natural and more intuitive through the use of higher-order predicates in background knowledge. More generally, the expressivity of HOL would make it possible to represent mathematical properties like symmetry, reflexivity or transitivity, which would allow to handle equational reasoning and functions within a logic-based framework. We could also represent such properties in the following fashion (in the case of symmetry) : R@X@Y ⇐ [sym@R,R@Y@X], and,

Key words and phrases: Inductive Logic Programming, Progol, Higher-order Logic, Higher-order Logic Learning, λProlog.

Technical Communications of the 26th International Conference on Logic Programming, Edinburgh, July, 2010
Editors: Manuel Hermenegildo, Torsten Schaub
LIPIcs - Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/LIPIcs.ICLP.2010.281

abduce for example that the move of the bishop in chess is symmetric: sym@bishop_move. About the use of probability in a logic-based setting, [Ng08] advocates for probability to be captured directly in the theory itself, which can be done naturally and directly with HOL, as opposed to almost all approaches having a clear separation between the logical statements and the probabilities.

λProlog [Mil98] is a higher-order logic programming language handling scoping over names and procedures, the use of lambda terms as data structures and higher-order programming. It is based on Higher-order Horn Clauses (HOHC) (introduced in [Nad90] where a theorem proving procedure for HOHC based on Huet's unification algorithm in typed λ-calculus [Hue75] is also outlined), which are "a generalization of Horn clauses to a higher-order logic" obtained "by supplanting first-order terms with the terms of a typed λ-calculus and by permitting quantification over function and predicate symbols".

1. Goal of the Research and Overview of the existing Literature

The goal of my PhD research is, therefore, to develop Higher-order Logic Learning (HOLL), which consists of generalizing logic-based Machine Learning, and particularly ILP, from the first-order to the higher-order context. We have already made a first working implementation of λProgol, a HOLL system adapting the ILP system Progol and the HOL formalism λProlog. We decided to choose Higher-order Horn Clauses (HOHC) [Nad90] as a HOL formalism, since it is one of the logical foundations of λProlog. As a ILP system, we chose to adapt Progol [Mug95], which is a popular and efficient implementation.

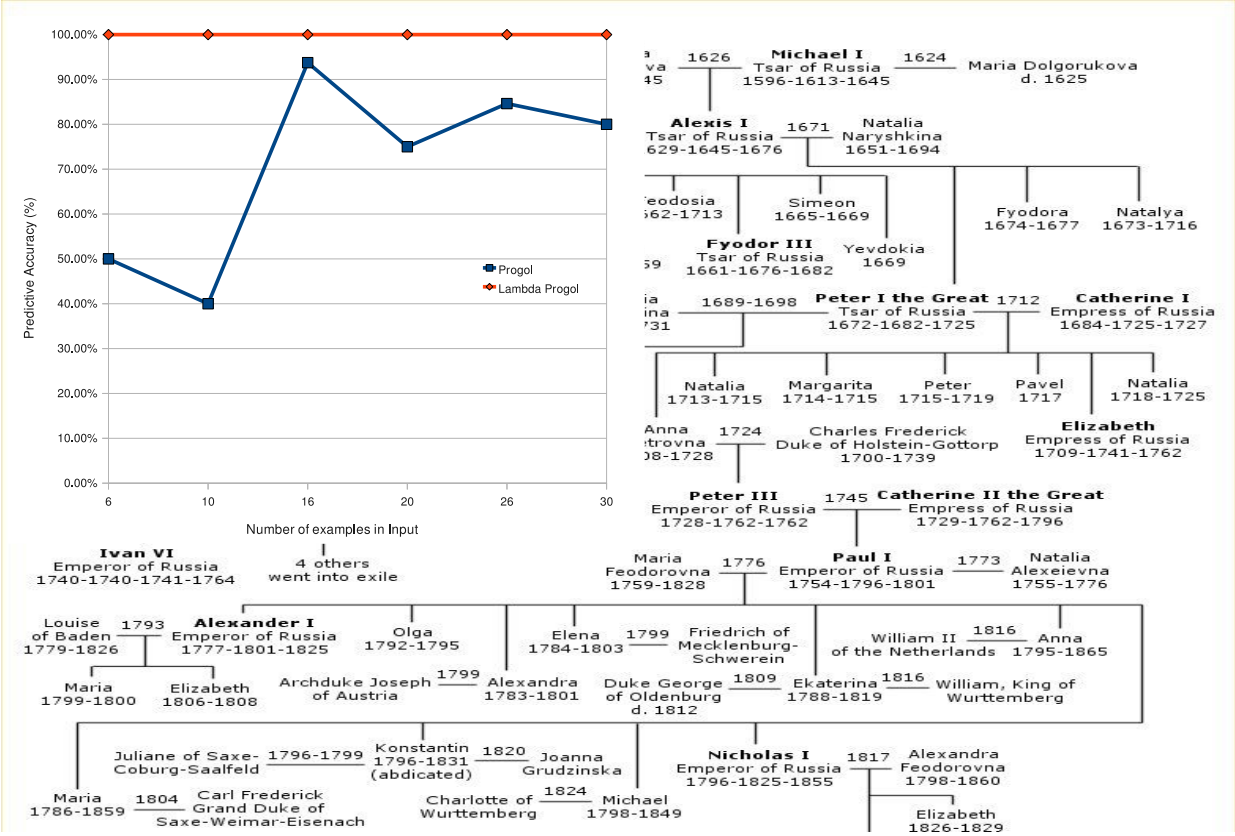
We also want to determine whether HOLL can outperform First-order Logic Learning (FOLL), assessing how the power of expressivity of HOL can be used to improve significantly the learning capacity and efficiency of FOLL, and study the trade-off that there may be between learnability and searching costs (the use of Henkin semantics as in [Wol94], seems to alleviate these and maintain the structure of the search space). ILP seems to be rather intuitively adaptable to a HOL formalism and we aim at developing a theory of HOLL as well.

There have been attempts to use HOL for logic-based Machine Learning such as by Harao starting in [Har90], Feng and Muggleton, and Furukawa and Goebel [Fur96]. They provide different higher-order extensions of least general generalization to handle higher-order terms in a normal ILP setting, whereas we use λProlog, a HOL framework, as a logical foundation to extend first-order ILP to a higher-order context. The main similar work is [Llo03] by Lloyd and Ng, where higher-order machine learning is also developed. It details a learning system, called $ALKEMY$. A main difference is that Lloyd's approach is not based on Logic Programming and therefore on ILP. According to Flach, "it is almost a rational reconstruction of what ILP could have been, had it used Escher-style HOL rather than Prolog"; whereas we intend, through the use of higher-order Horn clauses to keep the Horn clauses foundations of LP and ILP and to extend it.

2. Current Status of the Research and Preliminary Results

λProgol, a λProlog adaptation of the popular and efficient ILP system Progol was introduced in [Pah09a] and [Pah09b] along with its algorithm adapting closely and intuitively Progol and Mode-Directed Inverse Entailment. A first working implementation of λProgol has since been made, tested, is described in [Pah10] and is available at [Pah]. Our first

Figure 1: Left: Comparison between Progol and λProgol on the Ancestor example. Right: Part (around one third) of the Romanov dynasty tree used in the experiments



choice of implementation was based on λProlog but revealed to be too inconvenient and inefficient to use; instead the current implementation is in Prolog, which is more convenient and more efficient; a requirement is the use of a λProlog interpreter, which was implemented using a depth-first approach.

Initial promising results have been obtained so far about learning recursive theories. In order to stress the difference in the learnability of a given problem between HOLL and FOLL, and to ensure fairness and soundness, standard λProgol was compared against standard Progol, whose algorithms are almost the same.

One of the results (used in [Mal03]) consists of learning the predicate *ancestor* given a genealogical tree defined by facts for the predicates *parent* and *married* as background knowledge and positive and negative examples of the predicate *ancestor*; for λProgol, the higher-order predicate *trans*, which "given a predicate of two arguments, constructs its transitive closure" is added to the background knowledge. The genealogical tree used for this experiment is described in Fig.1 and contains 119 members over 11 relations of the Romanov Russian dynasty.

