Maurizio Gabbrielli
Gopal Gupta (Eds.)

LNCS 3668

Logic Programming

21st International Conference, ICLP 2005
Sitges, Spain, October 2005
Proceedings

Lecture Notes in Computer Science 3668

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

Editorial Board

Maurizio Gabbrielli Gopal Gupta (Eds.)

Logic Programming

21st International Conference, ICLP 2005
Sitges, Spain, October 2-5, 2005
Proceedings

Springer

Volume Editors

Maurizio Gabbrielli
Università di Bologna
Dipartimento di Scienze dell'Informazione
Mura Anteo Zamboni 7, 40127 Bologna, Italy
E-mail: gabbri@cs.unibo.it

Gopal Gupta
University of Texas at Dallas
Department of Computer Science
Richardson, TX 75083-0688, USA
E-mail: gupta@utdallas.edu

Library of Congress Control Number: 2005932757

CR Subject Classification (1998): D.1.6, I.2.3, D.3, F.3, F.4

ISSN 0302-9743
ISBN-10 3-540-29208-X Springer Berlin Heidelberg New York
ISBN-13 978-3-540-29208-1 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 11562931 06/3142 5 4 3 2 1 0

Preface

This volume contains the proceedings of the 21st International Conference on Logic Programming which was held in Sitges (Barcelona), Spain, from October 2nd to 5th, 2005. The conference was colocated with the International Conference on Constraint Programming (CP 2005) and the following 6 post-conference workshops:

- CICLOPS 2005: Colloquium on Implementation of Constraint and Logic Programming Systems
- CSLP 2005: Constraint Solving and Language Processing
- WCB 2005: Constraint Based Methods for Bioinformatics
- WLPE 2005: Logic-Based Methods in Programming Environments
- MoVeLog 2005: Mobile Code Safety and Program Verification Using Computational Logic Tools
- CHR 2005: Constraint Handling Rules

The conference coincided with a solar eclipse, which occurred on October 3rd and was visible in Sitges. No conference activities were scheduled at the time of the eclipse to allow delegates to view this extraordinary astronomical event.

Since the first conference that was held in Marseilles in 1982, ICLP has been the premier international conference for presenting research in logic programming. In this edition of the conference, extra attention was given to novel applications of logic programming and to work providing novel integrations of different areas. Colocation with CP 2005 further reinforced these themes, as it provided an opportunity for the exchange of ideas and cross-fertilization among two areas which have common roots. ICLP 2005 and CP 2005 shared the invited speakers to underscore this effort. ICLP 2005 broke new ground by holding a doctoral consortium for the first time in the ICLP series of conference.

One hundred and four abstracts were submitted in response to the call for papers. All but a few abstracts were followed by the submission of a full paper 1 week later. Each full paper was reviewed by at least three referees and the Program Committee finally selected 25 contributed papers and 15 poster papers. In addition, the program included invited talks by Ian Horrocks, Francesca Rossi, and Peter Stuckey, an industrial invited talk by Walter Wilson, a tutorial by Vítor Santos Costa on "Inductive Logic Programming, Statistical Relational Learning, and Its Applications", a panel on "Future Logic Programming Languages" and a doctoral consortium. The seven students selected for the doctoral consortium also presented their research as a poster in the program. The extended abstracts of the invited talks and of all the posters are also included in these proceedings.

ICLP 2005 was organized by the Association for Logic Programming (ALP), in collaboration with the Artificial Intelligence Research Institute of the Spanish Council for Scientific Research (IIIA-CSIC) and the Technical University of

Catalonia (UPC). Other sponsors included the Asociación Española de Inteligencia Artificial (AEPIA), the University of Texas at Dallas (UTD), CoLogNET, the University of Lleida, the Spanish Ministry of Education and Science and Cambridge University Press.

There are many people who deserve thanks for contributing to the success of the conference as well as to the creation of this volume. The PC members, aided by several external referees, produced timely and accurate reviews for the large number of submitted papers. The conference co-chairs, Pedro Meseguer and Javier Larrosa, did a superb job in organizing the joint event, as well as in successfully solving the many problems that were encountered. Hai-Feng Guo, the workshop chair, Felip Manya, the publicity chair, and Enrico Pontelli, the doctoral consortium chair, significantly contributed to the conference's success through their untiring efforts. Thanks to Bart Demoen for having accepted to organize again the traditional Programming Contest. Finally, thanks to all the authors who took an interest in ICLP 2005 and submitted papers, and to the developers of the Easy Chair conference management system that made our job as program co-chairs so much easier.

July 2005 Maurizio Gabbrielli and Gopal Gupta

Organization

Organizing Committee

Conference Co-chairs	Pedro Meseguer (IIIA-CSIC, Spain) Javier Larrosa (Technical University of Catalonia, Spain)
Program Co-chairs	Maurizio Gabbrielli (University of Bologna, Italy) Gopal Gupta (University of Texas at Dallas, USA)
Workshop Chair	Hai-Feng Guo (University of Nebraska at Omaha, USA)
Doctoral Consortium Chair	Enrico Pontelli (New Mexico State University, USA)
Publicity Chair	Felip Manya (IIIA-CSIC, Spain)

Program Committee

Roberto Bagnara (University of Parma, Italy)
Maurice Bruynooghe (KU Leuven, Belgium)
Stefan Decker (Digital Enterprise Research Institute, Ireland)
Giorgio Delzanno (University of Genoa, Italy)
Thom Fruehwirth (University of Ulm, Germany)
Maurizio Gabbrielli (University of Bologna, Italy, Program Co-chair)
Gopal Gupta (University of Texas at Dallas, USA, Program Co-chair)
Patricia Hill (University of Leeds, UK)
Joxan Jaffar (University of Singapore, Singapore)
Bharat Jayaraman (SUNY Buffalo, USA)
Javier Larrosa (Technical University of Catalonia, Spain, Conference Co-chair)
Michael Leuschel (University of Dusseldorf, Germany)
Massimo Marchiori (University of Venice, Italy and W3C, MIT, USA)
Pedro Meseguer (IIIA-CSIC, Spain, Conference Co-chair)
Juan Moreno Navarro (Technical University of Madrid, Spain)
Gopalan Nadathur (University of Minnesota, USA)
Illka Niemela (Helsinki University of Technology, Finland)
Catuscia Palamidessi (INRIA, France)
Enrico Pontelli (New Mexico State University, USA)
I.V. Ramakrishnan (SUNY Stony Brook, USA)
Vitor Santos Costa (Federal University of Rio de Janeiro, Brazil)
Harald Sondergaard (University of Melbourne, Australia)
Peter Stuckey (University of Melbourne, Australia)
Frank Valencia (University of Uppsala, Sweden)

External Reviewers

Marco Alberti
Etienne Payet
James Bailey
Marcello Balduccini
Nicola Stokes
Maria Garcia de la Banda
Ajay Bansal
Peter Baumgartner
Ralph Becket
Marc Bezem
Paolo Bouquet
Andrea Bracciali
Sebastian Brand
Maarten Marien
Nguyen Manh Thang
Hou Ping
Rudradeb Mitra
Daniel Cabeza
Manuel Carro
Wei-Ngan Chin
Tom Clothia
Michael Codish
Alvaro Cortes-Calabuig
Vitor Santos Costa
Stephen-John Craig
Marc Denecker
Yuxin Deng
Juergen Dix
Jin-Song Dong
Phan Minh Dung
Amy Felty
Antonio Fernandez
Michel Ferreira
Andrzej Filinski
Mario Florido
Alan M. Frisch
Marco Gavanelli
Juergen Giesl
Cinzia Di Giusto
Roberta Gori
Haifeng Guo
Gopal Gupta
James Harland
Andreas Harth
Angel Herranz
Tomi Janhunen
Michael Kifer
Andy King
Herbert Kuchen
Narayan Kumar
Jorge Lobo
Ricardo Lopes
Ruediger Lunde
Michael Maher
Ajay Mallya
Maarten Marien
Julio Mario
Kim Marriott
Viviana Mascardi
Marc Meister
Maria Chiara Meo
Fred Mesnard
Dale Miller
Roberto Montagna
Saikat Mukherjee
Susana Muñoz Hernández
George Necula
Shiri Nematollaah
Paulo Oliva
Mauricio Osorio
Sascha Ossowski
Jorge Andres Perez
Andrea Pescetti
Alessandra di Pierro
Inna Pivkina
Axel Polleres
Enrico Pontelli
Luis Omar Quesada
C.R. Ramakrishnan
Christophe Rigotti
Ricardo Rocha
Abhik Roychoudhury
Fernando Silva
Kostis Sagonas
Diptikalyan Saha
Chiaki Sakama

Peter Schachte
Tom Schrijvers
Luke Simon
Jan-Georg Smaus
Zoltan Somogyi
Tran Cao Son
Fausto Spoto
Martin Sulzmann
Tommi Syrjanen
Paolo Tacchella
Ana Paula Tomas
Francesca Toni
Mauricio Varea
V.N. Venkatakrishnan
Joost Vennekens
Razvan Voicu
Mark Wallace
Hui Wan
Qian Wang
David Warren
Herbert Wiklicky
Limsoon Wong
Eric Van Wyk
Guizhen Yang
Roland Yap
Enea Zaffanella

Table of Contents

OWL: A Description Logic Based Ontology Language
(Extended Abstract)

Ian Horrocks

School of Computer Science, University of Manchester,
Oxford Road, Manchester M13 9PL, UK
horrocks@cs.man.ac.uk

Description Logics (DLs) are a family of class (concept) based knowledge representation formalisms. They are characterised by the use of various constructors to build complex concepts from simpler ones, an emphasis on the decidability of key reasoning tasks, and by the provision of sound, complete and (empirically) tractable reasoning services.

Although they have a range of applications (e.g., reasoning with database schemas and queries [1,2]), DLs are perhaps best known as the basis for ontology languages such as OIL, DAML+OIL and OWL [3]. The decision to base these languages on DLs was motivated by a requirement not only that key inference problems (such as class satisfiability and subsumption) be decidable, but that "practical" decision procedures and "efficient" implemented systems also be available.

That DLs were able to meet the above requirements was the result of extensive research within the DL community over the course of the preceding 20 years or more. This research mapped out a complex landscape of languages, exploring a range of different language constructors, studying the effects of various combinations of these constructors on decidability and worst case complexity, and devising decision procedures, the latter often being tableaux based algorithms.

At the same time, work on implementation and optimisation techniques demonstrated that, in spite of the high worst case complexity of key inference problems (usually at least ExpTime), highly optimised DL systems were capable of providing practical reasoning support in the typical cases encountered in realistic applications [4]. With the added impetus provided by the OWL standardisation effort, DL systems are now being used to provide computational services for a rapidly expanding range of ontology tools and applications [5,6,7,8,9].

1 Ontology Languages and Description Logics

The OWL recommendation actually consists of three languages of increasing expressive power: OWL Lite, OWL DL and OWL Full. Like OWL's predecessor DAML+OIL, OWL Lite and OWL DL are basically very expressive description logics with an RDF syntax. OWL Full provides a more complete integration with RDF, but its formal properties are less well understood, and key inference

M. Gabbrielli and G. Gupta (Eds.): ICLP 2005, LNCS 3668, pp. 1–4, 2005.

problems would certainly be *much* harder to compute.[1] For these reasons, OWL Full will not be considered here.

More precisely, OWL DL is based on the $\mathcal{SHOIQ}$ DL [11]; it restricts the form of number restrictions to be unqualified (see [4]), and adds a simple form of Datatypes (often called concrete domains in DLs [12]). Following the usual DL naming conventions, the resulting logic is called $\mathcal{SHOIN}(\mathbf{D})$, with the different letters in the name standing for (sets of) constructors available in the language: $\mathcal{S}$ stands for the basic $\mathcal{ALC}$ DL extended with transitive roles [10], $\mathcal{H}$ stands for role hierarchies (equivalently, inclusion axioms between roles), $\mathcal{O}$ stands for nominals (classes whose extension is a single individual) [13], $\mathcal{N}$ stands for unqualified number restrictions and ($\mathbf{D}$) stands for datatypes) [14]. OWL Lite is equivalent to the slightly simpler $\mathcal{SHIF}(\mathbf{D})$ DL (i.e., $\mathcal{SHOIQ}$ without nominals, and with only functional number restrictions).

These equivalences allow OWL to exploit the considerable existing body of description logic research, e.g.:

- to define the semantics of the language and to understand its formal properties, in particular the decidability and complexity of key inference problems [15];
- as a source of sound and complete algorithms and optimised implementation techniques for deciding key inference problems [16,10,14];
- to use implemented DL systems in order to provide (partial) reasoning support [17,18,19].

Practical Reasoning Services. Most modern DL systems use *tableaux* algorithms to test concept satisfiability. Tableaux algorithms have many advantages: it is relatively easy to design provably sound, complete and terminating algorithms; the basic technique can be extended to deal with a wide range of class and role constructors; and, although many algorithms have a higher worst case complexity than that of the underlying problem, they are usually quite efficient at solving the relatively easy problems that are typical of realistic applications.

Even in realistic applications, however, problems can occur that are much too hard to be solved by naive implementations of theoretical algorithms. Modern DL systems, therefore, include a wide range of optimisation techniques, the use of which has been shown to improve typical case performance by several orders of magnitude; key techniques include lazy unfolding, absorption and dependency directed backtracking [16,20,19,21].

2 Research Challenges

The effective use of logic based ontology languages in applications will critically depend on the provision of efficient reasoning services to support both ontology design and deployment. The increasing use of DL based ontologies in areas such

[1] Inference in OWL Full is clearly undecidable as OWL Full does not include restrictions on the use of transitive properties which are required in order to maintain decidability [10].

as e-Science and the Semantic Web is, however, already stretching the capabilities of existing DL systems, and brings with it a range of challenges for future research.

These challenges include: improved scalability, not only with respect to the number and complexity of classes, but also with respect to the number of individuals that can be handled; providing reasoning support for more expressive ontology languages; and extending the range of reasoning services provided to include, e.g., explanation [22,23], and so-called "non-standard inferences" such as matching, approximation, and difference computations [24,25,26].

Finally, some applications will almost certainly call for ontology languages based on larger (probably undecidable) fragments of FOL [27], or on hybrid languages that integrate DL reasoning with other logical knowledge representation formalisms such as Datalog rules [28,29] or Answer Set Programming [30]. The development of such languages, and reasoning services to support them, extends the research challenge to the whole logic based knowledge representation community.

References

1. Calvanese, D., De Giacomo, G., Lenzerini, M., Nardi, D., Rosati, R.: Description logic framework for information integration. In: Proc. of the 6th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR'98). (1998) 2–13
2. Calvanese, D., De Giacomo, G., Lenzerini, M.: On the decidability of query containment under constraints. In: Proc. of the 17th ACM SIGACT SIGMOD SIGART Symp. on Principles of Database Systems (PODS'98). (1998) 149–158
3. Horrocks, I., Patel-Schneider, P.F., van Harmelen, F.: From $\mathcal{SHIQ}$ and RDF to OWL: The making of a web ontology language. J. of Web Semantics **1** (2003) 7–26
4. Baader, F., Calvanese, D., McGuinness, D., Nardi, D., Patel-Schneider, P.F., eds.: The Description Logic Handbook: Theory, Implementation and Applications. Cambridge University Press (2003)
5. Knublauch, H., Fergerson, R., Noy, N., Musen, M.: The protégé OWL plugin: An open development environment for semantic web applications. In Proc. of the 2004 International Semantic Web Conference (ISWC 2004). (2004) 229–243
6. Liebig, T., Noppens, O.: Ontotrack: Combining browsing and editing with reasoning and explaining for OWL Lite ontologies. In Proc. of the 2004 International Semantic Web Conference (ISWC 2004). (2004) 229–243
7. Visser, U., Stuckenschmidt, H., Schuster, G., Vögele, T.: Ontologies for geographic information processing. Computers in Geosciences (to appear)
8. Oberle, D., Sabou, M., Richards, D.: An ontology for semantic middleware: extending daml-s beyond web-services. In: Proceedings of ODBASE 2003. (2003)
9. Wroe, C., Goble, C.A., Roberts, A., Greenwood, M.: A suite of DAML+OIL ontologies to describe bioinformatics web services and data. Int. J. of Cooperative Information Systems (2003) Special Issue on Bioinformatics.
10. Horrocks, I., Sattler, U., Tobies, S.: Practical reasoning for expressive description logics. In Proc. of the 6th Int. Conf. on Logic for Programming and Automated Reasoning (LPAR'99). (1999) 161–180
11. Horrocks, I., Sattler, U.: A tableaux decision procedure for $\mathcal{SHOIQ}$. In: Proc. of the 19th Int. Joint Conf. on Artificial Intelligence (IJCAI 2005). (2005) To appear.

12. Baader, F., Hanschke, P.: A schema for integrating concrete domains into concept languages. In: Proc. of the 12th Int. Joint Conf. on Artificial Intelligence (IJCAI'91). (1991) 452–457
13. Blackburn, P., Seligman, J.: Hybrid languages. J. of Logic, Language and Information **4** (1995) 251–272
14. Horrocks, I., Sattler, U.: Ontology reasoning in the $\mathcal{SHOQ}$(D) description logic. In: Proc. of the 17th Int. Joint Conf. on Artificial Intelligence (IJCAI 2001). (2001) 199–204
15. Donini, F.M., Lenzerini, M., Nardi, D., Nutt, W.: The complexity of concept languages. Information and Computation **134** (1997) 1–58
16. Baader, F., Franconi, E., Hollunder, B., Nebel, B., Profitlich, H.J.: An empirical analysis of optimization techniques for terminological representation systems or: Making KRIS get a move on. Applied Artificial Intelligence. Special Issue on Knowledge Base Management **4** (1994) 109–132
17. Horrocks, I.: The FaCT system. In Proc. of the 2nd Int. Conf. on Analytic Tableaux and Related Methods (TABLEAUX'98). (1998) 307–312
18. Patel-Schneider, P.F.: DLP system description. In: Proc. of the 1998 Description Logic Workshop (DL'98), CEUR Electronic Workshop Proceedings, `http://ceur-ws.org/Vol-11/` (1998) 87–89
19. Haarslev, V., Möller, R.: RACER system description. In: Proc. of the Int. Joint Conf. on Automated Reasoning (IJCAR 2001). (2001) 701–705
20. Horrocks, I.: Using an expressive description logic: FaCT or fiction? In: Proc. of the 6th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR'98). (1998) 636–647
21. Horrocks, I., Patel-Schneider, P.F.: Optimizing description logic subsumption. J. of Logic and Computation **9** (1999) 267–293
22. Borgida, A., Franconi, E., Horrocks, I.: Explaining $\mathcal{ALC}$ subsumption. In: Proc. of the 14th Eur. Conf. on Artificial Intelligence (ECAI 2000). (2000)
23. Schlobach, S., Cornet, R.: Explanation of terminological reason-ing: A preliminary report. In: Proc. of the 2003 Description Logic Workshop (DL 2003). (2003)
24. Baader, F., Küsters, R., Borgida, A., McGuinness, D.L.: Matching in description logics. J. of Logic and Computation **9** (1999) 411–447
25. Küsters, R.: Non-Standard Inferences in Description Logics. Volume 2100 of Lecture Notes in Artificial Intelligence. Springer Verlag (2001)
26. Brandt, S., Küsters, R., Turhan, A.Y.: Approximation and difference in description logics. In: Proc. of the 8th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR 2002). (2002) 203–214
27. Horrocks, I., Patel-Schneider, P.F., Bechhofer, S., Tsarkov, D.: OWL rules: A proposal and prototype implementation. J. of Web Semantics **3** (2005) 23–40
28. Motik, B., Sattler, U., Studer, R.: Query answering for owl-dl with rules. J. of Web Semantics **3** (2005) 41–60
29. Rosati, R.: On the decidability and complexity of integrating ontologies and rules. J. of Web Semantics **3** (2005) 61–73
30. Eiter, T., Lukasiewicz, T., Schindlauer, R., Tompits, H.: Combining answer set programming with description logics for the semantic web. In: Proc. of the 9th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR 2004), Morgan Kaufmann, Los Altos (2004) 141–151

Preference Reasoning*

Francesca Rossi

Department of Pure and Applied Mathematics, University of Padova, Italy
frossi@math.unipd.it

Abstract. Constraints and preferences are ubiquitous in real-life. Moreover, preferences can be of many kinds: qualitative, quantitative, conditional, positive or negative, to name a few. Our ultimate goal is to define and study formalisms that can model problems with both constraints and many kind of preferences, possibly defined by several agents, and to develop tools to solve such problems efficiently.

In this paper we briefly report on recent work towards this goal.

Motivation and Main Goal. Preferences are ubiquitous in real life. In fact, most problems are over-constrained and would not be solvable if we insist that all their requirements are strictly met. Moreover, solvable problems have solutions with different desirability. Finally, many problems are more naturally described via preferences rather than hard statements. In some cases it could be more natural to express preferences in quantitative terms, while in other situations it could be better to use qualitative statements. Moreover, preferences can be unconditional or conditional. Furthermore, in many real life problems, constraints and preferences of various kinds may coexist.

Unfortunately, there is no single formalism which allows all the different kinds of preferences to be specified efficiently and reasoned with effectively. For example, soft constraints [1] are most suited for reasoning about constraints and quantitative preferences, while CP-nets [2] are most suited for representing qualitative and possibly conditional preferences. Our ultimate goal is to define and study formalisms that can model problems with both constraints and many kind of preferences, and to develop tools to solve such problems efficiently. Moreover, we also want to be able to deal with scenarios where preferences are expressed by several agents, and preference aggregation is therefore needed to find the optimal outcomes.

Preference Modelling Frameworks: Soft Constraints and CP-Nets. Soft constraints [1] model quantitative preferences by generalizing the traditional formalism of hard constraints. In a soft constraint, each assignment to the variables of a constraint is annotated with a level of its desirability, and the desirability of a complete assignment is computed by a combination operator applied to the local preference values. By choosing a specific combination operator and an ordered set of levels of desirability, we can select a specific class of soft constraints. Given a set of soft constraints, an ordering is induced over the assignments of the variables of the problem, which can be partial

* This is joint work with C. Domshlak, M. S. Pini, S. Prestwich, A. Sperduti, K. B. Venable, T. Walsh, and N. Yorke-Smith.

M. Gabbrielli and G. Gupta (Eds.): ICLP 2005, LNCS 3668, pp. 5–8, 2005.

or total. Given two solutions, checking whether one is preferable to the other one is easy: we compute the desirability values of the two solutions and compare them in the preference order. However, finding an optimal solution for a soft constraint problem is a combinatorially difficult problem.

CP-nets [2] (Conditional Preference networks) are a graphical model for compactly representing conditional and qualitative preference relations. They exploit conditional preferential independence by structuring a user's possibly complex preference ordering with the ceteris paribus assumption. CP-nets are sets of conditional ceteris paribus preference statements (cp-statements). For instance, the statement "I prefer red wine to white wine if meat is served." asserts that, given two meals that differ only in the kind of wine served and both containing meat, the meal with a red wine is preferable to the meal with a white wine. Given a CP-net, an ordering is induced over the set of assignments of its features. In general, such an ordering is a preorder (that is, reflexive and transitive). Given an acyclic CP-net, finding an optimal assignment to its features can be done in linear time. However, for cyclic CP-nets, it becomes NP-hard. Comparing two outcomes is NP-hard as well, even when the CP-net is acyclic.

Summarizing, CP-nets and soft constraints have complementary advantages and drawbacks. CP-nets allow one to represent conditional and qualitative preferences, but dominance testing is expensive. On the other hand, soft constraints allow to represent both hard constraints and quantitative preferences, and have a cheap dominance testing.

Comparing the Expressive Power of Different Formalisms. It would be very useful to have a single formalism for representing preferences that have the good features of both soft constraints and CP-nets. To achieve this goal, we may start by comparing their expressive power.

We could say that a formalism B is at least as expressive than a formalism A if from a problem expressed using A it is possible to build in polynomial time a problem expressed using B such that the optimal solutions are the same. If we use this definition to compare CP-nets and soft constraints, we see that hard constraints are at least as expressive as CP-nets. In fact, given any CP-net, we can obtain in polynomial time a set of hard constraints whose solutions are the optimal outcomes of the CP-net. On the contrary, there are some hard constraint problems for which it is not possible to find in polynomial time a CP-net with the same set of optimals. If instead, not only we must maintain the set of optimals, but also the rest of the ordering over the solutions, then CP-nets and soft or hard constraints are incomparable.

However, it is possible to approximate a CP-net ordering via soft constraints, achieving tractability of dominance testing while sacrificing precision to some degree [4]. Different approximations can be characterized by how much of the original ordering they preserve, the time complexity of generating the approximation, and the time complexity of comparing outcomes in the approximation.

Constraints and Preferences Together. Many problems have both constraints and qualitative and/or quantitative preferences. Unfortunately, reasoning with them both is difficult as often the most preferred outcome is not feasible, and not all feasible outcomes are equally preferred. For example, consider a constrained CP-net, which is a CP-net plus a set of hard constraints. This structure allows to model both qualitative conditional preferences and hard constraints. Its optimal outcomes (called "feasible

Pareto optimals" in [3]) are all the outcomes which are feasible and not dominated in the CP-net by any other feasible outcome. It is possible to obtain all such optimal outcomes by just solving a set of hard constraints [7]. In well defined cases, this avoids expensive dominance testing. If we want to avoid dominance testing completely, we can do that at the price of obtaining a superset of the feasible Pareto optimals by hard constraint solving. The same constraint-based procedure can be used also when we add soft constraints to a CP-net.

Learning Preferences. It is usually hard for a user to describe the correct preferences for his real-life problem. This is especially true for soft constraints, which do not have an intuitive graphical representation. We have shown that the use of learning techniques can greatly help in this respect, allowing users to state preferences both on entire solutions and subsets of the variables [8].

Preferences and Uncertainty. Preferences are a way to describe some kind of uncertainty. However, there is also uncertainty which comes from lack of data, or from events which are under Nature's control. Fortunately, in the presence of both preferences and uncertainty in the context of temporal constraints, we can reason with the same complexity as if we just had preferences [11]. Many approaches to deal with uncertainty are based on possibility theory. The handling of the coexistence of preferences and uncertainty via possibility theory allows for a natural merging of the two notions and leads to several promising semantics for ordering the solutions according to both their preference and their robusteness to uncertainty [6].

Preference Aggregation: Fairness and Non-manipulability. In many situations, we need to represent and reason about the simultaneous preferences of several agents. To aggregate the agents' preferences, which in general express a partial order over the possible outcomes, we can query each agent in turn and collect together the results. We can see this as each agent "voting" whether an outcome dominates another. We can thus obtain different semantics by collecting these votes together in different ways [9].

Having cast our preference aggregation semantics in terms of voting, it is appropriate to ask if classical results about voting theory apply. For example, what about Arrow's theorem [5], which states the impossibility of a fair voting system? Can we fairly combine together the preferences of the individual agents?

The definition of fairness considered by Arrow consists of the following desirable properties:

- Unanimity: if all agents agree that A is preferable to B, then the resulting order must agree as well.
- Independence to irrelevant alternatives: the ordering between A and B in the result depends only on the relation between A and B given by the agents.
- Monotonicity: whenever an agent moves up the position of one outcome in her ordering, then (all else being equal) such an outcome cannot move down in the result.
- Absence of a dictator: a dictator is an agent such that, no matter what the others say, will always dictate the resulting ordering among the outcomes.

Under certain conditions, it is impossible for a preference aggregation system over partially ordered preferences to be fair [10]. This is both disappointing and a little surprising. By moving from total orders to partial orders, we expect to enrich greatly our ability to combine preferences fairly. In fact, we can use incomparability to resolve conflict and thereby not contradict agents. Nevertheless, under the conditions identified, we still do not escape the reach of Arrow's theorem. Even if we are only interested in the most preferred outcomes of the aggregated preferences, it is still impossible to be fair.

Of course fairness is just one of the desirable properties for preference aggregations. Other interesting properties are related to the non-manipulability of a preference aggregation system: if an agent can vote tactically and reach its goal, then the system is manipulable. Results for totally ordered preferences show that non-manipulability implies the existence of a dictator. Unfortunately, this continues to hold also for partially ordered preferences.

Future Work. Much work has yet to be done to achieve the desired goal of a single formalism to model problems with both constraints and preferences of many kinds, and to solve them efficiently. For example, we are currently considering extensions of the soft constraint formalism to model both positive and negative preferences. Also, we are studying the relationship between optimal solutions in preference formalisms and Nash equilibria in game theory. Finally, we plan to study the notion of privacy in the context of multi-agent preference aggregation.

References

1. S. Bistarelli, U. Montanari, and F. Rossi. Semiring-based Constraint Solving and Optimization. Journal of the ACM, vol. 44, n. 2, pp. 201-236, 1997.
2. C. Boutilier, R. I. Brafman, C. Domshlak, H. H. Hoos, and D. Poole. CP-nets: A tool for representing and reasoning with conditional ceteris paribus preference statements. Journal of Artificial Intelligence Research, 21:135–191, 2004.
3. C. Boutilier, R. I. Brafman, C. Domshlak, H. H. Hoos, and D. Poole. Preference-based constraint optimization with CP-nets. Computational Intelligence, vol. 20, pp.137-157, 2004.
4. C. Domshlak, F. Rossi, K. B. Venable, and T. Walsh. Reasoning about soft constraints and conditional preferences: complexity results and approximation techniques. Proc. IJCAI-03, 215–220. Morgan Kaufmann, 2003.
5. J. S. Kelly. Arrow Impossibility Theorems. Academic Press, 1978.
6. M. S. Pini, F. Rossi, K. B. Venable. Possibility theory for reasoning about uncertain soft constraints. Proc. ECSQARU 2005, Barcelona, July 2005, Springer-Verlag LNAI 3571.
7. S. Prestwich, F. Rossi, K. B. Venable, T. Walsh. Constraint-based Preferential Optimization. Proc. AAAI 2005, Morgan Kaufmann, 2005.
8. F. Rossi and A. Sperduti. Acquiring both constraint and solution preferences in interactive constraint systems. Constraints, vol.9, n. 4, 2004, Kluwer.
9. F. Rossi, K. B. Venable, and T. Walsh. mCP Nets: Representing and Reasoning with Preferences of Multiple Agents. Proc. AAAI 2004, AAAI Press, 2004.
10. F. Rossi, M. S. Pini, K. B. Venable, and T. Walsh. Aggregating preferences cannot be fair. Proc. TARK X, Singapore, June 2005, ACM Digital Library.
11. F. Rossi, K. B. Venable, N. Yorke-Smith. Controllability of Soft Temporal Constraint Problems. Proc. CP 2004, Toronto, Springer LNCS 3258, 2004.

The G12 Project: Mapping Solver Independent Models to Efficient Solutions

Peter J. Stuckey[1], Maria Garcia de la Banda[2], Michael Maher[3], Kim Marriott[2], John Slaney[4], Zoltan Somogyi[1], Mark Wallace[2], and Toby Walsh[3]

[1] NICTA Victoria Laboratory, Department of Computer Science and Software Engineering, University of Melbourne, 3010 Australia
{pjs, zs}@cs.mu.oz.au
[2] School of Comp. Sci. & Soft. Eng., Monash University, Australia
{mbanda, marriott, mgw}@mail.csse.monash.edu.au
[3] NICTA Kensington Laboratory, University of New South Wales, 2052, Australia
{michael.maher, toby.walsh}@nicta.com.au
[4] NICTA Canberra Laboratory, Canberra ACT 2601, Australia
john.slaney@nicta.com.au

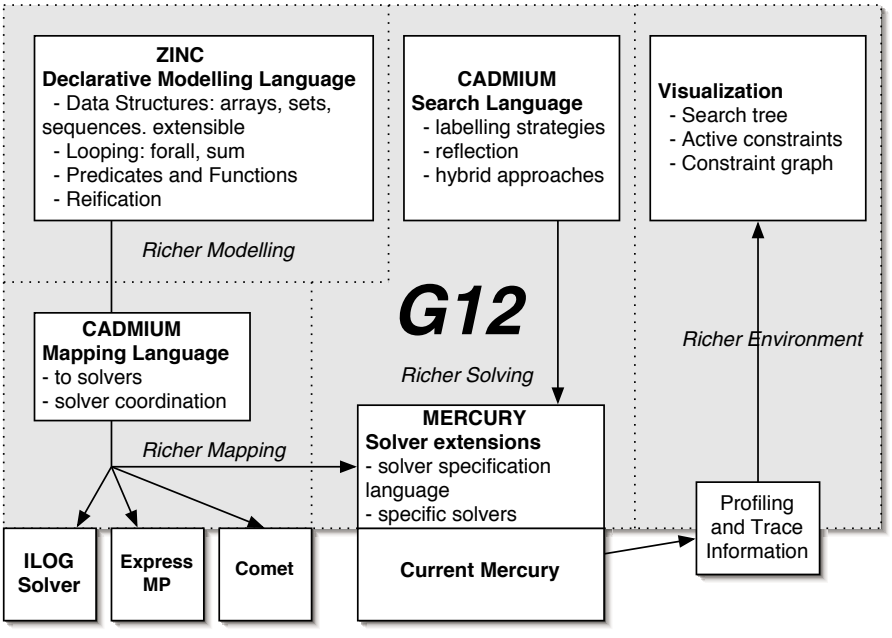
Abstract. The G12 project recently started by National ICT Australia (NICTA) is an ambitious project to develop a software platform for solving large scale industrial combinatorial optimisation problems. The core design involves three languages: Zinc, Cadmium and Mercury (Group 12 of the periodic table). Zinc is a declarative modelling language for expressing problems, independent of any solving methodology. Cadmium is a mapping language for mapping Zinc models to underlying solvers and/or search strategies, including hybrid approaches. Finally, existing Mercury will be extended as a language for building extensible and hybridizable solvers. The same Zinc model, used with different Cadmium mappings, will allow us to experiment with different complete, local, or hybrid search approaches for the same problem. This talk will explain the G12 global design, the final G12 objectives, and our progress so far.

1 Introduction

The G12 project aims to build a powerful and easy-to-use open source constraint programming platform for solving large scale industrial combinatorial optimization (LSCO) problems. The research project is split into four related threads: building richer modelling languages, building richer solving capabilities, a richer control language mapping the problem model to the underlying solving capabilities, and a richer problem-solving environment.

The underlying implementation platform will be the Mercury system. On top of Mercury the project will build a generic modelling language, called Zinc, and a mapping language, called Cadmium, which takes a Zinc model and generates a Mercury program. We also plan that Zinc and Cadmium will combine to output

M. Gabbrielli and G. Gupta (Eds.): ICLP 2005, LNCS 3668, pp. 9–13, 2005.



programs for different constraint solving systems such as ILOG Solver [6], Xpress MP [7] and Comet [2]. A diagram showing the four threads and how they interact with existing solvers and the current language Mercury is shown below.

2 Richer Modelling

The process of solving LSCO problems can be separated into creating the conceptual model, and an algorithm development process for mapping the conceptual model to a design model. This depends upon a language for writing conceptual models, and usually another language for writing design models.

In order to maintain clarity, flexibility, simplicity and correctness, we separate the conceptual modelling language Zinc from the mapping language Cadmium, which is both the design modelling language and the search language.

The best starting point for a universal conceptual modelling language is a purely declarative modelling language. Such a language allows the modeller to give a high-level specification of the constraint problem in terms natural to the problem itself. In order to do so it must include data structures that naturally arise in modelling such as arrays and sets, as well as be extensible in order to incorporate new problem specific structures such as jobs and tasks. We need natural constructs for specifying large constraints and large conjunctions of constraints. In order to encapsulate common problem structure we need to be able to specify predicates and functions in the modelling language for reuse.

The modeller needs to be able to specify requirements for robust, as well as optimal, solutions. *Robust* solutions are less sensitive to change in parameters, and reflect the reality that real solutions often need to be repaired when they are put into practice. It must be possible for the modelling language to specify the required type of robustness.

There are many challenges in the design of the Zinc language. For example, how can we make the language suitable for both an operations researcher experienced in using restricted mathematical modelling languages such as AMPL [1], as well as computer scientists used to the flexibility and power of full programming languages. OPL [5] the closest current language to what we envisage Zinc will be.

3 Richer Mapping

In order to make use of a conceptual model we must have some way of compiling it, that is mapping it to a design model. One advantage of separating of the conceptual modelling language from the design model is the ability to then rapidly experiment with different design models for the same conceptual model.

We wish to provide transparent and flexible ways of specifying how a conceptual model is mapped to a design model. Experience in developing solutions to industrial constraint problems has shown that we will often need to use two or more solving technologies to tackle a hard constraint problem. Various constraints will be treated by one solver, while other constraints will be treated by another. Some constraints may be treated by two or more solvers. When we are using multiple solvers we not only need to specify which constraints are sent to each solver, and how they are mapped to that solver, but how the solvers will interact. This must be supported by Cadmium.

G12 will not only need to provide a modelling interface to distinct solving methods from mixed integer programming (MIP), constraint programming (CP) and local search, but will also need to provide a modelling and mapping interface to methods for integrating these techniques. The design models for such an integrated scheme may involve combinations of algorithms from all three areas. The Cadmium language in which the design models are expressed must therefore subsume the expressive power of all the above languages. Much more is required however, since the interaction between local search and *branch-and-infer* search open a huge space of possible hybridisations.

4 Richer Solving

Constraint programming systems typically employ tree search to complement constraint propagation. Moreover the search is depth first and alternative search choices are only explored after backtracking to the relevant choice point. By contrast MIP search typically explores the search tree in a best-first fashion, which requires a multitude of *open* nodes to be recorded, ready for expansion at a later time. Recently systems like Mozart [4] have incorporated the open nodes

approach in CP. With G12 we shall pursue the convergence of CP and MIP search by reducing the cost of jumping between open nodes, and maintaining flexibility between the many different tree search strategies.

However local search techniques are playing an increasingly important role in CP. The Comet CP system [2] supports a wide range of local search techniques, with constraint handlers adapted to the local search paradigm. The final addition to the arsenal of search methods offered by G12 will be population-based search methods, such as genetic algorithms. These methods explore a whole population of solutions concurrently, and then combine the results from the population to focus the search on promising areas of the search space.

To date no system has enabled the user to specify the problem in terms of an algorithm-independent conceptual model, and have the computer map this into, say, an ant colony optimisation algorithm. The challenge for Cadmium is to make this mapping straightforward and concise, yet precise and flexible.

Another important research direction for richer solving will be developing algorithms for returning more robust solutions, more diverse solutions, or finding similar solutions to previous solutions.

5 Richer Environment

The key to solving complex industrial application problems is rapid applications development, with close end-user involvement. One of the most important factors in application development is the quality of the development environment. To support rapid application development, a rich solution development environment is essential.

The first stage in developing an application is constructing a correct Zinc and Cadmium model. This is much easier for the application programmer if solutions are graphically realized in a way that they can readily understand. The second and more time consuming phase is performance debugging in which we study the behaviour of the algorithms at runtime and understand exactly what is going on. Interaction with a running algorithm is necessary to detect its weaknesses, and to understand and build on its strengths. To support close end-user involvement, the problem solving behaviour must be made meaningful and transparent to the end-user. This requires that the algorithm behaviour be mapped back onto the problem model, so that the user can understand the behaviour in terms of the original application.

6 Conclusion

The G12 project aims, using the separation of the conceptual model from the design model, to provide a software framework where many, perhaps all, optimizations approaches can be experimented with efficiently. By allowing this exploration we hope to get closer to the ultimate goal of simply specifying the problem and letting the G12 system determine the best way to solve it.

References

1. AMPL: www.ilog.com/products/ampl/
2. Comet: www.cs.brown.edu/people/pvh/comet1.html
3. Mercury: www.cs.mu.oz.au/mercury/
4. Mozart: www.mozart-oz.org
5. OPL Studio: www.ilog.com/products/oplstudio/
6. ILOG SOLVER: www.ilog.com/products/solver/
7. Xpress MP: www.dashoptimization.com

Use of Logic Programming for Complex Business Rules

Walter G. Wilson

Symplicity Business Logic, Inc, Cendant Corporation
wgw@symplicity.biz

Abstract. This paper describes the move from a proprietary mainframe application to open systems application running the *360 º Fares System* whose business logic component is implemented in Prolog. It is one of the largest and most profitable Prolog applications written. Prolog is the business-rule component in a multi-component application that includes network, user interface, security data access tiers. There has been no downtime, scheduled or unscheduled, in the 360 º Fares System since June 2004.

1 Introduction

Prolog is a very powerful and economically important technology for implementing rule engines in complex systems. The successful experience at Cendant verifies this. A $19.8 billion Fortune 100 company, Cendant (NYSE: CD) is one of the foremost providers of travel and real estate services in the world. The subsidiary Cendant Travel Distribution Services (TDS) includes Galileo, a leading global distribution services (GDS) company, serving more than 44,000 travel agencies. TDS handles more than 206 billion airline fare quotes per year. In addition to high performance, reliability is an essential requirement. Galileo's GDS solutions, including its "360 º Fares System" have dependent businesses, ranging from agencies to airlines, world-wide, operating 24x7. The core of this system is a Prolog based rule engine for computing airline fares.

First we try to explain the scale of the company and the magnitude of the application. Then we briefly describe the two-layer rule architecture used in the implementation. This includes the base fare rules (filed by airlines, for example, in an industry standard format) and the meta-rule layer which applies governmental constraints and general fare-calculation rules. Finally, we discuss some of the strengths of LP for this type of application and describe some desired features.

2 Who Is Cendant

A $19.8 billion Fortune 100 company, Cendant is one of the foremost providers of travel and real estate services in the world; one of the world's largest hotel franchisers, the world's largest vacation ownership organization, and one of the world's largest car rental operators; the world's largest real estate brokerage franchiser, and the world's largest provider of outsourced corporate employee relocation services; and leading providers of travel information processing services worldwide.

M. Gabbrielli and G. Gupta (Eds.): ICLP 2005, LNCS 3668, pp. 14 – 20, 2005.

Cendant Travel Distribution Services (TDS), a subsidiary of Cendant Corporation (NYSE:CD), is one of the world's largest and most geographically diverse collections of travel brands and distribution businesses. A leading global travel intermediary, the division includes: Galileo, a leading global distribution services (GDS) company, serving more than 44,000 travel agencies and over 60,000 hotels; hotel distribution and services businesses (TRUST, THOR, WizCom and Neat Group); leading online travel agencies (Orbitz, CheapTickets®, Lodging.com, HotelClub.com and RatesToGo.com); Shepherd Systems, an airline market intelligence company; Travelwire, an international travel technology and software company; Travel 2/Travel 4, a leading international provider of long-haul air travel and travel product consolidator; online global corporate travel management solutions, through Travelport and Orbitz for Business.TDS connects the most buyers and sellers in the global travel marketplace.

Cendant TDS processes 5 billion transactions a month. Today, every one is either partially or entirely processed on open systems. On average, the TDS Data Center handles more than 350 million messages per day, 4051 messages per second and more than 912 million bookings/cancellations a year as well as more than 206 billion fare quotes per year.

Galileo's GDS solutions, including its "360 º Fares System", have dependent businesses, ranging from agencies to airlines, world-wide, operating 24x7. Outages to any of these systems have a direct impact on these users and as such must target operations that have no scheduled or unscheduled downtime, seven days a week. Proper operational procedures around the core Prolog application achieve this goal.

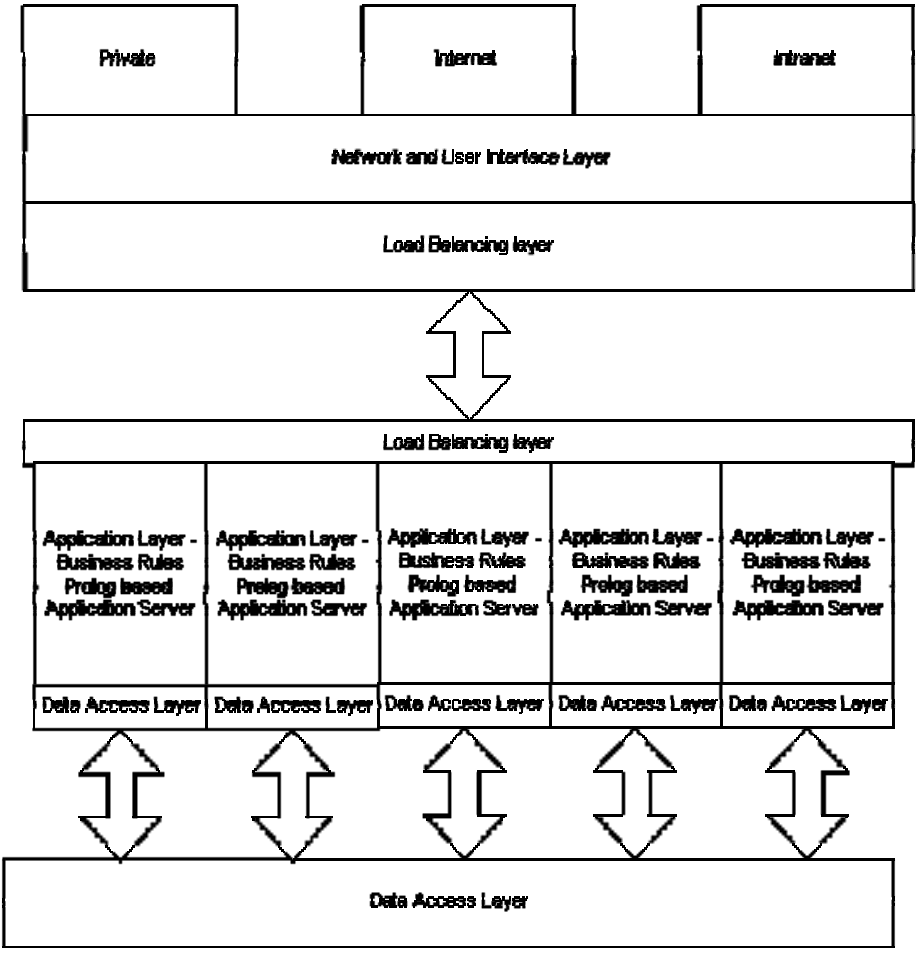
3 Architecture

A proprietary mainframe environment was previously used to host a mission critical "fare locating service", which became increasingly expensive and hard to maintain. The overall goal of the company's migration considerations was to move this key application to a low cost, high performing, highly redundant Intel based Linux complex.

The main mission critical fares application a subcomponent of Galileo (GDS) referred to as the 360 º Fares System. It is part of a large collection of software services and components that have evolved to serve the travel industry. It can be considered to be a distributed N-tier application in the usual style, with a user interface layer, an application layer, and a data access layer. The original implementation was mainframe based, using IBM's TPF high-performance transaction operating system. Portions of the TPF system dealing with processing of fare rules were migrated to the Linux/Prolog system. A 360 º Fares System data access layer written in C takes feeds of public and private fares and makes it available to the 360 º Fares System rules system.

The performance goal is subsecond average response time. This is much more rigorous than internet-only access typically requires. Due to its worldwide nature, the application must be available on a 24x7 continuous basis. This includes availability while new rules are being added or old ones refreshed, which you would expect -- but also during meta-rule (code) refresh. The system computes 200+ thousand million fare quotes/year. It contains millions of fare rules, and hundreds of thousands of meta-rules.

Here is a schematic diagram of the 360 º Fares System rules system. Not shown is the administrative GUI system written in Tcl/TK.)



Each of these layers is complex. For example, the *user interface layer* includes proprietary international private networks as well as web access components.

The *data access layer* is custom written in C. Data is stored on centralized data stores which are network connected to the application servers. NFS is the underlying protocol and is tuned for this environment, which is monotonic. Data is added, but not often deleted. Each application server in the complex has two layers of caching – one is the basic NFS file caching protocol, and the other is a custom, indexed cache for the base data. It is updated every few minutes from an industry standard common format and private, proprietary data that is converted to this format. Access to this layer from the Prolog application rule server is by use of the SICStus foreign-resource library to access custom C routines.

The *business rule layer* is implemented in Prolog. It has a socket connection to the communications and user interface layer. It effectively implements the functions of the "application server" in conventional N-tier models. Some number of Prolog servers are started on each machine in the complex.

The remainder of this section will discuss the rule architecture.

The rules are formed in *two layers*:

- A "base" layer that stores fare rules in an industry standard rule format (binary file) for specific fares: ATPCO and SITA – e. g.: "An infant fare for route R on carrier C is 85% of adult fare F when accompanied by an adult using that fare and when sold by agent A".
- A "meta" layer that implements the rules about how these rules are applied – e. g.: "fares must be priced using the currency of the country location of the initial departing flight, unless the purchaser is in countries C1, C2…"; and "A single itinerary can only enter each country in Europe N times."

Base rules include the basic fare information for particular flight and route combinations. This can include calculated fares. That is, simple fares such as "One adult from A to B on Sunday on this flight has this fare". They also provide more complex fare rules such as "One child when accompanied by an adult using fare X pays 50% of fare Y, if it would be available to an adult".

The meta layer interprets these base fares filed in the industry standard format. It also implements fare computation logic, such as how to handle minimum fare requirements, taxes, governmental travel restrictions, currency conversions, and much more. It also interprets Special indexes are created for these base rules as well as the meta rules.

One interesting characteristic to note is that the system must be correct. Many contract provisions with clients include reimbursement for wrong answers. Compare this with other applications where the license says in effect, "not responsible for errors, use at your own risk"!

4 Characteristics of Logic Programming-Based Rule System

Several characteristics of Logic Programs, specifically Prolog, were instrumental in the success of the application. We point out some of them. We then describe some additional tools that would be useful for such applications. Finally, some issues in using Prolog are highlighted.

4.1 Characteristics

These will not be a surprise to practitioners. For those interested in finding out more about these beneficial characteristics, I will outline a few technical aspects.

- *Managed storage*: also known as "garbage collection". This may be a given to practitioners and those programmers moving to dotNet and J2EE environments, but is not generally to C and C++ programmers. The difference was stark since the data access layer was written in C.
- *Symbolic term processing and unification*. This is hugely powerful. The creation, analysis, search and two-way patten matching provided by symbolic term unification saves thousands of lines of programming code. Specifically, I would point out two specific benefits:
  - Terms are structurally similar to XML. However, parsing (into internal structures) and generation appears to be much faster because they are intrinsic to

the language constructs. This means that these structures can be used like XML for network and intersystem communication, and data storage internal to they system, but in a more efficient fashion. Packages are available to convert to and from XML for external communication.
    - Marshalling and export of data structures are trivial, compared to C- or Java-like languages. If a variable X is "bound" to a symbolic term (read: "a data structure"), then a simple write(X) will traverse this structure in all its glory – including cyclic structures!
- *Table-based computation*. If you are familiar with Relational databases, then you should love Prolog. You can include relational tables and views directly in your program. Many instances of if-then-else logic can be turned into tables directly in the code.
- *Caching*: Controlled assert/retract. For practitioners, this was used as a means of caching some of the base rules and intermediate results, or "lemmas".
- *Repeat-fail loop* for cleaning stacks. Practitioners will know that there are two choices for processing loops: repeat-fail and recursion. Repeat-fail works well in this environment, since apart from lemmas there is no state that is kept between transactions, and this way we are assured that data can be released.
- *Term expansion*. Since Prolog programs themselves are much more easily manipulated as data than Java or C#, this came in very useful for the following:
    - *Debugging*: trace data for programmers and intelligible output for analysts and "business owners".
    - *Instrumentation*: term expansion is useful for embedding instrumentation when code is compiled, without having it cluttering up the code when writing or reading it.
    - *Data access*: many predicates had large numbers of arguments, so the implementation makes extensive use of O'Keefe's [3] suggestions on declaring names for arguments and referring to them by name. Term expansion converted the references into argument locations.
- *Modules*. This was of course useful. It is limited in that it is a flat namespace. I would prefer to see a hierarchical namespace like Java uses.
- *Libraries*. The large library of common routines was handy, particularly lists, heaps, and so-on.

4.2 Tools That Would Be Useful in Logic Programming-Based Complex Rule Systems

Here are Logic Programming tools that would be useful. Some dialects and experimental systems have some of these already.

- *CLP(fares).* This would be a constraint system designed to handle the specific constraints found in fare systems. For example, one might think that we are trying to find the minimum path between two nodes in a system so Dijkstra's algorithm (for example) would suffice. But the cost of an edge in the graph changes depending on the path you take to get there (e. g., round-trip vs. one-way), who you are (special discounts), and the path you will eventually take (minimum fare requirements). The application successfully coded specific algorithms and search strategies. It would be nice to let the programming system handle more of these.

- *Composite indexing* on terms (SICStus limitation). This is available in various forms in different dialects. Having to write our own indexing rather than just declare it is time consuming.
- CIAO-like relational data base *query-string optimization***.** This would allow us to more easily exploit the relational nature of Prolog and gain practitioners by declaring certain predicates to be stored externally (in a relational data base) and have the connections and queries optimized for that situation.
- *Unfolding* tools. This would be especially useful in conjunction with the RDB declarations, and the optimization of frequent queries.
- *Hierarchical* (Java-like) *name-spaces.* Those of you interested in Semantic Web know that namespace is important. The Prolog module name space is very flat and bushy.
- *More Logic Programmers*. The availability of competent programmers in such a "niche" language caused project development problems. More logic program applications might help that – or vice-verse.
- An *IDE* that does a lot for you
  - (including *source debugging* like AMZI Prolog)
  - *integrated* Prolog/foreign-resource debugging (via GDB for example)
  - logical variable *watchers*…
- *decimal arithmetic* (IEEE standard binary arithmetic rounding caused problems). Yes, we wrote our own.
- SMP-safe *threads*. Other application servers run on threads – why not Prolog? (no excuses – answers!).

4.3 Issues

The application is a huge, profitable success. Technically, Logic Programming is ideal for embedding as the business rule component in a multi-tier system. The primary issues are not technical; rather they are operational and based on perception:

- Prolog is perceived as "*non-standard*", meaning not everyone else is doing it. For comparison, .NET is non standard – or at least non Java - but has Microsoft behind it so it became a separate standard. Being "uncommon" has real consequences in developing large applications:
  - Unavailability of LP development *expertise*: experienced programmers were in short supply – the developers learned by doing.
  - Managers of large software projects tend to be very *risk-averse*. They have little motivation to choose the *best* solution. Rather, they have a lot of motivation to choose a good-enough solution that has fewer unknowns.
  - The "support team" of practitioners worldwide is not as populous as for Java or dotNet. Web sites containing "best-practices", shared code and experiences are few.
- All changes must be *ROI justified* – the "best" solution may not be worth it, or worth the cost of redoing the system. These real business constraints preclude large scale experimentation.

5 Conclusion

The architectural solution deployed by Cendant, which used highly redundant groups of server/storage clusters, has allowed us to achieve huge stability improvements. Since deployment of the solution in June 2003, we have not had a single customer affecting outage – scheduled or unscheduled.

In addition, the high performance and low cost of the environment has presented cost avoidance opportunities. Cendant has saved more than $100 million dollars compared to continuing to expand on the mainframe. Functionality has increased and time-to-market has decreased using the new application configuration.

References

1. AMZI: http://www.amzi.com
2. CIAO Prolog: http://clip.dia.fi.upm.es/Software/Ciao/
3. Cendant: www.cendant.com
4. O'Keefe, Richard. *The Craft of Prolog*. MIT Press. 1990
5. SICStus: http://www.sics.se/sicstus

A Generator of Efficient Abstract Machine Implementations and Its Application to Emulator Minimization*

José F. Morales[1], Manuel Carro[1], Germán Puebla[1]
and Manuel V. Hermenegildo[1,2]

[1] School of Computer Science, T. University of Madrid (UPM)
jfran@clip.dia.fi.upm.es, {mcarro, german, herme}@fi.upm.es
[2] Depts. of Comp. Sci. and El. and Comp. Eng., U. of New Mexico (UNM)
herme@unm.edu

Abstract. The implementation of abstract machines involves complex decisions regarding, e.g., data representation, opcodes, or instruction specialization levels, all of which affect the final performance of the emulator and the size of the bytecode programs in ways that are often difficult to foresee. Besides, studying alternatives by implementing abstract machine variants is a time-consuming and error-prone task because of the level of complexity and optimization of competitive implementations, which makes them generally difficult to understand, maintain, and modify. This also makes it hard to generate specific implementations for particular purposes. To ameliorate those problems, we propose a systematic approach to the automatic generation of implementations of abstract machines. Different parts of their definition (e.g., the instruction set or the internal data and bytecode representation) are kept separate and automatically assembled in the generation process. Alternative versions of the abstract machine are therefore easier to produce, and variants of their implementation can be created mechanically, with specific characteristics for a particular application if necessary. We illustrate the practicality of the approach by reporting on an implementation of a generator of production-quality WAMs which are specialized for executing a particular fixed (set of) program(s). The experimental results show that the approach is effective in reducing emulator size.

1 Introduction

The use of intermediate abstract machines as a means to compile and tune programs (specially those written in high-level languages with complex features) requires several components. In order to execute programs written in a source language $\mathcal{L}_P$, a compiler into the abstract machine language, $\mathcal{L}_A$, is needed.

* Work partially funded by the European Commission IST-FET programme, IST-2001-38059 *ASAP* project, and by the Spanish Ministry of Science and Education, TIC 2002-0055 *CUBICO* project. Manuel Hermenegildo is also supported by the Prince of Asturias Chair in Information Science and Technology at UNM. J. Morales is also supported by an MCYT/MEC fellowship co-financed by the European Social Fund.

M. Gabbrielli and G. Gupta (Eds.): ICLP 2005, LNCS 3668, pp. 21–36, 2005.

An emulator for $\mathcal{L}_A$, usually written in some lower-level language $\mathcal{L}_C$ for which there is a compiler to native code, performs the actual execution.[1] Traditional implementations based on abstract machines start with a fixed set of abstract machine instructions and then develop the compiler and the emulator.

One important concern when implementing such emulators is that of efficiency (see [1,2,3,4,5]), which depends greatly on the complexity of $\mathcal{L}_P$ and, of course, on the compiler and emulator technology. As a result, emulators are very often difficult to understand, maintain, and modify. This makes the implementation of *variants* of abstract machines a hard task, since both the compiler and emulator, which are rather complex, have to be rewritten by hand for each case. Variants of emulators have been (naturally) used to evaluate different implementation options for a language [4], often manually. Automating the creation of these variants will, additionally, make it possible to tailor a general design to particular applications or environments with little effort. A particularly daunting task is to adapt existing emulators to resource-constrained tasks, such as those found in pervasive computing. While this can clearly be done by carefully rewriting existing emulators, selecting alternative data representations, and, maybe, adapting them to the type of expected applications, we deem that this task is a too difficult one, especially taking into account the amount of different small devices which are ubiquitous nowadays.

In this work we propose an approach in which, rather than being hand-written, emulators and (back-end) compilers are automatically generated from a high-level description of the abstract machine instruction set. This makes it possible to easily experiment with alternative abstract machines and to evaluate the impact of different implementation decisions, since the corresponding emulator and compiler are obtained automatically.

In order to do so, rather than considering emulators for a particular abstract machine, we formalize emulators as parametric programs, written, for purposes of improved expressiveness, in a syntactical extension of $\mathcal{L}_C$ (as explained in Example 1) that can represent directly elements of $\mathcal{L}_A$ and which receive two inputs: a *program* to be executed, written in language $\mathcal{L}_A$, and a description of the abstract machine language $\mathcal{L}_A$ in which the operational definition of each instruction of $\mathcal{L}_A$ is given in terms of $\mathcal{L}_C$. E.g., we define a generic emulator as a procedure **interpret**$(program, M)$ which takes as input a program in the abstract machine language $\mathcal{L}_A$ and a definition M of the abstract machine itself and interprets the *program* according to M.

For the sake of maintainability and ease of manipulation, $\mathcal{L}_A$ is to be as close as possible to its conceptual definition. This usually affects performance negatively, and therefore a refinement step, based on pass separation [6], a form of staging transformations [7], is taken to convert programs written in $\mathcal{L}_A$ into programs written in $\mathcal{L}_B$, a lower-level representation for which faster interpreters can be written in $\mathcal{L}_C$. By formalizing adequately the transformation from $\mathcal{L}_A$ to $\mathcal{L}_B$ it is possible to do automatically:

[1] Implementations of abstract machines are usually termed *virtual machines*. We will, however, use the term *emulator* or *bytecode interpreter* to denote a virtual machine. This is in line with the tradition used in the implementation of logic programming languages.

- The translation of programs from $\mathcal{L}_A$ into $\mathcal{L}_B$.
- The generation of efficient emulators for programs in $\mathcal{L}_B$ based on interpreters for $\mathcal{L}_A$.
- The generation of compilers from $\mathcal{L}_P$ to $\mathcal{L}_B$ based on compilers from $\mathcal{L}_P$ to $\mathcal{L}_A$.

A high-level view of the different elements we will talk about in this paper appears in Figure 1. When the abstract machine description M is available, it is possible (at least conceptually) to partially evaluate the procedure **interpret** into an emulator for a (now fixed) M. Although this approach is attractive in itself, it has the disadvantage that the existence of a partial evaluator of programs written in $\mathcal{L}_C$ is required. Depending on $\mathcal{L}_C$, this may or may not be feasible.

A well known result in partial evaluation [8] is that it is possible to partially evaluate a partial evaluator w.r.t. itself and a particular program as static data. By taking the parametric emulator as static data for the partial evaluator, we can obtain an emulator generator (*emucomp*), which will produce an efficient emulator when supplied with a description of an abstract machine. This approach, known as the second Futamura projection [9], not only requires the availability of a partial evaluator for programs in $\mathcal{L}_C$ but also needs the partial evaluator to be self-applicable. Somewhat surprisingly, the structure of emulator generators is often easy to understand. The approach we will follow is therefore to write such an emulator generator directly by hand. The emulator generator we propose has been defined in such a way that it can produce an emulator whose code is comparable to a hand-written one when provided with a description of an abstract machine.

The benefits of our approach are multifold. Writing an emulator generator is clearly much more profitable than writing a particular emulator (though more difficult to achieve for the general case) since, with no performance penalty, it will make it possible to easily experiment with multiple variations of the original abstract machine. For example, and as discussed later, it is straightforward to produce reduced emulators. As an example of the application of our technique, and taking as starting point the instruction set of an existing emulator (a production-quality implementation of a modern version of the Warren Abstract Machine for Prolog [10,11]), we generate emulators which can be sliced with

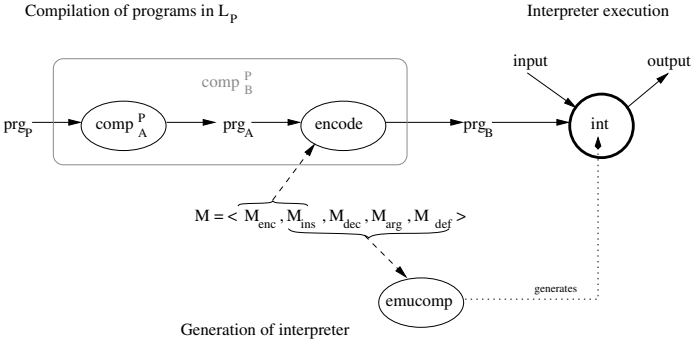


Fig. 1. "Big Picture" view of the generation of emulators

respect to the set of abstract machine instructions which a given application or sets of applications are going to actually use.

2 Automatic Generation of Emulators

In this section we will develop a compiler for emulators which takes a description of the machine and can produce emulators which are very close (and in some cases identical) to what a skilled programmer would craft.

Our initial source language is $\mathcal{L}_P$, and we assume that there is a compiler *comp* from $\mathcal{L}_P$ to an $\mathcal{L}_A$, a symbolic representation of a lower-level language intended to be interpreted by an emulator. We want *comp* to be relatively simple and independent from the low-level details of the implementation of the final emulator. The definition of $\mathcal{L}_A$ will be kept separate in M so that it can be used later (Section 2.2) in a generic emulator. Instructions in $\mathcal{L}_A$ can, in general, consult and modify a global state and change the control flow with (conditional) *jump*/*call* instructions.

2.1 Scheme of a Basic Emulator

Emulators have usually a main loop implementing a fetch-execute cycle. Figure 2 portrays an example, where that cycle is performed by a tail-recursive procedure. The reason to choose this scheme is because it allows a shorter description of some further transformations, but note that it can be converted automatically into a proper loop. The function:

$$fetch_A : locator_A \times program_A \rightarrow \langle ins_A, locator_A \rangle$$

returns, for a given program and program point, the instruction at that point (of type ins_A, a tuple containing instruction name and arguments) and the next location in the program, in sequential order. This abstracts away program counters, which can be symbolic, and indirections through the program counter. We will reuse this function, in different contexts, in the following sections.

Example 1 ($\mathcal{L}_A$ instructions and their semantics written in $\mathcal{L}_C$). The left hand side of each of the branches in the *case* expression of Figure 2 corresponds to one instruction in $\mathcal{L}_A$. The emulator $\mathbf{emu_A}$ is written in $\mathcal{L}_C$ (syntactically extended

```
emuA(p, program) ≡
    ⟨ins, p'⟩ = fetch_A(p, program)
    case ins of
        ⟨move, [r(i), r(j)]⟩   : reg[j] := reg[i];  p'' := p'
        ⟨jump, [label(l)]⟩     : p'' := l
        ⟨call, [label(l)]⟩     : push(p');  p'' := l
        ⟨ret, []⟩              : p'' := pop()
        ⟨halt, []⟩             : return
        otherwise              : error
    emuA(p'', program)
```

Fig. 2. An example of simple $\mathcal{L}_A$-level interpreter

to represent $\mathcal{L}_A$ instructions), and the semantics of each instruction is given in terms of $\mathcal{L}_C$ in the right hand side of the corresponding branch. The implementation of the memory model is implicit in the right hand side of the *case* branches; we assume that appropriate declarations for types and global variables exist. $\mathcal{L}_A$ instructions are able to move data between registers, do **jump**s and **call**s to subroutines, and stop the execution with the **halt** instruction. Alternative emulators can be crafted by changing the way $\mathcal{L}_A$ instructions are implemented. This must, of course, be done homogeneously across all the instruction definitions.

2.2 Parameterizing the Emulator

In order to make emulators parametric with respect to the abstract machine definition, we need to settle on an emulator scheme first (Figure 3) and to make the definition of the abstract machine precise. We will use a piecewise definition $M = (M_{def}, M_{arg}, M_{ins}, M_{absexp})$ of $\mathcal{L}_A$ which is passed as a parameter to the emulator scheme and which relates different parts of the abstract machine with a feasible implementation thereof. The meaning of every component of M (see also Example 2) is as follows:

M_{def}**.** The correspondence between every instruction of $\mathcal{L}_A$ and the code to execute it in $\mathcal{L}_C$.

M_{arg}**.** The correspondence between every argument for the instructions in $\mathcal{L}_A$ and the corresponding data in $\mathcal{L}_C$. M_{args} generalizes M_{arg} by mapping lists of arguments in $\mathcal{L}_A$ into lists of arguments in $\mathcal{L}_C$. The definitions of M_{def} and M_{arg} are highly dependent, and quite often updating one will require changes in the other.

M_{ins}**.** The instruction set, described as the name and the *format* every instruction in $\mathcal{L}_A$ accepts, i.e., which kinds of expressions in $\mathcal{L}_A$ can be handled by the instruction. The format is given as a list of abstract expressions of $\mathcal{L}_A$, whose definition is also included in M (see next item). For example, a **jump** instruction might be able to jump to a (static) label, but not to the address contained in a register, or a **move** instruction might be able to store a number in a register but not directly in a memory location. Note that the same instruction name can be used with different formats.

M_{absexp}**.** An abstraction function which returns the type of an instruction argument.

The interpreter in Figure 3 uses the definition of the semantics of $\mathcal{L}_A$ in terms of $\mathcal{L}_C$. For every instruction, arguments in $\mathcal{L}_A$ are translated into arguments in $\mathcal{L}_C$ by M_{args}, and M_{def} selects the right code for the instruction. Both M_{def} and M_{arg} are functions which return unevaluated pieces of code, which are meant to

$$
\begin{array}{l}
\mathbf{int_1}(p, program, M) \equiv \\
\quad \langle \langle name, args \rangle, p' \rangle = fetch_A(p, program) \\
\quad \text{if } \neg valid_A(\langle name, args \rangle, M_{ins}, M_{absexp}) \text{ then } \mathbf{error} \\
\quad cont = \lambda a \rightarrow [p'' := a] \\
\quad [\![M_{def}(p', cont, name, M_{args}(args))]\!] \\
\quad \mathbf{int_1}(p'', program, M)
\end{array}
$$

Fig. 3. Parametric interpreter for $\mathcal{L}_A$

$M_{def}(next, cont, name, args) =$
  case $\langle name, args\rangle$ of
    $\langle \mathbf{move}, [a, b]\rangle \rightarrow [a := b;\ cont(next)]$
    $\langle \mathbf{jump}, [a]\rangle \rightarrow [cont(a)]$
    $\langle \mathbf{call}, [a]\rangle \rightarrow [push(next);\ cont(a)]$
    $\langle \mathbf{ret}, []\rangle \rightarrow [cont(pop())]$
    $\langle \mathbf{halt}, []\rangle \rightarrow [return]$

$M_{ins} =$
  $\{\ \langle \mathbf{move}, [\mathbf{r}, \mathbf{r}]\rangle$
    $\langle \mathbf{jump}, [\mathbf{label}]\rangle$
    $\langle \mathbf{call}, [\mathbf{label}]\rangle$
    $\langle \mathbf{ret}, []\rangle$
    $\langle \mathbf{halt}, []\rangle\ \}$

$M_{arg}(arg) =$
  case arg of
    $\mathbf{r}(i) \rightarrow reg[i]$
    $\mathbf{label}(l) \rightarrow l$

$M_{absexp}(arg) =$
  case arg of
    $\mathbf{r}(\_) \rightarrow \mathbf{r}$
    $\mathbf{label}(\_) \rightarrow \mathbf{label}$
    otherwise $\rightarrow \bot$

Fig. 4. Definition of M for our example

be executed by $\mathbf{int_1}$ — this is marked by enclosing the function call by double square brackets. The next program location is set by a function *cont* which is handed in to M_{def} as an argument. The language expressions not meant to be evaluated but passed as data are enclosed inside square brackets. The context should be enough to distinguish them from those used to access array elements or to denote lists.

In order to ensure that no ill-formed instruction is executed (for example, because a wrongly computed location tries to access instructions outside the program scope), the function $valid_A$ checks that the instruction named *name* can understand the arguments *args* which it receives. It needs to traverse every argument, extract its type, which defines an argument format, and check that the instruction *name* can be used with arguments following that format.

Example 2 (Definitions for a trivial abstract machine in $\mathbf{int_1}$*).* In the definitions for M in Figure 4, the higher-order argument *cont* is used to set the program counter pointing to the instruction to be executed next. The instruction definitions do not check operator and operand types, since that has been taken care of by $valid_A$ by checking that the type of every argument which matches those accepted by the instruction at hand.

Instructions can in general admit several argument formats. For example, arithmetic instructions might accept integers and floating-point numbers. That would make M_{ins} have several entries for some instructions. This poses no problem, as long as M_{absexp} returns all abstractions for a given pattern and there is a suitable selection rule (e.g., the most concrete applicable pattern) is used to choose among different possibilities. For the sake of simplicity we will not deal with that case in this paper. Multi-format instructions are helpful when compiling weakly-typed languages, or languages with complex inheritance rules, where types of expressions might not be completely known until runtime. If this happens, compiling to a general case to be dynamically checked is the only solution.

2.3 A More Specialized Intermediate Language and Its Interpreter

The symbolic nature of $\mathcal{L}_A$, which should be seen as an intermediate language, makes it convenient to express instruction definitions and to associate internally

properties to them, but it is not designed be directly executed. Most emulators use a so-called *bytecode* representation, where many details have been settled: operation codes for each instruction (which capture the instruction name and argument types), size of every instruction, values of some arguments, etc. In return bytecode interpreters are quite fast, because a great deal of the work $\mathbf{int_1}$ has been statically encoded, so that many overheads may be removed. In short, the bytecode design focuses on achieving speed.

On the other hand, working right from the beginning with a low-level definition is cumbersome, because many decisions percolate through the whole language and seemingly innocent changes can force the update of a significant part of the bytecode definition (and, therefore, of its emulator). This is the main reason to keep $\mathcal{L}_A$ at a high level, with many details still to be filled in. It is however possible to translate $\mathcal{L}_A$ into a lower-level language, $\mathcal{L}_B$, closer to $\mathcal{L}_C$ and easier to represent using $\mathcal{L}_C$ data structures. That process can be instrumented so that programs written in $\mathcal{L}_A$ are translated into $\mathcal{L}_B$ and interpreters for $\mathcal{L}_A$ are transformed in interpreters for $\mathcal{L}_B$ using a similar encoding. Translating from $\mathcal{L}_A$ to $\mathcal{L}_B$ is done by a function:

$$encode : \mathcal{L}_A \rightarrow \mathcal{L}_B$$

encode accepts instructions in $\mathcal{L}_A$ (including name and arguments) and returns tokens in $\mathcal{L}_B$. The encoding function has to:

1. Assign a unique operation code (*opcode*) to each instruction in $\mathcal{L}_A$ when the precondition expressed by $valid_A$ holds (a compile-time error would be raised otherwise). This moves the overhead of checking formats from runtime to compile-time.
2. Take the arguments of instructions in $\mathcal{L}_A$ and translate them into $\mathcal{L}_B$.

encode is used to generate a compiler from $\mathcal{L}_P$ into $\mathcal{L}_B$ from a compiler from $\mathcal{L}_P$ into $\mathcal{L}_A$ (Figure 1). As *encode* gives a unique opcode to every combination of instruction name and format, it has an associated function:

$$decode : \mathcal{L}_B \rightarrow \mathcal{L}_A$$

which brings bytecode instructions back to its original form.[2] In order to capture the meaning of *encode* / *decode*, we augment and update the abstract machine definition to be $M = (M_{def}, M_{arg}, M_{ins}\ , M_{absexp}, M_{enc}, M_{dec})$ (see Figure 5 and Example 3). M_{ins} is derived from M_{ins} by capturing the opcode assignment. It accepts an opcode and returns the corresponding instruction in $\mathcal{L}_A$ as a pair $\langle name, format \rangle$. Argument encoding is taken care of by a new function M_{enc}. M_{dec} is the inverse of M_{enc}.

An interpreter $\mathbf{int_2}$ for $\mathcal{L}_B$ (see Figure 6) can be derived from $\mathbf{int_1}$ with the help of bytecode decoding. $\mathbf{int_2}$ receives an (extended) definition of M and uses it to retrieve the original instruction $\langle name, format \rangle$ in $\mathcal{L}_A$ corresponding to an opcode in a bytecode program (returned by $program[p]$, where p is a program counter in $\mathcal{L}_B$). The arguments are brought from the domain of $\mathcal{L}_B$ to the domain of $\mathcal{L}_A$ by M_{dec}, and code and argument translations defined by M_{def} and M_{arg} can then be employed as in $\mathbf{int_1}$.

[2] Both *encode* and *decode* may need to resolve symbols. As this is a standard practice in compiling (which can even be delayed until link time), we will not deal with that problem here.

$$
\begin{array}{lll}
M_{ins}\ (opcode) = & M_{enc}(arg) = & M_{dec}(t, f) = \\
\quad \text{case } opcode \text{ of} & \quad \text{case } arg \text{ of} & \quad \text{case } \langle t, f\rangle \text{ of} \\
\quad\quad 0 \rightarrow \langle \mathbf{move}, [\mathbf{r}, \mathbf{r}]\rangle & \quad\quad \langle \mathbf{r}(a)\rangle \rightarrow a & \quad\quad \langle a, \mathbf{r}\rangle \rightarrow \mathbf{r}(a) \\
\quad\quad 1 \rightarrow \langle \mathbf{jump}, [\mathbf{label}]\rangle & \quad\quad \langle \mathbf{label}(l)\rangle \rightarrow symbol(l) & \quad\quad \langle l, \mathbf{label}\rangle \rightarrow \mathbf{label}(l) \\
\quad\quad 2 \rightarrow \langle \mathbf{call}, [\mathbf{label}]\rangle & & \\
\quad\quad 3 \rightarrow \langle \mathbf{ret}, []\rangle & & \\
\quad\quad 4 \rightarrow \langle \mathbf{halt}, []\rangle & &
\end{array}
$$

Fig. 5. New Parts of the Abstract Machine Definition

$$
\begin{array}{l}
\mathbf{int_2}(p, prg, M) \equiv \\
\quad opcode = prg[p] \\
\quad \langle name, format\rangle = M_{ins}\ (opcode) \\
\quad \langle args, p'\rangle = decode_{ins}(format, [p], [prg], M) \\
\quad cont = \lambda a \rightarrow [\mathbf{int_2}(a, prg, M);\ return] \\
\quad [\![M_{def}(p', cont, name, M_{args}(args));\ cont(p')]\!]
\end{array}
\qquad
\begin{array}{l}
decode_{ins}(\langle f_1, \ldots, f_n\rangle, p, prg, M) = \\
\quad \langle \langle d_1, \ldots, d_n\rangle, p + 1 + n\rangle \text{ where} \\
\quad d_i = M_{dec}([prg[p + i]], f_i)
\end{array}
$$

Fig. 6. Parametric interpreter for $\mathcal{L}_B$

We want to note that in Figure 6 the recursive call has been placed inside the continuation code, which avoids the use of the intermediate variable p'' used in Figure 2 and makes it easier to apply program transformations.

Example 3 (Encoding instructions). Every combination of instruction name and format from Example 2, Figure 4, is assigned a different opcode. M_{ins} retrieves both the corresponding instruction name and format for every opcode. In Figure 8, the sample $\mathcal{L}_A$ program on the left is translated by *encode* into the program $\mathcal{L}_B$ on the right, which can be interpreted by $\mathbf{int_2}$ using the definitions for M.

2.4 A Final Emulator

The interpreter $\mathbf{int_2}$ in Section 2.3 still has the overhead associated with using continuously the abstract machine definition M. However, once M is fixed, it is possible to instantiate the parts of $\mathbf{int_2}$ which depend statically on M, to give another emulator $\mathbf{int_3}$. This can be seen as a partial evaluation of $\mathbf{int_2}$ with respect to M, i.e., $\mathbf{int_3} \equiv [\![spec]\!](\mathbf{int_2}, M)$. This returns an emulator written in $\mathcal{L}_C$ without the burden of translating instructions in $\mathcal{L}_B$ to the level of $\mathcal{L}_A$ in order to access the corresponding code and argument definitions in M_{def} and M_{arg}. Finally, and although $program[p]$ is not known at compile time, we can

$$
\begin{array}{l}
emucomp(M) = \\
\quad [\,\mathbf{emu_B}(p, prg) \equiv \\
\quad\quad \text{case } get\_opcode(p, prg) \text{ of} \\
\quad\quad\quad opcode_1 : inscomp(opcode_1, M) \\
\quad\quad\quad \ldots \\
\quad\quad\quad opcode_n : inscomp(opcode_n, M)] \\
\quad\quad \text{where } opcode_i \in domain(M_{ins}\ )
\end{array}
\qquad
\begin{array}{l}
inscomp(opcode, M) = \\
\quad [M_{def}(p', cont, name, M_{args}(args));\ cont(p')] \\
\text{where} \\
\quad \langle name, format\rangle = M_{ins}\ (opcode) \\
\quad \langle args, p'\rangle = decode_{ins}(format, [p], [prg], M) \\
\quad cont = \lambda a \rightarrow [\mathbf{emu_B}(a, prg);\ return]
\end{array}
$$

Fig. 7. Emulator Compiler

$\mathcal{L}_A$ **program**

```
move  r(0) r(2)
move  r(1) r(0)
move  r(2) r(1)
halt
```

$\mathcal{L}_B$ **program**

0 0 2	0 1 0	0 2 1	4

Fig. 8. Sample program

$$
\begin{array}{l}
\mathbf{emu_B}(p, program) \equiv \\
\quad \text{case } program[p] \text{ of} \\
\quad\quad 0 : reg[program[p+1]] := reg[program[p+2]]; \\
\quad\quad\quad \mathbf{emu_B}(p+3, program); return \\
\quad\quad 1 : \mathbf{emu_B}(program[p+1], program); return \\
\quad\quad 2 : push(p+2); \\
\quad\quad\quad \mathbf{emu_B}(program[p+1], program); return \\
\quad\quad 3 : \mathbf{emu_B}(pop(), program); return \\
\quad\quad 4 : return; return
\end{array}
$$

Fig. 9. Generated emulator

introduce a case statement which enumerates the possible values for the opcode, thus becoming *static*. This is a common technique to make partial evaluation possible in similar cases.

Since the interpreter structure is fixed, a compiler of emulators could be generated by specializing the partial evaluator for the case of $\mathbf{int_2}$, i.e.,

$$
\begin{array}{l}
emucomp : M \rightarrow code_C \\
emucomp = [\![spec]\!](spec, \mathbf{int_2})
\end{array}
$$

which is equivalent to the emulator compiler in Figure 7. It reuses the definition of $decode_{ins}$ seen in the previous section. Note that, as stated before, this emulator compiler has a regular structure, and we have opted to craft it manually, instead of relying on a self-applicable partial evaluator for $\mathcal{L}_C$. This compiler emulator, of course, does not need to be implemented in $\mathcal{L}_C$, and, in fact, in our particular implementation it is written in Prolog and it generates emulators in C.

Example 4 (The generated emulator). Figure 9 depicts an emulator for our working example, obtained by specializing $\mathbf{int_2}$ with respect to the machine definition in Example 3. Note the recursive call and *return*s at the end of every case branch which ensure that no other code after those statements is executed. All the recursive calls are tail recursions.

3 An Example Application: Minimal and Alternative Emulators

We will illustrate our technique with two (combined) applications: generating WAM emulators which are specialized for executing a fixed set of programs, and using different implementations of the WAM data structures. The former is very relevant in the context of applications meant for embedded devices and pervasive computing. It implements an automatic specialization scheme which starts at the Prolog level (by taking as input the programs to be executed) and, by slicing the abstract machine definition, traverses the compilation chain until the final specialized emulator for these programs is generated. The latter makes it possible to easily experiment with alternative implementation decisions.

We have already introduced how a piecewise definition of an abstract machine can allow making emulator generation automatic. In the rest of this section we will see how this technique can be used to generate such application-specific emulators, and we will report on a series of experiments performed around those

ideas. We will focus, for the moment, on generating correct emulators of *minimal size*, although the technique can obviously also be applied to investigating the impact of alternative implementations on performance.

3.1 Obtaining Specialized Emulators

The objective of specializing a program with respect to some criteria is to obtain a new program that preserves the initial semantics and is smaller or requires fewer operations. The source and target language are typically the same; this is expected, since specialization which operates across different translation levels is harder. It is however highly interesting, and applicable to several cases, such as the compilation to virtual machines and JIT compilation.

Among previous experiences which cross implementation boundaries we can cite [12], where automatically specialized WAM instructions are used as an intermediate step to generate C code which outperforms compilers to native code, and the Aquarius Prolog compiler [13] which carried analysis information along the compilation process to generate efficient native code.

As mentioned before, simplifying automatically hand-coded emulators (in order to speed them up or to reduce the executable size) written in $\mathcal{L}_C$ requires a specializer for $\mathcal{L}_C$ programs able to understand the emulator structure. The task can be quite difficult for efficient, complex emulators. Even in the case that the emulator can be dealt with, there are very few information sources to use in order to perform useful optimizations: the input data is, in principle, any bytecode program.

One way to propagate bytecode properties about a particular program p down to the emulator so that the specializer can do some effective optimization is by partially evaluating the emulator w.r.t. p and specializing the resulting program. Even if the specializer is powerful enough to work with this input, this solution has some drawbacks. The resulting code lacks some interesting properties: it is not as portable as the bytecode (since it is written in $\mathcal{L}_C$) and it is presumably less compact than the combination emulator + bytecode. Portability can often be sacrificed if compactness is preserved; in exchange, the resulting program is usually self-contained and generating stand-alone applications is in principle easier. This is not a bad scenario if there are automatic tools which can do a good job on these tasks (i.e., the code explosion generated by the partial evaluator is then taken care of by the specializer). Unfortunately, this is usually not the case.

An alternative approach is to express the specialization of the emulator in terms of *slicing* [14,15,16]. A slicing algorithm and the properties that it focus on, ϕ, of the emulator input, such as, e.g., bytecode reachable points, output variables, etc., are defined so that only the parts of the emulator (or a conservative approximation thereof) needed to maintain those properties have to be kept by the transformation.

One problem with this approach is that the bytecode is quite low level and the emulator too complicated to be automatically manipulated. However, our emulator generation scheme makes this problem more tractable. In our case $\mathcal{L}_B$ programs are generated from a higher-level representation which can be changed quite freely (even enriched with compiler-provided information to be later discarded by *encode*) and which aims at being easily manageable rather than efficient. It seems therefore more convenient to work at the level of $\mathcal{L}_A$ to

extract the slicing information, since it offers more simplification opportunities. It has to be noted that transforming the $\mathcal{L}_C$ emulator code using some $\mathcal{L}_A$ properties may be extremely difficult: to start with, suitable tools to work with $\mathcal{L}_C$ are needed, and they should be able to understand the relationship between $\mathcal{L}_B$ and $\mathcal{L}_A$ elements. It is much easier to work at the level of the definition of the abstract machine M, where $\mathcal{L}_A$ is completely captured, and where its relationship with $\mathcal{L}_B$ is contained.

We therefore formulate a slicing transformation that deals directly with M and whose result is used to generate a specialized emulator $\mathbf{emu_s}$:

$$\mathbf{emu_s} = emucomp([\![slice_M]\!](M, \phi))$$

$\mathbf{emu_s}$ can also be viewed as the result of slicing $emucomp(M)$ (i.e., $\mathbf{emu_B}$) with a particular slicing algorithm that, among other things, preserves the (loop) structure of the emulator.[3] That is, $slice_M$ deals with the instruction set or the instruction code definitions, and leaves complex data and control issues, quite common in efficient emulators, untouched and under the control of *emucomp*. Slicing can change all the components of the definition of M, including M_{def}, which may cause the compiled emulator to lose or specialize instructions. Note that when M_{ins} is modified, the transformation affects the compiler, because the *encode* function uses definitions in M.

3.2 Some Examples of Opportunities for Simplification

There is a variety of simplifications at the level of M that preserve the loop structure. They can be expressed in terms of the previously presented technique.

Instruction Removal: Programs compiled into bytecode can be scanned and brought back into $\mathcal{L}_A$ using M_{ins} to find the set I of instructions used in them. M is then sliced with respect to I and a new, specialized emulator is created as in Section 2.4. The new emulator may be leaner than the initial one since it probably has to interpret fewer instructions.

Removing Format Support: If $\mathcal{L}_A$ has instructions which admit arguments of different types (e.g., arithmetical operations which admit both integers and floating point numbers), programs that only need support for some of the available types can be executed in a reduced emulator. This can be achieved, again, by slicing M with respect to the remaining instruction and argument formats.

Removing Specialized Instructions: M can define specialized instructions (for example, for special argument values) or collapsed instructions (for often-used instruction sequences). Those instructions are by definition redundant, but sometimes useful for the sake of performance. However, not all programs require or benefit from them. When the compiler to $\mathcal{L}_A$ can selectively choose whether using or not those versions, a smaller emulator can be generated.

Obtaining the optimal set of instructions (w.r.t. some metric) for a particular program is an interesting problem. It is however out of the scope of this paper.

[3] Due to the simplicity of the interpreter scheme, this is not a hard limitation for most emulator transformations, as long as the transformation output is another emulator.