To compare the two systems, we created files containing positive and negative examples of *ancestor*. These files contain an equal number of positive and negative examples generated randomly. We then compared the respective predicative accuracy of Progol and

λProgol on these examples by doing a leave-one-out cross-validation. The results of this experiment are shown in Fig.1.

For Progol, which has to learn the definition recursively, the larger the input and the smaller the number of examples, the smaller the probability to learn the definition correctly. Hence the difficulty to learn and the observation that the accuracy seems to decrease with the number of examples. On the other hand, λProgol learns the correct definition in all the cases, which is ancestor@X@Y ⇐ [trans@parent@X@Y]. This definition is non recursive and can be learned from any given positive example. On this example consisting of learning a recursive definition from large data with few examples, we have showed that HOLL can outperform FOLL. This result along with similar others are available at [Pah].

3. Expected Achievements and Open Issues

We intend to continue the tests and comparisons of λProgol against already existing ILP systems to determine how HOLL may outperform FOLL as it was shown above. We aim to present theoretical results for HOLL. ILP theory seems to be rather intuitively adaptable within a HOL framework. For λProgol, we will have to prove that higher-order inverse entailment is possible and to generalize correctness and complexity results for the Progol Bottom Clause and Search algorithms. In [Wol94], a model-theoretic semantics for HOHC is provided. We also want to investigate tasks and discoveries not learnable by first-order ILP. It could be of interest to look at HOL theorem provers, or integrated functional logic programming languages and Mathematical Discovery. Further objectives may be to investigate abduction within λProgol, active learning, introduce Probability and adapt Probabilistic Logic Learning (that I have studied during my MSc, [Mug06b] and [Mug06a]) within HOL, look at applications such as Bioinformatics, where ILP has been successfully applied, and consider other logics within λProlog.

Acknowledgement

I mostly wish to thank my supervisor Stephen Muggleton and Imperial College London for allowing me to pursue my PhD in great conditions. I am also grateful to NICTA for having given me the opportunity to visit their Canberra Research Laboratory for a month and thoroughly interact and discuss my work with Kee Siong Ng, John Lloyd and Scott Sanner.

References

[Far08] W. Farmer. The seven virtues of simple type theory. *J. Applied Logic*, 2008.

[Fur96] K. Furukawa, M. Imai, and R. Goebel. Hyper least general generalization and its application to higher-order concept learning. Tech. report, Keio University, 1996.

[Har90] M. Harao. Analogical reasoning based on higher-order unification. In *ALT*. 1990.

[Hue75] G. Huet. A unification algorithm for typed λcalculus. *Theor. Comp. Sci.*, 1975.

[Llo03] J.W. Lloyd. *Logic for Learning*. Springer, Berlin, 2003.

[Mal03] D. Malerba. Learning recursive theories in the normal ILP setting. *Fundam. Inform.*, 57(1):39–77, 2003.

[Mil98] Dale Miller. *λProlog: An Introduction to the Language and its Logic*. 1998.

[Mug95] S.H. Muggleton. Inverse entailment and Progol. *New Generation Computing*, 13:245–286, 1995. URL `http://www.doc.ic.ac.uk/\~shm/Papers/InvEnt.pdf`

[Mug06a] S.H. Muggleton and N. Pahlavi. The Complexity of Translating BLPs to RMMs. In *Proceedings of the 16th International Conference on Inductive Logic Programming*. Springer-Verlag, 2006.

[Mug06b] S.H. Muggleton and N. Pahlavi. Stochastic Logic Programs: A Tutorial. In L. Getoor and B. Taskar (eds.), *Statistical Relational Learning*. MIT Press, 2006.

[Nad90] G. Nadathur and D. Miller. Higher-order Horn Clauses. *Journal of the ACM*, 1990.

[Ng08] K. S. Ng, J. W. Lloyd, and W. T. B. Uther. Probabilistic modelling, inference and learning using logical theories. *Ann. Math. Artif. Intell.*, 54(1-3):159–205, 2008.

[Pah] N. Pahlavi. λProgol Homepage. http://www.doc.ic.ac.uk/~namdp05/.

[Pah09a] N. Pahlavi and S. Muggleton. Higher-order Logic Learning. 19th International Conference on Inductive Logic Programming. 2009. (Poster).

[Pah09b] N. Pahlavi and S. Muggleton. Higher-order Logic Learning and λProgol. IJCAI09 Workshop on Abductive and Inductive Knowledge Development. 2009.

[Pah10] N. Pahlavi and S. Muggleton. Can HOLL outperform FOLL? In *Proceedings of the 20th International Conference on Inductive Logic Programming*. 2010. To Appear.

[Wol94] D. A. Wolfram. A semantics for λProlog. *Theor. Comp. Sci.*, pp. 277–289, 1994.

Technical Communications of the International Conference on Logic Programming, 2010 (Edinburgh), pp. 286–288
http://www.floc-conference.org/ICLP-home.html

LOCAL BRANCHING IN A CONSTRAINT PROGRAMMING FRAMEWORK

FABIO PARISINI

DEIS, University of Bologna,
V.le Risorgimento 2, 40136, Bologna, Italy
E-mail address: fabio.parisini@unibo.it

Local branching is a general purpose heuristic method which searches locally around the best known solution by employing tree search. It has been successfully used in Mixed Integer Programming (MIP) where local branching constraints are used to model the neighborhood of an incumbent solution and improve the bound.

The neighborhoods are obtained by linear inequalities in the MIP model so that MIP searches for the optimal solution within the Hamming distance of the incumbent solution. The linear constraints representing the neighborhood of incumbent solutions are called local branching constraints and are involved in the computation of the problem bound. Local branching is a general framework to effectively explore solution subspaces, making use of the state-of-the-art MIP solvers.

The local branching framework is not specific to MIP: it can be integrated in any tree search strategy. In our research work we propose the integration of the local branching framework in Constraint Programming (CP).

The local branching technique, as introduced in [Fis03], is a complete search method designed for providing solutions of better and better quality in the early stages of search by systematically defining and exploring large neighborhoods. On the other hand, the idea has been used mainly in an incomplete manner since [Fis03]: linear constraints defining large neighborhoods are iteratively added and the neighborhoods are explored, generally in a non-exhaustive way. When this is done within a local search method, the overall algorithm follows the spirit of both *large neighborhood search* [Sha98] and *variable neighborhood search* [Mla97]. The main peculiarity of local branching is that the neighborhoods and their exploration are general purpose.

Integration of local search and CP aided tree search has been long advocated in the literature. See for instance Chapter 9 in [Mil03] and more recently the use of propagation within a large neighborhood search algorithm [Per04], the use of local search to speed up complete search [Sel06] and Beck's *solution-guided multi-point constructive search* [Bec07]. The technique which resembles most to our work is the latter which makes use of the existing solutions to guide the search.

We argue that integrating local branching in CP merges the advantages of the intensification and diversification mechanisms specific to local search methods, with constraint propagation that speeds up the neighborhood exploration by removing infeasible variable value assignments.

Key words and phrases: Local Branching, LDS, Local Search, Tree Search, Constraint Programming.

Technical Communications of the 26th International Conference on Logic Programming, Edinburgh, July, 2010
Editors: Manuel Hermenegildo, Torsten Schaub
LIPIcs - Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/LIPIcs.ICLP.2010.286

Integrating local branching in CP is not simply a matter of implementation but instead requires significant extensions to the original search strategy. The main extensions to the traditional MIP approach we have developed follow:

- First, using a linear programming solver for computing the bound of each neighborhood is not computationally affordable in CP. We have therefore studied a lighter way to compute the bound of the neighborhood which is efficient, effective and incremental, using the additive bounding technique.
- Second, we developed a cost-based filtering algorithm for the local branching constraint by extracting reduced-costs out of additive bounding.
- Third, we have studied a CP-tailored diversification technique that can push the search arbitrarily far from the current incumbent solution. This technique allows us to explore large sections of a big diversification space, using CP-specific modeling elements.

All these aspects have been thoroughly tested on a set of instances of the Asymmetric Traveling Salesman Problem with Time Windows [Asc95], having as first term of comparison a pure CP approach, using the same model and cost based filtering as our CP local branching implementation. Our experimental results demonstrate the practical value of integrating local branching in CP. The results can be summarized as follows: (i) on small-size instances where pure CP proves optimality, we find the optimal solution in a shorter time and prove optimality quicker, (ii) on medium-size instances, we can prove optimality where CP fails, (iii) on large-size instances, where both methods fail to prove optimality, we obtain a better solution quality within the same time limit. Moreover, we obtain even better results when compared with Limited Discrepancy Search (pure and enriched with bound computation) and with pure local search.

The first results are encouraging, but much research work still has to be done. In particular, the diversification technique we have developed for escaping the local minima of Local Branching is very promising and its study is just at an initial phase. This technique works on the best solution found during Local Branching iterations, named as reference solution, and performs a diversification search by explicitely setting difference constraints on a subset of the problem variables. For example, if we have the reference solution $\bar{x}$ having values $\bar{x_1} = 4, \bar{x_2} = 2, \bar{x_3} = 5, \bar{x_4} = 1, \bar{x_5} = 2$ we can perform diversification by simply setting constraints like $x_2 \neq 2, x_3 \neq 5, x_5 \neq 2$ and executing a normal CP search using the reduced variables domains.

This diversification approach has a very strong potentiality for CP Based Local Branching, as the capability of effectively setting difference constraints is something natural in Constraint Programming framework, while it is not in MIP; using this kind of diversification approach exploits CP specific peculiarities, still mantaining the Local Branching framework completely general. We are also working on the definition of problem independent criteria to help the selection process of the variables to set difference constraints for (currently the best results are given by random choices); tests needs to be done to understand how many constraints is better to set on different kinds of problem instances.

Moreover a similar approach can be used for intensification processes, by using a reference solution $\bar{x}$ as guide and setting equality constraints of the kind $x_i = \bar{x_i}$ on a subset of variables to explore the neighborhood of $\bar{x}$; we can easily explore portions of the search space which are quite far in terms of discrepancy from the reference solution $\bar{x}$, reaching discrepancy values that we could never reach with Limited Discrepancy Search (LDS) [Har95].

In practice, when dealing with a problem instance modeled by n domain variables, we can explore portions of the search space up to a discrepancy value of k by setting $n-k$ equality constraints and performing a normal CP search on the remaining k variables, having a much reduced search space.

It is immediate to observe that both the intensification and the diversification techniques that we are studying are based on strong randomization elements, i.e. the choice of the variables to set constraints for; this is absolutely normal in a setting where we give up on performing a complete search to obtain the optimal solution because of the extreme complexity of the problem instances.

The intensification and diversification techniques that we have briefly outlined above must be carefully studied and tested over real problem instances. Many parameters have to be tuned, like the number of equality or difference constraints to set, the relation of this number with the problem size, whether it is opportune to set both equality and difference constraints together, but above all if there is an effective and general way to select the best variables to set constraints for. This kind of work is very interesting as intensification and diversification techniques are basic elements of many search techniques, first of all in the neighborhood and diversification searches within Local Branching, but in general any time we have to perform a tree search in CP.

References

[Asc95] N. Ascheuer. *Hamiltonian path problems in the on-line optimization of flexible manufacturing systems.* Ph.D. thesis, Technische Universität Berlin, 1995.

[Bec07] J. C. Beck. Solution-guided multi-point constructive search for job shop scheduling. *J. Artif. Intell. Res. (JAIR)*, 29:49–77, 2007.

[Fis03] M. Fischetti and A. Lodi. Local branching. *Mathematical Programming*, 98:23–47, 2003.

[Har95] W. Harvey and M.L. Ginsberg. Limited discrepancy search. In *Proc. of IJCAI-95*, pp. 607–615. Morgan Kaufmann, 1995.

[Mil03] M. Milano. *Constraint and Integer Programming: Toward a Unified Methodology.* Kluwer Academic Publishers, 2003.

[Mla97] N. Mladenovic and P. Hansen. Variable neighbourhood search. *Computers and Operations Research*, 24:1097–1100, 1997.

[Per04] L. Perron, P. Shaw, and V. Furnon. Propagation guided large neighbourhood search. *Proc. of CP-04, LNCS*, 3258:468–481, 2004.

[Sel06] M. Sellmann and C. Ansotegui. Disco - novo - gogo: integrating local search and complete search with restarts. In *Proc. of AAAI-06*, pp. 1051–1056. AAAI Press, 2006.

[Sha98] P. Shaw. Using constraint programming and local search methods to solve vehicle routing problems. *Proc. of CP-98, LNCS*, 1520:417–431, 1998.

Technical Communications of the International Conference on Logic Programming, 2010 (Edinburgh), pp. 289–293
http://www.floc-conference.org/ICLP-home.html

LOGIC PROGRAMMING FOUNDATIONS OF CYBER-PHYSICAL SYSTEMS

NEDA SAEEDLOEI [1]

[1] Department of Computer Science
University of Texas at Dallas
Richardson, TX 75080, USA
E-mail address: neda.saeedloei@student.utdallas.edu

Abstract. Cyber-physical systems (CPS) are becoming ubiquitous. Almost every device today has a controller that reads inputs through sensors, does some processing and then performs actions through actuators. These controllers are discrete digital systems whose inputs are continuous physical quantities and whose outputs control physical (analog) devices. Thus, CPS involve both digital and analog data. In addition, CPS are assumed to run forever, and many CPS may run concurrently with each other. we will develop techniques for faithfully and elegantly modeling CPS. Our approach is based on using *constraint logic programming over reals, co-induction*, and *coroutining*.

1. Introduction and Problem Description

Cyber-physical systems (CPS) are becoming ubiquitous. Almost every device today has a controller that reads inputs through sensors, does some processing and then performs actions through actuators. Examples include controller systems in cars (Anti-lock Brake System, Cruise Controllers, Collision Avoidance, etc.), automated manufacturing, smart homes, robots, etc. These controllers are discrete digital systems whose inputs are continuous physical quantities (e.g., time, distance, acceleration, temperature, etc.) and whose outputs control physical (analog) devices. Thus, CPS involve both digital and analog data. In addition, CPS are assumed to run forever, and many CPS may run concurrently with each other [Lee08, Gup06].

CPS have the following four characteristics [Lee08, Gup06]: (i) they perform discrete computations, (ii) they deal with continuous quantities, (iii) they are concurrent, and (iv) they run forever. Due to fundamentally discrete nature of computation, researchers have had difficulty dealing with continuous quantities in computations (typical approaches discretize continuous quantities, e.g., time). Likewise, modeling of perpetual computations is not well understood (only recently, techniques such as co-induction [Sim07, Gup07] have been introduced to formally model rational, infinite computations). Concurrency is reasonably well understood, but when combined with continuous quantities and with perpetual computations, CPS become extremely hard to model faithfully. In this research work we will develop techniques for faithfully and elegantly modeling CPS for which no good formalisms exist within computer science.

Key words and phrases: Cyber-Physical Systems, Constraint Logic Programming over reals, Co-induction, Coroutining.

Technical Communications of the 26th International Conference on Logic Programming, Edinburgh, July, 2010
Editors: Manuel Hermenegildo, Torsten Schaub
LIPIcs - Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/LIPIcs.ICLP.2010.289

2. Background and Overview of the Existing Literature

CPS are highly complex systems for which today's computing and networking technologies do not provide adequate foundations. In fact, Edward Lee states [Lee08]:

> Cyber-physical systems are integrations of computation with physical processes. Embedded computers and networks monitor and control the physical processes, usually with feedback loops where physical processes affect computation and vice versa. In the physical world, the passage of time is inexorable and concurrency is intrinsic. Neither of these properties is present in today's computing and networking abstractions.