3.3 Experimental Evaluation

We tested the feasibility of the techniques proposed herein for the particular case of the compilation of Prolog to a WAM-based bytecode. We started off with Ciao [17], a real, full-fledged logic programming system, featuring a (Ciao)Prolog to WAM compiler, a complex bytecode interpreter (the emulator) written in C, and the machinery necessary to generate multi-platform, bytecode-based executables. We refactored the existing emulator as an abstract machine as described in the previous sections, and we implemented an emulator compiler which generates emulators written in C. We also implemented a slicer for removing unused instructions from the abstract machine definition.

Specialized emulators were built for a series of benchmark programs. For each of them, the WAM code resulting from its compilation was scanned to collect the set I of actually used instructions, and the general instruction set M_{ins} was sliced with respect to I in order to remove unused instructions. The resulting description was used to encode the WAM code into bytecode and to generate the specialized emulator. We have verified that, when no changes are applied to the abstract machine description, the generated emulator and bytecode representation are as optimized as the original ones. Orthogonally, we defined three slightly different instruction sets and generated specialized emulators for each of these sets and each of the benchmark programs, and we measured the resulting size (Table 1).

Table 1. Emulator sizes for different instruction sets

	Basic				***ivect***				***iblt***			
	loop (29331)		**bytecode**		**loop (33215)**		**bytecode**		**loop (34191)**		**bytecode**	
	full	strip	full	strip	full	strip	full	strip	full	strip	full	strip
hw	28%	71%	33116	48	29%	74%	31548	48	29%	75%	31136	48
boyer	26%	46%	40198	8594	27%	50%	38606	8542	28%	52%	38168	8512
crypt	27%	58%	33922	2318	28%	62%	32306	2242	28%	63%	31842	2186
deriv	27%	56%	33606	2002	28%	59%	32022	1958	28%	61%	31606	1950
exp	28%	59%	32102	498	29%	63%	30542	478	29%	63%	30114	458
fact	28%	69%	31756	152	29%	72%	30216	152	29%	73%	29804	148
fib	28%	70%	31758	154	29%	74%	30218	154	29%	74%	29798	142
knights	27%	54%	32306	702	28%	56%	30726	662	29%	57%	30298	642
nrev	27%	65%	31866	262	28%	69%	30322	258	28%	70%	29910	254
poly	26%	48%	34682	3078	27%	52%	33098	3034	27%	53%	32664	3008
primes	27%	56%	32082	478	28%	61%	30526	462	29%	62%	30102	446
qsort	27%	58%	32334	730	28%	61%	30778	714	28%	62%	30370	714
queens11	28%	55%	32248	644	29%	59%	30696	632	29%	60%	30220	564
query	28%	59%	32816	1212	29%	63%	31256	1192	29%	64%	30840	1184
stream_dyn	25%	42%	36060	2992	25%	45%	34420	2920	26%	45%	33890	2802
stream_opt	26%	46%	35152	2084	26%	49%	33516	2016	26%	49%	32990	1902
stream	26%	46%	34496	1428	27%	49%	32868	1368	28%	49%	32402	1314
tak	28%	67%	31886	282	29%	70%	30334	270	29%	71%	29910	254
Average	27%	56%			27%	60%			28%	61%		

The benchmarks feature both symbolic and numerical computation, and they are thus representative of several possible scenarios. The list of benchmarks includes some widely known programs which we will not describe here. Other programs, used less often as benchmarks, include `hw` (which prints "Hello world!"), `exp` (which computes 13^{7111} with a linear- and a logarithmic-time algorithm), `knights` (chess knight tour visiting once every board cell), `poly` (symbolically raise `1+x+y+z` to the n^{th} power), and `query` (query a database of countries, population, and area). Specially interesting are a set of signal processing programs, applied in wearable computing: `stream`, which generates 3-D stereo audio from mono audio, compass, and GPS signals to simulate the movement of a subject in a virtual world; `stream_dyn`, an improved version of `stream` which can use any number of different input signals and sampling frequencies, and `stream_opt`, an optimized version where number of signals and sampling frequency is fixed.

It has to be noted that, although most of these benchmarks are of moderate size, our aim in this section is precisely to show how to reduce automatically the footprint of an otherwise large engine for these particular cases. On the other hand, reduced size does not necessarily make them unrealistic, in the sense that they effectively perform non-trivial tasks. As an example, `stream_opt` processes audio in real time and with constant memory usage using Ciao Prolog in a 200MHz GumStix (a computer the size of a chewing gum).

The whole compilation process is fairly efficient. On a Pentium M at 1400MHz, with 512MB of RAM, and running Linux 2.6.10, the compiler compiles itself and generates a specialized emulator in 31.6 seconds: less than 0.1 seconds to generate the code of the emulator loop itself, 11.3 seconds to compile the compiler to bytecode (written in Prolog), and 20.3 seconds to compile all the C code: Prolog-accessible predicates written in C (e.g., builtins and associated glue code) and the generated emulator using `gcc` with optimization grade `-O2`. Both the Prolog compiler and emulator generator are written in (Ciao-)Prolog.

The results of the benchmarks are in Table 1, were different instruction sets were used. Columns under the ***basic*** label correspond to the instruction set of the original emulator. The ***ivect*** label presents the case for an instruction set where several compact instructions which are specialized to move register values before calls to predicates have been added to the studied emulator. Finally, columns below the label ***iblt*** shows results for the instruction set *iblt*, where specialized WAM instructions for the arithmetic builtins have been added to the emulator. In each of these set of columns, and for each benchmark, we studied the impact of specialization in the emulator size (the **loop** columns) and bytecode size (the **bytecode** columns).

The **bytecode** columns show two different figures: *full* is the bytecode size including all libraries used by the program and the initialization code (roughly constant for every program) automatically added by the standard compiler. The numbers in the *strip* column were obtained after performing dead code elimination at the Prolog level (such as removing unused Prolog library modules and predicates, producing specialized versions, etc. using information from analysis – see, e.g., [18] and its references) and then generating the bytecode. This specialization of Prolog programs at the source and module level is done by the Ciao preprocessor and is beyond the scope of this paper.

The **loop** columns contain, right below the label, the size of the main loop of the standard emulator with no specialization. For each benchmark we also show the percentage of reduction achieved with bytecode generated from full or specialized program with respect to the original, non-specialized emulator — the higher, the more savings.

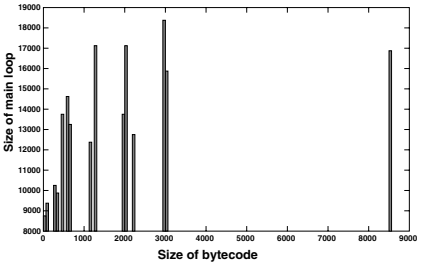


Fig. 10. Relationship between stripped bytecode size (x axis) and emulator size (y axis)

Even in the case when the emulator is specialized with respect to the *full* bytecode, we get a steady savings of around 27%, including library and initialization code. We can deduce that this is a good approximation of the amount of reduction that can be expected from typical programs where no redundant code exists. Of course, programs which use all the available WAM instructions can be crafted on purpose, but this is not the general case. In our experience, not even the compiler itself uses all the abstract machine instructions: we also generated an abstract machine specialized for it which was simpler (although only marginally) than the original one.

The savings obtained when the emulator is generated from specialized bytecode are more interesting. Savings range from 45% to 75%, averaging 60%. This shows that substantial size reductions can be obtained with our technique. The absolute sizes do not take into account ancillary pieces, such as I/O and operating system interfaces, which would be compiled or not with the main emulator as necessary, and which are therefore subject to a similar process of selection.

It might be expected that smaller programs would result in more emulator minimization. In general terms this is so, but with a wide variation, as can be seen in Figure 10. Thus, predicting in advance which savings will be obtained from a given benchmark in a precise way is not immediate.

4 Conclusions and Further Work

We have presented the design and implementation of an emulator compiler that generates efficient code using a high-level description of the instruction set of an abstract machine and a set of rules which define how intermediate code is to be represented as bytecode. The approach allowed separating details of the low-level data and code representation from the set of instructions and their semantics. We were therefore able to perform, at the abstract machine description level, transformations which affect both the bytecode format and the generated emulator without sacrificing efficiency.

We have applied our emulator compiler to a description of the abstract machine underlying a production, high-quality, hand-written emulator. The automatically generated emulator is as efficient as the original one. By using a slicer at the level of the abstract machine definition, we were able to reduce automatically its instruction set, producing a smaller, dedicated, but otherwise completely

functional, emulator. By changing the definition of the code corresponding to the instructions we were able to produce automatically emulators with substantial internal implementation differences, but still correct and efficient.

We expect to use the emulator compiler to also perform extensive experimentation with variations of abstract machine instruction sets and bytecode representations. We are already applying it in order to generate *ad-hoc* emulators for specific cases, such as those often found in pervasive computing. We are also experimenting with the combination of the emulator minimization with our automatic dead code elimination, slicing, and partial evaluation, in part at the level of the emulator and ancillary machinery and quite fully at the level of $\mathcal{L}_P$ (Ciao/Prolog, in our case) in order to generate high-quality, small executables.

There is also a strong connection with [19]: the fundamental pieces of the C code generation performed there and the code definitions for instructions in $\mathcal{L}_A$ are intimately related, and we have reached a single abstract machine definition in the Ciao system which is used both to generate bytecode emulators and to compile to C code. Also, as in [19], we are using compile-time information (such as type, mode, and determinism information), to generate better $\mathcal{L}_A$ code (e.g., generating specialized instructions or removing unnecessary instructions).

We also plan to redefine and refine the initial instruction set using information from execution profiling in order to merge frequently contiguous instructions, specialize them with respect to some frequently used argument value, etc. These variations have been explored in [20] for a fixed set of benchmarks, but emulators were hand-coded, somewhat limiting the per-application use of this approach.

References

1. Diaz, D., Codognet, P.: Design and Implementation of the GNU Prolog System. Journal of Functional and Logic Programming **2001** (2001)
2. Van Roy, P.: Can Logic Programming Execute as Fast as Imperative Programming? PhD thesis, Univ. of California Berkeley (1990) Report No. UCB/CSD 90/600.
3. Santos-Costa, V.: Optimising Bytecode Emulation for Prolog. In: International Conference on Principles and Practice of Declarative Programming (PPDP'99). Volume 1702 of LNCS., Springer-Verlag (1999) 261–277
4. Demoen, B., Nguyen, P.L.: So Many WAM Variations, So Little Time. In: Computational Logic 2000, Springer Verlag (2000) 1240–1254
5. Van Roy, P.: 1983-1993: The Wonder Years of Sequential Prolog Implementation. Journal of Logic Programming **19/20** (1994) 385–441
6. Hannan, J.: Staging Transformations for Abstract Machines. In: Partial Evaluation and Semantics-Based Program Manipulation (PEPM), ACM SigPlan Notices (1991)
7. Jørring, U., Scherlis, W.: Compilers and staging transformations. In: Thirteenth ACM POPL. (1986) 86–96
8. Jones, N., Gomard, C., Sestoft, P.: Partial Evaluation and Automatic Program Generation. Prentice Hall, New York (1993)
9. Futamura, Y.: Partial evaluation of computation process - an approach to a compiler-compiler. Systems, Computers, Controls **2** (1971) 45–50
10. Warren, D.: An Abstract Prolog Instruction Set. Technical Report 309, SRI International (1983)
11. Ait-Kaci, H.: Warren's Abstract Machine, A Tutorial Reconstruction. MIT Press (1991)

12. Ferreira, M., Damas, L.: Multiple Specialization of WAM Code. In: Practical Aspects of Declarative Languages. Number 1551 in LNCS, Springer (1999)
13. Van Roy, P., Despain, A.: High-Performance Logic Programming with the Aquarius Prolog Compiler. IEEE Computer Magazine (1992) 54–68
14. Tip, F.: A Survey of Program Slicing Techniques. Journal of Programming Languages **3** (1995) 121–189
15. Reps, T., Turnidge, T.: Program Specialization via Program Slicing. In Danvy, O., Glück, R., Thiemann, P., eds.: Partial Evaluation. Dagstuhl Castle, Germany, February 1996, Springer LNCS 1110 (1996) 409–429
16. Weiser, M., ed.: Information and Software Technology: Special Issue on Program Slicing. Volume 40. Elsevier (1999)
17. Bueno, F., Cabeza, D., Carro, M., Hermenegildo, M., López-García, P., Puebla, G.: The Ciao Prolog System. Reference Manual (v1.8). Technical Report CLIP4/2002.1, School of Computer Science, UPM (2002) Available at `http://clip.dia.fi.upm.es/Software/Ciao/`.
18. Puebla, G., Hermenegildo, M.: Abstract Specialization and its Applications. In: Proc. of PEPM'03, ACM Press (2003) 29–43 Invited talk.
19. Morales, J., Carro, M., Hermenegildo, M.: Improving the Compilation of Prolog to C Using Moded Types and Determinism Information. In: Intnl. Symposium on Practical Aspects of Declarative Languages. Number 3507 in LNCS, Springer (2004) 86–103
20. Nässén, H., Carlsson, M., Sagonas, K.: Instruction Merging and Specialization in the SICStus Prolog Virtual Machine. In: Proceedings of the 3rd ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming, ACM Press (2001) 49–60

On the Relation Between Answer Set and SAT Procedures (or, Between CMODELS and SMODELS)

Enrico Giunchiglia and Marco Maratea

STAR-Lab, DIST, University of Genova,
viale Francesco Causa, 13 — 16145 Genova, Italy
{enrico, marco}@dist.unige.it

Abstract. Answer Set Programming (ASP) is a declarative paradigm for solving search problems. State-of-the-art systems for ASP include SMODELS, DLV, CMODELS, and ASSAT.

In this paper, our goal is to study the computational properties of such systems both from a theoretical and an experimental point of view. From the theoretical point of view, we start our analysis with CMODELS and SMODELS. We show that though these two systems are apparently different, they are equivalent on a significant class of programs, called tight. By equivalent, we mean that they explore search trees with the same branching nodes, (assuming, of course, a same branching heuristic). Given our result and that the CMODELS search engine is based on the Davis Logemann Loveland procedure (DLL) for propositional satisfiability (SAT), we are able to establish that many of the properties holding for DLL also hold for CMODELS and thus for SMODELS. On the other hand, we also show that there exist classes of non-tight programs which are exponentially hard for CMODELS, but "easy" for SMODELS. We also discuss how our results extend to other systems.

From the experimental point of view, we analyze which combinations of reasoning strategies work best on which problems. In particular, we extended CMODELS in order to obtain a unique platform with a variety of reasoning strategies, and conducted an extensive experimental analysis on "small" randomly generated and on "large" non randomly generated programs. Considering these programs, our results show that the reasoning strategies that work best on the small problems are completely different from the ones that are best on the large ones. These results point out, e.g., that we can hardly expect to develop one solver with the best performances on all the categories of problems. As a consequence, (i) developers should focus on specific classes of benchmarks, and (ii) benchmarking should take into account whether solvers have been designed for specific classes of programs.

1 Introduction

Answer Set Programming (ASP) is a declarative paradigm for solving search problems. State-of-the-art systems for ASP include SMODELS, DLV, CMODELS, and ASSAT.[1] Our

[1] See http://www.tcs.hut.fi/Software/smodels, http://www.dbai.tuwien.ac.at/proj/dlv, http://assat.cs.ust.hk, http://www.cs.utexas.edu/users/tag/cmodels.html, respectively.

M. Gabbrielli and G. Gupta (Eds.): ICLP 2005, LNCS 3668, pp. 37–51, 2005.

goal is to study the computational properties of such systems both from a theoretical and an experimental point of view.

From the theoretical point of view, we start our analysis with CMODELS and SMODELS. Given a program Π, while SMODELS (and also DLV) is a native procedure which directly operate on Π, CMODELS (and ASSAT) computes a set of clauses corresponding to the Clark's completion of Π, and then invoke a propositional satisfiability (SAT) solver based on Davis Logemann Loveland procedure (DLL). We show that though CMODELS and SMODELS are apparently different, they are equivalent on a significant class of programs, called tight. By equivalent, we mean that they explore search trees with the same branching nodes, (assuming, of course, a same branching heuristic). Given our equivalence result and that CMODELS search engine is based on DLL, we are able to establish that many of the properties holding for DLL also hold for CMODELS and thus, when considering tight programs, also for SMODELS. For instance we show that:

1. There exist classes of tight formulas which are exponentially hard both for CMODELS and SMODELS.
2. There exist classes of non tight programs which are exponentially hard for CMODELS but very easy (i.e., solved without search) by SMODELS.
3. In SMODELS, deciding the "best" literal to branch on, is both NP-hard and co-NP hard and in PSPACE for tight programs.

These are just a few of the many results (i) that are already known for DLL, (ii) that can be easily shown to hold for CMODELS, and (iii) that –thanks to our equivalence result– can be easily shown to hold also for SMODELS.

From the experimental point of view, we analyze which combinations of reasoning strategies work best on which problems. In particular,

- we extended CMODELS in order to obtain a unique platform with various "look-ahead" strategies (used while descending the search tree); "look-back" strategies (used for recovering from a failure in the search tree); and "heuristic" (used for selecting the next literal to branch on), and
- we considered various combinations of strategies, and conducted an extensive experimental analysis, on a wide variety of tight and non tight programs.

Our experimental results show that:

1. On "small" (i.e., with a few hundreds variables), randomly generated problems, look-ahead solvers (featuring a rather sophisticated look-ahead based on "failed literal", a simple look-back –essentially backtracking– and a heuristic based on the information gleaned during the look-ahead phase) are best.
2. On "large" (i.e., with tens of thousands variables) problems,"look-back" solvers (featuring a simple but efficient look-ahead –essentially unit-propagation with 2 literal watching–, a rather sophisticated look-back based on "learning" and a constant time heuristic based on the information gleaned during the look-back phase), are best.
3. Adding a powerful look-back (resp. look-ahead) to a look-ahead (resp. look-back) solver does not lead to better performances if the resulting solver is run on the small (resp. large) problems that we considered.

Using the terminology in [1], our comparison is "fair" because all the reasoning strategies are realized on a common platform (thus, our experimental evaluation is not biased by the differences due to the quality of the implementation) and is "significant" because our solver implements current state-of-the-art look-ahead/look-back strategies and heuristics.

As discussed in more details in the conclusions, our experimental results have some important consequences both for developers and also for people interested in benchmarking ASP systems. For instance, our results say that we can hardly expect to develop one solver with the best performances on all the categories of problems. As a consequence, (i) developers should focus on specific classes of benchmarks (e.g., on randomly generated programs), and (ii) benchmarking should take into account whether solvers have been designed for specific classes of programs: indeed, it hardly makes sense to run a solver designed for random (resp. large, real-world) programs on large, real-world (resp. random) programs.

The paper is structured as follows. In Section 2 we give the basic definitions. Sections 3 and 4 are devoted to the definition of the algorithms of CMODELS and SMODELS respectively, and that are used in our formal analysis of their computational properties (done in Section 5). Section 6 is dedicated to the experimental analysis of different look-ahead/look-back strategies and heuristics. We end the paper in Section 7 with the conclusions.

2 Basic Definitions

Let P be a set of atoms. If p is a an atom, $\overline{p}$ is the *negation* of p, and $\overline{\overline{p}}$ is p. We will also use the logical symbols $\bot$ and $\top$ (standing for FALSE and TRUE respectively), and assume that $\overline{\bot} = \top$ and $\overline{\top} = \bot$. Atoms, their negations, and the symbols $\bot$, $\top$ form the set of *literals*. If S is a set of literals, we define $\overline{S} = \{\overline{l} : l \in S\}$.

A *rule* is an expression of the form

$$p_0 \leftarrow p_1, \ldots, p_m, \overline{p}_{m+1}, \ldots, \overline{p}_n \tag{1}$$

where $p_0 \in P \cup \{\bot\}$, and $\{p_1, \ldots, p_n\} \subseteq P$ $(0 \leq m \leq n)$. If r is a rule (1), *head*$(r) = p_0$ is the *head* of r, and *body*$(r) = \{p_1, \ldots, p_m, \overline{p}_{m+1}, \ldots, \overline{p}_n\}$ is the *body* of r. A *(logic) program* is a finite set of rules.

Consider a program Π, and let X be a set of atoms. In order to give the definition of an answer set we consider first the special case in which the body of each rule in Π contains only atoms (i.e., for each rule (1) in Π, $m = n$). Under these assumptions, we say that

- X is *closed* under Π if for every rule (1) in Π, $p_0 \in X$ whenever $\{p_1, \ldots, p_m\} \subseteq X$, and that
- X is an *answer set* for Π if X is the smallest set closed under Π.

Now we consider the case in which Π is an arbitrary program. The *reduct* Π^X of Π relative to X is the set of rules

$$p_0 \leftarrow p_1, \ldots, p_m$$

for all rules (1) in Π such that $X \cap \{p_{m+1}, \ldots, p_n\} = \emptyset$. X is an *answer set* for Π if X is an answer set for Π^X.

In the following, we say that a program Π is *tight* if there exists a function λ from atoms to ordinals such that, for every rule (1) in Π whose head is not $\bot$, $\lambda(p_0) > \lambda(p_i)$ for each $i = 1, \ldots, m$.

3 CMODELS

CMODELS reduces the problem of answer set computation to the satisfiability problem of propositional formulas via Clark's completion, and uses a SAT solver as search engine. Formally, a *clause* is a finite set of literals different from $\bot, \top$, and a *(propositional) formula* is a finite set of clauses. An *assignment* is a set of literals. An assignment S *satisfies* a formula Γ if S is consistent and for each clause C in Γ, $C \cap S \neq \emptyset$. If S satisfies Γ then we also say that S is a *model* of Γ and that Γ is *satisfiable*.

There are various versions of CMODELS (see the web page of CMODELS). Here we consider the one proposed in [2] (called ASP-SAT in that paper), and it is represented in Figure 1, in which

- Π is the input program; Γ is a set of clauses; S is an assignment; p and l are an atom and a literal respectively.
- *lp2sat*(Π) is the set of clauses –corresponding to the Clark's completion of Π– formally defined below.
- *s-assign*(l, Γ) returns the formula obtained from Γ by (i) deleting the clauses $C \in \Gamma$ with $l \in C$, and (ii) deleting $\overline{l}$ from the other clauses in Γ.
- *test*(S, Π) returns TRUE if $S \cap P$ is an answer set of Π, and FALSE otherwise.
- *ChooseLiteral*(S) returns a literal not assigned by S. We say that a *literal* l *is assigned by an assignment* S if $\{l, \overline{l}\} \cap S \neq \emptyset$. For simplicity, we assume that *ChooseLiteral*(S) returns the first –according to a fixed total order ρ on $P \cup \overline{P}$– literal in $P \cup \overline{P}$ which is unassigned by S.

We assume that parameters are passed to a procedure by value, as in [3].

function CMODELS(Π) **return** DLL-REC(*lp2sat*(Π),$\emptyset$,Π);

function DLL-REC(Γ,S,Π)
1 $\langle \Gamma, S \rangle$:= *unit-propagate*(Γ, S);
2 **if** ($\emptyset \in \Gamma$) **return** FALSE;
3 **if** ($\Gamma = \emptyset$) **return** *test*(S,Π);
4 l := *ChooseLiteral*(S);
5 **return** DLL-REC(*s-assign*$(l, \Gamma)), S \cup \{l\}, \Pi)$ **or**
6 DLL-REC(*s-assign*$(\overline{l}, \Gamma)), S \cup \{\overline{l}\}, \Pi)$;

function *unit-propagate*(Γ,S)
7 **if** ($\{l\} \in \Gamma$) **return** *unit-propagate*(*s-assign*$(l, \Gamma), S \cup \{l\})$;
8 **return** $\langle \Gamma, S \rangle$;

Fig. 1. The algorithm of CMODELS

CMODELS(Π) simply invokes DLL-REC(*lp2sat*(Π),$\emptyset$,Π). It is easy to see that DLL-REC(Γ,S,Π) is a variation of the standard DLL procedure. In particular, at line 3, instead of just returning TRUE as in the standard DLL (meaning that the input set of clauses is satisfiable), it invokes *test*(S, Π) (see [2] for more details): such a modification is needed only if the input program Π is non tight. Indeed, if Π is tight we are guaranteed that any model of *lp2sat*(Π) corresponds to an answer set of Π [4], and thus SAT solvers can be used as black-box (as it is the case for some versions of CMODELS).

In order to precisely define *lp2sat*(Π) we need the following definitions. If p_0 is an atom, the *translation of* Π *relative to* p_0, denoted with *lp2sat*(Π, p_0), consists of

1. for each rule $r \in \Pi$ of the form (1) and whose head is p_0, the clauses:

$$\{p_0, \overline{n}_r\},$$
$$\{n_r, \overline{p}_1, \ldots, \overline{p}_m, p_{m+1}, \ldots, p_n\},$$
$$\{\overline{n}_r, p_1\}, \ldots, \{\overline{n}_r, p_m\}, \{\overline{n}_r, \overline{p}_{m+1}\}, \ldots, \{\overline{n}_r, \overline{p}_n\},$$

   where n_r is a newly introduced atom, and
2. the clause $\{\overline{p}_0, n_{r_1}, \ldots, n_{r_q}\}$ where $n_{r_1}, \ldots, n_{r_q}$ $(q \geq 0)$ are the new symbols introduced in the previous step.

The *translation of* Π *relative to* $\bot$, denoted with *lp2sat*($\Pi, \bot$), consists of a clause

$$\{\overline{p}_1, \ldots, \overline{p}_m, p_{m+1}, \ldots, p_n\},$$

one for each rule in Π of the form (1) with head $\bot$. Finally, the *translation of* Π, denoted with *lp2sat*(Π), is $\cup_{p \in P \cup \{\bot\}}$*lp2sat*($\Pi, p$).

Proposition 1. *Let* CMODELS *be the procedure in Figure 1. For each program* Π, CMODELS(Π) *returns* TRUE *if* Π *has an answer set, and* FALSE *otherwise.*

A few remarks are in order:

1. As we said, there are various versions of CMODELS. However, if the input program Π is tight, all the versions are equivalent at the algorithmic level. In other words, the presentation of CMODELS in Figure 1 can be considered as representative for all the various versions of CMODELS, in the case of tight programs.
2. Figure 1 is indeed a simple presentation of CMODELS. CMODELS incorporates, e.g., a pre-processing for the simplification of the input program. Analogously, DLL-REC is based on the standard simple recursive presentation of DLL: actual SAT solvers (including the ones used by CMODELS) feature far more sophisticated look-ahead/look-back strategies and heuristics.
3. Given a program Π, its translation *lp2sat*(Π) to SAT is exactly the one used by CMODELS (see [5]).

Considering other ASP systems, ASSAT also computes a set Γ of clauses corresponding to the Clark's completion of the input program Π, and then invokes a SAT solver on Γ. Assuming that Γ is computed as *lp2sat*(Π), ASSAT and CMODELS have different behavior only if Π is non tight.[2]

[2] Unfortunately, for ASSAT the way a program Π is converted into a set of clauses is not specified (see [6]).

4 SMODELS

Given a program Π, SMODELS searches for answer sets by extending an assignment S till either S becomes inconsistent (in which case backtracking occurs) or each atom is assigned by S (in which case $S \cap P$ is an answer set). A simple, recursive presentation of SMODELS is given in Figure 2, where

- Π is a program; S is an assignment; p is an atom; r is a rule; and l is a literal.
- *p-elim*(S, Π) returns the program obtained from Π by eliminating the rules $r \in \Pi$ such that for some literal $l \in S$, $\overline{l} \in body(r)$. For simplicity, when S is a singleton $\{l\}$, we write *p-elim*(l, Π) for *p-elim*$(\{l\}, \Pi)$.

function SMODELS(Π) **return** SMODELS-REC($\Pi, \{\top\}$);

function SMODELS-REC(Π,S)
1 $\langle \Pi, S \rangle$:= *expand*(Π, S);
2 **if** $(\{l, \overline{l}\} \subseteq S)$ **return** FALSE;
3 **if** $(\{p : p \in P, \{p, \overline{p}\} \cap S \neq \emptyset\} = P)$ **return** TRUE;
4 p := *ChooseLiteral*(S);
5 **return** SMODELS-REC(*p-elim*$(p, \Pi)), S \cup \{p\})$ **or**
6 SMODELS-REC(*p-elim*$(\overline{p}, \Pi)), S \cup \{\overline{p}\})$;

function *expand*(Π,S)
7 $S' := S$;
8 S := *AtLeast*(Π, S);
9 Π := *p-elim*(S, Π);
10 $S := S \cup \{\overline{p} : p \in P, p \notin AtMost(\Pi^{\emptyset}, S)\}$;
11 Π := *p-elim*(S, Π);
12 **if** $(S \neq S')$ **return** *expand*(Π,S);
13 **return** $\langle \Pi, S \rangle$;

function *AtLeast*(Π,S)
14 **if** $(r \in \Pi$ **and** $body(r) \subseteq S$ **and** $head(r) \notin S)$
return *AtLeast*(*p-elim*$(head(r), \Pi), S \cup \{head(r)\})$;
15 **if** $(\{p, \overline{p}\} \cap S = \emptyset$ **and** $\nexists r \in \Pi : head(r) = p)$
return *AtLeast*(*p-elim*$(\overline{p}, \Pi), S \cup \{\overline{p}\})$;
16 **if** $(r \in \Pi$ **and** $head(r) \in S$ **and** $body(r) \not\subseteq S$ **and** $\nexists r' \in \Pi, r' \neq r : head(r') = head(r))$
return *AtLeast*(*p-elim*$(body(r), \Pi), S \cup body(r))$;
17 **if** $(r \in \Pi$ **and** $\overline{head(r)} \in S$ **and** $body(r) \setminus S = \{l\})$
return *AtLeast*(*p-elim*$(\overline{l}, \Pi)), S \cup \{\overline{l}\})$;
18 **return** S;

function *AtMost*(Π,S)
19 **if** $(r \in \Pi$ **and** $body(r) \subseteq S$ **and** $head(r) \notin S)$
return *AtMost*$(\Pi, S \cup \{head(r)\})$;
20 **return** S;

Fig. 2. The algorithm of SMODELS

– *ChooseLiteral*(S) is the same function used by CMODELS at line 4 in Figure 1. Thus, our presentation of CMODELS and SMODELS incorporates the assumption that the two systems use the same heuristic.

The computation of SMODELS-REC(Π, S) proceeds as follows (in the following, we say that a set of atoms X *extends an assignment* S if $S \cap P \subseteq X$ and $\overline{S} \cap X = \emptyset$):

– *Line 1:* The program Π is simplified and the assignment S is extended by the routine *expand*(Π, S), explained below.
– *Line 2:* if S is inconsistent, no answer set extending S exists, and FALSE is returned,
– *Line 3:* if each atom $p \in P$ is assigned, then (i) $S \cap P$ is an answer set of the initial program, and (ii) TRUE is returned.
– *Lines 4-6:* if none of the above applies, an atom p is selected (line 4), an answer set extending $S \cup \{p\}$ (line 5) or $S \cup \{\overline{p}\}$ (line 6) is searched.

expand(Π, S) extends the assignment S generated so far by recursively invoking *AtLeast* (line 8) and then *AtMost* (line 10) till it is no longer possible to extend S (lines 12- 13). *AtLeast* encodes the following facts:

– *Line 14:* if there exists a rule r whose body is a subset of S, then every answer set extending S includes the head of r.
– *Line 15:* if an unassigned atom p is not the head of any rule, then every answer set extending S does not include p.
– *Line 16:* if there is only one rule with head p, and $p \in S$, then each answer set extending S, also extends $S \cup$ *body*(r).
– *Line 17:* if there is a rule with head p and whose body contains only one literal l which is not in S, then if $\overline{p}$ is in S, then every answer set extending S also extends $S \cup \{\overline{l}\}$.

When no further simplification is possible, (i) the set S is returned by *AtLeast*(Π, S) (line 18); (ii) the program Π is simplified accordingly (line 9); and (iii) *AtMost* is invoked with $\Pi^{\emptyset}$ –the reduct of Π relative to the empty set– and S as arguments (line 10). *AtMost* incrementally adds to (the local copy of) S the heads of the rules in $\Pi^{\emptyset}$ whose body is a subset of S (line 19). If S' is the set returned by *AtMost*$(\Pi^{\emptyset}, S)$ (i.e., if S' is the set returned at line 20), if an atom p does not belong to S' then $\overline{p}$ can be safely added to the current assignment S (line 10) (see [7] for more details). To get an intuition of why this is the case, assume for simplicity that the head of each rule in Π is not $\bot$:

1. $\Pi^{\emptyset}$ has a unique answer set, and
2. any answer set of Π which extends S has to be a subset of S union the answer set of $\Pi^{\emptyset}$.

Proposition 2. *Let* SMODELS *be the procedure in Figure 2. For each program* Π, SMODELS(Π) *returns* TRUE *if* Π *has an answer set, and* FALSE *otherwise.*

The above presentation of SMODELS is a recursive reformulation of the description of SMODELS provided in [7], pag. 17. As for CMODELS, the actual implementation of SMODELS features more complex look-ahead/look-back strategies and heuristic. SMODELS has been extended with clause learning in [8], and SMODELS-CC is the name given to the resulting system.

5 Relating CMODELS and SMODELS

Consider a program Π. Our goal is to prove that the computations of CMODELS and SMODELS are tightly related if Π is tight, and that this is not necessarily the case otherwise. To this end, we will compare the search trees of CMODELS and SMODELS on Π, i.e., the search trees of SMODELS-REC$(\Pi, \{\top\})$ and DLL-REC$(\mathit{lp2sat}(\Pi), \emptyset, \Pi)$ respectively. In doing this, the first problem is that the translation *lp2sat* introduces additional atoms not in P. In the following we assume that both SMODELS-REC and DLL-REC operate in the signature of the input program and formula respectively. However, we still assume that *ChooseLiteral*(S) returns the first literal in $P \cup \overline{P}$ which is unassigned by S: notice that once all the atoms in P are assigned, also the atoms introduced by *lp2sat* will be assigned by *unit-propagate* in DLL-REC.

Given this, one possibility for achieving our goal would be to consider the search trees corresponding to the assignments generated by the two procedures, and try to prove that they are the same. However, this is not the case:

- *lp2sat* introduces additional atoms not in P and also these atoms get assigned, and
- The order followed by *expand* and *unit-propagate* to assign literals may differ.

However, if we do not take into account the above differences, we have that the two procedures generate the "same" search tree. In order to formally state this result we introduce the following definitions.

We say that a set of literals S is a *branching node of* SMODELS(Π) (resp. *of* CMODELS(Π)) if there is a call to SMODELS-REC(Π', S) (resp. DLL-REC(Γ', S, Π)), following the invocation of SMODELS(Π) (resp. CMODELS(Π)). If $proc$ is SMODELS(Π) or CMODELS(Π), we define

$$\mathit{Branches}(proc) = \{S \cap (P \cup \overline{P}) : S \text{ is a branching node of } proc\}.$$

Finally, we say that SMODELS(Π) and CMODELS(Π) are *equivalent* if

$$\mathit{Branches}(\text{SMODELS}(\Pi)) = \mathit{Branches}(\text{CMODELS}(\Pi)).$$

Theorem 3. *Let* CMODELS *and* SMODELS *be the procedures in Figures 1 and 2 respectively. For each tight program* Π, CMODELS(Π) *and* SMODELS(Π) *are equivalent.*

The idea underlying the proof is that the atoms in P assigned by *expand* in SMODELS-REC correspond exactly to those assigned by *unit-propagate* in DLL-REC, and vice-versa. Indeed, for such result to hold, it is essential that *lp2sat*() is defined as in Section 3.

Theorem 3 states a strong relation between SMODELS and CMODELS, and, ultimately, between SMODELS and DLL: to a certain extent, SMODELS() and DLL(*lp2sat*()) are the same procedure on tight programs. Further, the results hold independently from the specific heuristic used by SMODELS-REC and DLL-REC, as long as they are guaranteed to return the same literal at every point of the two search trees. Because of this, similar results would hold if we enhance SMODELS-REC and DLL-REC with more powerful look-ahead techniques based on *expand* and *unit-propagate* respectively. For instance, SMODELS has been enhanced with the following check, performed before each branch:

for every unassigned literal l in the program, check whether assigning l would "fail", i.e., if *expand*(*p-elim*$(l, \Pi), S \cup \{l\}$) returns (as second argument) an inconsistent set of literals. If this is the case, we can safely assign $\overline{l}$ before branching. However, if l fails, then also branching on l would fail, and the tree generated by SMODELS-REC extended with such "failed literal" strategy corresponds to the tree generated by SMODELS-REC with a specific heuristic. Using the same heuristic in DLL-REC (i.e., using a similar "failed literal" strategy based on *unit-propagate*) would lead to an equivalent search tree.

The established correspondence between CMODELS and SMODELS gives us the possibility to derive lower/upper bounds and average case results for CMODELS and SMODELS. Here there are a few.

First, observe that the search tree explored by CMODELS and SMODELS when run on a program Π, critically depends on the specific heuristic used, i.e., in our terminology and with reference to Figures 1 and 2, by the fixed total ordering ρ on the set $P \cup \overline{P}$ used by *ChooseLiteral*(S). In order to highlight the dependency from ρ, we now write *Branches*$^{\rho}$(SMODELS(Π)) (resp. *Branches*$^{\rho}$(CMODELS(Π))) to indicate the set of branching nodes of SMODELS (resp. CMODELS) when run on a program Π, assuming that ρ is the total order on the set $P \cup \overline{P}$ used by *ChooseLiteral*(S). We are now ready to define the *complexity of* SMODELS *on a program* Π as the smallest number in

$$\{|\textit{Branches}^{\rho}(\text{SMODELS}(\Pi))| : \rho \text{ is a total order on } P \cup \overline{P}\}.$$

Analogously, the *complexity of* CMODELS *on a program* Π is the smallest number in

$$\{|\textit{Branches}^{\rho}(\text{CMODELS}(\Pi))| : \rho \text{ is a total order on } P \cup \overline{P}\}.$$

Intuitively, the complexity of SMODELS (resp. CMODELS) on Π is the minimum number of branching nodes that SMODELS (resp. CMODELS) has to explore for solving Π.

Consider the formula PHP_n^m $(n \geq 0, m \geq 0)$ consisting of the clauses

$$\begin{array}{ll} \{p_{i,1}, p_{i,2}, \ldots, p_{i,n}\} & (i \leq m), \\ \{\overline{p}_{i,k}, \overline{p}_{j,k}\} & (i, j \leq m, k \leq n, i \neq j). \end{array}$$

The formulas PHP_n^m are from [9] and encode the pigeonhole principle. If $n < m$, PHP_n^m are unsatisfiable and it is well known that any procedure based on resolution (like DLL) has an exponential behavior. Here we state a similar result for CMODELS and SMODELS. First, if C is a clause $\{l_1, \ldots, l_l\}$ $(l \geq 0)$ we define *sat2tlp*(C) to be the rule $\bot \leftarrow \overline{l}_1, \ldots, \overline{l}_l$. Then, if Γ is a formula, the *translation of* Γ, denoted with *sat2tlp*(Γ), is $\cup_{C \in \Gamma}$*sat2tlp*$(C) \cup \cup_{p \in P}\{p \leftarrow \overline{p'}, p' \leftarrow \overline{p}\}$, where, for each atom $p \in P$, p' is a new atom associated to p. For each n, *sat2tlp*(PHP_{n-1}^n) is tight and has no answer sets.

Corollary 4. *The complexity of* SMODELS *and* CMODELS *on sat2tlp*(PHP_{n-1}^n) *is exponential in* n.

The above result can be easily proved for CMODELS starting from [9]. For SMODELS, it relies on the fact that *sat2tlp*(PHP_{n-1}^n) is tight, and thus on such programs SMODELS and CMODELS are equivalent. The pigeonhole formulas give us the opportunity to define a class of formulas which are exponentially hard for CMODELS but easy for SMODELS. For each formula Γ, define *sat2nlp*(Γ) to be the program $\cup_{C \in \Gamma}$*sat2tlp*$(C) \cup \cup_{p \in P}\{p \leftarrow p\}$.

Corollary 5. *The complexity of* SMODELS *and* CMODELS *on sat2nlp*(PHP^n_{n-1}) *is* 0 *and exponential in* n *respectively.*

In this case, *sat2nlp*(PHP^n_{n-1}) is non tight, and SMODELS can determine the non existence of answer sets without branching mainly thanks to the procedure *AtMost*.[3] To see why this is the case, notice that, if $\Pi =$ *sat2nlp*(PHP^n_{n-1}), then

- *AtLeast*$(\Pi, \{\top\})$ returns $\{\top\}$,
- $\Pi^\emptyset$ consists of the rules

$$\bot \leftarrow; \quad \bot \leftarrow p_{i,k}, p_{j,k} \ (i,j \leq n, k \leq n-1, i \neq j); \quad p_{i,k} \leftarrow p_{i,k} \ (i \leq n, k \leq n-1)$$

  and thus *AtMost*$(\Pi^\emptyset, \{\top\})$ returns $\{\bot, \top\}$.
- At line 10 in Figure 2, the set S is set to $S' = \overline{P} \cup \{\top\}$, thus causing one more recursive call to *expand*.
- If $\Pi' =$ *p-assign*(S', Π), *AtLeast*(Π', S') returns the set $S' \cup \{\bot\} = \overline{P} \cup \{\bot, \top\}$, and this is also the set returned by *expand*.
- At line 2 in Figure 2, SMODELS returns FALSE, without performing any branch.

Indeed, the above results can be easily generalized to any formula Γ which is known to be exponentially hard for DLL: *sat2tlp*(Γ) will be exponentially hard for both SMODELS and CMODELS, while *sat2nlp*(Γ) will be exponentially hard for CMODELS but easy for SMODELS. We mention one more of such results, because it involves a class of programs that have been frequently used in the literature as a benchmark for ASP systems.

Define a formula Γ to be a *k-cnf* if each clause in Γ consists of k literals. The *random family of k-cnf formulas* is a k-cnf whose clauses have been randomly selected with uniform distribution among all the clauses C of k literals and such that, for each two distinct literals l and l' in C, $\overline{l} \neq l'$.

Corollary 6. *Consider a random k-cnf formula Γ with n atoms and m clauses. With probability tending to one as n tends to infinity, the complexity of* SMODELS *and* CMODELS *on sat2tlp*(Γ) *is exponential in n if the density $d = m/n$ is $d \geq 0.7 \times 2^k$.*

As in the case of Corollary 4, this result is easy to show for CMODELS starting from [10], and then it follows for SMODELS from Theorem 3. Programs corresponding to random k-cnf formulas have been used, e.g., in [11,12,8]. Also notice that since the results in [9] and [10] hold for any proof system based on resolution, enhancing SMODELS and CMODELS with "learning" look-back strategies does not lower the exponential complexity of the procedures. Thus, the above corollaries also hold for SMODELS-CC, and all the different versions of CMODELS. (assuming that CMODELS use a procedure based on DLL as search engine, as it is indeed the case in practice).

Other results that have been proven for DLL can now easily be shown to hold also for SMODELS. Define a literal l as *optimal for a program Π* if there exists a minimal search tree of SMODELS(Π) whose root is labeled with l. The following result echoes the one in [13] for DLL.

[3] In the real implementation of CMODELS, rules like $p \leftarrow p$ will be removed during the preprocessing, and thus the implementation of CMODELS concludes that *sat2nlp*(PHP^n_{n-1}) does not have answer sets without a single branch. However, instead of $p \leftarrow p$, we could have considered, e.g., the two rules $p \leftarrow p'$, $p' \leftarrow p$, (where p' is a newly introduced atom associated to p) and the result in the corollary would still hold.

Corollary 7. *In* SMODELS*, deciding the optimal literal to branch on, is both NP-hard and co-NP hard, and in PSPACE for tight programs.*

There are many other results in the SAT literature studying the proof-complexity of DLL and/or resolution that are applicable also to SMODELS and CMODELS. See, e.g., [14] for a study on the average complexity of coloring randomly generated graphs with DLL, and [15], which derives exponential lower bounds on the running time of DLL on random 3-SAT formulas also for densities significantly below the satisfiability threshold $d \approx 4.23$. The first result applies also to SMODELS and CMODELS when run on a program Π being the standard tight formulation of a graph coloring problem:[4] *lp2sat*(Π) corresponds to the SAT formulation considered in [14]. Analogously for the second result.

6 On the Relation Between AS and SAT Solvers

Given the results established in the previous Section, we can expect that the combinations of reasoning strategies that work best in SAT, should also work best in ASP, at least when considering tight programs. We show that this is indeed the case, also on non tight programs. We now report about an extensive experimental comparison that we have conducted on a wide variety of programs, and using the combinations of reasoning strategies that, along the years, proved to be more effective in SAT. Indeed, current state-of-the-art SAT solvers can be divided in two main categories:

- "look-ahead" solvers, featuring a rather sophisticated look-ahead based on "failed literal", a simple look-back (essentially backtracking) and a heuristic based on the information gleaned during the look-ahead phase. These solvers are best for dealing with "small but relatively difficult" instances, typically random k-cnf formulas. A solver in this category is SATZ [16].
- "look-back" solvers, featuring a simple but efficient look-ahead (essentially unit-propagation with 2 literal watching), a rather sophisticated look-back based on "1-UIP learning" and a constant time heuristic based on the information gleaned during the look-back phase. These solvers are best for dealing with "large but relatively easy" instances, typically encoding real-world problems. A solver in this category is ZCHAFF [17].

The terminology "small but relatively difficult" and "large but relatively easy" refer to the number of variables and are used to convey the basic intuitions about the instances. To get a more precise idea, consider that in the SAT2003 competition, instances in the random and industrial categories had, on average, 442 and 42703 atoms respectively [18]. Given this, the reasoning strategies that we considered are:

- *Look-ahead:* fast unit-propagation based on 2 literal watching (denoted with "u"); and unit-propagation+failed literal (denoted with "f").
- *Look-back:* basic backtracking (denoted with "b"); and backtracking+1-UIP learning from [17] (denoted with "l").

[4] See, e.g., the formulation in `http://www.tcs.hut.fi/~ini/papers/niemela-iclp04-tutorial.ps.gz`.

– *Heuristic:* VSIDS from [17] (denoted with "v"); unit (given an unassigned atom p, while doing failed literal on p we count the number $u(p)$ of unit-propagation caused, and then we select the atom with maximum $1024u(p) \times u(\overline{p}) + u(p) + u(\overline{p})$. This heuristic is denoted with "u").

The above search strategies and heuristics are not novel: they have been already presented and implemented in the literature. For example, failed literal is already incorporated in SMODELS, and the heuristic of SMODELS-CC is similar to VSIDS. However, here, for the first time, all these techniques are implemented, combined and analyzed in a common platform.

We considered 4 combinations of reasoning strategies: ulv, flv, flu and fbu, where the first, second and third letter denote the look-ahead, look-back and heuristic respectively, used in the combination. ulv is a standard look-back, "ZCHAFF"-like, solver, similar to SMODELS-CC and CMODELS2. fbu is a look-ahead, "SATZ"-like, solver. flv and flu have both a powerful look-ahead and look-back but different heuristic. As we already anticipated, we can expect that ulv (resp. fbu) has good performances on "large but relatively easy" (resp. "small but relatively difficult") programs. By comparing flv with ulv (resp. flu with fbu) we will see under which conditions a more powerful look-ahead (resp. look-back) leads to better performances. Also notice that the 4 combinations of reasoning strategies that we consider, are the only meaningful. Indeed, the "v" heuristic requires learning, while the "u" heuristic requires that failed literal is enabled.

All the tests were run on a Pentium IV PC, with 2.8GHz processor, 1024MB RAM, running Linux. The timeout has been set to 600 seconds of CPU time for random problems, and to 3600 seconds for real-world problems. In order to have our results not biased by the differences due to the quality of the implementation, we implemented all the reasoning strategies in CMODELS ver. 2 [2]. CMODELS ver. 2, besides being the solver that we knew best, had the following features:

– Its front-end is LPARSE [7], a widely used grounder for logic programs.
– Its back-end solver already incorporates lazy data structures for fast unit-propagation as well as some state-of-the-art strategies and heuristics evaluated in the paper; and
– Can be also run on non-tight programs.

There is no other publicly available AS system having the above features, and that we know of. In particular, SMODELS does not contain lazy data structures, and adding them to SMODELS would basically boil down to re-implement the entire solver. Though our analysis has been conducted using CMODELS ver. 2, thanks to the equivalence result established in Theorem 3, analogous results are to be expected for any system based on SMODELS and implementing the techniques that we consider.

Table 1 shows the results on "small" randomly generated programs (lines 1-3, 10-12), and "large" non random programs (lines 4-6, 7-9, 13-15). More in details,

1. Benchmarks (1-3) are tight programs being the translation of randomly generated 3-SAT instances with a ratio of clauses to atoms as in the column "PB". They have been used in [11,12,8].
2. Benchmarks (4-6) and (7-9) are tight programs encoding blocks world planning problems and 4-colorability graph problems, respectively. These benchmarks are

Table 1. Performances on tight (1-9)and non-tight (10-15) problems. For each row, the best result is in bold, and the results within a factor of 2 from the best, are underlined.

	PB	#VAR	ulv	flv	flu	fbu
1	4	300	**0.41**	0.52	0.85	0.66
2	4.5	300	TIME	TIME	81.92	**22.53**
3	5	300	448.21	485.36	8.27	**4.72**
4	bw*d9	9956	**1.02**	5.84	2.69	2.75
5	bw*e9	12260	**0.98**	1.91	1.92	1.93
6	bw*e10	13482	**1.29**	7.51	5.03	4.95
7	p1000	14955	**0.48**	37.86	15.41	15.23
8	p3000	44961	**8.86**	369.27	144.12	142.83
9	p6000	89951	**99.50**	TIME	583.55	578.98
10	4	300	265.43	218.48	41.97	**31.05**
11	5	300	TIME	TIME	136.67	**99.75**
12	6	300	TIME	TIME	107.34	**65.83**
13	np60c	10742	**2.83**	1611.32	44.12	44.12
14	np70c	14632	**4.69**	TIME	97.44	97.89
15	np80c	19122	**6.91**	TIME	192.29	196.32

publicly available at `http://www.tcs.hut.fi/Software/smodels/tests/`.

3. Benchmarks (10-12) are non tight programs, randomly generated according to the methodology proposed in [19]. As before, the number in the column "PB" is the ratio of clauses to atoms.
4. Benchmarks (13-15) are non tight programs encoding Hamiltonian Circuit problems on complete graphs. The encoding is from [20].

For the randomly generated programs, for each ratio, we generated 10 instances and show the median results. In each row, #VAR represents the number of atoms in the instance.

The first observation is that we get the results that we expected, (except for the results on the first row, where the positive results of ulv are due to the relative simplicity of the problems): on "small but relatively difficult" programs fbu is best, while on "large but relatively easy" programs ulv is best. The second observation is that adding failed literal (resp. learning) to ulv (resp. fbu) does not improve performances when considering the "large" (resp. "small") programs.

We also considered other classes of programs, both non large and non randomly generated. For these programs, the situation of which reasoning strategy is best is less clear, and (as it can be expected) it varies from class to class.

7 Conclusions

We studied the relation existing between SMODELS and CMODELS, and, ultimately, between AS and SAT solvers. From a theoretical point of view, we proved that the two

systems have the same behavior on tight programs. Given that CMODELS is based on DLL, our equivalence results allow to easily derive many other interesting properties about the two procedures, and in particular about SMODELS. We also conducted an extensive experimental analysis showing that the combination of reasoning strategies that are best in SAT, are also best in ASP on randomly generated or on large real world problems.

We believe that our paper is particularly important for ASP researchers who are interested in formally establishing the computational behavior of systems, but also for developers and, more in general, for people involved in benchmarking ASP systems. In particular, for developers, our theoretical results should foster the design of systems incorporating reasoning strategies that provably allow to easily solve problems otherwise exponential: in SAT, this led to the development, e.g., of ZAP [21]. Further our experimental results suggest that developers (in order to advance the state-of-the-art) should focus either on randomly generated problems (and thus develop a look-ahead solver) or on real-world problems (and thus develop a look-back solver): this already happened in SAT. Finally, the results in this paper are particularly important also to people interested in benchmarking systems (see the recent ASPARAGUS initiative [22]). Our theoretical results tell us, e.g., that there exist classes of programs on which SMODELS and/or CMODELS (but also ASSAT) are *bound to* be exponential. Our conclusive experimental analysis points out that it hardly makes sense to run a solver like SMODELS-CC [8] on randomly generated programs, and, vice-versa, that it hardly makes sense to use CMODELS with SATZ [16] as SAT solver on large problems coming from real-world applications.

Finally, we believe that this paper is a major step in the direction of closing the gap between SAT and ASP, as advocated by Miroslaw Truszczyński in his invited talk at the last NMR workshop in Whistler, Canada.[5]

Acknowledgments

We would like to thank Nicola Leone, Vladimir Lifschitz and Mirek Truszczynski for discussions related to the subject of this paper. This work has been partially supported by MIUR.

References

1. E. Giunchiglia, M. Maratea, A. Tacchella, and D. Zambonin. Evaluating search heuristics and optimization techniques in propositional satisfiability. In *Proc. IJCAR*, 2001.
2. E. Giunchiglia, M. Maratea, and Y. Lierler. SAT-based answer set programming. In *Proc. AAAI*, 2004.
3. T. H. Cormen, C. E. Leiserson,R. L. Rivest and C. Stein Introduction to Algorithms. MIT Press, 2001.
4. François Fages. Consistency of Clark's completion and existence of stable models. *Journal of Methods of Logic in Computer Science*, 1:51–60, 1994.

[5] Slides available at `http://cs.engr.uky.edu/~mirek/stuff/nmr-inv.pdf`.

5. Yu. Babovich and V. Lifschitz. Computing Answer Sets Using Program Completion. Available at `http://www.cs.utexas.edu/users/tag/cmodels/cmodels-1.ps`, 2003.
6. F. Lin and Y. Zhao ASSAT: Computing answer sets of a logic program by SAT solvers. In *Proc. AAAI*, 2002.
7. P. Simons. Extending and implementing the stable model semantics. *PhD Thesis*, 2000.
8. Jeffrey Ward and John S. Schlipf. Answer set programming with clause learning. In *Proc. LPNMR*, 2004.
9. Haken. The intractability of resolution. *TCS*, 39:297-308, 1985.
10. V. Chvátal and E. Szemerédi. Many hard examples for resolution. *J. ACM*, 35(4):759–768, 1988.
11. W. Faber, N. Leone, and G. Pfeifer. Experimenting with heuristics for ASP. In *Proc. IJCAI*, 2001.
12. P. Simons, I. Niemelä, and S. Timo. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1–2):181–234, 2002.
13. Paolo Liberatore. On the complexity of choosing the branching literal in DPLL. *Artificial Intelligence*, 116(1-2):315–326, 2000.
14. Rémi Monasson. On the analysis of backtrack procedures for the coloring of random graphs. In *Complex Networks*, Lecture Notes in Physics, pages 232–251. Springer, 2004.
15. D. Achlioptas, P. Beame, and M. Molloy. A sharp threshold in proof complexity. In *Proc. STOC*, pages 337–346, 2001.
16. Chu Min Li and Anbulagan. Heuristics based on unit propagation for satisfiability problems. In *Proc. IJCAI*, 1997.
17. M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *Proc. DAC*, 2001.
18. D. Le Berre and L. Simon. The essentials of the SAT'03 competition. In *Proc. SAT*, 2003.
19. Fangzhen Lin and Yuting Zhao. ASP phase transition: A study on randomly generated programs. In *Proc. ICLP*, 2003.
20. I. Niemelä Logic programs with stable model semantics as a constraint programming paradigm. In *Annals of Mathematics and Artificial Intelligence*, 25:241–273, 1999.
21. H. Dixon, M. Ginsberg, E. Luks, and A. Parkes. Generalizing Boolean satisfiability II: Theory. *JAIR*, 22:481–534, 2004.
22. P. Borchert, C. Anger, T. Schaub, and M. Truszczynski. Towards systematic benchmarking in answer set programming: The dagstuhl initiative. In *Proc. LPNMR*, 2004.

Towards an Integration of Answer Set and Constraint Solving

S. Baselice[1], P.A. Bonatti[1], and M. Gelfond[2]

[1] Università di Napoli Federico II
[2] Texas Tech University

Abstract. Answer set programming (ASP for short) is a declarative problem solving framework that has been recently attracting the attention of researchers for its expressiveness and for its well-engineered and optimized implementations. Still, state-of-the-art answer set solvers have huge memory requirements, because the ground instantiation of the input program must be computed before the actual reasoning starts. This prevents ASP to be effective on several classes of problems. In this paper we integrate answer set generation and constraint solving to reduce the memory requirements for a class of multi-sorted *logic programs with cardinality constraints*. We prove some theoretical results, introduce a provably sound and complete algorithm, and report experimental results showing that our approach can solve problem instances with significantly larger domains.

1 Introduction

Nonmonotonic reasoning was initially introduced for commonsense reasoning and reasoning about action and change [15,19,16]. It was later applied to model a variety of combinatorial problems, where nonmonotonic logics proved to be powerful representation formalisms [5]. One of the most promising results in this respect, is a declarative problem solving framework called *answer set programming* (ASP for short), with well-engineered and optimized implementations [14,17,7]. The most popular ASP languages are basically extensions of function-free logic programs (a.k.a. Datalog) where negation as failure is interpreted according to the *stable model semantics* [9,10]. From the expressiveness point of view, ASP languages are able to encode efficiently and uniformly all search problems within the first two levels of the polynomial hierarchy [13,3]. Moreover, answer set solvers are proving to be competitive with other reasoners on several benchmarks [20], and are being used successfully as planners and plan verifiers in the RCS/USA Advisor system [1,18], a decision support system for NASA shuttle controllers (`http://krlab.cs.ttu.edu/~marcy/RCS/`).

Still, state-of-the-art answer set solvers have a major limitation: they use huge amounts of memory, because the ground instantiation of the input program must be computed before the actual reasoning starts. This problem is mitigated to some extent through intelligent grounding techniques that partially evaluate program rules when possible, thereby deleting some rule instances that are surely not applicable. However, this technique is not effective enough on some classes of programs, including several programs for reasoning about actions and change.

In this paper we integrate answer set generation and constraint solving to reduce the memory requirements for a class of multi-sorted *logic programs with cardinality con-*

M. Gabbrielli and G. Gupta (Eds.): ICLP 2005, LNCS 3668, pp. 52–66, 2005.

straints [20] whose signature can be partitioned into: (i) a set of so-called *regular predicates* over domains whose size can be handled by a standard answer set solver; (ii) a set of *constrained* predicates that can be handled by a constraint solver in a way that does not require grounding (so larger domains can be allowed here); (iii) a set of predicates—called *mixed predicates*—that create a "bridge" between the above two partitions.

Then reasoning can be implemented by having an answer set solver interact with a constraint solver. A critical aspect is the form that the definitions of mixed predicates may take. If they were completely general, then that part of the program would be just as hard to reason with as unrestricted programs because mixed predicates may range over arbitrary domains. Accordingly, the framework introduced in this paper supports restricted definitions for mixed predicates, that can be either functions from "regular" to "large" domains (*strong semantics*) or slightly weaker mappings where each combination of "regular" values must be associated to at least one vector of values from "large" domains (*weak semantics*).

An integration of ASP and constraint solving has been proposed in [8]. The architecture adopted there starts with a complete grounding phase, just like standard ASP systems, therefore it does not address the problems we are tackling here. In [8] rules are completely general, so there seems to be no opportunity for avoiding a complete grounding.

In the following, we study the relationships between strong and weak semantics, and introduce an algorithm for computing the strong semantics efficiently under the simplifying assumption that mixed predicates do not occur in the scope of negation. Moreover, we report experimental results providing preliminary evidence that our approach can solve problem instances with significantly larger domains. In this first paper we focus only on the comparison with a standard answer set programming approach.

The paper is organized as follows. The next section is devoted to preliminaries. Then, in Section 3, we introduce the class of programs we deal with, and prove some of their theoretical properties. The algorithm for reasoning on these programs is described and proved to be correct and complete in Section 4. Section 6 reports the experiments and Section 7 concludes the paper with a final discussion and possible directions for future work.

2 Preliminaries

We adopt a sorted first-order language based on a given signature Σ. Letters x, y, z range over variables, a, b, c range over constant symbols, letters f, g, h over function symbols, and letters p, q, r over predicate symbols. Let $\mathcal{S}$ be a finite set of *sorts*.[1] and assume a *sort specification* is given, that is, a function sort mapping:

- each constant c onto a set $\mathsf{sort}(c) \subseteq \mathcal{S}$;
- each variable x onto a (single) sort $\mathsf{sort}(x) \in \mathcal{S}$;

[1] Sorts are just atomic entities, basically type *names*. Sometimes, we shall abuse terminology and say that a sort s *contains* a term t when the sort of t according to the function sort defined below is s.

- each n-ary function symbol f onto a tuple $\mathsf{sort}(f) = \langle S_1, \ldots, S_{n+1} \rangle \in \mathcal{S}^{n+1}$;
- each n-ary predicate symbol p onto a tuple $\mathsf{sort}(p) = \langle S_1, \ldots, S_n \rangle \in \mathcal{S}^n$.

Note that sorts may overlap because constants may be associated to two or more sorts.

Example 1. A sort *steps*, modelling plan steps, may contain the integer constants in the interval $[0, 10]$, while a sort *time*, modelling time points, may contain the integer constants in $[0, 600000]$.

All the other terms have a unique sort. Intuitively, in $\mathsf{sort}(f)$, S_i is the sort of the i-th argument of f ($1 \leq i \leq n$) and S_{n+1} is the sort of the output. Similarly, in $\mathsf{sort}(p)$, S_i is the sort of the i-th argument of predicate p ($1 \leq i \leq n$).

Terms and atoms are defined accordingly. Each variable x with $\mathsf{sort}(x) = S$ and each constant c such that $S \in \mathsf{sort}(c)$ are terms of sort S. Each expression $f(t_1, \ldots, t_n)$ such that $\mathsf{sort}(f) = \langle S_1, \ldots, S_n, S \rangle$ and each t_i is a term of sort S_i is a term of sort S. Nothing else is a term. We write $t : s$ to state that term t has sort s.

All expressions $p(t_1, \ldots, t_n)$ such that $\mathsf{sort}(p) = \langle S_1, \ldots, S_n \rangle$ and each t_i is a term of sort S_i are atoms. Literals are either atoms (positive literals) or expressions of the form `not` A where A is an atom (negative literals).

A variable substitution over $\{x_1, \ldots, x_n\}$ is a function mapping each variable x_i onto a term of $\mathsf{sort}(x_i)$. The notions of instance and ground instantiation are defined as usual from the above notion of (typed) substitution. The ground instantiation of a set of expressions E will be denoted by $\mathsf{ground}(E)$.

Given a logic program P consisting of *normal rules* $A \leftarrow L$ and *constraints* (or *denials*) $\leftarrow L$, where L is a collection of literals, the *stable models* of P [9] are defined as follows.

We first need a notion of program *reduct* P^I, where I is a set of ground atoms. The reduct P^I is obtained from $\mathsf{ground}(P)$ by removing:

- all the rules and constraints with a literal `not` B in their body, s.t. $B \in I$;
- all negative literals from the remaining rules and constraints.

Note that P^I is a set of Horn clauses. Therefore, if P^I is consistent, then it has a unique minimal Herbrand model, that will be denoted by $\mathsf{lm}(P^I)$.

Now I is a *stable model* of P if and only if $I = \mathsf{lm}(P^I)$.

The most popular answer set frameworks are based on the above notions of program and semantics, and extensions thereof. Answer sets are identified with stable models; each answer set represents a possible solution to the given problem instance (programs may have no stable models, as well as multiple stable models). One important extension consists of *cardinality constraints* [20], that in their simplest version are expressions of the form

$$l\{A\}u$$

where A is an atom, l and u are integers. Roughly speaking, $l\{A\}u$ forces the answer sets of the given program to contain a number n of instances of A, such that $l \leq n \leq u$ (u may be omitted in case there is no upper bound). The complete framework is more general. It allows for cardinality constraints in rule bodies and *weight constraints*, that generalize cardinality constraints and allow programmers to express preferences and

optimization criteria on problem solutions. For a general and precise definition of cardinality and weight constraints, the reader is referred to [20]. They are fully supported by SMODELS.

3 Constrained Programs

The sorts of constrained programs are partitioned into *regular* and *constrained* sorts. Intuitively, regular sorts are small enough to be handled by standard answer set solvers, while constrained sorts are large enough to require reasoners that do not instantiate the corresponding variables.

Variables and constants are called *regular* or *constrained* according to their sorts. A function f is regular (resp. constrained) if all the sorts in $\mathsf{sort}(f)$ are regular (resp. constrained). Function f is *mixed* if $\mathsf{sort}(f)$ comprises both regular and constrained sorts. Predicate symbols are classified in a similar way.

In this paper we assume that the output sort of all functions is a constrained sort. The reason is that most answer set solvers do not (yet) support function symbols, while constraint solvers do (functions are typically standard arithmetic functions).

According to the above classification, signature Σ is partitioned into Σ_r, Σ_c and Σ_m, where r, c and m stand for *regular*, *constrained* and *mixed* respectively.

The atoms over Σ_r, Σ_c, and Σ_m are referred to as r-atoms, c-atoms, and m-atoms respectively. Similarly for literals. The parameters of an m-atom whose sorts are constrained (regular) will be often referred to as c-parameters (r-parameters).

We assume that c-predicates have a predefined interpretation, and that the equality predicate is a c-predicate. The intended interpretation of c-predicates will be represented by a set of ground atoms M_c (the set of all true ground c-atoms).

Regular predicates can be defined with normal programs, as in standard ASP. The definitions of mixed predicates are restricted, instead. Let an atom be *free* if its arguments are all pairwise distinct variables. For all free atoms A we write $A(\boldsymbol{x}_r, \boldsymbol{x}_c)$ to state that the r-variables (resp. c-variables) of A are those in $\boldsymbol{x}_r$ (resp. $\boldsymbol{x}_c$). We denote with $A(\boldsymbol{a}, \boldsymbol{b})$ the instance of A such that $\boldsymbol{x}_r$ is replaced by $\boldsymbol{a}$ and $\boldsymbol{x}_c$ with $\boldsymbol{b}$.

In this paper we deal with two possible semantics of mixed predicates.[2] Under the *weak semantics*, for all free mixed atoms $A(\boldsymbol{x}_r, \boldsymbol{x}_c)$ there is an implicit axiom[3]

$$\forall \boldsymbol{x}_r \exists \boldsymbol{x}_c . A(\boldsymbol{x}_r, \boldsymbol{x}_c)\,, \tag{1}$$

that can be expressed by including into the program a cardinality constraint $1\{A(\boldsymbol{a}, \boldsymbol{x}_c)\}$ for each sequence of ground arguments $\boldsymbol{a}$ of the appropriate type and length.[4]

Under the *strong semantics*, for all free mixed atoms $A(\boldsymbol{x}_r, \boldsymbol{x}_c)$ there is an implicit axiom

$$\forall \boldsymbol{x}_r \exists ! \boldsymbol{x}_c . A(\boldsymbol{x}_r, \boldsymbol{x}_c)\,, \tag{2}$$

[2] A more general approach is described in the final discussion.

[3] Note that by our definition of *term*, quantification is implicitly restricted to well-typed terms.

[4] In SMODELS this can be done with a single rule having a cardinality constraint in the head. A similar remark applies to the encoding of (2). We refer the reader to [20] for more details.

that can be encoded in a similar way with a suitable set of cardinality constraints like $1\{A(\boldsymbol{a}, \boldsymbol{x}_c)\}1$.

Moreover, constrained programs may contain constraints that relate all kinds of predicates (regular, constrained, and mixed).

Definition 1.

1. *A* regular rule *(r-rule) is a rule of the form $A \leftarrow B$ or $\leftarrow B$ where A is an r-atom and B is a collection of r-literals.*
2. *A (proper)* constraint *is a rule of the form $\leftarrow B$ where B is a collection of arbitrary literals, including at least one nonregular literal.*
3. *A* constrained program*, P, is the union of a set of regular rules, $R(P)$, and a set of constraints, $C(P)$.*

Example 2. In our running example (a planning and scheduling problem) we have two regular sorts: *step* (representing plan steps) and *action*. We write $step : 0..10$ to state that the constants c with $step \in \mathsf{sort}(c)$ are those in the integer interval $[0, 10]$. Analogously, we may write $action : a_1, \ldots, a_n$ to enumerate all possible actions. The regular signature Σ_r contains only one relation o over $action \times step$. Intuitively, $o(A, S)$ means that action A occurs at step S. The regular part $R(P)$ contains n rules that force at least one action to be executed at each step. For $i = 1, \ldots, n$:

$$o(a_i, S) \leftarrow \mathtt{not}\, o(a_1, S), \ldots, \mathtt{not}\, o(a_{i-1}, S), \mathtt{not}\, o(a_{i+1}, S), \ldots, \mathtt{not}\, o(a_n, S).$$

Moreover, $R(P)$ contains a denial that forbids concurrent actions:

$$\leftarrow o(A, S1), o(A, S2), \mathtt{not}\, eq(S1, S2).$$
$$eq(X, X).$$

The constraint signature Σ_c comprises the sort $time : 0..600000$ with the standard arithmetic functions: $+, -, |\,|$ etc., and relations: $>, \geq$, etc.

The mixed signature Σ_m comprises a relation $time(S, T)$ associating each plan step S to at least one time point T under the weak semantics (exactly one under the strong semantics).

The following constraints $C(P)$ ensure that time is assigned to steps monotonically and that each step is associated to exactly one time point (the latter is needed only under the weak semantics);

$$\leftarrow time(S1, T1), time(S2, T2), S1 < S2, T1 \geq T2.$$
$$\leftarrow time(S, T1), time(S, T2), T1 \neq T2.$$

Moreover, one can specify a minimal duration for each action, e.g., 3 time units for a_1

$$\leftarrow o(a_1, S1), time(S1, T1), o(A2, S2), time(S2, T2), |T2 - T1| < 3\,. \qquad (3)$$

Formally, the semantics of constrained programs is a specialization of the stable model semantics for logic programs with weight constraints, taking into account the intended interpretation M_c of Σ_c and the implicit semantics of mixed predicates.

Definition 2. *A* weak answer set *of a constrained program P is a set of ground atoms $M = M_r \cup M_m$ satisfying the following conditions:*

AS1 *M_r is a set of r-atoms and M_m is a set of m-atoms;*
AS2 *$R(P)^{M_r}$ is consistent and $M_r = \mathsf{lm}(R(P)^{M_r})$;*
AS3 *each constraint $(\leftarrow \boldsymbol{L}) \in \mathsf{ground}(C(P))$ contains a literal L_i false in $M \cup M_c$;*
AS4 *for each free m-atom $A(\boldsymbol{x}_r, \boldsymbol{x}_c)$, and for each vector of r-constants $\boldsymbol{a}$ of the appropriate length, M_m contains at least one instance of $A(\boldsymbol{a}, \boldsymbol{x}_c)$.*

A strong answer set *of a constrained program P is a weak answer set $M = M_r \cup M_m$ satisfying the following additional condition:*

AS5 *for each free m-atom $A(\boldsymbol{x}_r, \boldsymbol{x}_c)$, and for each vector of r-constants $\boldsymbol{a}$ of the appropriate length, M_m contains* at most *one instance of $A(\boldsymbol{a}, \boldsymbol{x}_c)$.*

Note that AS2 basically states that M_r is a stable model of the regular part of P.

Remark 1. We might have alternatively specified the semantics of a constrained program P as the stable models of the program obtained by extending P with M_c and with the cardinality constraints that encode (1) and (2). Then AS1-AS5 might have been proved as theorems. This requires an extension of the *splitting set theorem* [12]. The details have been worked out in [2] and are omitted here due to space limitations.

Theorem 1 (Strong vs. Weak semantics). *Let P be a constrained program in which m-atoms never occur in the scope of negation. For each weak answer set M of P, there exists a strong answer set M' of P such that $M' \subseteq M$ and $M \setminus M'$ is a set of m-atoms.*

Proof. Let M be a weak answer set of P. Then $M = M_r \cup M_m$ is a set of ground atoms and M satisfies the properties AS1, AS2, AS3, AS4.

Let $K_{11}(M)$, …, $K_{1m_1}(M)$, …, $K_{n1}(M)$, …, $K_{nm_n}(M)$ be the subsets of M s.t., for each $1 \leq i \leq n$ and $1 \leq j \leq m_i$, $K_{ij}(M) = \{A_i(\boldsymbol{a}_i^j, \boldsymbol{b}) : A_i(\boldsymbol{a}_i^j, \boldsymbol{b}) \in M_m$ is a ground instance of $A_i(\boldsymbol{a}_i^j, \boldsymbol{x_c})\}$. Note that no $K_{ij}(M)$ is empty because M satisfies the property AS4.

If there exists at least one couple (i, j) ($1 \leq i \leq n$ and $1 \leq j \leq m_i$) s.t. the set $K_{ij}(M)$ has cardinality greater than one, then let $ma \in M$ be a ground m-atom belonging to $K_{ij}(M)$. Note that, by construction, ma must belong to only one of the sets $K_{11}(M)$, …, $K_{1m_1}(M)$, …, $K_{n1}(M)$, …, $K_{nm_n}(M)$. Because ma is not the unique element of $K_{ij}(M)$, then $M' = M \setminus \{ma\}$ must satisfy the property AS4.

Moreover M' satisfies the property AS3. In fact, for each constraint $(\leftarrow \boldsymbol{L}) \in ground(C(P))$, either ma doesn't occur in $\boldsymbol{L}$, and then the value of each L_i is the same in M' than M, or ma occurs in $\boldsymbol{L}$ and so $(\leftarrow \boldsymbol{L})$ contains one more literal false in M' than M because negative m-literals don't occur in $\boldsymbol{L}$.

Moreover M' and M have the same r-literals and then M' satisfies also the properties AS1 and AS2. Then M' is a weak answer set of P as M is.

By iterating the same process starting from M' we can obtain a set M^* s.t. all sets $K_{11}(M^*)$, …, $K_{1m_1}(M^*)$, …, $K_{n1}(M^*)$, …, $K_{nm_n}(M^*)$ contain only one element. By analogy withthe proof for M', M^* is still a weak answer set of P and $M \setminus M^*$ is a set of m-atoms by construction. Note that M^* is a strong answer set of P because it satisfies also property AS5 by construction.

Note that the assumption on negative m-atoms is satisfied by our running example.

Corollary 1. *Under the hypothesis of Theorem 1, the strong answer sets of P are the minimal weak answer sets of P.*

Corollary 2. *Under the hypothesis of Theorem 1, the strong and weak skeptical semantics of P (i.e., the intersection of the strong, resp. weak answer sets) coincide.*

In the light of the above corollaries, we shall focus on the strong semantics, which is a way of computing a "representative" class of answer sets.

4 Computing Strong Answer Sets

In this section we introduce a nondeterministic algorithm for computing strong answer sets. The actual implementation used in the experiments is derived from the nondeterministic algorithm by adding backtracking. The algorithm we introduce can be applied to constrained programs where mixed predicates have only positive occurrences. More general approaches require further work (cf. Section 7).

Our algorithm computes *strong kernels*, that is, compact representations of a (potentially large) set of strong answer sets.

Definition 3.

1. *A* strong completion *of a set of ground atoms I is a set $I \cup J$ such that:*
   - *J is a set of ground m-atoms;*
   - *for each free m-atom $A(\boldsymbol{x}_r, \boldsymbol{x}_c)$ and each vector of r-constants $\boldsymbol{a}$ of the appropriate length, $I \cup J$ contains exactly one instance of $A(\boldsymbol{a}, \boldsymbol{x}_c)$.*
2. *A* strong kernel *of a constrained program P is a set of ground atoms K which has at least one strong completion, and such that all the strong completions of K are strong answer sets of P.*

In general, K is the intersection of exponentially many strong answer sets of P. Since *all* strong completions of K are strong answer sets, it is trivial to generate any particular answer set including K, given K itself (for each mixed atom $A(\boldsymbol{a}, x_c)$ having no instance in K, add one instance choosing x_c arbitrarily).

The algorithm that integrates answer set solving and constraint solving is formulated in terms of a generic answer set solver and a generic constraint solver. The former, called ASGEN, takes as input a regular program P and a set of ground literals S. Intuitively, ASGEN is an incremental solver, and S is the previous partial attempt at constructing an answer set for P. The solver may either fail to further extend S to an answer set of P, or it may return a refined attempt S'. So we assume that ASGEN enjoys of following formal properties:

1. ASGEN(P, S) returns either NULL or a consistent set S' of ground literals.
2. If ASGEN(P, S) returns a set S' then $S \subset S'$.
3. If ASGEN(P, S) returns a complete set S' then S' is an answer set of P; here, by *complete* we mean that each ground literal occurs in S', either positively or negatively.

4. ASGEN is nondeterministically complete, that is for each answer set S of P there exists an integer $n \geq 0$ s.t. at least one computation of $\text{ASGEN}^n(P, \emptyset)$ returns S.

As usually, when we write $\text{ASGEN}^n(P, \emptyset)$ we mean:

$$\text{ASGEN}^0(P, \emptyset) = \emptyset$$
$$\text{ASGEN}^n(P, \emptyset) = \text{ASGEN}(P, \text{ASGEN}^{n-1}(P, \emptyset)).$$

Note that this formulation is compatible with virtually any strategy for interleaving the answer set construction and constraint solving. Note also that as a special case, ASGEN may immediately return complete sets (upon success) like SMODELS.

The only requirements on the constraint solver are that it should be sound and non-deterministically complete for each set of c-clauses χ. In other words, all substitutions σ returned by the constraint solver should be solutions of χ (i.e., $\chi\sigma$ should be satisfiable), and for each solution σ of χ, there should be a computation that returns σ.

The constraint solver is applied to a partially evaluated version of the constraints. To specify the partial evaluation procedure we need some auxiliary notation.

For each constraint $c = \leftarrow B$, we denote by $\mathsf{reg}(c)$, $\mathsf{con}(c)$, and $\mathsf{mix}(c)$, respectively, the collections of regular, constrained and mixed literals belonging to B.

We say that a substitution γ is *r-grounding* iff γ replaces each r-variable with a ground r-term and leaves the other variables unchanged.

Definition 4. *The* partial r-evaluation *of a set of constraints C w.r.t. a set of ground literals S, denoted by $\mathsf{PE}(C, S)$, is defined by*

$$\mathsf{PE}(C, S) = \{(\leftarrow \mathsf{mix}(c), \mathsf{con}(c))\gamma \mid c \in C,\ \gamma \textit{ r-grounding, and } \mathsf{reg}(c)\gamma \subseteq S\}\,.$$

Note that the members of $\mathsf{PE}(C, S)$ contain no r-atoms and no r-variables, because the former have been simplified away and the latter have been replaced with r-constants. Note also that in this process some constraints may disappear, as $\mathsf{reg}(c)$ may match no literals in S. Intuitively, S is to be provided by the answer set solver.

The constraint processing algorithm applies to a *normalized* version of $\mathsf{PE}(C, S)$, denoted by $\mathsf{PE}^n(C, S)$, satisfying the following properties:

N1 No m-literal occurring in $\mathsf{PE}^n(C, S)$ contains two or more occurrences of the same variable;

Moreover, for all free m-atoms $A(\boldsymbol{x}_r, \boldsymbol{x}_c)$,

N2 If both $A(\boldsymbol{a}, \boldsymbol{y}_c)$ and $A(\boldsymbol{a}, \boldsymbol{z}_c)$ occur in $\mathsf{PE}^n(C, S)$, then $\boldsymbol{y}_c = \boldsymbol{z}_c$.
N3 If both $A(\boldsymbol{a}, \boldsymbol{y}_c)$ and $A(\boldsymbol{b}, \boldsymbol{z}_c)$ occur in $\mathsf{PE}^n(C, S)$ and $\boldsymbol{a} \neq \boldsymbol{b}$, then $\boldsymbol{y}_c$ and $\boldsymbol{z}_c$ have no variables in common.

Note that condition N2 is the opposite of the classic standardization apart approach. N2 and N3 together require the vectors of c-variables to be in one-to-one correspondence with the vectors of regular arguments. Condition N1 can be fulfilled by introducing equations $x_i = x_j$ in $\mathsf{con}(c)$ when needed. Condition N2 and N3 can be fulfilled by variable renaming.

Algorithm 1
CASPSOLVER (P)

Inputs: $P = R(P) \cup C(P)$: a constrained program with no negative m-literals.
Outputs: either a strong kernel of P or FAIL
begin
$S := \emptyset$;
loop
  $S := \text{ASGEN}(R(P), S)$;
  if $S = \text{NULL}$ **then**
    FAIL;
  else
    $C := \mathsf{PE}^n(C(P), S)$;
    if $\bigwedge_{c \in C} \neg\mathsf{con}(c)$ has no solution **then**
      FAIL;
    else if S is complete **then**
      choose a solution σ of $\bigwedge_{c \in C} \neg\mathsf{con}(c)$;
      Let $M(C)$ be the set of mixed literals in C;
      return $S \cup M(C)\sigma$;
end

Example 3. In the running example, whenever S contains the pair $o(a_1, 1),\ o(a_i, 2)$, constraint (3) yields the partially evaluated constraint

$$\leftarrow time(1, T1), time(2, T2), |T2 - T1| < 3.$$

After normalization, and assuming this particular constraint has not been modified, for all the atoms $time(1, x)$ occurring in $\mathsf{PE}^n(C(P), S)$, we have $x = T1$. In this way—roughly speaking—any solution to the constraints is forced to fulfil the property (2) of strong semantics.

We are now ready to prove soundness and completeness for Algorithm 1.

Theorem 2. *If a non-failed run of Algorithm 1 returns a set of literals K, then K is a strong kernel of P.*

Proof. Let K be a set returned by a non-failed run of Algorithm 1.

In order to prove that K is a strong kernel of P, we have to prove that for each set of m-atoms J, if $K \cup J$ is a strong completion of K then $K \cup J$ is a strong answer set of P. That is, we need to prove that $K \cup J$ satisfies the properties AS1–AS5, when $K \cup J$ satisfies the properties of the definition 3 of strong completion.

If a run r of the algorithm returns a set K then $K = S \cup M(C)\sigma$ where S is a stable model of $R(P)$ and $M(C)\sigma$ is a set of ground m-atoms. Then $K \cup J = S \cup M(C)\sigma \cup J$ satisfies properties AS1 and AS2.

Suppose that $K \cup J$ does not satisfy property AS3. Then there exists a constraint $c = (\leftarrow \boldsymbol{L}) \in ground(C(P))$ s.t. all literals L_i in $\boldsymbol{L}$ are true in $K \cup J$. There must be a constraint $c' \in C(P)$ and a ground substitution $\gamma = \gamma_r\gamma_c$ of c' s.t. $c = c'\gamma$ and γ_r is r-grounding. Since $\boldsymbol{L}$ is true in $K \cup J$, it holds that $reg(c) \subseteq S$ and hence

$\leftarrow (mix(c'))\gamma_r, (con(c'))\gamma_r \in PE(C(P), S)$. Now, since $con(c) = (con(c'))\gamma = (con(c'))\gamma_r\gamma_c$ and $mix(c) = (mix(c'))\gamma = (mix(c'))\gamma_r\gamma_c$ are true in $K \cup J$, then γ_c is not a solution of $\neg(con(c'))\gamma_r$. Then the solution σ of $\bigwedge_{c\in C} \neg\mathsf{con}(c)$ choosen at the step 14 of the algorithm cannot be factorized in $\sigma = \sigma_1\gamma_c\sigma_2$ (where σ_1 and σ_2 are possibly empty substitutions). Consequently, $mix(c) = (mix(c'))\gamma_r\gamma_c$ cannot be added to K at the step 16, while $(mix(c'))\gamma_r\sigma$ is added to K. Moreover, by hypotheses, $K \cup J$ is a *strong* completion of K, therefore $(mix(c'))\gamma_r\gamma_c$ cannot belong to J, either. So $mix(c)$ is false in $K \cup J$ and this is a contradiction. This proves that $K \cup J$ satisfies property AS3.

By the definition of strong completion, $K \cup J$ satisfies also the properties AS4 and AS5. Consequently $K \cup J$ is a strong answer set of P.

Theorem 3. *For each strong answer set M of P there exists a run of Algorithm 1 that returns a strong kernel $K \subseteq M$.*

Proof. By Definition 2, if M is a strong answer set of P then $M = M_r \cup M_m$ and M satisfies properties AS1–AS5. According to properties AS1 and AS2, M_r is a stable model of $R(P)$. Then, by the properties of $ASGen$, there exists a set of runs, RUN, of the algorithm that execute with success the test at the step 13 on the set M_r. For each $r \in RUN$, if r doesn't return FAIL, then r returns a set $K = M_r \cup M(C)\sigma$ that, by the soundness of the algorithm, is a strong kernel of P.

Now, we must only prove that there always exists an $r \in RUN$ that at the step 14 chooses a solution σ of $\bigwedge_{c\in C} \neg\mathsf{con}(c)$ s.t. $K \subseteq M$. From $M = M_r \cup M_m$ and $K = M_r \cup M(C)\sigma$ follows that $K \subseteq M$ iff $M(C)\sigma \subseteq M_m$.

So we need to prove that there must always exists a solution σ s.t. $M(C)\sigma \subseteq M_m$. If such a substitution σ exists then σ can be nondeterministically computed by a run $r \in RUN$.

Let M_{cfree} be a set of all r-grounded m-atoms of $C(P)$. Then $M_m = M_{cfree}\gamma$ where γ is a ground substitution of M_{cfree} such that for each $A(\boldsymbol{a}, \boldsymbol{x'})$ and $A(\boldsymbol{a}, \boldsymbol{x''})$, $A(\boldsymbol{a}, \boldsymbol{x'})\gamma = A(\boldsymbol{a}, \boldsymbol{x''})\gamma$. It immediately follows that $M(C) \subseteq M_{cfree}$. We can always factorize γ in $\gamma = \sigma\rho$ where σ is a ground substitution of $M(C)$. Then $M(C)\sigma \subseteq M_{cfree}\sigma\rho$, but it must also be proved that σ is a solution of $\bigwedge_{c\in C} \neg\mathsf{con}(c)$.

Suppose that σ is not a solution of $\bigwedge_{c\in C} \neg\mathsf{con}(c)$. Then there exists a constraint $c \in C$ s.t. $(con(c))\sigma$ is true in M. Then $mix(c) \in M(C)$, because $c \in C$, and $mix(c)\sigma \in M_m$, because $M(C) \subseteq M_{cfree}$. By construction of C, there exists a constraint $c' \in ground(C(P))$ s.t. $reg(c') \subseteq S$ and $mix(c') = (mix(c))\sigma$ and $con(c') = (con(c))\sigma$. This implies that the constraint c' is not true in M because its body is true in M, but this is a contradiction because M is a strong answer set of P.

Then there exists a solution σ of $\bigwedge_{c\in C} \neg\mathsf{con}(c)$ s.t. $M(C)\sigma \subseteq M_m$.

5 The CASP Prototype

The CASP prototype is a simplified implementation of Algorithm 1, based on the answer set solver SMODELS [17]. CASP is meant to be an exploratory prototype, built with off-the-shelf components. While this strategy accelerated prototype deployment, it

prevented us from exploiting the potential interleaving of answer set solving and constraint solving, supported by Algorithm 1. In this first prototype, the answer set solver always returns a complete answer set, so the loop in Algorithm 1 makes always one iteration.

Let P be the input program. When P has a strong answer set, CASP returns a strong kernel for P, plus auxiliary information useful for analyzing the behavior of the system including the number of atoms, conjunctions, disjunctions, and variables occurring in $\bigwedge_{c\in C} \neg \mathsf{con}(c)$.

CASP consists of a script CASPSCRIPT that first runs the answer set solver on $R(P)$. Then for each answer set S of $R(P)$, CASPSCRIPT calls a GNU Prolog constraint logic program with finite domains, that implements steps 10-16 of Algorithm 1. In case of failure (step 12), CASPSCRIPT does not always fail; if $R(P)$ has more stable models, CASPSCRIPT feeds the next one to the Prolog module.

The finite domain (FD) constraint solver of GNU Prolog is an instance of the Constraint Logic Programming scheme introduced by Jaffar and Lassez in 1987 [11] and is based on the CLP(FD) framework [6]. Constraints are defined on FD variables and solved by means of arc-consistency (AC) techniques [21]. Arc consistency is not a complete inference mechanism; it ensures only that all solutions (if any) are in the current variable domains. In general, some variable assignments over the current domains are not solutions. Therefore, a final solution generation and checking phase is needed. In many cases, though, the domains produced by arc consistency are tight enough to speed up significantly the computation of solutions.

6 Experimental Results

We experimented with a few variants of the constrained program illustrated in the examples. Of course, this can only be regarded as a preliminary evaluation. Still, the example we choose is of significant interest. Programs similar to our running example have been used in the USA Advisor project, related to NASA missions [1,18], and for protocol verification [4]. In both cases memory requirements happened to cause problems.

We did not insist much on the performance of the answer set solver, because there exists a rich body of literature on experimental evaluations and benchmarking of SMODELS. We focused on the performance of the constraint solver as $\bigwedge_{c\in C} \neg \mathsf{con}(c)$ and the number of disjunctions occurring in it grow.

The tests have been run on a Pentium(R) M processor 1.5GHz, with 1Mb cache and 512Mb core memory.

Recall that the example has two regular sorts, *action* and *step*, and one constrained sort *time*. We started by encoding the planning and scheduling problem as an SMODELS program with weight constraints [20]. In particular, the implicit semantics of mixed predicates has been encoded with the weight constraint

$$1\{\mathtt{time(S,T)} : \mathtt{time(T)}\}1 :- \mathtt{step(S)}. \tag{4}$$

This constraint says that for all steps S there exists exactly one time point T satisfying $\mathtt{time}(S,T)$.

step	*atoms*	*var*	*conj*	*disj*	*attempts*	*Smodels*	*time*
{0,1}	8	2	5	2	1	0m0.016s	0m1.259s
{0,...,2}	19	3	12	6	1	0m0.007s	0m1.363s
{0,...,3}	35	4	22	12	1	0m0.007s	0m3.379s
{0,...,4}	56	5	35	20	1	0m0.012s	0m7.440s
{0,...,5}	82	6	51	30	1	0m0.092s	0m14.091s
{0,...,6}	113	7	70	42	1	0m0.112s	0m24.253s
{0,...,7}	149	8	92	56	1	0m0.165s	0m38.430s
{0,...,8}	190	9	117	72	1	0m0.356s	0m57.580s
{0,...,9}	236	10	145	90	1	0m0.758s	1m22.542s
{0,...,10}	287	11	176	110	1	0m1.309s	1m54.056s
SORT: action={1,2} – time={0,...,6.000.000}							

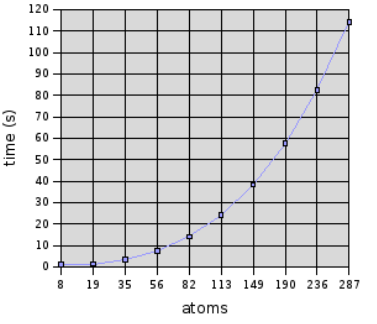


Fig. 1. Test-1 results

Sort *time* is the interval of integers $[0 - 600000]$. These values are determined by the following requirement: scheduling should cover plans at least one week long with the granularity of seconds.

With 2 actions and 2 steps, the front-end of SMODELS (*lparse*), responsible of the ground instantiation of the program and its simplification, did not terminate within 95 minutes and was killed (the main reasoning process was never reached). On the same program (without weight constraints, which are implicit in the strong semantics) CASP solves up to 10 steps in about 30 seconds. If the time domain is increased to 6 million points, then *lparse* crashes (probably because of exceeding memory needs), while CASP solves up to 10 steps in less than 2 minutes.

The details of the experiment with 6 million time points are given in Figure 1. Column *step* represents the corresponding regular sort, the fields *atoms*, *var*, *conj*, and *disj*, respectively, show the number of atoms, variables, conjunctions and disjunctions of the formula $\bigwedge_{c \in C} \neg \mathsf{con}(c)$ fed to the constraint solver. Field *attempts* is related to the number of backtracks; it counts the number of stable models of the regular part fed into the Prolog module before the first strong kernel is found. Finally, column *Smodels*

step	*atoms*	*var*	*conj*	*disj*	*attempts*	*Smodels*	*time*
{0,1}	12	2	9	2	1	0.011s	0.306s
{0,...,2}	28	3	21	6	1	0.019s	0.579s
{0,...,3}	51	4	38	12	1	0.009s	1.157s
{0,...,4}	81	5	60	20	1	0.012s	1.654s
{0,...,5}	118	6	87	30	1	0.020s	3.416s
{0,...,6}	162	7	119	42	1	0.039s	6.184s
{0,...,7}	213	8	156	56	1	0.166s	10.175s
{0,...,8}	271	9	198	72	1	0.364s	15.811s
{0,...,9}	336	10	245	90	1	0.461s	23.151s
{0,...,10}	408	11	297	110	1	1.308s	32.949s
SORT: action={1,2} -- time={0,...,600.000}							

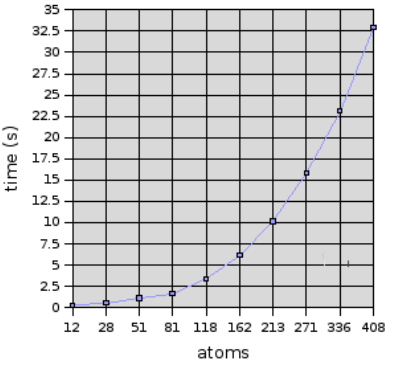


Fig. 2. Test-2 results

step	*atoms*	*var*	*conj*	*disj*	*attempts*	*Smodels*	*time*
{0,1}	10	2	8	1	1	0m0.097s	0m0.216s
{0,...,2}	22	3	18	3	1	0m0.053s	0m0.117s
{0,...,3}	39	4	32	6	1	0m0.057s	0m0.131s
{0,...,4}	61	5	50	10	1	0m0.070s	0m0.181s
{0,...,5}	88	6	72	15	1	0m0.098s	0m0.209s
{0,...,6}	120	7	98	21	1	0m0.203s	0m0.245s
{0,...,7}	157	8	128	28	2	0m0.171s	0m45.349s
{0,...,8}	199	9	162	36	3	0m0.364s	3m54.919s
{0,...,9}	246	10	200	45	8	0m0.556s	35m11.777s
SORT: action={1,2} -- time={0,...,600.000}							

Fig. 3. Test-3 results

reports the time needed by Smodels to compute the stable models of the regular part, and column *time* shows the overall time needed to produce the first strong kernel.

The results with $600,000$ time points are reported in Figure 2. In this experiment constraints are trivial. Basically, they only assign a minimal length to each action execution, so they are always satisfiable, for all action sequences chosen by the answer set solver, and without any backtracking.

Now, if we make constraints more difficult by posing upper bounds on the entire plan execution (so that constraints cannot be trivially satisfied and some backtracking is needed), we obtain the results illustrated in Figure 3. The time needed for constraint solving significantly increases. In future work, it will be interesting to explore different constraint solution strategies on a wider selection of examples.

7 Conclusions

Preliminary experimental results show that the integration of answer set programming and constraint solving techniques may significantly enhance the applicability range of ASP. A simple planning and scheduling problem can be naturally formulated and solved, while one of the most powerful state-of-the-art answer set solvers cannot even reach the main reasoning phase. Our method shares with constraint logic programming frameworks the ability of returning answers that may be compact representations of exponentially many distinct problem solutions, each of which can be easily extracted from the answer.

This work can be extended along several directions. First of all we are looking for more classes of examples of practical interest to extend our experimentation. It may be interesting to try different front-ends (such as DLV's) and different constraint solvers.

A second line of research concerns the interplay of the two solvers. A tighter integration of answer set generation and constraint solving may anticipate inconsistency detection, thereby improving failure handling. It would be interesting to explore dependency-directed forms of backtracking. Such a refined system should be compared through benchmarking to planners and schedulers based on different logics and reasoning meth-

ods (for a collection of pointers to such approaches, see `http://www.aaai.org/AITopics/html/planning.html`).

We mentioned that constrained programs are basically a subclass of weight constraint programs. It may be possible to extend the class of weight constraints supported by our approach, e.g., by using different bounds (e.g., mixing weak and strong semantics), and by dropping the requirement that for all free m-atoms A and all vector of r-constants $\boldsymbol{a}$, answer sets must contain at least one instance of $A(\boldsymbol{a}, \boldsymbol{x}_c)$. Many of our results can be adapted under the assumption that for all distinct weight constraints $l_1\{A_1\}u_1$ and $l_2\{A_2\}u_2$ in a program, A_1 and A_2 are not unifiable.

Moreover, it would be nice to support negative mixed literals. Unfortunately, our approach cannot be easily adapted; the solutions we have explored so far require blind grounding over constrained domains, which is exactly what should be avoided.

Acknowledgments. Work partially supported by the EU working group WASP (5FP), IST-2001-37004. The last author is supported by ARDA contract.

References

1. M. Balduccini, M. Gelfond, R. Watson, and M. Nogueira. The USA-Advisor: A case study in answer set planning. In *Logic Programming and Nonmonotonic Reasoning, 6th International Conference, LPNMR 2001*, volume 2173 of *Lecture Notes in Computer Science*, pages 439–442. Springer, 2001.
2. S. Baselice. Integrazione di tecniche di Answer Set Programming e Constraint Solving. Tesi di laurea, Università degli studi di Napoli Federico II, Naples, Italy, October 2004.
3. M. Cadoli, F.M. Donini, and M. Schaerf. Is intractability of nonmonotonic reasoning a real drawback? *Artificial Intelligence*, 88(1-2):215–251, 1996.
4. L. Carlucci Aiello and F. Massacci. Verifying security protocols as planning in logic programming. *ACM Trans. Comput. Logic*, 2(4):542–580, 2001.
5. P. Cholewiński, V. Marek, A. Mikitiuk, and M. Truszczyński. Experimenting with nonmonotonic reasoning. In *Proceedings of the 12th International Conference on Logic Programming, ICLP 1995*, pages 267–281. MIT Press, 1995.
6. P. Codognet and D. Diaz. Compiling constraints in clp(FD). *Journal of Logic Programming*, 27(3):185–226, 1996.
7. T. Eiter, N. Leone, C. Mateis, G. Pfeifer, and F. Scarcello. A deductive system for nonmonotonic reasoning. In *Logic Programming and Nonmonotonic Reasoning, 4th International Conference, LPNMR'97, Proceedings*, volume 1265 of *LNCS*, pages 364–375. Springer, 1997.
8. Omar El-Khatib, Enrico Pontelli, and Tran Cao Son. Asp-prolog: A system for reasoning about answer set programs in prolog. In *Practical Aspects of Declarative Languages, 6th International Symposium, PADL 2004, Dallas, TX, USA, June 18-19, 2004, Proceedings*, volume 3057 of *Lecture Notes in Computer Science*, pages 148–162. Springer, 2004.
9. M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proc. of the 5th ICLP*, pages 1070–1080. MIT Press, 1988.
10. M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9(3-4):365–386, 1991.
11. J. Jaffar and M. J. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19/20:503–582, May/July 1994.

12. V. Lifschitz and H. Turner. Splitting a Logic Program. In *Proceedings of the 12th International Conference on Logic Programming, Kanagawa 1995*, MIT Press Series Logic Program, pages 581–595. MIT Press, 1995.
13. V.W. Marek and J.B. Remmel. On the expressibility of stable logic programming. In *Logic Programming and Nonmonotonic Reasoning, 6th International Conference, LPNMR 2001*, volume 2173 of *LNCS*, pages 107–120. Springer, 2001.
14. W. Marek and M. Truszczyński. Stable models and an alternative logic programming paradigm. In *The Logic Programming Paradigm: a 25-Year Perspective*, pages 375–398. Springer-Verlag, 1999.
15. J. McCarthy. Circumscription: a form of nonmonotonic reasoning. *Artificial Intelligence*, 13:27–39, 1980.
16. R. C. Moore. Semantical considerations on nonmonotonic logics. *Artificial Intelligence*, 25:75–94, 1985.
17. I. Niemelä and P. Simons. Smodels — an implementation of the stable model and well-founded semantics for normal lp. In *Logic Programming and Nonmonotonic Reasoning, 4th International Conference, LPNMR'97, Proceedings*, volume 1265 of *LNCS*, pages 421–430. Springer, 1997.
18. M. Nogueira, M. Balduccini, M. Gelfond, R. Watson, and M. Barry. An A-Prolog decision support system for the Space Shuttle. In *Practical Aspects of Declarative Languages, Third International Symposium, PADL 2001*, volume 1990 of *Lecture Notes in Computer Science*, pages 169–183. Springer, 2001.
19. R. Reiter. A logic for default reasoning. *Artificial Intelligence*, 13:81–132, 1980.
20. P. Simons, I. Niemelä, and T. Soininen. Extending and implementing the stable model semantics. *Artif. Intell.*, 138(1-2):181–234, 2002.
21. C. Teng, P. Van Hentenryck, and Y. Deville. A generic arc-consistency algorithm and its specializations, June 11 1992.

A Comparison of CLP(FD) and ASP Solutions to NP-Complete Problems

Agostino Dovier[1], Andrea Formisano[2], and Enrico Pontelli[3]

[1] Univ. di Udine, Dip. di Matematica e Informatica
dovier@dimi.uniud.it
[2] Univ. di L'Aquila, Dip. di Informatica
formisano@di.univaq.it
[3] New Mexico State University, Dept. Computer Science
epontell@cs.nmsu.edu

Abstract. This paper presents experimental comparisons between declarative encodings of various computationally hard problems in both Answer Set Programming (ASP) and Constraint Logic Programming (CLP) over finite domains. The objective is to identify how the solvers in the two domains respond to different problems, highlighting strengths and weaknesses of their implementations and suggesting criteria for choosing one approach versus the other. Ultimately, the work in this paper is expected to lay the ground for transfer of concepts between the two domains (e.g., suggesting ways to use CLP in the execution of ASP).

1 Introduction

The objective of this work is to experimentally compare the use of two distinct logic-based paradigms in solving computationally hard problems. The two paradigms considered are *Answer Set Programming (ASP)* [2] and *Constraint Logic Programming over Finite Domains (CLP(FD))* [19]. The motivation for this investigation arises from the successful use of both paradigms in dealing with various classes of combinatorial problems, and the need to better understand their respective strengths and weaknesses. Ultimately, we hope this work will provide indication for integration and cooperation between the two paradigms (e.g., along the lines of [8]).

It is well-known [17,2] that, given a propositional normal logic program P, deciding whether or not it admits an *answer set* [11] is an NP-complete problem. As a consequence, any NP-complete problem can be encoded as a propositional normal logic program under answer set semantics. Answer-set solvers [22] are programs designed for computing the answer sets of normal logic programs; these tools can be seen as theorem provers, or model builders, enhanced with several built-in heuristics to guide the exploration of the solution space. Most ASP solvers rely on variations of the Davis-Putnam-Longeman-Loveland procedure in their computations. Such solvers are often equipped with a front-end that transforms a collection of non-propositional normal clauses (with limited use of function symbols) in a *finite* set of ground instances of such clauses. Some solvers provide also classes of *optimization statements*, used to select answer sets that maximize or minimize an objective function dependent on the content of the answer set.

An alternative framework, frequently adopted to handle NP-complete problems, is *Constraint Logic Programming over Finite Domains* [13,19]. In this context, a finite

M. Gabbrielli and G. Gupta (Eds.): ICLP 2005, LNCS 3668, pp. 67–82, 2005.

domain of objects (typically integers) is associated to each variable in the problem specification, and the typical constraints are literals of the forms $s = t$, $s \neq t$, $s < t$, $s \leq t$, where s and t are arithmetic expressions. Encodings of NP-complete problems and of search strategies are very natural and declarative in this framework. Indeed, a large literature has been developed presenting applications of CLP(FD) to a variety of search and optimization problems [19].

In this paper, we report the outcomes of preliminary experiments aimed at comparing these two declarative approaches in solving combinatorial problems. We address a set of computationally hard problems—in particular, we mostly consider decision problems known to be NP-complete. We formalize each problem, both in CLP(FD) and in ASP, by taking advantage of the specific features available in each logical frameworks, attempting to encode the various problems in the *most declarative* possible way. In particular, we adopt a *constraint-and-generate* strategy for the CLP code, while in ASP we exploit the usual *generate-and-test* approach. Wherever possible, we make use of solutions to these problems that have been presented and accepted in the literature.

With this work we intend to develop a bridge between these two logic-based frameworks, in order to emphasize the strengths of each approach and to promote cross-fertilizations. This study also complements the system benchmarking studies, that have recently appeared for both CLP(FD) systems [10,20] and ASP solvers [1,16,14].

2 The Experimental Framework

In this paper we report on the experimentation we conducted by using one CLP(FD) implementation and two ASP-solvers. The CLP programs have been designed for execution by SICStus Prolog 3.11.2 (using the library `clpfd`)—though the code is general enough to be used on different platforms (e.g., BProlog, ECLiPSe, GNU-Prolog) with minor syntactic adjustment [23].The ASP programs have been designed to be processed by *lparse*, the grounding preprocessor adopted by both the SModels (version 2.28) and the CModels (version 3.03) systems [22]. The CModels system makes use of a SAT solver to compute answer sets—in our experiments we selected the default underlying SAT solver, namely mChaff.

We focused on well-known computationally-hard problems. Among them: Graph k-coloring (Section 3), Hamiltonian circuit (Section 4), Schur numbers (Section 5), protein structure prediction on a 2D lattice [3] (Section 6), planning in a block world (Section 7), and generalized Knapsack (Section 8). Observe that, while some of the programs have been drawn from the best proposals appeared in the literature, others are novel solutions, developed in this project (e.g., the ASP implementation of the PF problem and the planning implementation in CLP(FD)).

In the remaining sections of this paper, we describe the solutions to the various problems and report the results from the experiments. All the timing results, expressed in seconds, have been obtained by measuring only the CPU usage time needed for computing the first solution, if any—thus, we ignore the time spent in reading the input, as well as the time spent to ground the program, in the case of the ASP solvers. We used the `runtime` option to measure the time in CLP(FD), that does not account for the time spent for garbage collection and for system calls. All tests have been performed on a

PC (P4 processor 2.8 GHz, and 512 MB RAM memory) running Linux kernel 2.6.3. Complete codes and results (as well as encodings of other problems) are reported in [7].

3 k-Coloring

The k-coloring problem computes the coloring of a graph using k colors. The main source of case studies adopted in our experiments is the repository of "Graph Coloring and its Generalizations" [24], which provides a rich collection of instances, mainly aimed at benchmarking algorithms and approaches to graph problems. Let us describe the two formalizations of k-coloring.

CLP(FD): In this formulation, we assume that the input graph is represented by a single fact of the form `graph([1,2,3],[[1,2],[1,3],[2,3]])`, where the first argument represents the list of nodes, while the second argument is the list of edges. This is a possible constrain-and-generate CLP(FD)-encoding of k-coloring:

```
coloring(K, Output) :- graph(Nodes, Edges),
      create_output(Nodes, Colors, Output), domain(Colors,1, K),
      different(Edges, Output), labeling([ff], Colors).
create_output([],[],[]).
create_output([N|Nodes], [C|Colors], [N-C|Output]) :-
      create_output(Nodes, Colors, Output).
different([],_).
different([[A,B]|R], Output) :- member(A-CA, Output),
      member(B-CB, Output), CA #\= CB, different(R, Output).
```

In this program, `Output` is intended to be a list of pairs of variables `N-C` where, for each node `N` we introduce a color variable `C` in the range `1...K`. The predicate `different` imposes disequality constraints between variables related to adjacent nodes. We used the `ff` option of `labeling` since it offered the best results for this problem.

ASP: Regarding the ASP encoding of k-coloring we adopt a different representation for graphs. Each node `V` is described by a fact `node(V).` If nodes are natural numbers, a compact interval notation is allowed (e.g., `node(1..138)`). Edges are described by facts, e.g., `edge(1,36). edge(2,45). edge(138,36).`

A natural ASP encoding of the k-coloring problem is:

```
(1)     col(1..k).
(2)     :- edge(X,Y), col(C), color(X,C), color(Y,C).
(3)     1 {color(X,C):col(C)} 1 :- node(X).
```

Rule `(1)` states that there are `k` colors (`k` is a constant to be initialized during the grounding stage). The ASP-constraint `(2)` asserts that two adjacent nodes cannot have the same color, while `(3)` states that each node has exactly one color. Note that, by using domain restricted variables, the single ASP-constraint `(2)` states the property that two adjacent nodes cannot have the same color for all edges $\langle X, Y \rangle$. The same property is described by the predicate `different` in the CLP(FD) code, but in that case a recursive definition is required. This fact shows a common situation that will be observed again in the following sections: ASP often permits a significantly more compact encoding of the problem w.r.t. CLP(FD).

Table 1. Graph k-coloring ('–' denotes no answer in at least 30 minutes of CPU-time— '?' means that none of the three solvers gave an answer)

Instance		3-colorability				4-colorability				5-colorability			
Graph	V×E		SModels	CModels	CLP(FD)		SModels	CModels	CLP(FD)		SModels	CModels	CLP(FD)
1-FullIns_5	282×3247	N	1.06	0.15	0.10	N	–	0.23	2.90	N	–	107.78	–
4-FullIns_4	690×6650	N	0.94	0.29	0.46	N	2.20	0.35	1.98	N	10.02	0.42	–
5-FullIns_4	1085×11395	N	1.72	0.47	1.26	N	4.67	0.57	3.58	N	23.79	0.70	–
3-FullIns_5	2030×33751	N	5.92	1.23	7.24	N	21.31	1.51	13.69	N	–	1.96	–
4-FullIns_5	4146×77305	N	15.11	2.69	33.44	N	69.30	3.37	42.53	N	414.93	4.19	–
3-Insertions_3	56×110	N	4.28	4.16	1281.18	Y	0.03	0.04	<0.01	Y	0.04	0.04	<0.01
4-Insertions_3	79×156	N	328.25	1772.14	–	Y	0.05	0.04	<0.01	Y	0.06	0.05	<0.01
2-Insertions_4	149×541	N	1.20	0.15	2.04	?	–	–	–	Y	0.25	0.07	0.01
4-Insertions_4	475×1795	N	–	1443.33	–	?	–	–	–	Y	3.402	0.32	–
DSJR500.1	500×3555	N	0.53	0.18	0.18	N	2.78	0.21	0.18	N	–	0.26	0.19
DSJC500.1	500×12458	N	2.19	0.45	0.64	N	12.30	0.57	0.76	N	6328.45	6.21	46.55
DSJR500.5	500×58862	N	25.76	1.81	2.97	N	175.63	2.26	2.98	N	971.46	2.71	3.09
DSJC500.5	500×62624	N	28.29	1.92	3.15	N	376.35	2.36	3.19	N	2707.64	2.84	3.47
flat300_20_0	300×21375	N	6.39	0.68	0.63	N	86.91	0.84	0.64	N	1555.37	1.08	0.69
flat300_26_0	300×21633	N	6.45	0.70	0.65	N	131.91	0.87	0.67	N	3711.80	1.13	0.69
flat300_28_0	300×21695	N	6.51	0.70	0.65	N	34.76	0.86	0.69	N	322.99	1.02	0.67
fpsol2.i.1	496×11654	N	2.75	0.41	0.77	N	24.98	0.52	0.77	N	205.12	0.61	0.84
fpsol2.i.2	451×8691	N	1.92	0.33	0.53	N	16.66	0.40	0.54	N	279.96	0.52	0.55
fpsol2.i.3	425×8688	N	1.91	0.32	0.5	N	16.63	0.40	0.51	N	277.91	0.49	0.51
gen200_p0.9_44	200×17910	N	5.53	0.57	0.36	N	30.87	0.70	0.36	N	306.81	0.84	0.38
gen200_p0.9_55	200×17910	N	5.54	0.57	0.36	N	39.56	0.71	0.36	N	287.14	0.85	0.38
gen400_p0.9_55	400×71820	N	38.91	2.19	2.88	N	656.07	2.68	2.89	N	4892.74	3.24	2.93
gen400_p0.9_65	400×71820	N	39.02	2.16	2.88	N	275.33	2.67	2.87	N	1563.52	3.22	2.92
wap05a	905×43081	N	11.39	1.38	2.96	N	62.81	1.73	2.96	N	949.66	2.07	2.96
wap06a	947×43571	N	11.63	1.42	3.25	N	62.70	1.75	3.24	N	1326.84	2.13	3.26
wap07a	1809×103368	N	31.98	3.28	15.14	N	191.06	4.12	15.14	N	2861.64	4.99	15.19
wap08a	1870×104176	N	32.07	3.31	16.17	N	192.54	4.15	16.22	N	3604.96	5.08	16.18

Table 2. The M-N-Queens problem ('–' denotes no answer in 10 min. of CPU-time)

Instance		Solvability for $M = N - 1$					Solvability for $M = N$					Solvability for $M = N + 1$				
N	$V \times E$		SModels	CModels	CLP(FD)	ugraphs		SModels	CModels	CLP(FD)	ugraphs		SModels	CModels	CLP(FD)	ugraphs
5	25×320	N	0.06	0.07	0.01	<0.01	Y	0.06	0.07	<0.01	<0.01	Y	0.07	0.08	<0.01	<0.01
6	36×580	N	1.00	0.11	0.01	<0.01	N	63.80	198.65	1.33	0.02	Y	0.66	0.19	<0.01	0.16
7	49×952	N	341.17	0.20	0.02	0.03	Y	1.95	0.18	<0.01	0.29	Y	0.54	14.08	0.02	0.35
8	64×1456	N	–	0.42	0.16	0.89	N	–	–	–	224.11	Y	116.50	1.28	1.04	807.22
9	81×2112	N	–	0.85	1.37	106.64	?	–	–	–	–	Y	–	–	138.85	131.27
10	100×2940	N	–	3.63	14.53	–	?	–	–	–	–	?	–	–	–	–
11	121×3960	N	–	10.62	148.74	–	?	–	–	–	–	?	–	–	–	–

Results: We tested the above programs on more than one hundred instances drawn from [24]. Such instances belong to various classes of graphs which come from different sources in the literature. Table 1 shows an excerpt of the results we obtained for k-coloring with $k = 3, 4, 5$. The columns report the time (in seconds) using the three systems; the first column of each result indicates whether a solution exists for the problem instance.

A particular class of graph coloring problems listed in [24] originates from encoding a generalized form of the N-queens problem. Graphs for the M-N-queen problems are obtained as follows. The nodes correspond to the cells of a $N \times N$ chess-board. Two nodes u and v are connected by an (undirected) edge if a queen in the cell u attacks the

cell v. Solving the M-N-queens problem consists of determining whether such graph is M-colorable. In the particular case where $M = N$, this is equivalent to finding N independent solutions to the classical N-queens problem. Observe that, for $M < N$ the graph cannot be colored. We ran a number of tests on this specific class of graphs. Table 2 lists the results obtained for $N = 5, \ldots, 11$ and $M = N - 1, N, N + 1$. For the sake of completeness, we also experimented, on these instances, using the library `ugraphs` of SICStus Prolog (a library independent from the library `clpfd`), where the `coloring/3` predicate is provided as a built-in feature. `ugraphs` is slower than CLP(FD) for small instances, however, it finds solutions in acceptable time for some larger instances, whereas CLP(FD) times out.

4 Hamiltonian Circuit

In this section we deal with the problem of establishing whether a directed graph admits an Hamiltonian circuit. The graph representations adopted are the same as in the previous section, with the restriction that graph nodes are `1..N` (needed to correctly use the built-in predicate `circuit` of SICStus Prolog).

CLP(FD): A possible CLP(FD) encoding is the following:

```
hc(Path) :- graph(Nodes, Edges), length(Nodes, N),
     length(Path, N), domain(Path, 1, N),
     make_domains(Path, 1, Edges, N),
     circuit(Path), labeling([ff], Path).
make_domains([], _, _, _).
make_domains([X|Y], Node, Edges, N) :-
     findall(Z, member([Node,Z], Edges), Successors),
     reduce_domains(N, Successors, X),
     Node1 is Node+1, make_domains(Y, Node1, Edges, N).
reduce_domains(0, _, _) :- !.
reduce_domains(N, Successors, Var) :- N>0, member(N,Successors),
     !, N1 is N-1, reduce_domains(N1, Successors, Var).
reduce_domains(N, Successors, Var) :-
     Var #\= N, N1 is N-1, reduce_domains(N1, Successors, Var).
```

We use the built-in predicate `circuit`, provided by `clpfd` in SICStus. In the literal `circuit(List)`, the `List` is a list of domain variables or integers. The goal `circuit`$([X_1, \ldots, X_n])$ constrains the variables so that the set of edges $\langle 1, X_1\rangle$, $\langle 2, X_2\rangle, \ldots, \langle n, X_n\rangle$ is an Hamiltonian circuit. The predicate `make_domains` restricts the admissible values for the variable X_i to the successors of node i in the graph.

ASP: The following program for Hamiltonian circuit comes from the ASP literature [18]:

```
(1)   1 {hc(X,Y) : edge(X,Y)} 1 :- node(X).
(2)   1 {hc(Z,X) : edge(Z,X)} 1 :- node(X).
(3)   reachable(X) :- node(X), hc(1,X).
(4)   reachable(Y) :- node(X), node(Y), reachable(X), hc(X,Y).
(5)   :- not reachable(X), node(X).
```

Table 3. Hamiltonian circuit ('–' denotes no answer within 30 minutes of CPU-time)

Instance	node×edges	Hamiltonian?			
			SModels	CModels	CLP(FD)
hc1	200×1250	Y	2.99	37.59	0.34
hc2	200×1250	Y	2.99	1394.15	0.34
hc3	200×1250	Y	3.03	20.06	0.32
hc4	200×1250	Y	2.98	93.10	0.34
hc5	200×1250	N	1.44	0.22	0.24
hc6	200×1250	N	1.44	0.21	0.10
hc7	200×1250	N	1.44	0.20	0.25
hc8	200×1250	N	1.44	0.20	0.26
np10c	10×90	Y	0.01	0.05	0.0
np20c	20×380	Y	0.07	0.82	0.0
np30c	30×870	Y	0.26	0.27	0.01
np40c	40×1560	Y	0.91	4.38	0.02
np50c	50×2450	Y	2.59	118.18	0.03
np60c	60×3540	Y	7.38	24.81	0.05
np70c	70×4830	Y	15.68	9.47	0.07
np80c	80×6320	Y	27.79	12.55	0.11
np90c	90×8010	Y	45.66	128.25	0.15

Instance	node×edges	Hamiltonian?			
			SModels	CModels	CLP(FD)
2xp30	60×316	N	0.14	0.02	0.03
2xp30.1	60×318	Y	0.18	4.61	0.02
2xp30.2	60×318	Y	–	2.69	5.38
2xp30.3	60×318	Y	–	2.70	5.38
2xp30.4	60×318	N	–	–	–
4xp20	80×392	N	0.24	0.04	0.04
4xp20.1	80×395	N	–	1.47	0.04
4xp20.2	80×396	Y	0.37	3.32	0.03
4xp20.3	80×396	N	0.24	2.65	–
nv70a440	70×423	Y	0.28	1.33	0.05
nv70a460	70×429	Y	0.28	3.00	0.03
nv70a480	70×460	Y	0.29	1.66	0.06
nv70a500	70×473	Y	0.29	1.73	0.03
nv70a520	70×478	Y	0.29	0.36	0.05
nv70a540	70×507	Y	0.31	4.19	0.04
nv70a560	70×516	Y	0.32	0.62	0.05
nv70a580	70×540	Y	0.32	1.00	0.04

The description of the search space is given by rules `(1)` and `(2)`: for each node `X`, exactly one outgoing edge `(X,Y)` and one incoming edge `(Z,X)` belong to the circuit (represented by the predicate `hc`). Rules `(3)` and `(4)` define the transitive closure of the relation `hc` starting from node number `1`. The "test" phase is expressed by the ASP-constraint `(5)`, which weeds out the answer sets that do not represent solutions to the problem. Also in this case, the ASP approach permits a more compact encoding (even if in CLP(FD) we exploited the built-ins `circuit` and `findall`).

Results: Most of the problem instances have been taken from the benchmarks used to compare ASP-solvers [16]. Graphs `hc1`–`hc8` are drawn from `www.cs.uky.edu/ai/benchmark-suite/hamiltonian-cycle.html`. All other graphs are chosen from `assat.cs.ust.hk/Assat-2.0/hc-2.0.html`. The graphs `np`n`c` are complete directed graphs with n nodes and one edge $\langle u, v\rangle$ for each pair of distinct nodes. The graphs `nv`v`a`a are randomly generated graphs, having at most v nodes and a edges. The instances `2xp30` (resp., `4xp20`) are obtained by joining 2 (resp., 4) copies of the graph `p30` (resp., `p20`) plus 2 (resp., 3–4) new edges. Graphs `p20` and `p30` are graphs provided in the SModels' distribution [22]. Table 3 lists the results.

5 Schur Numbers

A set $S \subseteq \mathbb{N}$ is *sum-free* if the intersection of S and the set $S + S = \{x + y : x \in S, y \in S\}$ is empty. The *Schur number* $S(P)$ is the largest integer n for which the set $\{1, \ldots, n\}$ can be partitioned in P sum-free sets. For instance, $\{1, 2, 3, 4\}$ can be partitioned in $S_1 = \{1, 4\}$ and $S_2 = \{2, 3\}$. Observe that the sets $S_1 + S_1 = \{2, 5, 8\}$ and $S_2 + S_2 = \{4, 5, 6\}$ are sum-free. The set $\{1, 2, 3, 4, 5\}$, instead, originates at least 3 sum-free subsets, thus, $S(2) = 4$. It should be noted that, so far, only 4 Schur numbers have been computed, i.e., $S(1) = 1$, $S(2) = 4$, $S(3) = 13$, and $S(4) = 44$. The best known bound for $S(5)$ is $160 \leq S(5) \leq 315$ [21]. Here, we focus on the decision problem: is $S(P) \geq N$? Namely, we look for a function $B : \{1, \ldots, N\} \longrightarrow \{1, \ldots, P\}$ such that: $(\forall I \in \{1, \ldots, N\})(\forall J \in \{I, \ldots, N\})(B(I) = B(J) \rightarrow B(I + J) \neq B(I))$.

Table 4. Schur numbers ('–' denotes no answer within 30 minutes of CPU-time)

Instance $\langle P, N\rangle$	is $Schur(P) \geq N$?	SModels	CModels	CLP(FD)	Instance $\langle P, N\rangle$	is $Schur(P) \geq N$?	SModels	CModels	CLP(FD)
$\langle 4, 43\rangle$	Y	0.27	0.25	0.03	$\langle 5, 111\rangle$	Y	–	0.53	0.16
$\langle 4, 44\rangle$	Y	0.29	3.37	0.56	$\langle 5, 112\rangle$	Y	–	0.55	0.16
$\langle 4, 45\rangle$	N	510.01	892.54	1204.86	$\langle 5, 113\rangle$	Y	–	0.55	0.16
$\langle 4, 46\rangle$	N	561.80	813.73	1340.64	$\langle 5, 114\rangle$	Y	–	11.75	0.16
$\langle 4, 47\rangle$	N	767.80	791.37	1473.02	$\langle 5, 115\rangle$	Y	–	82.48	8.63
$\langle 4, 48\rangle$	N	978.84	805.69	1565.28	$\langle 5, 116\rangle$	Y	–	60.47	8.92
$\langle 4, 49\rangle$	N	1258.57	679.20	1698.08	$\langle 5, 117\rangle$	Y	–	762.91	9.74
$\langle 5, 109\rangle$	Y	–	14.05	0.13	$\langle 5, 118\rangle$	Y	–	21.84	10.19
$\langle 5, 110\rangle$	Y	–	33.29	0.14	$\langle 5, 119\rangle$	Y	–	–	66.95

CLP(FD): For doing that, in $CLP(FD)$ we introduce a list of constrained variables `List` $= [$`B1`$, \ldots,$ `BN`$]$ ranging on `1..P`.

```
schur(N,P) :- length(List,N), domain(List,1,P),List=[1,2|_],
              constraints(List,N), labeling([leftmost],List).
constraints(List, N) :- recursion(List,1,1,N).
recursion(_,I,_,N):- I>N, !.
recursion(List,I,J,N):- I+J>N,!,I1 is I+1,recursion(List,I1,1,N).
recursion(List,I,J,N):- I>J,!,J1 is J+1,recursion(List,I,J1,N).
recursion(List,I,J,N):- K is I+J, J1 is J+1,
      nth(I,List,BI), nth(J,List,BJ), nth(K,List,BK),
      (BI #= BJ) #=> (BK #\= BI), recursion(List,I,J1,N).
```

Each variable `Bi` in `List` can assume values in `1..P`. Its value identifies the block of the partition i belongs to. The predicate `recursion` states that for all `I` and `J`, with $1 \leq$ `I` $\leq$ `J` $\leq$ `N`, the numbers `I`, `J` and `I+J` must not be all in the same block. We set `B1 = 1` and `B2 = 2` to remove some simple symmetries, for a fair comparison w.r.t. the ASP solution that uses rules `(4)` and `(5)`—see below. We have chosen the `leftmost` option of labeling.

ASP: The function B mentioned above is here implemented by a predicate `inpart(X,P)` representing the fact that number X is assigned to part P:

```
(1) number(1..n).       part(1..p).
(2) 1 { inpart(X,P) : part(P) } 1 :- number(X).
(3) :- number(X;Y), part(P), X<=Y, inpart(X,P),
       inpart(Y,P), inpart(X+Y,P).
(4) :- number(X),part(P;P1),inpart(X,P),P1<P,not occupied(X,P1).
(5) occupied(X,P) :- number(X;Y), part(P), Y<X, inpart(Y,P).
```

Rule `(2)` states that `inpart` is a function from numbers to partitions. The ASP-constraints `(3)` states that, for any X and Y, the three numbers X, Y, and $X + Y$ cannot belong to the same partition. The declarative formalization of the problem could be ended here. However, it is customary to add also the constraints `(4)` and `(5)`, that remove symmetries, by selecting the free partition with the lowest index.

Results: Table 4 reports the execution times we obtained. Let us observe that, unfortunately, we are still far from the best known lower bound of 160 for $S(5)$.

6 Protein Structure Prediction

Given a sequence $S = s_1 \cdots s_n$, with $s_i \in \{h, p\}$, the *2D HP-protein structure prediction problem* (reduced from [3]) is the problem of finding a mapping (*folding*) $\omega : \{1, \ldots, n\} \longrightarrow \mathbb{N}^2$ such that

$$(\forall i \in [1, n-1]\})\mathsf{next}(\omega(i), \omega(i+1)) \text{ and } (\forall i, j \in [1, n])(i \neq j \rightarrow \omega(i) \neq \omega(j))$$

and minimizing the energy:

$$\sum_{\substack{1 \\ i+2}} \sum_{\substack{i \\ n}} \sum_{\substack{n \\ j}} \sum_{\substack{2 \\ n}} \mathsf{Pot}(s_i, s_j) \cdot \mathsf{next}(\omega(i), \omega(j))$$

where $\mathsf{Pot}(s_i, s_j) \in \{0, -1\}$ and $\mathsf{Pot} = -1$ if and only if $s_i = s_j = h$. The condition $\mathsf{next}(\langle X_1, Y_1\rangle, \langle X_2, Y_2\rangle)$ holds between two adjacent positions of a given lattice if and only if $|X_1 - X_2| + |Y_1 - Y_2| = 1$. Without loss of generality, we set $\omega(1) = \langle n, n\rangle$ and $\omega(2) = \langle n, n+1\rangle$, to remove some symmetries in the solution space. To remove all symmetries, we should require, that when the sequence turns for the first time, it turns, e.g., to the right. For the sake of simplicity we did not add this disjunctive constraint in the code. Instead, to reduce the solution's space in these experiments, we further add the heuristics constraints $X_i, Y_i \in [N - \sqrt{N}, N + \sqrt{N}]$.

Intuitively, we look for a self-avoiding walk that maximizes the number of contacts between occurrences of objects (aminoacids) of kind h (see Figure 1). Contiguous occurrences of h in the input sequence S contribute in the same way to the energy associated to each spatial conformation and thus they are not considered in the objective function. Note that two objects can be in contact only if they are at an odd distance in the sequence (odd property of the lattice). This problem is a version of the protein structure prediction problem, whose decision problem is known to be NP-complete [4].

CLP(FD): A complete CLP(FD) encoding of this problem (based on the ideas in [3]) can be found in [7]. An extension of this code (in 3D, inside a realistic lattice, and with a more complex energy function) has been used to predict the spatial shape of real proteins [5]. In this case the labeling parameter chosen is `ff`.

ASP: As far as we know, there are no ASP formulations of this problem available in the literature. A specific instance of the problem is represented as a set of facts, describing the sequence of aminoacids. For instance, the protein denoted by `hpphpphpph` (or simply $(\mathtt{hpp})^3\mathtt{h}$ using regular expressions) is described as:

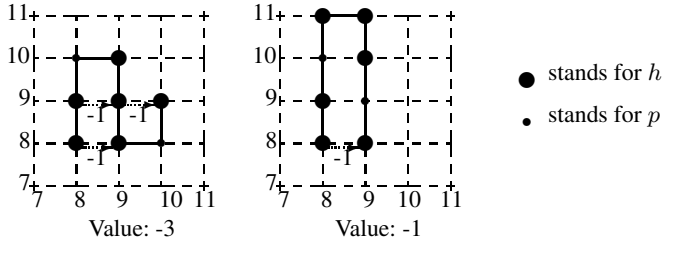


Fig. 1. Two foldings for $S = hhphhhph$ ($n = 8$). The leftmost one is minimal.

```
prot(1,h). prot(2,p). prot(3,p). prot(4,h). prot(5,p).
prot(6,p). prot(7,h). prot(8,p). prot(9,p). prot(10,h).
```

The ASP code is as follows:

```
(1) size(10).      %%% size(N) where N is input length
(2) range(7..13).  %%% [ N-sqrt{N}, N+sqrt{N} ]
(3) sol(1,N,N)   :- size(N).
(4) sol(2,N,N+1) :- size(N).
(5) 1 { sol(I,X,Y) : range(X;Y) } 1 :- prot(I,Amino).
(6) :- prot(I1,A1), prot(I2,A2), I1 < I2,
       sol(I1,X,Y), sol(I2,X,Y), range(X;Y).
(7) :- prot(I1,A1), prot(I2,A2),  I2>1,
       I1 == I2-1, not next(I1,I2).
(8) next(I1,I2) :- prot(I1,A1), prot(I2,A2), I1<I2,
       sol(I1,X1,Y1), sol(I2,X2,Y2), range(X1;Y1;X2;Y2),
       1==abs(Y1-Y2)+abs(X2-X1).
(9) energy_pair(I1,I2) :- prot(I1,h), prot(I2,h),
       next(I1,I2), I1+2<I2, 1==(I2-I1) mod 2.
(10) maximize{ energy_pair(I1,I2) : prot(I1,h): prot(I2,h) }.
```

Rules `(1)` and `(2)`, together with the predicate `prot`, define the domains. Rule `(5)` implements the "generate" phase: it states that each aminoacid occupies exactly one position. Rules `(3)` and `(4)` fix the positions of the two initial aminoacids (they eliminates some symmetric solutions). The ASP-constraints `(6)` and `(7)` state that there are no self-loops and that two contiguous aminoacids must satisfy the `next` property. Rule `(8)` defines the `next` relation, also including the odd property of the lattice. The objective function is defined by Rule `(9)`, which determines the energy contribution of the aminoacids, and rule `(10)`, that searches for answer sets maximizing the energy. For the decision version of the problem, if `en` is the desired energy value, (10) is replaced by

```
(10')   en { energy_pair(I1,I2) : prot(I1,h): prot(I2,h) }.
```

Results: The experimental results for the two programs are reported in Table 5. The nature of this problem (prediction of *the* structure of a protein) justify us to test the systems on the optimization problem. Since CModels does not support optimization statements, we can only compare the performance of SICStus and SModels. Nevertheless, we performed a series of tests relative to the decision version of this problem, namely,

Table 5. Protein structure prediction ('–' denotes no answer within 30 hours of CPU-time)

Instance			Optimization problem		Decision problem		
Input Sequence	Length	Min	CLP(FD)	SModels	CLP(FD)	SModels	CModels
h^{10}	10	-4	0.13	0.74	<0.01	0.53	1.01
h^{15}	15	-8	5.50	10.61	0.05	2.29	2.73
h^{20}	20	-12	766.22	1679.79	0.50	28.23	52.43
h^{25}	25	-16	103962.57	–	1664.35	2169.49	2620.94
$(hpp)^3h$	10	-4	0.10	0.51	<0.01	0.35	0.33
$(hpp)^5h$	16	-6	0.22	22.33	0.08	16.06	15.94
$(hpp)^7h$	22	-8	46.87	1059.86	5.52	96.45	1609.75
$(hpp)^9h$	28	-10	14007.07	–	2815.62	2309.14	5813.23

answering the question "can the given protein fold to reach a given energy level?", using the energy results obtained by solving the optimization version of the problem. The results are also reported in Table 5.

7 Planning

Planning is one of the most interesting applications of ASP. CLP(FD) has been used less frequently to handle planning problems. A planning problem is based on the notions of `State` (a representation of the world) and `Actions` that change the states. We focus on solving a planning problem in the block world domain. Let us assume to have N blocks (blocks $1, \ldots, N$). In the *initial state*, the blocks are arranged in a single stack, in increasing order, i.e., block 1 is on the table, block 2 is on top of block 1, etc. Block N is on top of the stack. In the *goal state*, there must be two stacks, composed of the blocks with odd and even numbers, respectively. In both stacks the blocks are arranged in increasing order, i.e., blocks 1 and 2 are on the table and blocks $N-1$ and N are on top of the respective stacks. The planning problem consists of finding a sequence of T actions (*plan*) to reach the goal state, starting from the initial state. Some additional restrictions must be met: first, in each state at most three blocks can lie on the table. Moreover, a block x cannot be placed on top of a block y if $y \geq x$.

CLP(FD): We study the encoding of block world planning problem in CLP(FD). The code can be easily generalized as a scheme for encoding general planning problems. The plan can be modeled as a list `States` of $T+1$ states. Each `State` is a N-tuple `[B1,...,BN]`, where `Bi=j` means that block `i` is placed on block `j`. The case `j=0` represents the fact that the block `i` lies on the table. The initial state and the final state are represented by the lists $[0, 1, 2, 3, \ldots, \mathtt{N}-1]$ and $[0, 0, 1, 2, \ldots, \mathtt{N}-2]$.

```
planning(NBlocks,NTime) :- init_domains(NBlocks,NTime,States),
 initial_state(States), final_state(States),
 init_actions(NBlocks,NTime,Actions),
 forward(Actions,States), no_rep(Actions),
 action_properties(Actions,States),term_variables(Actions,Vars),
 labeling([leftmost],Vars).
init_domains(NBlocks,NTime,States) :- T1 is NTime+1,
 length(States,T1), init_domains(NBlocks,States).
init_domains(_,[]).
init_domains(N,[S|States]) :- length(S,N),
  init_domains(N,States), domain(S,0,N), count(0,S,'#=<',3).
initial_state([State|_]) :- increasing_list(State).
final_state(Sts) :- append(_,[[0|FS]],Sts), increasing_list(FS).
init_actions(_,0,[]) :- !.
init_actions(N,T,[[Block,To_Block]|Acts]) :- T1 is T-1,
  Block#\=To_Block, Block in 1..N, To_Block in 0..N,
  (Block#<To_Block #=> To_Block#=0), init_actions(N,T1,Acts).
forward([],_).
forward([[Block,To_Block]|B],[CurrState,NextState|Rest]) :-
  element(Block,NextState,To_Block), is_clear(CurrState,Block),
  is_clear(CurrState,To_Block), element(Block,CurrState,Old),
```

Table 6. Planning in blocks world ('–' denotes no answer in less than 3 hours)

Instance Blocks	Instance Length	Plan exists	SModels	CModels	SICStus CLP(FD)	Instance Blocks	Instance Length	Plan exists	SModels	CModels	SICStus CLP(FD)
5	12	N	0.29	0.12	0.01	7	50	N	–	542.98	586.73
5	13	Y	0.33	0.16	0.02	7	51	N	–	991.56	824.61
6	26	N	8.64	8.31	0.32	7	52	N	–	1091.54	1097.13
6	27	Y	12.17	6.56	0.26	7	53	N	–	2044.34	1509.35
7	42	N	355.66	220.00	42.83	7	54	Y	–	431.32	1104.16
7	43	N	565.60	74.19	58.91	8	56	N	3308.28	4667.86	3875.05
7	44	N	1126.52	169.01	80.59	8	57	N	4290.26	866.58	5101.24
7	45	N	2710.53	139.66	111.98	8	58	N	5672.42	287.16	7240.92
7	46	N	7477.13	299.01	158.03	8	59	N	7791.38	1769.51	9838.83
7	47	N	–	180.63	217.26	8	60	N	11079.03	903.10	13917.36
7	48	N	–	209.73	299.31	8	61	N	18376.59	488.78	19470.35
7	49	N	–	463.56	417.63	8	62	N	35835.76	4639.58	27030.19

```
  Old#\=To_Block, forward(B,[NextState|Rest]).
is_clear([],_).
is_clear([A|B],X) :- (X#\=0 #=> A#\=X), is_clear(B,X).
no_rep([_]).
no_rep([[X1,_],[X2,Y2]|Rest]):- X1#\=X2, no_rep([[X2,Y2]|Rest]).
action_properties([],_).
action_properties([[Block,_To]|Rest],[Current,Next|States]) :-
  inertia(1,Block,Current,Next),
  action_properties(Rest,[Next|States]).
inertia(_,_,[],[]).
inertia(N,X,[A|B],[C|D]) :-
  N1 is N+1, inertia(N1,X,B,D), (X#\=N #=> A#=C).
increasing_list(List) :- sequence(List,0).
sequence([],_).
sequence([N|R],N) :- M is N+1, sequence(R,M).
```

The code follows the usual constrain-and-generate methodology. The `init_domains` predicate generates the list of the `NTime` states and fixes the maximum number of objects admitted on the table in each state (using the built-in constraint `count`). After that, the initial and final states are initialized. The predicate `init_actions` specifies that a block can be moved either to the table or to another block having a smaller number. `forward` states that if a block is placed on another one, then both of them must be *clear*, i.e., without any block on top of them. The predicate `no_rep` guarantees that two consecutive actions cannot move the same block. Finally, `action_properties` forces the inertia laws (i.e., if a block is not moved, then it remains in its position).

ASP: There are several standard ways to encode a block world in ASP (e.g., [15,2]). The code used in our experiments is reported in [7].

Results: Table 6 reports the execution times from the three systems, for different number of blocks and plan lengths.

8 Knapsack

In this section we discuss a generalization of the knapsack problem. Let us assume to have n types of objects, and each object of type i has size w_i and it costs c_i. We wish to fill a knapsack with X_1 objects of type 1, X_2 objects of type 2, and so on, so that:

$$\sum_{i=1}^{n} X_i w_i \leq \texttt{max\_size} \quad \text{and} \quad \sum_{i=1}^{n} X_i c_i \geq \texttt{min\_profit}. \tag{1}$$

where `max_size` is the capacity of the knapsack and `min_profit` is the minimum profit required.

CLP(FD): We represent the types of objects using two lists (containing the size and cost of each type of object). For instance, in our tests:

```
objects([2,4, 8,16,32,64,128,256,512,1024],
        [2,5,11,23,47,95,191,383,767,1535]).
```

The CLP(FD) encoding is:

```
knapsack(Max_Size,Min_Profit) :- objects(Weights,Costs),
       length(Sizes,N), length(Vars,N), domain(Vars,0,Max_Size),
       scalar_product(Sizes,Vars,#=<,Max_Size),
       scalar_product(Costs,Vars,#>=,Min_Profit),
       labeling([ff],Vars).
```

Observe that we used the built-in predicate `scalar_product` for implementing (1). The built-in predicate `knapsack`, available in SICStus Prolog, is a special case of `scalar_product` where the third argument is the equality constraint.

ASP: Input representation is given by facts of the form: `item(Item,Weight,Cost).`

```
item(1,2,2).    item(2,4,5).   item(3,8,11).    item(4,16,23).
item(5,32,47).  item(6,64,95). item(7,128,191). item(8,256,383).
item(9,512,767).               item(10,1024,1535).
```

The knapsack problem can be encoded as follows:

```
(1)  occs(0..max_size).
(2)  item_occs(I,Item_Occurences,W,C) :-
        item(I,W,C), occs(O), Item_Occurences = O/W.
(3)  1{in_sack(I,IO,W,C):item_occs(I,IO,W,C)}1 :- item(I,W,C).
(4)  cond_cost :-
      min_profit [in_sack(I,IO,W,C):item_occs(I,IO,W,C) = IO*C].
(5)  :- not cond_cost.
(6)  cond_weight :-
       [in_sack(I,IO,W,C):item_occs(I,IO,W,C) = IO*W] max_size.
(7)  :- not cond_weight.
```

Fact `(1)` fixes the domain for the occurrences of items in the knapsack. Rule `(2)`, instead, fixes the possible occurrences for each item in the knapsack. Rule `(3)` states that, for each type of objects `I`, there is only one fact `in_sack(I,IO,W,C)` in the answer set, representing the number of objects of type `I` in the knapsack. This trick is not needed in CLP(FD), where the same effect is obtained by bounds consistency. Rules `(4)`–`(7)` establish the constraints of minimum profit and maximum size. The two constants `max_size` and `min_profit` must be provided to lparse during grounding.

Table 7. Knapsack instances ('–' denotes no answer within 30 minutes of CPU-time)

max_size	min_profit	Answer	SICStus	SModels
255	374	Y	0.02	0.04
255	375	N	0.03	3.08
511	757	Y	0.36	0.12
511	758	N	0.36	130.82
1023	1524	Y	8.81	0.49
1023	1525	N	8.75	–
2047	3059	Y	368.50	1.84
2047	3060	N	366.79	–

Results: Table 7 reports some of the results we obtained. CModels seems unable to properly deal with this problem: for any of the instances we experimented with (except the smallest ones, involving at most five types of objects) the corresponding process was terminated by the operative system. The reason for this could be found by observing that the run-time images of such processes grow very large in size (up to 4.5GB, in some instances). We have also encoded this problem using the `weight` declarations, but the presented code is faster and less sensible to size of the numbers.

9 Discussion and Conclusions

We tested the CLP(FD) and ASP codes for various combinatorial problems. In the Tables
1–7 we reported the running times (in seconds) of the solutions to these problems on different problem instances. Let us try here to analyze these results.

First of all, from the benchmarks, it is clear that ASP provides a more compact, and probably more declarative, encoding; in particular, the reliance on grounding and domain-restricted variables allows ASP to avoid the use of recursion in many situations.

As far as running times are concerned, CLP(FD) definitely wins the comparison vs. SModels. In a few cases, the running times are comparable, but in most of the cases CLP(FD) runs significantly faster. Observe also that CModels is, in most of the problems, faster than SModels; part of this can be justified by the fact that the programs we are using are mostly tight [9], and by the high speed of the underlying SAT solver used by CModels.

The comparison between CLP(FD) and CModels is more interesting. In the k-coloring and N-M-queens cases, running times are comparable. In some of the classes of graphs, CModels performs slightly better on all instances. More in general, whenever the instances of a single class are considered, one of the two systems tends to always outperform the other. This indicates that the behavior of the solver is significantly affected by the nature of the specific problem instances considered (recall that each class of graphs comes from encodings of instances of different problems [24]).

As one may expect, the bottom-up search strategy of ASP is less sensitive to the presence of solutions w.r.t. the top down search strategy of CLP(FD). As a matter of fact, CLP(FD) typically runs faster than CModels when a solution exists. Moreover, CLP(FD) behaves better on small graphs. For the Hamiltonian circuit problem, CLP(FD) runs significantly faster—we believe this is due to the use of the built-in global constraint `circuit`, which guarantees excellent constraint propagation. In this

Table 8. Schematic results' analysis. + (-) means that the formalism is (not) applicable. ++ that it is the best when the two formalisms are applicable.

	Coloring	Hamilton	Schur	PF	Planning	Knapsack
CLP(FD)	+	++	+	+	+	+
ASP CModels	++	+	+	-	+	-

case, only in absence of solutions the running times are comparable—i.e., when the two approaches are forced to traverse the complete search tree. A similar situation arises in computing Schur numbers. When the solution exists CLP(FD) performs better. On the other hand, whenever there is no solution, running times are favorable to CModels.

Regarding the protein folding problem, CLP(FD) solves the optimization problems much faster than ASP. In the decision version, times are closer. Also in this case, however, the ASP code appears to be simpler and more compact than the CLP(FD) one. In general, in designing the CLP code, the programmer cannot easily ignore knowledge about the inference strategy implemented in the CLP engine. The fact that CLP(FD) adopts a top-down depth-first strategy influences programmer's choices in encoding the algorithms.

For the planning problem, we observe that SModels runs faster than CModels for small instances. In general, CLP(FD) performs better for small dimensions of the problem. On the other hand, when the dimension of the problem instance becomes large, the behavior of CLP(FD) and SModels become comparable while CModels provides the best performance. In fact, the performance of CModels does not seem to be significantly affected by the growth in the size of the problem instance, as clearly happens for CLP(FD) and SModels. The same phenomenon can be also observed in other situations, e.g., in the Hamiltonian circuit and Schur numbers problems. In these cases, the time spent by CModels to obtain a solution does not appear to be directly related to the raw dimension of the problem instance. Initial experiments reveal that this phenomenon arises even when different SAT-solvers are employed. Further studies are needed to better understand to which extent the intrinsic structure of an instance biases CModels' behavior, in particular the way in which CModels' engine translates an ASP program into a SAT-instance.

For the Knapsack problem, CModels is not applicable. CLP(FD) runs definitively faster than SModels; furthermore, SModels becomes inapplicable for large problem instances.

Table 9 intuitively summarizes our observations drawn from the different benchmarks. Although these experiments are quite preliminary, they already provide some concrete indications that can be taken into account when choosing a paradigm to tackle a problem. We can summarize the main points as follows:

- graph-based problems have nice compact encodings in ASP and the performance of the ASP solutions is acceptable and scalable;
- problems requiring more intense use of arithmetic and/or numbers are declaratively and efficiently handled by CLP(FD);
- for problems with no arithmetic, the exponential growth w.r.t. the input size is less of an issue for ASP.

A comparison between CLP(FD) implementations is outside the scope of this paper (see, e.g., [10,20,6]). Nevertheless, we tested the CLP(FD) programs using B-Prolog, ECLiPSe, and GNU Prolog [23]. As far as the running times are concerned, these experiments indicated that B-Prolog and SICStus Prolog have comparable behavior, GNU Prolog is the fastest, and ECLiPSe the slowest.

We have excluded the grounding phase from the ASP timings. In our tests it is negligible w.r.t. the SModels/CModels running time, save for easier instances (with running times shorter than one second). Moreover, we also tested CModels with other SAT solvers (SIMO, RELSAT, ZCHAFF). As one can expect (see, e.g., [12]) the choice of the SAT solver influences performance, but there is no clear winner.

In the future we plan to extend our analysis to other problems and to other constraint solvers (e.g., ILOG) and ASP-solvers (e.g., ASSAT, aspps, DLV). In particular, we are interested in answering the following questions:

- is it possible to formalize domain and problem characteristics to lead the choice of which paradigm to use?
- is it possible to introduce strategies to split problem components and map them to cooperating solvers (using the best solver for each part of the problem)?

In particular, we are interested in identifying those contexts where the ASP solvers perform significantly better than CLP. It seems reasonable to expect this behavior, for instance, whenever incomplete information comes into play.

Acknowledgments. We thank the anonymous referees for their patience and for their useful suggestions; in particular for the ASP encoding of the Knapsack problem. This work is partially supported by the GNCS2005 project on constraints and their applications and NSF grants CNS-0220590, HRD-0420407, and CNS-0454066.

References

1. C. Anger, T. Schauß, and M. Truszczyński. ASPARAGUS – the Dagstuhl Initiative. *ALP Newsletter*, 17(3), 2004.
2. C. Baral. *Knowledge representation, reasoning and declarative problem solving*. Cambridge University Press, 2003.
3. P. Clote and R. Backofen. *Computational Molecular Biology*. Wiley & Sons, 2001.
4. P. Crescenzi et al. On the complexity of protein folding. In *STOC*, pages 597–603, 1998.
5. A. Dal Palù, A. Dovier, and F. Fogolari. Constraint logic programming approach to protein structure prediction. *BMC Bioinformatics*, 5(186):1–12, 2004.
6. D. Diaz and P. Codognet. Design and Implementation of the GNU Prolog System. *Journal of Functional and Logic Programming* 2001(6), 2001.
7. A. Dovier, A. Formisano, and E. Pontelli. A comparison of CLP(FD) and ASP solutions to combinatorial problems. `www.di.univaq.it/~formisano/CLPASP`
8. I. Elkabani, E. Pontelli, and T. C. Son. SModels with CLP and Its Applications: A Simple and Effective Approach to Aggregates in ASP. In *ICLP*, 73–89, 2004.
9. E. Erdem and V. Lifschitz. Tight Logic Programs. In *TPLP*, 3:499-518, 2003.
10. A. J. Fernandez and P. M. Hill. A Comparative Study of 8 Constraint Programming Languages Over the Boolean and Finite Domains. *Constraints*, 5(3):275–301, 2000.
11. M. Gelfond and V. Lifschitz. The Stable Model Semantics for Logic Programming. In *ICLP*, pages 1070–1080, MIT Press, 1988.

12. E. Giunchiglia, Y. Lierler, and M. Maratea. SAT-Based Answer Set Programming. In Proc. of *AAAI'04*, pages 61–66, AAAI/Mit Press, 2004.
13. J. Jaffar and M. J. Maher. Constraint Logic Programming: A Survey. *J. of Logic Programming*, 19/20:503–581, 1994.
14. Y. Lierler and M. Maratea. CModels-2: SAT-based Answer Set Solver Enhanced to Non-tight Programs. In *LPNMR*, pages 346–350. Springer Verlag, 2004.
15. V. Lifschitz. Answer Set Planning. In *Logic Programming and Non-monotonic Reasoning*, pages 373–374. Springer Verlag, 1999.
16. F. Lin and Y. Zhao. ASSAT: Computing Answer Sets of A Logic Program By SAT Solvers. In Proc. of *AAAI'02*, pages 112–117. AAAI/MIT Press, 2002.
17. V. W. Marek and M. Truszczyński. Autoepistemic Logic. *JACM*, 38(3):588–619, 1991.
18. V. W. Marek and M. Truszczyński. Stable models and an alternative logic programming paradigm. *The Logic Programming Paradigm*, 375–398. Springer, 1999.
19. K. Marriott and P. J. Stuckey. *Programming with Constraints*. MIT Press, 1998.
20. M. Wallace, J. Schimpf, K. Shen, and W. Harvey. On Benchmarking Constraint Logic Programming Platforms. *Constraints*, 9(1):5–34, 2004.
21. E. W. Weisstein. *Schur Number*. From MathWorld–A Wolfram Web Resource. `mathworld.wolfram.com/SchurNumber.html`.
22. Web references for some ASP solvers. ASSAT: `assat.cs.ust.hk`. CCalc: `www.cs.utexas.edu/users/tag/cc`. CModels: `www.cs.utexas.edu/users/tag/cmodels`. DeReS and aspps: `www.cs.uky.edu/ai`. DLV: `www.dbai.tuwien.ac.at/proj/dlv`. SModels: `www.tcs.hut.fi/Software/smodels`.
23. Web references for some CLP(FD) implementations. SICStus Prolog: `www.sics.se/isl/sicstuswww/site/index.html`. B-Prolog: `www.probp.com`. ECLiPSe: `www.icparc.ic.ac.uk/eclipse`. GNU Prolog: `pauillac.inria.fr/~diaz/gnu-prolog`.
24. Web site of COLOR02/03/04: Graph Coloring and its Applications: `http://mat.gsia.cmu.edu/COLORING03`.

Guard and Continuation Optimization for Occurrence Representations of CHR

Jon Sneyers*, Tom Schrijvers**, and Bart Demoen

Dept. of Computer Science, K.U.Leuven, Belgium
{jon, toms, bmd}@cs.kuleuven.ac.be

Abstract. Constraint Handling Rules (CHR) is a high-level rule-based language extension, commonly embedded in Prolog. We introduce a new occurrence representation of CHR programs, and a new operational semantics for occurrence representations, equivalent to the widely implemented refined operational semantics. The occurrence representation allows in a natural way to express guard and continuation optimizations, which remove redundant guards and eliminate redundant code for subsumed occurrences. These optimizations allow CHR programmers to write self-documented rules with a clear logical reading. We show correctness of both optimizations, present an implementation in the K.U.Leuven CHR compiler, and discuss speedup measurements.

1 Introduction

Constraint Handling Rules (CHR) is a high-level multi-headed rule-based programming language extension originally designed for writing constraint solvers. We assume that the reader is familiar with CHR, referring to [5,4] for an overview Our optimizations are formulated independent of host language, but the implementation and examples described in this paper are in Prolog.

The original theoretical operational semantics for CHR (ω_t), defined in [5], is nondeterministic since it does not specify the order in which rules are tried. However, all recent implementations use a more specific operational semantics, called the *refined* operational semantics (ω_r) [4]. In ω_r, the rules are tried in textual order. In practice, CHR programmers use the ω_r semantics. Their programs possibly are non-terminating or produce unintended results under ω_t semantics.

The dilemma CHR programmers face is the following: either they make sure their programs are valid under ω_t semantics, or they write programs that behave correctly only under ω_r semantics. Sticking to ω_t semantics results in more declarative code with a clear logical reading. Using ω_r semantics can result in more efficient code and allows easier implementation of programming idioms like key lookup. However, source code readability decreases significantly since rules not necessarily contain all preconditions for applying it: preconditions that are implicitly entailed by the rule order are often omitted by the programmer.

* This work was partly supported by project G.0144.03 funded by F.W.O.-Vlaanderen.
** Research Assistant of the Research Foundation - Flanders (F.W.O.-Vlaanderen).

M. Gabbrielli and G. Gupta (Eds.): ICLP 2005, LNCS 3668, pp. 83–97, 2005.

In this paper, we propose compiler optimizations that are a major step towards allowing CHR programmers to write more readable and declarative programs without sacrificing efficiency. They automatically remove redundant guard conditions, on the occurrence level. This paper extends [11,13], which introduced guard simplification, a special case of guard optimization for rules.

The optimizations presented here are mainly based on reasoning about the guards of the CHR rules. This work is orthogonal to other optimization techniques, and they can be combined effortlessly, as we have done in the K.U.Leuven CHR compiler [8].

The next section intuitively describes the optimizations by discussing some examples. Section 3 introduces a new operational semantics for occurrence representations of CHR programs, equivalent to the refined operational semantics. Then, in Section 4, the new guard and continuation optimizations are defined formally, and their correctness is showed w.r.t. this new semantics. Section 5 briefly discusses an implementation of the optimizations in the K.U.Leuven CHR system, and the speedups we have measured. Finally, we conclude in Section 6.

2 Motivating Examples

Example 1 (guard optimization).

```
pos  @ sign(P,S) <=> P > 0 | S = positive.
zero @ sign(Z,S) <=> Z =:= 0 | S = zero.
neg  @ sign(N,S) <=> N < 0 | S = negative.
```

If the third rule, `neg`, is tried, we know `pos` and `zero` did not fire, because otherwise, the `sign/2` constraint would have been removed. Because the first rule, `pos`, did not fire, its guard must have failed, so we know that $N \leq 0$. From the failing of the second rule, `zero`, we can derive $N \neq 0$. Now we can combine these results to get $N < 0$, which trivially entails the guard of the third rule. Hence this guard will always succeed, and we can safely remove it. This results in slightly more efficient generated code, and — maybe more importantly — it might also be useful for other analyses. In this example, guard optimization reveals that all `sign/2` constraints are removed after the third rule, allowing the *never-stored* analysis [10] to detect that `sign/2` is never-stored. □

Example 2 (types and modes).

```
sum([],S) <=> S = 0.
sum([X|Xs],S) <=> sum(Xs,T), S is X + T.
```

We consider head matchings to be an implicit part of the guard: the last rule can be written as "`sum(A,S) <=> A = [X|Xs] | sum(Xs,T), S is X + T.`". Guard optimization can be much more effective if the types (and modes) of constraint arguments are known. If we know the first argument of constraint `sum/2` is an instantiated list, these two simplification rules cover all possible cases and thus the constraint is never-stored. In [9], optional mode declarations

were introduced to specify the mode of constraint arguments: ground (+) or unknown (?). Inspired by the Mercury type system [14], we have added optional type declarations to define types and specify the type of constraint arguments. For the above example, the CHR programmer would add:

```
:- chr_type list(T) ---> [] ; [T | list(T)].
:- constraints sum(+list(int), ?int).
```

The first line is a recursive and generic type definition for lists of type `T`, where `T` can be instantiated with built-in types like `int`, `float`, the general type `any`, or a user-defined type. The constraint declaration on the second line includes mode and type information. Using this knowledge, we can rewrite the last rule to "`sum(A,S) <=> true | A = [X|Xs], sum(Xs,S2), S is X + S2.`", keeping its behavior intact while again helping never-stored analysis. □

Note that it is often crucial to provide type and mode information to get the optimization results presented in this paper.

Example 3 (occurrence subsumption).

(a) `a(X,A,B), a(X,C,D) <=> A < B, C < D |` Body.
(b) `b(X,Y,Z), b(Y,Z,X), b(Z,X,Y) <=>` Body.
(c) `c(X,Y,Z), c(Y,Z,X), c(Z,X,Y) <=> (p(X); p(Y)) |` Body.
(d) `d(A,B), d(B,C) <=> A \== C |` Body.
`d(A,B), d(B,C) <=>` Body.

In examples (a) to (d) above, underlined occurrences are *subsumed* by earlier occurrences: the head constraints and guard are symmetric. Because a constraint is removed by a rule application the subsumed occurrences are redundant. The underlined occurrences are derived to be `passive`, meaning they can be skipped in the execution of the program. In the occurrence representation (introduced in the next section) we can express this by setting the guard for these occurrences to `fail`.

A strong occurrence subsumption analysis takes away the need for CHR programmers to write `pragma passive` declarations to improve efficiency, since the compiler automatically adds such declarations. As a result, the CHR source code contains less non-declarative operational pragmas, improving compactness and readability. □

Examples 1 and 2 may seem trivial, and similar optimizations have been proposed in the context of Prolog, but example 3 is very specific for CHR. They are all covered by the optimizations introduced in Section 4.

3 Semantics for Occurrence Representations

We will use $[H|T]$ to denote a list where the first element is H and remaining elements are T; $+\!\!+$ for list concatenation and $\square$ or `[]` for the empty list; $\uplus$ for multiset union, $\sqcap$ for multiset intersection, and $\sqsubseteq$ for multiset subset. We will

sometimes abuse notation by implicitly converting between lists and multisets, sets and multisets, and lists and conjunctions (where the conjunction is evaluated in the same order as the order of the list elements). We use $vars(E)$ to denote the variables of a syntactic expression E.

3.1 CHR Programs

CHR constraint symbols are drawn from the set of predicate symbols, denoted by a functor/arity pair. CHR constraints are atoms constructed from these symbols. To improve readability, we will often omit the arguments of CHR constraints. Constraints are either CHR constraints or *built-in* constraints in some constraint domain $\mathcal{D}$. The former are manipulated by the CHR execution mechanism while the latter are handled by the underlying constraint solver of the host language. We will assume this underlying solver supports at least equality, `true` and `fail`.

Definition 1 (CHR program). *A CHR program P is a list of CHR rules R_i of the form $H_i^k \setminus H_i^r \iff g_i \mid B_i$ where H_i^k and H_i^r (kept/removed heads) are lists of CHR constraints ($H_i = H_i^k +\!\!+ H_i^r \neq \square$); g_i (guard) is a list of built-in constraints; B_i (body) is a list of constraints.*

If H_i^k is empty, then the rule R_i is a *simplification* rule. If H_i^r is empty, then R_i is a *propagation* rule. Otherwise it is a *simpagation* rule. The guard and body of a rule are often treated as conjunctions. We assume all arguments of the CHR constraints in H_i to be unique variables, making any head matchings explicit in the guard. This head normalization procedure is explained in [3].

Note that built-in constraints used in the guard are always *ask*-constraints when variables occurring in the head are involved. For example, the rule `p(X) <=> X = foo | B` is identical to the rule `p(X) <=> X == foo | B`. In other words, guards cannot modify variable bindings of constraint arguments.

3.2 Occurrence Representation

Most CHR compilers generate one block of host-language code (e.g. one Prolog clause) for every occurrence of a constraint in the head of a rule. Therefore it makes sense to represent a CHR program on the occurrence-level instead of the rule-level. This *occurrence representation* corresponds more closely to the generated code of current compilers. Its finer granularity allows the formulation of more powerful optimizations.

The head constraint occurrences are numbered from top to bottom and from left to right (but first the removed constraints, then the kept constraints). The i-th occurrence of a constraint c is denoted as $c : i$, the number of the rule in which it occurs as $\mathrm{rnum}(c : i)$. We will write $\mathrm{Occ} = \mathrm{Occ}^k \cup \mathrm{Occ}^r$ for the set of all occurrences of a given CHR program, where Occ^k are occurrences from the kept heads H_i^k and Occ^r are occurrences from the removed heads H_i^r.

Definition 2 (Occurrence representation). *An occurrence representation $O = (g, b, p_r, p_k, n_s, n_f)$ of a CHR program P is a 6-tuple of functions:*

- *g maps occurrences to lists of built-in constraints;*
- *b maps occurrences to lists of any constraints;*
- *p_r (p_k) map occurrences to subsets of* Occ^r *(*Occ^k*);*
- *n_s and n_f map occurrences to occurrences.*

We say g returns the *guard* for an occurrence, b returns the *body*, p_r returns the *removed partner* constraint occurrences and p_k returns the *kept partner* constraint occurrences, n_s returns the *success continuation* occurrence and n_f returns the *fail continuation* occurrence. For an occurrence $o \in \mathrm{Occ}$, we will write $p(o)$ to denote $p_k(o) +\!\!+ p_r(o)$ and $h(o)$ to denote $[o|p(o)]$.

3.3 The ω_o Semantics for Occurrence Representations

In this section we will introduce the call-based refined operational semantics for occurrence representations, referred to as ω_o semantics. It is a variant of the call-based refined operational semantics ω_c [10], formulated in terms of occurrence representations. The ω_c semantics is an equivalent variant of the refined operational semantics ω_r [4]. The difference between these two semantics lies in their formulation. The transition system of ω_r linearizes the dynamic call-graph of CHR constraints into the execution stack of its execution states. However, in ω_c (and ω_o), constraints are treated as procedure calls: each newly added *active* constraint searches for possible matching rules in order, until all matching rules have been executed or the constraint is deleted from the store. As with a procedure, when a matching rule fires, other CHR constraints may be executed as subcomputations and, when they finish, the execution returns to finding rules for the current active constraint. The latter semantics are much closer to the procedure-based target languages of current CHR compilers, like Prolog and HAL. This makes the ω_c (and ω_o) semantics much more suitable for reasoning about optimizations.

Execution State of ω_o. An *identified* CHR constraint $c\#i$ is a CHR constraint c associated with some unique integer i. This number serves to differentiate among copies of the same constraint. An *occurrenced* identified CHR constraint $c : j\#i$ is an identified CHR constraint associated with an integer j, indicating that only matches with occurrence $c : j$ should be considered (in other work, the notation $c\#i : j$ is used). We introduce functions $\xi(c\#i : j) = \xi(c\#i) = \xi(c) = c$ and $\mathrm{id}(c\#i : j) = \mathrm{id}(c\#i) = i$, and extend them to lists, sets and multisets of (identified) CHR constraints in the obvious manner.

The execution state of the ω_o semantics is identical to the execution state of the ω_c semantics: it is a tuple $\langle G, A, S, B, T\rangle_n$ where G, A, S, B, T and n, represent the goal, call stack, CHR store, built-in store, propagation history and next free identifier respectively. We use σ_i to denote execution states.

The *goal* G is a list of CHR constraints and built-in constraints. The *execution stack* A is a list of identified CHR constraints, with a strict ordering where the top-most constraint is called *active*. The *CHR constraint store* S is a multiset of identified CHR constraints. The *built-in constraint store* B is a conjunction of built-in constraints that have been passed to the underlying solver.

The *propagation history* T is a set of lists, each recording the identities of the CHR constraints which fired a rule, and the number of the rule itself. This is necessary to prevent trivial non-termination for propagation rules: a propagation rule is allowed to fire on a set of constraints only once. Finally, the *next free identifier* n represents the next integer which can be used to identify a CHR constraint. Given an initial goal G, the initial state is $\langle G, \Box, \emptyset, \emptyset, \emptyset \rangle_1$.

Definition 3 (Matching conditions). *Given an occurrence representation O. For every occurrence $o \in \mathrm{Occ}$ and multisets S, K and R of CHR constraints (possibly identified and/or occurrenced), we define the following two conditions:*

$$\mathsf{sat}_{kr}(o, S) \triangleq g(o) \wedge \big(\xi(h(o)) \trianglelefteq \xi(S)\big)$$

$$\mathsf{sat}_{h}(o, K, R) \triangleq g(o) \wedge \big(\xi(p_k(o)) = \xi(K)\big) \wedge \big(\xi(p_r(o)) = \xi(R)\big)$$

If S is the CHR store and the built-in store entails $\mathsf{sat}_{kr}(o, S)$, then the rule of occurrence o can be applied. The condition $\mathsf{sat}_{h}(o, K, R)$ is used when we need to distinguish between the kept and removed partner constraints. Note that $\mathsf{sat}_{kr}(o, S) \Leftrightarrow \exists K, R\ \big(\mathsf{sat}_{h}(o, K, R) \wedge \xi(K \uplus R \uplus \{o\}) \trianglelefteq \xi(S)\big)$.

Transition Rules of ω_o. Execution proceeds by exhaustively applying transitions to the initial execution state until the built-in solver state is unsatisfiable or no transitions are applicable. We denote transitions from state σ_0 to σ_1 with $\sigma_0 \rightarrowtail_N \sigma_1$ where N is the (shorthand) name of the transition. We define $\rightarrowtail^*$ to be the reflexive transitive closure of $\rightarrowtail$.

We define $\mathit{solutions}_V(B)$ to be the set of all substitutions (unordered assignments to all variables of V) satisfying B. We say a set of variables $I \subseteq V$ is independent for c w.r.t. B iff $\mathit{solutions}_I(B) = \mathit{solutions}_I(B \wedge c) = Y$ and X_1 and X_2 exist such that $X_1 \times Y = \mathit{solutions}_V(B)$ and $X_2 \times Y = \mathit{solutions}_V(B \wedge c)$ where $\times$ denotes the Cartesian product of two sets (ignoring order). We define $\mathit{affected\_vars}_B(c) \triangleq \mathit{vars}(B \wedge c) \setminus I$, where I a maximal independent (for c w.r.t. B) subset of $\mathit{vars}(B \wedge c)$. Intuitively, when adding a built-in constraint c to the built-in store B, we have to trigger at least the CHR constraints containing one or more affected variables (variables from $\mathit{affected\_vars}_B(c)$).

The possible transitions are defined in Figure 1. The actual definition of the *solve* function will depend on the built-in solver. The lower bound of the ω_c semantics is a subset of the lower bound defined here. However, for the Herbrand solver (the built-in solver of Prolog), this lower bound corresponds to current implementations: it boils down to triggering the constraints containing a variable that is touched (instantiated or bound to another variable) by adding c. For other host languages, it might not be feasible (or worth the overhead) to implement this lower bound. The new lower bound for Solve is closer to current implementations, and avoids references to guards. Because of this change, ω_o semantics may demand more constraints to be triggered (without causing any additional rule applications). However, the lower bound of ω_c is much harder to compute, possibly causing more overhead than what is gained by avoiding redundant constraint triggering.

1. Solve: $\langle c, A, S, B, T\rangle_n \rightarrowtail_{So} \langle \Box, A, S', B', T'\rangle_{n'}$
where c is a built-in constraint. If $\mathcal{D} \models \neg\bar{\exists}_\emptyset c \wedge B$, then $S' \triangleq S$, $B' \triangleq c \wedge B$, $T' \triangleq T$, $n' \triangleq n$. Otherwise ($\mathcal{D} \models \bar{\exists}_\emptyset c \wedge B$), there is a series of transitions $\langle S_1, A, S, c \wedge B, T\rangle_n \rightarrowtail^* \langle \Box, A, S', B', T'\rangle_{n'}$, where the triggered constraints $S_1 \triangleq \mathit{solve}(S, B, c)$ are a subset of S such that $L \subseteq S_1 \subseteq U$, where
• *Lower bound*: $L \triangleq \{x \in S \mid \mathit{vars}(x) \cap \mathit{affected\_vars}_B(c) \neq \emptyset\}$
• *Upper bound*: $U \triangleq \{x \in S \mid \mathit{vars}(x) \not\subseteq \mathit{fixed}(B)\}$ where $\mathit{fixed}(B)$ is the set of variables fixed by B ($v \in \mathit{fixed}(B)$ if $\mathcal{D} \models \bar{\exists}_v(B) \wedge \bar{\exists}_{\rho(v)}\rho(B) \rightarrow v = \rho(v)$ for arbitrary renaming ρ). Hence, ground constraints are not triggered.

2a. Activate: $\langle c, A, S, B, T\rangle_n \rightarrowtail_A \langle c : 1\#n, A, \{c\#n\} \uplus S, B, T\rangle_{(n+1)}$
where c is a (non-identified) CHR constraint.

2b. Reactivate: $\langle c\#i, A, S, B, T\rangle_n \rightarrowtail_R \langle c : 1\#i, A, S, B, T\rangle_n$
where $c\#i$ is a CHR constraint in the store (back in the queue through **Solve**).

---

3. Drop: $\langle c : j\#i, A, S, B, T\rangle_n \rightarrowtail_{Dp} \langle \Box, A, S, B, T\rangle_n$ where $c : j \notin \mathrm{Occ}$.

---

4. Simplify: $\langle o\#i, A, H \cup S, B, T\rangle_n \rightarrowtail_{Si} \langle \Box, A, S', B', T'\rangle_{n'}$
where $o = c : j \in \mathrm{Occ}^r$ and $H = P_k \cup P_l \cup \{c\#i\}$, and

$$\langle \theta(b(o)), A, P_k \cup S, \theta \wedge B, T \cup \{h\}\rangle_n \rightarrowtail^* \langle \Box, A, S', B', T'\rangle_{n'}$$

where θ is a matching substitution such that $\mathcal{D} \models B \rightarrow \bar{\exists}_H \theta(\mathsf{sat}_h(o, P_k, P_r))$. Furthermore, $h \triangleq (\mathrm{id}(H), \mathrm{rnum}(o)) \notin T$. If no such matching substitution exists then $\langle o\#i, A, S, B, T\rangle_n \rightarrowtail_{\neg Si} \langle n_f(o)\#i, A, S, B, T\rangle_n$

---

5. Propagate: $\langle o\#i, A, \{c\#i\} \cup S, B, T\rangle_n \rightarrowtail_{Pr} \langle G, A, S_k, B_k, T_k\rangle_{n_k}$
where $o = c : j \in \mathrm{Occ}^k$. Let $S_0 \triangleq \{c\#i\} \cup S$, $B_0 \triangleq B$, $T_0 \triangleq T$, $n_0 \triangleq n$.
Now assume, for $1 \leq l \leq k$ and $k \geq 0$, there is a series of transitions

$$\langle C_l, [o\#i|A], S_{l-1} \setminus P_r^l, B_{l-1}, T_{l-1} \cup \{h_l\}\rangle_{n_{l-1}} \rightarrowtail^* \langle \Box, [o\#i|A], S_l, B_l, T_l\rangle_{n_l}$$

where $H = \{c\#i\} \cup P_k^l \cup P_r^l \subseteq S_{l-1}$ and $h_l = (\mathrm{id}(H), \mathrm{rnum}(o)) \notin T_{l-1}$, and a matching substitution θ_l exists such that $\mathcal{D} \models B_{l-1} \rightarrow \bar{\exists}_H \theta_l(\mathsf{sat}_h(o, P_k^l, P_r^l))$ and $C_l = \theta_l(b(o))$, where θ_l renames apart all variables only appearing in $g(o)$ and $b(o)$ (separately for each l). Furthermore, for $k+1$ no such transition is possible. The resulting goal G is $G \triangleq \Box$ if $\mathcal{D} \models \bar{\exists}_\emptyset(\neg B_k)$ (i.e. failure occurred), $G \triangleq n_f(o)\#i$ if $k = 0$ (i.e. the rule was not applied; in this case we annotate the transition with $\neg Pr$ instead of Pr) and otherwise ($k \geq 1$) $G \triangleq n_s(o)\#i$.

---

6. Goal: $\langle [c|C], A, S, B, T\rangle_n \rightarrowtail_G \langle G, A, S', B', T'\rangle_{n'}$ where $[c|C]$ is a list of built-in and CHR constraints, $\langle c, A, S, B, T\rangle_n \rightarrowtail^* \langle \Box, A, S', B', T'\rangle_{n'}$, and $G \triangleq \Box$ if $\mathcal{D} \models \neg\bar{\exists}_\emptyset B'$ (i.e. calling c caused failure), otherwise $G \triangleq C$.

Fig. 1. Transition rules of ω_o

The transitions in Figure 1 are a formulation of the ω_c semantics in terms of occurrences, except for the Solve transition. In the following, we will consider the original definitions of ω_c semantics [10] and ω_r semantics [4], where the definition for Solve has been replaced by the one described above. We will use $\overset{\omega_o}{\longmapsto}_O$ to

denote $\rightarrowtail^*$ under ω_o semantics for an occurrence representation O, and $\overset{\omega_c}{\longmapsto}_P$ ($\overset{\omega_r}{\longmapsto}_P$) to denote $\rightarrowtail^*$ under ω_c (ω_r) semantics for a CHR program P.