Lee goes on to argue that "the mismatch between these abstractions and properties of physical processes impede technical progress." Thus according to Lee, a major challenge is to find the right abstractions for CPS. Similarly, Rajesh Gupta [Gup06] urges researchers "to achieve the goal of semantic support for location and time at all levels," and address the following technical problems for CPS:

(1) "How do we capture location (and timing) information into CPS models that allows for validation of the logical properties of a program against the constraints imposed by its physical (sensor) interaction."
(2) "What are useful models for capturing faults and disconnections within the coupled physical-computational system? ..."
(3) "What kind of properties that can be verified, ..."
(4) "What programming model is best suited for CPS applications ..."

3. Goal of the Research

The goal of our research is to develop techniques for faithfully and elegantly modeling cyber-physical systems. our approach is based on using logic programming for modeling computations, constraint logic programming for modeling continuous physical quantities, co-induction for modeling perpetual execution and coroutining for modeling concurrency in CPS. CPS are thus represented as coroutined co-inductive constraint logic programs which are subsequently used to elegantly verify cyber-physical properties of the system relating to safety, liveness and utility. This logic program can also be used for automatically generating implementation code for the CPS.

4. Current Status of the Research and Preliminary Results Accomplished

We assume that most CPS are state machines (finite automata) that control physical systems. In our formalism, state machines are modeled as logic programs [Llo87, Ste94], physical quantities are represented as continuous quantities (i.e., not discretized) and the constraints imposed on them by CPS physical interactions are faithfully modeled with *constraint logic programming over reals (CLP(R))* [Jaf94]. By considering co-inductive logic programming [Bar96, Gup07], we are able to model the non-terminating nature of CPS, and finally concurrency will be handled by allowing *coroutining* within logic programming computations.

Hybrid automata (of which timed automata and pushdown timed automata are instances) constitute the foundations for CPS. We have developed a general framework based on constraint logic programming and co-induction for modeling/verifying CPS [Sae10b].

The formalism that are used in this framework are timed automata and pushdown timed automata (PTA) which can be computationally modeled by combination of co-inductive logic programming (or Co-LP) and CLP(R). These can be generalized to hybrid automata and pushdown hybrid automata. We have developed a general method of converting timed automata and PTA to co-inductive CLP(R) programs. The method takes the description of a pushdown timed automaton (timed automaton) and generates a co-inductive constraint logic program over reals. We have shown how a co-inductive CLP(R) rendering of a pushdown timed automaton can be used to verify safety and liveness properties of complex timed systems. We have illustrated the effectiveness of our approach by showing how the well-known *generalized railroad crossing (GRC)* problem [Hei94] can be naturally modeled, and how its various safety and utility properties can be elegantly verified.

We have also developed timed grammars as a simple and natural formalism for describing timed languages. Timed grammars describe words that have real-time constraints placed on the times at which the words' symbols appear. Timed grammars can be generalized to hybrid grammars to model other types of continuous phenomena.

We extended the concept of context-free grammars (CFGs) to timed context-free grammars (TCFGs) and timed context-free ω-grammars (ω-TCFGs for brevity) [Sae10a]. Informally, a timed context-free grammar is obtained by associating clock constraints with terminal and non-terminal symbols appearing in the productions of a CFG. Timed context-free grammars describe timed context-free languages (TCFLs). A TCFL contains those strings that are accepted by the underlying untimed CFG but which also satisfy the timing constraints imposed by the associated clock constraints. Timed context-free ω-grammars describe timed context-free languages containing infinite-sized words, and are a generalization of timed ω-regular languages recognized by *timed automata*.

The words in a timed language consist of a sequence of symbols from the alphabet of the language the grammar accepts paired with the time-stamp indicating the time that symbol was seen. Timed languages are useful for modeling complex real-time, hybrid and cyber-physical systems.

We have shown how DCGs together with CLP(R) and co-induction can be used to develop efficient and elegant parsers for timed grammars. We have developed a system that takes an ω-TCFG and converts it into a DCG augmented with co-induction and CLP(R). The resulting co-inductive constraint logic program acts as a parser for the ω-TCFL recognized by the ω-TCFG. We have applied our general method of converting timed grammars to DCGs to the GRC problem with two tracks and presented simple timed context-free ω-grammar for *controller*, *gate*, and *track* components of this problem. The logic programming rendering of these ω-grammars are also generated by our system.

5. Open Issues and Expected Achievements

Our research group has done significant amount of work over the last few years to model CPS. However, most of it was focused on verifying on properties of systems [Sae10b, Sae10a, Ban10, Gup07]. Also, we were focused on solving the harder problems of logically modeling continuous quantities and perpetual nature of these systems. The concurrency aspect received less attention. As part of my research, I will continue my work on specification and verification of CPS but focus also on concurrency exhibited by CPS as well as generation their implementation from specifications in a provably correct manner. Research will be pursued along the following lines:

Timed π-calculus: I am studying the extension of π-calculus with continuous time. π-calculus [San02] is a well known formalism for modeling concurrency. Theoretically, the π-calculus can model concurrency, message exchange as well as infinite computation (through the infinite replication operator '!'), however, it does not deal with modeling of continuous quantities. I am developing an executable operational semantics of π-calculus in which concurrency is modeled by coroutining in logic programming (realized via *delay declarations of Prolog* [Ste94]) and infinite computations by co-induction [Saeon]. This operational semantics will be extended with continuous real time, which will be modeled with CLP(R). The executable operational semantics thus realized will automatically lead to an implementation of the timed π-calculus. The timed π-calculus will be used to model the GRC more faithfully and to verify its safety and utility properties. There is past work on developing executable operational semantics of the π-calculus (but not timed π-calculus) [Yan], that is based on logic programming, but it falls short as it is unable to model perpetual processes and infinite replication since co-inductive logic programming is a recent concept developed by our group.

Generating Implementation: Thus far we have seen how a cyber-physical system can be specified and its cyber-physical properties verified. We would like to use the specification to also generate the implementation code automatically. This way we can ensure that the implementation is faithful to the (verified) specification.

In order to generate the implementation code, the actions to be taken in the situation that a constraint in not met has to be specified as well. For example considering the GRC problem, what happens if the crossing-gate does not close within 2 units of time since the approach signal of a train was received). That is, normal situations as well as error situations have to be covered. Once the error situations are also specified, then it is relatively straightforward to generate the implementation along with all the exceptions and failsafe checks. Thus, research will be conducted on automatically deriving implementation of CPS from their verified specifications.

Real-life Applications: The modeling and implementation infrastructure we develop will be tested on real-life applications. These applications will come from manufacturing companies.

Acknowledgment

I would like to thank my dissertation advisor, Prof. Gupta, for his constant guidance, support and advice.

References

[Ban10] Ajay Bansal, Neda Saeedloei, and Gopal Gupta. Automated planning under realtime constraints. In *Florida AI Research Symposium*. To appear, 2010.

[Bar96] Jon Barwise and Lawrence Moss. *Vicious circles: on the mathematics of non-wellfounded phenomena*. Center for the Study of Language and Information, Stanford, CA, USA, 1996.

[Gup06] Rajesh Gupta. Programming models and methods for spatiotemporal actions and reasoning in cyber-physical systems. In *NSF Workshop on CPS*. 2006.

[Gup07] Gopal Gupta, Ajay Bansal, Richard Min, Luke Simon, and Ajay Mallya. Coinductive logic programming and its applications. In *ICLP*, pp. 27–44. Springer, 2007.

[Hei94] Constance L. Heitmeyer and Nancy A. Lynch. The generalized railroad crossing: A case study in formal verification of real-time systems. In *IEEE RTSS*, pp. 120–131. 1994.

[Jaf94] Joxan Jaffar and Michael J. Maher. Constraint logic programming: A survey. *J. Log. Program.*, 19/20:503–581, 1994.

[Lee08] Edward A. Lee. Cyber physical systems: Design challenges. In *ISORC*. 2008.

[Llo87] J. W. Lloyd. *Foundations of logic programming / J.W. Lloyd*. Springer-Verlag, Berlin ; New York :, 2nd edn., 1987.

[Sae10a] Neda Saeedloei and Gopal Gupta. Timed definite clause omega-grammars. In *Leibniz International Proceedings in Informatics*. To appear, 2010.

[Sae10b] Neda Saeedloei and Gopal Gupta. Verifying complex continuous real-time systems with coinductive clp(r). In *Languages and Automata Theory*. To appear, 2010.

[Saeon] Neda Saeedloei and Gopal Gupta. Timed pi-calculus and its applications. In preparation.

[San02] Davide Sangiorgi and David Walker. *The pi-Calculus*. Cambridge University Press, 2002.

[Sim07] Luke Simon, Ajay Bansal, Ajay Mallya, and Gopal Gupta. Co-logic programming: Extending logic programming with coinduction. In *ICALP*, pp. 472–483. 2007.

[Ste94] Leon Sterling and Ehud Shapiro. *The art of Prolog (2nd ed.): advanced programming techniques*. MIT Press, Cambridge, MA, USA, 1994.

[Yan] Ping Yang, C. R. Ramakrishnan, and Scott A. Smolka. A logical encoding of the pi-calculus: Model checking mobile processes using tabled resolution. In *VMCAI 2003*, pp. 116–131.

Technical Communications of the International Conference on Logic Programming, 2010 (Edinburgh), pp. 294–299
http://www.floc-conference.org/ICLP-home.html

REALIZING THE DEPENDENTLY TYPED λ-CALCULUS

ZACHARY SNOW [1]

[1] Department of Computer Science and Engineering
University of Minnesota
4-192 EE/CS Building
200 Union Street SE
Minneapolis, MN 55455
E-mail address: snow@cs.umn.edu

Abstract. Dependently typed λ-calculi such as the Edinburgh Logical Framework (LF) can encode relationships between terms in types and can naturally capture correspondences between formulas and their proofs. Such calculi can also be given a logic programming interpretation: the system is based on such an interpretation of LF. We have considered whether a conventional logic programming language can also provide the benefits of a Twelf-like system for encoding type and term dependencies through dependent typing, and whether it can do so in an efficient manner. In particular, we have developed a simple mapping from LF specifications to a set of formulas in the higher-order hereditary Harrop (*hohh*) language, that relates derivations and proof-search between the two frameworks. We have shown that this encoding can be improved by exploiting knowledge of the well-formedness of the original LF specifications to elide much redundant type-checking information. The resulting logic program has a structure that closely follows the original specification, thereby allowing LF specifications to be viewed as meta-programs that generate *hohh* programs. We have proven that this mapping is correct, and, using the Teyjus implementation of λProlog, we have shown that our translation provides an efficient means for executing LF specifications, complementing the ability the Twelf system provides for reasoning about them. In addition, the translation offers new avenues for reasoning about such specifications, via reasoning over the generated *hohh* programs.

1. Introduction and problem description

There is significant and growing interest in tools for specifying and reasoning about formal systems. These systems, such as programming languages and logics are typically defined in terms of a rules-based operational semantics. This leads to one obvious technique for specification: through the use of predicate logics, and languages like Prolog. In this setting we can encode expressions in the formal system as terms in the language, and use predicates to define the operational semantics. The systems that we might wish to specify can have a rich structure, for instance they may include a notion of binding or abstraction, and require operations like capture-avoiding substitutions and properties like α-equivalence. Implementing these features anew, for each logic or language that one wishes to specify, is time consuming and error prone, and so might benefit from language integration. Therefore

1998 ACM Subject Classification: Languages, Theory.
Key words and phrases: logical frameworks, logic programming.

Technical Communications of the 26th International Conference on Logic Programming, Edinburgh, July, 2010
Editors: Manuel Hermenegildo, Torsten Schaub
LIPIcs - Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/LIPIcs.ICLP.2010.294

logics or languages that embody these notions are often preferred, and have been widely used in the specification, and particularly the implementation, of such systems [Wha05].

Moving in a different direction, we might think of encoding properties of terms, and relationships between terms, not through *explicit* predicate definitions, but instead *implicitly* through types. Dependent types, like those provided by the dependently typed λ-calculus, provide powerful and natural methods for expressing these kinds of constraints. Furthermore, analyzing such specification for properties of correctness can often be reduced to type checking. This approach, distinct from that of predicate encodings, has also found widespread adoption in specifying and implementing formal systems, as well [Nec97, Ler06].

But specification is only the first goal. Given a specification of a formal system, we can think of doing several things: we can reason over the specifications, and thereby prove various properties about the logic or language being specified. We can also *animate* the specifications: for instance, having specified a language and its operational semantics, we might execute the specification in order to evaluate programs written in that language. The latter possibility actually benefits from the former; whereas traditionally we might specify and reason in one language, and then implement in another, here we execute exactly the same program about which we have reasoned. This handily removes the question of whether the implementation matches the specification.

Therefore our focus has been on problems associated with specifying formal systems using dependently typed languages, and then efficiently animating these specifications. In particular, we have sought to leverage existing work in the realm of efficient implementations of predicate logics when doing so, by designing translations from dependently typed languages to predicate logics.

2. Background and overview of the existing literature

As we have described, we can think of using various languages for specifying systems. On the one hand, we have higher order predicate logics like *hohh*, a logic based on Horn clauses, in which terms are those of the λ-calculus, but with support for handling the binding structure inherent in such terms. λProlog [Nad88] is a higher order logic programming language based on *hohh*, and extended in various ways (for instance, with a module system that supports programming in the large, with *ad hoc* polymorphism, and with facilities for interacting with the outside world). Furthermore, λProlog admits an efficient compiled implementation, as realized by the Teyjus system [Gac08]. Finally, there has already been work in analyzing and reasoning over programs written in λProlog [Gac09b, Bae10a], and there exist tools [Gac09a, Bae10b] for reasoning over it, both interactively and automatically, as well.

On the other hand, we have logics and languages founded on the dependently typed λ-calculus, for instance the The Edinburgh Logical Framework (LF) [Har93]. Twelf [Pfe99] is an implementation of LF that allows for reasoning over such specifications, and animating them. In and of itself, LF is strictly a specification language; it has no operational semantics of its own. However, one can apply the Curry-Howard Isomorphism [How80] to realize a *logic programming interpretation* of LF. In this context one defines types that correspond to judgments; then searching for an inhabitant of such a type corresponds to searching for a proof of the given judgment. Constructors for the type play the role of inference rules for constructing derivations of the judgment. And the discovered inhabitant, called a *proof term*, is itself a proof of the relevant judgment.

Twelf animates specifications in an interpreted fashion. There has already been research into improving this implementation by way of optimization (*e.g.*, [Pie06, Pie03], which have proved quite fruitful. In the end, however, the existing implementation of Twelf suffers due to its interpreted nature, and we find that it cannot be used on many realistically sized programs.

3. Goal of the research

The specific goal of my research as described herein has been to develop an efficient implementation of logic programming search for LF specifications, in particular through translation to λProlog, so that they may be executed using the Teyjus system. In addition, an important aspect of this work has been to ensure that this translation is *transparent*, so that the structure of the LF specification is clear from the structure of the generated logic program. This facilitates an understanding of the translation that allows the programmer to view LF specifications as a meta-programs, and enables reasoning over LF specifications using existing tools for reasoning over *hohh* programs.