3.4 Properties of ω_o Semantics

Definition 4 (Refined Occurrence Representation). *The refined occurrence representation $O_{\text{ref}}(P) = (g, b, p_r, p_k, n_s, n_f)$ for a CHR program P (notation as in definition 1) is defined as follows: for every occurrence $c : i \in \text{Occ}$:*

$$g(c:i) = g_{\text{rnum}(c:i)} \qquad b(c:i) = B_{\text{rnum}(c:i)}$$
$$p_r(c:i) = H^r_{\text{rnum}(c:i)} \setminus \{c:i\} \qquad p_k(c:i) = H^k_{\text{rnum}(c:i)} \setminus \{c:i\}$$
$$n_s(c:i) = n_f(c:i) = c : (i+1).$$

Theorem 1 (ω_r is equivalent to ω_o for O_{ref}). *For any CHR program P:*
$\langle G, \Box, \emptyset, \emptyset, \emptyset\rangle_1 \overset{\omega_o}{\longmapsto}_{O_{\text{ref}}(P)} \langle \Box, \Box, S, B, T\rangle_n \quad \Leftrightarrow \quad \langle G, \emptyset, \emptyset, \emptyset\rangle_1 \overset{\omega_r}{\longmapsto}_P \langle \Box, S, B, T\rangle_n$

Proof. For a refined occurrence representation $O_{\text{ref}}(P)$, the definition of ω_o transitions corresponds trivially to the definition of ω_c transitions. In [2], a proof is given for the equivalence of ω_c and ω_r semantics. Hence, P under ω_r and $O_{\text{ref}}(P)$ under ω_o are equivalent. □

Note that `pragma passive` constructions can be expressed using occurrence representations: if occurrence o is declared to be passive, it suffices to modify the continuation functions: all occurrences with (fail/success) continuation o should get a new (fail/success) continuation $n_f(o)$. If o is the first occurrence of some constraint, this approach will not prevent o from becoming active. An alternative way to express `pragma passive` is by replacing the guard of o by `fail`.

4 Guard and Continuation Optimizations

In this section, we will present optimizations that simplify the guard function and the continuation functions of (originally refined) occurrence representations, improving efficiency without affecting the behavior of the resulting program.

Definition 5 (Condition Simplification). *Given a condition g which is a conjunction of built-in constraints $g = g_1 \wedge g_2 \wedge \ldots \wedge g_n$ and a condition D, we define* $\mathsf{simpl}(g, D) = g' = g'_1 \wedge g'_2 \wedge \ldots \wedge g'_n$*, where*

$$g'_i \triangleq \begin{cases} \texttt{fail} & \text{if } \mathcal{D} \models D \wedge \bigwedge_{j<i} g_j \rightarrow \neg g_i \\ \texttt{true} & \text{if } \mathcal{D} \models D \wedge \bigwedge_{j<i} g_j \rightarrow g_i \text{ and } \mathcal{D} \not\models D \wedge \bigwedge_{j<i} g_j \rightarrow \neg g_i \\ g_i & \text{otherwise.} \end{cases}$$

In other words, $\mathsf{simpl}(g, D)$ returns the result of removing parts of g that are entailed by D and earlier parts of g. For example:

$$\mathsf{simpl}(X > 3 \wedge Y < 1 \wedge X > 2, X \leq 3 \vee Y < 0) = X > 3 \wedge \texttt{true} \wedge \texttt{true}$$

The following implication is obviously satisfied:

$$\mathcal{D} \models D \rightarrow (g \leftrightarrow \mathsf{simpl}(g, D))$$

Definition 6 (Earlier occurrences condition). *For every occurrence* $o \in \mathrm{Occ}$ *and multiset* S*, we define the following condition:*

$$\mathsf{EOC}(o,S) \triangleq \xi(h(o)) \trianglelefteq \xi(S) \wedge \bigwedge\Big\{\neg\theta(\mathsf{sat}_{kr}(o',S)) \mid o' \prec o \wedge \xi(\theta(o')) \in \xi(h(o))\Big\}$$

where $c:i \prec c:j$ *iff* $i<j$ *and* $(p_r(c:i) \neq \emptyset \vee c:i \in \mathrm{Occ}^r)$*, and* $d:i \prec c:j$ *(for* $d \neq c$*) iff* $\mathrm{rnum}(d:i) < \mathrm{rnum}(c:j)$ *and* $(p_r(d:i) \neq \emptyset \vee d:i \in \mathrm{Occ}^r)$*.*

This is a conjunction of negated matching conditions, for all possible matching substitutions θ and earlier occurrences o'. Intuitively, if S is (a subset of) the constraint store, the $\mathsf{EOC}(o,S)$ condition expresses that no earlier rules were applicable that would have removed a partner-constraint of o. Note that different matching substitutions θ must be considered for a previous occurrence $o'=c:i$ if the head constraints $h(o')$ contain at least two other occurrences of the constraint c or at least two occurrences of another constraint.

If mode or type information is available, it can be added to the $\mathsf{EOC}(o,S)$ conjunction without affecting the following results, as long as this information is correct at any given point in any derivation. For example, the EOC condition for the second occurrence of `sum/2` in Example 2 from Section 2 could be the following (the last part is derived from type information) :

$$\mathsf{EOC}(\texttt{sum(A,B)},S) = \texttt{sum(A,B)} \in \xi(S) \wedge \neg\big(\texttt{A=[]} \wedge \texttt{sum(A,B)} \in \xi(S)\big) \wedge \big(\texttt{A=[]} \vee \texttt{A=[\_|\_]}\big)$$

4.1 Optimizations

Definition 7 (Guard optimization). *Given an occurrence representation* $O=(g,b,p_r,p_k,n_s,n_f)$*. Optimizing the guard of occurrence* o *results in an occurrence representation* $G(O,o)=(g',b,p_r,p_k,n_s,n_f)$*, where* $\forall o' \neq o : g'(o') = g(o')$ *and* $g'(o) = \mathsf{simpl}(g(o), \mathsf{EOC}(o,h(o)))$*.*

Let us illustrate this definition by considering the following example:

```
X in1 A:B <=> A>B | fail.
X in2 A:B <=> A =:= B | X is A.
X in3 A:B, X in4 C:D <=> A<B, C<D | X in max(A,C):min(B,D).
```

Computing $\mathsf{EOC}(\texttt{X in}_3\ \texttt{A:B}, h(\texttt{X in}_3\ \texttt{A:B}))$, we get $\neg(\texttt{A} > \texttt{B}) \wedge \neg(\texttt{A} =:= \texttt{B}) \wedge \neg(\texttt{C} > \texttt{D}) \wedge \neg(\texttt{C} =:= \texttt{D})$. Optimizing the guard of the third occurrence of `in/2` results in the empty guard `true`, because both `A<B` and `C<D` are entailed by the above EOC condition. The (partial) entailment checker we have implemented is strong enough to discover such entailed conditions.

Now consider the EOC condition for occurrence $\texttt{in}_4$: $\mathsf{EOC}(\texttt{X in}_4\ \texttt{C:D}, S) = \neg(\texttt{C} > \texttt{D}) \wedge \neg(\texttt{C} =:= \texttt{D}) \wedge \neg(\texttt{C} < \texttt{D}) \wedge \ldots = \texttt{fail}$. Because anything is entailed by `fail`, the guard of the fourth occurrence is optimized to `fail`. This means we can skip this always-failing occurrence. Note that if we would have used the optimized guard for the third occurrence, we would get an EOC condition containing $\neg\texttt{true}$. The following continuation optimizations modify the continuation functions to skip occurrences like $\texttt{in}_4$.

Definition 8 (Failure Continuation optimization). *Given an occurrence representation $O = (g, b, p_r, p_k, n_s, n_f)$. Optimizing the failure continuation of occurrence o results in an occurrence representation $C_f(O, o) = (g, b, p_r, p_k, n_s, n'_f)$, where $\forall o' \neq o : n'_f(o') = n_f(o')$, and $n'_f(o) = n_f(n_f(o))$ if*

$$\mathcal{D} \models \mathsf{EOC}(n_f(o), h(n_f(o))) \wedge \neg\exists\theta\ \mathsf{sat}_{kr}(o, \theta(h(n_f(o)))) \rightarrow \neg g(n_f(o))$$

(otherwise $n'_f(o) = n_f(o)$).

In the above example, optimizing the failure continuation of $\mathtt{in}_3$ results in $n'_f(\mathtt{in}_3) = \mathtt{in}_5$. Note that $\mathtt{in}_5$ may be a non-existent occurrence.

Definition 9 (Success Continuation optimization). *Given an occurrence representation $O = (g, b, p_r, p_k, n_s, n_f)$. Optimizing the success continuation of occurrence o results in an occurrence representation $C_s(O, o) = (g, b, p_r, p_k, n'_s, n_f)$, where $\forall o' \neq o : n'_s(o') = n_s(o')$, and $n'_s(o) = n_f(n_s(o))$ if*

$$\mathcal{D} \models \mathsf{EOC}(n_s(o), h(n_s(o))) \wedge g(o) \rightarrow \neg\theta(g(n_s(o)))$$

where $\xi(o) = \xi(\theta(n_s(o)))$; otherwise $n'_s(o) = n_s(o)$.

Consider the following example:

```
fib1(0,M) ==> M = 1.
fib2(1,M) ==> M = 1.
fib3(N,M) ==> N > 1 | fib(N-1,M1),fib(N-2,M2),M is M1 + M2.
```

Optimizing the success continuation of $\mathtt{fib}_i$ (for $i \in \{1, 2\}$) results in $n'_s(\mathtt{fib}_i) = \mathtt{fib}_4$. Note that it is not meaningful to optimize the success continuation of an occurrence that is removed by applying its rule.

Note that the definitions of these optimizations crucially depend on entailments. This may be problematic because of the undecidability (in general) and complexity properties of testing entailment. We have implemented an incomplete entailment checker, which exhaustively propagates conditions entailed from the left hand side until the right hand side is found. Its worst-case time complexity is quite bad, but this seems not to be a problem in practice: in most CHR programs, constraints are defined by a small number of rules, so the EOC condition in the left hand side of the entailment is often small.

4.2 Correctness

Because of limited space, we will only present some results, without proof. The proofs are given in [12]. First we introduce the auxiliary notion of EOC-satisfying occurrence representations. Intuitively, such representations have increasing continuation functions and at any point in a derivation, the EOC condition for the active constraint is entailed by the built-in store.

Definition 10 (EOC-satisfying). *An EOC-satisfying occurrence representation $O = (g, b, p_r, p_k, n_s, n_f)$ is an occurrence representation where*

- $\forall\ c:i \in Occ : n_f(c:i) = c:j$ *and* $n_s(c:i) = c:k$ *where* $j > i$ *and* $k > i$;
- *If a derivation reaches an execution state* $\sigma_k = \langle c : j\#i, A, H \uplus S, B, T\rangle_n$, *and* $\xi(H) = \xi(\theta(h(c:j)))$, *then* $\mathcal{D} \models B \rightarrow \mathsf{EOC}(c:j, H)$.

Intuitively it should be clear that the refined occurrence representations from definition 4 have this property.

Lemma 1. *Refined occurrence representations are EOC-satisfying.*

We can show that for EOC-satisfying occurrence representations, the optimizations preserve applicability of ω_o transitions and EOC-satisfiability.

Lemma 2. *Any EOC-satisfying occurrence representation* O *is operationally equivalent w.r.t.* ω_o *semantics (in the strong sense: exactly the same transitions are applicable to any execution state in a derivation) to* $G(O, o)$, $C_s(O, o)$ *and* $C_f(O, o)$, *for every occurrence* o.

Lemma 3. *If an occurrence representation* O *is EOC-satisfying, then* $G(O, o)$, $C_s(O, o)$ *and* $C_f(O, o)$ *are also EOC-satisfying (for every occurrence* o*).*

Combining these results, we get the following correctness result:

Theorem 2 (Correctness). *Repeated application of* G, C_s *and* C_f *to a refined occurrence representation* O *results in an occurrence representation* O' *which is operationally equivalent to* O *w.r.t.* ω_o *semantics.*

5 Implementation and Experimental Evaluation

We have implemented the optimizations in the K.U.Leuven CHR compiler [8], which can be found in recent releases of SWI-Prolog [15]. In our implementation, replacing parts of a rule guard by `true` is called guard simplification (see [13]), while replacing part of an occurrence guard by `fail` is called occurrence subsumption, and it basically causes the occurrence to be declared passive. Guard simplification is a special case of guard optimization, and occurrence subsumption corresponds to failure continuation optimization. Figure 2 illustrates the effect of the optimizations on a small example.

Experimental Results. To get an idea of the efficiency gain obtained by our optimizations, we have measured the performance of several CHR benchmarks, both with and without the optimizations. All benchmarks were performed with SWI-Prolog [15] version 5.5.2, on a Pentium 4 (1.7 GHz) GNU/Linux machine with a low load. We have measured similar results [11] in hProlog [1].

Figure 3 gives an overview of our results. The first column indicates the benchmark name and the parameters that were used. These benchmarks are available at [7]. The second column indicates whether the optimizations were enabled, where "type" means "yes and additional type information was provided". Mode declarations were provided for all programs, which allows a speedup factor

```
:- op(700,xfx,in).

:- constraints
    in(?int,+interval).

:- chr_type
    interval ---> int:int.


X in A:B <=>
    A > B |
    fail.

X in A:B <=>
    A =:= B |
    X is A.

X in A:B, X in C:D <=>
    A < B, C < D |
    X in min(A,C):max(B,D).
```

(a) Example CHR program.

```
X in I :-
   'in/2__0'(X,I,_).

'in/2__0'(_,A:B,Z) :-
   A > B, !,
   remove_constraint(Z),
   fail.

'in/2__0'(X,A:B,Z) :-
   A =:= B, !,
   remove_constraint(Z),
   X is A.

'in/2__0'(X,A:B,Z) :-
   A < B,
   find_partner(in(Y,I)),
   Y == X,
   I = C:D, C < D, !,
   remove_constraint(Z),
   X in min(A,C):max(B,D).

'in/2__0'(X,C:D,Z) :-
   C < D,
   find_partner(in(Y,I)),
   Y == X,
   I = A:B, A < B, !,
   remove_constraint(Z),
   X in min(A,C):max(B,D).

'in/2__0'(X,I,Z) :-
   insert_constraint(Z)
```

(b) Not optimized.

```
X in I :-
   'in/2__0'(X,I,_).

'in/2__0'(_,A:B,Z) :-
   A > B, !,
   remove_constraint(Z),
   fail.

'in/2__0'(X,A:B,Z) :-
   A =:= B, !,
   remove_constraint(Z),
   X is A.

'in/2__0'(X,J,Z) :-

   find_partner(in(Y,I)),
   Y == X,
   !,
   remove_constraint(Z),
   J = A:B, I = C:D,
   X in min(A,C):max(B,D).

'in/2__0'(X,I,Z) :-
   insert_constraint(Z)
```

(c) Optimized.

Fig. 2. Comparing generated code. Note that the redundant clause for the fourth occurrence of `in/2` and the redundant guard of the last rule are removed in (c).

Benchmark	*Optimize*	*# clauses*	*# lines*	*Runtime (%)*	
sum	no	3	10	5.03	(100)
(10000,500)	type	2	6	4.49	(89)
nrev	no	6	20	13.97	(100)
(30,50000)	type	4	11	8.44	(60)
dfsearch	no	4	16	37.58	(100)
(16,500)	yes	4	15	31.63	(84)
	type	3	11	29.97	(80)
bool_chain	no	180	2861	12.8	(100)
(200)	yes	147	2463	7.0	(55)
fib	no	10	154	11.2	(100)
(22)	yes	9	125	8.5	(76)
leq	no	18	218	14.1	(100)
(60)	yes	13	162	11.7	(83)

Fig. 3. Benchmark results

of two to three [13] in these cases. We have measured the additional speedups on top of the speedups we get from using mode information. The next two columns show the size of the generated Prolog code, not including constraint-store related auxiliary predicates. The last column shows the runtime in seconds and a percentage comparing it to the non-optimized version.

We compared the `sum`, `nrev` and `dfsearch` benchmarks to a native Prolog version of the program. The native Prolog version turned out to be almost identical to the generated code for the CHR program, the only difference being some redundant cuts (`!/0`) in the latter. We could not measure any difference in runtime. There is no straightforward way to make a native Prolog version of the `bool_chain`, `fib` and `leq` benchmarks, since they crucially depend on storing constraints.

Overall, for these benchmarks, doing guard simplification and occurrence subsumption — combined with never-stored analysis and use of mode information to remove redundant variable triggering code — results in cleaner and more efficient code which is much closer to what a Prolog programmer would write. As a result, a major performance improvement is observed in these benchmarks.

Writing Auxiliary Predicates in Prolog. CHR programs that implement deterministic algorithms (like the first three benchmarks) have a relatively low performance when compiled naively using the general schema, compared to native Prolog versions. For that reason, CHR programmers usually write such algorithms as auxiliary predicates in Prolog instead of formulating them as CHR constraints. Such mixed-language programs often use inelegant constructs, like rules of the form `foo(X) \ getFoo(Y) <=> Y = X`, to read information from the constraint store in the host-language parts when this information is needed. By implementing these parts as multi-headed CHR rules, the need for 'host-language interface' constraints like `getFoo/1` is drastically reduced. Thanks to our new optimizations and other analyses, the programmer can now implement the entire program in CHR, relying on the compiler to generate efficient code.

6 Conclusion

We have described new guard and continuation optimizations. We have defined them formally and showed their correctness, implemented them in K.U.Leuven CHR compiler and evaluated them experimentally. Guard optimization encourages CHR programmers to include all preconditions for rule application in the rule guards, since redundant tests are compiled out. Continuation optimization dramatically reduces the need for `pragma passive` directives. Hence, the optimizations allow writing CHR programs that are more declarative, readable and self-documenting, without sacrificing efficiency.

Our new optimizations contribute to the state-of-the-art level of performance of code generated by the K.U.Leuven CHR compiler. Guard optimization reduces the overhead of testing redundant guard conditions, while continuation optimization reduces the overhead of trying rules that are not applicable. Furthermore,

our optimization helps other analyses (like the never-stored analysis) to reduce constraint store related overhead. Earlier work introduced mode declarations used for hash tabling and other optimizations. We have added type declarations for CHR programs. Using both mode and type information we have realized further optimization of the generated code.

Related Work. The idea of continuation optimization was originally introduced in [6]. Guard optimization originates from [11,13], where a weaker optimization called guard simplification was introduced. Guard simplification is basically guard optimization for every occurrence in a rule, which is weaker than guard optimization in the case of multi-headed rules.

Future Work. It would be interesting to implement a stronger version of $\mathsf{simpl}(g, D)$ (Definition 5, page 90) by replacing an expensive condition g_j by a cheaper condition g'_j, as long as $\mathcal{D} \models D \wedge \bigwedge_{k<j} g_k \rightarrow (g_j \leftrightarrow g'_j)$. For example, consider the following program:

```
p(X) <=> X >= 0, g(X) | ...
p(X) <=> X < 0, \+ g(X) | ...
p(X) <=> g(X) | ...
```

If `g/1` is a predicate that takes a very long time to evaluate, we could change the guard of the last rule to `X<0`, because $\neg(X >= 0 \wedge g(X)) \wedge \neg(X < 0 \wedge \neg g(X))$ entails $g(X) \leftrightarrow X < 0$.

Our current knowledge base for entailment checking is limited to the most common Prolog built-ins. To be able to recognize more redundant guards, one could extend this knowledge base, add support for additional declarations that would be added to the knowledge base during the program analysis, and even analyze the host-language implementation of user-defined predicates used in guards, inferring extensions to the knowledge base automatically.

When the "earlier occurrences condition" is large, compilation time may become an issue. We intend to improve the scalability of our implementation, although it does not present an immediate problem.

The information entailed by the failure and success of guards seems also useful in other program analyses and transformations. One application is program specialization: the code for executing a constraint is specialized for a particular call from an occurrence body. This may lead to the elimination of more redundant guards (and even redundant rules) for the specialized case.

Finally we would like to integrate our optimizations into the bootstrapped CHR compiler which is currently being developed by Christian Holzbaur et al.

References

1. Bart Demoen. hProlog home page. http://www.cs.kuleuven.ac.be/~bmd/hProlog.
2. Gregory Duck, Tom Schrijvers, and Peter Stuckey. Abstract Interpretation for Constraint Handling Rules. Report CW 391, K.U.Leuven, Department of Computer Science, Leuven, Belgium, September 2004.

3. Gregory Duck, Peter Stuckey, María García de la Banda, and Christian Holzbaur. Extending Arbitrary Solvers with Constraint Handling Rules. In D. Miller, editor, *Proceedings of the 5th International Conference on Principles and Practice of Declarative Programming (PPDP'03)*. ACM Press, 2003.
4. Gregory Duck, Peter Stuckey, María García de la Banda, and Christian Holzbaur. The Refined Operational Semantics of Constraint Handling Rules. In *Proceedings of the 20th Intl. Conference on Logic Programming (ICLP'04)*, September 2004.
5. Thom Frühwirth. Theory and Practice of Constraint Handling Rules. In P. Stuckey and K. Marriot, editors, *Special Issue on Constraint Logic Programming, Journal of Logic Programming*, volume 37 (1–3), October 1998.
6. Christian Holzbaur, María García de la Banda, Peter Stuckey, and Gregory Duck. Optimizing compilation of Constraint Handling Rules in HAL. In *Special Issue of Theory and Practice of Logic Programming on CHR*, 2005. To appear.
7. Tom Schrijvers. CHR benchmarks and programs. Available at the K.U.Leuven CHR home page at http://www.cs.kuleuven.ac.be/~toms/Research/CHR/.
8. Tom Schrijvers and Bart Demoen. The K.U.Leuven CHR system: implementation and application. In T. Frühwirth and M. Meister, editors, *First Workshop on Constraint Handling Rules: Selected Contributions*, number 2004-01, 2004.
9. Tom Schrijvers and Thom Früwirth. Implementing and Analysing Union-Find in CHR. Technical Report CW 389, K.U.Leuven, Dept. Computer Science, July 2004.
10. Tom Schrijvers, Peter Stuckey, and Gregory Duck. Abstract Interpretation for Constraint Handling Rules. In *Proceedings of the 7th Intl. Conference on Principles and Practice of Declarative Programming (PPDP'05)*, Lisbon, Portugal, July 2005.
11. Jon Sneyers, Tom Schrijvers, and Bart Demoen. Guard Simplification in CHR programs. Technical Report CW 396, K.U.Leuven, Dept. CS, November 2004.
12. Jon Sneyers, Tom Schrijvers, and Bart Demoen. Guard and Continuation Optimization for Occurrence Representations of CHR. Technical Report CW 420, K.U.Leuven, Dept. CS, July 2005.
13. Jon Sneyers, Tom Schrijvers, and Bart Demoen. Guard Simplification in CHR programs. In *Proceedings of the 19th Workshop on (Constraint) Logic Programming (W(C)LP'05)*, Ulm, Germany, February 2005.
14. Zoltan Somogyi, Fergus Henderson, and Thomas Conway. Mercury: an efficient purely declarative logic programming language. In *Proceedings of the 18th Australian Computer Science Conference*, pages 499–512, Glenelg, Australia, 1995.
15. Jan Wielemaker. SWI-Prolog home page. http://www.swi-prolog.org.

Coordination of Many Agents

Joxan Jaffar, Roland H.C. Yap, and Kenny Q. Zhu

School of Computing, National University of Singapore,
Republic of Singapore
{joxan, ryap, kzhu}@comp.nus.edu.sg

Abstract. This paper presents a reactive programming and triggering framework for the coordination of a large number of distributed agents with shared knowledge. At the heart of this framework is a highly structured shared store in the form of a constraint logic program (CLP), which is used as a knowledge base and being reacted to by agents through the use of "reactors". The biggest challenge arising from such a reactive programming framework using CLP is to develop a trigger mechanism that allows efficient "wakeup" of blocked reactors. This paper addresses the architecture of this open framework, and discusses a general methodology for doing triggering of logic conditions using views and abstractions.

1 Introduction

In online applications such as an automated marketplace, many agents with shared knowledge need to interact and synchronize with each other by reacting to some conditions. The agents block when the condition they are waiting for is not satisfied and unblock when the condition becomes true some time later. Because the number of agents participating in these activities is very large, and the blocking conditions may be very complex, existing technologies such as blackboard architectures [10] and active databases [15] are inadequate.

This paper introduces a shared-store programming framework for interacting distributed agents which combines the power of Constraint Logic Program (CLP) [8] and the *triggering* of complex conditions.

The framework allows agents to *react* to their environments taking into account complex conditions, and allows for coordination with each other through a structured shared store, the *knowledge base*, represented by a CLP. The distributed agents interact with the shared CLP store via embedded program fragments called *reactors* programmed in a simple stylized concurrent language. The key features of this language include the use of CLP goals as *guards* and the use of *committed choice*. CLP goals as guards give a unique way of handling reactivity in distributed programs which leverages CLP's power of declarative semantics, databases and complex *views*.

Since reactors may block on complex logical conditions and there are many of them, the key technical challenge is how to identify efficiently a set of reactors, among all the blocked reactors, that need to be checked for "wakeup" given a state change to the CLP store, e.g. update of a base predicate. We call this process *triggering*. This is more

M. Gabbrielli and G. Gupta (Eds.): ICLP 2005, LNCS 3668, pp. 98–112, 2005.

difficult than determining when a guard becomes entailed in Concurrent Constraint Programming (CCP) [12] as the CLP program itself can change and the test is not simply entailment.

The idea behind efficient triggering is to exploit a notion of *locality*. To explain a simple form of locality, consider the popular online chat client, MSN messenger. While there may be millions of users logged on at the same time, every user typically only has a small contact list. When a user logs on (which can be viewed as a state change), it is easy to exhaustively search the vicinity of this user (in this case, her contact list) to find out which contacts are online and notify (trigger) those people. However, there are more complex forms of locality for which no such efficient algorithms are readily at hand. For example, a marketplace has *frequent* and *diverse* updates; an individual trader, however, is typically interested in a small fraction of these. Given an update, it is difficult to identify the few interested traders.

In this paper, we deal with more complex forms of locality. We employ a general assumption that *any single state change in the CLP is unlikely to affect a large portion of the blocked reactor*, therefore it pays to design an index structure for the reactor conditions so that triggering can be done efficiently as a search in the index. To facilitate the indexing of complex conditions, the trigger framework exploits the semantics of CLP views in *abstracting* complex views to simple ones. It is also possible to develop analysis tools with CLP technology to verify the correctness of such abstraction.

Since this framework is completely *open*, in the sense that any agents can interact with the system at any time, and CLP goals are used in reactive conditions, we call this framework *Open Constraint Programming* or OCP. It is also an open system in the sense that the agents are language neutral. We argue that open, reactive, and concurrent systems with powerful modeling capabilities are useful for distributed coordinated applications such as those in E-commerce.

This paper addresses the design of the architecture and the reactor language, and focuses in particular on a triggering framework and methodology which are essential for managing large number of reactors. The main contribution of this paper is two-fold:

- the use of CLP as a knowledge base and for reactivity in an open programming framework; and
- a general triggering framework for re-enabling a small set of reactors, among a large number of blocked ones.

1.1 Related Work

The OCP framework is related to shared-memory concurrent languages such as CCP [12] and blackboard architectures [10]. In the CCP framework (including GHC [14] and Oz [13]), processes communicate by interacting with a set of shared variables in a *store* on which they can either post ("tell") or test ("ask") for the presence of some constraint. These languages also use committed choice for non-determinism. However, the constraint store is monotonic in CCP, whereas in OCP, the store is a knowledge base and can be non-monotonic. Just as in a database or blackboards, the store needs to be non-monotonic as it is meant to be stateful.

Blackboard languages such as Linda [7] and ActorSpace [1] use a set of tuples or actors as medium of communication and synchronization. OCP can be thought of as

generalizing Linda's tuple space, to use more powerful constructs and going from a structured shared memory store to richer store which uses CLP. By doing so, the primitive operations on the store can utilize complex reasoning to express how the agents interact and synchronize. OCP is also related to active databases [15] because the ECA rules can be treated as the special case of a basic kind of OCP reactors. Triggering is required in both active databases and OCP, though the active database has a much simpler trigger paradigm, which is on simple events such as insertion/deletion to the base tables. In particular, active databases do not address our problem of minimizing the cost of triggering.

Another class of reactive languages are the synchronous languages such as Esterel [2], Lustre [3] and SIGNAL [6]. These languages are designed for reactive systems in which reaction is *instantaneous*. These synchronous systems are also *deterministic*, while CCP and OCP are non-deterministic.

In what follows, we will discuss the architecture of OCP, which includes the reactor programming language and motivating examples, the triggering mechanism which is critical for the efficient implementation of a runtime system, and finally a description of an initial prototype system with some experimental results.

2 Architecture

In the basic OCP framework (Fig. 1), every software agent consists of a program written in a suitably convenient language. Small program fragments, which we call *reactors*, written in an OCP reactor language, are embedded in the agent program. This is similar to how one can embed SQL in a host program. When the agent wants to execute a reactor, this reactor is submitted to the OCP runtime system, in a similar fashion to a remote procedure call.

The purpose of the reactor is to perform some actions to the shared CLP store or knowledge base. In the rest of this paper, we will use the notation Δ to refer to the shared CLP store (this suggests it is stateful). The action may be guarded by some conditions defined in the reactor as well as other logic defined in Δ. We expect that some part of the requirements for the condition may be expressed in the submitted reactor and the rest of it could be reasoning expressed in the knowledge base itself. We use a rather

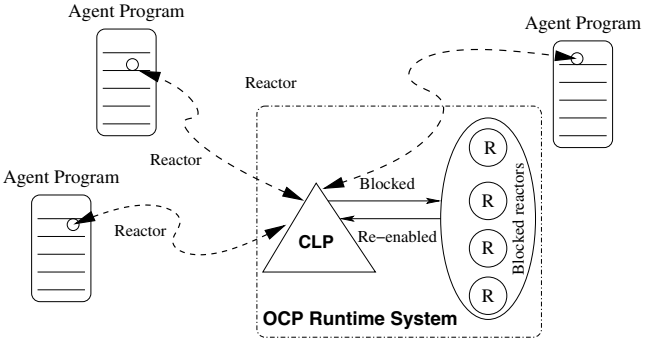


Fig. 1. The basic model of OCP

general form of the condition which is any CLP goal expressed in the language of the knowledge base. A typical action may be to update or delete some data in the store and the condition could be some consistency condition, e.g. ensure minimum balance in a bank account. The action can only be performed if the knowledge base Δ is consistent with the guard. When that is not the case, the reactor blocks until some change in Δ makes the condition true, we say the reactor is *re-enabled* and can be executed as long as the condition is true. One can express concurrent alternatives in a reactor, committed choice is used to control the non-determinism arising from the alternatives. As a reactor behaves like a remote procedure call from the agent program, it only returns when the submitted reactor has completely finished executing its reactor program.

Since the knowledge base Δ is non-monotonic, we can think of it as consisting of some static predicates which do not change and some dynamic predicates which can be changed by a reactor. In this paper, we consider dynamic predicates to be ground facts, which we call *base predicates*.

2.1 Syntax and Semantics

We now give the inductive definition of a simple reactor language as follows. Let r be a reactor, then

r	$::=$	
	δ	atomic update to Δ
$\mid$	$r_1; r_2$	sequence
$\mid$	$r_1 \mid\mid r_2$	choice
$\mid$	`commit`	commit the enclosing choice
$\mid$	$c \Rightarrow \delta$	guarded actions

The reactors are intended to be embedded in agent programs. The sequence construct is self-explanatory, and we just focus on atomic update, choice/commit and guarded action. We define the operational semantics of these constructs using the reactor transition relations $r,\ \Delta \xrightarrow{\delta} \Delta',\ r'$, where δ is an atomic action that changes the store from Δ to Δ', and r progresses to r'.

- An atomic update is an action that consists of one or more of the following subactions in an *atomic* sequence: an insertion, a deletion or an update to a base predicate. Often the sequence is coded in a CLP goal to be executed atomically by the runtime system. While, in principle, these operations can be used on any predicate in Δ, in this architecture, we restrict the atomic updates to only base predicates. The rule for atomic update is

$$\delta, \Delta \xrightarrow{\delta} \delta(\Delta)$$

- The choice construct provides a form of early-committed non-determinism. The semantics of choice is that both branches execute concurrently until one of them makes an update to Δ or issues a `commit`. At that time, the other choice branch is aborted with no effect on Δ. The `commit` operation can be thought of as a special atomic update to Δ which has no effect, like a `noop`. Let ϵ denote an action that does not change the store, such a read; and let u denote an update that changes the store or a `commit`, the rules for the choice construct are

$$\frac{r_2, \Delta \xrightarrow{\epsilon} \Delta, r_2'}{r_1 \mid\mid r_2, \Delta \xrightarrow{\epsilon} \Delta, r_1 \mid\mid r_2'} \quad \frac{r_2, \Delta \xrightarrow{u} \Delta', r_2'}{r_1 \mid\mid r_2, \Delta \xrightarrow{u} \Delta', r_2'} \quad \frac{r_2, \Delta \xrightarrow{u} \Delta'}{r_1 \mid\mid r_2, \Delta \xrightarrow{u} \Delta'}$$

– Guarded actions are used for synchronization or for ensuring consistency conditions. c is called a *blocking condition* or simply a *condition*. A guarded atomic update, $c \Rightarrow \delta$, blocks until condition c is true, i.e. $\Delta \models c$, and then atomically performs the update δ. In particular, c is any CLP goal defined over Δ which is evaluated by the CLP system. Variables in condition c are in the same scope as the action and can be used to bind variables in the action. The rule for guarded actions is

$$\frac{\delta, \Delta \xrightarrow{\delta} \Delta'}{c \Rightarrow \delta, \Delta \xrightarrow{\delta} \Delta'} \quad \text{if } \Delta \models c$$

2.2 A Motivating Example

We use the following example of shipping marketplace as a motivating example for OCP throughout this paper. The agents interacting with the marketplace are clients who want to ship cargo and transportation companies which offer cargo ships of various load capacity and sailing schedules. The knowledge base is a CLP program which contains static facts of a distance table (map) among cities, and dynamic facts about the availability of the vessels, such as the following.

```
map(seoul, shanghai, 4668).
vessel('star', hongkong, shanghai, 20000, 0.012, 15, 18).
```

The `map` predicate records the distance between two cities, e.g. the distance between Seoul and Shanghai is 4668 km. The `vessel` predicate specifies a vessel named "star", which is scheduled to go from Hong Kong to Shanghai, with a load capacity of 20000 tons, and a shipping price of 1.2 cent per ton per km. It will depart at time 15 and arrive at time 18. The departure and arrival time along with the distance table implies travel speed of the vessel. We assume that the unit price for shipping is roughly proportional to the speed of travel.

A client wants to ship cargo from place A to B, either directly or via some other transit points, by a certain deadline and within budget. Constraints are on the load capacity of the vessel and the feasibility of arrival/departure times. The reactivity arises because it may not be possible to ship the cargo given the existing state of the store, however, changes to the store may make the request feasible. Clients will update the capacity as they are committed to a particular vessel.

The relation (and a knowledge base) `deliverable` is used to specify blocking conditions c of the clients' reactors. It returns `Dep` and `Arr` as the departure and arrival times for tracking, and a list of vessel identifiers.

```
%base case for one segment
deliverable(A, B, Weight, Budget, Deadline, Dep, Arr, [ID]):-
  Budget>0, Weight>0,
  LoadCap>=Weight, Weight*Dist*Price<=Budget, Arr<=Deadline,
```

```
  map(A, B, Dist),
  vessel(ID, A, B, LoadCap, Price, Dep, Arr).

%base case for two segments: A-C and C-B
deliverable(A, B,Weight,Budget,Deadline,Dep, Arr, [ID1, ID2]):-
  Budget>0, Weight>0,
  LoadCap1>=Weight, LoadCap2>=Weight,
  Weight*(Dist1*Price1+Dist2*Price)<=Budget,
  Arr1<=Dep2, Arr2<=Deadline,
  map(A, C, Dist1), map(C, B, Dist2),
  vessel(ID1, A, C, LoadCap1, Price1, Dep1, Arr1),
  vessel(ID2, C, B, LoadCap2, Price2, Dep2, Arr2).

%recursive case: A...C-D...B
deliverable(A, B, Weight, Budget, Deadline, Dep, Arr, L):-
  LoadCap>=Weight, Weight*Dist*Price<Budget, Dep2>=Arr1,
  map(C, D, Dist),
  deliverable(A, C, Weight, Budget1, Dep1, Dep, _, L1),
  vessel(ID, C, D, LoadCap, Price, Dep1, Arr1),
  deliverable(D, B, Weight, Budget-Budget1-Weight*Dist*Price,
            Deadline, Dep2, Arr, L2),
  L=concat(L1, ID, L2).
```

There is more than one way to define `deliverable`. Here, we choose to define base cases of one segment and two segments, and then a recursive rule that consists of a path from A to C, a segment C to D and another path from D to B. The reason for this set-up is to have more efficient triggering which will become clear in the next section. When the `deliverable` condition is satisfied, the cargo can be shipped and the `do_ship` action will update the corresponding capacities along the route.

A client who wants to ship a cargo weighing 100 tons from Singapore to Seoul by time 25 and with maximum $5000, can submit the following reactor to the OCP system:

$$\texttt{deliverable}(\mathit{singapore},\ \mathit{seoul},\ 100,\ 5000,\ 25,\ D,\ A,\ \mathit{IDs}) \Rightarrow \\ \texttt{do\_ship}(\mathit{singapore},\ \mathit{seoul},\ 100,\ \mathit{IDs})$$

From the above example, we argue that there exists a large class of applications like the shipping marketplace where the use of a CLP program as a knowledge base and for reactivity is not only elegant but, we believe, also essential. The recursion and constraint solving capability of a CLP offers an extremely concise but expressive way to specify general logical rules, such as `deliverable`, to be used by many different reactors from different agents with their own instantiations or possible additional constraints.

2.3 The Runtime System

This section describes the design of the runtime system architecture. Central to the system design is the notion of *triggering*. A trigger model determines which blocked reactors to *fire* given an update δ to the knowledge base Δ. When a reactor is fired, it is re-enabled and starts execution by re-evaluating the blocking condition which failed earlier, and if it succeeds, proceeds to executing the action.

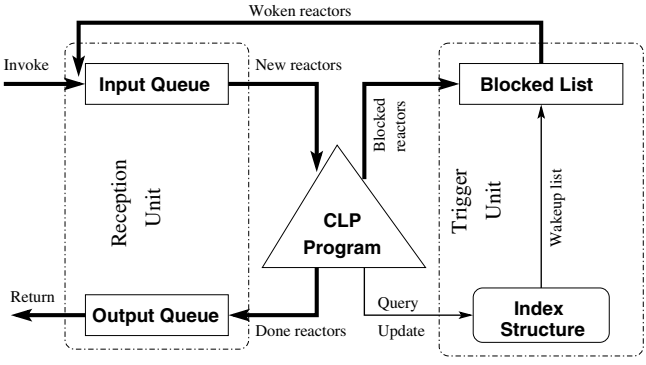


Fig. 2. The registry

To manage the execution, blocking and wakeup of the reactors, the runtime system employs a registry that wraps around the CLP system and the blocked reactors. The registry is composed of the following elements (see Figure 2):

1. a CLP program loaded in a CLP system;
2. a receptionist that handles the I/O of reactors; and
3. a trigger unit for triggering blocked reactors.

The dark arrows in the figure represent the flow of reactors in the system. A reactor is executed against the CLP store once it enters the registry. The blocking conditions and the actions in the reactors are implemented as CLP goals which are executed by a CLP system. If the blocking condition is satisfied, the action is executed, probably updates the CLP and exits the system; otherwise, the reactor is blocked in the blocked list, until the condition becomes true in the future.

We suggest that this architecture is extremely versatile because CLP can integrate databases with logic programs, constraints and concurrency. At one end of the spectrum, a CLP program can be reduced to just ground facts which is similar to Linda. At the other end, it can be a full-fledged knowledge base complete with a reasoning system and constraint solvers.

3 Triggering

The problem of triggering is present in many applications and scenarios. Consider the popular online application MSN messenger. One of its features is when a user Jane logs in, all her contacts who are currently online must be notified, or *triggered*. As we know, MSN messenger has millions of users online at any time, certainly we don't want to test every online user to see if he or she is a friend of Jane. In this particular case, a simple hash-based triggering can be used as any user's contact list is typically small, in fact, bounded. But in general, the problem of determining just which agents are to be triggered by an often-occurring event is intractable.

In this section, we discuss the triggering problem for OCP, and present a methodology for dealing with it.

3.1 Views and Blocking Conditions

The basic problem can be defined as follows: given an update δ to a base predicate of the CLP knowledge base, and given a set of blocked conditions $\mathcal{C}$ which are currently false, efficiently return a subset of $\mathcal{C}$ which become true as a result of the update.

To facilitate discussion below, let us first define:

Definition 1 (View). *A* view *is simply a rule defining a distinguished set of non-base predicates. It has the general form:*

$$p(\tilde{X}_0) :\text{-} q_1(\tilde{X}_1),\ q_2(\tilde{X}_2),\ \ldots,\ q_n(\tilde{X}_n),\ \Psi(\tilde{X}_0, \cdots, \tilde{X}_n). \tag{1}$$

where p is not a base predicate. We say that this view is basic *if the $q_i, 1 \leq i \leq n$, are all base predicates. Otherwise, we say that the view is* composite.

Note that not all CLP predicates provide views. Views are essentially interface predicates for the agents to interact with the CLP program. The blocking conditions of reactors are defined based on views.

Definition 2 (Blocking Condition). *A blocking condition c is of the form:*

$$p(\tilde{X}),\ \Psi(\tilde{X}),$$

where p is a view on variables $\tilde{X}$, and $\Psi(\tilde{X})$ is a constraint. We say a blocking condition is basic (composite) if the view it refers to is basic (composite).

Typically, $\Psi(\tilde{X})$ specifies a value or a range for some of the variables in $\tilde{X}$, such as $c ::= p(X, Y),\ X = 5,\ 0 \leq Y \leq 5$.

Definition 3 (Induced View). *Let p be a view of the form $p(\tilde{X}) :\text{-} Body$. Let c be a blocking condition $p(\tilde{X}),\ \Psi(\tilde{X})$. The view of p* induced *by c is the rule*

$$p(\tilde{X}) :\text{-} Body, \Psi(\tilde{X}).$$

To determine if a blocking condition c on a view p is enabled by an update δ is in general an undecidable problem. Naively, one can execute c as a goal against the newly updated CLP knowledge base. This is tantamount to testing if the induced view of c has any solutions.

Running induced views is, unfortunately, unacceptable if c is a composite condition that depends on complex views whose resolution is very expensive, e.g. when recursive joins are involved. Preferably, we could discover some constraints, from the definitions of both c and δ, which could answer this question directly. This is clearly more desirable, and this optimization represents our first objective.

However, if the total number of blocked reactors is very large, even this optimization is insufficient, because having to consider every blocking condition is prohibitively expensive (recall the MSN example). We therefore seek to build an *index* for the blocked conditions so that large number of conditions can be excluded from an update without testing any one of them. Constructing this index thus becomes our second optimization objective.

In what follows, we will first show how to index basic blocking conditions by a spatial index structure called the *RC-tree*. We then show how to reduce composite views to basic ones so that the RC-tree can be used.

3.2 The RC-Tree

This section considers the problem of indexing multi-dimensional geometrical objects. There is a wealth of publications in the the area of spatial databases [5] and computational geometry [11]. However, these spatial indexes, especially those used for geographic information systems applications, assume little or no overlapping among the objects, and when the objects are large, static segmentation is used to reduce large objects to many small rectangles, which increases the space and insertion cost. In addition, they index the Minimum Bounding Rectangles (MBRs) of the original objects, rather than the objects themselves. The original shapes of the objects are thus lost and not made use of in such approximation.

We propose a new spatial index structure, called RC-tree, which is better suited for indexing dynamic, overlapping regions. RC-tree is a clipping-based spatial index which combines some features of the kd-tree and the R^+-tree. Every intermediate node of a RC-tree is a hyper-plane that partitions the space assigned to this node. The space is thus divided into two sub-spaces. All objects entirely contained in the left half-space will be stored in the left sub-tree at the node; and all objects contained in the right half-space go into the right sub-tree. If an object intersects the hyperplane, it is clipped and the two resulting clipped objects go into respective subtrees where they belong. The root node is assigned the entire space.

The novelty of the RC-tree is that instead of indexing MBRs of the objects, it indexes the actual shape of the objects, and dynamically clips the objects on demand when there is need to discriminate a number of them. This enables the RC-tree to index objects of large extension and with heavy overlapping.

RC-tree's dynamic clipping can be seen as doing the segmentation dynamically and on demand. A very important technique used in the RC-tree is *domain reduction* which dynamically updates the MBRs of clipped objects such that insertion and search costs as well as space requirements are reduced. In the left part of Fig. 3, domain reduction strategy creates a tree with only two items ("L1" and "L2") in the leaves. In the right part of the figure, the other tree, which is similar to the R^+-tree, has four items (denoted

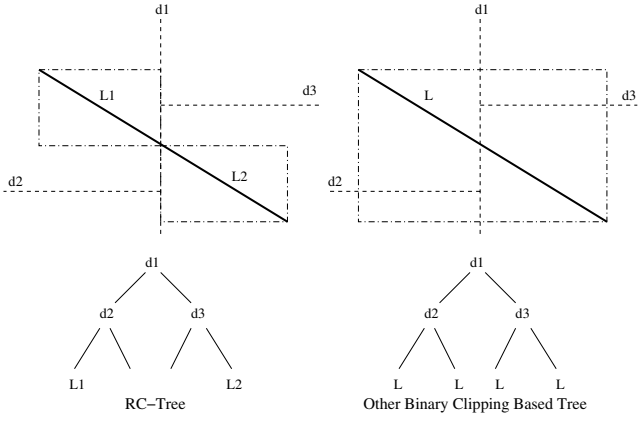


Fig. 3. Advantage of clipping and domain reduction in insertion

by "L"). In addition, during the insertion, the RC-tree clips object "L" and inserts the two sub-objects "L1" and "L2" into the tree while the other strategy inserts four times.

We have conducted a range of traditional and synthetic benchmarks on the RC-tree, and have observed an amortized $\log(n)$ insertion time, amortized $\log(n)$ point query time, where n is the number of objects to be indexed. Our experiments also show that RC-tree performs much better in query performance than other R-tree variants and the quad-tree.

3.3 Basic Views

The RC-tree can be used to efficiently index multi-dimensional shapes and to search using a query of the same dimensionality.

Let $\Psi(\tilde{X})$ be a constraint and q a base predicate. Then the basic condition $q(\tilde{X}), \Psi(\tilde{X})$ can be treated as a geometrical shape in an RC-tree for $\tilde{X}$. Accordingly, the update on $q(\tilde{X})$ where $\tilde{X}$ has been grounded can be treated as a query to that RC-tree. Therefore, reactor conditions on a single base predicate can be indexed and triggered using the RC-tree in a straightforward manner.

Consider an example of such a basic blocking condition:

```
vessel(_,A,B,C,P,_,_), A=beijing, B=taipei, C>=500, P<=0.02.
```

One can construct a 4-dimensional shape on (A, B, C, P) such that

$$(A =' beijing') \wedge (B =' taipei') \wedge (C \geq 500) \wedge (P \leq 0.02)$$

and index shapes like this in a 4-d RC-tree on variables (A, B, C, P). When a new vessel becomes available or an existing vessel changes, variables (A, B, C, P) get updated simultaneously. The ground values (A, B, C, P) can then be used as a point query to the RC-tree index. For example, an update of

```
vessel('dragon', beijing, taipei, 10000, 0.015, 23, 40)
```

is one of such updates that would enable the above blocking condition.

Another type of basic condition is of the form $p(\tilde{X}_0), \Psi(\tilde{X}_0)$, where p is a basic view, which means

$$p(\tilde{X}_0) \text{:-} q_1(\tilde{X}_1),\ \ldots,\ q_n(\tilde{X}_n),\ \Psi_0(\tilde{X}_0, \cdots, \tilde{X}_n),$$

where q_1 through q_n are all base predicates. Of course, one can immediately replace $p(\tilde{X}_0), \Psi(\tilde{X}_0)$ by

$$q_1(\tilde{X}_1),\ \ldots,\ q_n(\tilde{X}_n), \Psi'(\tilde{X}_0, \cdots, \tilde{X}_n),$$

where

$$\Psi'(\tilde{X}_0, \cdots, \tilde{X}_n) = \Psi_0(\tilde{X}_0, \cdots, \tilde{X}_n) \wedge \Psi(\tilde{X}_0).$$

For example,

$$c ::= q_1(X),\ q_2(Y),\ X + Y = 10,\ X \geq 0,\ X \leq 5.$$

The condition c can be formulated as a shape in the (X, Y) space, and an RC-tree can be built for shapes in the (X, Y) space. The problem is, updates are only on `q1/1`

and `q2/1` separately, which means either X or Y is updated at a time, but not both. Therefore, we cannot construct a complete query of (X, Y), but instead we have either $(x,\ *)$ or $(*,\ y)$, where * denotes unknown values. There are two possible ways to solve this problem. The first and the "default" method is to construct a range query using a wildcard for the variable that is not instantiated. For example, if `q1(5)` is written to the CLP, a query $(5,\ *)$, which is essentially an infinite range query to the RC-tree, can be produced to query the index tree. This method is applicable but not always effective. Suppose the blocked conditions are all binary constraints on X and Y, and there is no bound on X, then $(5,\ *)$ will not be able to discriminate any shapes in the index, and hence all conditions will be triggered.

The second way is to instantiate or constrain some of the unknown variables at the time when an update occurs. This is possible if there exists in the CLP a constraint or functional relationship between the value of the known variable (X) and the value of unknown (Y). For example, if the following rule exists in the CLP:

```
q2(Y):- q1(X), Y = 2*X+1.
```

Then given $X = 5$, the system can infer by the above rule that $Y = 11$, and thus produce a complete query $(5, 11)$.

Alternatively, if there exists a constraint between X and Y such as,

```
q2(Y):- q1(X), Y<=X.
```

then a finite range query can be produced: $((5,\ Y) :\ Y \leq 5)$, which is more specific and effective than querying with $(5,\ *)$.

We conclude this subsection with a few comments on the issues of *aggregation* and *materialization*. Aggregation is a concept that originates from the relational databases. An aggregate is a function of some tuples in the same relation. Common aggregation functions such as *min*, *max*, *average* can be computed incrementally and are included in some versions of CLP as system predicates or as meta-level predicates. In the shipping example, the prices and speeds of vessels vary over time, but their ranges can be defined as aggregates of the `vessel` predicate using the *min/max* functions.

When a blocking condition contains a view that uses aggregation, how do we deal with it? One way to handle aggregation is to materialize the value of aggregates if they are not changed often, such as price ranges of all vessels. Once the aggregate values are materialized, they can be treated as constants and used in the basic views. However, when the aggregate value *does* change later, the views constructed based on the materialized values must be updated. This may involve deleting of the corresponding shape from the index, reconstructing it and then re-inserting it into the index.

3.4 Composite Views

Having shown how to index and query basic conditions in the last section, this section considers a methodology to reduce composite conditions to basic conditions so that they can be handled like in Section 3.3.

The essence of our method is to translate the definition of the composite view p at hand into a basic view. There are two ways, which can be repeatedly interleaved, to progress towards this.

The first and obvious way is to perform an *unfolding* of the definition of p. Clearly unfolding alone cannot, in general, obtain a basic view, because of recursion.

The alternative way, which represents the main contribution of this section, is to replace the remaining non-base predicates in the definition by an *abstraction*, that is, a sequence of other predicates and constraints, in such a way the resulting definition of p is *at least as general* as the original definition. Though this step is seemingly difficult, it may be the case in applications that the abstraction is in fact evident from the domain. We shall demonstrate this below; meanwhile, we shall call this methodology an *application-based abstraction*.

For example, for a view $p(\tilde{X})$ whose recursive definition refers to a base predicate q, it is possible to unfold $p(\tilde{X})$ a number of times such that q is *exposed* along with some subgoals of p:

$$p(\tilde{X}) \texttt{:-} p(\tilde{X}'),\ q_1(\tilde{X}_1),\ p(\tilde{X}'),\ \Psi(\tilde{X}_1).$$

Now if one can replace the two $p(\tilde{X}')$'s with a constraint and combine it with $\Psi(\tilde{X}_1)$ to obtain:

$$p'(\tilde{X}) \texttt{:-} q_1(\tilde{X}_1),\ \Psi'(\tilde{X})$$

Then $p'(\tilde{X})$ is an abstraction of the recursive view $p(\tilde{X})$.

We now give a more concrete example of abstracting the view `deliverable` in the shipping example of Sec 2.2. We shall however simplify the relation `vessel/7` to three arguments: source, destination and cost. We further assume that the value of cost, for each (source,destination) pair, is precisely the distance between them according to `map/3`. We also simplify `deliverable/8` to three arguments: source, destination and budget. Fig. 4 shows the simplified definition of `deliverable`.

By inspecting the third rule of `deliverable`, one can see that `deliverable` is the transitive closure of `vessel` and thus the cost of a direct vessel from point A to C and from D to B is no bigger than cost of the transitive closure. In other words, given `map(A, C, Co1)` and `map(D, B, Co3)`, `Co1` $\leq$ `Cost1` and `Co3` $\leq$ `Cost3`, where `Cost1` and `Cost3` are defined in Fig. 4. We illustrate this scenario in Fig. 5, where the dark arrow refers to an actual vessel, the light arrows refer to straight distances (map), and the dashed arrows are the transitive closure of vessels (deliverable). Therefore we can abstract `deliverable` to a basic view `abs_deliverable` as follows.

```
deliverable(A, B, Budget):-
  vessel(A, B, Cost),
  0<Cost<=Budget.
deliverable(A, B, Budget):-
  vessel(A, C, Cost1),
  vessel(C, B, Cost2),
  0<Cost1+Cost2<=Budget.
deliverable(A, B, Budget):-
  deliverable(A, C, Cost1),
  vessel(C, D, Cost2),
  deliverable(D, B, Cost3),
  Cost1+Cost2+Cost3<=Budget.
```

Fig. 4. Simplified `deliverable`

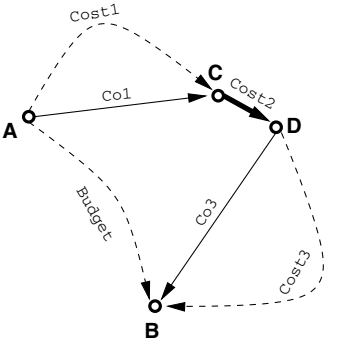


Fig. 5. Abstraction of `deliverable`

```
abs_deliverable(A, B, Budget):-
  Co1+Cost2+Co3<=Budget,
  map(A, C, Co1), vessel(C, D, Cost2), map(D, B, Co3).
```

The above abstraction captures the intuition that if a new vessel segment (from C to D) is too far away from both the source (A) and destination (B) of a reactor, then this reactor should be excluded from triggering. In other words, we are exploiting "locality" of the source and destination in this example.

We summarize our methodology of abstraction as follows.

- it reduces composite views to basic views so that direct indexing can be done on the basic conditions;
- it provides a *conservative* estimate of the original view, ie. the set of facts satisfying the abstraction is a *superset* of that of the original view. Thus an update that does not trigger an abstracted condition does not trigger the original blocking condition. In other words, exclusion of reactors by abstraction is *safe*.

It is, of course, undecidable in general to replace a composite view with an equivalent one, while trivial to replace it with an abstraction. The challenge is to find a *useful* abstraction. While it is not possible to characterize this condition formally, we suggest that if the application intuitively satisfies the condition that an individual reactor is not likely to be triggered by an average update, then it is likely that a desired abstraction can be discovered without great effort. We have tried to indicate this with the example above.

4 Implementation and Evaluation

We have implemented a prototype OCP system. Rather than using a tailor-made CLP system, our prototype for simplicity integrates the CLP($\mathcal{R}$) system [9] with a server registry explained in Section 2.3 that manages a collection of reactors especially to handle triggering and communicates with external agents. The multi-threaded registry was implemented in C++. The OCP registry and the CLP($\mathcal{R}$) system communicate through Unix message queues.

Agents are written using the language Python which is a relatively rich and extensible scripting language. A special Python reactor library handles the submission of reactors to the OCP system.

4.1 Trigger Efficiency

To evaluate the effectiveness of triggering using RC-tree, we conduct the following experiments on a Pentium 4 2.4GHz PC running Linux 2.4.20. We implement the shipping marketplace example in Section 2.2 for transporting cargo between a set of 7 Asian and 6 North American cities. The distance matrix among these cities is approximated by the flight distances between them [4].

We identify two types of reactor blocking conditions and two types of vessel updates: *intra-continental*, *inter-continental*. For instance, a reactor waiting to ship a cargo

Table 1. Hit rate and average trigger time

	reactors(intra)	reactors(inter)	reactors(mixed)
vessels(intra)	**10.176%** (12.6ms)	37.743% (13.2ms)	23.964% (12.9ms)
vessels(inter)	**0%** (8.7ms)	33.893% (13.1ms)	16.937% (12.7ms)
vessels(mixed)	**4.364%** (12.4ms)	33.694% (13.2ms)	19.025% (12.8ms)

from an Asian city to an American city is inter-continental, whereas a reactor waiting to ship within two Asian cities is an intra-continental reactor. Similar definition applies to the available vessels. We thus created three sets of reactors (1000 reactors in each set): intra-continental only, inter-continental only, and mixed; and similarly three sets of vessel updates (1000 updates in each set): intra-continental only, inter-continental only, and mixed. We use the abstraction in Section 3.4 for triggering the reactors. We did 9 experiments, corresponding to the 9 possible combinations of sets of reactors and updates. The experimental results are given in Table 1. The first number in each data cell is the average percentage of reactors being triggered out of 1000 blocked reactors in each scenario. The second number in parentheses is the time for the triggering mechanism to determine which reactors to be fired with a sequence of 1000 vessel updates. All times are measured in milliseconds.

From Table 1, we see that for intra-continental reactors, the best case for triggering excludes all reactors from wakeup (intra-column, row 2). This is intuitive because long-haul voyages are more expensive and take longer and thus do not affect short range shipping needs. For inter-continental reactors, triggering excludes about two thirds of the reactors. This is simply because for inter-continental reactors, the budgets are larger and deadlines are later, and thus short range vessels are more likely to affect these reactors. The experimental results demonstrate that indexing and abstraction are effective optimizations: (a) the triggering mechanism is effective in avoiding the wakeup of a substantial number of blocked reactors; and (b) the triggering mechanism is itself relatively fast.

5 Concluding Remarks

This paper has presented a new distributed programming framework which allows distributed program agents to react to a CLP program like a shared common store. Agents modify the CLP program through the use of reactors which are guarded on logic conditions with respect to the CLP. The key challenge is then how to efficiently manage these reactors to allow blocking and wake-up. We detailed a triggering framework which incorporates a novel spatial index structure to solve this problem.

Some of the future work includes the development of an automatic verification tool for application-based abstraction used in the triggering, and the classification of various kinds of advanced views as well as abstraction recommendations for these classes. We are also enhancing OCP with a more generalized version of committed choice in which commit can happen anywhere in the choice branch.

References

1. G. Agha and C. J. Callsen. Actorspace: an open distributed programming paradigm. In *ACM PPoPP*, 23–32. 1993.
2. G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.
3. P. Caspi, D. Pilaud, N. Halbwachs, and J. Plaice. Lustre: A declarative language for programming synchronous systems. In *ACM POPL*, 178–188. 1987.
4. Distances between 325 cities in the world. `http://www.etn.nl/distanc4.htm`.
5. V. Gaede. Multidimensional access methods. *ACM Computing Survey*, 30(2):170–231, 1998.
6. T. Gautier and P. L. Guernic. SIGNAL: A declarative language for synchronous programming of real-time systems. In *FPCA*, pages 257–277. Springer, 1987.
7. D. Gelernter. Generative communication in Linda. In *ACM TOPLAS*, 7(1):80–112, 1985.
8. J. Jaffar and J.-L. Lassez. Constraint logic programming. In *ACM POPL*, 111–119, 1987.
9. J. Jaffar, S. Michaylov, P. J. Stuckey, and R. H. C. Yap. The CLP(R) language and system. In *ACM TOPLAS*, 14(3):339–395, 1992.
10. H. P. Nii. *Blackboard Systems*. Addison Wesley, 1989.
11. H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, 1990.
12. V. A. Saraswat. *Concurrent Constraint Programming*. MIT Press, 1993.
13. G. Smolka. The Oz programming model. In *Computer Science Today*, pages 324–343. Springer, 1995.
14. K. Ueda. Guarded horn clauses. In *4th Logic Programming '85*, LNCS 221, 168–179, 1986.
15. J. Widom and S. Ceri. *Active Database Systems: Triggers and Rules for Advanced Database Processing*. Morgan Kaufmann Publishers, Inc., 1996.

Parallelizing Union-Find in Constraint Handling Rules Using Confluence Analysis

Thom Frühwirth

Faculty of Computer Science, University of Ulm, Germany
www.informatik.uni-ulm.de/pm/mitarbeiter/fruehwirth/

Abstract. Constraint Handling Rules is a logical concurrent committed-choice rule-based language. Recently it was shown that the classical union-find algorithm can be implemented in CHR with optimal time complexity. Here we investigate if a parallel implementation of this algorithm is also possible in CHR. The problem is hard for several reasons:

- Up to now, no parallel computation model for CHR was defined.
- Tarjan's optimal union-find is known to be hard to parallelize.
- The parallel code should be as close as possible to the sequential one.

It turns out that confluence analysis of the sequential implementation gives almost all the information needed to parallelize the union-find algorithm under a rather general parallel computation model for CHR.

1 Introduction

Constraint Handling Rules (CHR) [7] is a concurrent committed-choice constraint logic programming language consisting of guarded rules that transform multi-sets of constraints (atomic formulae) until a fixpoint is reached.

CHR was initially developed for writing constraint solvers, but is more and more used as a general-purpose programming language. Recent applications of CHR range from type systems and time tabling to ray tracing and cancer diagnosis [2,15]. In these applications, conjunctions of constraints are best regarded as interacting collections of concurrent agents or processes.

So far, there have been several operational semantics for CHR (standard, refined and compositional), but none of them explicitely addresses *parallelism*. We develop a general parallel execution model for CHR relying on a monotonicity property (applicable rules cannot become in-applicable during a computation). Analogous concurrency constructions were suggested for other (constraint) logic programming languages, e.g. [12].

The clean semantics of CHR facilitates non-trivial program analysis and transformation. In particular, confluence analysis is an issue in CHR, since rule application is committed-choice, it is never undone (unlike Prolog). *Confluence* aks the question if a program produces the same result no matter which of the applicable rules are applied in which order. It turns out that this property facilitates the parallel execution of a CHR program. Since there is a decidable, sufficient and necessary criterion for confluence [3] that returns the problematic

M. Gabbrielli and G. Gupta (Eds.): ICLP 2005, LNCS 3668, pp. 113–127, 2005.

cases of rules applications that rule out each other, we can use this analysis to construct a parallel program from a sequential one. Of course, some crucial insights in the nature of the algorithm to be implemented are still necessary. As a side-effect of our work, we implemented a practical confluence checker.

We will apply this methodology of *parallization to the classical union-find* (also: disjoint set union) algorithm of Tarjan [17]. This essential algorithm efficiently solves the problem of maintaining a collection of disjoint sets under the operation of union [9]. It is the basis for many graph algorithms and for dealing with equality, e.g. in unification algorithms. We have chosen union-find, because it was recently shown that it is possible to implement it with optimal time complexity in CHR [13,14,16], something that is not known to be possible in other pure logic programming languages. The bad news is that union-find is inherently sequential in most parts, and therefore hard to parallelize. Its worst case time performance can actually get worse upon parallelization if the sequential algorithm is used as a basis [4]. Often, other data structures and algorithms are used for the parallel union-find problem [10].

So this is our *challenge*: Can we come up with an optimal parallel union-find algorithm that is close to the sequential one? Can confluence analysis help? This paper gives a preliminary answer tending towards positive.

Outline of the Paper. In the next two sections, we introduce union-find algorithms and CHR. In Section 4 we introduce a parallel execution model for CHR. In the next section we give a sequential CHR program for the basic union-find algorithm. Section 6 uses confluence analysis to parallize this implementation. The next two sections carry this approach over to optimized union-find. At the end of Section 8, correctness of the parallel version is argued by simulating the sequential one and vice versa. Section 9 concludes with future work.

2 The Union-Find Algorithm

The union-find algorithm maintains disjoint sets under union. Each set is represented by a rooted tree, whose nodes are the elements of the set. The root is called the *representative* of the set. The representative may change when the tree is updated by a union operation. With the algorithm come three operations:

- `make(X)`: generate a new tree with the only node `X`, i.e. `X` is the root.
- `find(X)`: follow the path from the node `X` to the root of the tree by repeatedly going to the parent node of the current node until the root is reached. Return the root as representative.
- `union(X,Y)`: to join the two trees, find the representatives of `X` and `Y` (they are roots). Then `link` them by making one point to the other.

The basic algorithm requires $\mathcal{O}(N)$ time per find (and union) in the worst case, where N is the number of elements (make operations). With two independent optimizations that keep the tree shallow and balanced, one can achieve logarithmic worst-case and quasi-constant (i.e. almost constant) amortized running time per operation.

The first optimization is *path compression* for find. It moves nodes closer to the root after a find. After `find(X)` returned the root of the tree, we make every node on the path from `X` to the root point directly to the root.

The second optimization is *union-by-rank*. It keeps the tree shallow by pointing the root of the smaller tree to the root of the larger tree. *Rank* refers to an upper bound of the tree depth (tree height). If the two trees have the same rank, either direction of pointing is chosen but the rank is increased by one. With this optimization, the height of the tree can be bound by $log(N)$. Thus the worst case time complexity for a single find or union operation is $\mathcal{O}(log(N))$.

Parallelization can worsen the performance of optimized union-find, because the find operation is inherently sequential and parallel tree updates can counteract the effects of path compression and union-by-rank [4] so that deep trees (with long paths) are generated. In order to achieve logarithmic worst case time complexity per operation, one has to restrict the parallelism, use special auxiliary data and operations [4] or has to rely on different special-purpose data structures and algorithms altogether [10,5].

3 Constraint Handling Rules (CHR)

In this section we give an overview of syntax and semantics for Constraint Handling Rules (CHR) [7,8].

Syntax of CHR. We use two disjoint sets of predicate symbols for two different kinds of constraints: built-in (pre-defined) constraint symbols which are solved by a given constraint solver, and CHR (user-defined) constraint symbols which are defined by the rules in a CHR program. There are three kinds of rules:

$$\begin{aligned} &\textit{Simplification rule: } \textit{Name} \ @\ H \Leftrightarrow C \mid B, \\ &\textit{Propagation rule: } \ \textit{Name} \ @\ H \Rightarrow C \mid B, \\ &\textit{Simpagation rule: } \ \textit{Name} \ @\ H \setminus H' \Leftrightarrow C \mid B, \end{aligned}$$

where *Name* is an optional, unique identifier of a rule, the *head* H, H' is a non-empty comma-separated conjunction of CHR constraints, the *guard* C is a conjunction of built-in constraints, and the *body* B is a goal. A *goal (query, problem)* is a conjunction of built-in and CHR constraints. A trivial guard expression "`true |`" can be omitted from a rule.

Simpagation rules abbreviate simplification rules of the form

$$\textit{Name} \ @\ H \wedge H' \Leftrightarrow C \mid H \wedge B.$$

Standard Operational Semantics of CHR. The operational semantics of CHR is given by a transition system (Fig. 1) . States are goals, i.e. conjunctions of built-in and CHR constraints. In the figure, all upper case letters are meta-variables that stand for conjunctions of constraints. CT is the constraint theory for the built-in constraints. G_{bi} denotes the built-in constraints of G, which is the remainder of the state/goal.

Simplify

If $(H \Leftrightarrow C \mid B)$ is a fresh variant of a rule with variables $\bar{x}$
and $CT \models \forall\,(G_{bi} \rightarrow \exists \bar{x}(H{=}H' \wedge C))$
then $(H' \wedge G) \mapsto (B \wedge G \wedge H{=}H' \wedge C)$

Propagate

If $(H \Rightarrow C \mid B)$ is a fresh variant of a rule with variables $\bar{x}$
and $CT \models \forall\,(G_{bi} \rightarrow \exists \bar{x}(H{=}H' \wedge C))$
then $(H' \wedge G) \mapsto (H' \wedge B \wedge G \wedge H{=}H' \wedge C)$

Fig. 1. Computation Steps of Constraint Handling Rules

CHR rules are applied exhaustively, until a fixed-point is reached, to the initial state. A simplification rule $H \Leftrightarrow C \mid B$ *replaces* instances of the CHR constraints H by B provided the guard C holds. A propagation rule $H \Rightarrow C \mid B$ instead *adds* B to H. If new constraints arrive, rule applications are restarted.

A rule is *applicable*, if its head constraints are matched by constraints in the current goal one-by-one and if, under this matching, the guard of the rule is implied by the built-in constraints in the goal. Any of the applicable rules can be applied, and the application cannot be undone, it is committed-choice (in contrast to Prolog).

When a simplification rule is applied, the matched constraints in the current goal are replaced by the body of the rule. When a propagation rule is applied, the body of the rule is added to the goal without removing any constraints. When a simpagation rule is applied, all constraints to the right of the backslash are replaced by the body of the rule.

To avoid trivial non-termination, a CHR propagation rule is never applied a second time to the same constraints. A *final state* is one where either no computation step is possible anymore or where the built-in constraints are inconsistent.

The *refined operational semantics* [6] specializes the CHR standard semantics as given here to the one that is usally implemented: Similar to Prolog, constraints in a state are evaluated depth-first from left to right and rules are applied in textual program order. (The refined semantics thus rules out some computations that are possible in the standard semantics.)

4 Parallelism for CHR

Intuitively, we expect that in a parallel execution of a CHR program, rules can be applied to separate parts of the problem in parallel without interference. We will interpret conjunction as parallel operator and we will use an *interleaving semantics* for parallelism in CHR. It means that a parallel computation step can be performed by a sequence of sequential computation steps. A similar approach was taken for other concurrent constraint/logic languages, e.g. [11,12].

To avoid the technicalities of special cases, we relax the operational semantics of CHR with regard to final states. We allow a finite, bounded number of additional computation steps from inconsistent states (they will stay inconsistent).

We now define two notions of parallelism, weak and strong (Fig. 2). (In the figures, A, B, C, D and E are conjunctions of arbitrary constraints.) Such straightforward interleaving semantics for parallelism are not possible for imperative languages, where computations may give raise to conflicting (over-)writes. However, it is possible for many (constraint) logic programming languages, due to the *monotonicity (or stability) property* (left of Fig. 3): Adding constraints to a state cannot inhibit the applicability of a rule. Monotonicity of CHR was proven in [3]. The property also implies that constraints can be processed incrementally in CHR, giving rise to an online algorithm behavior.

We can now justify *weak parallelism* by a consequence of monotonicity which we call *trivial confluence* (right of Fig. 3), because independent of the intermediate state, we will arrive at the same successor state.

If $A \mapsto B$	If $A \wedge E \mapsto B \wedge E$
and $C \mapsto D$	and $C \wedge E \mapsto D \wedge E$
then $A \wedge C \mapsto B \wedge D$	then $A \wedge E \wedge C \mapsto B \wedge E \wedge D$

Fig. 2. Weak Parallelism of CHR Strong Parallelism of CHR

If $A \mapsto B$	If $A \mapsto B$
then $A \wedge C \mapsto B \wedge C$	and $C \mapsto D$
	then $A \wedge C \mapsto S \mapsto B \wedge D$
	(S is either $A \wedge D$ or $B \wedge C$)

Fig. 3. Monotonicity of CHR Trivial Confluence of CHR

The definition of *strong parallelism* (right of Fig. 2) shows that there is more potential for parallelism in CHR than working on separate parts of the problem. Constraints in E may be necessary for rule application, but since both rules do not alter these constraints, we can still apply them in parallel. With strong parallelism, rules may work on *common* constraints at the same time if they do not change them. (With weak parallelism, we would need two copies of the constraints E.) Clearly all built-in constraints are common. Propagation rules only add CHR constraints, so any CHR constraints they match can be common. Simpagation rules do not remove some of the constraints they match, so these can be common as well. We will assume strong parallelism in the rest of the paper.

We assume for now that rule applications (hence, computation steps) are *instantaneous*, i.e. the removal and addition of constraints caused by the application of a rule is an atomic action. With this requirement, a rule can still take arbitrary time to check its applicability to constraints in (a snapshot of) the current state. When a rule is to apply, it will first flag the constraints it matched. If some are not there anymore, the rule application simply is not done and flags are reset.

5 Implementing Basic Union-Find in CHR

The CHR program `ufd_basic` (in concrete ASCII syntax) implements the operations and data structures of the basic union-find algorithm optimizations as CHR constraints [16].

ufd_basic

```
make      @ make(A) <=> root(A).
union     @ union(A,B) <=> find(A,X), find(B,Y), link(X,Y).

findNode  @ A ~> B  \ find(A,X) <=> find(B,X).
findRoot  @ root(A) \ find(A,X) <=> X=A.

linkEq    @ link(A,A) <=> true.
link      @ link(A,B), root(A), root(B) <=> B ~> A, root(A).
```

The constraints `make/1`, `union/2`, `find/2` and `link/2` define the *operations*. The find operation is implemented as a relation `find/2` whose second argument returns the result. `link/2` is an auxiliary operation for performing union of two roots. The *tree (data) constraints* `root/1` and `~>/2` ("points to") represent the tree data structure. This program is operationally equivalent for allowed queries to implementations in imperative languages under the refined sequential CHR semantics [16]. An *allowed query* is - as usual for union-find - a sequence of `make`, `find` and `union` operations. The second argument of a `find` is a new variable. Nodes are typically constants. Each node is introduced by one `make`.

Now we discuss *parallel execution* of the above rules for allowed queries. We can accomodate different find operations on the same node, since the tree constraints are not altered by a find. The `link` rule replaces `root(B)` by `B~>A`. Since rule application is instantaneous and atomic in our model, there will always be a tree constraint for each node that has been introduced. So if one of the processes performing an operation fails, it can do no harm to the overall computation, since each rule defines exactly one operation. Actually, the operations make, union and find can always proceed at their own speed. The link operation obviously has to wait for the result of the find operations. Moreover, when we are about to apply the `link` rule, another link operation may remove one of the roots that we need for linking. The next section explains how we can detect and avoid such problematic situations using confluence analysis and additional rules.

6 Confluence for Parallelism

We already have used trivial confluence to justify our model of parallelism. But the relationship is deeper. The interleaving semantics of a parallel execution

can be given as the semantics of all its possible interleavings, i.e. sequential executions that lead to the same resulting state. For analysis, we then have to consider all possible sequential execution orders. These different orders of constraints in a goal may mean that different rules are applied. Confluence tells us that no matter which of the applicable rules we apply, we always can reach the same resulting state. In other words, a particular parallel execution cannot go "astray", resulting in a different state (that may well correspond to a *deadlock*). We will see that such deadlocks actually can occur in the basic union-find algorithm.

Confluence Analysis in CHR. Before we discuss confluence of the union-find algorithm, we introduce the basic idea behind confluence analysis. The papers [1,3] give a decidable, sufficient and necessary condition for confluence for terminating CHR programs.

For checking confluence, one takes two rules (not necessarily different) from the program. The heads of the rules are overlapped by equating at least one head constraint from one rule with one from the other rule. For each *overlap*, we consider the two states resulting from applying one or the other rule. These two states form a so-called *critical pair*. One tries to *join* the states in the critical pair by finding two computations starting from the states that reach a common state. If the critical pair is not joinable, we have found a counterexample for confluence of the program.

6.1 Confluence of Basic Sequential Union-Find for Parallelization

A detailed confluence analysis of the sequential union-find algorithms in CHR is in [13,14]. Union-find is not confluent under the standard sequential operational semantics. The relative order of find and union (link) operations matters for the outcome of find. This behavior is inherent in the union-find algorithm due to its update of the tree structure and the resulting changes of the representatives.

But non-confluence can also be caused by *incompatible tree constraints* such as `root(A), A~>B` (that can be shown not to occur when computing with allowed queries), and due to *competing link operations* for the same roots (that cannot occur in the left-to-right execution order of the refined semantics, but in parallel execution). A *deadlock* means that an operation cannot finish. In the last case, link operations deadlock, and the restoration of confluence by adding proper rules can avoid or break these deadlocks.

Since there is a combinatorial explosion in the number of critical pairs with program size, it is important to filter out "trivial" non-joinable critical pairs that either stem from overlaps that are not possible for allowed queries or that we would like to consider equivalent for our purposes. For union-find, the former means e.g. to detect incompatible tree constraints, the latter means to regard tree constraints that describe the same sets as equivalent. We revised the confluence checker of [13] so that it performs the necessary additional checks before and normalization after computation of a critical pair. These program-specific filters are encoded as Prolog or CHR rules, e.g. `check, root(A,_), A~>_ <=>`

`fail`. The confluence checker and its results for the union-find programs of this paper are available at `www.informatik.uni-ulm.de/pm/mitarbeiter/fruehwirth/union-find/`.

For the union-find implementation `ufd_basic`, there are 8 non-joinable critical pairs [13,14]. Three non-joinable critical pairs, between the pairs of rules `findNode-findNode`, `findNode-findRoot`, and `linkEq-link`, feature incompatible tree constraints. We avoid the remaining non-trivial critical pairs by modifying the given program.

The critical pair between `find` and `link` reveals that the *relative order of find and link* operations matters for the outcome of the find.

```
            find(B,A),root(B),root(C),link(C,B)
      findRoot                              link
root(C),B~>C,A=B                    root(C),B~>C,A=C
```

The first line gives an overlap of the two rules mentioned in the second line. The third line gives the critical pair, i.e. the two final states reachable when the corresponding rule from the second line is applied to the overlap.

The last four non-joinable critical pairs come from overlapping the `link` with itself. They feature *pending competing links*. Two link operations have at least one tree node in common. So when one link is performed, at least one node in the other link operation is not a root anymore, and so this link operation will *deadlock*, for example:

```
        root(A),root(B),link(B,A),link(A,B)
      link                            link
root(B),A~>B,link(A,B)      root(A),link(B,A),B~>A
```

Insight #1. To handle these non-confluences, we first concentrate on the critical pair between `findRoot` and `link`. We replace the culprit built-in equality constraint `=/2` by our own new CHR constraint `found/2`, that we can tailor to our needs. In the `findRoot` rule, `X=A` becomes `found(A,X)`. It holds the result of the find operation in the first argument. Now we can add a rule for `found` (given below) that joins the corresponding critical pair. (The rule mimics `findNode` so that the `found` constraint keeps track of the updates of the tree.)

```
            find(B,A),root(B),root(C),link(C,B)
      findRoot                              link
                  root(C),B~>C,found(C,A)
```

The `link` rules are modified by replacing instances of `link(A,B)` in the head of a rule by the proper instances of `link(X,Y), found(A,X), found(B,Y)`. The resulting program is `ufd_basic1`. Also the critical pairs of the `link` rules can be joined now, because `found` can update itself so that its result argument is a root.

ufd_basic1

```
findNode   @ A ~> B  \ find(A,X) <=> find(B,X).
findRoot1  @ root(A) \ find(A,X) <=> found(A,X).

found      @ A ~> B \ found(A,X) <=> found(B,X).

linkEq1    @ link(X,Y), found(A,X), found(A,Y) <=> true.
link1      @ link(X,Y), found(A,X), found(B,Y),
                        root(A), root(B) <=> B ~> A, root(A).
```

Confluence of Basic Parallel Union-Find. The confluence analysis lead us to a revised, parallel version `ufd_basic1` of the basic algorithm. There are now many more non-joinable critical pairs, because the introduction of the `found` constraints gives rise to many possible overlaps for the `link` rules. All the critical pairs are trivial, because they either cannot occur for allowed queries or they feature different tree constraints that represent the same set of nodes.

findNode	findNode	1
findNode	findRoot	1
linkEq	linkEq	6
link	linkEq	13
link	link	65
found	link	2
found	found	1

Number of crit. pairs between pairs of rules

7 Optimized Union-Find

The CHR program *ufd_rank* from [16] implements the optimized classical union-find algorithm, derived from the basic version by adding path compression for find and union-by-rank [17].

ufd_rank

```
make      @ make(A) <=> root(A,0).
union     @ union(A,B) <=> find(A,X), find(B,Y), link(X,Y).

findNode  @ A ~> B, find(A,X) <=> find(B,X), A ~> X.
findRoot  @ root(A,_) \ find(A,X) <=> X=A.

linkEq    @ link(A,A) <=> true.
linkLeft  @ link(A,B), root(A,N), root(B,M) <=> N>=M |
               B ~> A, N1 is max(N,M+1), root(A,N1).
linkRight @ link(B,A), root(A,N), root(B,M) <=> N>=M |
               B ~> A, N1 is max(N,M+1), root(A,N1).
```

When compared to the basic version `ufd_basic`, we see that `root` has been extended with a second argument that holds the rank of the root node. (The first two rules, `make` and `union`, will stay the same for all remaining programs in this paper, they are therefore omitted from now on.)

The rule `findNode` has been extended for immediate path compression: the logical variable `X` serves as a place holder for the result of the find operation. The `link` rule has been split into two rules `linkLeft` and `linkRight` to reflect the optimization of union-by-rank: The smaller ranked tree is added to the larger ranked tree without changing its rank. When the ranks are the same, either tree is chosen (both rules are applicable) and the rank is incremented.

Confluence. The non-joinable critical pairs are in principle analogous to the ones discussed for `ufd_basic` in Section 6, but their numbers significantly increase due to the optimizations of path compression and union-by-rank that complicate the rules for the find and link operations. The confluence checker found 73 non-joinable critical pairs [13,14]. The number of critical pairs is dominated by those 68 of the link rules.

Unlike the basic versions, in the optimized algorithm, two `findNode` rule applications on the same node will interact, because one will compress, and then the other cannot proceed until the first find operation has finished:

```
                find(B,A),B~>C,find(B,D)
        findNode                  findNode
find(A,D),find(C,A),B~>D    find(D,A),find(C,D),B~>A
```

The critical pairs for the find rules tell us that parallel finds have to wait for the result of path compression from one of the finds. In the worst case, if that find process fails, other finds will deadlock (which was not the case in the basic version of the algorithm). As a remedy we introduce an explicit compression operation that runs in parallel to the other operations.

8 Optimal Union-Find Parallelized

We first introduce `found` into the program of optimal union-find as for the basic algorithm.

Insight #2. We make compression explicit by a new operation `compr/2`. We modify the `findNode` rule to call `compr(A,X)` instead of immediately producing a tree data constraint that points to a yet free variable. As a consequence, the rule for explicit compression should take the found root node and the corresponding tree constraint to be compressed and replace it by the compressed tree constraint that points to the root. We should not forget to add compression also for `found`. The result are the following tentative rules.

```
findNode1?@ A ~> B \ find(A,X) <=> find(B,X), compr(A,X).
found1?   @ A ~> B \ found(A,X) <=> found(B,X), compr(A,X).
compress? @ root(C,_),found(C,X) \ A ~> B,compr(A,X) <=>A ~> C.
```

Compression is performed in parallel to the main part of the algorithm that performs the find and link operations. But since both linking and compression update the tree data structure, we may expect interferences. These are revealed by our confluence analysis. First of all, linking takes away `found`, so compression deadlocks after linking. We may compress to a root different from what has been used for linking. We may compress too early and thus too little: Consider parallel unions on different new nodes: all find and compress operations can immediately finish, because they are on roots. No compression is performed. But linking is sequentialized because roots change. `found` constraints handle these changes, but again no compression is performed.

ufd_foundc_compr

```
findNode1 @ A ~> B \ find(A,X) <=> find(B,X), compr(A,X).
findRoot1 @ root(A,_) \ find(A,X) <=> found(A,X).

found1    @ A ~> B \ found(A,X) <=> found(B,X), compr(A,X).

compress  @ foundc(C,X) \ A ~> B, compr(A,X) <=> A ~> C.

linkEq1c    @ found(A,X), found(A,Y), link(X,Y) <=>
                        foundc(A,X), foundc(A,Y).
linkLeft1c  @ found(A,X), found(B,Y), link(X,Y),
                        root(A,N), root(B,M) <=> N>=M |
                        foundc(A,X), foundc(B,Y),
                        B ~> A, N1 is max(N,M+1), root(A,N1).
linkRight1c @ found(A,X), found(B,Y), link(Y,X),
                        root(A,N), root(B,M) <=> N>=M |
                        foundc(A,X), foundc(B,Y),
                        B ~> A, N1 is max(N,M+1), root(A,N1).
```

The real problem is the *interference* between different compressions along the same sub-path, because roots change and because compression destroys paths. An old compress performed after a new one on the same node may "undo" the new compression, the overall result may be worse than without compression. Competing compressions may destroy the tree, even lead to cycles in the tree. These problems are well-known in the literature, different solutions have been proposed like comparing certain counters for nodes or time-stamps, or using a different compression technique like path halving [4]. We prefer a solution that does not introduce additional auxiliary operations or data and that is as close as possible to the original sequential optimal code. We do not want to be "too clever"[1]. This will also make it easier to verify the correctness of the implementation.

[1] Actually, we tried, investigated many variants, but confluence analysis usually revealed one of the problems mentioned here.

Insight #3. Our solution is to compress the nodes of a path to the root that was used for linking. So compression is performed *after* the corresponding linking operation. We do not think that this sequentiality is a disadvantage given the fact that before linking the roots of the tree may frequently change due to other link operations.

In the program `ufd_foundc_compr`, the rule `compress` now uses its own `found` named `foundc`. The program will leave the `foundc` constraints in the store. They represent the result of a find computation. If necessary, the `foundc` constraints can be *garbage collected*, when their second variable does not occur in any other constraint. Alternatively, their removal can be accommodated by keeping a counter on how often `foundc` will be used in path compression. When the counter is zero, the `foundc` constraint is removed.

8.1 Confluence of Parallelized Optimal Union-Find

The confluence analysis of this program finds several hundred non-joinable critical pairs, of the same nature as for the parallel basic version. The table shows that almost all critical pairs are between the rules for the link operations. 35 of them are joinable modulo equivalence of the nodes in the trees that are produced. All but one of the remaining critical pairs can be shown not to occur for allowed queries.

findNode1	findNode1	1
findNode1	findRoot1	1
findRoot1	findRoot1	1
found1	found1	1
found1	linkEq1c	2
found1	linkLeft1c	2
found1	linkRight1c	2
compress	findNode1	1
compress	found1	1
compress	compress	3

linkEq1c	linkEq1c	18
linkEq1c	linkLeft1c	39
linkEq1c	linkRight1c	39
linkLeft1c	linkLeft1c	191
linkLeft1c	linkRight1c	193
linkRight1c	linkRight1c	191

In the only non-trivial critical pair of the `compress` rule with itself, competing compressions may produce different trees, but the nodes are the same as the nodes `A, B, C` and `D` must be on the same path. In particular, trees are not destroyed and compression always improves the tree.

```
        foundc(C,X),compr(A,X), A~>B, foundc(D,Y),compr(A,Y)
    compress                                          compress
foundc(C,X),foundc(D,Y),A~>D          foundc(C,X),foundc(D,Y),A~>C
```

8.2 Correctness

We show that the new parallel CHR implementation and the optimal sequential implementation which was proven correct [16] *simulate each other* - to some

extent - by mapping computations between the two. We use the *refined operational semantics* of CHR for the sequential computations as in [13,14,16][2] and our proposed parallel semantics for CHR for the parallel computations.

We map states (constraints) and computation steps (rule applications). The *mapping* is inspired by the program transformation that we have performed to arrive at the parallel program: first, introduction of `found` that behaves like `find` until it is involved in a link (then it behaves like built-in equality), and second, replacement of implicit immediate path compression by an explicit one with `compr` that relies on `foundc` (which behaves like built-in equality) that is produced by linking.

In principle, the *rule applications* of parallel `ufd_foundc_compr`, i.e. `make,union,findNode1,findRoot1,linkEq1c,linkLeft1c,linkRight1c`, are mapped into the rule applications of sequential optimal `ufd_rank`, i.e. `make,union,findNode, findRoot, linkEq, linkLeft, linkRight`, and vice versa. (The rules `make` and `union` are identical in both programs, there is no need to further discuss them.)