4. Current status of the research

We have developed several translations from LF specifications into *hohh*. The problem of translating an LF specification into equivalent *hohh* has been investigated by Felty [Fel89, Fel90] — in this context, "equivalence" should be understood to mean the following: if an LF judgment has a derivation under a particular LF specification, then the translated judgment has a derivation under the translated specification in *hohh*. However this translation is not suitable for the purposes of logic programming, as it assumes that the proof term is already known, whereas when animating specifications this is exactly what is *not* known. Thus, taking inspiration from this translation we have developed our "simplified" translation that is suitable for logic programming.

Next we have improved this translation in two ways. First, the simplified translation is inefficient in that there are redundancies in proof search, that can be avoided through various observations about the nature of valid LF specifications. Indeed, this aspect of LF specifications, (that is, that they contain significant amounts of redundant typing information) has been investigated by, *e.g.*, Reed [Ree08], for the purpose of limiting the size of proof terms, which can become quite large. Addressing these redundancies is critical to the usefulness of the translation as an implementation mechanism for a separate reason: these redundancies can lead to inefficiencies, and even asymptotic changes in the complexity of algorithms implemented in specifications. Second, the simplified translation generates *hohh* logic programs that are relatively opaque, in the sense that it is not obvious that the logic program corresponds to the original specification. This is largely due to the fact that the simplified translation does not make much use of the rich type system afforded us by *hohh*.

We address these issues in a second, "optimized" translation. We first develop a technique for identifying and eliminating redundancies in proof search. And we improve the transparency of the translation by making a deeper use of the type system of *hohh*, to, for instance, reflect the non-dependent aspects of LF types as *hohh* types. Our final translation includes these optimizations, along with a few others, and results in a translation that is efficient and transparent. Because the generated λProlog programs share the same structure as the original LF specification we can view LF specifications as meta-programs. What's

more, as we've proved that our various translations are equivalent to LF, it is possible to reason over the resulting logic programs in order to reach conclusions about the properties of the original specification. And finally, we've developed a system that implements the translation.

Our implementation, named Parinati [Sno10] and written in Objective Caml, is released under the GNU General Public License version 3. Given a valid LF specification written in the concrete sytnax of Twelf, along with various types for which inhabitants should be sought, it generates a λProlog program that can be compiled and run using the Teyjus system.

5. Preliminary results accomplished

Preliminary experimental results comparing the efficiency of our implementation with that of Twelf are quite good: we have obtained an increase in efficiency of anywhere from 2 times to over 100 times in many cases. What's more, for sufficiently large problem sizes our implementation is almost always more efficient in terms of running time, apparently due to the extreme memory consumption that Twelf can exhibit — this is characteristic of certain kinds of interpreted implementations [Bri94] of logic programming search, including Twelf.

Beyond various performance metrics we have also demonstrated the transparency of the translation. In fact, our translation generates λProlog programs that almost exactly matches code that might be written "by hand", and the underlying structure of the original LF specification is completely clear. As already described, this allows the programmer to view the LF specification as a kind of meta-program for generating λProlog, and furthermore allows for reasoning over the resulting program as a method for reasoning over the original LF specification. What's more, this transparency is not only enabling, it is also elucidating: the generated *hohh* program is easier to reason about because it highlights those types that could have logical importance, and elides those that do not.

6. Open issues and expected achievements

There are a number of possible directions to take this work. First, there are still some examples in which our implementation is only as efficient, or even less efficient, than that of Twelf. We have begun a series of experiments to determine what factors are causing this slowdown, which we believe to be due to differences in the treatment of occurs checking between the two systems. Next, the efficiency of the implementation depends on our ability to accurately identify and eliminate redundancies. Any improvements we might make to this identification process should lead to performance increases.

Much of our work has been on optimizing our translation to λProlog; however, a different approach is to compile directly to, for instance, the Teyjus virtual machine's instruction set. By employing such an approach we might avoid some of the thorny questions associated with redundancy elimination. More generally, direct compilation could allow us to regain opportunities for those improvements that might be lost by translating first to λProlog and then relying on its implementation that is not specially optimized to treat LF-specific programs. However, this would clearly eliminate the possibility of treating LF as a meta-programming language for writing complex λProlog programs, as the requirement of transparency could not be fulfilled.

Twelf has several extensions aimed at the practicalities of programming. One particularly useful extension is the ability to use metavariables in the type for which an inhabitant is to be sought; these are instantiated during search. While the translation we have described includes this extension, we have not yet fully understood the theoretical aspects of it in terms of correctness of the translated programs.

Finally, we have only begun to understand how our translation fares when the purpose is to reason over an LF specification by analyzing the resulting *hohh* program. In the future we could apply existing tools to both LF specifications and their *hohh* counterparts generated by the translation, to judge the relative merits of reasoning in either system.

Acknowledgements

This work has been supported by the NSF grants CCR-0429572 and CCF-0917140. Opinions, findings, and conclusions or recommendations expressed in this papers are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[Bae10a] David Baelde, Dale Miller, and Zach Snow. Focused inductive theorem proving, 2010. Accepted for publication at IJCAR'10.

[Bae10b] David Baelde, Zach Snow, and Alexandre Viel. The Tac system, 2010.

[Bri94] Pascal Brisset and Olivier Ridoux. The architecture of an implementation of lambda-prolog: Prolog/mali. In *ILPS Workshop: Implementation Techniques for Logic Programming Languages*. 1994.

[Fel89] Amy Felty. *Specifying and Implementing Theorem Provers in a Higher-Order Logic Programming Language*. Ph.D. thesis, University of Pennsylvania, 1989.

[Fel90] Amy Felty and Dale Miller. Encoding a dependent-type λ-calculus in a logic programming language. In Mark Stickel (ed.), *Proceedings of the 1990 Conference on Automated Deduction*, *LNAI*, vol. 449, pp. 221–235. Springer, 1990.

[Gac08] Andrew Gacek, Steven Holte, Gopalan Nadathur, Xiaochu Qi, and Zach Snow. The Teyjus system – version 2, 2008. Available from `http://teyjus.cs.umn.edu/`.

[Gac09a] Andrew Gacek. Abella, 2009. Available from `http://abella.cs.umn.edu/`.

[Gac09b] Andrew Gacek. *A Framework for Specifying, Prototyping, and Reasoning about Computational Systems*. Ph.D. thesis, University of Minnesota, 2009.

[Har93] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1993.

[How80] William A. Howard. The formulae-as-type notion of construction, 1969. In J. P. Seldin and R. Hindley (eds.), *To H. B. Curry: Essays in Combinatory Logic, Lambda Calculus, and Formalism*, pp. 479–490. Academic Press, New York, 1980.

[Ler06] Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In J. Gregory Morrisett and Simon L. Peyton Jones (eds.), *POPL*, pp. 42–54. ACM, 2006.

[Nad88] Gopalan Nadathur and Dale Miller. An Overview of λProlog. In *Fifth International Logic Programming Conference*, pp. 810–827. MIT Press, Seattle, 1988.
URL `http://www.lix.polytechnique.fr/Labo/Dale.Miller/papers/iclp88.pdf`

[Nec97] George C. Necula. Proof-carrying code. In *Conference Record of the 24th Symposium on Principles of Programming Languages 97*, pp. 106–119. ACM Press, Paris, France, 1997.

[Pfe99] Frank Pfenning and Carsten Schürmann. System description: Twelf — A meta-logical framework for deductive systems. In H. Ganzinger (ed.), *16th Conference on Automated Deduction (CADE)*, no. 1632 in LNAI, pp. 202–206. Springer, Trento, 1999.

[Pie03] Brigitte Pientka and Frank Pfenning. Optimizing higher-order pattern unification. In *19th International Conference on Automated Deduction*, pp. 473–487. Springer-Verlag, 2003.

[Pie06] Brigitte Pientka. Eliminating redundancy in higher-order unification: A lightweight approach. In Ulrich Furbach and Natarajan Shankar (eds.), *IJCAR*, *Lecture Notes in Computer Science*, vol. 4130, pp. 362–376. Springer, 2006.