Sequential to Parallel. The mapping from a sequential to a parallel execution is as follows. Under the refined semantics, the sequential program for each union will do the two find operations and then the linking. A find operation is a sequence of `findNode` rule applications followed by a single application of `findRoot`. In the mapping, immediate path compression by rule `findNode` is replaced by explicit path compression with `compr` constraints. The built-in equality constraints produced by `findRoot` in the sequential computation are replaced by `found` constraints until they are involved in a link operation. From then on, the equalities are replaced by `foundc`. Immediately after linking, we have to insert applications of the `compress` rule into the resulting parallel computation, so that compression is actually performed (removing all `compr` constraints).

Parallel to Sequential. Due to the interleaving semantics we have introduced for parallel CHR, any parallel computation can be described by a set of sequential computations involving the same rules and same result. Given such a sequentialized parallel computation, the following partial mapping will give us a computation of the sequential program.

By intended construction, not every execution of the parallel program can be mapped into one of the sequential program. Consider the critical pair for competing compressions (Subsect. 8.1): Since compression is immediate and implicit, only one of the computations can be simulated by the sequential program.

If we rule out these competing compressions (i.e. find or found constraints operating on the same nodes concurrently), parallel executions can be simulated by the sequential program: We map constraints `A~>B,compr(A,X)` into `A~>X` to achieve immediate compression. As a consequence, the `compress` rule applications become obsolete, because they do not change any constraints under the mapping.

[2] The correctness and optimality of the code was proven under the refined semantics.

We also map `found` into `find` constraints and thus applications of the rule `found1` into applications of `findNode`. Under the mapping, applications of the rule `findRoot1` do not change constraints, they are therefore removed. Just before a link rule is applied, we insert two rule applications of the rule `findRoot` that apply to the two involved `find` constraints that come from mapping `found`. Finally, we map `foundc` constraints into built-in equalities. The result of the transformation is a correct computation of the sequential implementation of the optimal union-find algorithm in the standard semantics. Hence we claim that our parallel program for union-find is correct for computations without competing compressions.

9 Conclusion

In this exploratory paper, we introduced a parallel execution model for CHR. We parallelized basic and optimal sequential versions of the classical union-find algorithm with the help of confluence analysis and three insights. The resulting code is close to the original one and promises to be as efficient, even though it is acknowledged in the literature that this is hard to achieve due to the inherent sequential nature of the algorithm when it comes to tree updates.

The URL `www.informatik.uni-ulm.de/pm/mitarbeiter/fruehwirth/union-find` contains a list of programs, confluence checkers and results of confluence analysis used for this paper.

It was beyond the scope of the paper to give a time complexity analysis, but let us speculate shortly on the topic. We showed that each rule application in one program corresponds to a rule application in the other program, with exception of the `compress` rule applications that only occur in the parallel program. But their number is bounded by the number of `findNode` rule applications. Hence if rule applications cost the same in sequential and parallel CHR, the optimal worst-case time complexity is preserved. Since find operations can run in parallel (but not linking), we can expect a reduction in latency for simultaneous queries and updates.

The preliminary, exemplary findings in this paper can just be the starting point for a number of challenging research topics:

- parallel union-find correctness and time complexity analysis,
- parallel model for CHR, its implementation and empirical evaluation,
- more practical confluence analysis, including automatic detection of critical pairs that cannot occur for allowed queries,
- development of a confluence-based parallelization methodology and its application to other CHR programs, in particular constraint solvers.

As pointed out by a reviewer, our confluence-based parallelization could also be used to convert a program using the refined semantics to a program using the standard semantics (where parallelization is straightforward).

Acknowledgements. We would like to thank Marc Meister and Tom Schrijvers for helpful discussions. We also thank the referees that provided us with detailed comments.

References

1. S. Abdennadher. Operational Semantics and Confluence of Constraint Propagation Rules. In *Third International Conference on Principles and Practice of Constraint Programming, CP97*, LNCS 1330. Springer, 1997.
2. S. Abdennadher, T. Frühwirth, and C. H. (Eds.). *Special Issue on Constraint Handling Rules, Journal of Theory and Practice of Logic Programming (TPLP).* Cambridge University Press, to appear 2005.
3. S. Abdennadher, T. Frühwirth, and H. Meuss. Confluence and Semantics of Constraint Simplification Rules. *Constraints Journal*, 4(2), 1999.
4. R. J. Anderson and H. Woll. Wait-free parallel algorithms for the union-find problem. In *STOC '91: Proceedings of the twenty-third annual ACM symposium on Theory of computing*, pages 370–380. ACM Press, Revision of November 1994.
5. M. J. Atallah, M. T. Goodrich, and S. R. Kosaraju. Parallel algorithms for evaluating sequences of set-manipulation operations. *J. ACM*, 41(6):1049–1088, 1994.
6. G. J. Duck, P. J. Stuckey, M. G. de la Banda, and C. Holzbaur. The Refined Operational Semantics of Constraint Handling Rules. In B. Demoen and V. Lifschitz, editors, *Proceedings of the 20th International Conference on Logic Programming*, 2004.
7. T. Frühwirth. Theory and Practice of Constraint Handling Rules, Special Issue on Constraint Logic Programming. *Journal of Logic Programming*, pages 95–138, October 1998.
8. T. Frühwirth and S. Abdennadher. *Essentials of Constraint Programming.* Springer, 2003.
9. Z. Galil and G. F. Italiano. Data Structures and Algorithms for Disjoint Set Union Problems. *ACM Comp. Surveys*, 23(3):319ff, 1991.
10. M. C. Pinotti, V. A. Crupi, and S. K. Das. A parallel solution to the extended set union problem with unlimited backtracking. In *IPPS '96: Proceedings of the 10th International Parallel Processing Symposium*, pages 182–186, Washington, DC, USA, 1996. IEEE Computer Society.
11. V. A. Saraswat and M. Rinard. Concurrent constraint programming. In *POPL '90: Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 232–245, New York, NY, USA, 1990. ACM Press.
12. V. A. Saraswat, M. Rinard, and P. Panangaden. The semantic foundations of concurrent constraint programming. In *POPL '91: Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 333–352, New York, NY, USA, 1991. ACM Press.
13. T. Schrijvers and T. Frühwirth. Union-Find in CHR. Technical Report CW389, Department of Computer Science, K.U.Leuven, Belgium, July 2004.
14. T. Schrijvers and T. Frühwirth. Analysing the CHR Implementation of Union-Find. In *19th Workshop on (Constraint) Logic Programming (W(C)LP 2005).* Ulmer Informatik-Berichte 2005-01, University of Ulm, Germany, February 2005.
15. T. Schrijvers and T. Frühwirth. CHR Website, `www.cs.kuleuven.ac.be/~dtai/projects/CHR/`, May 2005.
16. T. Schrijvers and T. Frühwirth. Optimal Union-Find in Constraint Handling Rules, Programming Pearl. *Journal of Theory and Practice of Logic Programming (TPLP)*, to appear.
17. R. E. Tarjan and J. van Leeuwen. Worst-case Analysis of Set Union Algorithms. *J. ACM*, 31(2):245–281, 1984.

An Optimised Semantic Web Query Language Implementation in Prolog

Jan Wielemaker

Human Computer Studies (HCS), University of Amsterdam,
Kruislaan 419, 1098 VA Amsterdam, The Netherlands
wielemak@science.uva.nl

Abstract. The Semantic Web is a rapidly growing research area aiming at the exchange of *semantic* information over the World Wide Web. The Semantic Web is built on top of RDF, an XML-based *exchange* language representing a triple-based data model. Higher languages such as the description logic based OWL language family are defined on top of RDF. Making inferences over triple collections is a promising application area for Prolog.

In this article we study query translation and optimization in the context of the SeRQL RDF query language. Queries are translated to Prolog goals, which are optimised by reordering *literals*. We study the domain specific issues of this general problem. Conjunctions are often large, but the danger of poor performance of the optimiser can be avoided by exploiting the nature of the triple store. We discuss the optimisation algorithms as well as the information required from the low level storage engine.

1 Introduction

The Semantic Web [1] initiative provides a common focus for Ontology Engineering and Artificial Intelligence based on a simple uniform triple based data model. Prolog is an obvious candidate language for managing graphs of triples.

Semantic Web languages, such as RDF [2] RDFS and OWL, [4] define which new triples can be deduced from the current triple set (i.e. are *entailed* by the triples under the language). In this paper we study our implementation of the SeRQL [3] query language in Prolog. SeRQL provides a declarative search specification for a sub-graph in the deductive closure under a specified Semantic Web language of an RDF triple set. The specification can be augmented with conditions to match literal text, do numerical comparison, etc.

The original implementation of the SeRQL language is provided by Sesame [3], a Java based client/server system. Sesame realises entailment reasoning by computing the complete deductive closure under the currently activated Semantic Web language and storing this either in memory or in an external database. I.e. Sesame uses pure *forward reasoning*.

We identified several problems using the Sesame implementation. Sesame stores both the explicitely provided triples and the triples that can be derived from them given de semantics of a specified Semantic Web language (e.g. 'RDFS')

M. Gabbrielli and G. Gupta (Eds.): ICLP 2005, LNCS 3668, pp. 128–142, 2005.

in one database. This implies that changing the language to (for example) 'OWL-DL' requires deleting the derived triples and computing the deductive closure for the new language. Also, where the full deductive closure for RDFS is still fairly small, it explodes for more expressive languages like OWL. Sesame is sensitive to the order in which path expressions are formulated in the query, which is considered undesirable for a declarative query language. Finally, Sesame is written in Java an we feel much more comfortable using Prolog for manipulating RDF graphs to prototype new inferencing strategies.

To overcome the above mentioned problems we realised a server hosting multiple reasoning engines realised as Prolog modules. Queries can be formulated in the SeRQL language and both queries and results are exchanged through the language independent Sesame HTTP based client/server protocol. We extend the basic storage and query system described in [18] with SeRQL over HTTP and a query optimiser.

Naive translation of a SeRQL query to a Prolog program is straightforward. Being a declarative query language however, authors of SeRQL queries should and do not pay attention to efficient ordering of the path expressions in the query and therefore naive translations often produces inefficient programs. This problem as well as our solution is very similar to what is described by Struyf and Blockeel in [16] for Prolog programs generated by an ILP [11] system. We compare our work in detail with Struyf in Sect. 11.

In Sect. 2 and Sect. 3 we describe the already available software components and introduce RDF. Section 4 to Sect. 9 discuss native translation of SeRQL to Prolog and optimizing the naive translation through reordering of literals.

2 Available Components and Targets

Sesame[1] and its query language SeRQL is one of the leading implementations of semantic web RDF storage and query systems [9]. Sesame consists of two Java based components. The *server* is a Java *servlet* providing HTTP access to manage the RDF store and run queries on it. The *client* provides a Java API to the HTTP server.

The SWI-Prolog[2] SemWeb package [18] is a library for loading and saving triples using the W3C RDF/XML standard format and making them available for querying through the Prolog predicate **rdf/3**. After several cycles we realised the memory-based triple-store as a foreign language extension to SWI-Prolog. Using foreign language (C) we optimised the data representation and indexing for RDF triples, dealing with upto 40 million triples on 32-bit hardware or virtually unlimited on 64-bit hardware. The SWI-Prolog HTTP client/server packagehttp://www.swi-prolog.org/packages/http.html provides a multi-threaded [17] HTTP server and client library.

By reimplementing the Sesame architecture in Prolog we make our high performance triple-store available to the Java world. The options are illustrated

[1] http://www.openrdf.org
[2] http://www.swi-prolog.org

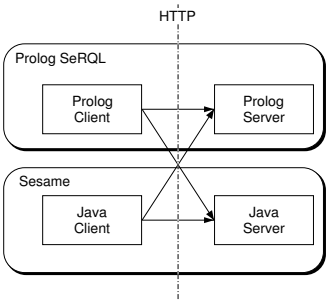


Fig. 1. With two client/server systems sharing the same HTTP API we have created four options for cooperation

in Fig. 1. In our project we needed access from Java applications to the Prolog server. Other people are interested in fetching sub-graphs from huge Sesame hosted triple sets stored in an external database to Prolog for further processing.

3 RDF Graphs and SeRQL Queries Graphs

In this section we briefly introduce RDF graphs and SeRQL queries. The RDF data model is a set of triples of the format <*Subject Predicate Object*>. The model knows about two data types:[3] *resources* and *literals*. Resources are *Universal Resource Identifiers* (URI), in our toolkit represented by Prolog atoms. Representing resources using atoms exploits the common representation of atoms in Prolog implementations as a unique handle to a string. This representation avoids duplication of the string and allows for efficient equality testing, the only operation defined on resources. Literals are represented by the term **literal**(*atom*), where *atom* represents the textual literal.

A triple informally states *Subject* has an attribute named *Predicate* with value *Object*. Both *Subject* and *Predicate* are resources, *Object* is either a resource or a literal. As a resource appearing as *Object* can also appear as *Subject* or even *Predicate*, a set of triples form a *graph*. A simple RDF graph is shown in Fig. 2.

RDF triples are naturally expressed using the predicate **rdf/3** with the obvious arguments **rdf**(*Subject, Predicate, Object*). Finding a subgraph with certain properties is now easily expressed as a Prolog conjunction, for example

```
reports_by_person(Report, Name) :-
     rdf(Author, 'http://www.example.org/terms/fullName', literal(Name)),
     rdf(Report, 'http://www.example.org/terms/author', Author).
```

SeRQL is a language with a syntax inspired in SQL, useful to represent target subgraphs as a set of edges, possibly augmented with conditions. An example is given in Fig. 3.

[3] Actually literals can be typed using a subset of the XML Schema primitive type hierarchy.

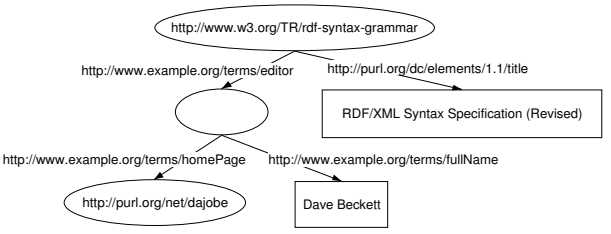


Fig. 2. A simple RDF graph. Ellipses are resources. Rectangles are literal values. Arrows point from *Subject* to *Object* and are labeled with the *Predicate*.

4 Compiling SeRQL Queries

The SWI-Prolog SeRQL implementation translates a SeRQL query into a Prolog goal, where edges on the target subgraph are represented as calls to **rdf**(*Subject, Predicate, Object*) and the WHERE clause is represented using natural Prolog conjunction and disjunction of predicates provided in the SeRQL runtime support module. The compiler is realised by a DCG parser, followed by a second pass resolving SeRQL namespace declarations and introducing variables. We illustrate this translation using an example from the SeRQL examples.[4] First we present the query in Fig. 3.

```
SELECT Painter, FName
FROM {Painter} <rdf:type>        {<cult:Painter>};
               <cult:first_name> {FName}
WHERE FName like "P*"
USING NAMESPACE
        cult = <!http://www.icom.com/schema.rdf#>
```

Fig. 3. Example SeRQL query asking for all resources of type `cult:Painter` whose name starts with `P`

Below is the naive translation represented as a Prolog clause and modified for better readability using the variable names from the SeRQL query. To solve the query, this clause is executed in the context of an *entailment module* as illustrated in Fig. 4. An entailment module is a Prolog module providing a pure implementation of the predicate **rdf/3** that can generate as well as test all triples that can be derived from the actual triple store using the Semantic Web language the module defines. This implies the predicate can be called with any instantiation pattern, will bind all arguments and produce all alternatives that follow from the entailment rules on backtracking. If **rdf/3** satisfies these criteria, any naive translation of the SeRQL query is a valid Prolog program to solve the query. Primitive conditions from the WHERE clause are mapped to predicates

[4] http://www.openrdf.org/sesame/serql/serql-examples.html

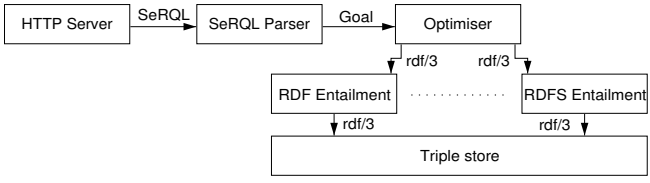


Fig. 4. Architecture, illustrating the role of *entailment modules*. These modules provide a pure implementation of **rdf/3** for the given Semantic Web language.

defined in the SeRQL runtime module which is imported into the entailment module. As the translation of the WHERE clause always follows the translation of the path expression all variables have been instantiated.

```
q(row(Painter, FName)) :-
        rdf(Painter,
            'http://www.w3.org/1999/02/22-rdf-syntax-ns#type',
            'http://www.icom.com/schema.rdf#Painter'),
        rdf(Painter,
            'http://www.icom.com/schema.rdf#first_name',
            FName),
        serql_compare(like, FName, 'P*').
```

SeRQL path expressions between square brackets ([...]) are *optional*. They bind variables if they can be matched, but they do not change the core of the matched graph. Such path expressions are translated using the SWI-Prolog *soft-cut* control structure represented by `*->`,[5] for example, the SeRQL statement

```
SELECT Artist, FName
FROM   {Artist} <rdf:type>        {<cult:Artist>};
               [<cult:first_name> {FName}]
USING NAMESPACE
        cult = <!http://www.icom.com/schema.rdf#>
```

is translated into the code below. Note that this prolog code generates all available first names, leaving *FName* unbound if no first name can be found. The final `true` is the translation of the omitted WHERE clause.

```
q(row(Artist, FName)) :-
       rdf(Artist,
         'http://www.w3.org/1999/02/22-rdf-syntax-ns#type',
         'http://www.icom.com/schema.rdf#Artist'),
       ( rdf(Artist, 'http://www.icom.com/schema.rdf#first_name', FName)
       *-> true
       ;   true
       ),
       true.
```

[5] Some Prolog dialects (e.g. SICStus) call this construct **if/3**.

5 The Ordering Problem

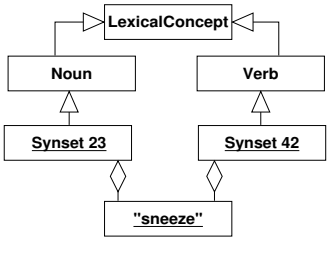
Given the purely logical definition of **rdf/3**, conjunctions of these goals can be placed in any order without influencing the result-set. Literals resulting from the WHERE clause are side-effect free boolean tests that can be executed as soon as the arguments have been instantiated. Note that Gooley [8] distinguishes 4 types of equivalence under optimization: *reflexive*, *set*, *tree* and *inequivalence*. We demand *set* equivalence, returning the same set of results where we do not care about ordering or duplicates.

To study the ordering problem in more detail we will consider the following example query on WordNet [10]. The query looks for words that can be interpreted in at least two different lexical categories.

```
SELECT DISTINCT L
   FROM {S1} <wns:wordForm> {L},
        {S2} <wns:wordForm> {L},
        {S1} <rdf:type> {C1},
        {S2} <rdf:type> {C2},
        {C1} <serql:directSubClassOf> {<wns:LexicalConcept>},
        {C2} <serql:directSubClassOf> {<wns:LexicalConcept>}
   WHERE not C1 = C2
   USING NAMESPACE
         wns = <!http://www.cogsci.princeton.edu/~wn/schema/>
```

WordNet is organised in *synsets*, an abstract entity roughly described by the associated *wordForms*. Synsets are RDFS instances of one of the subclasses of LexicalConcept. We are looking for a wordForm belonging to two synsets of a different subtype of LexicalConcept. Figure 5 illustrates a query result and gives some relevant metrics on WordNet.

To illustrate the need for optimisation as well as to provide material for further discussion we give two translations of this query. Figure 6 shows the direct translation, which requires 3.58 seconds CPU time on an AMD 1600+ processor as well as an alternative which requires 8,305 CPU seconds to execute, a slowdown of 2,320 times. Note that this translation could be the direct translation of another SeRQL query with the same semantics.



WordNet metrics	
Distinct wordForms	123,497
Distinct synsets	99,642
wordForm triples	174,002
Subclasses of LexicalConcept	4

Fig. 5. According to WordNet, the word "sneeze" can be interpreted as a *noun* as well as a *verb*. The tabel to the right gives some metrics of WordNet.

```
s1(L) :-
        rdf(S1, wns:wordForm, L), rdf(S2, wns:wordForm, L),
        rdf(S1, rdf:type, C1), rdf(S2, rdf:type, C2),
        rdf(C1, rdfs:subClassOf, wns:'LexicalConcept'),
        rdf(C2, rdfs:subClassOf, wns:'LexicalConcept'),
        C1 \== C2.

s2(L) :-
        rdf(C1, rdfs:subClassOf, wns:'LexicalConcept'),
        rdf(C2, rdfs:subClassOf, wns:'LexicalConcept'),
        C1 \== C2,
        rdf(S1, rdf:type, C1), rdf(S2, rdf:type, C2),
        rdf(S1, wns:wordForm, L), rdf(S2, wns:wordForm, L).
```

Fig. 6. Two translations for our query on WordNet. The first executes in 3.58 seconds, the second in 8,305.

Before we start discussing the alternatives for optimising the execution we explain *why* the execution times of these equivalent programs differs. Suppose we have a conjunction of completely independent literals A, B, C, where independent means no variables are shared between the members of the conjunction. If $b()$ denotes the number of solutions for a literal, the total solution space is $b(A) \times b(B) \times b(C)$ and therefore independent from the order. If we take the number of visited states rather than the solution space as a measure the formula becomes

$$b(A) + b(A) \times b(B) + b(A) \times b(B) \times b(C)$$

This measure is proportional to the number of *logical inferences* executed by Prolog and a good measure for the expected execution time [6]. It suggests to place literals with the smallest number of alternatives first, but as the last component is normally dominant the difference is not large and certainly cannot explain the difference between the two translations shown in Fig. 6. In fact the second is ordered on the branching factor *without considering dependencies*.

To understand this difference we must look at the dependencies, expressed by shared variables. Executing an **rdf/3** literal causes all its arguments to be grounded, restricting the number of alternatives for **rdf/3** literals sharing the grounded variables. What is really important is how much the set of alternatives of a literal is reduced by executing another literal before it. The order of **s1/1** in Fig. 6 executes the most unbound literal first (174,002 solutions), but wins because after the execution of this literal not much further branching is left.

6 Estimating the Complexity

The first step towards optimising is having an estimate of the complexity of a particular translation. We use the number of visited nodes in the search-tree as an estimate for the execution time, ignoring the (small) differences in time

required to execute the different **rdf/3** literals. Our estimate is based on two pieces of information extracted from the low-level database we have realised in the C language.

Estimated number of solutions for an rdf/3 call. For each **rdf/3** goal for which zero or more of the arguments have a known value and the remaining arguments are known to be unbound we can easily estimate the complexity. If no arguments are known this estimate is the total number of triples in the database, a number that is easily incrementally maintained by the database manipulation routines. If all arguments are known the literal is a boolean test, whose solution set we estimate as 0.5 (see *Boolean tests* below). In all other cases we compute the indexing and return the length of the hash-chain for the computed index. Assuming a well distributed hash-function this is a reasonable estimate for the number of solutions the goal will provide, while the information can be maintained incrementally by the database primitives.

Estimating the branching factor of predicates. Execution of literals binds variables, but unfortunately we do not know with what value(s). Observing queries however we see that for many literals we do know the predicate (2nd argument of **rdf/3**) at query time, leaving two interesting cases: subject bound to unknown value and object unbound and the other way around. We deal with those by defining *subject branch factor* (sbf) resp. *object branch factor* (obf):

$$sbf(P) = \frac{triples(P)}{distinctSubjects(P)}$$

This figure is not cheaply maintained on incremental basis and therefore computed. The result is cached with the predicate and only recomputed if the number of triples on the predicate has changed considerably.

Boolean tests. Boolean tests resulting from the WHERE clause cannot cause branching. They can succeed or fail and their branching factor is estimated as 0.5, giving preference to locations early in the conjunction. This number may be wrong but, as we explained in Sect. 5, reordering of independent members of the conjunction only has marginal impact on the execution time of the query. If not all arguments to the test are sufficiently instantiated computation of the branching factor fails, causing the conjunction permutation generator to generate a new alternative.

The total complexity of a conjunction is now easily expressed as the summed sizes of the search spaces after executing $1, 2, \ldots n$ steps of the conjunction. The branching factor for each step is deduced using symbolic execution of the conjunction, replacing each variable in a literal with a Skolem instance. Skolem instantiation is performed using SWI-Prolog *attributed variables* [5].

7 Optimising the Conjunction

With a good and quick to compute metric for the complexity of a particular order, the optimisation problem is reduced to a generate-and-test problem. A

conjunction of N members can be ordered in $N!$ different ways. As we have seen actual examples of N nearing 40, naive permutation is not an option. We do not have to search the entire space however as the order of sub-conjunctions that do not share any variables can be established independently, after which they can be ordered on the estimated number of solutions.

Initially, for most conjunctions all literals are related. After having executed a few literals, the grounded variables often break the remaining literals in multiple independent groups that can be optimised separately. The algorithm is show in Fig. 7.

```
order(conj)
{ make_subgraphs(conj, subconjs);
  if ( count(subconjs) > 1 )
  { maplist(order, subconjs, ordered_subs);
    sort_by_complexity(ordered_subs, sorted);

    return join_subgraphs(sorted);
  } else
  { first = select(conj, rest);         (*)
    skolem_bind(first);
    make_subgraphs(rest, subconjs);
    maplist(order, subconjs, ordered_subs);
    sort_by_complexity(ordered_subs, sorted);

    return first + join_subgraphs(sorted);
  }
}
```

Fig. 7. Generating permutations of a conjunction. Note that select(), marked (*), is non-deterministic.

Combining this generator with the complexity estimate of Sect. 6 and selecting the best completes the order selection process. As the permutation algorithm only returns results from reordering independent subgraphs and it selects the best one by sorting the independent subgraphs on their branching, the returned order is guaranteed to be optimal if the complexity estimate is perfect. In other words, the maximum performance difference between the optimal order and the computed order is the error of our estimation function.

8 Optional Path Expressions and Control Structures

As explained in Sect. 4, SeRQL optional path expressions are compiled into (Goal `*->` true ; true), where *Goal* is the result of compiling the path expression. We must handle *Goal* as well as other goals appearing in Prolog control structures resulting from compiling the WHERE clause as a unit. If such units

are conjunctions they are subject to recursive invocation of our ordering algorithm. Please do note that the ordering and complexity of a conjunction depends on the variables that are already instantiated when the conjunction is entered. Conjunctions in control structures must therefore be ordered and have their complexity determined as part of estimating the complexity of the outer conjunction, as illustrated by the following simplified Prolog code fragment:

```
complexity((Goal0 *-> true ; true),
           (Goal *-> true ; true), Complexity) :-
        optimise(Goal0, Goal),
        complexity(Goal, Complexity).
```

Optional path expressions do not change the result set of the obligatory part of the query. It can only produce more variable bindings. Therefore we can simplify the optimisation process of a conjunction by first splitting it into its obligatory and optional part and then optimise the obligatory part followed by the optional part:

```
optimise(Goal0, Goal) :-
        split_optional(Goal0, Obligatory0, Optional0),
        optimise(Obligatory0, Obligatory),
        skolem_bind(Obligatory),
        optimise(Optional0, Optional),
        Goal = (Obligatory, Optional).
```

9 Solving Independent Path Expressions

As we have seen in Sect. 7, the number of distinctive permutations is much smaller than the number of possible permutations of a goal due to the fact that after executing a few literals the remainder of the query breaks down into independent subgraphs. Independent subgraphs can be solved independently and the total result is simply the Cartesian product of all partial results. This approach has several advantages:

- The complexity of solving two independent goals A and B separately is $b(A) + b(B)$ rather than $b(A) + b(A) \times b(B)$.
- If any of the independent goals has no solutions we can abort the whole query and report it has no solutions.
- The subgoals can be solved in parallel.
- The result-set can be expressed as the Cartesian product of partial results, requiring much less communication between server and client.
- It eliminates the need for the 'sort_by_complexity' step in Fig. 7.

This optimisation can be performed after the reordering. It simply does symbolic evaluation and Skolem instantiation of the conjunction statement-by-statement and splits the remainder into subgraphs. If conjunction is represented as a list, the simplified Prolog code fragment below suffices.

```
cartesian(Conjunction, Carhesian) :-
        append(Before, After, Conjunction),
        skolem_bind(Before),
        make_subgraphs(After, SubGraphs),
        SubGraphs = [_,_|_], !,         % demand at least two
        append(Before, serql_carthesian(SubGraphs), Carhesian).
cartesian(Conjunction, Conjunction).
```

This optimisation can be performed recursively on the created independent subgraphs as well as on conjunctions inside control structures.

10 Results

The total code size of the server is approximately 6,700 lines (including comments, but excluding the 25-line GPL file headers). Major categories are show in Tab. 1. We think it is not realistic to compare this to the 86,000 lines of Java code spread over 439 files that make up Sesame. Although both systems share considerable functionality, they differ too much in functionality and what parts are reused from the respective libraries to make a detailed comparison feasible.

Table 1. Size of the various components, counted in lines. RDF/XML I/O is only a wrapper around the SWI-Prolog RDF library.

Category	**lines**
HTTP server actions	2,521
Entailment modules (3)	281
Result I/O (HTML, RDF/XML, Turtle)	1,307
SeRQL runtime library	192
SeRQL parser and naive compiler	874
Optimiser	878
Miscellaneous	647

We have evaluated our optimiser on two domains, the already mentioned WordNet and an RDF database about cultural relations in Spain with real-life queries on this database. Measurements have been executed on a dual AMD 2600+ machine running SuSE Linux and SWI-Prolog 5.5.15.

We have tested our optimiser on artificial as well as real-life SeRQL queries. In all observed cases the optimisation time is only a modest fraction of the execution time of the optimal order as well as generally shorter than the time required to parse the query.

First we study the example of Fig. 6. Our optimiser converts either translation into the goal shown in Fig. 8. The code for **s1/1** was handcrafted by us and can be considered an educated guess for best performance. The result of the optimiser came as a surprise, but actual testing proved the code of Fig. 8 is 1.7 times faster than the code of **s1/1** and indeed the fastest possible. The optimiser requires only 90ms, or just 4.3% of the execution time for the optimal solution.

```
q(L) :-
        rdf(S1, rdf:type, C1),
        rdf(S1, wns:wordForm, L),
        rdf(C2, rdfs:subClassOf, wns:'LexicalConcept'),
        rdf(S2, rdf:type, C2),
        rdf(S2, wns:wordForm, L),
        C1 \== C2,
        rdf(C1, rdfs:subClassOf, wns:LexicalConcept))
```

Fig. 8. Optimised WordNet query

The second test-set consisted of three queries on a database of 97,431 triples coming from a real project carried out at Isoco[6] These queries were selected because Sesame [3] could not answer them (2 out of 3) or performed poorly. Later examination revealed these queries consisted of multiple largely independent sub-queries, turning the result in a huge Cartesian product. Splitting them into multiple queries turned them into manageable queries for Sesame. Exploiting the analysis of independent path expressions described in Sect. 9, our server does not need this rewrite. The results are shown in Tab. 2. We could only verify the result of the 2nd query against Sesame, which returns the same 3,826 solutions in 132.72 seconds.

Table 2. Results on complex queries. The engine has been modified slightly to return the Cartesian product as a description instead of expanding it as the expansion does not fit in memory.

		time (ms)	complexity			time (s)	
Id	Edges	optimise	initial	final	speedup	total	solutions
1	38	10ms	1.4e16	1.4e10	1e6	2.48s	79,778,496
2	30	10ms	2e13	1.3e5	1.7e8	0.51s	3,826
3	29	10ms	1.4315	5.1e7	2.7e7	11.7s	266,251,076

11 Related Work

Using logic for Semantic Web processing has been explored by various research groups. See for example [12] which exploits *Denotational Semantics* to provide a structured mapping from language to semantics. Most of these approaches concentrate on correctness, while we concentrate on engineering issues and performance.

Much work has been done on optimising Prolog queries as well as database joins by reordering. We specifically refer to the work of Struyf and Blockeel [16] because it is recent and both the problem and solution are closely related. They describe the generation of programs through ILP [11]. The ILP system itself

[6] www.isoco.com

does not consider ordering for optimal execution performance, which is similar to compiling declarative SeRQL statements not producing optimal programs. In ILP, the generated program must be used to test a large number of positive and negative examples. Optimising the program before running is often worthwhile.

The described ILP problem differs in some places. First of all, for ILP one only has to prove that a certain program, given a certain input succeeds or fails, i.e. goals are ground. This implies they can use the cut to separate independent parts of the conjunction (section 4.2 of [16]). As we have non-ground goals and are interested in all distinct results we cannot use cuts but instead use the Cartesian product approach described in Sect. 9. Second, Struyf and Blockeel claim complexity of generate-and-test (order $N!$) is not a problem with the observed conjunctions with a maximum length of 6. We have seen conjunctions with 40 literals. We introduce breaking the conjunctions dynamically in independent parts (Sect. 7) can deal with this issue. Finally, the uniform nature of our data gave us the opportunity to build the required estimates for non-determinism into the low-level data structures and maintain them at low cost (Sect. 6).

12 Discussion

Sofar, we have been using Prolog in the Semantic Web domain for reasoning for annotation [13]. This reasoning was not based on formal Semantic Web languages, but using ad-hoc defined schemas. With our SeRQL implementation we have proven that we can deal completely and efficiently with RDFS. We have proven that SWI-Prolog, supporting threading, attributed variables and equipped with extensive libraries for graphics, XML, RDF triple store and HTTP is a suitable tool for building a variety of Semantic Web applications, covering both interactive and network server applications.

As the Semantic Web evolves with more powerful formal languages such as OWL and SWRL[7], it becomes unlikely we can compile these easily to efficient Prolog programs. TRIPLE [15] is an example of an F-logic based RDF query language realised in XSB Prolog [7]. We believe extensions to Prolog that facilitate more declarative behaviour will prove necessary to deal with the Semantic Web. Both XSB's tabling and constraint logic programming, notably CHR [14] are promising extensions.

13 Conclusions

We have employed Prolog for storing and querying Semantic Web data. In [18] we have demonstrated the performance and scalability of the storage module for use in Prolog. In this paper we have demonstrated the feasibility realising an efficient implementation of the declarative SeRQL RDF query language in Prolog. The resulting system can easily be expanded with new entailment reasoners and can be accessed from both Prolog and Java through the common HTTP interface.

[7] http://www.daml.org/2003/11/swrl

The provided algorithm for optimising the matching process of SeRQL queries reaches optimal results if the complexity estimate is perfect. The worse case complexity of ordering a conjunction is poor, but for tested artificial and real-life queries the optimisation time is shorter than the time needed to execute the optimised query. For trivial queries this is not the case, but here the response time is dictated by the HTTP protocol overhead and parsing the SeRQL query.

The SeRQL server is available under the SWI-Prolog LGPL/GPL license from http://www.swi-prolog.org/packages/SeRQL.

Acknowledgements

We would like to thank Oscar Corcho for providing real-life data and queries. This research has been carried out as part of the HOPS project[8], IST-2002-507967. Jeen Broekstra provided useful explanations on SeRQL.

References

1. Tim Berners-Lee, James Hendler, and Ora Lassila. The semantic Web. *Scientific American*, 284(5):34–43, May 2001.
2. D. Brickley and R. V. Guha (Eds). Resource description framework (RDF) schema specification 1.0. W3C Recommendation, March 2000. http://www.w3.org/TR/2000/CR-rdf-schema-20000327/.
3. Jeen Broekstra, Arjohn Kampman, and Frank van Harmelen. Sesame: A generic architecture for storing and querying rdf and rdf schema. In I. Horrocks and J. Hendler, editors, *Proceedings of the First Internation Semantic Web Conference*, number 2342 in Lecture Notes in Computer Science, pages 54–68. Springer Verlag, July 2002.
4. Mike Dean, Guus Schreiber, Frank van Harmelen, Jim Hendler, Ian Horrocks, Deborah L. McGuinness, Peter F. Patel-Schneider, and Lynn Andrea Stein. OWL web ontology language reference. Working draft, W3C, March 2003.
5. Bart Demoen. Dynamic attributes, their hProlog implementation, and a first evaluation. Report CW 350, Department of Computer Science, K.U.Leuven, Leuven, Belgium, oct 2002. URL = http://www.cs.kuleuven.ac.be/publicaties/rapporten/cw/CW350.abs.html.
6. Carlos Escalante. A simple model of prolog's performance: extensional predicates. In *CASCON '93: Proceedings of the 1993 conference of the Centre for Advanced Studies on Collaborative research*, pages 1119–1132. IBM Press, 1993.
7. Juliana Freire, David S. Warren, Konstantinos Sagonas, Prasad Rao, and Terrance Swift. XSB: A system for efficiently computing well-founded semantics. In *Proceedings of LPNMR 97*, pages 430–440, Berlin, Germany, jan 1997. Springer Verlag. LNCS 1265.
8. Markian M. Googley and Benjamin W. WAH. Efficient reordering of PROLOG programs. *IEEE Transactions on Knowledge and Data Engineering*, pages 470–482, 1989.

[8] http://www.hops-fp6.org

9. Peter Haase, Jeen Broekstra, Andreas Eberhart, and Raphael Volz. A comparison of rdf query languages. In *Proceedings of the Third International Semantic Web Conference, Hi roshima, Japan, 2004.*, NOV 2004.
10. G. Miller. WordNet: A lexical database for english. *Comm. ACM*, 38(11), November 1995.
11. S. Muggleton and L. De Raedt. Inductive Logic Programming: Theory and Method. *Journal of Logic Programming*, 19-20:629–679, 1994.
12. Kunal Patel and Gopal Gupta. Semantic processing of the semantic web. *Lecture Notes in Computer Science*, 2870:80–95, January 2003.
13. Guus Schreiber, Barbara Dubbeldam, Jan Wielemaker, and Bob Wielinga. Ontology-based photo annotation. *IEEE Intelligent Systems*, may/june 2001.
14. Tom Schrijvers and Bart Demoen. The K.U. Leuven CHR system: implementation and application. In Thom Frühwirth and Marc Meister, editors, *First Workshop on Constraint Handling Rules: Selected Contributions*, pages 430–440, 2004. ISSN 0939-5091.
15. Michael Sintek and Stefan Decker. *TRIPLE* — A query, inference, and transformation language for the Semantic Web. *Lecture Notes in Computer Science*, 2342:364–, 2002.
16. J. Struyf and H. Blockeel. Query optimization in inductive logic programming by reordering literals. In T. Horváth and A. Yamamoto, editors, *Proceedings of the 13th International Conference on Inductive Logic Programming*, volume 2835 of *Lecture Notes in Artificial Intelligence*, pages 329–346. Springer-Verlag, 2003.
17. Jan Wielemaker. Native preemptive threads in SWI-Prolog. In Catuscia Palamidessi, editor, *Practical Aspects of Declarative Languages*, pages 331–345, Berlin, Germany, december 2003. Springer Verlag. LNCS 2916.
18. Jan Wielemaker, Guus Schreiber, and Bob Wielinga. Prolog-based infrastructure for RDF: performance and scalability. In D. Fensel, K. Sycara, and J. Mylopoulos, editors, *The Semantic Web - Proceedings ISWC'03, Sanibel Island, Florida*, pages 644–658, Berlin, Germany, october 2003. Springer Verlag. LNCS 2870.

A Distributed and Probabilistic Concurrent Constraint Programming Language

Luca Bortolussi[1] and Herbert Wiklicky[2]

[1] Dep. of Maths and Computer Science, University of Udine, Udine, Italy
bortolussi@dimi.uniud.it
[2] Dep. of Computing, Imperial College, London, UK
herbert@doc.ic.ac.uk

Abstract. We present a version of the CCP paradigm, which is both distributed and probabilistic. We consider networks with a fixed number of nodes, each of them possessing a local and independent constraint store. While locally the computations evolve asynchronously, following the usual rules of (probabilistic) CCP, the communications among different nodes are synchronous. There are channels, and through them different objects can be exchanged: constraints, agents and channel themselves. In addition, all this activities are embedded in a probabilistic scheme based on a discrete model of time, both locally and globally. Finally we enhance the language with the capability of performing an automatic remote synchronization of variables belonging to different constraint stores.

1 Introduction

In this paper we present a probabilistic and distributed version of Concurrent Constraint Programming (CCP). This version integrates CCP [17] in a network framework which resembles KLAIM [3], but which has also some features of π-calculus [14]. In addition, the language is provided with a probabilistic semantic.

Concurrent Constraint Programming was introduced in [17], and it is based on a computational paradigm founded on constraints. In particular, the usual Von Neumann's store is replaced by a store that contains constraints on the variables into play. Computations evolve monotonically: constraints can only be added in the store, but never removed. The idea behind this approach is to attach to variables not a single value, but rather an interval of possible values, which is refined as long as new information, in the form of constraints, is available. An important feature about this language is that it allows a concurrent reasoning about problems which naturally involve constraints, like optimization problems. In a CCP program, different agents can run in parallel, and the communication between them is performed through shared variables in the constraint store. In particular, agents can either tell a constraint in the constraint store, or they can ask if a particular relation is satisfied by the current configuration of the system. Constraint system are recalled in Section 3.

In this work we extend CCP in two directions: following [5], we provide it with a probabilistic semantic and we introduce the possibility of distributing programs in a network of computing machines. There are several reasons for having at disposal a probabilistic and distributed version of CCP. First of all, distributed randomized algorithms

M. Gabbrielli and G. Gupta (Eds.): ICLP 2005, LNCS 3668, pp. 143–158, 2005.

are getting more and more popular. In fact, several languages have been proposed to describe them, like [11] and [15]. In particular, a lot of efforts are put in developing distributed metaheuristics for combinatorial optimization algorithms (cf. [16] and [13]), which involve an objective function to optimize and constraints on the variables into play. These features are naturally described in a constraint-based framework, like the one offered by CCP, while in other languages their description could be cumbersome. Therefore, extending CCP with both probabilistic and distributed features makes possible a fast prototyping of these algorithms, and also permits to perform some reasoning on them.

The extension of CCP in a distributed setting presents some obstacles. The main problem is essentially related to the way communication occurs in CCP. In fact, processes interact by posting constraints on global shared variables, and therefore communication modifies globally the system. But this form of communication is very unsuitable in a distributed environment: here we need a clear distinction between the evolution of the computation at the local level (the nodes of the network), and the global interactions of the single nodes. Hence there's the necessity of providing CCP with new primitives taking into account this distinction between the global level and the local one. Approaches in this direction can be found in [9,8,1,12,20], and we discuss them in more detail in Section 2.

Our main idea is to cast CCP into a KLAIM-like description of the network (cf. [3]). The KLAIM language is a distributed version of the LINDA paradigm, with the peculiar characteristic that it presents a clear distinction between the local level and the network level. The basic distributed components are nodes, which are then combined together to form a network. The topology of the network is encoded in the *environment* functions, which assign to each locality variable the address of a particular node. In the language presented, we define the basic distributed entity as a node, and then we compose nodes together to form a network. In each node the computation evolves (locally) according to the CCP paradigm: we have several agents which interact with ask and tell primitives. Moreover, each node has its own constraint store. While composing together nodes, we have to model properly the "union" of the local constraint store, and we do this via a direct product construction (cf. Section 3). This means that constraint stores located at different nodes are completely independent, hence the local actions performed in a node cannot interfere with the computations going on in other localities. The choice of the direct product, in particular, is suitable for the application of probabilistic abstract interpretation techniques [6] to perform a statical analysis of programs, like in [7]. The main problem we have to face, however, relates to the flow of information between different nodes. In fact, in CCP information can be exchanged only using global shared variables. But all our variables are local, and CCP offers no solution to transfer information between local and independent variables. In fact, the first, naive, idea of letting an agent tell constraints to other constraint stores is not going to work, due to variable name clashing. In addition, even if the name conflicts can be solved in some way, an agent will still be unable to match pieces of information located in different parts of the network.

Therefore, to let nodes exchange information, we must define new primitives. Following the approach in [9] and [1], we define a synchronous message-based

communication which resembles the π-calculus paradigm: there are channels and messages flowing along them.

We pinpoint that the mechanisms presented up to now create a neat distinction between the local level, consisting of nodes and of computations localized in them, and the global level, related to the network, its topology, its dynamics. This distinction allows us to model separately the evolution of the single nodes and of the network, thus simplifying the task of adding probabilities to the system. In detail, we decide to use a discrete time both for local and global dynamics (which means probabilistic scheduling policies), but we impose an asynchronous local communication versus a synchronous global one. The price of this distinction is an increased complexity both in the description of the configurations and in the transition rules.

In this language there are three basic objects that can be communicated along the network: constraints, channel names and agents. To communicate constraints, however, we must tackle the problem of variable name conflicts between different nodes. This is achieved by "abstracting" a constraint with respect to a subset of its free variables (hiding all the others), thus creating a kind of template, whose (template) variables can be replaced by suitable ones specified by the receiver. We give more details in Section 4.1. Channels can be exchanged in a way that resembles closely the π-calculus. Their communication allows to reconfigure the topology of the network, by changing the communication links. In addition, new channel names can be created dynamically, providing a form of private communication. The third object that can be communicated are agents. As for the case of constraints, we need to abstract them in order to avoid variables clashes. Therefore, (some of) the free variables of a transmitted agent are replaced by suitable ones belonging to the receiving node. In addition, an agent that migrates can carry with itself some information about (part of) its free variables.

Another feature that may be needed in a distributed CCP is the possibility of synchronizing variables belonging to different constraint stores. This would allow for an automatic information flow between different nodes, and would greatly simplify the writing of programs. This linking of variables is realized by adding information about linked variables directly in the configuration of the system, at the network level. We refer to Section 6 for further details.

We want to remark the important fact that the language not only is distributed, but it also evolves following probabilistic rules, meaning that every form of non-determinism is solved probabilistically. As already said, this feature allows for an easy description of distributed randomized algorithms and especially, due to the constraint-based framework, parallel stochastic optimization algorithms. In addition, the different primitives for local and global communication make possible to model such algorithms in different ways, permitting a rapid comparison between them.

The paper is organized as follows: in Section 2 we present some related work. In Section 3 we recall the concept of constraint system, while in Section 4 we introduce the syntax of the language. In Sections 4.1 and 4.2 we spend some words on constraint and agent abstractions, and we discuss some issues related with communication channels. The Structural Operational Semantics is introduced and discussed in detail in Section 5, while Section 6 deal with the linking mechanisms. Finally in Section 7 we draw the final conclusions.

2 Related Work

In this Section we discuss briefly some related work. In particular, CCP has been extended both in the probabilistic setting and in the distributed one. As far as we know, there is no extension covering both aspects together.

Probabilistic CCP (pCCP) has been introduced in [5] to declaratively model randomized algorithms. It is now a quite well established language, with its own operational [5] and denotational [4] semantic. There is also a probabilistic version of abstract interpretation [6] that can be used to perform some static analysis of pCCP programs.

On the other hand, there has been a lot of work also for devising a distributed version of CCP. This is not an easy task, due to the fact that communication in CCP proceeds by posting constraints acting on global variables. To tackle this problem, different forms of communication between agents have been added. In [9], Réty proposes a π-calculus approach to model the communication between agents located in different nodes. In particular, new instructions for exchanging messages are introduced, even if there is still a single, global, constraint store, and the independence among variables is imposed as a further condition in the definition of the agents. In addition, the concepts of constraint abstraction and linking are defined. In [1], De Boer et alt. present a similar approach, with global communication performed via message passing and local communication performed in CCP style. They also introduce a concept of independent local stores, even if the global configuration of a network is taken to be the least upper bound of the local constraint stores (cf. Section 3). In [12] a different type of synchronous communication is discussed, based on a redefinition of the semantics of ask and tell primitives: a constraint is told only if there in an agent asking for it. Also this version makes use of a single global constraint store.

Our approach towards distribution shares some features with the previous approaches, but we impose a strict independence between local constraint stores, such that the interaction between localized variables must always be made explicit. This division creates a clear distinction between the local and the global level, at the price of introducing more instructions and a more complex representation of the configuration of the system. This increased complexity, however, allows a clear modelling the probabilistic evolution of the system, by distinguishing between local and global time. In addition, this separation can be exploited in the context of probabilistic abstract interpretation [6], in particular in the static analysis of distributed networks, as done by Di Pierro et alt. in [7]. Finally, having a concept of computational site at disposal, we can define easily mobility of agents.

Regarding the migration of CCP agents, work has been done by Palamidessi et alt. in [8], where CCP is enriched by a hierarchical network structure and agents can move from one node to another one (bringing with them their subagents). Our network structure, instead, is simpler (there is no hierarchical organization), as we focused mainly on the communication issues.

3 Background and Preliminaries

Computations in CCP are performed through a monotonic update of the so called constraint store, which is usually modeled as a constraint system. We follow here a well

established approach (cf. [18] or [2]), which represents a constraint system as a complete algebraic lattice, where the ordering $\sqsubseteq$ is given by the inverse of the entailment relation $\vdash$. Usually, such a constraint system is derived from a first-order language together with an interpretation, where constraints are formulas of the language, and a constraint c entails a constraint d, $c \vdash d$, if every valutation satisfying c satisfies also d. In this case we write $d \sqsubseteq c$. Clearly, in every real implementation the predicate $\vdash$ must be decidable. In addition, to model hiding and parameter passing, the previous lattice is enriched with a cylindric algebraic structure (cf. [10]), i.e. with cylindrification operators and diagonal elements.

Formally, a constraint system $\mathcal{C} = (Con, Con_0, Var, \sqsubseteq, \sqcup, true, false, \exists_x, d_{xy})$ is a complete algebraic lattice where Con is the set of constraints, ordered by $\sqsubseteq$, Con_0 is the set of finite elements, $\sqcup$ is the least upper bound (lub) operation, Var is the set of variables, $true$ and $false$ are respectively the bottom and the top element, $\{\exists_x \mid x \in Var\}$ are the cylindrification operators and $\{d_{xy} \mid x, y \in Var\}$ are the diagonal elements.

In our distributed version of CCP, we assign an independent constraint store to each node composing the network: if we have m nodes in our system, then we have m constraint systems $\mathcal{C}_1, \ldots, \mathcal{C}_m$. Therefore, we can model a global configuration of the network by the direct product $\mathcal{C}_1 \times \ldots \times \mathcal{C}_m$.This product, where all the operations are performed elementwise, is still a cylindric algebra. The proof of this fact is straightforward, and follows from more general results in [10]. A fundamental property of this construction is that the variables are independent between different constraint stores.

4 Syntax

The distributed network is basically composed by a set of nodes, linked between them by communication channels. Every node contains CCP-based processes, while the CCP syntax is enriched with communication mechanisms at the network level.

We need several different syntactic objects:

1. *physical localities*, or *sites*, which are the addresses of the nodes of the network, and are taken from a set S. Every node must have a distinct address, and we can assume S to be countable and indexed by the natural numbers (i.e. $S = \{s_1, \ldots, s_n, \ldots\}$). We will impose, however, a restriction on the actual number of nodes that can be used in a program, i.e. we will assume that they are finite. Often we refer to a node with address s_j simply as node j;
2. *channel names* and *channel variables*. Channels specify the topology of the network, and they are communication routes between nodes. Channel names are taken from the set $\mathcal{L}$, and are indicated by Greek letters α, β, and so on. Channel variables are taken from the set $Var_{\mathcal{L}}$, and are indicated by underlined Greek letters, like $\underline{\alpha}$, $\underline{\beta}$;
3. *environments*. They are functions which associate to each channel a probability distributions over S. Formally, they are indicated by ϱ, and $\varrho : \mathcal{L} \to \mathcal{D}(S)$, where $\mathcal{D}(S)$ is the set of probability distributions over S. Therefore, environments specify how transmissions are performed through shared channels. We comment more on this fact in Section 5.

A node in the network is defined as:

$$n \ ::= \ s ::_{\varrho}^{p} P,$$

where P is the current agent that runs on this node, s is the address of the node, taken from S, ϱ is the environment associated to node n and p is a probability associated to n. This is the probability with which node n is chosen for execution by the global scheduler.

A network N is defined as a parallel composition of a finite number of nodes:

$$N \ ::= \|_{i=1}^{m} \ n_i.$$

To have a probability distribution over the network, the probabilities associated to each node should sum up to one. However, we omit this request, as we are going to deal with normalization structurally, i.e. via a congruence relation (cf. next Section). Moreover, the scheduling probabilities are fixed, and cannot change during the execution of the program.

We note that posing an upper bound on the number of nodes which compose a network is not a real limitation, as we can choose this number very high, with possibly many inactive nodes at the beginning of the computation which can be populated by processes in the future.

In Table 1 we describe the syntax of an agent. It is composed by a declaration of procedures and by an initial goal.

Table 1. Syntax of pDCCP

$$
\begin{array}{l}
Program = Decl.A \\
D = \varepsilon \mid Decl.Decl \mid p(\overrightarrow{x}) : -A \\
A = \mathbf{0} \mid \mathrm{tell(c).A} \mid \exists_{\mathrm{x}}\mathrm{A} \mid \mathrm{p}(\overrightarrow{x}) \mid \sum_{\mathrm{i=1}}^{\mathrm{k}} \mathrm{q_i} : \mathrm{ask(c_i).A_i} \mid \ |_{\mathrm{i=1}}^{\mathrm{k}} \mathrm{q_i} : \mathrm{A_i} \mid \\
\qquad \mathrm{out_c}(\lambda\overrightarrow{x}\, c)@\alpha.A \mid \mathrm{in_c}(\overrightarrow{y})@\alpha.A \mid \mathrm{out_{loc}}(\beta)@\alpha.A \mid \mathrm{in_{loc}}(\underline{\beta})@\alpha.A \mid \mathrm{new}(\underline{\beta}).A \mid \\
\qquad \mathrm{out_A}(\lambda\overrightarrow{x}\, A, \overrightarrow{x_0})@\alpha.A \mid \mathrm{in_A}(\overrightarrow{y})@\alpha.A \qquad (\overrightarrow{x_0} \subseteq \overrightarrow{x} \subseteq fv(A))
\end{array}
$$

The declaration section is standard, and we ask that $fv(A) \subseteq \overrightarrow{x}$, for all declarations of the form $p(\overrightarrow{x}) : -A$. The first 6 instructions of agent syntax are also standard and they are taken from the probabilistic version of CCP (cf. [5]). The remarkable facts are the probabilistic choice operator (thus non-determinism is replaced by probabilistic choice), and the probabilistic version of the parallel operator. The probability distribution associated to it introduces priorities in the local scheduler, and thus may bias the interleaving of processes. In both these instructions, q_i represents the probability associated either to the branch i or to the parallel agent i.

The out and in instructions are the primitives for communication. They appear in three different forms, as there are three different objects that we want to send over channels, i.e. constraint (abstractions), channel names and agent (abstractions). Constraint and agent abstractions are indicated respectively with $\lambda\overrightarrow{x}\, c$ and $\lambda\overrightarrow{x}\, A$; cf. below for an

explanation. Finally, the new instruction creates new channel names, in order to dynamically reconfigure the topology of the network.

All global communication instructions are of the form $\mathrm{out}(\cdot)@\alpha$ or $\mathrm{in}(\cdot)@\alpha$, where α can be both a channel name or a channel variable. Clearly, actual communications can be performed only on real channels, not on channel variables. Therefore, we must impose that all channel variables are bounded, i.e. they appear in the scope of an $\mathrm{in}_{\mathrm{loc}}$ or a new instruction. We write $A[\beta/\underline{\beta}]$ to indicate that the channel variable $\underline{\beta}$ has been replaced by the channel name β.

4.1 Constraint and Agent Abstractions

The independence of variables among different constraint stores poses some problems in the communications performed between nodes. In particular, if a process at node i wants to communicate a constraint c, then we have the problem that variables in c are related to the constraint store $\mathcal{C}_i$. If c is sent to another node, say j, then we must specify which variables of $\mathcal{C}_j$ it refers to. This choice can be performed by the receiving process at node j. However, to perform this renaming, we have to abstract the variables present in c, in order to create a kind of constraint template, where the new variables can be put. We follow the approach introduced in [9], making use of constraint abstractions.

A *constraint abstraction* is a couple $(\overrightarrow{x}, c)$, where $\overrightarrow{x} \subseteq fv(c)$, and will be denoted by $\lambda\overrightarrow{x}c$. The variables $\overrightarrow{x}$ are the template variables, and all other free variables of c are hidden (that is to say, c is projected over $\overrightarrow{x}$). We can define the projection operator $\mathit{\Pi}_{\overrightarrow{x}}c = \exists_{fv(c)\setminus\overrightarrow{x}}c$, i.e. we hide all the variables except the ones in $\overrightarrow{x}$. Then, if $\overrightarrow{y}$ is a vector of variables of the same length of $\overrightarrow{x}$, the application of $\overrightarrow{y}$ to $\lambda\overrightarrow{x}c$ is the constraint $(\mathit{\Pi}_{\overrightarrow{x}}c)[\overrightarrow{y}/\overrightarrow{x}]$.

The independence among variables creates an analogous problem if we want to move entire agents along the network. Therefore, we need to define also an abstract version of agents. Given an agent A and a subset $\overrightarrow{x}$ of its free variables $fv(A)$, an agent abstraction is a couple $(A, \overrightarrow{x})$, denoted by $\lambda\overrightarrow{x}A$. The variables $\overrightarrow{x}$ are "templates" that must be substituted by variables $\overrightarrow{y}$ belonging to the constraint store of the receiving node. All the free variables of A different from $\overrightarrow{x}$, instead, must be hidden, and consequently we define a projection operator also for agents: $\mathit{\Pi}_{\overrightarrow{x}}(A) = \exists_{fv(A)\setminus\overrightarrow{x}}A$.

4.2 Communication Channels

The communication we have at the global level is synchronous and message-based. The exchange of information between nodes in performed through communication channels shared among them. These channels, however, cannot be represented by the usual variables (as in [9]), because constraint stores are totally independent. Therefore, we need to define a new syntactical category of objects, with their own properties (as in [1]). Channels will be denoted by names, identified by Greek letters α, β, and so on, all belonging to the set $\mathcal{L}$. Each channel name identifies a unique channel, i.e. an idealized mean or link which represents communication bridges between different nodes of the network. To allow a dynamical reconfiguration of the topology of the network, we have also channel variables (indicated by $\underline{\alpha}, \underline{\beta} \in Var_{\mathcal{L}}$), which can be substituted, during the execution of the program, by a channel name. Because of the fact that communications

can only happen in channels, we must ask that each channel variable present in an agent is bounded, i.e. it appears under the scope of an $\mathrm{in}_{\mathrm{loc}}$ or new instruction. We suppose that the set of channel's names is infinite.

Channels introduce a form of non-determinism in the network communications. In fact, a channel α can be shared by several nodes, and at some time of the execution of the program, we can have a process that wants to send a message along α and a bunch of agents which may be able to receive it. This non-determinism is solved probabilistically by the environment ϱ, which is a function assigning to each channel name a probability distribution over nodes (or better, over node's addresses). Note, however, that the requirement that each node has an a-priori defined probability distribution assigned to each channel name is too strong. To relax this condition, we can proceed as follows: we define a global environment ϱ_N, which assigns to every channel the uniform probability over nodes. Then we allow the environments ϱ'_i assigned to nodes to be partial functions from $\mathcal{L}$ to $\mathcal{D}(S)$. The "real" environment ϱ_i associated to a node is $\varrho_i = \varrho_N \bullet \varrho'_i$, where $\bullet$ means that ϱ'_i is extended by ϱ_N, wherever it is undefined. In particular, we ask that the environment of every node is undefined for channels created by the instruction new.

We observe that the fact having probability distributions attached to channels let us define "unidirectional" communications, i.e. channels where the communication can happen in just one sense. In fact, consider a channel α shared between nodes i and j, such that $\varrho_i(\alpha)(s_j) > 0$ and $\varrho_j(\alpha)(s_i) = 0$: node j can only receive messages from node i along α, but can never send something to i ($\varrho(\alpha)(s) > 0$ is a condition in communication rules, cf. Section 5).

5 Operational Semantics

The operational semantic of the language is given by a congruence relation between agents, and by a transition relation between configurations, labeled with probabilities. As for the syntax, both these objects have two versions, one local and one global.

A configuration of the system at one particular node will be an element of $\mathcal{P} \times \mathcal{C}$, where $\mathcal{P}$ is the space of processes and $\mathcal{C}$ is the constraint store associated to the node. Therefore, it is a couple $\langle A, c\rangle$, with $A \in \mathcal{P}$ and $c \in \mathcal{C}$. Consequently, a configuration of the network will be of the form $s_1 ::^{p_1}_{\varrho_1} \langle A_1, c_1\rangle \parallel \ldots \parallel s_m ::^{p_m}_{\varrho_m} \langle A_m, c_m\rangle$, which can be written also ${}^{s_1}_{p_1}\langle A_1, c_1\rangle_{\varrho_1} \parallel \ldots \parallel {}^{s_m}_{p_m}\langle A_m, c_m\rangle_{\varrho_m}$, or equivalently as $\langle A_1, \ldots, A_m, c_1, \ldots, c_m\rangle$. That is to say, a global configuration will be a point of $\mathcal{P}_1 \times \ldots \times \mathcal{P}_m \times \mathcal{C}_1 \times \ldots \times \mathcal{C}_m$, where $\mathcal{P}_i$ and $\mathcal{C}_i$ indicate respectively the space of processes and the constraint store of node i.

The congruence relation is twofold. One relation regards agents, and it splits them into an equivalence class. The other relation, instead, regards network configurations. They are both defined in Table 2.

Rules (**CR1**) and (**CR2**) simply state that the order of a sum and of a parallel composition is immaterial. In fact, we ask that two agents are congruent if one is obtained by the other simply permuting the composing terms. (**CR3**) and (**CR4**) deal with normalization issues. In fact, a basic requirement of probabilistic declarative languages is that the values associated to the choice or parallel operator are positive and add up to one, i.e. they are a probability distribution. This implies, for instance, that, whenever an

agent is added or removed from a parallel composition, we have to go through a process of renormalization of the associated probabilities. Instead of doing this explicitly, we define the normalization in the congruence relation: two sums or two parallel compositions, different just in the numerical weights, are congruent if they are equal after a normalization of the coefficients. In practice, the congruence class of every choice or parallel agent, with associated vector of weights $\overrightarrow{q}$, contains all processes which coefficients are positive multiples of $\overrightarrow{q}$. We always choose as representative of this class the unique agent for which $\overrightarrow{q}$ is a p.d.

(**CR5**), instead, states that adding a null agent to a parallel composition does not modify the program. Note that adding or removing a null agents modifies the vector of coefficients. However, we do not need to renormalize it, as this is done automatically by rule (**CR4**). Rules (**CR6**), (**CR7**) and (**CR8**) simply state the basic properties of the hiding operator. Finally, rule (**CR9**) affirms the the order of parallel composition of nodes in a network is irrelevant, while rule (**CR10**) implements the automatic normalization trick also at the network level. We observe also that the algebraic laws associated to a constraint system (cf. [2]) induce an implicit congruence relation over the constraint store at each node.

The main ingredient of the SOS is the labeled transition relation (LTR). There are two of such relations, one for the local evolutions and one for the network's one. Both are labeled by a real number in $[0, 1]$, representing the *probability associated to each transition*. Local transitions are stated in Table 3, while in Table 4 the global transition system is described. Formally, the local transition relation is indicated by $\longrightarrow$ and it is a subset of $\mathcal{P}\times\mathcal{C}\times[0,1]\times\mathcal{P}\times\mathcal{C}$, while the global transition relation is represented by $\Longrightarrow$, and it is a subset of $\mathcal{P}_1\times\ldots\times\mathcal{P}_m\times\mathcal{C}_1\times\ldots\times\mathcal{C}_m\times[0,1]\times\mathcal{P}_1\times\ldots\times\mathcal{P}_m\times\mathcal{C}_1\times\ldots\times\mathcal{C}_m$.

Before entering into a detailed description of each transition rule, we must explain a general trick in the notation. Suppose at some point of the computation the current agent is a probabilistic choice. Generally, not all ask guards will be entailed, so we have to renormalize the probability distribution (p.d.) over the entailed branches. Formally, if the current configuration of the node is $\left\langle \sum_{i=1,\ldots,k} q_i : ask(c_i).A_i, d \right\rangle$, we define the

Table 2. Congruence relation

$(CR1)$	$\sum_{i=1}^{k} q_i : \mathrm{ask}(c_i).A_i \equiv \sum_{i=1}^{k} q_{\pi(i)} : \mathrm{ask}(c_{\pi(i)}).A_{\pi(i)},$	for all permutations π
$(CR2)$	$\|_{i=1}^{k}\ q_i : A_i \equiv \|_{i=1}^{k}\ q_{\pi(i)} : A_{\pi(i)},$	for all permutations π
$(CR3)$	$\sum_{i=1}^{k} q_i : \mathrm{ask}(c_i).A_i \equiv \sum_{i=1}^{k} \widetilde{q}_i : \mathrm{ask}(c_i).A_i$	where $\widetilde{q}_j = \frac{q_j}{\sum_{i=1}^{k} q_i}$
$(CR4)$	$\|_{i=1}^{k}\ q_i : A_i \equiv \|_{i=1}^{k}\ \widetilde{q}_i : A_i$	where $\widetilde{q}_j = \frac{q_j}{\sum_{i=1}^{k} q_i}$
$(CR5)$	$q_1 : \mathbf{0} \mid q_2 : A_2 \mid \ldots \mid q_k : A_k \equiv q_2 : A_2 \mid \ldots \mid q_k : A_k,$	
$(CR6)$	$\exists_x \exists_y A \equiv \exists_y \exists_x A$	
$(CR7)$	$\exists_x A \equiv \exists_y A[y/x]$	if y is not free in A
$(CR8)$	$\exists_x A \equiv A$	if x is not free in A
$(CR9)$	$\|_{i\in S}\ {}^{s_i}_{p_i} \langle A_i, c_i\rangle_{\varrho_i} \equiv \|_{i\in S}\ {}^{s_{\pi(i)}}_{p_{\pi(i)}} \langle A_{\pi(i)}, c_{\pi(i)}\rangle_{\varrho_{\pi(i)}}$	$\forall\pi$ permutation of S
$(CR10)$	$\|_{i\in S}\ {}^{s_i}_{p_i} \langle A_i, c_i\rangle_{\varrho_i} \equiv \|_{i\in S}\ {}^{s_i}_{\widetilde{p}_i} \langle A_i, c_i\rangle_{\varrho_i}$	where $\widetilde{p}_j = \frac{p_j}{\sum_k\ {}_{S}\, p_k}$

subset $Active \subset \{1, \ldots, k\}$ as $Active = \{i | d \vdash c_i\}$, and then, if $j \in Active$, its normalized probability is $\tilde{q}_j = q_j / \sum_{i \in Active} q_i$. Similar considerations apply also to the local and global parallel constructs. In general, we adopt the notational convention that, if q is a probability distribution associated to an instruction, then $\tilde{q}$ is the same probability distribution normalized over active components.

The transition rules that are sketched out in Table 3 resemble closely the one presented for pCCP in [5]. Rule (**LR1**) models the tell operation. This operation adds the constraint c to the store, and always succeeds with probability one. (**LR2**) deals with the probabilistic choice, and the probability associated to the transition is the normalized probability over active guards (cf. above). Rule (**LR3**) regards parallel composition of local agents. It states that, if a local transition can happen with probability p, then the same action can happen in a parallel composition, with probability $p\tilde{q}_j$, where $\tilde{q}_j$ is the scheduling probability normalized among active agents. Rules (**LR4**) and (**LR5**) are the usual rules for modelling the action of the hiding operator and the parameter passing (for a discussion of the operator $\Delta_y^x A = \exists_y^{d_{xy}} A$ see [2]). Actually, the delta operator defined here links vector of variables and not single variables, but it is a straightforward generalization: $\Delta_{\overrightarrow{x}}^{\overrightarrow{y}} = \Delta_{x_k}^{y_k} \cdots \Delta_{x_1}^{y_1}$, if $|\overrightarrow{x}| = |\overrightarrow{y}| = k$. Rule (**LR6**) is the only outsider, as it gives the semantics of the new instruction, which deals with dynamic reconfiguration of the network connections. In particular, its effect is that of creating a fresh channel name β_{new} and binding the channel variable $\underline{\beta}$ to it. Note that, according to the convention of Section 4.2, all environments associate a uniform p.d. to new channel names.

Table 4 contains the definition of the transition relation at the network level. Rule (**GR1**) links the local transition and the global ones. It says that, whenever a local action can be performed in node i with probability p, then a global transition of the network can be performed with a probability p multiplied by $\tilde{p}_i$, i.e. by the probability that the global scheduler chooses node i for execution, normalized among active nodes. Clearly a node is active if and only if it can perform a transition, be it a local update or a global communication.

Rules (**GR2**) to (**GR4**) concern the global communication between nodes of the network.

Table 3. Labeled transition system at the local level

$$(LR1) \quad \langle \mathrm{tell}(c).A, d\rangle \longrightarrow_1 \langle A, d \sqcup c\rangle$$

$$(LR2) \quad \left\langle \sum_{i=1}^{k} q_i : \mathrm{ask}(c_i).A_i, d \right\rangle \longrightarrow_{\tilde{q}_j} \langle A_j, d\rangle \qquad \text{if } d \vdash c_j$$

$$(LR3) \quad \frac{\langle A, d\rangle \longrightarrow_p \langle A', d'\rangle}{\left\langle q_1 : A \mid_{i=2}^{k} q_i : B_i, d\right\rangle \longrightarrow_{p\cdot\tilde{q}_1} \left\langle q_1 : A' \mid_{i=2}^{k} q_i : B_i, d'\right\rangle}$$

$$(LR4) \quad \frac{\langle A, d \sqcup \exists_x c\rangle \longrightarrow_p \langle B, d'\rangle}{\left\langle \exists_x^d A, c\right\rangle \longrightarrow_p \left\langle \exists_x^d B, c \sqcup \exists_x^d \right\rangle}$$

$$(LR5) \quad \langle p(\overrightarrow{y}), c\rangle \longrightarrow_1 \left\langle \Delta_{\overrightarrow{x}}^{\overrightarrow{y}} A, c\right\rangle \qquad \text{if } p(\overrightarrow{x}) :- A \in Decl$$

$$(LR6) \quad \left\langle \mathrm{new}(\underline{\beta}).A, d\right\rangle \longrightarrow_1 \left\langle A[\beta_{new}/\underline{\beta}], d\right\rangle, \qquad \text{where } \beta_{new} \text{ is fresh.}$$

Rule (**GR2**) gives the semantics of the in and out instructions devoted to send constraints abstractions among the network. This communication is synchronous, therefore we need a process at one node, i say, willing to communicate a constraint (abstraction) $\lambda\overrightarrow{x}c$ over a channel α, and an agent at another node, say j, ready to receive a communication along the *same* channel. In addition, the *length* of the template vector $\overrightarrow{x}$ must coincide with the length of the vector $\overrightarrow{y}$ specified by the receiver. If these conditions are satisfied, then the communication is performed and the constraint store at node j is updated by adding to it $(\Pi_{\overrightarrow{x}}c)[\overrightarrow{y}/\overrightarrow{x}]$. The probability associated to this transition is the product of the (normalized) probability of choosing node i multiplied by the normalized probability that the transmission along channel α reaches node j, and not another node with an active process waiting for a communication along α, with a matching template. This is the probability assigned to the channel by the environment ϱ_i. Observe that the premise of rule (**GR2**) asks explicitly for $\varrho_i(\alpha)(s_j) > 0$, so transmissions cannot be directed towards nodes s_k with $\varrho_i(\alpha)(s_k) = 0$. In addition, the above probabilities are also multiplied by $\tilde{q}_1$, which is the local parallel probability of the in agent, normalized between the other locally parallel agents capable of receiving the same transmission along α.

Communication of channel names is performed synchronously by the $\mathrm{out}_{\mathrm{loc}}(\beta)@\alpha$ and $\mathrm{in}_{\mathrm{loc}}(\underline{\beta})@\alpha$ instructions, which semantics is presented in rule (**GR3**). When the transmission happens, the sent channel name is received and bound to the the channel variable specified in the $\mathrm{in}_{\mathrm{loc}}$ instruction. The probability and preconditions associated to this transition are the same as in rule (**GR2**).

The last objects that can be moved along the network are agent abstractions. In addition to sending agents, we can also transmit the current status of some of their free variables. In particular, if $\overrightarrow{x}$ is the vector of template variables of an agent abstraction

Table 4. Labeled transition system at the network level

$$(GR1)\quad \frac{\langle A,d\rangle \longrightarrow_p \langle A',d'\rangle}{{}^{s_i}_{p_i}\langle A,d\rangle_{\varrho_i} \parallel N \;\Longrightarrow_{p\cdot\tilde{p}_i}\; {}^{s_i}_{p_i}\langle A',d'\rangle_{\varrho_i} \parallel N}$$

$$(GR2)\quad \frac{|\overrightarrow{x}| = |\overrightarrow{y}| \text{ and } \varrho_i(\alpha)(s_j) > 0}{\begin{array}{c}{}^{s_i}_{p_i}\langle \mathrm{out_c}(\lambda\overrightarrow{x}c)@\alpha.A_i, d_i\rangle_{\varrho_i} \parallel {}^{s_j}_{p_j}\langle q_1 : \mathrm{in_c}(\overrightarrow{y})@\alpha.A_j \mid A'_j, d_j\rangle_{\varrho_j} \parallel N \;\Longrightarrow_{\tilde{q}_1\tilde{p}_i\cdot\tilde{\varrho}_i(\alpha)(s_j)} \\ {}^{s_i}_{p_i}\langle A_i, d_i\rangle_{\varrho_i} \parallel {}^{s_j}_{p_j}\langle q_1 : A_j \mid A'_j, d_j \sqcup ((\Pi_{\overrightarrow{x}}c)[\overrightarrow{y}/\overrightarrow{x}])\rangle_{\varrho_j} \parallel N\end{array}}$$

$$(GR3)\quad \frac{\varrho_i(\alpha)(s_j) > 0}{\begin{array}{c}{}^{s_i}_{p_i}\langle \mathrm{out_{loc}}(\beta)@\alpha.A_i, d_i\rangle_{\varrho_i} \parallel {}^{s_j}_{p_j}\langle q_1 : \mathrm{in_{loc}}(\underline{\beta})@\alpha.A_j \mid A'_j, d_j\rangle_{\varrho_j} \parallel N \;\Longrightarrow_{\tilde{q}_1\tilde{p}_i\cdot\tilde{\varrho}_i(\alpha)(s_j)} \\ {}^{s_i}_{p_i}\langle A_i, d_i\rangle_{\varrho_i} \parallel {}^{s_j}_{p_j}\langle q_1 : A_j[\beta/\underline{\beta}] \mid A'_j, d_j\rangle_{\varrho_j} \parallel N\end{array}}$$

$$(GR4)\quad \frac{|\overrightarrow{x}| = |\overrightarrow{y}| \text{ and } \varrho_i(\alpha)(s_j) > 0}{\begin{array}{c}{}^{s_i}_{p_i}\langle \mathrm{out_A}(\lambda\overrightarrow{x}A, \overrightarrow{x_0})@\alpha.A_i, d_i\rangle_{\varrho_i} \parallel {}^{s_j}_{p_j}\langle q_1 : \mathrm{in_A}(\overrightarrow{y})@\alpha.A_j \mid A'_j, d_j\rangle_{\varrho_j} \parallel N \;\Longrightarrow_{\tilde{q}_1\tilde{p}_i\cdot\tilde{\varrho}_i(\alpha)(s_j)} \\ {}^{s_i}_{p_i}\langle A_i, d_i\rangle_{\varrho_i} \parallel {}^{s_j}_{p_j}\left\langle \tfrac{1}{2}q_1 : A_j \mid \tfrac{1}{2}q_1 : (\Pi_{\overrightarrow{x}}A)[\overrightarrow{y}/\overrightarrow{x}] \mid A'_j, d_j \sqcup \left((\Pi_{\overrightarrow{x_0}}d_i)[\overrightarrow{y_0}/\overrightarrow{x_0}]\right)\right\rangle_{\varrho_j} \parallel N \\ \text{where } \overrightarrow{y_0} \subseteq \overrightarrow{y} \text{ correspond to } \overrightarrow{x_0} \subseteq \overrightarrow{x}\end{array}}$$

$\lambda\overrightarrow{x}A$, then we can choose a subsequence $\overrightarrow{x_0} \subseteq \overrightarrow{x}$ and carry the information about $\overrightarrow{x_0}$ together with the agent. To move an agent from node i to node j, we need to specify which variables $\overrightarrow{y}$ of the constraint store of node j will be substituted to the template variables $\overrightarrow{x}$. Note that the subsequence $\overrightarrow{x_0}$ of $\overrightarrow{x}$ determines a subsequence $\overrightarrow{y_0}$ of $\overrightarrow{y}$, where the information carried about $\overrightarrow{x_0}$ will be stored. Rule (**GR4**) specifies the transition associated with agent migration. Communication can be performed between two active $\mathrm{out_A}$ and $\mathrm{in_A}$ instructions, and is realized by adding in parallel at the receiving node the agent $(\Pi_{\overrightarrow{x}}A)[\overrightarrow{y}/\overrightarrow{x}]$. The information about $\overrightarrow{x_0}$ is added by posting to the constraint store of the receiving node the constraint $(\Pi_{\overrightarrow{x_0}}d_i)[\overrightarrow{y_0}/\overrightarrow{x_0}]$. The preconditions and the probability associated to the transition are the same as (**GR2**) and (**GR3**). While putting the agent $\lambda\overrightarrow{x}A$ in parallel with other agents, we have to assign to it the probability for the local parallel operator. This is done by splitting in two the probability of the agent that performed the $\mathrm{in_A}$ instruction. In rule (**GR4**) this is realized by giving probability 0.5 to this agent and probability 0.5 to the newly added one. Compositionality of local parallelism and the automatic renormalization induced by the congruence relation assure that we end up with a coherent probability distribution in node j.

Computational Paths and Observables. A configuration of the network is a point in the space $\mathcal{P}_1 \times \ldots \times \mathcal{P}_m \times \mathcal{C}_1 \times \ldots \times \mathcal{C}_m$, and we indicate it as $(\overrightarrow{P}, \overrightarrow{c})$. The SOS defines a labeled transition relation between configurations of the network, where $(\overrightarrow{P_1}, \overrightarrow{c_1}) \Longrightarrow_p (\overrightarrow{P_2}, \overrightarrow{c_2})$ means that we can go from $(\overrightarrow{P_1}, \overrightarrow{c_1})$ to $(\overrightarrow{P_2}, \overrightarrow{c_2})$ in one step with probability p. Given a path $(\overrightarrow{P_1}, \overrightarrow{c_1}) \Longrightarrow_{p_1} (\overrightarrow{P_2}, \overrightarrow{c_2}) \Longrightarrow_{p_2} \ldots \Longrightarrow_{p_n} (\overrightarrow{P_{n+1}}, \overrightarrow{c_{n+1}})$, its probability p is the product of the probabilities of the single transitions, $p = \prod_{i=1}^{n} p_i$.

The $*$-closure of this relation is defined in the usual way: $(\overrightarrow{P_a}, \overrightarrow{c_a}) \Longrightarrow_p^* (\overrightarrow{P_b}, \overrightarrow{c_b})$ means that we can reach configuration $(\overrightarrow{P_b}, \overrightarrow{c_b})$ from $(\overrightarrow{P_a}, \overrightarrow{c_a})$ in one or more steps. However, particular care must be put in defining the probability p associated to it. In fact, in general we can have more than one path from $(\overrightarrow{P_a}, \overrightarrow{c_a})$ to $(\overrightarrow{P_b}, \overrightarrow{c_b})$, each one with its own probability. Therefore, the probability p will be the sum of the probabilities of all paths leading from $(\overrightarrow{P_a}, \overrightarrow{c_a})$ to $(\overrightarrow{P_b}, \overrightarrow{c_b})$.

A computation of the system is successful if it reaches a state where the vector of agents to be executed is $\overrightarrow{\mathbf{0}} = (\mathbf{0}, \ldots, \mathbf{0})$, i.e. when computations in every node have stopped correctly.

The observables we define hereafter correspond to the input / output behaviour of the system. Given the vector of declarations $\overrightarrow{D}$, the observable $\mathcal{O}_{\overrightarrow{D}}((\overrightarrow{A}, \overrightarrow{c}))$ for an initial configuration $(\overrightarrow{A}, \overrightarrow{c})$ is a probability distribution over $\mathcal{C}_1 \times \ldots \times \mathcal{C}_m$, defined as

$$\mathcal{O}_{\overrightarrow{D}}((\overrightarrow{A}, \overrightarrow{c})) = \left\{ (\overrightarrow{d}, p) \mid (\overrightarrow{A}, \overrightarrow{c}) \Longrightarrow_p^* (\overrightarrow{\mathbf{0}}, \overrightarrow{d}) \right\}.$$

For finite computations this notion of an observable is quite straight forward and results in a probability distribution over all possible outcomes. In the case of infinite, i.e. non-terminating, computations some probabilities could be "lost" which leads to observables corresponding so-called sub-probability distributions over the finite outcomes. One could also define a more complicated measure theoretic structure on the set of all computational paths based on so-called cylindric sets, see e.g. [19], but for the sake of simplicity we will, for the time being, avoid a further investigation of these possibilities.

6 Linking Variables

Each node in the network possesses its own constraint store, with an independent variable set. Therefore, the updating of information in each constraint store evolves independently, a part from the fact that constraint abstractions can be actively communicated between agents. But sometimes a more efficient and pervasive way of transmitting information may be necessary. In particular, we may want to link variables $\overrightarrow{x}$ and $\overrightarrow{y}$ belonging to different constraint stores, in such a way that whenever a variable $x_k \in \overrightarrow{x}$ is updated (its domain is restricted by the presence of a new constraint in the constraint store), then the corresponding variable $y_i \in \overrightarrow{y}$ is updated with the same information. Note that this linking of variables is unidirectional, i.e. the information flows from $\overrightarrow{x}$ to $\overrightarrow{y}$, and therefore constraints can be posted on $\overrightarrow{y}$ without affecting $\overrightarrow{x}$.

This mechanism, which is already present in [9], seems unavoidable if one wants to synchronize the information present in different constraint stores, and enhance the features of the language. However, we proceed in a different way than Réty. In fact, in [9] variables are linked by means of an auxiliary agent that is put in parallel with the existing ones. However, this approach has some drawbacks: the link agent can generate infinite spurious computations, by broadcasting the same constraint forever, and moreover there is an uncontrollable delay between the update of linking variables and the update of linked variables. While the first problem can be somewhat limited by asking for weak fairness, the second seems much more troublesome, especially if one want to reason about the effects of outdated information in computations.

To circumvent this problems, we lift the information about linking from the agent level to the configuration of the network. In detail, we define the set $\mathbb{L} = \{(\overrightarrow{x}, \overrightarrow{y}, i, j) \mid \overrightarrow{x} \subset Var_i, \overrightarrow{y} \subset Var_j, |\overrightarrow{x}| = |\overrightarrow{y}|\}$, which contains all possible couples of variables that can be linked together. In particular, if $(\overrightarrow{x}, \overrightarrow{y}, i, j) \in \mathbb{L}$, then the flow of information goes from $\overrightarrow{x}$ to $\overrightarrow{y}$. Then, at each configuration of the network $\mathcal{P}_1 \times \ldots \times \mathcal{P}_m \times \mathcal{C}_1 \times \ldots \times \mathcal{C}_m$ we associate a subset $\mathbf{L} \subset \mathbb{L}$. Therefore, a network configuration is now expressed by an element of the set $\mathcal{P}_1 \times \ldots \times \mathcal{P}_m \times \mathcal{C}_1 \times \ldots \times \mathcal{C}_m \times \wp(\mathbb{L})$. The set $\mathbf{L}$ contains all the couple of currently linked variables, and can change dynamically during the execution of a program, both by adding or removing elements from it.

In Table 5 we present both the syntax and the rules for dealing with the linking mechanism. The syntax of the language is extended by three instructions, which cover the possibility of adding or removing dynamically some links during the execution. The creation of a link between variables involves two agents located at two different nodes, one of them publishing some of its variables, which are going to receive information, and the other linking some of its variables with the published ones. This bidirectional communication is implemented via the instructions $\mathrm{in}_{\mathrm{link}}(\overrightarrow{x})@\alpha$ and $\mathrm{out}_{\mathrm{link}}(\overrightarrow{y})@\alpha$, and their functioning mechanism is depicted in rule (**GR5**). This rule works similarly to (**GR2**) and (**GR3**) for what regards the preconditions and the probability of transitions, while the establishment of a link is modeled by adding the tuple $(\overrightarrow{x}, \overrightarrow{y}, i, j)$ to $\mathbb{L}$.

Links can also be removed by the instruction $\mathrm{remove}_{\mathrm{link}}(\overrightarrow{y})$. This instruction can be called by an agent which has previously published (one or more times) the variables $\overrightarrow{y}$. Its effect is described in rule (**GR6**): it removes all links publishing information to all variables $\overrightarrow{y}$. Note that subvectors of $\overrightarrow{y}$ can still remain linked.