[Ree08] Jason Reed. Redundancy elimination for LF. *Electron. Notes Theor. Comput. Sci.*, 199:89–106, 2008. doi:http://dx.doi.org/10.1016/j.entcs.2007.11.014.

[Sno10] Zach Snow. Parinati, 2010. Available from `http://www.cs.umn.edu/~snow/parinati`.

[Wha05] Michael William Whalen. *Trustworthy translation for the requirements state machine language without events.* Ph.D. thesis, Minneapolis, MN, USA, 2005. Adviser-Heimdahl, Mats Per.

Technical Communications of the International Conference on Logic Programming, 2010 (Edinburgh), pp. 300–302
http://www.floc-conference.org/ICLP-home.html

STRUCTURED INTERACTIVE MUSICAL SCORES

MAURICIO TORO-BERMÚDEZ [1]

[1] Université de Bordeaux 1, Laboratoire Bordelais de Recherche en Informatique, Bâtiment A30. 351, cours de la Libération F-33405 Talence cedex, France.
URL: `http://www.labri.fr/perso/mtoro/`

Abstract. Interactive Scores is a formalism for the design and performance of interactive scenarios that provides temporal relations (TRs) among the objects of the scenario. We can model TRs among objects in Time Stream Petri nets, but it is difficult to represent global constraints. This can be done explicitly in the Non-deterministic Timed Concurrent Constraint (ntcc) calculus. We want to formalize a heterogeneous system that controls in one subsystem the concurrent execution of the objects using ntcc, and audio and video processing in the other. We also plan to develop an automatic verifier for ntcc.

Introduction

Interactive Scores (IS) are currently used for the design and performance of Electroacoustic music [All08a] and live spectacles [Bal09] (e.g., interactive theater plays and interactive museums). Both applications are based on Petri nets [All08b]. The main purpose of IS is to provide temporal relations; for instance, precedence between two objects and relations between their durations. Recently, we extended IS to support conditional branching together with temporal relations [TB10]. It is now possible to represent loops and choices.

We can model temporal relations in *Time Stream Petri nets (TSPN)* [Sen95], but it is difficult to represent global constraints involving (possibly) all the objects of the scenario. Instead, in *Concurrent Constraint Programming* (`ccp`) [Sar92] there are agents that reason about partial information contained in a constraint *store*; thus, global constraints are inherent in `ccp`. However, there is not discrete time in `ccp`, which makes it difficult to represent reactive systems.

There are some IS models based on extensions of `ccp` with discrete time. An example is the *Non-deterministic Timed Concurrent Constraint* (`ntcc`) model of IS [Nie02, All06]. `Ntcc` is an extension of `ccp` for non-determinism, asynchrony and discrete time. In the declarative view, `ntcc` processes can be interpreted as *linear temporal logic* formulae [Pnu77]. The `ntcc` model includes an inference system in this logic to verify properties of `ntcc` models. This inference procedure was proved to be of exponential time complexity. Nevertheless, we believe practical automatic verification could be envisioned for useful subsets of `ntcc` via model checking (see [Fal06]). At present, there is no such automatic verifier for `ntcc`.

1998 ACM Subject Classification: D. Software, D.1 Programming techniques, D.1.3 Concurrent programming, D.1.5 Logic programming.

Key words and phrases: ntcc, ccp, interactive scores, temporal relations, faust, ntccrt, heterogeneous systems, automatic verification.

Technical Communications of the 26th International Conference on Logic Programming, Edinburgh, July, 2010
Editors: Manuel Hermenegildo, Torsten Schaub
LIPIcs - Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/LIPIcs.ICLP.2010.300

Automated verification for `ntcc` will provide information about the correctness of the system to computer scientists, and will provide important properties about the scenario to its designers and users; for instance, reachability and liveness. We plan to augment `ntcc` models of IS with these features.

1. Current and future work

Functional AUdio STream (Faust) [Orl04] is a programming language for signal processing with formal semantics and *Ntccrt* [TB09] is a real-time capable interpreter for `ntcc`. We implemented a signal processing prototype where Faust and Ntccrt interact together. In the future, we want to define formal semantics to describe a heterogeneous system that includes three subsystems: (i) one based on `ntcc` to control discrete events from the user and to synchronize the objects of the scenario, (ii) another one based on Faust to process audio and video, and finally (iii) one in charge to load and play audio and video files.

At the time of this writing, there are no formal semantics of a heterogeneous system that synchronizes concurrent objects, handles global constraints, and controls audio and video streams. Modeling this kind of systems will be useful in other domains such as *machine musical improvisation* [Ass04] and *music video games*. An advantage over the existing implementations of these systems will be verification.

In the proof system of `ntcc`, we can prove properties like "10 time units (TUs) after the event e_A, during the next 4 TUs, the stream B is the result of applying a *gain filter* to the stream A". However, real-time audio processing cannot be implemented in Ntccrt because it requires to simulate 44100 TUs per second to process a 44.1 kHz sound. If we replace some `ntcc` processes by Faust plugins, we can execute such system efficiently, but we cannot verify that the properties of the system hold.

There are two open issues: (i) how to prove that a Faust plugin that replaces a `ntcc` process respect the temporal properties proved for the process, and (ii) whether an implementation of Interactive Scores in Ntccrt can be as efficient as the existing Petri nets implementation, or as one using synchronous languages such as *Signal* [Gau87], although the performance results from Ntccrt are promising[1].

Acknowledgement

I wish to acknowledge fruitful discussions with my supervisors Myriam Desainte-Catherine and Camilo Rueda. I also want to thank them for helping me writing this article and for guiding my current research.

References

[All06] Antoine Allombert, Gérard Assayag, M. Desainte-Catherine, and Camilo Rueda. Concurrent constraint models for interactive scores. In *Proc. of SMC '06*. 2006.

[All08a] Antoine Allombert, G. Assayag, and M. Desainte-Catherine. Iscore: a system for writing interaction. In *Proc. of DIMEA '08*, pp. 360–367. ACM, New York, NY, USA, 2008.

[All08b] Antoine Allombert, Myriam Desainte-Catherine, J. Larralde, and Gérard Assayag. A system of interactive scores based on qualitative and quantitative temporal constraints. In *Proc. of Artech 2008*. 2008.

[1]We ran a prototype of a score with conditional branching in Ntccrt. The score contains 500 temporal objects. The average duration of each time-unit was 30 ms, which is compatible with real-time interaction.

[Ass04] G. Assayag and Sholomo Dubnov. Using factor oracles for machine improvisation. *Soft Comput.*, 8(9):604–610, 2004.

[Bal09] P. Baltazar, A. Allombert, R. Marczak, J.M. Couturier, M. Roy, A. Sèdes, and M. Desainte-Catherine. Virage : Une reflexion pluridisciplinaire autour du temps dans la creation numerique. In *in Proc. of JIM*. 2009.

[Fal06] Moreno Falaschi and Alicia Villanueva. Automatic verification of timed concurrent constraint programs. *Theory Pract. Log. Program*, 6(4):265–300, 2006.

[Gau87] Thierry Gautier, Paul Le Guernic, and Löic Besnard. Signal: A declarative language for synchronous programming of real-time systems. In *Proc. of FPCA '87*. 1987.

[Nie02] M. Nielsen, C. Palamidessi, and F. Valencia. Temporal concurrent constraint programming: Denotation, logic and applications. *Nordic Journal of Comp.*, 1, 2002.

[Orl04] Y. Orlarey, D. Fober, and S. Letz. Syntactical and semantical aspects of faust. *Soft Comput.*, 8(9):623–632, 2004.

[Pnu77] A. Pnueli. The temporal logic of programs. In *In Proc. of the 18th IEEE Symposium on the Foundations of Computer Science (FOCS-77), pages 46-57. IEEE, IEEE Computer Society Press, 1977.* 1977.

[Sar92] Vijay A. Saraswat. *Concurrent Constraint Programming*. MIT Press, 1992.