Table 5. Syntax and rules for linking variables

$$A = \mathrm{in}_{\mathrm{link}}(\overrightarrow{x})@\alpha \mid \mathrm{out}_{\mathrm{link}}(\overrightarrow{y})@\alpha \mid \mathrm{remove}_{\mathrm{link}}(\overrightarrow{y})$$

$$(GR5)\quad \frac{\varrho_i(\alpha)(s_j) > 0,\ |\overrightarrow{x}| = |\overrightarrow{y}|}{\left(\begin{smallmatrix} s_i \\ p_i \end{smallmatrix} \langle \mathrm{out}_{\mathrm{link}}(\overrightarrow{y})@\alpha.A_i, d_i\rangle_{\varrho_i} \parallel \begin{smallmatrix} s_j \\ p_j \end{smallmatrix} \langle q_1 : \mathrm{in}_{\mathrm{link}}(\overrightarrow{y})@\alpha.A_j \mid A'_j, d_j\rangle_{\varrho_j} \parallel N, \mathbb{L}\right) \Longrightarrow_{\tilde{q}_1 \tilde{p}_i \cdot \tilde{\varrho}_i(\alpha)(s_j)} \left(\begin{smallmatrix} s_i \\ p_i \end{smallmatrix} \langle A_i, d_i\rangle_{\varrho_i} \parallel \begin{smallmatrix} s_j \\ p_j \end{smallmatrix} \langle q_1 : A_j \mid A'_j, d_j\rangle_{\varrho_j} \parallel N, \mathbb{L} \cup \{(\overrightarrow{x}, \overrightarrow{y}, i, j)\}\right)}$$

$$(GR6)\quad \left(\begin{smallmatrix} s_i \\ p_i \end{smallmatrix} \langle \mathrm{remove}_{\mathrm{link}}(\overrightarrow{y}).A_i, d_i\rangle_{\varrho_i} \parallel N, \mathbb{L}\right) \Longrightarrow_{\tilde{p}_i} \left(\begin{smallmatrix} s_i \\ p_i \end{smallmatrix} \langle A_i, d_i\rangle_{\varrho_i} \parallel N, \mathbb{L} \setminus \{(\overrightarrow{x}, \overrightarrow{y}, j, i) \mid \overrightarrow{x} \subset Var_j\}\right)$$

$$(GR7)\quad \frac{\begin{smallmatrix} s_i \\ p_i \end{smallmatrix} \langle A_i, d_i\rangle_{\varrho_i} \parallel N \Longrightarrow_p \begin{smallmatrix} s_i \\ p_i \end{smallmatrix} \langle A'_i, d'_i\rangle_{\varrho_i} \parallel N, \text{ and } d_i \neq d'_i}{\left(s_i ::^{p_i}_{\varrho_i} \langle A_i, d_i\rangle \parallel_{j=1,\ldots,m\ j\neq i} \begin{smallmatrix} s_j \\ p_j \end{smallmatrix} \langle A_j, d_j\rangle_{\varrho_j}, \mathbb{L}\right) \Longrightarrow_p \left(\begin{smallmatrix} s_i \\ p_i \end{smallmatrix} \langle A'_i, d'_i\rangle_{\varrho_i} \parallel^m_{j=1, j\neq i} \begin{smallmatrix} s_j \\ p_j \end{smallmatrix} \left\langle A_j, d_j \sqcup \left(\bigsqcup_{\overrightarrow{x}, \overrightarrow{y} : (\overrightarrow{x}, \overrightarrow{y}, i, j) \in \mathbb{L}} (\Pi_{\overrightarrow{x}} d'_i[\overrightarrow{y}/\overrightarrow{x}])\right)\right\rangle_{\varrho_j}, \mathbb{L}\right)}$$

The last rule of Table 5 deal with the actual transmission of information between linked variables. It states that, if the network, without any linking, can perform a transition with a probability p, and this transition modifies the content of the constraint store at a node (say i), then the linked network will automatically broadcast the information to all variables linked to some variable of node i. This transmission is an high level activity of the network, and it requires a global knowledge of the configuration of the network. In addition, it is performed simultaneously with the local evolution of node i. This can be justified by thinking that the broadcasting action is performed by the global scheduler before letting any other node evolve. Otherwise we can imagine that the linked variables represent some form of shared memory, in such a way that one process can write and another process can read it. Anyway, it would probably be better to allow some form of controlled delay for this transmission. In this way, we can model in a more physically sound way the flow of information among the network: the more two nodes are far away, the longer the delay in receiving the information. Introducing delays, however, requires a modification of the semantics of the system. In particular, we must modify the concept of configuration: storing the actual state of the computation is no more sufficient, as now the system must have a form of (limited) memory of its past history. Therefore, one needs to remember, for instance, the last n states of the system, if $n-1$ is the maximum delay occurring in a linking action. Note that the notions of computational path and input / output observables of the system can be extended in a straightforward way to include linking instructions.

7 Conclusions and Future Work

In this paper we extended the CCP computational paradigm with both distributed and probabilistic features. The resulting language is composed by two structural levels: one

local and one global. The local entities are nodes, possessing their own constrain store, where the computation evolves according to a probabilistic version of CCP rules [5]. At this local level time is discrete and communications are asynchronous. These nodes are then connected in a global network, and they can exchange information through communication channels. Communication at this level is synchronous and performed in a π-calculus style. Three different objects can be exchanged in the network: constraint abstractions, channels and agent abstractions. Moreover, also the evolution of the network proceeds following probabilistic rules, and the time is discrete also at this level. Finally, the topology of the network is extended with a mechanism for remote synchronization of variables belonging to different constraint stores.

This language can be used for modelling distributed optimization algorithm, like simulated annealing, genetic algorithms and similar ones. Its declarative nature makes these programs very simple to write, while its stochastic nature makes possible to derive some properties of the optimization process just by looking at its semantics (or, more realistically, at some suitable abstraction of it). We have also wrote a metainterpreter in prolog for a subset of the language, and we are extending it to the full featured one.

In the future, we plan to extend the language by adding more features at the network level. In particular, we want to introduce networks of dynamic size, where the scheduling probabilities can change during time. In addition, we plan to develop also a continuous time version of the language, where scheduling probabilities among nodes are substituted by exponential rates. Finally, we want to provide the language with a denotational semantic, following the approach of [4], ad then use probabilistic abstraction techniques [6] to perform some kind of static analysis [7].

References

1. F.S. de Boer, R.M. van Eijk, W. van der Hoek, and J-J.Ch. Meyer. Failure semantics for the exchange of information in multi-agent systems. In *Proceedings of CONCUR 2000*, 1998.
2. F.S. de Boer, A. Di Pierro, and C. Palamidessi. Nondeterminism and infinite computations in constraint programming. *Theoretical Computer Science*, 151(1), 1995.
3. R. De Nicola, G. Ferrari, and R. Pugliese. Klaim: a kernel language for agents interaction and mobility. *IEEE Transactions on Software Engineering*, 24(5):315–330, 1998.
4. A. Di Pierro and H. Wiklicky. A banach space based semantics for probabilistic concurrent constraint programming. In *Proceedings of CATS'98*, 1998.
5. A. Di Pierro and H. Wiklicky. An operational semantics for probabilistic concurrent constraint programming. In *Proceedings of IEEE Computer Society International Conference on Computer Languages*, 1998.
6. A. Di Pierro and H. Wiklicky. Probabilistic abstract interpretation and statistical testing. In H. Hermanns and R. Segala, editors, *Lecture Notes in Computer Science 2399*. Springer Verlag, 2002.
7. A. Di Pierro, H. Wiklicky, and C. Hankin. Quantitative static analysis of distributed systems. *Journal of Functional Programming*, To appear.
8. D. Gilbert and C. Palamidessi. Concurrent constraint programming with process mobility. In *Proceedings of CL 2000*, 2000.
9. Rety. J. H. Distributed concurrent constraint programming. *Fundamentae Informaticae*, 34(3):323–346, 1998.
10. L. Henkin, J.D. Monk, and A. Tarski. *Cylindric Algebras, Part I*. North-Holland, Amsterdam, 1971.

11. O. M. Herescu and C. Palamidessi. Probabilistic asynchronous π-calculus. In J. Tiuryn, editor, *Proceedings of FOSSACS 2000*, Lecture Notes in Computer Science, pages 146–160. Springer Verlag, 2000.
12. Brim L., Gilbert D., Jacquet J., and Kretinsky M. Multi-agent systems as concurrent constraint processes. In *Proceedings of SOFSEM 2001*, 2001.
13. M. Milano and A. Roli. Magma: A multiagent architecture for metaheuristics. *IEEE Trans. on Systems, Man and Cybernetics - Part B*, 34(2), 2004.
14. R. Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Springer Verlag, 1994.
15. C. Priami. Stochastic π-calculus. *Computer Journal*, 38(7):578–589, 1995.
16. M. Resende, P. Pardalos, and S. Duni Ekşioğlu. Parallel metaheuristics for combinatorial optimization. In R. Correa et al., editors, *Models for Parallel and Distributed Computation - Theory, Algorithmic Techniques and Applications*, pages 179–206. Kluwer Academic, 2002.
17. V. A. Saraswat. *Concurrent Constraint Programming*. MIT press, 1993.
18. V. A. Saraswat, M. Rinard, and P. Panangaden. Semantics foundations of concurrent constraint programming. In *Proceedings of POPL*, 1991.
19. Paul C. Shields. *The Ergodic Theory of Discrete Sample Paths*, volume 13 of *Graduate Studies in Mathematics*. American Mathematical Society, Providence, Rhode Island, 1996.
20. P. Van Roy, S. Haridi, P. Brand, G. Smolka, M. Mehl, and R. Scheidhauer. Mobile objects in distributed oz. *ACM Transactions on Programming Languages and Systems*, 19(5):804–851, 1997.

HYPROLOG: A New Logic Programming Language with Assumptions and Abduction

Henning Christiansen[1] and Veronica Dahl[2]

[1] Roskilde University, Computer Science Dept., Roskilde, Denmark
[2] Dept. of Computer Science, Simon Fraser University Burnaby, B.C., Canada
henning@ruc.dk, veronica@cs.sfu.ca

Abstract. We present HYPROLOG, a novel integration of Prolog with assumptions and abduction which is implemented in and partly borrows syntax from Constraint Handling Rules (CHR) for integrity constraints. Assumptions are a mechanism inspired by linear logic and taken over from Assumption Grammars. The language shows a novel flexibility in the interaction between the different paradigms, including all additional built-in predicates and constraints solvers that may be available. Assumptions and abduction are especially useful for language processing, and we can show how HYPROLOG works seamlessly together with the grammar notation provided by the underlying Prolog system. An operational semantics is given which complies with standard declarative semantics for the "pure" sublanguages, while for the full HYPROLOG language, it must be taken as definition. The implementation is straightforward and seems to provide for abduction, the most efficient of known implementations; the price, however, is a limited use of negations. The main difference wrt. previous implementations of abduction is that we avoid any level of metainterpretation by having Prolog execute the deductive steps directly and by treating abducibles (and assumptions as well) as CHR constraints.

1 Introduction

Assumption-based reasoning in general, or hypothetical reasoning is defined in [22] as a logic system in which a set of facts and a set of possible hypotheses are given. Its instances can be assumed if they are consistent with the facts. Both abduction (the unsound but useful assumption of B given A and given that B implies A) and linear and intuitionistic logic inspired assumptions (special facts that are made available as global resources within a specific scope [25]) fall into that general category. Their formalization within, respectively, Abductive Logic Programming [19] and Assumptive Logic Programming [14] refines this general notion by for instance requiring in the first case consistency with a special type of facts: integrity constraints. Both allow us to move beyond the limits of classical logic to explore "possible cause" and "what-if" scenarios. They have proved useful for diagnosis, recognition, sophisticated human language processing problems, and many other applications. However in practice, abduction in particular

M. Gabbrielli and G. Gupta (Eds.): ICLP 2005, LNCS 3668, pp. 159–173, 2005.

has not been used to its full potential owing to implementation indirections. Assumptions can be more efficiently implemented through continuation based processors such as BinProlog, but there is no Prolog in existence which efficiently provides both capabilities at the same time.

The present paper generalizes and improves an earlier proposal presented in the workshop paper [12]. In this article we present a new programming language, HYPROLOG, which augments Prolog with the following hypothetical reasoning capabilities:

- linear, intuitionistic and timeless assumption, in the sense of [14] to which we add the new feature of integrity constraints,
- abduction in the sense of abductive logic programming [19],
- integrity constraints that may refer to both abducibles and assumptions.

These results are significant in that they enhance Prolog's appeal as a programming language by transporting it beyond the rigid limits of classical logic and thus making it more appropriate for human-like reasoning in general and for AI in particular. In addition, they are *portable*, in the sense that programs in any Prolog that includes CHR can simply be augmented with our code, which provides the mentioned extensions, and *efficient*: the assumptive part runs only three times slower than in Prolog versions where assumptions are hardwired, and our abduction runs actually faster than previous implementations for programs that involve many resolution steps. Finally, our implementation principles can be seen as demonstration of how hypothetical reasoning can materialize in CHR even without our system.

We first overview the necessary background on abduction, assumptions, and CHR; then we present the new language's syntax and exemplify its use within both programs and grammars. We discuss implementation principles, semantic considerations, related work and benchmarks, and finally, we provide concluding remarks. System and sample programs are available at
http://www.ruc.dk/~henning/hyprolog.

2 Background

2.1 Abduction

An abductive logic program [19] is usually specified as a triplet $\langle \mathcal{P}, \mathcal{A}, \mathcal{IC} \rangle$ where $\mathcal{P}$ is a logic program, $\mathcal{A}$ a set of *abducible* predicates that do not occur in the head of any clause of $\mathcal{P}$, and $\mathcal{IC}$ a set of integrity constraints assumed to be consistent. Assume additionally that $\mathcal{P}$ and $\mathcal{IC}$ can refer to a set of *built-in* predicates that have a fixed meaning identified as a theory $\mathcal{B}$; a predicate in $\mathcal{P}$ that is neither abducible nor built-in is called *defined*. We assume for simplicity in the following that $\mathcal{IC}$ refers to abducible and built-in predicates only.

Given an abductive logic program $\langle \mathcal{P}, \mathcal{A}, \mathcal{IC} \rangle$, we define for pairs of sets of abducibles and built-in atoms $\langle A, B \rangle$, a *consistent ground instance* to be a common ground instance $\langle A', B' \rangle$ of $\langle A, B \rangle$ so that

– $\mathcal{B} \models B'$ (the instance of built-ins is satisfied)
– $\mathcal{B} \cup A' \models \mathcal{IC}$ (the instance of abducibles respects the integrity constraints)

For simplicity and without loss of generality, we consider only ground queries; an *abductive answer* to a query Q is a pair of finite sets of abducible and of built-in atoms $\langle A, B \rangle$ such that

– $\langle A, B \rangle$ has at least one consistent ground instance $\langle A', B' \rangle$,
– for any such $\langle A', B' \rangle$, we have $\mathcal{P} \cup A' \models Q$.

Minimality and Compaction. It is often required that an abductive answer be minimal measured in the number of abduced literals (or, alternatively, in a subset relation or subsumption ordering). Most published abduction algorithms try to unify a new abducible with one already produced (as to produce answers of a minimum number of literals), and tries out different alternatives under backtracking. This does not guarantee minimality in cases when, say, a proof needs abducibles a and b but another may need only a. Minimal answers can be selected by post-processing all answers found in this way. However, we argue that this principle which we call *compaction* is not always obvious or desirable, and we suggest it be optionally specified for selected abducible predicates. (If, for example, someone's car was stolen in Paris and his wallet in New York, it seems over-constrained to assume by default that the thieves are the same one.)

2.2 Assumptive Logic Programming

Assumptive logic programs [14] are logic programs augmented with a) linear, intuitionistic and timeless implications scoped over the current continuation, and b) implicit multiple accumulators, useful in particular to make the input and output strings invisible when a program describes a grammar (in which case we talk of Assumption Grammars [15]). More precisely, we use the kind of linear implications called *affine* implications, in which assumptions can be consumed at most once, rather than exactly once as in linear logic. Although intuitively easy to grasp and to use, the formal semantics of assumptions is relatively complicated, basically proof theoretic and based on linear logic [14,15,25]. Here we use a more recent and homogeneous syntax for assumptions introduced in [10]; we do not consider accumulators, and we note that Assumption Grammars can be obtained by applying the operators below within a DCG.

`+h(a)`	Assert linear assumption for subsequent proof steps. Linear means "can be used once".
`*h(a)`	Assert intuitionistic assumption for subsequent proof steps. Intuitionistic means "can be used any number of times".
`-h(X)`	Expectation: consume/apply existing int. assumption.
`=+h(a)`, `=*h(X)`, `=-h(X)`	Timeless versions of the above, meaning that order of assertion of assumptions and their application or consumption can be arbitrary.

A sequential expectation cannot be met by timeless assumption and vice versa, even when they carry same name. In [15], a query cannot succeed with a state which contains an unsatisfied expectation; for simplicity (and to comply with our implementation), this is not enforced in HYPROLOG but can be tested explicitly using a primitive called `expections_satisfied`. Assumption grammars have been used for natural language problems such as free word order, anaphora, coordination, and for knowledge based systems and internet applications. In the earlier work on Assumptions, only a semiformal semantics was given, and the semantics we show below are intended to make its principles precise.

2.3 Constraint Handling Rules, CHR

CHR [17] is a declarative, rule-based language for writing constraint solvers and is now included as an extension of several versions of Prolog. Operationally and implementation-wise, CHR extends Prolog with a constraint store, and the rules of a CHR program serve as rewriting rules over constraint stores. CHR is declarative in the sense that its rules can be understood as logical formulas. Constraint predicates must be declared as such and can then be called from a Prolog program; see [17] for details. The following example declares a constraint predicate `a` and defines a so-called propagation rule.

```
constraints a/1.
a(1), a(2) ==> fail.
```

This rule identifies a state as illegal if it contains the two indicated constraints. As first noticed by [3], there is a clear analogy between abducibles plus integrity constraints and CHR's constraints plus rules.

3 HYPROLOG, Syntax and Informal Semantics

3.1 Basic HYPROLOG

A HYPROLOG program is written as a Prolog program with additional declarations of assumptive and abductive predicates, the latter possibly with compaction. Notation for applying assumptions is shown in the previous section. Integrity constraints are written as any sort of CHR rules with abducibles and assumptions in the head. The following exemplifies such declarations.

```
abducibles a/1, b/2.
compaction a/1.
assumptions c/1.
timeless_assumption d/2.
```

The first declaration introduces abducible predicates `a/1`, `b/2` as well as `a_/1`, `b_/2` that represent their negation; compaction is defined for `a/1` (as described above). The declaration of `c/1` makes available assumptions and expectations of forms `'+c'/1`, `'-c'/1`, `'*c'/1` (the system reads, say, `+c(5)` as `'+c'(5)`).

3.2 HYPROLOG's Grammatical Counterpart

DCGs [21], included in most Prolog systems and compiled into Prolog when a source file is loaded, are also available in HYPROLOG, adequately augmented with abduction and assumptions as well. The following example has been adapted from [10,15] and shows two applications of assumptions: for resolving pronoun references and for a simple coordination problem. In a sentence *"Peter likes her"* the pronoun is expected to stand for a female character who has been mentioned earlier in the discourse. The following rule defines how the mention of a proper name produces an (intuitionistic) assumption that makes the individual available for future reference, as many times as needed.

```
assumptions acting/1.

np(X,Gender) --> name(X,Gender), {*acting(X,Gender)}
```

Let's suppose we have the following rules for sentences and sequences of sentences.

```
sentence(s(A,V,B)) -->  np(A,_), verb(V), np(B,_).
sentences((S1,S2)) --> sentence(S1),sentences(S2).
sentences(nil) --> [].
```

The following rules define how a pronoun can appear in a sentence with its meaning given by the consumption of an assumption made.

```
np(X,Gender) --> {-acting(X,Gender)}, pronoun(Gender).
pronoun(fem) --> [her].
```

The following query and answers show the grammar's behaviour.

```
?- phrase(sentences(S), [peter,likes,martha, mary,hates,her]).
S = (s(peter,like,martha),s(mary,hate,mary),nil) ? ;
S = (s(peter,like,martha),s(mary,hate,martha),nil) ? ;
no
```

The second answer expresses the interpretation we would expect, and the first one is an undesired consequence of the specification so far; we show below how it can be suppressed.

The discourse *"Peter likes and Mary hates Martha"* contains two coordinating sentences in the sense that the first incomplete one takes its object from the second one. This can be described by having an incomplete sentence put forward a timeless expectation that may be satisfied by a later assumption produced by a complete sentence; the following two grammar rules are sufficient.

```
sentence(s(A,V,B)) --> np(A,_), verb(V), np(B,_), {=*obj(B)}.
sentence(s(A,V,B)) --> np(A,_), verb(V), [and],   {=-obj(B)}.
```

3.3 Mixing Abduction and Assumptions

Abduction and assumptions can be mixed freely, which is handy for a better solution to the pronoun resolution problem above. We modify the grammar rule for sentences such that semantic interpretation is made abductively, i.e., the sentence can be told honestly provided the semantic context contains the necessary facts.

```
abducibles s/3.
sentence --> np(A,_), verb(V), np(B,_), {s(A,V,B)}.
s(X,hate,X) ==> fail.
```

With this modification, the analysis of *"Peter likes Martha, Mary hates her"* gives only one solution. This grammar is interesting as it shows how different layers of analysis can assist each other: semantic knowledge about the hating relation is applied for guiding pronoun resolution. This example illustrates a general approach to discourse analysis called Meaning-in-Context, described in [13].

3.4 Negation

Compared with other abductive systems, the use of negation is quite limited and restricted to a simple form of explicit negation [7]. When an abducible, say `a/1`, is declared, an additional predicate `a_/1` representing ¬`a` is introduced together with an integrity constraint (hidden from the user) `a(X), a_(X) ==> fail.`

Although useful for many applications, this implementation covers only one part of negation: "you cannot have P and $\neg P$ at the same time"; the condition that "either you have P or $\neg P$" cannot be expressed in a straightforward way.

If a program clause includes an application of negation-as-failure that refers to abducibles directly or indirectly, we inherit the dubious semantics of Prolog. So if `a/1` has been declared as abducible, with definition `p(X):-a(X)`, a call `\+p(Z)` (where `Z` is a currently uninstantiated variable) may succeed if the abduction of `a(Z)` triggers a failure producing integrity constraint.

4 Semantic Considerations

Like Prolog, CHR has a declarative semantics plus a procedural one, and for a substantial subset of the language, the two are in agreement. Each rule of a CHR program can be understood as a logical formula:

	Propagation rule	Simplification rule
CHR rule:	H==>$G\|B$	H<=>$G\|B$
Logical meaning:	$\forall\bar{x}\big((\exists\bar{y}G) \rightarrow (H \rightarrow \exists\bar{z}B)\big)$	$\forall\bar{x}\big((\exists\bar{y}G) \rightarrow (H \leftrightarrow \exists\bar{z}B)\big)$

where $\bar{x}$ refers to the variables in H, $\bar{y}$ to those in G not overlapping with $\bar{x}$, and $\bar{z}$ to those in B not overlapping with $\bar{x}$; for simplicity it is assumed that $\bar{y}$ and $\bar{z}$ do not overlap. Ignoring the problems with Prolog's negation as failure, we can say that the meaning (e.g., a model-based semantics) of a program that

combines Prolog and CHR is given by formulas as above plus a reading of the Prolog part as a completed definition.

However, as has been debated recently [16], the statements that can be made by using this semantics for CHR are often too weak to express the (implemented) meaning of even simple and intuitively clear programs. Even the classical, procedural semantics with nondeterminism in selection steps [2] is not sufficient; as noted by [16], even example programs in the reference manual of CHR depend on the implemented semantics, and this motivated [16] to describe a so-called refined procedural semantics.

The part of HYPROLOG without assumptions is an instance of abductive logic programs and conforms with the standard semantics given in section 2.1; this is independent of whether the logically redundant compaction principle is applied. However, as we use CHR for integrity constraints, the discussion above goes for this subset of HYPROLOG as well.

Assumptions, on the other hand, inherit the procedural flavour of linear logic, and a correct semantics for HYPROLOG without abduction, and even without integrity constraints, needs to reflect a left-to right execution of the clause bodies. In other words, the comma cannot be understood as conjunction but as a sequential operator that pushes a perhaps modified state forward.

Interestingly, [5] has proposed recently a semantics of CHR formulated in terms of linear logic, and a very interesting next step could be to generalize this for HYPROLOG. This may perhaps provide a more straightforward characterization of the assumption part of HYPROLOG, as assumptions can be mapped to their natural counterpart in linear logic. This possibility has not been investigated yet.

4.1 A Continuation Semantics for HYPROLOG

Here we specify the CHR engine and its constraint stores as an abstract data type (whose detailed specification can be found in [16]). For simplicity we assume only one built-in which can be used in clause bodies, "=" with the meaning of equality (unification). What may be allowed in bodies of integrity constraints is abstracted away. By a *constraint*, we mean an abducible atom, an atom of a built-in predicate, an assumption, or a timeless expectation; let Con refer to the set of all such. We assume a given HYPROLOG program of clauses $C\ell$ and integrity constraints IC.

Let $Store$ be a sort for all possible constraint stores whose internal structure is not specified and which is equipped with the following operations; Sub refers to the domain of substitutions; mgu is used to denote a most general unifier of two atoms. When relevant, a substitution σ can also be understood as a set of equations $\{x = t \mid x\sigma \text{ evaluates to } t\}$. The following operations are assumed:

- $\in$: $Con \times Store \to \{true, false\}$.
- $\setminus : Con \times Store \to Store$ representing the removal of a constraint from the store.
- $Accommodate : Con^* \times Store \to Store \times Subst$ corresponding to the CHR engine's behaviour given IC when one or more constraints are called from

a clause; whatever recursion takes place inside *Accommodate* is not specified; the output substitution represents possible side-effects that affect the remaining query. The function is partial, undefined meaning failure or loop.
- $\emptyset : Store$ which is the initial store.

Notice that the definitions allow *Accommodate* to take also a substitution as its first argument. This reflects the property that the unification of variables may trigger CHR rules to apply. This abstract data type is assumed to be *sound* in the sense that, whenever $Accommodate(A, S) = \langle S', \sigma \rangle$, we have that $IC \models \forall \bar{x}((A \wedge S) \leftrightarrow \exists \bar{z}(S' \wedge \sigma))$ where here a store is identified with the set of all constraints in it (given by $\in$); $\bar{x}$ are the variables in $A \wedge S$ and $\bar{z}$ any remaining variables in $S' \wedge \sigma$. For simplicity, let us ignore the risk of loops and also claim it *complete* meaning that $Accommodate(A, S)$ is defined whenever $IC \models \exists \bar{x}(A \wedge S)$. In case no integrity constraints are involved, the constraint store serves as a passive container for abducibles and assumptions.

A *query* is a sequence of atoms; ϵ is the empty query; concatenation and construction of sequences are indicated by a dot, and for readability the comma of the clause syntax is taken as dot. A *state* is a pair of a query and store. A *final* state is of the form $\langle \epsilon, S \rangle$. A *derivation* for Q is produced from a finite number of derivation steps (below), starting from $\langle Q, \emptyset \rangle$, and it is *successful* if it ends in a final state. The derivation relation $\rightsquigarrow$ is defined by the following rules.

1. $\langle A \cdot Q, S \rangle \rightsquigarrow \langle (B \cdot Q)\sigma\sigma', S' \rangle$ if there is a program clause which has a variant H `:-` B with fresh variables, $\sigma = mgu(H, A)$ and $Accommodate(\sigma, S) = \langle \sigma', S' \rangle$.
2. $\langle s = t \cdot Q, S \rangle \rightsquigarrow \langle Q\sigma\sigma', S' \rangle$ whenever $mgu(s, t) = \sigma$ and $Accommodate(\sigma, S) = \langle \sigma', S' \rangle$.
3. $\langle Ab \cdot Q, S \rangle \rightsquigarrow \langle Q\sigma\sigma', S' \rangle$ whenever Ab is a compacting abducible and there is some $A \in S$ with $mgu(A, Ab) = \sigma$ and $Accommodate(\sigma, S) = \langle \sigma', S' \rangle$
4. $\langle Ab \cdot Q, S \rangle \rightsquigarrow \langle Q\sigma, S' \rangle$ whenever Ab is an abducible and $Accommodate(Ab, S) = \langle \sigma, S' \rangle$.
5. $\langle$`-`$A \cdot Q, S \rangle \rightsquigarrow \langle Q\sigma\sigma', S' \rangle$ whenever, either
   - there is a `+`$A' \in S$ with $mgu(A, A') = \sigma$ and $Accommodate(\sigma, S \setminus \{$`+`$A'\}) = \langle S', \sigma' \rangle$, or
   - there is a `*`$A' \in S$ with $mgu(A, A') = \sigma$ and $Accommodate(\sigma, S) = \langle S', \sigma' \rangle$.
6. $\langle As \cdot Q, S \rangle \rightsquigarrow \langle Q\sigma, S' \rangle$ where $Accommodate(As, S) = \langle S', \sigma \rangle$, As of form `+`A or `*`A.
7. $\langle$`=-`$A \cdot Q, S \rangle \rightsquigarrow \langle Q\sigma\sigma', S' \rangle$ whenever, either
   - there is an `=+`$A' \in S$ with $mgu(A, A') = \sigma$ and $Accommodate(\sigma, S \setminus \{$`=+`$A'\}) = \langle S', \sigma' \rangle$,
   - there is an `=*`$A' \in S$ with $mgu(A, A') = \sigma$ and $Accommodate(\sigma, S) = \langle S', \sigma' \rangle$, or
   - $Accommodate($`=-`$A, S) = \langle S', \sigma' \rangle$ and $\sigma = \emptyset$.
8. $\langle$`=+`$A \cdot Q, S \rangle \rightsquigarrow \langle Q\sigma\sigma', S' \rangle$ where
   - there is an `=-`$A' \in S$ with $mgu(A, A') = \sigma$ and $Accommodate(\sigma, S \setminus \{$`=-`$A'\}) = \langle S', \sigma' \rangle$, or
   - $Accommodate($`=+`$A, S) = \langle S', \sigma' \rangle$ and $\sigma = \emptyset$.

9. $\langle \texttt{=*}A \cdot Q, S\rangle \rightsquigarrow \langle Q'\sigma\sigma', S'\rangle$ where
   - there is an $\texttt{=-}A' \in S$ with $mgu(A, A') = \sigma$ and $Accommodate(\sigma, S \setminus \{\texttt{=-}A'\}) = \langle S', \sigma'\rangle$, and Q' is one of $\texttt{=*}A \cdot Q$ or Q, or
   - $Accommodate(\texttt{=*}A, S) = \langle S', \sigma'\rangle$, $\sigma = \emptyset$, and $Q' = Q$.

We notice that step 1 defines an operational semantics for the pure Prolog subset of HYPROLOG, 1–2 for pure Prolog with built-in equality, and 1–4 (or, alternatively, 1–2 plus 4) for abductive logic programs (with CHRs as ICs and no interesting negation); these operational semantics are straightforward to prove sound and complete with respect to the respective declarative semantics. Step 1 plus 5–9 provides a formal semantics for Assumptive Logic Programs which has been lacking in earlier references.

In the lack of a truly declarative semantics (first-order or otherwise) for assumptions, we need to take 1–9 as *the* semantic definition for the full HYPROLOG language.

Finally, we notice that the actually implemented HYPROLOG system inherits the detailed procedural semantics of Prolog (trying rules in textual order) and of CHR (cf. the refined semantics [16]), both of which were abstracted away above. Being hosted in a realistic version of Prolog which includes a realistic version of CHR, all other low and high level features of these languages are available, including Prolog's negation as failure (with the usual caveats) and large collections of constraint solvers and built-in predicates.

5 Implementation

Our implementation uses SICStus Prolog [23] and its CHR library; we refer to the proper sections of the referenced manual for a detailed description of the facilities that we use. The principles shown can also be used for implementing various kinds of hypothetical reasoning in Prolog through CHR.

5.1 Implementing Abduction

The implementation in Prolog with CHR is simple: abducibles are viewed as constraints in the sense of CHR: the logic program is executed by the Prolog system; whenever an abducible is called it is added automatically by CHR to the constraint store and CHR will activate integrity constraints whenever relevant. The complete hand-coded implementation of an abducible predicate `a/1` is provided by the following three lines.

```
:- use_module(library(chr)).
handler abduction.
constraints a/1.
```

Compaction for `a/1` is implemented by a single CHR rule; the following provides a correct implementation.

```
a(X), b(Y) ==> true | (X=Y ; dif(X,Y)).
```

(The implementation of HYPROLOG applies a slightly optimized version using low-level facilities of CHR.) When a HYPROLOG program is read from file, declarations as shown in section 3.1 are translated into CHR as shown here.

The correctness is inherited from the correctness properties of the underlying Prolog plus CHR systems. For any program without occurs-check problems, the implementation produces correct abductive answers as defined above; if the program (including integrity constraints) does not loop, we also have that the total set of answers produced is complete.

Notice that the approach can interact with an arbitrary constraint solver by considering its constraints as built-ins (applied in bodies of clauses and integrity constraints). Possible soundness and completeness of such a combination will mirror the properties of the applied constraint solver.

5.2 Implementing Assumptions

Assumptions and expectation operators are implemented in CHR in a way similar to abduction, but need extra care for scoping and matching of expectations with assumptions. Each operator for each declared assumption (say `'-c'/1` is implemented by one single-headed CHR rule that employs the constraint store as container in a straightforward procedural way, optimized using the low-level primitive `findall_constraints` and `remove_constraints`; see the HYPROLOG website for details.

6 Examples and Benchmarks

Suppose we need to schedule the printing jobs of three printers. At any time, the status of each printer is represented by an assumption `+printer(` *name*`,` *ready-time*`)`.

Further, assume that all printers are covered by the same undersized electrical fuse that will melt down in case all three printers are running at the same time. Such situations is prevented by the following integrity constraint; assumptions have been extended with starting time for the most recent job and the guard refers to an auxiliary predicate that holds if and only if all three indicated time intervals have a point in common.

```
+printer(lexon2000,S1,F1), +printer(epsmark1993,S2,F2),
   +printer(pewhack2004,S3,F3) ==>
  overlapping((S1,F1),(S2,F2),(S3,F3))  |  fail.
```

We have compared the efficiency of our first implementation of assumptions [12] with the one hardwired into BinProlog for our HYPROLOG print scheduler program (whose complete version can be found in (website URL) for 10 printers and 50 print jobs. The BinProlog version was about 5 times faster. Our present implementation, with specialized predicates for each type of assumption, gave a speedup of 40 percent, or now only 3 times slower than BinProlog.

For abduction, we have compared our system's performance with that of the A-system [20]. The mentioned reference reports a test of an abductive n-queens program that runs very fast in A-system, considerably faster that in our system. However, an inspection of the example shows that the A-system for this program produces quickly one set of constraints which is then solved by a specialized finite-domain solver. It is difficult to translate this example into our system due to the mentioned limitations for negation, which made the program degenerate into a naive generate-and-test algorithm.

In theory, our approach should be superior for programs that involve many resolution steps, and to verify this, we constructed an example updating a database view involving complex joins. The query (update request) in the test is `w(monkey)` where the view is defined as `w(F):- pp(A,B,C,D), qq(C,D,E,F), rr(A,E,F)`. Each of `pp`, `qq`, `rr` provides a link to either a database predicate or an abducible. Integrity constraints express suitable key constraints and the database predicates contains 100, 99, 99 tuples selected carefully so that an immense collection of combinations needs to be tried out before a solution is found. The example is directly translatable between the two systems, and our program in Prolog plus CHR run through the optimizing SICStus Prolog compiler solves the problem in 3 ms (three) whereas A-system spends 6250 ms for the same job; the tests were performed on a 400 MHz Macintosh G4 Powerbook; the programs are available at the HYPROLOG website.

7 Conclusions and Related Work

We have presented HYPROLOG, a new logic programming language which directly and efficiently integrates abducibles and assumptions into Prolog itself, through simply extending it with a few lines of CHR code.

This provides an optimal combination, in which such programs can be written and executed directly, with only a small extra overhead involved when needed. In contrast, known metainterpreter based implementations of abduction incur heavy computational overhead (for instance, [18,20] has the overhead of alternating abductive steps with resolution steps, the latter also simulated by metainterpretation). An important advantage over other known abduction systems is that the full collection of Prolog's built-in facilities (logical as well as impure) are available, including all available libraries and constraint solvers.

We have also described a methodology for implementing HYPROLOG that obtains an execution speed comparable to that of traditional Prolog programs.

The component of a HYPROLOG program that corresponds to a logic program is executed directly as a Prolog program, and its integrity constraints directly as CHR rules. This means that HYPROLOG programs can be run through existing optimizing compilers for Prolog and CHR. It is interesting to point out that integrity constraints are automatically coroutined by virtue of CHR rules.

There are existing, efficient implementations of Assumptive Logic Programs but the present work extends the paradigm with integrity constraints and the option to combine with abduction in a common framework.

The price paid for this efficiency and flexibility is a limitation on the use of negation. Yet even with this restriction, many useful examples are made possible.

Some examples in the literature of abduction involving Event Calculus do not work in our approach but others, such as [24] on robot planning seem possible (this example has been implemented in CHR in an early experiment, but not tested in the present framework). Experiences with HYPROLOG and earlier experiments with similar techniques in CHR indicate a spectrum of interesting programs, and the fact that the paradigm can immediately be combined with any other constraint solvers available in the Prolog version at hand substantiates this viewpoint.

The first observation of the similarity between CHR and abductive logic programming was made by [3] showing that abducible predicates can be represented as constraints in CHR's sense and integrity constraints as rules in CHR. The referenced work describes a translation of a class of abductive logic programs with limited use of negation (similar to the present paper) into $\text{CHR}^{\vee}$ [4] which is an extension to CHR with disjunctions in rule bodies; the main difference is that [3] also translates the logic program component into $\text{CHR}^{\vee}$ so that the efficiency of having Prolog do the resolution steps is lost. CHR based abduction for language processing is applied in the CHRG system [10,9] which is based on bottom-up parsing in CHR.

A proposal for emulating abductive logic programming with assumptions was made in [14]. While less efficient than the present proposal, it allowed the same (abducible) predicate to be either proved normally, if this was possible, or abduced if not. It also put the ability to examine unconsumed assumptions to use in combining for instance defeasible reasoning with abductive logic programming, and in suggesting novel extensions such as conditional abductive logic programming—this latter, by abducing not only predicates, but also clauses.

Abduction by means of CHR has been applied by [11] for natural language grammars with automatic error detection and correction.

As we have noticed, negation is the more complicated part to which we have no solution; [6] sketches an extension of [3]'s method intended for a full use of negation-as-failure in program clauses and integrity constraints; as for [3], no integration with Prolog is provided. Unfortunately, it has not been possible to reconstruct the code from the description in [6] in order to test the method and there appears to be inherent looping problems.

The Demo system described in [8] seems to be the first application of CHR to abduction and similar problems, in the shape of a general metainterpreter for logic programs which is reversible in the sense that it can generate programs to make specified goals provable; this property provided by a constraint solver written in CHR for semantic primitives. In terms of efficiency this system is by no means comparable to what is described in the present paper.

The simultaneous availability of abduction and assumptions facilitates subtler reasoning by making it possible to clearly separate the generation of hypotheses from their confirmation, within a dynamic process where different strategies can be flexibly implemented.

A recent CHR-based system that extends abductive reasoning with the ability to confirm or disconfirm abduced facts (or events, since this system specializes to event-based programming) [1] requires a complex architecture to achieve similar results to ours. This system's hypotheses are, as in our own system, represented by abducibles, but their confirmation or disconfirmation is managed by a specialized proof procedure which can be tuned to be skeptical (i.e, to disconfirm at the end of a computation all hypotheses that remain consistent but have not specifically been confirmed) or credulous (i.e., to confirm all of those).

In HYPROLOG we also represent hypotheses as abduced facts, but their (dis)confirmation proceeds within Prolog's normal proof procedure, and can be done either dynamically, or in a postprocessing stage which can interpret the unconsumed assumptions in a variety of ways, ranging from credulous to skeptical, as needed by the particular application.

Acknowledgements. The authors want to thank Michael Cheng for experimentation with an early version of the methods and for helpful discussions, and Dulce Aguilar-Solis for help with benchmark testing. This work was supported by the CONTROL project, funded by the Danish Natural Science Research Council; we also gratefully acknowledge support from Canada's NSERC Discovery Grant program.

References

1. M.Alberti, F.Chesani, M.Gavanelli, E.Lamma, P.Mello, and P.Torroni. The CHR-based Implementation of a System for Generation and Confirmation of Hypotheses. In *19th Workshop on (Constraint) Logic Programming* Wolf, A., Frühwirth, Th., and Meister, M., (eds.). Ulmer Informatik-Berichte 2005-01, University of Ulm. pp. 111–122. Available at http://www.informatik.uni-ulm.de/epindata/user/11541.218,UIB_2005-01.pdf
2. S. Abdennadher. Operational semantics and confluence of constraint propagation rules. In G. Smolka, editor, *Principles and Practice of Constraint Programming - CP97*, volume 1330 of *Lecture Notes in Computer Science*, pages 252–266. Springer, 1997.
3. Abdennadher, S. and Christiansen, H., An Experimental CLP Platform for Integrity Constraints and Abduction. In: Proc. of FQAS2000, Flexible Query Answering Systems, Larsen, H.L., Kacprzyk, J., Zadrozny, S., Andreasen, T., and Christiansen, H. (Eds.) pp. 141–152, *Advances in Soft Computing series,* Physica-Verlag (Springer), 2000.
4. Abdennadher, S. and Schütz, H. CHR$^\vee$: A flexible query language. In: Proc. FQAS'98, Flexible Query Answering Systems, Andreasen, T., Christiansen, H., and Larsen, H.L. (Eds.) *Lecture Notes in Artificial Intelligence* 1495, pp. 1–14, Springer, 1998.
5. Betz, H. and Frühwirth, T. A linear logic semantics for Constraint Handling Rules. In: Proc. CP 2005, Eleventh International Conference on Principles and Practice of Constraint Programming. *Lecture Notes in Computer Science*. To appear, 2005.

6. Badea, L. and Tilivea D. Abductive Partial Order Planning with Dependent Fluents. In: KI 2001: Advances in Artificial Intelligence, Joint German/Austrian Conference on AI, Baader, F., Brewka, G., Eiter, T., (eds.) *Lecture Notes in Artificial Intelligence* 2174 p. 63-77, Springer, 2001.
7. Chan, D., Constructive negation based on the database completion, *Proc. of Fifth International Conference and Symposium on Logic Programming*, (eds. Kowalski, Bowen), pp. 111–125, MIT Press, 1988.
8. Christiansen, H. Automated reasoning with a constraint-based metainterpreter, *Journal of Logic Programming*, Vol 37(1–3) Oct–Dec, pp. 213–253, 1998.
9. Christiansen, H., Abductive Language Interpretation as Bottom-up Deduction. In: Natural Language Understanding and Logic Programming, Proceedings of the 2002 workshop, ed. Wintner, S., *Datalogiske Skrifter* vol. 92, Roskilde University, Comp. Sci. Dept., pp. 33–47, 2002.
10. Christiansen, H., CHR grammars. *To appear in International Journal on Theory and Practice of Logic Programming, special issue on Constraint Handling Rules*, 2005.
11. Christiansen, H., and Dahl, V., Logic Grammars for Diagnosis and Repair. *International Journal on Artificial Intelligence Tools*, Vol. 2, no. 3 (2003), pp. 227–248.
12. H. Christiansen and V. Dahl. Assumptions and abduction in Prolog. In S. Muñoz-Hernández, J. M. Gómez-Perez, and P. Hofstedt, editors, *Proceedings of WLPE 2004: 14th Workshop on Logic Programming Environments and MultiCPL 2004: Third Workshop on Multiparadigm Constraint Programming Languages Workshop Proceedings*, pages 87–101, 2004.
13. Christiansen, H., Dahl, V., Meaning in Context, In: Proc. Fifth International and Interdisciplinary Conference on Modeling and Using Context (CONTEXT-05). Dey, A.K., Kokinov, B.N., Leake, D.B., and Turner R.M. (Eds.) *Lecture Notes in Artificial Inteligence* vol. 3554, pp. 97–111, 2005.
14. Dahl, V., and Tarau, P. Assumptive Logic Programming. *Argentine Symposium on Artificial Intelligence (ASAI)* 2004, September 20-21, Cordoba, Argentina, 2004.
15. Dahl, V., Tarau, P., and Li, R., Assumption grammars for processing natural language. *Proc. Fourteenth International Conference on Logic Programming.* Naish, L. (ed.), pp. 256–270, MIT Press, 1997.
16. G. J. Duck, P. J. Stuckey, M. J. G. de la Banda, and C. Holzbaur. Compiling ask constraints. In B. Demoen and V. Lifschitz, editors, *ICLP*, volume 3132 of *Lecture Notes in Computer Science*, pages 90–104. Springer, 2004.
17. Frühwirth, T.W., Theory and Practice of Constraint Handling Rules, *Journal of Logic Programming*, Vol. 37(1–3), pp. 95–138, 1998.
18. Kakas, A.C., Michael, A., and Mourlas, C. ACLP: Abductive Constraint Logic Programming, *The Journal of Logic Programming*, vol 44, pp. 129–177, 2000.
19. Kakas, A.C., Kowalski, R.A., and Toni, F. The role of abduction in logic programming, *Handbook of Logic in Artificial Intelligence and Logic Programming*, vol. 5, Gabbay, D.M, Hogger, C.J., Robinson, J.A., (eds.), Oxford University Press, pp. 235–324, 1998.
20. Kakas, A.C., Van Nuffelen, B., and Denecker, M. A-System: Problem Solving through Abduction. *IJCAI 2001: Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence*, Nebel, B. (ed.), Morgan-Kaufmann, pp. 591–596, 2001.
21. Pereira, F.C.N., and Warren, D.H.D., Definite clause grammars for language analysis. A survey of the formalism and a comparison with augmented transition grammars. *Artificial Intelligence* 10, no. 3–4, pp. 165–176, 1980.

22. Poole, D., Mackworth, A., and Goebel, R. *Computational Intelligence*, Oxford University Press, 1998.
23. *SICStus Prolog user's manual*. Version 3.12, SICS, Swedish Institute of Computer Science, 2005. Most recent version available at `http://www.sics.se/isl`.
24. Shanahan, M., Reinventing Shakey. *Logic-Based Artificial Intelligence*, Minker, J. (ed). Kluwer Academic, pp. 233–253, 1999.
25. Tarau, P., Dahl, V., and Fall, A. Backtrackable State with Linear Affine Implication and Assumption Grammars. In: Concurrency and parallelism, Programming, Networking, and Security, Jaffar, J. and Yap, R. (eds.). *Lecture Notes in Computer Science* 1179, Springer Verlag, pp. 53–64, 1996.

Abduction of Linear Arithmetic Constraints

Michael J. Maher

National ICT Australia, Sydney, Australia
Michael.Maher@nicta.com.au

Abstract. Abduction is usually carried out on partially-defined predicates. In this paper we investigate abduction applied to fully-defined predicates, specifically linear arithmetic constraints over the real numbers. Abduction in this context has application to query answering using views and type inference, and potential relevance to analysis of concurrent/constraint/logic programs. We show that only rarely do abduction problems over linear arithmetic constraints have unique most general answers. We characterize the cases where most general answers exist. In general there may be infinitely many maximally general answers, or even answers that are not represented by maximally general answers. We take steps towards representing such answers finitely.

1 Introduction

Abduction is the inference rule that derives, from B and C, A such that $A, B \vdash C$. It was considered by Peirce [11] to be – along with deduction and induction – one of the fundamental forms of reasoning. Mostly, abduction has been addressed in a setting of partially determined predicates or propositions. *Constraint abduction* [10] refers to abduction in a setting where A, B and C come from a set of pre-defined, completely-defined relations (constraints, in the sense of constraint logic programming [4]) and A, the answer, ensures $A \wedge B \rightarrow C$.

Traditional forms of abduction differ from constraint abduction in that they address predicates that are incompletely defined by some properties. Abductive logic programming [5] and abductive constraint logic programming (ACLP) [6] differ further from constraint abduction in that the inference relation $\vdash$ is not material implication. Furthermore, constraints are used in ACLP only in support of the abduction of predicates; constraints alone are not abduced from.

It is usual in works on abduction to require that A is consistent with B, that is, that A is *not* stronger than $\neg B$. However, if $B \wedge C$ is unsatisfiable then, for any answer A, $A \wedge B$ is unsatisfiable. (This because if $(A \wedge B) \rightarrow C$ then $(A \wedge B) \rightarrow (B \wedge C)$.)

Thus we can divide the simple problem into two cases:

- $B \wedge C$ is unsatisfiable
  In this case the answers are exactly those constraints stronger than $\neg B$.
- $B \wedge C$ is satisfiable
  In this case we are only interested in answers A that are consistent with B, that is, constraints that are *not* stronger than $\neg B$. Notice that this implies that the strengthening of an answer A might not be an answer

In this paper we focus on the latter case.

M. Gabbrielli and G. Gupta (Eds.): ICLP 2005, LNCS 3668, pp. 174–188, 2005.

Clearly, determining whether such a problem has an answer is trivial: we can simply choose A to be equivalent to C. Instead, we will address the issue of characterizing all answers and/or finitely representing them. In particular, we will seek to identify when a single answer can represent all answers. In this paper we will address linear arithmetic constraints. In a companion paper [10], equational constraints over the Herbrand universe are addressed. It turns out that different kinds of constraints require different techniques to characterize the answers.

In the next section we outline some applications of constraint abduction. Then, following some brief background on constraints, constraint abduction is formally defined. In Section 5 an abstract notion of rank is introduced and several properties are established of constraint domains that support such a notion. These properties are used in characterizing answers to constraint abduction problems on linear arithmetic equalities in Section 6. Section 7 addresses the same issue for linear arithmetic inequalities but the development is more complicated, partly because this constraint domain does not support a rank.

2 Applications of Constraint Abduction

We outline situations in type inference and query answering with views where constraint abduction is needed. Constraint abduction is also important for analysis of logic programs [1,3,7], although that work focuses on finite, artificially constructed domains.

2.1 Type Inference

Type systems have developed in two different but compatible directions. Though these developments have addressed functional languages, they will be exemplified here in a logic language. Work on index types [17] and "practical" dependent types [15] introduces extra parameters to types which provide a refinement of the types and a consequent ability to assert type-checkable statements about functions/predicates.

For example, the type of the `append` predicate might be

```
append(list(α, m), list(α, n), list(α, m+n))
```

which asserts, in addition to the parametric polymorphism of `append`, that the length of the third argument is the sum of the lengths of the first two arguments. Here the type `list(α, n)` is defined by

```
list(α, n) is [] where n = 0
           or [ α | list(α, m) ] where n = m+1
```

where the length of a list is implicitly defined as part of the type. In this case, every occurrence of `append` gives rise to an equation $x_1 + x_2 = x_3$ over integer variables when type-checking.

Work on guarded recursive types [16] presents the opportunity to refine types in a different way. These can allow different rules in the definition of a predicate to be typed differently, based on differing patterns in the head.

For example, given the type definition

```
type both(α) is i(α) where α = int
             or b(α) where α = bool
```

which expresses a discriminated union type, we can define

```
p(i(X), Y) :- Y = X + 1.
p(b(X), Y) :- Y = true.
```

When attempting to infer the acceptable types for X and Y, the rules of p generate expressions $t_X = int \rightarrow t_Y = int$ and $t_X = bool \rightarrow t_Y = bool$ where t_X and t_Y are variables representing the type of X and Y respectively.

In general, such refined types lead to similar implication constraint expressions [12]. Principal types associate a single parameterized type to each type variable, perhaps under some restrictions (such as those coming from type classses), such that other acceptable types are instances of the principal type. An algorithm is given in [13] for computing principal types, where they exist, in a guarded recursive type system. Principal types correspond to most general answers of constraint abduction problems over $\mathcal{FT}$, and abduction of such constraints was investigated in [10].

For the program above, p has a principal type

```
p(both(α), α)
```

which can be inferred as the answer $t_X = t_Y$ of the joint constraint abduction problem involving both implication constraint expressions.

The use of both these refinements of types leads for a need to handle expressions of the form $C_1 \rightarrow C_2$, where the C_i are constraints involving both equations on type expressions and other constraints on index type variables. In particular, linear arithmetic constraints are useful to express relations on sizes of data structures. Although we might expect to solve such constraints with an integer constraint solver, solving the constraints over the reals has a lower complexity and experience suggests that solving the constraints over the reals is sufficient [15]. Since we would like to assign a single principal type to each expression, it turns out that we are looking for a most general answer to the constraint abduction problem involving C_1 and C_2.

2.2 Query Answering

In database systems we sometimes want to answer a query as much as possible using previously defined relations. In a distributed database, this can limit the copying of large relations from one site to another. In data integration via a mediated schema, this provides a way to coherently query data from autonomous data sources. In both cases, previously defined relations are formulated as *views* – queries over the base relations – and the problem is to (partially) answer an input query, using the available views. We can allow pre-defined relations (constraints) in queries and views. In particular, for this paper, these are linear arithmetic constraints. See [2] for a more thorough and detailed survey of work on query answering using views than can be presented here.

If we have an input query

$$Q(\tilde{x}) : -P_1(\tilde{x}, \tilde{y}), \ldots, P_n(\tilde{x}, \tilde{y}), C(\tilde{x}, \tilde{y})$$

and views

$$V_i(\tilde{u}) : -P_{i1}(\tilde{u}, \tilde{v}), \ldots, P_{ik_i}(\tilde{u}, \tilde{v}), B_i(\tilde{u}, \tilde{v})$$

then we are looking for a query

$$Q'(\tilde{x}) : -V'_i(\tilde{x}, \tilde{w}), \ldots, V'_m(\tilde{x}, \tilde{w}), A(\tilde{x}, \tilde{w})$$

such that the relation Q' is a subrelation of Q, independent of the data in the base relations. Here P refers to a base relation, V to a view, and A, B, and C to constraints. Given the use of views $V'_1, \ldots, V'_m$, and some extra conditions on either these views or the constraint domain, A is required to satisfy $A \wedge \bigwedge_{i=1}^{m} B_i \rightarrow C$ to ensure that Q' is a subrelation of Q. That is, A must be an answer to the constraint abduction problem involving $\bigwedge_{i=1}^{m} B_i$ and C. Obviously, a finite representation of such answers is important to the computation of the subrelation Q'.

3 Constraints

The syntax and semantics of constraints are defined by a constraint domain. Given a signature Σ, and a set of variables $Vars$ (which we assume is infinite), a *constraint domain* is a pair $(\mathcal{D}, \mathcal{L})$ where $\mathcal{D}$ is a Σ-structure and $\mathcal{L}$ (the language of constraints) is a set of Σ-formulas closed under conjunction and renaming of free variables. When $\mathcal{D}$ is the real numbers and $\mathcal{L}$ is all conjunctions of linear equalities (respectively, inequalities) then these constraint domains will be denoted $\Re_{LinEqn}$ (resp. $\Re_{LinIneq}$). We will also discuss constraint domains $\mathbf{Q}_{LinEqn}$ and $\mathbf{Q}_{LinIneq}$, where $\mathcal{D}$ is the rational numbers, and $\mathbf{Z}_{LinEqn}$ and $\mathbf{Z}_{LinIneq}$, where $\mathcal{D}$ is the integers. For linear constraints, constraint domains based on $\Re$ or $\mathbf{Q}$ are elementarily equivalent, so all results for $\Re$ extend to $\mathbf{Q}$. We use $\mathcal{FT}$ to denote the constraint domain of equations over finite terms.

For most of the results of this paper we assume that constraint languages are generated, by conjunction and variable renaming, from a set of primitive constraints. Thus a constraint can be viewed as a set of primitive constraints, and every subset of a constraint is a constraint. We say that such constraint languages are *generated from primitive constraints*. This is certainly true of the constraint languages of the two constraints domains that are the main focus of this paper: $\Re_{LinEqn}$ and $\Re_{LinIneq}$. For these constraint domains we also have the property that the constraint language is closed under existential quantification: $\forall c \in \mathcal{L}\ \forall x \in Vars\ \exists c' \in \mathcal{L}\ \mathcal{D} \models c' \leftrightarrow \exists x\ c$. As a consequence, it will suffice to consider answers A such that $vars(A) \subseteq vars(B) \cup vars(C)$. (In contrast, this does not hold for $\mathcal{FT}$ [10].) We will write $\tilde{\exists}$ to express the existential closure of a formula, and sometimes use a comma to express conjunction.

We say C is *more general than* C' (or, equivalently, C' is *stronger than* C or, C' is *more specific than* C) if $C' \rightarrow C$. Two constraints C and C' are *equivalent* if $C \rightarrow C'$ and $C' \rightarrow C$.

The constraints modulo equivalence form a partially ordered set, where $C_1 \leq C_2$ iff $C_1 \rightarrow C_2$. The poset has top element $true$ and bottom element $false$. Any pair of constraints C_1 and C_2 has a greatest lower bound defined by $C_1 \wedge C_2$, as a consequence of

the assumption that $\mathcal{L}$ is closed under conjunction. On the other hand, the existence of a least upper bound $C_1 \sqcup C_2$ for any pair of constraints depends on the constraint domain.

4 Constraint Abduction

We can now formally define the simple constraint abduction problem.

Definition 1. *The* Simple Constraint Abduction (SCA) Problem *is as follows:*

Given a constraint domain $(\mathcal{D}, \mathcal{L})$, and given two constraints $B, C \in \mathcal{L}$ such that $\mathcal{D} \models \tilde{\exists}\, B \wedge C$, for what constraints $A \in \mathcal{L}$ does

$$\mathcal{D} \models (A \wedge B) \rightarrow C$$

and

$$\mathcal{D} \models \tilde{\exists}\, (A \wedge B)$$

An instance *of the problem has a fixed constraint domain and fixed constraints B and C.*

Throughout this paper, A, B and C refer to the constraints in a simple constraint abduction problem. We omit reference to the constraint domain when it is clear from the context and, for example, simply write $A \wedge B \rightarrow C$. In an abuse of terminology, we often will refer to an instance as a SCA problem.

There are two classes of problem instances where the SCA problem is easy. If $B \leftrightarrow true$ then C is an answer. If $B \rightarrow C$ then $true$ is an answer. We refer to these instances as *trivial*.

Of all the answers, we are most interested in the *maximally general answers*, that is, constraints A such that $(A \wedge B) \rightarrow C$ and for every A', if $A \rightarrow A'$ and $(A' \wedge B) \rightarrow C$ then $A' \rightarrow A$. (That is, there is no answer strictly more general than A.)

Under some circumstances, the maximally general answers represent all answers.

Definition 2 (Abductive Ascending Chain Property). *A problem has the* Abductive Ascending Chain (AAC) property *if whenever all constraints A_i in a chain satisfy $(A_i \wedge B) \rightarrow C$, the least upper bound of the chain exists and is an answer.*

If this is true for every problem in a constraint domain $(\mathcal{D}, \mathcal{L})$ then we say $(\mathcal{D}, \mathcal{L})$ has the Abductive Ascending Chain property.

Some constraint domains – such as $\mathcal{FT}$, various finite domains, and $\Re_{LinEqn}$ – do not have infinite ascending chains. Thus these constraint domains vacuously have the Abductive Ascending Chain property.

If a constraint domain has the AAC property then all answers are represented by the maximally general answers.

Proposition 1. *If a problem has the Abductive Ascending Chain property then all answers can be obtained, modulo equivalence, as a conjunction of a maximally general answer and another constraint.*

Thus, under the AAC property assumption, A is an answer iff for some maximally general answer A', $A \rightarrow A'$ and $A \wedge B$ is satisfiable.

Of particular interest are the problem instances in which one constraint represents all answers. In such cases there is a compact representation of the answers to the problem. An answer A to a SCA problem is a *most general answer* if, for every answer A' of the problem, $A' \rightarrow A$. Clearly a most general answer is unique up to equivalence of constraints. The main focus of this paper is on characterizing SCA problems that have a most general answer.

It is often convenient to eliminate unnecessary primitive constraints from a constraint, to simplify reasoning. A primitive constraint a in A is *redundant* in A if $(A - a) \rightarrow a$[1]. A is *redundancy-free* if there is no redundant constraint in A. A constraint a in A is *redundant* in A with respect to B if $(A - a) \wedge B \rightarrow a$. A is *redundancy-free* wrt B if there is no constraint in A that is redundant wrt B.

A *core* of a constraint C wrt another constraint B is a subset C' of C such that $(B \wedge C') \leftrightarrow (B \wedge C)$, and C' is redundancy-free wrt B. A core can be obtained from C by repeatedly deleting constraints that are redundant wrt B until there are no redundant constraints remaining.

In general, there is not a unique core.

Example 1. Consider the constraint domain $\Re_{LinEqn}$. Let B be $x + y = 0$ and let C be $2x - y = 0, 3x + y = 0$. Then each of the constraints in C is redundant wrt B, but not simultaneously. Thus each constraint in C is a core of C wrt B. Notice that the two cores are not equivalent.

Nevertheless, the replacement of C by a core does not alter the simple abduction problem.

Proposition 2. *Let A, B and C be constraints and let C' be a core of C wrt B. Then $A \wedge B \rightarrow C$ iff $A \wedge B \rightarrow C'$*

Obviously, a core of C wrt B is an answer for the corresponding SCA problem. Although a core of C wrt B might seem to be a good candidate for a maximally general answer, and perhaps even a most general answer, it is not always maximally general.

Example 2. Consider the constraint domain $\mathcal{FT}$. Let B be $x = a$ and let C be $f(x, y) = f(a, b)$. Then C is the core of C wrt B, but C is not a maximally general answer. This problem has a most general answer $y = b$.

Even if the use of a core C' of C wrt B does not immediately result in a maximally general answer, it does simplify the SCA problem by eliminating some irrelevant constraints. We can simplify further by reducing B to a core B' wrt C'. In this case we have $(B' \wedge C') \leftrightarrow (B \wedge C)$ but the simplification does not preserve answers to the SCA problem.

Example 3. Consider the constraint domain $\Re_{LinEqn}$. Let B be $x + y \geq 0$, and C be $x \geq 0 \wedge y \geq 0$. Then C', the core of C wrt B, is the same as C and B', the core of B wrt C', is $true$. However, let A be $x - y = 0$. Clearly $A \wedge B \rightarrow C$ but $A \wedge B' \not\rightarrow C'$.

However, it is not difficult to show that any answer to the SCA problem formed by B' and C' is also an answer to the problem formed by B and C.

[1] If A is a constraint that, when considered as a set of primitive constraints, contains a then we write $A - a$ to denote the conjunction of all primitive constraints in A except a.

We say a constraint A is *equivalent to C modulo B* if $(A \wedge B) \leftrightarrow (B \wedge C)$. Most general answers A have this property. Clearly, any core of C wrt B is equivalent to C modulo B. We might expect that such constraints are maximally general answers, but consider the following example. Let B be $y = d$ and C be $u = v$. Two answers are $u = v$ and $y = d \wedge u = v$, and both are equivalent to C modulo B, but clearly the second is not a maximally general answer.

This example is representative of how constraints equivalent to C modulo B might not be maximal. However the property of being equivalent to C modulo B is a kind of upper limit on answers to a SCA problem:

Proposition 3. *Suppose A is equivalent to C modulo B and A' is an answer more general than A. Then A' is equivalent to C modulo B.*

Thus if the problem has the Abductive Ascending Chain Property then every constraint equivalent to C modulo B is stronger than (or equivalent to) a maximally general answer equivalent to C modulo B.

Clearly many maximally general answers are equivalent to C modulo B. However, not all maximally general answers have this property. In particular, it can depend on the constraint domain.

Example 4. Consider a SCA problem over $\mathcal{FT}$ where B is $y = a$ and C is $x = h(u)$. Apart from C, this SCA problem has the maximally general answer A: $x = h(y) \wedge u = a$. (Notice that $A \not\rightarrow C$; for example, in a context where $y = b$.) Although A is maximally general it results in $A \wedge B$ being strictly stronger than $B \wedge C$.

We say an answer A is *fully maximal* if A is a maximally general answer and $A \wedge B$ is maximally general among all expressions $A' \wedge B$ where A' is an answer. Equivalently, A is fully maximal if A is a maximally general answer and A is equivalent to C modulo B. The answer A of Example 4 is not fully maximal. In [10], fully maximal answers were proposed as a way to handle the proliferation of maximally general answers over $\mathcal{FT}$.

5 Constraint Domains with Rank

It is difficult to establish further properties of constraint abduction without introducing conditions on the constraint domain. We now consider constraint domains that support a weak notion of rank in the following sense.

It is convenient in the following definition to simultaneously view constraints as conjunctions of primitive constraints and as sets of primitive constraints.

Definition 3. *Consider a constraint domain $\mathcal{D}$ in which the constraints are generated from primitive constraints. Let* rank *be a function mapping constraints to a totally ordered set. Let X, Y and Z range over constraints, that is, conjunctions (sets) of primitive constraints. Consider the following axioms:*

1. *if $X \rightarrow Y$ then* $\text{rank}(X) \geq \text{rank}(Y)$
2. *if $X \rightarrow Y$ and* $\text{rank}(X) = \text{rank}(Y)$ *then $X \leftrightarrow Y$*
3. *if $X \rightarrow Y$ and* $\text{rank}(X) > \text{rank}(Y)$ *then there exists $Z \subset X$ such that $X \rightarrow Z$, $Z \rightarrow Y$, and* $\text{rank}(X) > \text{rank}(Z) \geq \text{rank}(Y)$

We say a constraint domain supports a rank *if a function* rank *satisfying these axioms can be defined.*

It is well-known that $\Re_{LinEqn}$ supports a rank. Notice that we do not need to require the integers – not even a well-ordered set – as the co-domain of the rank function. Although there are many abstract formulations of dimension, an abstract formulation of rank seems to have been missing.

All constraint domains that support a rank have the Abductive Ascending Chain property. As a result, maximal general answers represent all answers.

Proposition 4. *Suppose a constraint domain $\mathcal{D}$ supports a rank. Then $\mathcal{D}$ has no infinite strictly increasing sequence. Consequently, $\mathcal{D}$ has the Abductive Ascending Chain property.*

When a constraint domain supports a rank there is a close relationship between maximally general answers and constraints equivalent to C modulo B. As a result, there is no distinction between maximally general answers and fully maximal answers when the constraint domain supports a rank.

Proposition 5. *Consider a constraint domain $\mathcal{D}$ that supports a rank, and a SCA problem over $\mathcal{D}$. Every maximally general answer is also a fully maximal answer.*

Under some further conditions the converse statement also holds.

Proposition 6. *Suppose a constraint domain $\mathcal{D}$ supports a rank that satisfies*

$$\text{rank}(C_1 \wedge C_2) \leq \text{rank}(C_1) + \text{rank}(C_2)$$

for all constraints C_1 and C_2.

Consider a SCA problem over $\mathcal{D}$.

If A is equivalent to C modulo B and $\text{rank}(A \wedge B) = \text{rank}(A) + \text{rank}(B)$ *then A is a maximally general answer of the problem.*

A notion of rank for $\mathcal{FT}$ was identified in [9] as the number of equations in the solved form of a constraint. Unfortunately, that definition fails to satisfy the third axiom. Thus the results of this section do not apply to $\mathcal{FT}$. There is a detailed analysis of constraint abduction over that constraint domain in [10].

6 Real Linear Equations

The constraint domain $\Re_{LinEqn}$ consists of linear equations over the real numbers. Any equalities can be expressed by two inequalities. The solved form of $\Re_{LinEqn}$ is a conjunction of equations of the form $\tilde{x} = \tilde{t}(\tilde{y})$ where $\tilde{x} \cap \tilde{y} = \emptyset$. A solved form also represents a substitution that replace each x_i by $t_i(\tilde{y})$. The solved form is essentially the same as achieved by Gauss-Jordan elimination.

In this section we assume that the rank of a constraint A is defined in the traditional way: the number of independent equations in A or, equivalently, the number of

non-trivial equations in the solved form of A. It is straightforward to see that the axioms for rank specified in Definition 3 are satisfied by this definition. Furthermore, $\text{rank}(A \wedge B) \leq \text{rank}(A) + \text{rank}(B)$ for any constraints A and B in $\Re_{LinEqn}$. Thus the propositions of the previous section apply to $\Re_{LinEqn}$. The rank of a constraint is closely related to redundancy, since the rank is the number of independent equations. There are some further relationships that are relevant to this paper.

Lemma 1. *Suppose A and B are redundancy-free. Then the following are equivalent:*

- $\text{rank}(A \wedge B) = \text{rank}(A) + \text{rank}(B)$
- *A is redundancy-free wrt B*
- *B is redundancy-free wrt A.*

By Proposition 4, in $\Re_{LinEqn}$ the maximally general answers represent all answers. However there may be infinitely many maximally general answers.

Example 5. Let B be $y = x$ and C be $y = -x$. Let A_m be $y = mx$ for any constant m. Then A_m is a maximally general answer for $m \neq 1$. There is similar behavior at higher dimensions.

Intuitively, B and C define intersecting, inequivalent hyperplanes. Any rotation of C about its intersection with B (except for B itself) is an answer.

As an immediate consequence of this observation we must look to characterize the maximally general answers rather than compute them individually. Fortunately, in $\Re_{LinEqn}$ we can do this.

Theorem 1. *Consider the SCA problem over $\Re_{LinEqn}$.*
A is a maximally general answer of the problem iff A is equivalent to C modulo B and $\text{rank}(A \wedge B) = \text{rank}(A) + \text{rank}(B)$.

In $\Re_{LinEqn}$ a SCA problem has a most general answer only in trivial cases.

Theorem 2. *Consider the SCA problem over $\Re_{LinEqn}$.*
There is a most general answer of the problem iff $B \leftrightarrow true$ or $B \rightarrow C$.

Notice that, if it exists, the most general answer is a core of C wrt B.

As a corollary to the proof of the above theorem we have a 1-∞ law for the number of maximally general answers of a SCA problem over $\Re_{LinEqn}$.

Corollary 1. *Every SCA problem over $\Re_{LinEqn}$ either has a most general answer or has infinitely many maximally general answers.*

7 Real Linear Inequalities

The constraint domain $\Re_{LinIneq}$ consists of linear inequalities over the real numbers.

It will be convenient to write all inequalities in the form $t \leq 0$ where t is a linear expression containing at most one occurrence of each variable, and one constant. All linear inequalities can be placed in this form.

A *linear combination* of inequalities C_i (or $t_i \leq 0$), $i = 1, \ldots$ is an inequality $\sum_i \alpha_i t_i + (-\delta) \leq 0$, where the α_i are constants and δ is a non-negative constant, and may be written $\sum_i \alpha_i C_i + (-\delta \leq 0)$. In a non-negative (positive) linear combination all α_i's are non-negative (respectively, positive).

If c is a single inequality $t \leq 0$ then $c^=$ denotes the corresponding equality $t = 0$. An *implicit equality* in a set of inequalities C is a non-tautologous inequality c in C such that $C \rightarrow c^=$. Geometrically, an equation defines a hyperplane, and the hyperplane of $c^=$ is said to be the supporting hyperplane of c (it is the boundary of the half-space defined by c). We say a constraint is *full dimensional* if it has no implicit equalities.

Two classes of techniques are used to prove the results in this section. The first is an algebraic approach to the duality of linear programming, based on linear combinations of inequalities, which allows us to address implicit equalities using the characterization of [8]. The second is the use of convexity, density, and topological characterizations of boundaries in the construction of answers. Unfortunately, space constraints prevent the presentation of proofs.

7.1 Abduction

It is substantially more difficult to address, for $\Re_{LinIneq}$, the issues addressed in the previous section for $\Re_{LinEqn}$. One problem is that $\Re_{LinIneq}$ does not support a rank.

Example 6. Let X be $x \geq 0, y \geq 0, x + y \leq 5$ and Y be $x \geq 0, y \geq 0, x + y \leq 6$, so that $X \rightarrow Y$. By axioms 1 and 2, if there is a rank, $\text{rank}(X) > \text{rank}(Y)$. However there is no $Z \subset X$ such that $Z \rightarrow Y$. Thus the third axiom cannot be satisfied. Hence $\Re_{LinIneq}$ does not support a rank.

A second problem is that, unlike many other constraint domains, the AAC property fails for $\Re_{LinIneq}$, so that the maximally general answers may not represent all answers of a SCA problem.

Proposition 7. *The constraint domain $\Re_{LinIneq}$ does not have the Abductive Ascending Chains Property.*

Proof. It is sufficient to exhibit one SCA problem with an ascending chain of answers where the least upper bound is not an answer. Let B be $-1 \leq x, x \leq 0, 0 \leq y, y \leq 2$ and C be $0 \leq x, x \leq 1, 0 \leq y, y \leq 1$. The second constraint in B is an implicit equality in $B \wedge C$. Note that $B \wedge C$ defines the line segment $x = 0, 0 \leq y, y \leq 1$.

Let A_k be $x \geq 0, x \geq k(y-1)$, where k is a rational number. Then A_k is an answer, for $0 < k < \infty$, and within that range for k, if $s < t$ then $A_t \rightarrow A_s$. The least upper bound of $\{A_k \mid k > 0\}$ is A_0. However, A_0 is not an answer. Consequently, this SCA problem does not have a maximally general answer.

Intuitively, for smaller and smaller values of $k > 0$, A_k defines a larger and larger proportion of the half-plane $x \geq 0$. Furthermore, the boundary of each A_k passes only through the point $y = 1$, when $x = 0$, and ensures $B \wedge A_k$ is equivalent to $B \wedge C$. However the least upper bound does not have this property: $B \wedge A_0$ defines the line segment $x = 0, 0 \leq y, y \leq 2$, and this does not imply $y \leq 1$ in C.

In addition to the trivial cases, $\Re_{LinIneq}$ has another case where the existence of a most general answer is obvious. If B and C only involve a single variable x, say, then the problem is *one-dimensional* and the constraints are simply bounds on x. In this case there is always a most general answer. For the remainder of this section we implicitly assume that the problem is not one-dimensional in this way.

7.2 Abduction Without Implicit Equalities

The presence or absence of implicit equalities is a major factor in the existence of most general answers. This is, perhaps, already apparent from considering $\Re_{LinEqn}$. We begin by establishing some properties that hold when implicit equalities are absent from B.

Lemma 2. *Consider the constraint domain $\Re_{LinIneq}$. Suppose $A \wedge B$ is consistent, A is redundancy-free wrt B, $A \rightarrow A'$, and $A' \wedge B \rightarrow A$.*

If no $B_0 \in B$ is an implicit equality in $A \wedge B$, then $A' \leftrightarrow A$.

The lemma has a useful corollary.

Corollary 2. *Consider a SCA problem over the constraint domain $\Re_{LinIneq}$ where no constraint in B is an implicit equality of $B \wedge C$.*

If A is equivalent to C modulo B and A is redundancy-free wrt B then A is a maximally general answer and a fully maximal answer for the SCA problem.

In particular, any core C' of C wrt B is a maximally general answer and a fully maximal answer. If the SCA problem has a most general answer, then C' is the most general answer.

The hypothesis that B has no implicit equalities in $B \wedge C$ is necessary, as the following example demonstrates.

Example 7. Let B be $-1 \leq x, x \leq 0, 0 \leq y, y \leq 1$ and C be $0 \leq x, x \leq 1, -1 \leq y, y \leq 0$. The second and third constraints in B are implicit equalities in $B \wedge C$.

Let A_k be $y \leq kx$ where $k > 0$ is a constant. Then, for each k, A_k is a maximally general answer. To see this note that any more general constraint A' would have the form $y \leq kx + c$ where $c > 0$ is a constant. But then $x = 0, y = min(c, 1)$ is a solution of $A' \wedge B$, but not of C. Thus A' is not an answer.

Hence the SCA problem does not have a most general answer and, indeed, has infinitely many maximally general answers.

As we saw in Proposition 7, for some SCA problems over $\Re_{LinIneq}$ not all answers are represented by maximally general answers. Nevertheless, we now turn our attention to characterizing the instances of SCA over $\Re_{LinIneq}$ that have most general answers. This is substantially more complicated than for $\Re_{LinEqn}$ and we will need several lemmas to develop different parts of the characterization.

Lemma 3. *Let C' be a core of C wrt B. Suppose $C' \not\rightarrow B$ and no inequality in B is an implicit equality in $B \wedge C'$.*

If some $C_0 \in C'$ is an implicit equality in $B \wedge C'$, then there is no most general answer for the SCA problem.

The following example illustrates this lemma.

Example 8. Let B be $x \geq y$ and let C be $y \geq 0, y \leq 0$. Then the SCA problem satisfies the conditions of the Lemma 3. The proof of the lemma constructs the answer A as $y \geq 0, x \leq 0$. Then $A \wedge B$ is satisfied only by $x = 0, y = 0$. Clearly $A \not\rightarrow C$ since $x = -1, y = 1$ satisfies A, but not C. Similarly, $C \not\rightarrow A$ since $x = 1, y = 0$ satisfies C, but not A. Furthermore, there is no constraint A' that is more general than both A and C and is an answer to the SCA problem: the least upper bound of A and C is $y \geq 0$, but this is not an answer.

We say two inequalities, C_1 and C_2 *oppose* each other if their corresponding hyperplanes are parallel $C_1 \not\rightarrow C_2$, $C_2 \not\rightarrow C_1$, and $C_1 \wedge C_2$ is satisfiable. Thus $C_1 \wedge C_2$ defines a polytope with two non-intersecting faces (or a single hyperplane if the boundaries of C_1 and C_2 coincide). The polytope is bounded in the direction perpendicular to the hyperplanes, but unbounded in every other orthogonal direction.

Lemma 4. *Let C' be a core of C wrt B and let B' be a core of B wrt C'. Suppose the SCA problem is non-trivial, $B \wedge C$ is full-dimensional, and $C' \not\rightarrow B$.*

Then there is a most general answer for the SCA problem iff B' and C' each consists of a single inequality, and they oppose each other.

Example 9. If B is $x + y \geq 2$ and C is $x + y \leq 4$ then the most general answer to this SCA problem is C. However, if B is $x + y \geq 2, x \geq 0$ and C is $x + y \leq 4$ then C is a maximally general answer, but it is not more general than the answer $y \geq -2x + 4$ and hence, by Corollary 2, there is no most general answer.

We say that a constraint B *touches* a constraint C if $C \rightarrow B$ and, for some $B_0 \in B$, $B_0^{=} \wedge C$ is satisfiable. This describes a situation where the polyhedron of B encloses the polyhedron of C but (at least) one facet of B makes glancing contact (i.e., touches) the boundary of C.

Lemma 5. *Let C' be a core of C wrt B. Suppose $C' \rightarrow B$ and no inequality of B is an implicit equality of $B \wedge C$. Then*

the SCA problem has a most general answer iff B does not touch C'

Example 10. Let B be $y \leq x + 1$ and let C be $x \geq 0, y \leq 1$. C is a core of C wrt B. Then B touches C at the point $x = 0, y = 1$. There is an answer $y \leq -x+1, y \geq -x+1$ that is not less general than C. Thus this SCA problem does not have a most general answer.

In the proof of Lemma 5 an answer is constructed that is something like $y \leq -x + 1, y \geq -x + 1, -1 \leq x, x \leq 0$.

We can now characterize when most general answers exist, under the assumption that B does not participate in any implicit equalities in $B \wedge C$.

Theorem 3. *Consider an SCA problem over $\Re_{LinIneq}$. Let C' be a core of C wrt B and let B' be a core of B wrt C'. Suppose no inequality of B is an implicit equality of $B \wedge C$.*

The problem has a most general answer iff and only if one of the following conditions is satisfied:

- *the problem is trivial,*
- *the problem is one-dimensional,*
- *$C' \rightarrow B$ and B does not touch C'*
- *$C' \not\rightarrow B$, and B' and C' are single opposing inequalities*

7.3 Abduction with Implicit Equalities

The previous section characterized the availability of a most general answer when B contains no implicit equalities. We now establish some lemmas addressing the issue in the presence of implicit equalities. If C contains both an inequality that is an implicit equality in $B \wedge C$ and one that is not then there is no most general answer.

Lemma 6. *Consider a SCA problem. Let C' be a core of C wrt B and suppose it contains both an implicit equality C_0 of $B \wedge C$ and an inequality C_1 that is not an implicit equality. Then this SCA problem does not have a most general answer.*

The next result is in some sense an extension of Lemma 4, since it corresponds to the case where the two opposing inequalities have the same supporting hyperplane.

Lemma 7. *Suppose B and C are full-dimensional, but $B \wedge C$ is one smaller in dimension.*

This SCA problem has a most general answer iff a core C' of C wrt B is a single inequality.

Except for the trivial case, if B has implicit equalities in B then there is no most general answer.

Lemma 8. *Suppose there is a non-tautological equation e such that $B \rightarrow e$.*

Then the SCA problem has a most general answer iff $B \rightarrow C$.

Informally, this result holds because if e is $t = 0$ then any constraint $s \leq 0$ in a core C' of C wrt B could be replaced by $s + t \leq 0$, giving a different answer.

Although we cannot use a most general answer to represent all answers in this case, a simple observation allows us to reduce this problem to a simpler one.

Lemma 9. *Suppose B has implicit equalities and we write them as explicit equations $\hat{\theta}$ in solved form. Then $A \wedge B \rightarrow C$ iff $A\theta \wedge B\theta \rightarrow C\theta$*

Thus we can extract all the implicit equalities from B, which can be done with standard techniques, and apply them as a substitution. The answers of the SCA problem on $B\theta$ and $C\theta$ are answers to the original problem and, furthermore, constraints that are equivalent to such an answer, modulo $\hat{\theta}$, are also answers to the original problem, and these are the only answers. Here θ is the substitution corresponding to solved form $\hat{\theta}$.

However the substitution θ does not simplify the problem into a form addressed in the previous subsection because B might have an implicit equality in $B \wedge C$ that is not an implicit equality in B. For example, if B is $x \leq y$ and C is $y \leq z, z \leq x$ then $B \wedge C$ implies $x = y$. Thus we need a stronger reduction.

Lemma 10. *Suppose $B \wedge C$ has implicit equations $\hat{\theta}$. Then $A \wedge B \rightarrow C$ iff $A \wedge B \rightarrow \hat{\theta}$ and $A\theta \wedge B\theta \rightarrow C\theta$.*

This still does not directly simplify the problem to the cases of the previous subsection because inequalities in B might be implicit equalities in $\hat{\theta}$. That is, for some $B_0 \in B$, it might be that $\hat{\theta} \rightarrow B_0^=$. Nevertheless, it shows that the problem of representing answers for SCA problems can be broken down into two specialized parts: abduction from implicit equalities, and abduction in a full-dimensional context.

We can now characterize when a SCA problem has a most general answer. It is apparent that the presence of implicit equalities in B precludes most general answers, except in the case addressed in Lemma 7.

Theorem 4. *Consider an SCA problem over $\Re_{LinIneq}$. Let C' be a core of C wrt B and let B' be a core of B wrt C'.*

The problem has a most general answer if and only if one of the following conditions is satisfied:

- *the problem is trivial,*
- *the problem is one-dimensional,*
- *no inequality of B is an implicit equality of $B \wedge C$, $C' \rightarrow B$ and B does not touch C'*
- *no inequality of B is an implicit equality of $B \wedge C$, $C' \not\rightarrow B$, and B' and C' are single opposing inequalities*
- *B is full-dimensional, $B \wedge C$ is one dimension less, and C' consists of a single inequality*

In [7] it was shown that a constraint language with primitive constraints of the form $ax + by \otimes 0$ where $a, b \in \{-1, 0, 1\}$, $\otimes \in \{<, \leq\}$ has the property that every SCA problem has a most general answer. However, as several of our examples suggest, a slightly richer language loses this property.

8 Discussion

Only simple constraint abduction has been addressed in this paper, although the applications discussed earlier require extensions to this basic idea. Query answering requires that A be restricted to variables in the query Q', while type inference requires the simultaneous solution of several SCA problems. The relationship between these extensions and SCA problems is addressed in [10]. In particular, using Proposition 6 of [10] and quantifier elimination we can determine whether a SCA problem can be solved under a given variable restriction. For $\Re_{LinEqn}$ and $\Re_{LinIneq}$ the quantifier elimination is relatively straightforward. Maximally general answers for the variable restricted abduction problem are simply those maximally general answers for the SCA problem that satisfy the variable restriction. Simultaneous solution of SCA problems can be reduced to solution of the individual SCA problems when the AAC property holds. However, this leaves the problem unaddressed for constraint domains like $\Re_{LinIneq}$.

While the results of this paper are also valid for linear constraints over the rational numbers, they do not necessarily extend to the integers. However, it is easy to see that the example showing $\Re_{LinIneq}$ does not have the AAC property also applies to $\mathbf{Z}_{LinIneq}$. This is an indication that characterizing SCA problems with most general answers over this domain will be no easier than for $\Re_{LinIneq}$.

Acknowledgement. I thank an anonymous referee for careful reading of an earlier draft.

References

1. R. Giacobazzi, Abductive analysis of modular logic programs, *Journal of Logic and Computation* 8(4):457-484, 1998.
2. A.Y. Halevy, Answering queries using views: A survey, *VLDB J.* 10(4): 270–294, 2001.
3. J. M. Howe, A. King, & L. Lu, Analysing Logic Programs by Reasoning Backwards in: M. Bruynooghe and K.-K. Lau (Eds), Program Development in Computational Logic, LNCS 3049, 152–188. 2004.
4. J. Jaffar & M.J. Maher, Constraint Logic Programming: A Survey, *Journal of Logic Programming 19 & 20*, 503–581, 1994.
5. A.C. Kakas, R.A. Kowalski, F. Toni, Abductive Logic Programming, *J. Logic and Computation* 2(6): 719–770, 1992.
6. A.C. Kakas, A. Michael, C. Mourlas, ACLP: Abductive Constraint Logic Programming, *J. Logic Programming* 44(1-3): 129–177, 2000.
7. A. King, & L. Lu, Forward versus Backward Verification of Logic Programs, *Proc. ICLP*, 315–330, 2003.
8. J-L. Lassez & M.J. Maher, On Fourier's Algorithm for Linear Arithmetic Constraints, *Journal of Automated Reasoning 9,* 373–379, 1992.
9. J-L. Lassez, M.J. Maher & K.G. Marriott, Unification Revisited, in: *Foundations of Deductive Databases and Logic Programming,* J. Minker (Ed), Morgan Kaufmann, 587–625, 1988.
10. M.J. Maher, Herbrand Constraint Abduction, *Proc. LICS*, 397–406, 2005.
11. C.S. Peirce, Collected Papers of Charles Saunders Peirce, C. Hartshorne & P. Weiss (Eds), Belknap Press of Harvard University Press, 1965.
12. V. Simonet & F. Pottier, Constraint-based type inference with guarded algebraic data types, submitted for publication, 2004. Available at `pauillac.inria.fr/~fpottier/biblio/pottier.html`
13. P.J. Stuckey, M. Sulzmann & J. Wazny, Existentially Quantified Type Classes, draft paper, 2004.
14. J. Wang, M. Maher & R. Topor, Rewriting Unions of General Conjunctive Queries Using Views, *Proc. Conf. on Extending Database Technology*, LNCS 2287, 52–69, 2002.
15. H. Xi, Dependent Types in Practical Programming, Ph.D. thesis, Carnegie-Mellon University, 1998.
16. H. Xi, C. Chen & G. Chen, Guarded recursive datatype constructors, *Proc. POPL*, 224–235, 2003.
17. C. Zenger, Indizierte Typen, Ph.D. thesis, Universität Karlsruhe, 1999.

Towards Implementations for Advanced Equivalence Checking in Answer-Set Programming*

Hans Tompits and Stefan Woltran

Institut für Informationssysteme 184/3, Technische Universität Wien,
Favoritenstraße 9-11, A-1040 Vienna, Austria
{tompits, stefan}@kr.tuwien.ac.at

Abstract. In recent work, a general framework for specifying program correspondences under the answer-set semantics has been defined. The framework allows to define different notions of equivalence, including the well-known notions of *strong* and *uniform equivalence*, as well as refined equivalence notions based on the *projection* of answer sets, where not all parts of an answer set are of relevance (like, e.g., removal of auxiliary letters). In the general case, deciding the correspondence of two programs lies on the fourth level of the polynomial hierarchy and therefore this task can (presumably) not be efficiently reduced to answer-set programming. In this paper, we describe an approach to compute program correspondences in this general framework by means of linear-time constructible reductions to *quantified propositional logic*. We can thus use extant solvers for the latter language as back-end inference engines for computing program correspondence problems. We also describe how our translations provide a method to construct *counterexamples* in case a program correspondence does not hold.

1 Introduction

Answer-set programming (ASP) is widely recognised as a fruitful paradigm for declarative knowledge representation and reasoning. It is based on the idea that problems are encoded in terms of theories of some suitable language, associated with a declarative semantics, such that the solutions of the given problems are determined by the models of the corresponding theories. Among the different instances of the ASP paradigm, the class of nonmonotonic logic programs under the answer-set semantics [14], with which we are concerned with in this paper, represents the canonical and, due to the availability of efficient answer-set solvers, like DLV [18], Smodels [26], and ASSAT [22], arguably most widely used ASP approach.

An important issue for the further deployment of ASP is to provide methods and tools for *engineering ASP solutions*. This includes techniques for the *simplification* and (offline) *optimisation* of programs, tools for supporting the user with *debugging* or *verification* features, and methods for *modular programming*. Crucial for all these issues are mechanisms for determining the *equivalence of* (*parts of*) *logic programs*.

In previous work [13], a general framework for specifying correspondences between logic programs under the answer-set semantics has been introduced. In this framework,

* This work was partially supported by the Austrian Science Fund (FWF) under grant P18019, and by the European Commission via projects FET-2001-37004 WASP, IST-2001-33570 INFOMIX, and IST-2001-33123 CologNeT.

M. Gabbrielli and G. Gupta (Eds.): ICLP 2005, LNCS 3668, pp. 189–203, 2005.

the correspondence of two programs is determined in terms of a class $\mathcal{C}$ of *context programs* and a comparison relation ρ such that correspondence between two programs P and Q holds iff the answer sets of $P \cup R$ and $Q \cup R$ satisfy ρ, for any program $R \in \mathcal{C}$. The framework includes as special cases the well-known notions of *strong equivalence* [20], *uniform equivalence* [10], and relativised notions thereof [28], as well as the practicably important case of program comparison under *projected* answer sets. In the latter setting, not a whole answer set of a program is of interest, but only its intersection on a subset of all letters; this includes, in particular, removal of auxiliary letters in computation.

For the case of propositional disjunctive logic programs, correspondence checking in the above framework under projected answer sets is surprisingly hard, viz. Π_4^P-complete in general [13], i.e., lying on the fourth level of the polynomial hierarchy. Hence, this task can (presumably) not be efficiently reduced to propositional answer-set programming. Such an approach (used, e.g., by Oikarinen and Janhunen [23] for ordinary equivalence) reduces equivalence checking to problems like program consistency such that equivalence holds iff the resultant program possesses no answer set. Taking the results of Eiter *et al.* [9] into account, a compact reduction as such cannot even be obtained by using non-ground programs as long as we restrict the arities of predicates to a fixed constant. This indicates that advanced equivalence tests in answer-set programming cannot be straightforwardly solved using ASP systems themselves.

In this paper, we describe an approach to compute program correspondences in the framework of Eiter *et al.* [13] by means of efficient reductions to *quantified propositional logic*. The latter is an extension of classical propositional logic characterised by the condition that its sentences, usually referred to as *quantified Boolean formulas* (QBFs), are permitted to contain quantifications over atomic formulas. More specifically, our reductions enjoy the following properties:

1. a solution correspondence under projected answer sets between two given logic programs holds iff the associated QBF is valid in quantified propositional logic,
2. the reduction is constructible in *linear time and space*, and
3. determining the validity of the resultant QBFs under the translations is not computationally harder than checking the original correspondence problem.

Besides the reduction of correspondence problems, we also describe how our translations provide a method to construct *counterexamples* in case a program correspondence does not hold.

The rationale to consider a reduction approach to QBFs is twofold: On the one hand, complexity results about quantified propositional logic imply that decision problems from the polynomial hierarchy can be efficiently represented in terms of QBFs, and, on the other hand, several practicably efficient solvers for quantified propositional logic are currently available (like, e.g., the solvers QuBE [15] and semprop [19]; for others, cf. [17,16]). Hence, tools of the latter kind can be used as back-end inference engines to compute the correspondence problems under consideration. We note that a similar reduction approach to QBFs has been successfully applied in diverse fields like nonmonotonic reasoning [6,5,12], paraconsistent reasoning [3,1,2], planning [25], and automated deduction [7].

2 Preliminaries

We deal with propositional disjunctive logic programs, which are finite sets of rules of form

$$a_1 \vee \cdots \vee a_l \leftarrow a_{l+1}, \ldots, a_m, not\, a_{m+1}, \ldots, not\, a_n, \tag{1}$$

with $n \geq m \geq l \geq 0$, and where all a_i are propositional atoms from a universe $\mathcal{U}$ and not denotes default negation. We call a rule of the above form a *fact* if $l = 1$ and $m = n = 0$, and a *constraint* if $l = 0$. If all atoms occurring in a program P are from a given set $A \subseteq \mathcal{U}$ of atoms, we say that P is a program *over* A. The set of all programs over A is denoted by $\mathcal{P}_A$.

Let I be an interpretation, i.e., a set of atoms, and P a program. Following Gelfond and Lifschitz [14], I is an *answer set* of P iff it is a minimal model of the *reduct* P^I, resulting from P by (i) deleting all rules containing default negated atoms $not\, a$ such that $a \in I$ and (ii) deleting all default negated atoms in the remaining rules. The set of all answer sets of P is denoted by $\mathcal{AS}(P)$. We also write $I \models P$ to indicate that each rule in P is true under I (in the sense of classical logic).

Under the answer-set semantics, two programs P and Q are regarded as (ordinarily) equivalent iff $\mathcal{AS}(P) = \mathcal{AS}(Q)$. The more restrictive form of *strong equivalence* [20] has recently been generalised as follows [28]: Let P, Q be programs over $\mathcal{U}$, and let $A \subseteq \mathcal{U}$. Then, P and Q are *strongly equivalent relative to* A iff, for any $R \in \mathcal{P}_A$ $\mathcal{AS}(P \cup R) = \mathcal{AS}(Q \cup R)$. If $A = \mathcal{U}$, strong equivalence relative to A reduces to strong equivalence; if $A = \emptyset$, we obtain ordinary equivalence.

We use the following notation: For an interpretation I and a set $\mathcal{S}$ of interpretations (resp., pairs of interpretations), we write $\mathcal{S}|_I = \{Y \cap I \mid Y \in \mathcal{S}\}$ (resp., $\mathcal{S}|_I = \{(X \cap I, Y \cap I) \mid (X, Y) \in \mathcal{S}\}$). If $\mathcal{S} = \{s\}$, we usually write $s|_I$ instead of $\mathcal{S}|_I$.

For any $A \subseteq \mathcal{U}$, a pair (X, Y) of interpretations, where $Y \subseteq \mathcal{U}$, is an *A-SE-interpretation* (over $\mathcal{U}$) iff either $X = Y$ or $X \subset Y|_A$. (X, Y) is an *A-SE-model* of a program P iff (i) $Y \models P$, (ii) for all $Y' \subset Y$ with $Y'|_A = Y|_A$, $Y' \not\models P^Y$, and (iii) $X \subset Y$ implies the existence of an $X' \subseteq Y$ with $X'|_A = X$ such that $X' \models P^Y$ holds. A pair (X, Y) is *total* iff $X = Y$, and *non-total* otherwise. The set of all A-SE-models of P is denoted by $SE^A(P)$.

For $A = \mathcal{U}$, the notion of an A-SE-interpretation (resp., A-SE-model) coincides with the notion of an *SE-interpretation* (resp., *SE-model*) as defined by Turner [27], and we write $SE(P)$ instead of $SE^{\mathcal{U}}(P)$. Thus, $(X, Y) \in SE(P)$ iff $X \subseteq Y, Y \models P$, and $X \models P^Y$.

Proposition 1 ([28]). *Two programs P and Q are strongly equivalent relative to A iff $SE^A(P) = SE^A(Q)$.*

Example 1. Consider the following two programs, P and Q:

$$P = P_0 \cup \{c \vee d \leftarrow a;\ c \vee d \leftarrow b\},$$
$$Q = P_0 \cup \{c \vee d \leftarrow a, b;\ d \leftarrow b, not\, c;\ c \leftarrow a, not\, d\},$$

for $P_0 = \{a \leftarrow c;\ b \leftarrow c;\ a \leftarrow d;\ b \leftarrow d;\ \leftarrow not\, c, not\, d\}$.

They have the following SE-models:[1]

$$SE(P) = \{(\emptyset, abc), (\emptyset, abd), (\emptyset, abcd), (abcd, abcd), (abc, abcd), (abd, abcd), (abc, abc), (abd, abd)\},$$
$$SE(Q) = SE(P) \cup \{(b, abc), (a, abd), (b, abcd), (a, abcd)\}.$$

Hence, P and Q are not strongly equivalent. On the other hand, $\mathcal{AS}(P) = \mathcal{AS}(Q) = \emptyset$, i.e., P and Q are (ordinarily) equivalent. Moreover, P and Q are strongly equivalent relative to A precisely if $A \cap \{a, b\} = \emptyset$. For $A = \{a, b\}$, we get

$$SE^A(P) = \{(\emptyset, abc),\ (\emptyset, abd),\ (abc, abc),\ (abd, abd)\},$$
$$SE^A(Q) = SE^A(P) \cup \{(b, abc),\ (a, abd)\}.$$

Thus, P and Q are not strongly equivalent relative to $A = \{a, b\}$. For instance, adding a fact $a \leftarrow$ yields $\mathcal{AS}(P \cup \{a \leftarrow\}) = \{abc, abd\}$, while $\mathcal{AS}(Q \cup \{a \leftarrow\}) = \{abc\}$. □

A set $\mathcal{S}$ of SE-interpretations is *complete* iff, for each $(X, Y) \in \mathcal{S}$, also $(Y, Y) \in \mathcal{S}$ as well as $(X, Z) \in \mathcal{S}$, for any Z such that $Y \subseteq Z$ and $(Z, Z) \in \mathcal{S}$. It can be shown that, for any program P, the set $SE(P)$ of all SE-models of P is always complete. Conversely, any complete set $\mathcal{S}$ of SE-interpretations can be represented by some program P. As a general result, taking also a restricted alphabet A into account, the following result holds:

Proposition 2 ([13]). *Let $\mathcal{S}$ be a complete set of SE-interpretations, and let A be a set of atoms. Then, there exists a program $P_{\mathcal{S},A} \in \mathcal{P}_A$ such that $SE(P_{\mathcal{S},A})|_A = \mathcal{S}|_A$.*

One possibility to obtain $P_{\mathcal{S},A}$ from $\mathcal{S}$ is as follows:

1. for each $Y \subseteq A$ with $(Y, Y) \notin \mathcal{S}|_A$, add rules $\bot \leftarrow Y, not\ (A \setminus Y)$, and
2. for each $X \subset Y$ with $(X, Y) \notin \mathcal{S}|_A$ and $(Y, Y) \in \mathcal{S}|_A$, add rules $\bigvee_{p \in (Y \setminus X)} p \leftarrow X, not\ (A \setminus Y)$.

3 Correspondence Checking

In order to deal with differing notions of program equivalence in a uniform manner, taking in particular strong equivalence and its relativised version, as well as equivalence notions based on the projection of answer sets into account, Eiter *et al.* [13] introduced a general framework for specifying differing notions of equivalence. In this framework, one parameterises, on the one hand, the class of programs used to be added to the programs under consideration, and, on the other hand, the relation that has to hold between the collection of answer sets of the extended programs.

Definition 1. *A* correspondence frame*, or simply* frame*, $\mathcal{F}$, is a triple $(\mathcal{U}, \mathcal{C}, \rho)$, where (i) $\mathcal{U}$ is a set of atoms, called the* universe of $\mathcal{F}$*, (ii) $\mathcal{C} \subseteq \mathcal{P}_\mathcal{U}$, called the* context of $\mathcal{F}$*, and (iii) $\rho \subseteq 2^{2^\mathcal{U}} \times 2^{2^\mathcal{U}}$.*

*For every program $P, Q \in \mathcal{P}_\mathcal{U}$, P and Q are $\mathcal{F}$-*corresponding*, in symbols $P \simeq_\mathcal{F} Q$, iff, for all $R \in \mathcal{C}$, $(\mathcal{AS}(P \cup R), \mathcal{AS}(Q \cup R)) \in \rho$.*

[1] We write abc instead of $\{a, b, c\}$, a instead of $\{a\}$, etc.

It is quite obvious that the equivalence notions presented in the previous section are special cases of $\mathcal{F}$-correspondence. Indeed, for any universe $\mathcal{U}$ and any $A \subseteq \mathcal{U}$, strong equivalence relative to A coincides with $(\mathcal{U}, \mathcal{P}_A, =)$-correspondence, and ordinary equivalence coincides with $(\mathcal{U}, \{\emptyset\}, =)$-correspondence.

Following Eiter *et al.* [13], we are mainly concerned with correspondence frames of form $(\mathcal{U}, \mathcal{P}_A, \subseteq_B)$ and $(\mathcal{U}, \mathcal{P}_A, =_B)$, where $A, B \subseteq \mathcal{U}$ are sets of atoms, and $\subseteq_B$ and $=_B$ are projections of the standard subset and set-equality relation, respectively, defined as follows: for any set $\mathcal{S}, \mathcal{S}'$ of interpretations, $\mathcal{S} \subseteq_B \mathcal{S}'$ iff $\mathcal{S}|_B \subseteq \mathcal{S}'|_B$, and $\mathcal{S} =_B \mathcal{S}'$ iff $\mathcal{S}|_B = \mathcal{S}'|_B$.

A *correspondence problem*, Π, (*over* $\mathcal{U}$) is a quadruple $(P, Q, \mathcal{C}, \rho)$, where $P, Q \in \mathcal{P}_\mathcal{U}$ and $(\mathcal{U}, \mathcal{C}, \rho)$ is a frame. We say that Π *holds* iff $P \simeq_{(\mathcal{U},\mathcal{C},\rho)} Q$ holds. For a correspondence problem $\Pi = (P, Q, \mathcal{C}, \rho)$ over $\mathcal{U}$, we usually leave $\mathcal{U}$ implicit, assuming that it consists of all atoms occurring in P, Q, and $\mathcal{C}$. We call Π an *equivalence problem* if ρ is given by $=_B$, and an *inclusion problem* if ρ is given by $\subseteq_B$, for some $B \subseteq \mathcal{U}$. Note that $(P, Q, \mathcal{C}, =_B)$ holds iff $(P, Q, \mathcal{C}, \subseteq_B)$ and $(Q, P, \mathcal{C}, \subseteq_B)$ jointly hold.

For inclusion problems, we define the concept of a *counterexample*, which is easily extended to equivalence problems.

Definition 2. *A pair (Y, R), where Y is an interpretation and $R \in \mathcal{C}$, is a* counterexample *for $(P, Q, \mathcal{C}, \subseteq_B)$ iff (i) $Y \in \mathcal{AS}(P \cup R)$ and (ii) $Z \notin \mathcal{AS}(Q \cup R)$, for each Z with $Z =_B Y$.*

Example 2. We have already seen that for P, Q from Example 1, $(P, Q, \mathcal{P}_A, \subseteq_\mathcal{U})$ does not hold for $A = \{a, b\}$ and $\mathcal{U} = \{a, b, c, d\}$. What happens if we restrict the comparison of answer sets from $\mathcal{U}$ to A, i.e., does $(P, Q, \mathcal{P}_A, \subseteq_A)$ hold? Note that, e.g., $\mathcal{AS}(P \cup \{a \leftarrow\})|_A = \mathcal{AS}(Q \cup \{a \leftarrow\})|_A = \{ab\}$. Hence, the counterexample $(abc, \{a \leftarrow\})$ from Example 1 is no longer a counterexample for $(P, Q, \mathcal{P}_A, \subseteq_A)$. As we shall see below, there still exist counterexamples for this problem, but these are more involving. □

As shown by Eiter *et al.* [13], inclusion problems with projection may possess only counterexamples which are exponential in the size of the compared programs. Hence, instead of guessing concrete programs and checking whether they are counterexamples for a given inclusion problem, Eiter *et al.* provide a semantical structure, called *spoiler*, which operates on the compared programs alone, together with the notion of a *partial spoiler*.

Definition 3. *Let $\Pi = (P, Q, \mathcal{P}_A, \subseteq_B)$ be an inclusion problem, Y an interpretation, and $\mathcal{S} \subseteq SE^A(Q) \cap \{(X, Z) \mid Z =_{A \cup B} Y\}$ a complete set of A-SE-interpretations. The pair $(Y, \mathcal{S})$ is a* spoiler for Π *iff*

1. *$(Y, Y) \in SE^A(P)$,*
2. *each $(Z, Z) \in SE^A(Q)$ such that $Z =_{A \cup B} Y$ is also in $\mathcal{S}$,*
3. *for each $(Z, Z) \in \mathcal{S}$, some non-total $(X, Z) \in \mathcal{S}$ exists, and*
4. *for each non-total $(X, Z) \in \mathcal{S}$, $(X, Y) \notin SE^A(P)$.*

For a spoiler $(Y, \mathcal{S})$, the interpretation Y is referred to as a partial spoiler *for* Π.

Intuitively, in a spoiler $(Y, \mathcal{S})$ for $(P, Q, \mathcal{P}_A, \subseteq_B)$, the interpretation Y is an answer set of $P \cup R$ but not of $Q \cup R$, where R is a program which is semantically given by $\mathcal{S}$.

We collect and rephrase the main results from [13].

Proposition 3. *Let* $\Pi = (P, Q, \mathcal{P}_A, \subseteq_B)$ *be an inclusion problem. Then,* Π *holds iff there exists no spoiler for* Π.

As an immediate consequence, we obtain that a correspondence problem Π holds iff there exists no partial spoiler for Π. Moreover, we are able to connect spoilers to counterexamples using the generic programs $P_{\mathcal{S},A}$, as introduced in Section 2.

Proposition 4. *If* $(Y, \mathcal{S})$ *is a spoiler for an inclusion problem* $\Pi = (P, Q, \mathcal{P}_A, \subseteq_B)$, *then* $(Y, P_{\mathcal{S},A})$ *is a counterexample for* Π.

Example 3. For P and Q from Example 1 and $A = \{a, b\}$, the pairs $(Y_1, \mathcal{S})$ and $(Y_2, \mathcal{S})$ are the only spoilers for $(P, Q, \mathcal{P}_A, \subseteq_A)$, where $Y_1 = \{abc\}$ and $Y_2 = \{abd\}$ are the partial spoilers for $(P, Q, \mathcal{P}_A, \subseteq_A)$, and $\mathcal{S} = \{(a, abd), (b, abc), (abc, abc), (abd, abd)\}$. Invoking our program construction, we obtain $P_{\mathcal{S},A} = \{\bot \leftarrow a, not\ b;\ \bot \leftarrow b, not\ a;\ \bot \leftarrow not\ a, not\ b;\ a \vee b \leftarrow\}$. One can verify that both Y_1 and Y_2 are contained in $\mathcal{AS}(P_1 \cup P_{\mathcal{S},A})$, while no interpretation Z with $Z =_A Y_1$ is an answer set of $Q \cup P_{\mathcal{S},A}$. □

Finally, we recall the computational complexity of checking whether an equivalence or inclusion problem holds. As shown by Eiter *et al.* [13], deciding $(P, Q, \mathcal{P}_A, =_B)$ is of a significantly higher complexity compared to more restricted notions of equivalence, like strong equivalence (which is coNP-complete) or ordinary equivalence and relativised strong equivalence (which both are Π_2^P-complete).

Proposition 5 ([13]). *Given programs* P, Q, *sets of atoms* A, B, *and* $\rho \in \{\subseteq_B, =_B\}$, *deciding whether a correspondence problem* $(P, Q, \mathcal{P}_A, \rho)$ *holds is* Π_4^P*-complete.*

4 Reductions

In this section, we provide two approaches to map inclusion problems into quantified Boolean formulas. By combining the reduction for a problem $(P, Q, \mathcal{P}_A, \subseteq_B)$ with that of $(Q, P, \mathcal{P}_A, \subseteq_B)$, we straightforwardly obtain a method to check whether an equivalence problem $(P, Q, \mathcal{P}_A, =_B)$ holds. We start with a brief recapitulation of the basic facts about the quantified version of propositional logic.

4.1 Quantified Propositional Logic

Quantified propositional logic is an extension of classical propositional logic in which formulas are permitted to contain quantifications over propositional variables. More formally, formulas of quantified propositional logic are built from atomic formulas using the primitive sentential connectives $\neg$ and $\wedge$, the logical constant $\top$, and unary operators of form $\forall p$ (where p is some atom), called *universal quantifiers*. The operators $\vee$, $\rightarrow$, and $\leftrightarrow$, as well as the symbol $\bot$, are defined from the primitive ones, $\neg$, $\wedge$, and $\top$, as usual. Furthermore, similar to first-order logic, $\exists p$ is defined as the operator

$\neg\forall p\neg$, referred to as an *existential quantifier*. Formulas of this language are also called *quantified Boolean formulas* (QBFs) and we denote them by Greek upper-case letters.

An occurrence of an atom p is *free* in a QBF Φ if it does not occur in the scope of a quantifier $\mathsf{Q}p$, $\mathsf{Q} \in \{\exists, \forall\}$. In what follows, we tacitly assume that every subformula $\mathsf{Q}p\,\Phi$ of a QBF contains a free occurrence of p in Φ, and for two different subformulas $\mathsf{Q}p\,\Phi$, $\mathsf{Q}q\,\Psi$ of a QBF we require $p \neq q$. Moreover, given a finite set P of atoms, $\mathsf{Q}P\,\Psi$ stands for any QBF $\mathsf{Q}p_1\mathsf{Q}p_2 \ldots \mathsf{Q}p_n\Psi$ such that the variables $p_1, \ldots, p_n$ are pairwise distinct and $P = \{p_1, \ldots, p_n\}$.

Towards the definition of the semantics of QBFs, we introduce the following notation: For an atom p (resp., a set P of atoms) and a set I of atoms, $\Phi[p/I]$ (resp., $\Phi[P/I]$) denotes the QBF resulting from Φ by replacing each free occurrence of p (resp., each $p \in P$) in Φ by $\top$ if $p \in I$ and by $\bot$ otherwise.

For an interpretation I and a QBF Φ, the relation $I \models \Phi$ is inductively defined as follows:

1. $I \models \top$,
2. $I \models p$ iff $p \in I$,
3. $I \models \neg\Phi$ iff $I \not\models \Phi$,
4. $I \models \Phi_1 \wedge \Phi_2$ iff $I \models \Phi_1$ and $I \models \Phi_2$, and
5. $I \models \forall p\,\Phi$ iff $I \models \Phi[p/\{p\}]$ and $I \models \Phi[p/\emptyset]$.

From these conditions, corresponding ones for $\bot$, $\vee$, $\rightarrow$, $\leftrightarrow$, and $\exists p$, for any p, follow in the usual way.

A QBF Φ is *true under* I iff $I \models \Phi$, otherwise Φ is *false under* I. A QBF is *valid* iff it is true under any interpretation. Note that a *closed* QBF, i.e., a QBF without free variable occurrences, is either true under any I or false under any I.

A QBF Φ is said to be in *prenex normal form* iff it is closed and of the form

$$\mathsf{Q}_n P_n \ldots \mathsf{Q}_1 P_1\,\phi, \tag{2}$$

where $n \geq 0$, ϕ is a propositional formula, $\mathsf{Q}_i \in \{\exists, \forall\}$ such that $\mathsf{Q}_i \neq \mathsf{Q}_{i+1}$ for $1 \leq i \leq n-1$, $(P_1, \ldots, P_n)$ is a partition of the propositional variables occurring in ϕ, and $P_i \neq \emptyset$, for each $1 \leq i \leq n$. We call a QBF of the form (2) an (n, Q_n)-*QBF*.

Without going into details, we mention that any closed QBF Φ is easily transformed into an equivalent QBF in prenex normal form such that each quantifier occurrence from the original QBF corresponds to a quantifier occurrence in the prenex normal form. Let us call such a QBF the *prenex normal form of* Φ. Similar as in first-order logic, however, there are different ways how to obtain an equivalent prenex QBF (cf. [8] for more details on this issue).

The following property is essential:

Proposition 6. *For every $k \geq 0$, deciding the truth of a given $(k, \exists)$-QBF (resp., $(k, \forall)$-QBF) is Σ_k^P-complete (resp., Π_k^P-complete).*

Hence, any decision problem $\mathcal{D}$ in Σ_k^P (resp., Π_k^P) can be mapped in polynomial time to a $(k, \exists)$-QBF (resp., $(k, \forall)$-QBF) Φ such that $\mathcal{D}$ holds iff Φ is valid. In particular, Proposition 5 implies thus that any correspondence problem $(P, Q, \mathcal{P}_A, \rho)$, for $\rho \in \{\subseteq_B, =_B\}$, can be reduced in polynomial time to a $(4, \forall)$-QBF. In what follows, we construct two such mappings which are actually constructible in *linear space and time*.

4.2 Encodings

For our encodings, we use the following building blocks. The idea hereby is to use sets of globally new atoms in order to refer to different assignments of the atoms from the compared program within a single formula. More formally, given an indexed set V of atoms, we assume (pairwise) disjoint copies $V_i = \{v_i \mid v \in V\}$, for every $i \geq 1$. Furthermore, we introduce the following abbreviations:

1. $(V_i \leq V_j) := \bigwedge_{v \in V}(v_i \rightarrow v_j)$,
2. $(V_i < V_j) := (V_i \leq V_j) \wedge \neg(V_j \leq V_i)$, and
3. $(V_i = V_j) := (V_i \leq V_j) \wedge (V_j \leq V_i)$.

Observe that the latter is clearly equivalent to $\bigwedge_{v \in V}(v_i \leftrightarrow v_j)$.

Proposition 7. *Let I be an interpretation, and $A, X, Y \subseteq V$ such that, for some i, j, $I|_{V_i} = X_i$ and $I|_{V_j} = Y_j$. Then,*

1. *$X|_A \subseteq Y|_A$ iff $I \models (A_i \leq A_j)$,*
2. *$X|_A \subset Y|_A$ iff $I \models (A_i < A_j)$, and*
3. *$X|_A = Y|_A$ iff $I \models (A_i = A_j)$.*

In accordance to the above renaming of atoms, we use subscripts as a general renaming schema for formulas and rules. That is, for each $i \geq 1$, α_i expresses the result of replacing each occurrence of an atom p in α by p_i, where α is any formula or rule. Furthermore, for a rule r of form (1), we define $H(r) = a_1 \vee \cdots \vee a_l$, $B^+(r) = a_{l+1} \wedge \cdots \wedge a_m$, and $B^-(r) = \neg a_{m+1} \wedge \cdots \wedge \neg a_n$. We identify empty disjunctions with $\bot$ and empty conjunctions with $\top$. Finally, for a program P, we define $P_{i,j} = \bigwedge_{r \in P}\big((B^+(r_i) \wedge B^-(r_j)) \rightarrow H(r_i)\big)$.

Proposition 8. *Let P be a program over atoms V, I an interpretation, and $X, Y \subseteq V$ such that, for some i, j, $I|_{V_i} = X_i$ and $I|_{V_j} = Y_j$. Then, $X \models P^Y$ iff $I \models P_{i,j}$.*

Intuitively, this allows to refer to the reduct of P (in case that $i \neq j$) and to the classical formula associated to P (in case that $i = j$) simultaneously.

The central characterisation towards our encodings, given next, replaces the concept of an A-SE-model in Definition 3 by tests over program reducts.

Proposition 9. *An interpretation Y is a partial spoiler for $(P, Q, \mathcal{P}_A, \subseteq_B)$ iff*

(a) $Y \models P$,
(b) for each $Y' \subset Y$ with $Y' =_A Y$, $Y' \not\models P^Y$, and
(c) for each $Z =_{A \cup B} Y$, $Z \models Q$ implies the existence of an $X \subset Z$ such that $X \models Q^Z$ and, if $X \subset Z|_A = Y|_A$, then, for each $X' \subseteq Y$ with $X' =_A X$, $X' \not\models P^Y$.

Definition 4. *Let P, Q be programs over V and let $A, B \subseteq V$. Furthermore, consider $\Pi = (P, Q, \mathcal{P}_A, \subseteq_B)$. Then,*

$$\boldsymbol{S}_\Pi(V_1) := P_{1,1} \wedge \boldsymbol{S}^1(P, A) \wedge \forall V_3\big(\boldsymbol{S}^2(Q, A, B) \rightarrow \boldsymbol{S}^3(P, Q, A)\big), \text{ where}$$
$$\boldsymbol{S}^1(P, A) := \forall V_2\big(((A_2 = A_1) \wedge (V_2 < V_1)) \rightarrow \neg P_{2,1}\big),$$
$$\boldsymbol{S}^2(Q, A, B) := \big((A \cup B)_3 = (A \cup B)_1\big) \wedge Q_{3,3}, \text{ and}$$
$$\boldsymbol{S}^3(P, Q, A) := \exists V_4\big((V_4 < V_3) \wedge Q_{4,3} \wedge \big((A_4 < A_1) \rightarrow \forall V_5\big(((A_5 = A_4) \wedge (V_5 \leq V_1)) \rightarrow \neg P_{5,1}\big)\big)\big).$$

Lemma 1. *Let P and Q be programs over V, and let $A, B, Y \subseteq V$. Then, Y is a partial spoiler for $\Pi = (P, Q, \mathcal{P}_A, \subseteq_B)$ iff $Y_1 \models \boldsymbol{S}_\Pi(V_1)$.*

We do not give a formal proof here, but just provide the following explanations. The subformula $P_{1,1} \wedge \boldsymbol{S}^1(P, A)$ of $\boldsymbol{S}_\Pi(V_1)$ takes care of Conditions (a) and (b) from Proposition 9; we use atoms V_1 to refer to Y, and atoms V_2 to refer to the Y' therein. Note that $(A_2 = A_1) \wedge (V_2 < V_1)$ thus guarantees that we take only those Y' for testing $Y' \models P^Y$ into account, where $Y' \subset Y$ and $Y' =_A Y$. The next subformula, $\boldsymbol{S}^2(Q, A, B)$, "returns" all Z (via assignments to V_3) such that $Z =_{A \cup B} Y$ and $Z \models Q$. Finally, for each such Z, $\boldsymbol{S}^3(P, Q, A)$ has to be true. On account of $(V_4 < V_3)$, we let the assignments to V_4 (which refer to the X in Item (c) of Proposition 9) be a proper subset of those to V_3, i.e., we require $X \subset Z$. Then we test whether $X \models Q^Z$ via $Q_{4,3}$, as follows from Proposition 8, and in case that $X|_A \subset Y|_A$ (checked via $A_4 < A_3$), the remaining formula encodes the test whether for all X' (assignments to V_5) with $X' =_A X$ and $X' \subseteq Y$, $X' \not\models P^Y$, i.e., $P_{5,1}$ is false under the current assignment to V_1 and V_5.

In what follows, we give a more compact encoding, which in particular reduces the number of universal quantifications. The idea is to save on the fixed assignments as, e.g., in $\boldsymbol{S}^2(Q, A, B)$ where we have $(A \cup B)_3 = (A \cup B)_1$. That is, in $\boldsymbol{S}^2(Q, A, B)$, we implicitly ignore all assignments to V_3 where atoms from A or B have different truth values as those in V_1. Therefore, it makes sense to consider only atoms from $V_3 \setminus (A_3 \cup B_3)$ and using $A_1 \cup B_1$ instead of $A_3 \cup B_3$ in $Q_{3,3}$.

This calls for a more subtle renaming schema for programs, however. Let $\mathcal{V}$ be a set of indexed atoms, and let r be a rule. Then, $r^{\mathcal{V}}_{i,k}$ results from r by replacing each atom x in r by x_i, providing $x_i \in \mathcal{V}$, and by x_k otherwise. For a program P, we define

$$P^{\mathcal{V}}_{i,j,k} := \bigwedge_{r \in P} \big((B^+(r^{\mathcal{V}}_{i,k}) \wedge B^-(r^{\mathcal{V}}_{j,k})) \rightarrow H(r^{\mathcal{V}}_{i,k})\big).$$

Moreover, for every $i \geq 1$, every set V of atoms, and every set C, $V_i^C := (V \setminus C)_i$.

Definition 5. *Let P, Q be programs over V and $A, B \subseteq V$. Furthermore, let $\Pi = (P, Q, \mathcal{P}_A, \subseteq_B)$ be an inclusion problem and $\mathcal{V} = V_1 \cup V_2^A \cup V_3^{A \cup B} \cup V_4 \cup V_5^A$. Then,*

$$\boldsymbol{T}_\Pi(V_1) := P_{1,1} \wedge \boldsymbol{T}^1(P, A, \mathcal{V}) \wedge \forall V_3^{A \cup B}\big(Q^{\mathcal{V}}_{3,1,1} \rightarrow \boldsymbol{T}^3(P, Q, A, \mathcal{V})\big), \text{ where}$$
$$\boldsymbol{T}^1(P, A, \mathcal{V}) := \forall V_2^A\big((V_2^A < V_1^A) \rightarrow \neg P^{\mathcal{V}}_{2,1,1}\big) \text{ and}$$
$$\boldsymbol{T}^3(P, Q, A, \mathcal{V}) := \exists V_4\big(\big(V_4 < ((A \cup B)_1 \cup V_3^{A \cup B})\big) \wedge Q^{\mathcal{V}}_{4,3,1} \wedge \big((A_4 < A_1) \rightarrow \forall V_5^A((V_5^A \leq V_1^A) \rightarrow \neg P^{\mathcal{V}}_{5,1,4})\big)\big).$$

Note that the subformula $V_4 < ((A \cup B)_1 \cup V_3^{A \cup B})$ in $\boldsymbol{T}^3(P, Q, A, \mathcal{V})$ denotes

$$\big(\big((A \cup B)_4 \leq (A \cup B)_1\big) \wedge (V_4 \leq V_1)\big) \wedge \neg\big(\big((A \cup B)_1 \leq (A \cup B)_4\big) \wedge (V_1 \leq V_4)\big).$$

Also note that, compared to our first encoding $\boldsymbol{S}_\Pi(V_1)$, we do not have a pendant to subformula $\boldsymbol{S}^2$ here, which reduces simply to $Q^{\mathcal{V}}_{3,1,1}$ due to the new renaming schema.

Lemma 2. *Let P, Q be programs over V, and let $A, B, Y \subseteq V$. Then, Y is a partial spoiler for $\Pi = (P, Q, \mathcal{P}_A, \subseteq_B)$ iff $Y_1 \models \boldsymbol{T}_\Pi(V_1)$.*

Example 4. Consider the two programs $P = \{a \vee b \leftarrow c\}$ and $Q = \{a \leftarrow c, not\, b\}$, $A = \{a\}$, and $B = \{b\}$. The encodings for the problem $\Pi = (P, Q, \mathcal{P}_A, \subseteq_B)$ are as follows:

$$
\begin{aligned}
\boldsymbol{S}_\Pi(V_1) &= (c_1 \rightarrow a_1 \vee b_1) \wedge \boldsymbol{S}^1(P, A) \wedge \\
&\quad \forall a_3 b_3 c_3 (\boldsymbol{S}^2(Q, A, B) \rightarrow \boldsymbol{S}^3(P, Q, A)), \\
\boldsymbol{S}^1(P, A) &= \forall a_2 b_2 c_2 \big((a_2 \leftrightarrow a_1) \wedge (\{b_2, c_2\} < \{b_1, c_1\}) \rightarrow \neg(c_2 \rightarrow a_2 \vee b_2)\big), \\
\boldsymbol{S}^2(Q, A, B) &= (a_3 \leftrightarrow a_1) \wedge (b_3 \leftrightarrow b_1) \wedge (c_3 \wedge \neg b_3 \rightarrow a_3), \\
\boldsymbol{S}^3(P, Q, A) &= \exists a_4 b_4 c_4 \big((\{a_4, b_4, c_4\} < \{a_3, b_3, c_3\}) \wedge (c_4 \wedge \neg b_3 \rightarrow a_4) \wedge \\
&\qquad ((\{a_4\} < \{a_1\}) \rightarrow \forall a_5 b_5 c_5 ((a_5 \leftrightarrow a_4) \wedge \\
&\qquad (\{a_5, b_5, c_5\} \leq \{a_1, b_1, c_1\}) \rightarrow \neg(c_5 \rightarrow a_5 \vee b_5)))\big); \\
\boldsymbol{T}_\Pi(V_1) &= (c_1 \rightarrow a_1 \vee b_1) \wedge \boldsymbol{T}^1(P, A, \mathcal{V}) \wedge \\
&\quad \forall c_3 \big((c_3 \wedge \neg b_1 \rightarrow a_1) \rightarrow \boldsymbol{T}^3(P, Q, A, \mathcal{V})\big), \\
\boldsymbol{T}^1(P, A, \mathcal{V}) &= \forall b_2 c_2 ((\{b_2, c_2\} < \{b_1, c_1\}) \rightarrow \neg(c_2 \rightarrow a_1 \vee b_2), \\
\boldsymbol{T}^3(P, Q, A, \mathcal{V}) &= \exists a_4 b_4 c_4 \big((\{a_4, b_4, c_4\} < \{a_1, b_1, c_3\}) \wedge (c_4 \wedge \neg b_1 \rightarrow a_4) \wedge \\
&\qquad ((\{a_4\} < \{a_1\}) \rightarrow \forall b_5 c_5 ((\{b_5, c_5\} \leq \{b_1, c_1\}) \rightarrow \\
&\qquad \neg(c_5 \rightarrow a_4 \vee b_5)))\big).
\end{aligned}
$$

As mentioned before, the optimised encoding $\boldsymbol{T}_\Pi(\cdot)$ saves "fixed assignments", like $(a_2 \leftrightarrow a_1)$, which occur in $\boldsymbol{S}_\Pi(\cdot)$, by employing the advanced renaming schema in such a way that, instead of atom a_2, atom a_1 is used in the encoding. One effect of this refinement is the decrease of universally quantified atoms. □

Theorem 1. *For any inclusion problem $\Pi = (P, Q, \mathcal{P}_A, \subseteq_B)$, the following statements are equivalent: (i) Π holds; (ii) $\neg\exists V_1 \boldsymbol{S}_\Pi(V_1)$ is valid; and (iii) $\neg\exists V_1 \boldsymbol{T}_\Pi(V_1)$ is valid.*

Corollary 1. *Let $\Pi = (P, Q, \mathcal{P}_A, =_B)$ be an equivalence problem. Then, for $\Pi' = (P, Q, \mathcal{P}_A, \subseteq_B)$ and $\Pi'' = (Q, P, \mathcal{P}_A, \subseteq_B)$, the following statements are equivalent: (i) Π holds; (ii) $\neg\exists V_1 \boldsymbol{S}_\Pi\ (V_1) \wedge \neg\exists V_1 \boldsymbol{S}_\Pi\ (V_1)$ is valid; and (iii) $\neg\exists V_1 \boldsymbol{T}_\Pi\ (V_1) \wedge \neg\exists V_1 \boldsymbol{T}_\Pi\ (V_1)$ is valid.*

4.3 Applicability and Adequacy of the Encodings

In order to employ off-the-shelves QBF-solvers for deciding answer-set correspondence, we have to transform above encodings into prenex normal form. The propositional part of these prenex QBFs additionally has to be reduced to CNF, which can be accomplished by usual techniques. We thus focus here just on possible prenex normal forms of our encodings.

Recall that there are several ways to transform a QBF into prenex normal form. For our encodings, the situation is as follows. Take, e.g., the existential closure of $\boldsymbol{S}_\Pi(V_1)$, given by $\exists V_1 \boldsymbol{S}_\Pi(V_1)$: for this closed QBF, different prenex forms can be obtained, e.g.,

$$\exists V_1 \forall (V_2 \cup V_3) \exists V_4 \forall V_5\, \phi \quad \text{or} \quad \exists V_1 \forall V_3 \exists V_4 \forall (V_5 \cup V_2)\, \phi,$$

where ϕ represents the so-called *propositional skeleton* of the QBF $\boldsymbol{S}_\Pi(V_1)$ (cf. [8]), which, roughly speaking, results from $\boldsymbol{S}_\Pi(V_1)$ by deleting all quantifiers. For later purposes, we use in the following the second variant, and define $\boldsymbol{S}^{\mathsf{p}}_\Pi := \exists V_1 \forall V_3 \exists V_4 \forall (V_5 \cup V_2)\, \phi$. Likewise, we use $\boldsymbol{T}^{\mathsf{p}}_\Pi := \exists V_1 \forall V_3^{A \cup B} \exists V_4 \forall (V_5^A \cup V_2^A)\, \psi$ as a prenex form for $\exists V_1 \boldsymbol{T}_\Pi(V_1)$, where ψ is the propositional skeleton of $\boldsymbol{T}_\Pi(V_1)$.

Theorem 2. *For any inclusion problem $\Pi = (P, Q, \mathcal{P}_A, \subseteq_B)$, the following statements are equivalent: (i) Π holds; (ii) $\neg \boldsymbol{S}^{\mathsf{p}}_\Pi$ is valid; (iii) $\neg \boldsymbol{T}^{\mathsf{p}}_\Pi$ is valid.*

These prenex forms also give evidence that our encodings are *adequate* in a certain theoretical sense: Following [3], given decision problems $\mathcal{D} \subseteq \mathcal{L}$ and $\mathcal{D}' \subseteq \mathcal{L}'$ in languages $\mathcal{L}$ and $\mathcal{L}'$, respectively, we call an encoding $f : \mathcal{L} \to \mathcal{L}'$ *adequate* iff, for each $s \in \mathcal{L}$, (i) $s \in \mathcal{D}$ iff $f(s) \in \mathcal{D}'$, (ii) $f(s)$ is constructible in polynomial time from s, and (iii) deciding whether $f(s) \in \mathcal{D}'$ is not computationally harder than deciding whether $s \in \mathcal{D}$.

Now, both $\neg \boldsymbol{S}^{\mathsf{p}}_\Pi$ and $\neg \boldsymbol{T}^{\mathsf{p}}_\Pi$ obviously satisfy these conditions, for every inclusion problem Π. Indeed, by the above theorem, we have that Π holds iff $\neg \boldsymbol{S}^{\mathsf{p}}_\Pi$ is valid. Moreover, $\neg \boldsymbol{S}^{\mathsf{p}}_\Pi$ is computable in polynomial time (indeed, in linear time) in the size of Π (as easily verified from the definitions). Finally, $\neg \boldsymbol{S}^{\mathsf{p}}_\Pi$ can be transformed into a $(4, \forall)$-QBF in polynomial time. Hence, Proposition 6 implies that determining the truth of $\neg \boldsymbol{S}^{\mathsf{p}}_\Pi$ is thus in the same complexity class (viz. Π_4^P) as the encoded problem. All these properties hold for $\neg \boldsymbol{T}^{\mathsf{p}}_\Pi$ as well. This proves the adequacy of our encodings.

5 Obtaining Counterexamples

In this section, we provide a theoretical basis how to use our encodings to obtain counterexamples for an inclusion problem $(P, Q, \mathcal{P}_A, \subseteq_B)$. To this end, we use the concept of *policies* for prenex QBFs, along the lines of Coste-Marquis *et al.* [4].

Definition 6. *The set $P(k, \mathsf{Q}, X_k, \ldots, X_1)$ of* policies *for a (k, Q)-QBF of the form $\mathsf{Q}_k X_k \ldots \mathsf{Q}_1 X_1 \phi$ is inductively defined as follows:*

1. $P(0, \mathsf{Q}) = \{\lambda\}$,
2. $P(k, \exists, X_k, \ldots, X_1) = \{(I, \pi) \mid I \subseteq X_k,\ \pi \in P(k-1, \forall, X_{k-1}, \ldots, X_1)\}$, *and*
3. $P(k, \forall, X_k, \ldots, X_1) = \{\pi \mid \pi : 2^{X_k} \to P(k-1, \exists, X_{k-1}, \ldots, X_1)\}$,

where λ represents the empty policy.

Note that policies for $(k, \exists)$-QBFs are pairs (I, π), where I is an interpretation over atoms from the outermost group of quantifiers and π is a policy itself, whereas policies for $(k, \forall)$-QBFs are functions assigning to each interpretation over atoms from the outermost group of quantifiers a policy.

Definition 7. *A (k, Q)-QBF $\Phi = \mathsf{Q}_k X_k \ldots \mathsf{Q}_1 X_1 \phi$ is* satisfied *by a policy π (for Φ) iff one the following conditions applies (inductively):*

1. *$k = 0$, $\pi = \lambda$, and ϕ is true,*
2. *$k > 0$, $\mathsf{Q} = \exists$, $\pi = (I, \pi')$, and $\forall X_{k-1} \ldots \mathsf{Q}_1 X_1 \phi[X_k / I]$ is satisfied by π',*
3. *$k > 0$, $\mathsf{Q} = \forall$, and for any $I \subseteq X_k$, $\exists X_{k-1} \ldots \mathsf{Q}_1 X_1 \phi[X_k / I]$ is satisfied by $\pi(I)$.*

Denote by $SP(\Phi)$ the set of satisfying policies for a prenex QBF Φ.

Proposition 10. *A prenex QBF Φ is valid iff $SP(\Phi) \neq \emptyset$.*

Example 5. Consider $\phi = (p \rightarrow q) \wedge (q \rightarrow p)$ and the following QBFs:[2]

$$\Phi_1 = \exists pq\, \phi, \quad \Phi_2 = \forall pq\, \phi, \quad \Phi_3 = \exists p \forall q\, \phi, \quad \text{and} \quad \Phi_4 = \forall p \exists q\, \phi.$$

The set of policies for Φ_1 is given by $\{(I, \lambda) \mid I \subseteq \{p, q\}\}$, i.e., the satisfying policies for Φ_1 are in a one-to-one correspondence to the models of ϕ, and are given by $(\emptyset, \lambda)$ and $(\{p, q\}, \lambda)$. For Φ_2, the only policy is the function π assigning to each $I \subseteq \{p, q\}$ the empty policy λ. Note that π is not satisfying Φ_2 since, for instance, with $I = \{p\}$, we get $\pi(I) = \lambda$, but $\phi[\{p, q\}/I] = (\top \rightarrow \bot) \wedge (\bot \rightarrow \top)$ is not true. For Φ_3, we get as policies $\pi_1 = (\{p\}; \pi')$ and $\pi_2 = (\emptyset; \pi')$, where π' is defined as $\pi'(\{q\}) = \pi'(\emptyset) = \lambda$. It can be shown that neither π_1 nor π_2 satisfy Φ_3, by similar arguments as for the case of Φ_2. Finally, Φ_4 yields four policies, given as follows:

$$\begin{array}{llll} \pi(p) = (q, \lambda), & \pi(\emptyset) = (q, \lambda); & \pi'(p) = (q, \lambda), & \pi'(\emptyset) = (\emptyset, \lambda); \\ \pi''(p) = (\emptyset, \lambda), & \pi''(\emptyset) = (q, \lambda); & \pi'''(p) = (\emptyset, \lambda), & \pi'''(\emptyset) = (\emptyset, \lambda). \end{array}$$

One can verify that π' is the only satisfying policy for Φ_4. □

We now use the concept of policies to obtain the counterexamples from the satisfying policies of our encodings. Note that, in the definition below, we make use of our renaming schema as used in the encodings; e.g., $Z_3 = \{z_3 \mid z \in Z\}$.

Definition 8. *Let $\Pi = (P, Q, \mathcal{P}_A, \subseteq_B)$ be an inclusion problem. Furthermore, let $\boldsymbol{S}^{\mathsf{p}}_{\Pi}$ and $\boldsymbol{T}^{\mathsf{p}}_{\Pi}$ be as in Subsection 4.3 and $\Omega \in \{\boldsymbol{S}, \boldsymbol{T}\}$. Then,*

$$\sigma(\Omega, \Pi) := \{(Y, \Sigma_{\Omega, Y, \pi}) \mid (Y_1, \pi) \in SP(\Omega^{\mathsf{p}}_{\Pi})\},$$

where

$$\begin{array}{rl} \Sigma_{\boldsymbol{S}, Y, \pi} := \{(X, Z), (Z, Z) \mid Z =_{A \cup B} Y, (Z, Z) \in SE^A(Q), & \\ \qquad \pi(Z_3) = (X_4, \pi'), \textit{ for some } \pi'\} \quad \textit{and} & \\ \Sigma_{\boldsymbol{T}, Y, \pi} := \{(X, Y \dot{+} Z), (Y \dot{+} Z, Y \dot{+} Z) \mid (Y \dot{+} Z, Y \dot{+} Z) \in SE^A(Q), & \\ \qquad \pi(Z_3) = (X_4, \pi'), \textit{ for some } \pi'\}, & \end{array}$$

and $Y \dot{+} Z$ stands for $Y|_{A \cup B} \cup Z$.

These two projections, $\sigma(\boldsymbol{S}, \cdot)$ and $\sigma(\boldsymbol{T}, \cdot)$, on the satisfying policies for our two encodings are actually identical. Hence, our final two results in this section apply to both encodings.

Theorem 3. *Let $\Pi = (P, Q, \mathcal{P}_A, \subseteq_B)$ be an inclusion problem and $\Omega \in \{\boldsymbol{S}, \boldsymbol{T}\}$. Then, each $(Y, \Sigma) \in \sigma(\Omega, \Pi)$ is a spoiler for Π.*

In view of the construction of Proposition 2, we can thus construct counterexamples directly from the satisfying policies of our encodings.

[2] In what follows, we sometimes omit brackets "{" and "}" for ease of notation.

Corollary 2. *Let* $\Pi = (P, Q, \mathcal{P}_A, \subseteq_B)$ *be an inclusion problem and* $\Omega \in \{\boldsymbol{S}, \boldsymbol{T}\}$. *Then, each* $(Y, \Sigma) \in \sigma(\Omega, \Pi)$ *induces a counterexample* $(Y, P_{\Sigma,A})$ *for* Π.

From Proposition 10 and Theorem 2, in turn, we obtain that in case no satisfying policy for our encodings exists, the considered inclusion problem holds, and therefore does not possess any counterexample.

6 Special Cases

Finally, we analyse our encodings in the light of special instantiations of correspondence problems and give pointers to related work.

In what follows, for every equivalence problem $\Pi = (P, Q, \mathcal{P}_A, =_B)$, let $\Pi' = (P, Q, \mathcal{P}_A, \subseteq_B)$ and $\Pi'' = (Q, P, \mathcal{P}_A, \subseteq_B)$ be the associated inclusion problems (see also Corollary 1).

In case of *strong equivalence* [20], i.e., for problems of form $\mathit{\Pi} = (P, Q, \mathcal{P}_A, =_A)$ with $A = \mathcal{U}$, the encodings $\boldsymbol{T}_\Pi\ (V_1)$ and $\boldsymbol{T}_\Pi\ (V_1)$, as defined in Definition 5, can be drastically simplified since $V_2^A = V_3^A = V_5^A = \emptyset$. In particular, $\boldsymbol{T}_\Pi\ (V_1)$ is equivalent to

$$P_{1,1} \wedge \big(Q_{1,1} \rightarrow \exists V_4\big((V_4 < V_1) \wedge Q_{4,1} \wedge \neg P_{4,1}\big)\big).$$

Note that the composed encoding for deciding strong equivalence, i.e., the closed QBF $\neg\exists V_1 \boldsymbol{T}_\Pi\ (V_1) \wedge \neg\exists V_1 \boldsymbol{T}_\Pi\ (V_1)$, amounts to a propositional unsatisfiability test, witnessing the coNP-completeness complexity for checking strong equivalence [24]. One can show that the reductions due to Pearce *et al.* [24] and Lin [21] for testing strong equivalence in terms of propositional logic are simple variants thereof.

For strong equivalence *relative* to a set A of atoms [28], i.e., for $\mathit{\Pi}$ being of form $(P, Q, \mathcal{P}_A, =_B)$ with $B = \mathcal{U}$ but with arbitrary A, our encodings $\boldsymbol{T}_\Pi\ (V_1)$ and $\boldsymbol{T}_\Pi\ (V_1)$ can still be simplified since $V_3^{A \cup B} = \emptyset$. Indeed, $\boldsymbol{T}^{\mathsf{p}}_\Pi$ and $\boldsymbol{T}^{\mathsf{p}}_\Pi$ are then $(2, \exists)$-QBFs, reflecting the complexity of strong equivalence relative to A, which is on the second level of the polynomial hierarchy [28].

Next, we address the case of *bounded* relativised strong equivalence, as investigated by Eiter *et al.* [11]. This notion applies to problems of form $\Pi = (P, Q, \mathcal{P}_A, =)$, where the cardinality of $(\mathcal{U} \setminus A)$, i.e., the number of atoms missing in A, is bounded by a constant. Hereby, the sets V_2^A and V_5^A, which build the only universal quantifiers in the encoding $\boldsymbol{T}_{\mathit{\Pi}}\ (V_1)$ for relativised strong equivalence, are sets of a fixed size. Hence, we can eliminate these quantifiers according to the semantics and still get an adequate encoding for this particular notion of equivalence. Consequently, bounded relativised strong equivalence can be checked with a polynomial unsatisfiability test, once again reflecting the coNP-complexity of this problem [11].

Finally, we address the case of ordinary equivalence, i.e., considering problems of form $\Pi = (P, Q, \mathcal{P}_A, =)$ with $A = \emptyset$, which is well known to be $\mathit{\Pi}_2^P$-complete [23]. Here, the encoding $\boldsymbol{S}_\Pi\ (V_1)$ from Definition 4 can be simplified as follows:

$$P_{1,1} \wedge \forall V_2\big((V_2 < V_1) \rightarrow \neg P_{2,1}\big) \wedge \big(Q_{1,1} \rightarrow \exists V_4((V_4 < V_1) \wedge Q_{4,1})\big).$$

One can observe that this encoding is related to encodings for computing stable models via QBFs, as discussed by Egly *et al.* [6] and Pearce *et al.* [24]. Indeed, taking the two

main conjuncts from $\boldsymbol{S}_{\Pi}\ (V_1)$, $\Phi = P_{1,1} \wedge \forall V_2\big((V_2 < V_1) \rightarrow \neg P_{2,1}\big)$ and $\Psi = Q_{1,1} \rightarrow \exists V_4\big((V_4 < V_1) \wedge Q_{4,1})\big)$, we get, for any assignment $Y_1 \subseteq V_1$, $Y_1 \models \Phi$ iff Y is an answer set of P, and $Y_1 \models \Psi$ iff Y is not an answer set of Q. Note that once more the encodings reflect the inherent complexity of the reduced equivalence checking task, viz. the Π_2^P-completeness for ordinary equivalence in this case.

7 Conclusion

In this paper, we discussed a novel decision procedure for advanced program comparison in answer-set programming (ASP) via encodings into quantified propositional logic. This approach was motivated by the high computational complexity we have to face for this task, making a direct realisation via ASP hard to accomplish. Furthermore, we showed how to obtain counterexamples from policies, which satisfy these encodings, and discussed special instances of the considered correspondence problems. Since currently practicably efficient solvers for quantified propositional logic are available, they can be used as back-end inference engines to compute the correspondence problems under consideration using the proposed encodings. Moreover, since these correspondence problems are one of the few natural problems lying above the second level of the polynomial hierarchy, yet still part of the polynomial hierarchy, we believe that our encodings also provide valuable benchmarks for evaluating QBF-solvers, for which there is currently a lack of structured problems with more than one quantifier alternation (see [17,16]).

References

1. O. Arieli. Paraconsistent Preferential Reasoning by Signed Quantified Boolean Formulae. In *Proc. ECAI'04*, pages 773–777. IOS Press, 2004.
2. O. Arieli and M. Denecker. Reducing Preferential Paraconsistent Reasoning to Classical Entailment. *Journal of Logic and Computation*, 13(4):557–580, 2003.
3. P. Besnard, T. Schaub, H. Tompits, and S. Woltran. Representing Paraconsistent Reasoning via Quantified Propositional Logic. In *Inconsistency Tolerance*, volume 3300 of *LNCS*, pages 84–118. Springer, 2005.
4. S. Coste-Marquis, H. Fargier, J. Lang, D. Le Berre, and P. Marquis. Function Problems for Quantified Boolean Formulas. Technical Report 2003-15-R, Institut de Recherche en Informatique de Toulouse (IRIT), 2003. Available at `http://www.cril.univ-artois.fr/asqbf/pub/files/qbfeng7.pdf`.
5. J. Delgrande, T. Schaub, H. Tompits, and S. Woltran. On Computing Solutions to Belief Change Scenarios. *Journal of Logic and Computation*, 14(6):801–826, 2004.
6. U. Egly, T. Eiter, H. Tompits, and S. Woltran. Solving Advanced Reasoning Tasks using Quantified Boolean Formulas. In *Proc. AAAI'00*, pages 417–422. AAAI Press/MIT Press, 2000.
7. U. Egly, R. Pichler, and S. Woltran. On Deciding Subsumption Problems. *Annals of Mathematics and Artificial Intelligence*, 43(1–4):255–294, 2005.
8. U. Egly, M. Seidl, H. Tompits, S. Woltran, and M. Zolda. Comparing Different Prenexing Strategies for Quantified Boolean Formulas. In *Proc. SAT'03, Selected Revised Papers*, volume 2919 of *LNCS*, pages 214–228. Springer, 2004.

9. T. Eiter, W. Faber, M. Fink, G. Pfeifer, and S. Woltran. Complexity of Answer Set Checking and Bounded Predicate Arities for Non-ground Answer Set Programming. In *Proc. KR'04*, pages 377–387. AAAI Press, 2004.
10. T. Eiter and M. Fink. Uniform Equivalence of Logic Programs under the Stable Model Semantics. In *Proc. ICLP'03*, number 2916 in LNCS, pages 224–238. Springer, 2003.
11. T. Eiter, M. Fink, and S. Woltran. Semantical Characterizations and Complexity of Equivalences in Answer Set Programming. Technical Report INFSYS RR-1843-05-01, Institut für Informationssysteme, Technische Universität Wien, Austria, 2005. To appear in *ACM Transactions on Computational Logic*.
12. T. Eiter, V. Klotz, H. Tompits, and S. Woltran. Modal Nonmonotonic Logics Revisited: Efficient Encodings for the Basic Reasoning Tasks. In *Proc. TABLEAUX'02*, volume 2381 of *LNCS*, pages 100–114. Springer, 2002.
13. T. Eiter, H. Tompits, and S. Woltran. On Solution Correspondences in Answer Set Programming. In *Proc. IJCAI'05*, 2005.
14. M. Gelfond and V. Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, 9:365–385, 1991.
15. E. Giunchiglia, M. Narizzano, and A. Tacchella. Backjumping for Quantified Boolean Logic Satisfiability. *Artificial Intelligence*, 145:99–120, 2003.
16. D. Le Berre, M. Narizzano, L. Simon, and A. Tacchella. The Second QBF Solvers Comparative Evaluation, 2004. Available at `http://www.qbflib.org/`.
17. D. Le Berre, L. Simon, and A. Tacchella. Challenges in the QBF Arena: the SAT'03 Evaluation of QBF Solvers. In *Proc. SAT'03, Selected Revised Papers*, volume 2919 of *LNCS*, pages 468–485. Springer, 2004.
18. N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. The DLV System for Knowledge Representation and Reasoning. Technical Report cs.AI/0211004, arXiv.org. To appear in *ACM Transactions on Computational Logic*.
19. R. Letz. Lemma and Model Caching in Decision Procedures for Quantified Boolean Formulas. In *Proc. TABLEAUX'02*, volume 2381 of *LNCS*, pages 160–175. Springer, 2002.
20. V. Lifschitz, D. Pearce, and A. Valverde. Strongly Equivalent Logic Programs. *ACM Transactions on Computational Logic*, 2(4):526–541, 2001.
21. F. Lin. Reducing Strong Equivalence of Logic Programs to Entailment in Classical Propositional Logic. In *Proc. KR'02*, pages 170–176. Morgan Kaufmann, 2002.
22. F. Lin and Y. Zhao. ASSAT: Computing Answer Sets of a Logic Program by SAT Solvers. In *Proc. AAAI'02*, pages 112–117. AAAI Press / MIT Press, 2002.
23. E. Oikarinen and T. Janhunen. Verifying the Equivalence of Logic Programs in the Disjunctive Case. In *Proc. LPNMR'04*, volume 2923 of *LNCS*, pages 180–193. Springer, 2004.
24. D. Pearce, H. Tompits, and S. Woltran. Encodings for Equilibrium Logic and Logic Programs with Nested Expressions. In *Proc. EPIA'01*, volume 2258 of *LNCS*, pages 306–320. Springer, 2001.
25. J. Rintanen. Constructing Conditional Plans by a Theorem Prover. *Journal of Artificial Intelligence Research*, 10:323–352, 1999.
26. P. Simons, I. Niemelä, and T. Soininen. Extending and Implementing the Stable Model Semantics. *Artificial Intelligence*, 138:181–234, 2002.
27. H. Turner. Strong Equivalence Made Easy: Nested Expressions and Weight Constraints. *Theory and Practice of Logic Programming*, 3(4-5):602–622, 2003.
28. S. Woltran. Characterizations for Relativized Notions of Equivalence in Answer Set Programming. In *Proc. JELIA'04*, volume 3229 of *LNCS*, pages 161–173. Springer, 2004.

Hybrid Probabilistic Logic Programs with Non-monotonic Negation

Emad Saad and Enrico Pontelli

Department of Computer Science, New Mexico State University
{emsaad, epontell}@cs.nmsu.edu

Abstract. In [22], a new Hybrid Probabilistic Logic Programs framework has been proposed, and a new semantics has been developed to enable encoding and reasoning about real-world applications. In this paper, the language of Hybrid Probabilistic Logic Programs framework of [22] is extended to allow non-monotonic negation, and two alternative semantics are defined: stable probabilistic model semantics and probabilistic well-founded semantics. Stable probabilistic model semantics and probabilistic well-founded semantics generalize stable model semantics and well-founded semantics of traditional normal logic programs, and they reduce to the semantics of original Hybrid Probabilistic Logic programs framework of [22] for programs without negation. It is the first time that two different semantics for Hybrid Probabilistic Programs with non-monotonic negation as well as their relationships are described. This development provides a foundational ground for developing computational methods for computing the proposed semantics. Furthermore, it makes it clearer how to characterize non-monotonic negation in probabilistic logic programming frameworks for commonsense reasoning.

1 Introduction

Hybrid Probabilistic Programs (HPP) [5] is a probabilistic logic programming framework that enables the user to *explicitly* encode his/her knowledge about the type of dependencies existing between the probabilistic events being described by the programs. HPP generalizes the *probabilistic annotated logic programming framework*, originally proposed in [18] and further extended in [19]. Since the aim of probabilistic logic programming in general, and of the HPP framework in particular, is to allow reasoning and decision making under probabilistic and statistical knowledge, a generalization and a new semantics for HPP have been defined in [22]. The idea in [22] comes upon observing that commonsense reasoning about probabilities relies on how *likely* (probable) are the various events to occur, rather than how precise our knowledge about these probabilities is. The generalization includes adding the ability to encode the user's knowledge about how to combine the probabilities of the same event derived from different rules in HPP. In addition, the new semantics, intuitively, captures the probabilistic reasoning according to how likely are the various events to occur, by employing the

M. Gabbrielli and G. Gupta (Eds.): ICLP 2005, LNCS 3668, pp. 204–220, 2005.

truth order instead of the knowledge order [5]. It was shown that the modified HPP framework is more suitable for reasoning and decision making tasks, including those arising from probabilistic planning. In addition, it was shown that the new HPP framework subsumes Lakshmanan and Sadri's [11] probabilistic implication-based framework as well as it is a natural extension of classical logic programming.

It is known that non-monotonic negation is vital to capture the principles of commonsense reasoning [1]. Moreover, it is important to provide the ability to derive negative conclusions in the absence of positive information [20]. Therefore, it is essential to extend the different probabilistic logic programming frameworks to deal with non-monotonic negation. In this view, the probabilistic logic programming framework in [19] was extended in [20] to allow this important feature by developing a semantics based on a notion of stable models [7]. However, the stable model semantics extension in [20] is computationally expensive [15], since at every fixpoint iteration an exponential number of linear programs, each having an exponential number of variables, needs to be solved [15]. The main reason for this computational complexity arises from the fact that [20] allows annotated conjunctions and disjunctions to appear as heads of the rules. Also, it is worth noting that knowledge order was used in defining the stable model semantics in [20] as well as a fixed assumption (ignorance) is postulated among the dependencies of the various events encoded by the logic programs.

In [13,14], a well-founded like semantics [8] extension to the probabilistic logic programming framework of [11] was introduced along with well-founded like semantics [8] extension to various non-probabilistic logic programming frameworks with uncertainty [12]. Although the notion of non-monotonic negation in [20] is more natural and closer to the classical notion of non-monotonic negation, the notion of non-monotonic negation in [13,14] is closer to classical negation (since $Prob(\neg A) = 1 - Prob(A)$). An alternating fixpoint semantics [8] was introduced in [13] to describe the well-founded like semantics. It must be noted that no declarative account was given for the well-founded semantics [9] of the probabilistic logic programming in [13,14], due to the way non-monotonic negation is interpreted, which has an operational nature. This interpretation of non-monotonic negation makes it less natural to define stable model like semantics [7] and well-founded like semantics [9]. In [14], a framework to approximate the well-founded semantics for [12], including the probabilistic logic programming framework of [11], was described. The approximate well-founded semantics is based on the idea that uncertainty values are assigned as approximations to atoms of the Herbrand base, in the form of intervals, where the certainty values of the atoms lie within these intervals. However, it is not clear how the approximate well-founded semantics extension [14] works for the probabilistic logic programming framework of [11], where probabilities are originally represented as intervals. To this end, it seems that it is not feasible to define a natural notion of non-monotonic negation as well as stable model semantics [7] extension for the probabilistic logic programming framework in [11].

In this paper, we extend the language of Hybrid Probabilistic Logic Programs of [22], which allows multiple modes of probabilistic combinations and employs the truth order, to support non-monotonic negation, by considering only annotated atoms as heads of rules. This is to avoid the computational complexity inherited from allowing annotated conjunctions or disjunctions to appear as heads of rules. In [22], we have shown that it is possible to develop an algorithm to compute the least fixpoint of HPP without negation and with only annotated atoms as heads of rules in worst-case time complexity $O(n^2)$, where n is the size of a program. In addition, we define two alternative semantics for the extended language; the stable probabilistic model semantics and the probabilistic well-founded semantics and study their relationships. We show that the stable probabilistic model semantics and the probabilistic well-founded semantics generalize the stable model semantics [7] and the well-founded semantics [9] for normal logic programs, and they reduce to the semantics of HPP [22] in the absence of non-monotonic negation. An important result is that the relationship between the stable probabilistic model semantics and the probabilistic well-founded semantics *preserves* the relationship between the stable model semantics and the well-founded semantics for normal logic programs [9].

Another reason why these proposed semantics are interesting is that they provide a foundational ground for building algorithms and systems for computing the meaning of HPP with non-monotonic negation based on the stable probabilistic model semantics and the probabilistic well-founded semantics. The fact that these proposed semantics naturally generalize their classical counterparts suggests that efficient algorithms and implementations can be developed by *extending* the existing efficient algorithms and implementations developed for the stable models and the well-founded semantics for normal logic programs, such as SMODELS [17]. To show this point, an algorithm for computing the least fixpoint for HPP is described in [22], that extends Dowling-Gallier algorithm for computing the satisfiability of a set of Horn formulae [6], which is the ground base for developing the various auxiliary functions in SMODELS (preliminary design of these algorithms has been presented in [21]).

2 Hybrid Probabilistic Programs

In the following subsections, we present the syntax of the proposed Hybrid Probabilistic Programs with non-monotonic negation. We also review the basic syntax, as presented in [5,22], and the semantics, as described in [22], of Hybrid Probabilistic Programs without negation.

2.1 Probabilistic Strategies

Let $C[0,1]$ denotes the set of all closed intervals in $[0,1]$. In the context of HPP, probabilities are assigned to primitive events (atoms) and compound events (conjunctions or disjunctions of atoms) as intervals in $C[0,1]$. Let $[a_1,b_1],[a_2,b_2] \in C[0,1]$. Then the *truth order* asserts that $[a_1,b_1] \leq_t [a_2,b_2]$ iff $a_1 \leq a_2$ and $b_1 \leq b_2$. The *knowledge order* states that $[a_1,b_1] \leq_k [a_2,b_2]$ iff $[a_2,b_2] \subseteq [a_1,b_1]$.

The set $C[0,1]$ and the relation $\leq_t$ form a complete lattice. In particular, the join ($\oplus_t$) operation is defined as $[a_1,b_1] \oplus_t [a_2,b_2] = [max\{a_1,a_2\}, max\{b_1,b_2\}]$ and the meet ($\otimes_t$) is defined as $[a_1,b_1] \otimes_t [a_2,b_2] = [min\{a_1,a_2\}, min\{b_1,b_2\}]$ w.r.t. $\leq_t$. The type of dependency among the primitive events within a compound event is described by *probabilistic strategies*, which are explicitly selected by the user. We call ρ, a pair of functions $\langle c, md\rangle$, a probabilistic strategy (p-strategy), where $c : C[0,1] \times C[0,1] \rightarrow C[0,1]$, the *probabilistic composition function*, which is *commutative*, *associative*, *monotonic* w.r.t. $\leq_t$, and meets the following *separation* criteria: there are two functions c_1, c_2 such that $c([a_1,b_1],[a_2,b_2]) = [c_1(a_1,a_2), c_2(b_1,b_2)]$. Whereas, $md : C[0,1] \rightarrow C[0,1]$ is the *maximal interval function*. The maximal interval function md of a certain p-strategy returns an estimate of the probability range of a primitive event, A, from the probability range of a compound event that contains A. The composition function c returns the probability range of a conjunction (disjunction) of two events given the ranges of its constituents. For convenience, given a multiset of probability intervals $M = \{\!\{[a_1,b_1],\ldots,[a_n,b_n]\}\!\}$, we use cM to denote $c([a_1,b_1], c([a_2,b_2],\ldots,c([a_{n-1},b_{n-1}],[a_n,b_n]))\ldots)$. According to the type of combination among events, p-strategies are classified into *conjunctive* p-strategies and *disjunctive* p-strategies. Conjunctive (disjunctive) p-strategies are employed to compose events belonging to a conjunctive (disjunctive) formula (please see [5,22] for the formal definitions).

2.2 Language Syntax

In this subsection, we describe the syntax of Hybrid Probabilistic Programs and define a syntax for Hybrid Probabilistic Programs with non-monotonic negation. Let L be an arbitrary first-order language with finitely many predicate symbols, constants, and infinitely many variables. Function symbols are disallowed. In addition, let $S = S_{conj} \cup S_{disj}$ be an arbitrary set of p-strategies, where S_{conj} (S_{disj}) is the set of all conjunctive (disjunctive) p-strategies in S. The Herbrand base of L is denoted by B_L. An *annotation* denotes a probability interval and it is represented by $[\alpha_1, \alpha_2]$, where α_1, α_2 are called annotation items. An *annotation item* is either a constant in $[0,1]$, a variable (*annotation variable*) ranging over $[0,1]$, or $f(\alpha_1,\ldots,\alpha_n)$ (called *annotation function*) where f is a representation of a computable total function $f : ([0,1])^n \rightarrow [0,1]$ and $\alpha_1,\ldots,\alpha_n$ are annotation items. The building blocks of the language of HPP are *hybrid basic formulae*. Let us consider a collection of atoms $A_1,\ldots,A_n$, a conjunctive p-strategy ρ, and a disjunctive p-strategy ρ'. Then $A_1 \wedge_\rho \ldots \wedge_\rho A_n$ and $A_1 \vee_\rho \ldots \vee_\rho A_n$ are called *hybrid basic formulae*, and $bf_S(B_L)$ is the set of all ground hybrid basic formulae formed using distinct atoms from B_L and p-strategies from S. An annotated hybrid basic formula is an expression of the form $F : \mu$ where F is a hybrid basic formula and μ is an annotation. A *hybrid literal* is an annotated hybrid basic formula $F : \mu$ (positive annotated hybrid basic formula or positive hybrid literal) or the negation of an annotated hybrid basic formula *not* $(F : \mu)$ (negative annotated hybrid basic formula or negative hybrid literal).

Definition 1 (Rules). *A normal hybrid probabilistic rule (nh-rule) is an expression of the form*

$$A : \mu \leftarrow F_1 : \mu_1, \ldots, F_n : \mu_n, not\ (G_1 : \mu_{n+1}), \ldots, not\ (G_m : \mu_{n+m})$$

where A is an atom, $F_1, \ldots, F_n, G_1, \ldots, G_m$ are hybrid basic formulae, and μ, μ_i $(1 \leq i \leq m+n)$ are annotations.

A hybrid probabilistic rule (h-rule) is a nh-rule where $m = 0$—i.e., there are no negative hybrid literals.

The intuitive meaning of a nh-rule, in Definition 1, is that, if for each $F_i : \mu_i$, the probability interval of F_i is at least μ_i and for each *not* $(G_j : \mu_j)$, it is not *provable* that the probability interval of G_j is at least μ_j, then the probability interval of A is μ.

Definition 2 (Programs). *A normal hybrid probabilistic program over S (*nh-program*) is a pair $P = \langle R, \tau \rangle$, where R is a finite set of nh-rules with p-strategies from S, and τ is a mapping $\tau : B_L \rightarrow S_{disj}$. A hybrid probabilistic program (*h-program*) is a nh-program where all the rules are h-rules.*

The mapping τ in the above definition associates to each atomic hybrid basic formula A a disjunctive p-strategy that will be employed to combine the probability intervals obtained from different rules having A in their heads. A nh-program is ground if no variables appear in any of its rules. The following is a typical nh-program.

Example 1. Consider an insurance company which determines the premium categories, by calculating the risk factor according to a genetic test for cancer and the family history for this disease. Assume that customers who have a family history of the disease have a probability of developing cancer with at least 92%. The insurance company will assign high premiums to the customers who have family history of the disease and tested positive as long as their risk conditions are unchanged. Risk conditions can be changed by taking specific medications. This situation can be represented by the following nh-rules:

$risk(X) : [0.9, 1] \leftarrow (test(X) \wedge_{pc} history(X)) : [0.60, 0.75],$
$not\ changeRisk(X)[0.8, 1]$

$risk(X) : [0, 0.1] \leftarrow (test(X) \wedge_{pc} history(X)) : [0.60, 0.75],$
$changeRisk(X) : [0.8, 1]$

$changeRisk(X) : [0.9, 1] \leftarrow medicine(X, Med) : [0.65, 1]$

$highPremium(X) : [1, 1] \leftarrow risk(X) : [0.9, 1]$

$lowPremium(X) : [1, 1] \leftarrow risk(X) : [0, 0.1]$

$test(sam) : [0.92, 1] \leftarrow$

$history(sam) : [0.95, 1] \leftarrow$

$medicine(sam, medication) : [0.98, 1] \leftarrow$

and the mapping τ assigns *ncd* to $risk(sam)$ and an arbitrary disjunctive p-strategy [5,22] to the other hybrid basic formulae. The *ncd* denotes the disjunctive negative correlation p-strategy, which is defined as: $c_{ncd}([a_1, b_1], [a_2, b_2]) = [\min(1, a_1 + a_2), \min(1, b_1 + b_2)]$. The first nh-rule asserts that the risk factor

is at least 90% whenever the cancer genetic test for a customer is positive and that customer has a family history of cancer with probability between 60% and 75%, and it is not provable that his risk conditions have changed with probability at least 80%. Observe that *test* and *history* events are conjoined according to the *positive correlation* p-strategy (denoted by $\wedge_{pc}$) where $c_{pcc}([a_1, b_1], [a_2, b_2]) = [\min(a_1, a_2), \min(b_1, b_2)]$. The second rule says that the risk factor is at most 10% whenever the customer risk conditions are changed, even though the person tested positive and have a family history of the disease with probability between 60% and 75%. The third nh-rule describes the change of the risk conditions of a customer with probability at least 90% if a medication for the disease becomes available with probability at least 65%. The fourth and fifth nh-rules assert that definite high premium and low premium are considered whenever the probability of risk factors are at least 90% and at most 10% respectively. The last three nh-rules represent the facts available about a specific customer named sam.

2.3 Satisfaction and Models

In this subsection, we review the declarative semantics and the fixpoint semantics of h-programs [22] and we generalize the notions of interpretations, models, and satisfaction to deal with nh-programs. The notion of a probabilistic model (p-model) is based on *hybrid formula functions*.

Definition 3. *A hybrid formula function is a mapping* $h : bf_S(B_L) \rightarrow C[0,1]$ *that satisfies the following conditions:*

- *Commutativity:* $h(G_1 *_\rho G_2) = h(G_2 *_\rho G_1)$, $* \in \{\wedge, \vee\}$, $\rho \in S$
- *Composition:* $c_\rho(h(G_1), h(G_2)) \leq_t h(G_1 *_\rho G_2)$, $* \in \{\wedge, \vee\}$, $\rho \in S$
- *Decomposition. For any hybrid basic formula* F, $\rho \in S$, *and* $G \in bf_S(B_L)$: $md_\rho(h(F *_\rho G)) \leq_t h(F)$.

The notion of truth order can be extended to hybrid formula functions. Given hybrid formula functions h_1 and h_2, we say $(h_1 \leq_t h_2) \Leftrightarrow (\forall F \in bf_S(B_L) : h_1(F) \leq_t h_2(F))$. The set of all hybrid formula functions, HFF, and the truth order $\leq_t$ form a complete lattice. The meet $\otimes_t$ and the join $\oplus_t$ operations are defined respectively as: for all $F \in bf_S(B_L)$, $(h_1 \otimes_t h_2)(F) = h_1(F) \otimes_t h_2(F)$ and $(h_1 \oplus_t h_2)(F) = h_1(F) \oplus_t h_2(F)$.

Definition 4 (Probabilistic Interpretation). *A total (partial) probabilistic interpretation of a nh-program* P *is a total (partial) hybrid formula function.*

Before defining the notion of satisfaction for nh-programs, we introduce the following notations. Let h be a probabilistic interpretation, then $dom(h) \subseteq bf_S(B_L)$ denotes the domain of h ($dom(h) \subsetneqq bf_S(B_L)$ if h is a partial probabilistic interpretation). We use $negdom(h)$ to denote the set $\{F \,|\, F \in dom(h), h(F) = [0,0]\}$. We also define $posdom(h) = dom(h) \backslash negdom(h)$.

Definition 5 (Probabilistic Satisfaction). *Let* $P = \langle R, \tau \rangle$ *be a ground nh-program,* h *be a probabilistic interpretation, and* $r \equiv A : \mu \leftarrow F_1 : \mu_1, \ldots, F_n : \mu_n, not\ (G_1 : \beta_1), \ldots, not\ (G_m : \beta_m) \in R$. *Then*

- *h satisfies* $F_i : \mu_i$ *(denoted by* $h \models F_i : \mu_i$*) iff* $F_i \in dom(h)$ *and* $\mu_i \leq_t h(F_i)$.
- *h satisfies not* $(G_j : \beta_j)$ *(denoted by* $h \models not\ (G_j : \beta_j)$*) iff* $G_j \in dom(h)$ *and* $\beta_j \not\leq_t h(G_j)$.
- *h satisfies* $Body \equiv F_1 : \mu_1, \ldots, F_n : \mu_n, not\ (G_1 : \beta_1), \ldots, not\ (G_m : \beta_m)$ *(denoted by* $h \models Body$*) iff* $\forall (1 \leq i \leq n), h \models F_i : \mu_i$ *and* $\forall (1 \leq j \leq m), h \models not\ (G_j : \beta_j)$.
- *h satisfies* $A : \mu \leftarrow Body$ *iff* $h \models A : \mu$ *or h does not satisfy Body.*
- *h satisfies P iff h satisfies every nh-rule in R and for every atomic formula* $A \in dom(h)$, $c_{\tau(A)}\{\!\{\mu | A : \mu \leftarrow Body \in R \text{ and } h \models Body\}\!\} \leq_t h(A)$.

Definition 6 (Models). *Let P be a nh-program. A total probabilistic model of P (*p-model*) is a total probabilistic interpretation of P that satisfies P. A partial probabilistic model of P is a partial probabilistic interpretation of P that can be extended to a total probabilistic model of P.*

Proposition 1. *Let P be an* h-program. $h_P = \otimes_t\{h | h \text{ is a p-model of } P\}$ *is the least p-model of P.*

Associated with each h-program P, is an operator, T_P, called the *fixpoint operator*, which maps total probabilistic interpretations to total probabilistic interpretations.

Definition 7. *Let* $P = \langle R, \tau \rangle$ *be a ground h-program and h be a total probabilistic interpretation. The fixpoint operator* T_P *is a mapping* $T_P : HFF \rightarrow HFF$ *which is defined as follows:*

1. *if A is an atom,* $T_P(h)(A) = c_{\tau(A)}\ M_A$ *where* $M_A = \{\!\{\mu | A : \mu \leftarrow Body \in R$ *such that* $h \models Body\}\!\}$ *and* $M_A \neq \emptyset$ *. If* $M_A = \emptyset$*, then* $T_P(h)(A) = [0, 0]$
2. $T_P(h)(G_1 \wedge_\rho G_2) = c_\rho(T_P(h)(G_1), T_P(h)(G_2))$ *where* $(G_1 \wedge_\rho G_2) \in bf_S(B_L)$
3. $T_P(h)(G_1 \vee_\rho G_2) = c_\rho\ (T_P(h)(G_1), T_P(h)(G_2))$ *where* $(G_1 \vee_\rho G_2) \in bf_S(B_L)$.

Proposition 2. *Let P be an h-program. Then,* $h_P = lfp(T_P)$.

3 Probabilistic Well-Founded Semantics

In this section we define the probabilistic well-founded semantics for nh-programs. We start by defining the notion of probabilistic unfounded set and the immediate consequence operator of nh-programs with respect to a given probabilistic interpretation. Then the probabilistic well-founded semantics is defined inductively in terms of these two operators, which are natural extensions of their classical counterparts used in the well-founded semantics for normal logic programs [9].

Definition 8. *Let P be a nh-program,* H_P *be the set of all partial probabilistic interpretations of P, and* $h_1, h_2 \in H_P$*. We define the following partial order* $(\leq_w)$ *on* H_P*:* $h_1 \leq_w h_2$ *iff* $posdom(h_1) \subseteq posdom(h_2)$, $negdom(h_1) \subseteq negdom(h_2)$, *and* $\forall\ F \in dom(h_1), h_1(F) \leq_t h_2(F)$.

Definition 9. *Let h_1 and h_2 be two partial probabilistic interpretations. The meet $\otimes_w$ and join $\oplus_w$ operation corresponding to the partial order $\leq_w$ are defined respectively as:*

- $(h_1 \otimes_w h_2)(F) = h_1(F) \otimes_t h_2(F)$ *for all* $F \in ((posdom(h_1) \cap posdom(h_2)) \cup (negdom(h_1) \cap negdom(h_2)))$, *otherwise, undefined.*
- $(h_1 \oplus_w h_2)(F) = h_1(F) \oplus_t h_2(F)$ *for all* $F \in ((posdom(h_1) \cap posdom(h_2)) \cup (negdom(h_1) \cap negdom(h_2)))$, $(h_1 \oplus_w h_2)(F) = h_1(F)$ *for all* $F \in ((posdom(h_1) \setminus posdom(h_2)) \cup (negdom(h_1) \setminus negdom(h_2)))$, *and* $(h_1 \oplus_w h_2)(F) = h_2(F)$ *for all* $F \in ((posdom(h_2) \setminus posdom(h_1)) \cup (negdom(h_2) \setminus negdom(h_1)))$, *otherwise, undefined.*

Note that, the pair $\langle H_P, \leq_w \rangle$ *does not* form a lattice. In fact, if $h_1, h_2 \in H_P$ are probabilistic interpretations and $h_1 \not\leq_w h_2$, then $lub\{h_1, h_2\}$ may not exist. Consider $B_L = \{a, b, c, d\}$, $h_1(a) = h_1(b) = [0, 0], h_1(c) = h_1(d) = [1, 1]$, and $h_2(a) = [0, 0], h_2(b) = h_2(c) = h_2(d) = [1, 1]$. Then, according to the definition of $\leq_w$, $lub\{h_1, h_2\}$ must assign $[0, 0]$ to a, b and assign $[1, 1]$ to b, c and d which does not exist. However, $\langle H_P, \leq_w \rangle$ is *a complete partial order (cpo)*, i.e., a partial order in which the limit of each increasing chain exists. This is sufficient to allow the inductive construction of well-founded probabilistic models. The bottom element in this partial order is the partial probabilistic interpretation Φ whose domain is the empty set, and its top element is the total probabilistic interpretation which assigns $[1, 1]$ to each element in $bf_S(B_L)$. We say that a nh-program globally satisfies $F : \nu$ $(not\ (G : \beta))$ if the nh-program as a whole provides evidence for satisfying $F : \nu$ $(not\ (G : \beta))$.

Definition 10 (Global Satisfaction). *Let P be a nh-program and $F : \nu$ $(not(G : \beta))$ be a positive (negative) hybrid literal. We say that $F : \nu$ $(not(G : \beta))$ is* globally satisfied by *P if every minimal probabilistic interpretation that satisfies P also satisfies $F : \nu$ $(not\ (G : \beta))$.*

Definition 11 (Probabilistic Unfounded Sets). *Let $P = \langle R, \tau \rangle$ be a ground nh-program, $h \in H_P$, and $U \subseteq bf_S(B_L)$ such that for each non-atomic $F = A_1 \vee_\rho \ldots \vee_\rho A_n \in U$, all its constituent atoms belong to U and for each non-atomic $F = A_1 \wedge_\rho \ldots \wedge_\rho A_n \in U$ at least one $A_i \in U$ and the others are defined in h. U is called a* Probabilistic Unfounded Set *of P w.r.t. h if for each atomic $A \in U$ we have that for each nh-rule r in R whose head is $A : \mu$, at least one of the following conditions holds:*

- *there exist some positive hybrid literal $F : \nu$ in the body of r such that $F \in U$;*
- *h does not satisfy some hybrid literal $F : \nu$ or not $(G : \beta)$ in the body of r and P does not globally satisfy $F : \nu$.*

We consider h, in the above definition, to be what we already know about the intended probabilistic model of P. The idea is that the probabilistic unfounded

set corresponds to the set of negative conclusions of the nh-program P. Therefore, if a hybrid basic formula F is in a probabilistic unfounded set of P, then F should be assigned the probability interval $[0, 0]$ (representing absolute falsity) by the total or partial probabilistic model of P. The condition "P does not globally satisfies $F : \nu$" in the above definition is not required in the case of negative hybrid literals (*not* $(G : \beta)$). The reason is that if *not* $(G : \beta)$ is not satisfied by h, then *not* $(G : \beta)$ is not going to be satisfied by any $h \leq_w h'$. Instead, for positive literals $F : \nu$ we need to enforce the additional condition, used to guarantee that even by accumulating more knowledge, the probability interval assigned to F will not cover ν.

Definition 12 (Greatest Unfounded Set). *Let P be a ground nh-program and h be a partial probabilistic interpretation. The greatest probabilistic unfounded set $U_P(h)$ of P w.r.t. h is the union of all probabilistic unfounded sets of P w.r.t. h.*

Definition 13 (The Immediate Consequence Operator K_P). *Let $P = \langle R, \tau \rangle$ be a ground nh-program and $h \in H_P$. The immediate consequence operator K_P is the mapping $K_P : H_P \rightarrow H_P$ defined as follows:*

1. *For each atom A we have that $K_P(h)(A) = c_{\tau(A)}\ M_A$, where $M_A \neq \emptyset$ contains the probability intervals μ obtained from the nh-rules $A : \mu \leftarrow Body \in R$, such that h satisfies $Body$, and for each negative hybrid literal not $(G_j : \beta_j)$ in $Body$ we have that P globally satisfies not $(G_j : \beta_j)$.*
2. *$K_P(h)(G_1 \wedge_\rho G_2) = c_\rho(K_P(h)(G_1), K_P(h)(G_2))$ where $(G_1 \wedge_\rho G_2)$ contains only atoms from $dom(K_P(h)))$.*
3. *$K_P(h)(G_1 \vee_\rho G_2) = c_\rho\ (K_P(h)(G_1), K_P(h)(G_2))$ where $(G_1 \vee_\rho G_2)$ contains atoms from $(dom(h) \cup dom(K_P(h)))$ and at least one atom from $dom(K_P(h)))$.*

Intuitively, $K_P(h)$ corresponds to the set of positive conclusions of P with respect to the probabilistic interpretation h, where for each F defined in $K_P(h)$, $K_P(h)(F) \neq [0, 0]$. The condition "P globally satisfies *not* $(G_j : \beta_j)$" in (1) is not restrictive in the case of positive hybrid literals $F_i : \mu_i$. The reason is that if $F_i : \mu_i$ is satisfied by h, then $F_i : \mu_i$ is going to be satisfied by any $h \leq_w h'$. However, this is not the case with the negative hybrid literals *not* $(G_j : \beta_j)$. This is because if *not* $(G_j : \beta_j)$ in the body of a nh-rule is satisfied by h, then it might be not satisfied by some $h \leq_w h'$. Therefore, to guarantee that *not* $(G_j : \beta_j)$ is satisfied by h or by any $h \leq_w h'$, the condition in (1) is imposed. Since for any F defined in $K_P(h)$, $K_P(h)(F) \neq [0, 0]$, thanks to the properties of the conjunctive and disjunctive p-strategies (see [5,22] for more details), the condition "$(G_1 \wedge_\rho G_2)$ contains only atoms from $dom(K_P(h))$" in (2) and the condition "$(G_1 \vee_\rho G_2)$ contains atoms from $(dom(h) \cup dom(K_P(h)))$ and at least one atom from $dom(K_P(h))$" in (3) are imposed to determine $K_P(h)(G_1 \wedge_\rho G_2)$ and $K_P(h)(G_1 \vee_\rho G_2)$ respectively. This is because, for any $[a, b] \neq [0, 0]$ and any conjunctive p-strategy ρ, $c_\rho([a, b], [0, 0]) = [0, 0]$. Then it must be that $K_P(h)(G_1) \neq [0, 0]$ and $K_P(h)(G_2) \neq [0, 0]$ for $K_P(h)(G_1 \wedge_\rho G_2) \neq [0, 0]$.