[Sen95] Patrick Senac, Pierre de Saqui-Sannes, and Roberto Willrich. Hierarchical time stream petri net: A model for hypermedia systems. In *in Proc. of the 16th International Conference on Application and Theory of Petri Nets*, pp. 451–470. Springer-Verlag, London, UK, 1995.

[TB09] Mauricio Toro-B., Carlos Agón, Gérard Assayag, and Camilo Rueda. Ntccrt: A concurrent constraint framework for real-time interaction. In *Proc. of ICMC 09*. 2009.

[TB10] Mauricio Toro-B., Myriam Desainte-Catherine, and P. Baltazar. A model for interactive scores with temporal constraints and conditional branching. In *Proc. of Journées d'informatique musical (JIM) '10 (to appear)*. 2010.

Technical Communications of the International Conference on Logic Programming, 2010 (Edinburgh), pp. 303–305
http://www.floc-conference.org/ICLP-home.html

CUTTING-EDGE TIMING ANALYSIS TECHNIQUES

JAKOB ZWIRCHMAYR [1]

[1] Vienna University of Technology,
Argentinierstrasse 8,
1040 Vienna, Austria
E-mail address: jakob@complang.tuwien.ac.at
URL: http://www.complang.tuwien.ac.at

Abstract. This text gives an overview about my current research in timing analysis at the Vienna University of Technology. After a short introduction to the topic follows the description of an approach relying on CLP, the *implicit path enumeration technique (IPET)*. This technique is also used in a tool developed at the institute of Computer Languages (TuBound). Current timing analysis tools suffer from a few flaws worth further investigation in order to achieve better results than current state-of-the-art timing analysis tools.

Introduction

These days *embedded software systems (ESS)* are found in many devices we rely on in our daily lives. In many cases extensive testing gives us sufficient confidence in the product and it can be sold and used the way it was intended to. On the other hand, we rely on ESS that control very crucial mechanisms and functionality in devices and machines our safety and lives depend upon every day. Examples of such ESS usually come from the avionics and automotive industry, e.g. the technologies *fly-by-wire* and *drive-by-wire*. There are no mechanical links between the control column and the steering gear of an aircraft and the steering wheel and the wheels of a car [Kov10], or the system that is responsible for the proper functioning of the airbag in a car. These systems are considered safety-critical hard real-time systems (RTS). The airbag control software is required to compute and open the airbag fast enough if sensor data matches an accident condition. In this case, correct functioning of the ESS is not a matter of comfort and convenience but a matter of life and death. It is crucial to abide to certain resource bounds, e.g. memory consumption and time consumption. Guaranteeing program execution within a certain time bound is crucial for safety-critical hard RTS, i.e. guarantee that under no circumstances the time-bound is exceeded. The *worst-case execution time (WCET)* must in all cases be below the computed bound. The WCET-bound should be as precise as possible as overestimation usually implicates higher costs or redesign of the component. As a consequence, WCET underestimation is certainly not an option.

Key words and phrases: Verification, timing analysis, hard real-time systems, static analysis, worst-case execution time, loop-invariants, nested loop, symbolic computation, satisfiability modulo theories.

Technical Communications of the 26th International Conference on Logic Programming, Edinburgh, July, 2010
Editors: Manuel Hermenegildo, Torsten Schaub
LIPIcs - Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/LIPIcs.ICLP.2010.303

1. Background

Precision and performance of WCET analysis tools depend on the undecidable problem of identifying and separating feasible and infeasible program paths [Kov10]. Therefore WCET analyzers often require manual user intervention, often in the form of source- or binary-code annotations. Typical code elements that require user interaction (annotations) are loop constructs (upper bound on loop iterations) and recursive procedures (upper bound on recursion depth) [Pra09b]. There are two major problems due to this fact:

- Annotating binary code is tedious and even on source code level complications can occur, e.g. annotations in external components. Therefore, a fully automated procedure that infers this information is preferred.
- Manual annotations prevent the tool from formally establishing safety and accuracy of the analysis: the tool has to rely on a *trusted annotation base*, there are no guarantees that the user provided annotations are safe [Pra09b].

2. Goals

The *Cutting-edge Timing Analysis Techniques CeTAT* project is a cooperation between the Institute of Computer Languages and the Institute of Computer Engineering, at Vienna University of Technology. Some aspects of the problems of state-of-the-art WCET tools summarized in the previous sections can be overcome: there are approaches to verify the trusted annotation base supplied by the user. For example, the tool TuBound includes a bounded model checker that can be used to verify loop bounds inferred by the tool or provided by the user. The annotations are instrumented into the program as assertions that can be verified by the model checker. Nevertheless, there are complex loop constructs where such tools cannot find an upper bound on the number of loop iterations, preventing thus accurate WCET analysis.

The WCET community would benefit greatly from a common annotation language, such that tools can share information. Most tools have their own style of storing inferred information. Moreover, for easier tool comparison, a common annotation language would be helpful. Identifying (in)feasible paths is a complex task. Nevertheless there are various approaches in the area of symbolic computation (theorem proving) and termination analysis that are able to handle complex nested loops that would require manual annotations for most WCET tools. Our research aims at combining traditional timing analysis techniques and state-of-the-art approaches that use satisfiability modulo theories (SMT) to tackle the problem of complex nested loops [Gul09] with methods from symbolic computation and theorem proving.

3. Current status

I joined the CeTAT project group in March 2010. There is a good foundation of research in various directions of WCET analysis and the techniques we want to incorporate in order to pursue research in the CeTAT project. This includes in particular:

- Beyond Loop Bounds: Comparing Formative Annotation Languages for WCET Analysis [Kir10]. This work presents a survey of state-of-the-art annotation languages considered formative for the field. According to [Kir10], the precision, generality and efficiency of WCET analysis tools depend much on the expressiveness and usability of annotation languages.
- Constraint Solving for High-Level WCET Analysis [Pra09a]. The authors present the results achieved by their tool TuBound at the WCET tool-challenge 2008. TuBound is a constraint logic based approach for loop analysis developed at Vienna University of Technology.
- ABC: Algebraic Bound Computation for Loops. Presents a software tool for automatically computing symbolic upper bounds on the number of iterations of program loops [Bla09]. The authors of [Bla09] combine static analysis of programs with symbolic summation techniques to derive loops invariant relations among program variables.

As a starting point for the project we are currently investigating WCET benchmarks that contain loops that were not handled by TuBound in the tool challenge. It will be necessary to identify techniques that yield good usability, scalability, runtime performance, and most important, WCET results. Based on the results of our experimental evaluations we will design a new tool that outperforms current state-of-the-art tools in this respect.

References

[Bla09] Regis Blanc, Thomas Henzinger, Thibaud B. Hottelier, and Laura Kovacs. Abc: Algebraic bound computation for loops. In *LPAR-16 – 16th International Conference on Logic for Programming Artificial Intelligence and Reasoning*. 2009. Unpublished.

[Gul09] Sumit Gulwani and Florian Zuleger. The reachability-bound problem. Tech. Rep. MSR-TR-2009-146, Microsoft Research, 2009.

[Kir10] Raimund Kirner, Jens Knoop, Adrian Prantl, Markus Schordan, and Albrecht Kadlec. Beyond loop bounds: Comparing formative annotation languages for worst-case exeution time analysis. *Software and Systems Modeling*, 2010.

[Kov10] Dr. Laura Ildiko Kovacs. Cutting-edge timing analysis techniques for safety-critical real-time systems (cetat), 2010. Project description.

[Pra09a] A. Prantl, J. Knoop, M. Schordan, and M. Triska. Constraint solving for high-level WCET analysis. *ArXiv e-prints*, 2009.

[Pra09b] Adrian Prantl, Jens Knoop, Raimund Kirner, Albrecht Kadlec, and Markus Schordan. From trusted annotations to verified knowledge. In Niklas Holsti (ed.), *9th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, Dagstuhl, Germany, 2009.