However, for $K_P(h)(G_1 \vee_\rho G_2) \neq [0,0]$, it suffices that $K_P(h)(G_1) \neq [0,0]$ or $K_P(h)(G_2) \neq [0,0]$. Let us proceed with the definition of the probabilistic well-founded operator and the construction of the well-founded probabilistic models.

Definition 14. *Let P be a nh-program, h be a partial probabilistic interpretation, $K_P(h)$ be the immediate consequence operator, and $U_P(h)$ be the greatest probabilistic unfounded set of P w.r.t. h. Then, W_P is the mapping $W_P : H_P \rightarrow H_P$ such that*

- *$W_P(h)(F) = K_P(h)(F)$ for all $F \in dom(K_P(h))$, and*
- *$W_P(h)(F) = [0,0]$ for all $F \in U_P(h)$.*

Lemma 1. [1] *The operators W_P and K_P are monotonic w.r.t. $\leq_w$, and U_P is monotonic w.r.t. $\subseteq$.*

Definition 15. *The partial probabilistic interpretations h_α and h_∞ are defined recursively as follows:*

1. *$h_0 = \Phi$ where Φ is a partial probabilistic interpretation with an empty domain.*
2. *$h_\alpha = W_P(h_{\alpha-1})$ where α is the successor ordinal of $\alpha - 1$.*
3. *$h_\alpha = \oplus_w\{h_\beta \mid \beta < \alpha$ and α is a limit ordinal$\}$.*
4. *$h_\infty = \oplus_w\{h_\alpha \mid \alpha$ is an ordinal$\}$*

Definition 16. *Let P be a nh-program. $h_\infty = lfp(W_P)$ is the* well-founded (partial or total) probabilistic model *of P.*

Example 2. Let us consider the nh-program $P = \langle R, \tau\rangle$ from Example 1. It can be easily seen that P has a total well-founded probabilistic model h where

$h(risk(sam))$	$= [0, 0.1]$	$h(changeRisk(sam))$	$= [0.9, 1]$
$h(highPremium(sam))$	$= [0, 0]$	$h(lowPremium(sam))$	$= [1, 1]$
$h(test(sam))$	$= [0.92, 1]$	$h(history(sam))$	$= [0.95, 1]$
$h(medicine(sam, medication))$	$= [0.98, 1]$	$h(test(sam) \wedge_{pc} history(sam))$	$= [0.92, 1]$

.

Example 3. Consider the following nh-program $P = \langle R, \tau\rangle$ where R is

$$\begin{array}{ll} a : [0.89, 0.91] & \leftarrow not\ (b : [0.3, 0.4]) \\ b : [0.55, 0.60] & \leftarrow not\ (a : [0.7, 0.75]) \\ c : [0.2, 0.3] & \leftarrow d : [0.1, 0.15] \\ d : [0.1, 0.2] & \leftarrow not\ (e : [0.1, 0.3]) \end{array}$$

and $\tau(a) = \tau(b) = \tau(c) = \tau(d) = \pi$ where π is any arbitrary disjunctive p-strategy. This nh-program has a well-founded partial probabilistic model that assigns $[0.2, 0.3]$ to c, $[0.1, 0.2]$ to d, and $[0, 0]$ to e. This is because $h_1 = W_P(\Phi)$ assigns $[0, 0]$ to e since $K_P(\Phi) = \Phi$ and $U_P(\Phi) = \{e\}$. $h_2 = W_P(h_1)$ assigns $[0.1, 0.2]$ to d and $[0, 0]$ to e since $K_P(h_1)$ assigns $[0.1, 0.2]$ to d and $U_P(h_1) = \{e\}$.

[1] The proofs are omitted due to lack of space. All proofs can be found at http://www.cs.nmsu.edu/TechReports/2005/006.pdf

$h_3 = W_P(h_2)$ assigns $[0.1, 0.2]$ to d, $[0.2, 0.3]$ to c, and $[0, 0]$ to e since $K_P(h_2)$ assigns $[0.1, 0.2]$ to d and $[0.2, 0.3]$ to c, and $U_P(h_2) = \{e\}$. $h_3 = W_P(h_2)$ is the least fixpoint since $h_3 = W_P(h_2) = h_4 = W_P(h_3)$.

Example 4. Consider the following nh-program $P = \langle R, \tau \rangle$ where R is

$$\begin{array}{ll} a : [0.4, 0.7] & \leftarrow not\ (b : [0.5, 0.75]) \\ b : [0.5, 0.9] & \leftarrow not\ (a : [0.35, 0.6]) \\ r : [0.25, 60] & \leftarrow a : [0.4, 0.65] \\ r : [0.3, 0.65] & \leftarrow b : [0.5, 0.8] \end{array}$$

and $\tau(a) = \tau(b) = \tau(r) = \pi$. The well-founded p-model of P is Φ. This is because $W_P(\Phi) = \Phi$ since $K_P(\Phi) = \Phi$ and $U_P(\Phi) = \emptyset$.

Theorem 1. *Every h-program P has a well-founded total probabilistic model h iff h is the least p-model of P.*

Let us show that the probabilistic well-founded semantics generalizes the well-founded semantics of normal logic programs. A normal logic program P can be represented as a nh-program $P' = \langle R, \tau \rangle$ where each normal rule

$$a \leftarrow b_1, \ldots, b_n, not\ c_1, \ldots, not\ c_m \in P$$

can be encoded, in R, as a nh-rule of the form

$$a : [1, 1] \leftarrow b_1 : [1, 1], \ldots, b_n : [1, 1], not\ (c_1 : [1, 1]), \ldots, not\ (c_m : [1, 1])$$

where $a, b_1, \ldots, b_n, c_1, \ldots, c_m$ are atomic hybrid basic formulae and $[1, 1]$ represents the truth value *true*. τ is any arbitrary assignment of disjunctive p-strategies. We call the class of nh-programs that consists only of nh-rules of the above form as $NHPP_1$.

Proposition 3. *Let P be a normal logic program. Then, I is a well-founded partial or total model for P iff h is a well-founded partial or total probabilistic model for P' where $h(a) = [1, 1]$ iff $a \in I$ and $h(b) = [0, 0]$ iff not $b \in I$.*

4 Stable Probabilistic Model Semantics

In this section we introduce the notion of *stable probabilistic models (sp-models)*, which extends the notion of stable models for classical logic programming [7]. The semantics is defined in two steps. First, we guess a p-model h for a certain nh-program P, then we define the notion of the probabilistic reduct of P with respect to h—which is an h-program. Second, we determine whether h is a stable p-model for P or not by employing the fixpoint operator of the probabilistic reduct to verify whether h is its least p-model. All probabilistic interpretations and models that we consider in this section are total. It must be noted that every h-program has a unique least (total) p-model [22].

Definition 17 (Probabilistic Reduct). *Let $P = \langle R, \tau \rangle$ be a ground nh-program and h be a probabilistic interpretation. The probabilistic reduct P^h of P w.r.t. h is $P^h = \langle R^h, \tau \rangle$ where:*

$$R^h = \left\{ A : \mu \leftarrow F_1 : \mu_1, \ldots, F_n : \mu_n \;\middle|\; \begin{array}{l} A : \mu \leftarrow F_1 : \mu_1, \ldots, F_n : \mu_n, \\ \quad not\ (G_1 : \beta_1), \ldots, not\ (G_m : \beta_m) \in R \text{ and} \\ \forall (1 \leq j \leq m),\ \beta_j \not\leq_t h(G_j) \end{array} \right\}$$

The probabilistic reduct P^h is an h-program. For any *not* $(G_j : \beta_j)$ in the body of $r \in R$ with $\beta_j \not\leq_t h(G_j)$ is simply satisfied by h, and *not* $(G_j : \beta_j)$ is removed from the body of r. If $\beta_j \leq_t h(G_j)$ then the body of r is not satisfied and r is trivially ignored.

Definition 18 (Stable Probabilistic Model). *A probabilistic interpretation h is a stable p-model of a nh-program P if h is the least p-model of P^h.*

Example 5. It is easy to verify that the only stable p-model of the program in Example 1 is given by:

$h(risk(sam))$	$= [0, 0.1]$	$h(changeRisk(sam))$	$= [0.9, 1]$
$h(highPremium(sam))$	$= [0, 0]$	$h(lowPremium(sam))$	$= [1, 1]$
$h(test(sam))$	$= [0.92, 1]$	$h(history(sam))$	$= [0.95, 1]$
$h(medicine(sam, medication))$	$= [0.98, 1]$	$h(test(sam) \wedge_{pc} history(sam))$	$= [0.92, 1]$

Example 6. The nh-program in Example 3 has two stable p-models h_1 and h_2 where $h_1(a) = [0.89, 0.91], h_1(b) = [0, 0], h_1(c) = [0.2, 0.3], h_1(d) = [0.1, 0.2]$, $h_1(e) = [0, 0]$ and $h_2(a) = [0, 0], h_2(b) = [0.55, 0.60], h_2(c) = [0.2, 0.3], h_2(d) = [0.1, 0.2], h_2(e) = [0, 0]$. Since, for example, h_1 can be verified as a stable p-model because the probabilistic reduct of P w.r.t. h_1 contains the h-rules:

$$\begin{array}{lll} a : [0.89, 0.91] & \leftarrow & \\ c : [0.2, 0.3] & \leftarrow & d : [0.1, 0.15] \\ d : [0.1, 0.2] & \leftarrow & \end{array}$$

and $lfp(T_{P^{h_1}}) = h_1$.

Example 7. The nh-program in Example 4 has two stable p-models h_1 and h_2 where $h_1(a) = [0.4, 0.7], h_1(b) = [0, 0], h_1(r) = [0.25, 0.60]$ and $h_2(a) = [0, 0], h_2(b) = [0.5, 0.9]$,
$h_2(r) = [0.3, 0.65]$. Since, for example, h_2 can be verified as a stable p-model because the probabilistic reduct of P w.r.t. h_2 contains the h-rules:

$$\begin{array}{lll} b : [0.5, 0.9] & \leftarrow & \\ r : [0.25, 0.60] & \leftarrow & a : [0.4, 0.65] \\ r : [0.3, 0.65] & \leftarrow & b : [0.5, 0.8] \end{array}$$

and $lfp(T_{P^{h_2}}) = h_2$.

Theorem 2. *Every h-program P has a unique stable p-model h iff h is the least p-model of P.*

The following result shows that the stable p-model semantics generalizes the stable model semantics for classical logic programming [7].

Proposition 4. *Let P be a normal logic program. Then S' is a stable model of P iff h is a stable p-model of $P' \in NHPP_1$ that corresponds to P where $h(a) = [1,1]$ iff $a \in S'$ and $h(b) = [0,0]$ iff $b \in B_L \setminus S'$.*

In the rest of this section we define the *immediate consequence operator* of nh-programs and study its relationship to the stable p-model semantics.

Definition 19. *Let $P = \langle R, \tau \rangle$ be a ground nh-program and $h \in HFF$. The* immediate consequence operator T'_P *is a mapping $T'_P : HFF \to HFF$ defined as follows:*

1. $T'_P(h)(A) = c_{\tau(A)}\ M'_A$ *where*

$$M'_A = \left\{\!\!\left\{ \mu \;\middle|\; \begin{array}{c} A : \mu \leftarrow F_1 : \mu_1, \ldots, F_n : \mu_n, not\ (G_1 : \beta_1), \ldots, not\ (G_m : \beta_m) \in R \text{ and} \\ \forall (1 \le i \le n), \mu_i \le_t h(F_i) \text{ and } \forall (1 \le j \le m), \beta_j \not\le_t h(G_j) \end{array} \right\}\!\!\right\}$$

   and $M'_A \neq \emptyset$. $T'_P(h)(A) = [0,0]$ if $M'_A = \emptyset$.
2. $T'_P(h)(G_1 \wedge_\rho G_2) = c_\rho(T'_P(h)(G_1), T'_P(h)(G_2))$ *where* $(G_1 \wedge_\rho G_2) \in bf_S(B_L)$.
3. $T'_P(h)(G_1 \vee_\rho G_2) = c_\rho\ (T'_P(h)(G_1), T'_P(h)(G_2))$ *where* $(G_1 \vee_\rho G_2) \in bf_S(B_L)$.

It is easy to see that T'_P extends T_P to handle h-rules with negative hybrid literals and, hence, $T'_P = T_P$ for any h-program P. The operator T'_P is not monotonic w.r.t. $\le_t$. This can be seen in the following example.

Example 8. Consider the nh-program: $a : [0.2, 0.3] \leftarrow not\ (b : [0.6, 0.8])$. Let h_1 be a probabilistic interpretation that assigns $[0,0]$ to b and $[0,0]$ to a. In addition, let h_2 be a probabilistic interpretation that assigns $[0.65, 0.9]$ to b and $[0,0]$ to a. Hence, $h_1 \le_t h_2$. But $T'_P(h_1)(a) = [0.2, 0.3]$ and $T'_P(h_2)(a) = [0,0]$. Thus, $T'_P(h_1) \not\le_t T'_P(h_2)$

Theorem 3. *Let P be a nh-program and h be a stable p-model of P. Then h is a minimal fixpoint of T'_P.*

It is worth noting that not every minimal fixpoint of T'_P is a stable p-model of P. Consider the following nh-program P.

Example 9. Let $P = \langle R, \tau \rangle$ where τ is arbitrary and R

$$\begin{array}{ll} a : [0.7, 0.8] & \leftarrow not\ (a : [0.1, 0.17]) \\ a : [0.1, 0.33] & \leftarrow b : [0.6, 0.8] \\ b : [1,1] & \leftarrow a : [0.1, 0.24] \end{array}$$

It is easy to verify that the probabilistic interpretation $h(a) = [0.1, 0.33]$ and $h(b) = [1,1]$ is a minimal fixpoint of T'_P. However, P^h contains $a : [0.1, 0.33] \leftarrow b : [0.6, 0.8]$ and $b : [1,1] \leftarrow a : [0.1, 0.24]$ where $lfp(T_{P^h})(a) = lfp(T_{P^h})(b) = [0,0]$. Hence, h is not a stable p-model for P.

5 Stable P-Model Semantics and Probabilistic Well-Founded Semantics Relationships

There is a close relationship between the well-founded probabilistic models and the stable probabilistic models. In this section we study this relationship. For a given nh-program P, the following results show that for every total p-model h of P, h is a stable p-model of P if and only if it is a fixpoint of the probabilistic well-founded operator W_P. However, well-founded total probabilistic models are unique stable probabilistic models.

Theorem 4. *Let P be a nh-program and h be a total p-model of P. Then h is stable p-model of P iff h is a fixpoint of W_P.*

Corollary 1. *Let P be a nh-program and h be a well-founded total p-model of P. Then h is the unique stable p-model of P.*

The following result shows that the well-founded probabilistic model approximates the stable p-models of a nh-program, since the well-founded partial p-model of a nh-program P is contained (with respect to the partial order $\leq_w$) in every stable p-model of P.

Corollary 2. *Let P be a nh-program and h be a well-founded partial p-model of P. Then for every stable p-model g of P, $h \leq_w g$.*

6 Related Work

A stable model semantics extension to the probabilistic logic programming in [18,19] was presented in [20]. The notion of non-monotonic negation presented in [20] is closer to our definition of non-monotonic negation. The main difference with respect to [20] is that we employ the truth order instead of the knowledge order as well as our framework allows reasoning with different modes of probabilistic combinations. However, [20] is limited to a single mode of probabilistic combination. Moreover, the stable model semantics in [20] is computationally expensive, due to annotated conjunctions or disjunctions are allowed as heads of rules. On the other hand, we allow only annotated atoms as heads of rules, rather than annotated conjunctions or disjunctions as in [20], to avoid the high computational complexity of the semantics [15]. Another important difference between our framework and [20] is that we do not allow hybrid basic formulae with annotation $[0, 0]$ to appear neither in the heads nor in the bodies of the rules (this is an extension to our framework that we will consider in the future according to the open world assumption), however, [18,19,20,5] does, although these semantics as well as ours are based on the *closed world assumption*. The reason is that having a hybrid basic formula, A, in a nh-program with the annotation $[0, 0]$, i.e., $A : [0, 0]$, means that A is absolutely false. This $A : [0, 0]$ corresponds to classical negation $\neg A$, which in turn requires a different treatment when defining the semantics of the programs.

A probabilistic semantics, based on the possible world semantics, for disjunctive logic programs with non-monotonic negation has been presented in [16]. The semantics of [16] is based on multi-valued logic and a stable model semantics has been described. In addition to programs in [16] are disjunctive logic programs, probabilities are treated as a lattice of truth values, where the probability of the conjunction $Prob(A \wedge B) = min(Prob(A), Prob(B))$ and the probability of the disjunction $Prob(A \vee B) = max(Prob(A), Prob(B))$. This is considered a fixed mode of combination. Whereas, in our framework conjunctions and disjunctions are treated differently according to the type of dependency between events. In [2], a new methodology to probabilistic reasoning was presented under the possible world semantics by employing *answer set programming* for classical logic programming. Answer set programming in [2] is exploited to emulate the possible world semantics. However, [2] assumes independence of probabilities which is a fixed mode of probabilistic combination.

Our probabilistic well-founded semantics introduced in this paper differs from the well-founded semantics presented in [13,14] in various ways. The notion of non-monotonic negation in [13,14] is closer to the classical negation. In addition, our probabilistic well-founded semantics is based on the declarative well-founded semantics for normal logic programs [9], however, the well-founded semantics in [13,14] is based on the alternating fixpoint semantics [8]. A stable model semantics and well-founded semantics (based on alternating fixpoint semantics) have also been presented in [23]. However, the certainty values that are reasoned about are non-probabilistic values. In addition, no annotated conjunctions or disjunctions are allowed in the body of rules [23]. Furthermore, in [4], the semantics of [23] has been extended to allow classical negation as well as non-monotonic negation by proposing alternating fixpoint like semantics. A generalization of HPP of [5] was proposed in [3] by providing a more general semantical characterization in which HPP fits. However, [3] does not allow non-monotonic negation in defining its semantics. In addition, it relies on a complex translation process which is exponential in the size of HPP.

7 Conclusions and Future Work

We presented an extension of the language of hybrid probabilistic programs framework [22], called normal hybrid probabilistic programs, to cope with non-monotonic negation. The extension is a necessary requirement in many real-world applications (e.g., planning with incomplete and uncertain knowledge). We developed a semantical characterization of the extended framework, which relies on a probabilistic generalization of the well-founded semantics and stable model semantics, originally developed for normal logic programs. We showed that the probabilistic well-founded semantics and the stable probabilistic model semantics naturally generalize the well-founded semantics and the stable model semantics for classical logic programming. Furthermore, we showed that they naturally extend the semantics for HPP (without negation) proposed in [22]. Moreover, we showed that the relationship between the probabilistic well-founded semantics

and the stable probabilistic model semantics preserves the relationship between the well-founded semantics and the stable model semantics for normal logic programs.

A topic of future research is to extend the language of normal hybrid probabilistic programs to allow classical negation and disjunctions of annotated atomic formulae in the heads of nh-rules. We plan to develop an alternating fixpoint semantics for the language of nh-programs and analytically study its relationship to the probabilistic well-founded semantics proposed in this paper. In addition, we intend to investigate the computational aspects of the stable probabilistic model semantics and the probabilistic well-founded semantics—by developing algorithms and implementations for computing these semantics. The algorithms and implementations we will develop will be based on appropriate extensions of the existing techniques for computing the stable model semantics and the well-founded semantics for normal logic programs, e.g., SMODELS [17].

References

1. K.R. Apt and R.N. Bol. Logic programming and negation:a survey. *Journal of logic programming*, 19/20:9-71, 1994.
2. C. Baral et al. Probabilistic reasoning with answer sets. *In 7th International Conference on Logic Programming and Nonmonotonic Reasoning*, Springer Verlag, 2004.
3. C.V. Damasio and L. Moniz Pereira. Hybrid probabilistic logic programs as residuated logic programs. *JELIA*, 2000.
4. C.V. Damasio et al. Coherent well-founded annotated logic programs. *LPNMR*, Springer, 1999.
5. A. Dekhtyar and V.S. Subrahmanian. Hybrid probabilistic program. *Journal of Logic Programming*, 43(3): 187-250, 2000.
6. W.F. Dowling and J.H. Gallier. Linear-time algorithms for testing the satisfiability of propositional Horn formulae. *Journal of Logic Programming*, 1(3): 267-284, 1984.
7. M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. *ICSLP*, 1988, MIT Press.
8. A. Van Gelder. The alternating fixpoint of logic programs with negation. *Journal of Computer and System Sciences*, 47(1):185-221, 1993.
9. A. Van Gelder, K.A. Ross, and J.S. Schlipf. The well-founded semantics for general logic programs. *Journal of ACM*, 38(3):620-650, 1991.
10. M. Kifer and V.S. Subrahmanian. Theory of generalized annotated logic programming and its applications. *Journal of Logic Programming*, 12:335–367, 1992.
11. L.V.S. Lakshmanan and F. Sadri. On a theory of probabilistic deductive databases. *Journal of Theory and Practice of Logic Programming*, 1(1):5-42, January 2001.
12. L.V.S. Lakshmanan and N. Shiri. A parametric approach to deductive databases with uncertainty. *IEEE TKDE*, 13(4):554–570, 2001.
13. Y. Loyer and U. Straccia. The well-founded semantics in normal logic programs with uncertainty. *FLOPS*, 2002, Springer Verlag.
14. Y. Loyer and U. Straccia. The approximate well-founded semantics for logic programs with uncertainty. *In 28th International Symposium on Mathematical Foundations of Computer Science*, 2003.

15. T. Lukasiewicz. Probabilistic logic programming. *In 13th European Conference on Artificial Intelligence*, 388–392, 1998.
16. T. Lukasiewicz. Many-valued disjunctive logic programs with probabilistic semantics. *LPNMR*, 1999.
17. I. Niemela and P. Simons. Efficient implementation of the well-founded and stable model semantics. *In Joint International Conference and Symposium on Logic Programming*, 289-303, 1996.
18. R.T. Ng and V.S. Subrahmanian. Probabilistic logic programming. *Information & Computation*, 101(2), 1992.
19. R.T. Ng and V.S. Subrahmanian. A semantical framework for supporting subjective and conditional probabilities in deductive databases. *ARJ*, 10(2), 1993.
20. R.T. Ng and V.S. Subrahmanian. Stable semantics for probabilistic deductive databases. *Information & Computation*, 110(1), 1994.
21. E. Saad. *Hybrid probabilistic programs with non-monotonic negation: semantics and algorithms*. Ph.D. thesis, New Mexico State University, May 2005.
22. E. Saad and E. Pontelli. Towards a more practical hybrid probabilistic logic programming framework.In *Practical Aspects of Declarative Languages*. Springer Verlag, 2005.
23. V.S. Subrahmanian. Amalgamating knowledge bases. *ACM TDS*, 19(2):291–331, 1994.

Reducing Inductive Definitions to Propositional Satisfiability

Nikolay Pelov and Eugenia Ternovska

School of Computing Science, Simon Fraser University,
Vancouver, Canada

Abstract. The FO(ID) logic is an extension of classical first-order logic with a uniform representation of various forms of inductive definitions. The definitions are represented as sets of rules and they are interpreted by two-valued well-founded models. For a large class of combinatorial and search problems, knowledge representation in FO(ID) offers a viable alternative to the paradigm of Answer Set Programming. The main reasons are that (i) the logic is an extension of classical logic and (ii) the semantics of the language is based on well-understood principles of mathematical induction.

In this paper, we define a reduction from the propositional fragment of FO(ID) to SAT. The reduction is based on a novel characterization of two-valued well-founded models using a set of inequality constraints on level mappings associated with the atoms. We also show how the reduction to SAT can be adapted for logic programs under the stable model semantics. Our experiments show that when using a state of the art SAT solver both reductions are competitive with other answer set programming systems — both direct implementations and SAT based.

1 Introduction

Definitions, and inductive definitions in particular, are common in human reasoning [3]. Examples of inductive definitions include the definitions of the set of well-formed formulas and the satisfaction relation $\models$ in logic. An example of an inductive definition from common-sense reasoning concerns reasoning about actions. There, a description of the initial situation represents the base case, and causal laws specifying direct and indirect effects of action, represent inductive cases [6, 7, 21, 22]. Inductive definitions can be monotone (e.g. the definition of a well-formed formula) or non-monotone (e.g. the definition of $\models$ and many definitions in common-sense reasoning). Both monotone and non-monotone induction are formalized in a natural way in the logic FO(ID) which is an extension of first-order logic (FO) with (non-monotone) inductive definitions (ID) [4, 6]. The semantics of FO(ID) is a combination of the semantics of first-order logic and the well-founded semantics of logic programming. The usefulness of FO(ID) in knowledge representation has been demonstrated in several applications. The authors of [5] show that the situation calculus can be formalized as a (non-monotone) iterated inductive definition in the well-ordered set of situations. The resulting formalism provides a very general solution to the ramification problem. Another natural application of FO(ID) is to data integration in database theory [24].

M. Gabbrielli and G. Gupta (Eds.): ICLP 2005, LNCS 3668, pp. 221–234, 2005.

Recently, the FO(ID) logic has been used as the underlying logic of a declarative constraint programming framework [17]. The approach is formalized as a model expansion problem, which is based on the classical notion of expansion of a structure by new relations. A parameterized version of this problem captures precisely NP search problems.

The FO(ID) logic is similar [16] to the framework of Answer Set Programming (ASP) [1]. It can be argued that the representation of many problems in FO(ID) is more intuitive than the representation in logic programming under the answer set semantics. To a large extent this is due to the fact that FO(ID) is an extension of classical logic which is well-understood and widely used. Another reason is that the use of recursion is limited only to concepts where it is really needed and predicates which are defined recursively are interpreted as inductive definitions. On the other hand, recursion (very often over negation) is used much more frequently in ASP, for example for defining the set of possible solutions. So, for someone who is not familiar with the knowledge representation methodology of ASP, this is often confusing. Although recent extensions of the language, for example with choice rules [20], alleviate the problem to some extent, the basic critique remains.

Despite its attractive properties, no full-featured implementation of FO(ID) exists. Mariën, Gilis and Denecker [16] define a reduction from FO(ID) to ASP, however only total definitions are supported[1]. In this paper, we discuss an implementation of the propositional fragment of FO(ID) called PC(ID), which stands for *propositional calculus with inductive definitions*. Our approach is to define a reduction from the satisfiability problem of PC(ID) to propositional satisfiability (SAT). Since part of a PC(ID) theory is already a set of propositional formulas, the main challenge is the translation of inductive definitions.

The use of SAT solvers has already been shown to be successful for implementing ASP systems [12, 13, 14]. The approach which is typically used is to compute the Clark's completion of the program and call a SAT solver to return a model. Then, the model is verified if it is stable and, if it is not, a special type of formulas, called loop formulas, are added to the theory to eliminate the model. This approach works quite well in practice, although for some problems there may be an exponential number of loop formulas. There are also several "off-line" reductions to SAT [2, 11, 13, 15] but their performance is not yet well studied.

An important property of the reductions to SAT, called *faithfulness*, refers to the case where there is a one-to-one correspondence between the models of the original theory and its reduction. The reductions of Lin and Zhao [13] and Janhunen [11] are faithful while those of Ben-Eliyahu and Dechter [2] and Linke, Tompits and Woltran [15] are not. Faithful reductions are useful when one is interested to compute or to count all solutions of a problem. However, faithful reductions are normally larger than non-faithful ones. What influence this difference has on the speed of SAT solvers is still unknown. We are unaware of any comparison on the performance of faithful and non-faithful reductions

[1] A definition D is total if it has a two-valued well-founded model for every interpretation of the open predicates of D.

and the experiments in this paper are a first step in this direction. The reduction of PC(ID) to SAT which we define is non-faithful. The main reason is that the difference between the size of a faithful and non-faithful reduction for the well-founded semantics is much greater than the difference between faithful and non-faithful reductions for the stable semantics. The main difficulty comes from the computation of greatest unfounded sets.

The reduction from PC(ID) to SAT is based on a novel characterization of two-valued well-founded models by a set of inequality constraints on level mappings associated with the atoms in the theory. The level mapping of an atom a is related to the step of the well-founded operator W_P at which the truth value of a is derived. The reduction is very similar to the one by Ben-Eliyahu and Dechter [2] for head-cycle free disjunctive logic programs under the stable model semantics. Besides the difference in the language and semantics, the other main difference between the two reductions is that we use a binary encoding of level mappings as in [11] while [2] uses a unary encoding.

The paper is organized as follows. We start by recalling in Section 2 some preliminaries from logic programming. In Section 3 we recall the syntax and semantics of the logics FO(ID) and PC(ID). In Section 4 we develop the theoretical foundation of the reduction by characterizing well-founded models by certain types of level mappings. The reduction itself is defined in Section 5 and in Section 6 we evaluate its performance and compare it with other ASP systems.

2 Preliminaries

Let At be a set of atoms. A *literal* is an atom $a \in At$ or its negation $\neg a$. A *rule* is an expression of the form

$$a \leftarrow l_1 \wedge \ldots \wedge l_n \tag{1}$$

where $a \in At$ and l_i are literals over At. For a rule r of the form (1) we denote $hd(r) = a$ and $body(r) = \{l_1, \ldots, l_n\}$. The set of positive literals in the body of r is denoted with $pos(r)$ and the set of negative literals with $neg(r)$. A *logic program* is a finite set of rules. A *definite rule* is a rule without negative literals in the body and a *definite program* is a program consisting only of definite rules.

With $th(P)$ we denote the set of formulas obtained by replacing "$\leftarrow$" with "$\subset$" in every rule in P. The *only-if* part for a logic program P, denoted with $only\_if(P)$, is defined as the set of formulas $a \supset B_1 \vee \ldots \vee B_n$ for every atom $a \in At$ where $a \leftarrow B_1, \ldots, a \leftarrow B_n$ are all the rules with a in the head. If the body B_i of some rule for a is empty then B_i is understood as the constant $\mathbf{t}$. Finally, the *completion* of a program P is defined as $comp(P) = th(P) \cup only\_if(P)$.

The *dependency graph* G_P of a logic program P is a signed directed graph defined as follows. The atoms of At form the vertices of G_P. For every rule $r \in P$ there is an edge from $hd(r)$ to b for every positive literal $b \in pos(r)$ and there is an edge from $hd(r)$ to b labeled with "-" for every negative literal $\neg b \in neg(r)$. We say that an atom a *depends negatively on itself* if G_P contains a (directed) cycle containing a and an edge labeled with "-". A *strongly connected component*

(SCC) is a maximal set of atoms A such that there is a directed path in G_P for every pair of atoms in A. The set of all strongly connected components of G_P is denoted with $SCC(P)$ and the SCC to which an atom a belongs with $SCC(a)$. For a strongly connected component $S \in SCC(P)$ we denote with P_S the subprogram of P which is restricted only to rules with atoms in S in the head.

An *interpretation* I is a function $I\colon At \to \{\mathbf{f}, \mathbf{t}\}$. Frequently, we identify an interpretation with the set of atoms which are assigned the value $\mathbf{t}$. The *complement* $\overline{I}$ of an interpretation I is taken with respect to At, i.e., $\overline{I} = At \setminus I$. A *three-valued interpretation* is a consistent set of literals $\tilde{I}$. We denote the subset of positive literals of $\tilde{I}$ with $\tilde{I}^+$ and the subset of negative literals with $\tilde{I}^-$. For a set of literals L, we denote with $\neg L$ the set which consists of all literals from L with reversed polarity.

The *program reduct* P^I [10] of a program P with respect to an interpretation I is a program obtained from P by:

- deleting all rules which have a negative literal not satisfied by I;
- deleting all negative literals from the remaining rules.

The program reduct P^I is a definite logic program and it has a unique least model, denoted with $lm(P^I)$. An interpretation I is a *stable model* of P if $I = lm(P^I)$.

Next, we recall the definition of the well-founded semantics [23].

Definition 1 (Unfounded Set). *Let P be a logic program and let $\tilde{I}$ be a three-valued interpretation. We say that a set of atoms $A \subseteq At$ is an* unfounded set *(of P)* with respect to $\tilde{I}$ *if each atom $a \in A$ satisfies the following condition. For each rule $r \in P$ such that $hd(r) = a$ one of the following holds:*

1. *$\neg l_i \in \tilde{I}$ for some literal $l_i \in body(r)$;*
2. *some positive literal in $body(r)$ occurs in A, i.e., $pos(r) \cap A \neq \emptyset$.*

Taking the union of any collection of unfounded sets is also an unfounded set. So, for any given three-valued interpretation $\tilde{I}$, a program P has a *greatest unfounded set* with respect to $\tilde{I}$, denoted with $U_P(\tilde{I})$. Next we define the following three-valued operators:

$$\mathcal{T}_P(\tilde{I}) = \{hd(r)\colon r \in P \text{ and } body(r) \subseteq \tilde{I}\}$$
$$W_P(\tilde{I}) = \mathcal{T}_P(\tilde{I}) \cup \neg U_P(\tilde{I})$$

The W_P operator is monotone in the sense that if $\tilde{I}_1 \subseteq \tilde{I}_2$ then $W_P(\tilde{I}_1) \subseteq W_P(\tilde{I}_2)$. By a well-known result of Tarski, follows that W_P has a least fixpoint denoted with $\mathrm{lfp}(W_P)$. The *well-founded model* of a logic program P is defined as $\mathrm{lfp}(W_P)$ and denoted with $WF(P)$. The least fixpoint of the W_P operator can also be computed constructively by the following iteration of the W_P operator:

$$W_P^0 = \emptyset$$
$$W_P^{i+1} = W_P(W_P^i) \qquad \text{for } i \in \mathbb{N}$$

Proposition 1. *There exists a natural number n that $W_P^n = W_P^{n+1} = \mathrm{lfp}(W_P)$.*

The least natural number n which satisfies the conditions of the above proposition, i.e., $W_P^n = \mathrm{lfp}(W_P)$, is called the *closure number* of W_P. The *stage* of a literal $l \in WF(P)$, denoted with $|l|_{W_P}$, is defined as the least number $i \in \mathbb{N}$ such that $l \in W_P^i$.

3 Propositional Calculus with Inductive Definitions

An FO(ID) theory is a pair $\langle \mathcal{D}, \mathcal{A} \rangle$ where $\mathcal{D}$ is a set of definitions and $\mathcal{A}$ is a set of FO sentences[2]. A definition $D \in \mathcal{D}$ is a set of rules of the form $\forall \overline{x}(p(\overline{t}) \leftarrow \varphi)$ where $p(\overline{t})$ is an atom and φ is a FO formula. The set of predicates which appear in the heads of the rules in a definition D are *defined* by D and denoted with $def(D)$. All other predicate symbols in D are called *open* and their set is denoted with $open(D)$.

The propositional fragment of FO(ID) is denoted with PC(ID) and stands for *propositional calculus with inductive definitions*. For simplicity, we assume that a definition is a propositional logic program, i.e., the body of every rule is a conjunction of literals and every sentence in $\mathcal{A}$ is a propositional clause.

The semantics of FO(ID) and PC(ID) is defined as a combination of classical first-order semantics for the set of FO sentences and two-valued well-founded semantics for the definitions.

Definition 2. *An interpretation I is a* model *of a PC(ID) theory $\langle \mathcal{D}, \mathcal{A} \rangle$ if:*

1. *for every definition $D \in \mathcal{D}$, I is the (two-valued) well-founded model of $D \cup (I \cap open(D))$ where $I \cap open(D)$ is the set of open atoms in D which are true in I;*
2. *$I \models \mathcal{A}$.*

We illustrate the syntax of the logic with a formulation of the Hamiltonian Cycle problem [16].

Example 1 (Hamiltonian Cycle). The problem of finding a Hamiltonian cycle in a directed graph is encoded by the following FO(ID) theory $\langle \{D_1, D_2\}, \mathcal{A} \rangle$. The first definition D_1 encodes the input graph and a designated initial node as a set of facts:

$$D_1 = \left\{ \begin{array}{l} vertex(v). \\ \ldots \\ arc(u, v). \\ \ldots \\ initialnode(v). \end{array} \right\}.$$

The Hamiltonian cycle is described by a binary predicate $hc(x, y)$. The second definition defines, by a positive induction, the the set of nodes reachable through the $hc(x, y)$ relation starting from a designated initial node:

[2] In this paper we use a restricted syntax of FO(ID) and its propositional fragment PC(ID) which is more suitable for programming and implementation.

$$D_2 = \left\{ \begin{array}{l} \forall xy(reached(y) \leftarrow initialnode(x) \wedge arc(x,y) \wedge hc(x,y)) \\ \forall xy(reached(y) \leftarrow reached(x) \wedge arc(x,y) \wedge hc(x,y) \wedge \\ \qquad\qquad \neg\, initialnode(x)) \end{array} \right\}.$$

Finally, the formulas $\mathcal{A}$ are the following axioms:

$$\begin{array}{l} \forall x(vertex(x) \supset reached(x)), \\ \forall xyz(arc(x,y) \wedge arc(x,z) \wedge hc(x,y) \wedge hc(x,z) \supset y = z), \\ \forall xyz(arc(x,y) \wedge arc(z,y) \wedge hc(x,y) \wedge hc(z,y) \supset x = z), \\ \forall xy(hc(x,y) \supset arc(x,y)). \end{array}$$

□

A PC(ID) theory is obtained from the above FO(ID) formulation by a standard process of grounding which eliminates quantifiers by substituting all possible values in their domain.

4 Weak Level Mappings

The reduction of inductive definitions under the well-founded semantics to propositional satisfiability is based on modeling of the computation of the well-founded operator W_P. If we encode precisely the stages induced by the operator W_P (the function $|\cdot|_{W_P}$), we can obtain a faithful reduction. However, such reduction will be very expensive in terms of its size. Instead, we use a function $|\cdot|_{wk}\colon WF(P) \to \mathbb{N}$ called a weak level mapping which only captures the ordering of literals induced by $|\cdot|_{W_P}$: if $|l_1|_{W_P} < |l_2|_{W_P}$ then $|\cdot|_{wk}$ should satisfy $|l_1|_{wk} < |l_2|_{wk}$. Since there will be many such functions, the reductions which we obtain is non-faithful.

Definition 3. *Let P be a logic program and L a consistent set of literals. A* weak level mapping *is any function $|\cdot|_{wk}\colon L \to \mathbb{N}$ such that L and $|\cdot|_{wk}$ satisfy the following conditions.*

1. *For every positive literal $a \in L$, there exists a rule $r \in P$ such that $hd(r) = a$, $body(r) \subseteq L$ and for every literal $l_i \in body(r)$, $|a|_{wk} > |l_i|_{wk}$.*
2. *For every negative literal $\neg a \in L$ and for every rule $r \in P$ with $hd(r) = a$, there exists a literal $l_i \in body(r)$ such that $\neg l_i \in L$, and if a depends negatively on itself then:*
   (a) $|\neg a|_{wk} \geq |\neg l_i|_{wk}$ if l_i is positive;
   (b) $|\neg a|_{wk} > |\neg l_i|_{wk}$ if l_i is negative.

Example 2. Consider the following logic program:

$$\begin{array}{l} a \leftarrow a \wedge c \\ b \leftarrow a \\ c \leftarrow \neg\, b. \end{array}$$

Its well-founded model is $\{\neg a, \neg b, c\}$ and is computed as:

$$W_P^1 = W_P(\emptyset) = \{\neg a, \neg b\}$$
$$W_P^2 = W_P(W_P^1) = \{\neg a, \neg b, c\}$$

so the stages of the literals in the well-founded model are

$$|\neg a|_{W_P} = 1, \qquad |\neg b|_{W_P} = 1, \qquad |c|_{W_P} = 2.$$

One possible weak level mapping is the following:

$$|\neg a|_{wk} = 2, \qquad |\neg b|_{wk} = 3, \qquad |c|_{wk} = 5.$$

The above example illustrates several differences between a weak level mapping $|\cdot|_{wk}$ and the level mapping $|\cdot|_{W_P}$:

- the smallest value of $|\cdot|_{wk}$ may be higher than 1;
- $|\cdot|_{wk}$ may have gaps between levels;
- literals that have the same value of $|\cdot|_{W_P}$ may have different values of $|\cdot|_{wk}$.

The first result is soundness of the weak level mappings with respect to the well-founded semantics.

Proposition 2. *Let P be a logic program and L a consistent set of literals. If there exists a weak level mapping with domain L then $L \subseteq WF(P)$.*

For models of inductive definitions, we are interested only in two-valued well-founded models, so we have the following corollary.

Corollary 1. *Let P be a logic program and L a consistent set of literals which is two-valued, i.e., for every $a \in At$ either $a \in L$ or $\neg a \in L$. If there exists a weak level mapping with domain L then L is the well-founded model of P.*

Note that under the conditions of the corollary, the set L is also the unique stable model of P.

The next proposition is a completeness result, meaning that there exists a weak level mapping whose domain is the well-founded model of a program.

Proposition 3. *Let P be a logic program. The level mapping $|\cdot|_{W_P} : WF(P) \to \mathbb{N}$ is a weak level mapping.*

5 Reduction

The reduction from PC(ID) to SAT is obtained by encoding, as a set of clauses, the conditions on weak level mappings from Definition 3.

The reduction T_{WEAK} of a PC(ID) theory $\langle \mathcal{D}, \mathcal{A} \rangle$ to a CNF formula is defined as

$$T_{\text{WEAK}}(\langle \mathcal{D}, \mathcal{A} \rangle) = \Big(\bigcup_{D \in \mathcal{D}} T_{\text{WEAK}}(D) \Big) \cup \mathcal{A}$$

where the reduction $T_{\text{WEAK}}(D)$ of a definition D is defined as

$$T_{\text{WEAK}}(\mathcal{D}) = \bigcup_{a \in def(D)} \{\text{clauses } (2), \ldots, (9)\}.$$

Clauses $(2), \ldots, (9)$ are defined as follows. Let a be an atom and $r_1 \colon a \leftarrow B_1, \ldots, r_n \colon a \leftarrow B_n$ be all the rules in D with a in the head. For every rule r_i we introduce a new propositional variable r_i and the clause:

$$a \supset r_1 \vee \ldots \vee r_n. \tag{2}$$

For every rule $r_i \colon a \leftarrow B_i$ partition the literals in B_i in the set $def(r_i) = body(r_i) \cap def(D)$ of literals which are defined in D and the set $open(r_i) = body(r_i) \setminus def(r_i)$ of literals which are open in D. Then we add the clauses:

$$r_i \supset l_j \qquad \text{for every } l_j \in body(r_i) \tag{3}$$

$$r_i \supset |a|_{wk} > |l_j|_{wk} \qquad \text{for every } l_j \in def(r_i) \tag{4}$$

If a does not depend negatively on itself, we add the "if" part of r_i:

$$a \subset B_i \ . \tag{5}$$

Otherwise, if a depends negatively on itself, we introduce a new variable c_j for every literal $l_j \in def(r_i)$ and add the following clauses:

$$a \vee \bigvee_{l_j \in def(r_i)} c_j \vee \bigvee_{l_j \in open(r_i)} \neg l_j \tag{6}$$

$$c_j \supset \neg l_j \qquad \text{for every } l_j \in def(r_i) \tag{7}$$

$$c_j \supset |a|_{wk} \geq |l_j|_{wk} \qquad \text{for every pos. literal } l_j \in def(r_i) \tag{8}$$

$$c_j \supset |a|_{wk} > |l_j|_{wk} \qquad \text{for every neg. literal } l_j \in def(r_i) \tag{9}$$

Clauses (2)–(4) encode the first condition of weak level mappings (justification of true atoms) while clauses (5)–(9) encode the second condition (justification of false atoms). In case when there is no recursion through negation, false atoms are justified by (the contra-positive of) clause (5). Notice that by resolving (6) with the clauses (7) for all $c_j \in def(r_i)$ we obtain the clause

$$a \vee \bigvee \{\neg l_j \colon l_j \in body(r_i)\}$$

which is equivalent to (5).

We now explain the encoding of the comparisons between level mappings used in the definition of the reduction. The weak level mapping $|\cdot|_{wk} \colon At \to [1, n]$ is encoded by a set of vectors $\vec{a} = a_k, \ldots, a_1$ of propositional variables for each $a \in At$, representing the binary encoding of the value of $|a|_{wk}$. The length k of the vectors is $k = \lceil 1 + \log_2 n \rceil$. Since the well-founded operator W_P reaches a fixpoint in at most $|At|$ steps[3] we can take $n = |At|$. For two vectors $\vec{a}$ and $\vec{b}$ the encoding of $|a|_{wk} < |b|_{wk}$ is given by the following logic program $LT(\vec{a}, \vec{b}) =$

[3] A tighter bound on the number of steps is the length of the longest path in any strongly connected component [2], however this is an NP-complete problem.

$$lt(a,b)_i \leftarrow \neg a_i \wedge b_i \qquad i \in [1,k]$$
$$lt(a,b)_i \leftarrow a_i \wedge b_i \wedge lt(a,b)_{i-1} \qquad i \in [2,k]$$
$$lt(a,b)_i \leftarrow \neg a_i \wedge \neg b_i \wedge lt(a,b)_{i-1} \qquad i \in [2,k]$$

For a variable assignment $v\colon \{a_1,\ldots,a_k\} \to \{0,1\}$ we denote with $v(\vec{a})$ the number whose binary representation is $v(a_k)\ldots v(a_1)$.

Lemma 1. *Let $\vec{a} = a_k,\ldots,a_1$ and $\vec{b} = a_k,\ldots,a_1$ be two vectors of propositional variables and let v be a variable assignment for $\vec{a}$ and $\vec{b}$. Then*

- *$v \models lt(a,b)_k \wedge only\_if(LT(\vec{a},\vec{b}))$ implies $v(\vec{a}) < v(\vec{b})$;*
- *$v \models \neg lt(b,a)_k \wedge th(LT(\vec{b},\vec{a}))$ implies $v(\vec{a}) \leq v(\vec{b})$.*

So, in the clauses of the reduction $|a|_{wk} < |b|_{wk}$ stands for the variable $lt(a,b)_k$ and $only\_if(LT(\vec{a},\vec{b}))$ is added to the output of the translation. Similarly, $|a|_{wk} \leq |b|_{wk}$ stands for the variable $\neg lt(b,a)_k$ and $th(LT(\vec{b},\vec{a}))$ is added to the output of the translation.

Theorem 1. *Let $\mathcal{T}$ be a PC(ID) theory. Then: (i) for every model M of $\mathcal{T}$ there exists a model M' of $T_{\text{WEAK}}(\mathcal{T})$ such that $M = M' \cap At$; and (ii) the restriction $M' \cap At$ of every model M' of $T_{\text{WEAK}}(\mathcal{T})$ is a model of $\mathcal{T}$.*

To reduce the size of the translation, we split every definition $D \in \mathcal{D}$ in a PC(ID) theory $\langle \mathcal{D}, \mathcal{A}\rangle$ to a set of definitions $\{D_S\colon S \in SCC(D)\}$ — one for each strongly connected component S of D. This is an equivalence preserving transformation as shown in [6].

Proposition 4. *Let $\mathcal{T} = \langle \mathcal{D}, \mathcal{A}\rangle$ be a PC(ID) theory. The number of literals in $T_{\text{WEAK}}(\mathcal{T})$ is $O(size(\mathcal{T}) \times \log m)$ where $size(\mathcal{T})$ is the total number of literals in $\mathcal{T}$ and $m = \max\{|def(D)|\colon D \in \mathcal{D}\}$.*

5.1 Stable Semantics

The T_{WEAK} reduction can be adapted for logic programs under the stable semantics by assuming that the program does not contain a recursion over negation (even if it actually does). The intuition is that, under the stable semantics, no justification is necessary for atoms which are false — they can be assumed "false by default". So, clauses (6)–(9) are never used and clause (5) is used instead. In addition, atoms without rules should be set to false. Let T_{SM} denote this modified translation:

$$T_{\text{SM}}(P) = \bigcup_{a \in def(P)} \{\text{clauses } (2),\ldots,(5)\} \cup \bigcup_{a \in open(P)} \{\neg a\}.$$

Theorem 2. *Let P be a logic program. Then: (i) for every stable model M of P there exists a model M' of $T_{\text{SM}}(P)$ such that $M = M' \cap At$; and (ii) the restriction $M' \cap At$ of every model M' of $T_{\text{SM}}(P)$ is a stable model of P.*

Similarly to the T_{WEAK} reduction, the T_{SM} reduction can be applied separately for each strongly connected component of the program. It is possible to further improve the reduction by exploiting a well-known theorem by Fages [9] stating that the stable models of a tight[4] logic program P are equal to the models of $comp(P)$. The obvious application of this result is to change the reduction such that for a SCC which is tight to compute its completion. However, even for a non-tight component it is possible to avoid assigning level mapping to some atoms. This is done by refining the condition of tightness to individual atoms.

Definition 4. *Let P be a logic program. An atom a is* tight *if every cycle in the dependency graph of P which passes through a contains negation. A logic program P is* tight *if every atom in P is tight.*

The above definition of a tight logic program is equivalent to the one of [9] and the one of Erdem and Lifschitz [8] based on level mappings.

Example 3. Consider the following program:

$r_1 : p \leftarrow \neg q$

$r_2 : p \leftarrow p \wedge r$

$r_3 : q \leftarrow \neg p.$

The atom q is tight, however the atom p is not because of the positive dependence of p on itself from the second rule. Consequently, the program is not tight. □

The importance of tight atoms is that in the reduction it is not necessary to assign to them a level mapping. This optimization is easy to implement by changing the definition of the set $def(r)$ used in clause (4) as follows:

$$def(r) = \{a \in pos(r) : SCC(a) = SCC(hd(r)) \text{ and } a \text{ is not tight}\}.$$

Example 4. Applying the optimized reduction to the program from Example 3 we obtain the following theory:

$p \supset r_1 \vee r_2$	by rule (2)
$r_1 \supset \neg q$	by rule (3)
$p \subset \neg q$	by rule (5)
$r_2 \supset \neg p$	by rule (3)
$r_2 \supset \lvert p\rvert_{wk} > \lvert p\rvert_{wk}$	by rule (4)
$r_2 \supset r$	by rule (3)
$p \subset p \wedge r$	by rule (5)
$q \supset r_3$	by rule (2)
$r_3 \supset \neg p$	by rule (3)
$q \subset \neg p$	by rule (5)

$\neg r$

We conclude the section by giving a result on the size of the reduction.

[4] Originally called *positive-order-consistent*.

Proposition 5. *Let P be a logic program. The number of literals in $T_{\text{SM}}(P)$ is $O(size(P) \times \log|At|)$ where $size(P)$ is the total number of literals in P.*

6 Experiments

To test the different reductions we implemented a prototype system in perl called IDSAT[5]. As a front-end we used LPARSE for grounding, however only PC(ID) theories consisting of a single definition D can be encoded in the input language of LPARSE and the set of sentences must be written as a set of integrity constraints. To be able to support correctly open predicates we needed to make some changes to LPARSE. The problem is that under the stable semantics, predicates without definition are assumed to be false (while they are open for PC(ID)). Consequently, LPARSE will remove all rules which have such predicates in the body. After the change such rules were left in the output.

All experiments were performed on a 2GHz Intel Pentium 4 PC with 256MB memory running Linux with kernel 2.4.20. We report the number of seconds for finding first solution (or showing that no solution exists) averaged over 5 runs. Since all systems use LPARSE for grounding, LPARSE time is not included. A "-" means that no solution was found within 2 hours. We used SMODELS version 2.28 [20] and ASSAT version 2.02 [14]. The LP2SAT is an implementation of the faithful reduction of Janhunen [11]. For all SAT based systems we used the SAT solver *siege* variant 4 [19].

Table 1 reports the results for the Hamiltonian Cycle problem. For the T_{WEAK} reduction, we used the formulation in Example 1, and for all other systems (including the T_{SM} reduction), the encoding by Niemelä [18]. The second column "HC" shows whether the given graph has a Hamiltonian cycle or not. ASSAT is the only system which is able to solve all problems. The three reductions are faster than ASSAT on smaller graphs (p20–p30) where ASSAT needs to call a SAT solver several times before finding the first solution. However, on larger graphs, ASSAT is the fastest SAT based system. The performance of the three off-line reductions (T_{WEAK}, T_{SM}, and LP2SAT), is close to that of ASSAT with LP2SAT being the fastest among them. It is interesting that all three reductions to SAT have problems with graphs without a Hamiltonian cycle.

Table 2 compares the input sizes for the formulation of the problem in PC(ID) and in ASP (columns "input rules") and the increase in size for all reductions to SAT (columns "T_{WEAK} factor", "T_{SM} factor", "LP2SAT factor", and "ASSAT factor"). The input size is measured as the total number of rules and sentences for PC(ID) and the total number of rules for ASP after grounding with `lparse -d none`. The ASSAT reduction is measured as the number of clauses in the completion plus the number of loop formulas and is averaged over the 5 runs. The two "SCCs" columns give the number of strongly connected components with size greater than 1 for the two formulations. The additional SCCs for the ASP formulation come from the rules defining $hc(x, y)$ as an "open" predicate. The presence of more than one additional SCC for graphs 2xp30 and 4xp20

[5] `http://nik.pelov.name/idsat/`

Table 1. Times for the Hamiltonian Cycle problem: pN — a graph with N nodes[6]; KxpN.i — K copies of the the graph pN with some extra edges added, i stands for a variation of the graph[7]

		PC(ID)	ASP			
Graph	HC	T_{WEAK}	T_{SM}	LP2SAT	ASSAT	SMODELS
p20	y	0.84	0.46	0.19	5.03	0.07
p25	y	0.77	0.62	0.94	9.27	0.09
p29	y	0.97	1.15	0.20	9.59	1.19
p30	y	0.96	1.33	2.10	13.28	0.13
2xp30	n	-	-	-	0.21	0.18
2xp30.1	y	188.62	237.84	197.56	84.04	0.33
2xp30.2	y	168.53	212.76	112.86	112.17	-
2xp30.3	y	218.63	169.26	133.15	103.83	-
2xp30.4	n	-	-	-	98.91	-
4xp20	n	208.07	266.18	190.85	0.29	0.22
4xp20.1	n	266.30	314.07	183.66	2.45	-
4xp20.2	y	196.69	147.08	81.37	29.43	0.43
4xp20.3	n	-	-	202.87	5.16	0.22

Table 2. Size of the input and output of the reductions

	PC(ID)			ASP				
Graph	SCCs of size > 1	input rules	T_{WEAK} factor	SCCs of size > 1	input rules	T_{SM} factor	LP2SAT factor	ASSAT factor
p20	1	950	2.89	2	1048	3.79	18.64	2.59
p25	1	1297	2.82	2	1425	3.73	18.06	4.22
p29	1	1577	2.78	2	1729	3.70	17.73	5.02
p30	1	1644	2.78	2	1802	3.70	17.73	5.43
2xp30	2	3288	2.86	4	3604	3.77	18.34	1.47
2xp30.1	2	3350	2.83	3	3668	3.74	18.07	7.20
2xp30.2	1	3330	3.12	2	3648	4.01	21.24	8.96
2xp30.3	1	3330	3.12	2	3648	4.01	21.24	9.14
2xp30.4	1	3330	3.12	2	3648	4.01	21.24	2.01
4xp20	4	3800	3.04	8	4192	3.92	19.78	1.51
4xp20.1	4	3857	3.01	5	4252	3.89	19.55	1.50
4xp20.2	1	3876	3.61	2	4272	4.45	26.18	5.35
4xp20.3	1	3884	3.61	2	4280	4.44	26.13	1.69

indicates that they consist of several disconnected components and hence they do not contain a Hamiltonian cycle.

Comparing the size of the non-faithful reduction T_{SM} and the faithful reduction LP2SAT confirms the conjecture that faithful reductions are much bigger — for this example about 5 times. However, the performance of the SAT solver is not proportional to the size of the theory. Even to the contrary, the LP2SAT reduction has a consistently better performance.

[6] `http://www.tcs.hut.fi/Software/smodels/tests/lp-csp-tests.tar.gz`

[7] `http://assat.cs.ust.hk/Assat-2.0/hc-2.0.html`

7 Conclusion

In this paper we reported on an implementation of a system for finding models of the propositional fragment of FO(ID) by doing a reduction to propositional satisfiability. Our experiments showed that, on satisfiable problems, this approach is competitive with ASP systems — both direct implementations like SMODELS and SAT based systems like ASSAT and LP2SAT.

We also showed how the reduction from PC(ID) to SAT can be adapted for the stable model semantics. In our experiments we compared this reduction to the faithful reduction LP2SAT [11] and confirmed our hypothesis that developing a one-to-one reduction is much more costly in terms of its size. However, despite this difference in size, the faithful reduction performed better on all examples. To better understand the performance of the different reductions it is necessary to do experiments with other problems and different SAT solvers.

An interesting direction for future research is to follow the approach of ASSAT and CMODELS-2 and try to define loop formulas for two-valued well-founded semantics.

References

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

[2] R. Ben-Eliyahu and R. Dechter. Propositional semantics for disjunctive logic programs. *Annals of Mathematics and Artificial Intelligence*, 12(1–2):53–87, 1994.

[3] R. J. Brachman and H. Levesque. Competence in knowledge representation. In *Proc. of the National Conference on Artificial Intelligence*, pages 189–192, 1982.

[4] M. Denecker. Extending classical logic with inductive definitions. In *Computational Logic, First International Conference*, volume 1861 of *Lecture Notes in Computer Science*, pages 703–717. Springer, 2000.

[5] M. Denecker and E. Ternovska. Inductive situation calculus. In *Principles of Knowledge Representation and Reasoning: Proc. of the 9th International Conference*, pages 545–553. AAAI Press, 2004.

[6] M. Denecker and E. Ternovska. A logic of non-monotone inductive definitions and its modularity properties. In *Logic Programming and Nonmonotonic Reasoning: 7th International Conference*, volume 2923 of *Lecture Notes in Computer Science*, pages 47–60. Springer, 2004.

[7] M. Denecker, D. Theseider Dupré, and K. Van Belleghem. An inductive definition approach to ramifications. *Linköping Electronic Articles in Computer and Information Science*, 3(7):1–43, 1998. `http://www.ep.liu.se/ea/cis/1998/007/`.

[8] E. Erdem and V. Lifschitz. Tight logic programs. *Theory and Practice of Logic Programming*, 3(4–5):499–518, 2003.

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

[10] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Logic Programming, 5th International Conference and Symposium*, pages 1070–1080, 1988.

[11] T. Janhunen. Representing normal programs with clauses. In *Proc. of the 16th European Conference on Artificial Intelligence*, pages 358–362, 2004.

[12] Y. Lierler and M. Maratea. Cmodels-2: SAT-based answer set solver enhanced to non-tight programs. In *Logic Programming and Nonmonotonic Reasoning, 7th International Conference*, volume 2923 of *Lecture Notes in Computer Science*, pages 346–350. Springer, 2004.

[13] F. Lin and J. Zhao. On tight logic programs and yet another translation from normal logic programs to propositional logic. In *International Joint Conference on Artificial Intelligence*, pages 853–858. Morgan Kaufmann, 2003.

[14] F. Lin and Y. Zhao. ASSAT: Computing answer sets of a logic program by SAT solvers. *Artificial Intelligence*, 157(1–2):115–137, 2004.

[15] T. Linke, H. Tompits, and S. Woltran. On acyclic and head-cycle free nested logic programs. In *20th International Conference on Logic Programming*, volume 3132 of *Lecture Notes in Computer Science*, pages 225–239. Springer, 2004.

[16] M. Mariën, D. Gilis, and M. Denecker. On the relation between ID-Logic and Answer Set Programming. In *Logics in Artificial Intelligence, 9th European Conference (JELIA)*, volume 3229 of *Lecture Notes in Computer Science*, pages 108–120. Springer, 2004.

[17] D. Mitchell and E. Ternovska. A framework for representing and solving NP-search problems. In *Proc. of the National Conference on Artificial Intelligence*, pages 430–435, 2005.

[18] 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.

[19] L. Ryan. Efficient algorithms for clause-learning SAT solvers. Master's thesis, Simon Fraser University, Burnaby, Canada, 2004.

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

[21] E. Ternovskaia. Causality via inductive definitions. In *Working Notes of "Prospects for a Commonsense Theory of Causation", AAAI Spring Symposium Series*, pages 94–100, 1998.

[22] E. Ternovskaia. ID-logic and the ramification problem for the situation calculus. In *Proc. of the 14th European Conference on Artificial Intelligence*, pages 563–567, 2000.

[23] A. Van Gelder, K. A. Ross, and J. S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, 1991.

[24] B. Van Nuffelen, A. Cortés-Calabuig, M. Denecker, O. Arieli, and M. Bruynooghe. Data integration ising ID-logic. In *Advanced Information Systems Engineering, Proc. of the 16th International Conference (CAiSE)*, volume 3084 of *Lecture Notes in Computer Science*, pages 67–81. Springer, 2004.

Symbolic Support Graph: A Space Efficient Data Structure for Incremental Tabled Evaluation*

Diptikalyan Saha and C.R. Ramakrishnan

Department of Computer Science,
State University of New York at Stony Brook,
Stony Brook, New York, 11794-4400, U.S.A
{dsaha, cram}@cs.sunysb.edu

Abstract. In an earlier paper, we described a data structure, called support graph, for efficient incremental evaluation of tabled logic programs. The support graph records the dependencies between answers in the tables, and is crucial for efficiently propagating the changes to the tables when facts are deleted. Incremental computation with support graphs are hundreds of times faster than from-scratch evaluation for small changes in the program. However, the graph typically grows faster than the tables themselves, making it impractical to maintain the full support graph for large applications.

In this paper we present a data structure, called symbolic support graph, which represents support information compactly. For a variety of useful tabled logic programs, the size of the symbolic support graph grows no faster than the table size. We demonstrate its effectiveness using a large application: a logic-programming-based points-to analyzer for C programs. The incremental analyzer shows very good scalability in terms of space usage, and is hundreds of times faster than from-scratch analysis for small changes to the program.

1 Introduction

Tabled resolution [5,7,28] has become an important evaluation technique in logic programming. Many implementations of tabling have now emerged [10,20,29,31]. Tabling has enabled us to construct many practical applications— program analysis and verification systems [8,17], in particular— by encoding them as high-level logic programs.

Tabled resolution-based systems evaluate programs by memoizing subgoals (referred to as *calls*) and their provable instances (referred to as *answers*) in a set of tables. During resolution, if a subgoal is present in the call table, then it is resolved against the answers recorded in the corresponding answer table; otherwise the subgoal is entered in the call table, and its answers, computed by resolving the subgoal against program clauses, are also entered in the answer table. For instance, the call and answer tables created when evaluating the query `r(6,X)` over the program in Figure 1(a) is given in Figure 1(b). (The answers in the figure are labeled $a1$, $a2$ etc.)

Tabling opens up the possibility of *incremental* evaluation: when some of a program's facts or rules change, we can recompute only the results affected by the changes, instead of re-evaluating the program from scratch.

* This research was supported in part by NSF grants CCR-020537 and CCR-0311512.

M. Gabbrielli and G. Gupta (Eds.): ICLP 2005, LNCS 3668, pp. 235–249, 2005.

```
:- table r/2.
r(X,Y) :- b(X,Y).
r(X,Y) :- c(X,Z),
          r(Z,Y).

b(1,2). %f1
b(6,2). %f2
b(6,4). %f3

c(1,6). %f4
c(3,6). %f5
c(3,1). %f6
c(6,3). %f7
```

(a)

Calls	Answers	Supports
r(6,X)		
	[a1] **r(6,2)**	[s1] {**b(6,2)**}, [s10] {**c(6,3),r(3,2)**}
	[a2] r(6,4)	[s2] {b(6,4)}, [s11] {c(6,3),r(3,4)}
r(3,X)		
	[a3] **r(3,2)**	[s3] {**c(3,6),r(6,2)**}, [s8] {c(3,1),r(1,2)}
	[a4] r(3,4)	[s4] {c(3,6),r(6,4)}, [s9] {c(3,1),r(1,4)}
r(1,X)		
	[a5] r(1,2)	[s5] {b(1,2)}, [s6] {**c(1,6),r(6,2)**}
	[a6] r(1,4)	[s7] {c(1,6),r(6,4)}

(b) (c)

Fig. 1. Example program (a); calls and answers generated when evaluating query `r(6,X)` (b); and supports for the query evaluation (c)

Background: Incremental evaluation of tabled programs is closely related to the well-investigated problem of materialized view maintenance in databases [12, e.g.]. Most of these works handle two kinds of changes to a program, namely, insertion and deletion of facts; update is treated as deletion followed by insertion. Incremental processing of deletion is more challenging than that of addition, especially for maintaining recursively defined views. This paper focusses solely on incremental processing of deletion.

The *DRed* algorithm [11], which subsumes the other recursive view maintenance algorithms, first deletes all answers that may be affected by the deleted facts (the deletion phase). The second (rederivation) phase attempts to rederive the deleted answers without using the deleted facts. For instance, consider the deletion of fact `b(6,2)` from the program in Figure 1(a). Since there is one derivation of `r(6,2)` that contains `b(6,2)`, the DRed algorithm deletes `r(6,2)`. Similarly, `r(3,2)` and `r(1,2)` are also deleted. In the second phase, `r(1,2)` is rederived due to `b(1,2)`. Consequently, `r(3,2)` and `r(6,2)` are also rederived, thereby rederiving all the three originally deleted answers. The deletion-rederivation strategy appears to be universal for handling incremental deletion: it also appears in independently-developed incremental algorithms for program analysis [16,30] and model checking [25].

Two factors make the DRed algorithm impractical. First, the deletion phase uses program clauses to propagate the deletions. Second, many of the deleted answers are rederived in the second phase, again by applying program clauses. In [21], we proposed a solution to these two problems, as described below.

Support Graph: An instance of a rule that can be used to derive an answer is known as a *support* for that answer. For instance, the supports for the answers used to evaluate the query `r(6,X)` over the program in Figure 1(a), are given in Figure 1(c). The supports in the figure are labeled $s1$, $s2$, etc. A support graph has answers, facts and supports as vertices; the support graph corresponding to Figure 1(c) is shown in Figure 2. An "answer" edge connects a support (circled in the figure) with the answer it supports (shown with white arrowheads in the figure). An answer/fact contained in a support is connected to the support with a "uses-of" edge (filled arrowheads in the figure).

Our incremental algorithm [21] is based on DRed [11] and has two phases. In the first phase, when a fact is deleted, the supports containing the fact are marked. When a

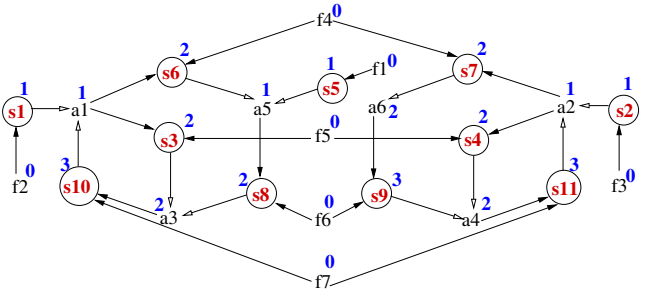


Fig. 2. Support graph for answers to query `r(6,X)` over example program in Figure 1(a)

support is marked, the answers it supports are marked; the marks are then further propagated, alternately marking supports and answers. Note that marking is done without using the program rules. The propagation of marks is optimized by (i) annotating the first support used to derive an answer as its *primary* support, and (ii) marking an answer only when its primary support is marked. In Figure 1(c), the primary support is listed first. In our running example, the deletion of `b(6,2)` will mark `r(6,2)`, and consequently `r(3,2)`. The marked supports and answers due to this deletion are shown in boldface in Figure 1(c). Note that `r(1,2)` is not even marked.

More importantly, support graphs significantly simplify rederivation. Observe that after the marks have been completely propagated, if a marked answer has unmarked supports, then it is derivable using any of those supports. Continuing with our running example, we can thus remove the mark on `r(3,2)`. Consequently, the marks on support $\{$`c(6,3)`, `r(3,2)`$\}$, and hence `r(6,2)` are removed. Note that rederivation can be done without using the program clauses.

The Problem: Our incremental algorithm in [21] based on support graphs shows very good time performance: incremental evaluation times for small changes are typically *0.1%* of the from-scratch evaluation time for most programs. However, support graphs are usually very large, and the memory overhead makes it impossible to maintain the full support graph for large applications.

In [22] we proposed an algorithm that kept only a limited number of supports for each answer, making its space requirements linear in the number of answers. The space savings and scalability however come at the price of increased rederivation time. Since all supports are not stored, an answer may have a derivation even when all of its stored supports are marked. Hence, we need to re-evaluate the program clauses to check if the answer can be rederived. The time penalty can be high: incremental evaluation time (for small changes) may be as much as 15% of that of from-scratch evaluation.

This raises an interesting question: *Can we store the entire support graph, which eases rederivation and significantly improves incremental evaluation time, without incurring a prohibitive space overhead*? We address this problem in this paper.

Our Solution: The key to storing the entire support graph is to make use of explicit sharing inherent in the supports. Consider the two answers to call `r(3,X)`, `r(3,2)` and `r(3,4)`, and their supports $s3$ and $s4$ respectively. Observe that the two supports share `c(3,6)`. Also notice that the literals which make the two supports different, i.e.

Call	Symbolic Supports
r(6,X)	⟨r(6,X),{},b(6,X)⟩, ⟨r(6,X),{c(6,3)},r(3,X)⟩
r(3,X)	⟨r(3,X),{c(3,6)},r(6,X)⟩, ⟨r(3,X),{c(3,1)},r(1,X)⟩
r(1,X)	⟨r(1,X),{},b(1,X)⟩, ⟨r(1,X),{c(1,6)},r(6,X)⟩

Fig. 3. Symbolic supports for query evaluation over the example program in Figure 1(a)

r(6,2) and r(6,4), are answers to the call r(6,X). Thus two supports for answers to r(3,X) can be represented in intensional form as: c(3,6), r(6,X). This intensional form is represented in a *symbolic support*, which consists of three parts, namely, the set of answers supported (e.g. r(3,X), the common part of all the supports (e.g. c(3,6), and the call whose answers distinguish the supports (e.g. r(6,X)). Now, when an answer to r(6,X), say r(6,2) is deleted, we can compute, using the symbolic support, that r(3,2) may be affected. A symbolic support captures dependencies between certain calls while our earlier notion of supports captured dependencies between answers. By lifting this to the level of calls, a symbolic support compactly represents multiple supports.

The symbolic supports for the evaluation of query r(6,X) over the program in Figure 1(a) appears in Figure 3. Marking can be readily done using the symbolic supports. Given a marked answer (e.g. r(6,2)), we first compute the substitution for the variables in a support corresponding to it (e.g. r(6,X), X = 2), and use this substitution to find the supported answer (e.g. r(3,2)). When the intensional form does not contain any join operations, we can compute the answer dependencies from the symbolic support in time proportional to the answer size.

Contributions: We propose Symbolic Support Graph (SSG), a data structure for space-efficient and time-efficient incremental evaluation of tabled logic programs (Section 3). We give efficient algorithms for incremental evaluation with SSGs (Section 4). SSGs grow no faster than the tables for an important class of tabled programs and queries (Section 5). In practice, SSGs take much less space than full support graphs, and yet show time performance comparable to the latter (Section 6). We demonstrate the scalability of incremental evaluation using a points-to analyzer for C programs as an example. Our incremental analyzer scales to programs with over 60K lines of code. In many cases, the size of SSG is even smaller than that of partial support graphs used in [22]. Thus, SSGs enable incremental evaluation of large, realistic applications.

The relationship between this paper and the previous work is explored in Section 7. In this paper, we describe SSGs for incremental evaluation when facts are deleted from a definite tabled logic program. The use of SSGs is orthogonal to other issues in incremental evaluation, such as the handling of insertion of facts/rules, deletion of rules, updates, and stratified negation. A brief discussion of these issues appear in Section 8.

2 Preliminaries

We now formally define the notions of supports and support graph. We consider definite logic programs, and partition the predicates into *intensional* and *extensional* predicates. Extensional predicates are defined solely by facts. For simplicity of notation, we assume

that only the definitions of extensional predicates may be deleted. The techniques in this paper can be generalized to handle the deletion of rules along the lines described in [21].

Definition 1 (Support). Let P be a definite logic program, and let T be a set of answer tables obtained when evaluating a query γ over P. A set $\{b_1, b_2, \ldots, b_n\}$ is called a support of an answer a of γ if there exists a clause in P of the form $\alpha :- \beta_1, \beta_2, \ldots, \beta_n$ and a substitution θ, such that $\alpha\theta = a$, and, for all $i \in [1, n]$ $\beta_i\theta = b_i$ and b_i is an instance of an answer in T or a fact in P.

A support graph maintains the relationships between the answers and supports generated during query evaluation.

Definition 2 (Support graph). Let P be a definite logic program, and let T be a set of answer tables obtained when evaluating a query γ over P.

The support graph for the evaluation of γ is a directed graph (V, E) where V contains the facts in P, answers in T and their supports. The set of edges E is such that

- $(b_i, s) \in E$ for all supports $s \in V$ such that $s = \{b_1, b_2, \ldots b_n\}$, and for all $i \in [1, n]$. We say that $s \in b_i.\textit{uses\_of}$ and $b_i \in s.\textit{part\_of}$.
- $(s, a) \in E$ for all $a \in T$ and $s \in V$ such that s is a support of a. We say that $s.\textit{answer} = a$ and $s \in a.\textit{support}$.

The primary support of an answer is the first support used to derive the answer in some least fixed point computation procedure. It follows from the property of least fixed point computations that the primary support will be independent of the answer itself. We generalized this to *acyclic supports* in [22], defined using the notion of *derivation length* described below:

$$v.dl = \begin{cases} 0 & \text{if } v \text{ is a fact} \\ 1 + \max\{a.dl \mid v \in a.\textit{uses\_of}\} & \text{if } v \text{ is a support} \\ s.dl \mid s \text{ is the primary support of } v & \text{if } v \text{ is an answer} \end{cases}$$

The derivation length represents the height of a proof tree for an answer. Note that if the derivation length of a support s is no greater than its supported answer a, then s has a derivation independent of a. A support s is *acyclic* if $s.dl \leq s.\textit{answer}.dl$. Thus an answer need not be marked in the first phase until all of its acyclic supports are marked. In our running example in Figure 2, we have annotated the vertices with their derivation lengths. Based on the derivation lengths, we determine $s1$, $s2$, $s3$, $s4$, $s5$, $s7$ and $s8$ as acyclic supports. Thus deletion of $f2$ causes only $a1$ to be marked. Note, however, an acyclic support may not remain acyclic after rederivation. For instance, $a1$ is rederived due to $s10$ which now becomes its acyclic support by updating $a1.dl$ to 3. Consequently, the derivation lengths $s3$ and $s6$ are changed to 4 making $s3$ non-acyclic.

3 Symbolic Support Graphs

We now formally define the notion symbolic supports, and describe the data structure to represent symbolic support graphs.

Definition 3 (Symbolic Support). Let P be a definite logic program with a set of facts F, and let C and A be a set of call and answer tables respectively, obtained when evaluating a query ξ over P. The triple $S = \langle h, s, d\rangle$ is a symbolic support for a call $\gamma \in C$ if there is a clause in P of the form $\alpha :- \beta_1, \beta_2, \ldots, \beta_{n-1}, \beta_n$ and a substitution θ such that

1. h, called the *head* of S, is such that $\alpha\theta = h$;
2. s, called the *static part* of S, is such that $s = \{b_1, b_2, \ldots b_{n-1}\}$, and $\forall i \in [1, n-1]$, $b_i = \beta_i\theta$ and $b_i \in A \cup F$;
3. d, called the *dynamic part* of S, is such that $\beta_n\theta = d$.

Note that a symbolic support is shared between a non-empty set of answers of a call. The set of non-symbolic supports represented by a symbolic support S are called embedded supports of S, defined below.

Definition 4 (Embedded Supports). Let P be a definite logic program with set of facts F, and let A be answers in the tables obtained when evaluating a query γ over P. A non-symbolic support s is embedded in a symbolic support $S = \langle h, s', d\rangle$ if there is a substitution σ such that $s.answer = h\sigma \in A$, $s = s' \cup \{d\sigma\}$, and $d\sigma \in A \cup F$.

Given a symbolic support $S = \langle h, s, d\rangle$, then answer a' is said to be a *supported answer* for an answer/fact a w.r.t. S if there is a substitution σ such that $a = d\sigma$ and $a' = h\sigma$. In that case, we also say that a is a *supporting answer* w.r.t. S.

When the mark on an answer is propagated, we need to find an embedded support that contains this answer. For instance, consider our running example and its symbolic supports in Figures 1(a) and 3 respectively. If `r(6,2)` is marked, since it is an instance of `r(6,X)`, we need to mark supports embedded in $\langle$`r(1,X)`, $\{$`c(1,6)`$\}$, `r(6,X)`$\rangle$ and $\langle$`r(3,X)`, $\{$`c(3,6)`$\}$, `r(6,X)`$\rangle$. This lookup can be efficiently done if the dynamic part of a symbolic support is a tabled call. Moreover, we can maintain, for each tabled call, the set of symbolic supports that contain it. If the dynamic part of the symbolic support is not a tabled call, then we need to maintain additional indexing structures to find the embedded supports. *Hence we do not use a symbolic support when its dynamic part is not a tabled call, and use non-symbolic supports instead.*

Symbolic support graphs (SSG) are an extension of the support graphs that has calls, answers, symbolic as well as non-symbolic supports as vertices and the relationships between them as edges. The edges in an SSG are described below.

- *uses_of*, *part_of*, *support*, *answer*: as in Definition 2.
- *set_uses_of*: If a fact or an answer a is in the static part of a symbolic support *SS* then there is a *set_uses_of* edge from a to *SS*.
- *set_uses_of_call* and *dynamic_call*: If a call C is the dynamic part of a symbolic support *SS* then there is a *set_uses_of_call* edge from C to *SS*. There is also a *dynamic_call* edge from *SS* to C.
- *supported_call* and *symsupport*: If a symbolic support *SS* supports a nonempty set of answers of a call C then there is a *supported_call* edge from *SS* to C, and a *symsupport* edge from C to *SS*.
- *answers* and *subgoal*: If A is the set of answers for call C, then there is a *answers* edge from C to elements of A and a *subgoal* edge from each element of A to C.

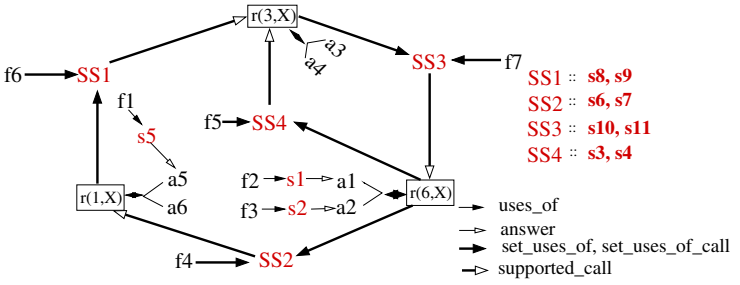


Fig. 4. Symbolic support graph for answers to query `r(6,X)`

The SSG corresponding to the supports in Figure 2 is shown in Figure 4. Note that any tabling engine will give unique identities to each tabled call (e.g. the *subgoal frame* in the SLG-WAM [7]) and tabled answers. We use these identifiers in our implementation of the SSG to denote calls and answers (we use terms to represent calls in examples, for clarity). The information about the variables in the head and the dynamic part, needed to compute the embedded supports, is also kept in a symbolic support. This implementation detail is not shown in the examples.

The *set_uses_of*, *set_uses_of_call*, and *supported_call* edges in an SSG are required for propagation of marks and rederivation. They are analogues of *uses_of* and *answer* in a support graph. The *symsupport* edges are used to adjust the derivation length of an answer after rederivation. Finally, *dynamic_call* is used to compute the embedded supports of a symbolic support.

In addition, for each answer we maintain the total number of unmarked supports in attribute *total_support_count* and the number of unmarked acyclic supports in attribute *acyclic_support_count*. These attributes counts the number of embedded supports represented by a symbolic support. In Figure 4, *total_support_count* of $a1$ is 2 and *acyclic_support_count* of $a1$ is 1.

4 The Incremental Table Maintenance Algorithm

We now describe the incremental algorithm for maintaining the tables using symbolic supports. The algorithm extends the one in [21] and handles graphs with a mixture of symbolic and non-symbolic supports. We have already seen in the previous section how to compute the embedded (non-symbolic) supports for each symbolic support. Note that information such as derivation length and marking are specific to the non-symbolic embedded supports; computing this information based on symbolic supports is the key issue in the algorithm. Note also that the static part of a symbolic support is common to all its embedded supports. Hence we associate the information due to the static part in the symbolic support. For each symbolic support node we maintain an attribute *static_maxdl* that stores the maximum of derivation lengths of the answers and facts in its static part. We use this information to compute the derivation length of each embedded support. Similarly, with each non-symbolic support we maintain an attribute *falsecount* which counts the number of marked answers/facts in the support. With each

symbolic support, we maintain *static_falsecount* which counts the number of marked answers and facts in its static part.

The algorithm has two phases analogous to the two phases of DRed and other incremental recursive-view maintenance algorithms.

Marking Phase. The algorithm for the marking phase is shown in the Figure 5(a). The *falsecount* attributes of symbolic and non-symbolic supports are initialized to zero before the marking phase. An answer is marked by setting its *marked* flag to true; this attribute is initialized to false. The answers to be marked are placed in a queue, and the marking phase ends when the queue is empty. The marked answers are placed in a set *marked_set* for processing in the rederivation phase.

The functions *mark_answer* and *mark_fact* propagates the effect of marking an answer/fact to the supports containing it. The function *mark_fact* propagates the effect of deleting a fact to the supports containing it. In addition *mark_answer* places a mark on the answer. Function *mark_support* marks a support and propagates this mark to the answer supported by it; functions *mark_static* and *mark_dynamic* mark a symbolic support and if needed propagate this mark to the answer(s) supported by it. Note that a (symbolic) support is marked if its (*static_*) *falsecount* is nonzero.

We illustrate the working of the marking phase using the deletion of $f2$ and $f4$ from Figure 4 as an example. A call to mark_fact($f2$) will call mark_support($s1$), and subsequently propagate_mark($a1$, 1). This will decrement $a1$'s total and acyclic support counts (to 1 and 0, resp.), and place $a1$ in the queue. We will call mark_fact($f4$) next. Since $f4$ is in the static part of symbolic support $SS2$, we call mark_static($SS2$). This sets *static_falsecount* of $SS2$ to 1, iterates over the answers of the dynamic part of $SS2$, i.e. `r(6,X)`. The supported answers of $a1$ and $a2$ w.r.t $SS2$ are $a5$ and $a6$, resp., and $a6$ is added to the queue as $a5$ has an unmarked acyclic support $s5$. Note this is equivalent to propagation of marking through $s6$ and $s7$ in support graph based algorithm. Continuing further, we pick up $a1$ for processing from the queue. Since $a1$ appears in the dynamic parts of $SS2$ and $SS4$ *mark_dynamic* is called for both the symbolic supports. However, *mark_dynamic*($SS2$,$a1$) has no effect as its *static_falsecount* is already 1; *mark_dynamic*($SS4$,$a1$) will call *propagate_mark*($a3$, 2) which reduces the total and acyclic support counts of $a3$ to 1 (due to acyclic embedded support $s8$ in $SS1$). Similarly, processing $a6$ from the queue does not mark $a4$ as it has an acyclic embedded support in $SS4$. Thus at the end of marking phase $a1$ and $a6$ are marked.

Rederivation Phase. Each marked answer that has some unmarked support at the end of the marking phase is known to have a proof not involving its previously known acyclic supports. In addition to resetting its mark, we need to compute its new derivation length (due to the new proofs). In our running example we compute the new derivation length of $a1$ by computing derivation length of its unmarked support ($s10$ in SG) embedded in $SS3$. This is done by finding the supporting answer for $a1$ w.r.t. $SS3$, i.e. answer $a3$ ($dl = 3$), and computing the *dl* of the embedded support. When some of the marked answers are rederived, we propagate rederivation using the function *rederive_answer*. Figure 5(b) gives the rederivation algorithm, which is very similar to the marking algorithm.

Correctness. The correctness of the algorithm can be shown by induction on the number of steps needed to derive an answer. The soundness of the algorithm follows

```
mark()
 mark_queue = empty
 ∀ deleted facts f
  mark_fact(f)
 while (mark_queue != empty)
  a = dequeue(mark_queue)
  mark_answer(a)

mark_fact(f)
 ∀ Support s ∈ f.uses_of
  mark_support(s)
 ∀ SymbolicSupport S ∈ f.set_uses_of
  mark_static(S)

mark_answer(a)
 a.marked = true
 ∀ s ∈ a.uses_of
  mark_support(s)
 ∀ S ∈ a.set_uses_of
  mark_static(S)
 subg = a.subgoal
 ∀ S ∈ subg.set_uses_of_call
  mark_dynamic(S, a)

mark_support(s)
 s.falsecount++
 if (s.falsecount == 1)
  propagate_mark(s.answer, s.dl)
```

```
mark_dynamic(S, sourceans)
 (* Propagate via dynamic part *)
 if (S.static_falsecount == 0)
  targetans = supported answer of sourceans w.r.t. S
  support_dl =
    1 + max(S.static_maxdl, sourceans.dl)
  propagate_mark(targetans, support_dl)

mark_static(S)
 (* Propagate via static part *)
 S.static_falsecount++
 if (S.static_falsecount == 1)
  ∀ sourceans ∈ answers(S.dynamic_call)
   if (! sourceans.marked)
    targetans = supported answer of sourceans w.r.t. S
    support_dl =
      1 + max(S.static_maxdl, sourceans.dl)
    propagate_mark(targetans, support_dl)

propagate_mark(ans, support_dl)
 ans.total_support_count−−
 if (ans.dl ≥ support_dl)
  ans.acyclic_support_count−−
  if (ans.acyclic_support_count == 0)
   enqueue(mark_queue, ans)
   marked_set = marked_set ∪ { ans }
```

(a)

```
rederive()
 ∀ ans ∈ marked_set
  if (ans.total_support_count > 0)
   ans.acyclic_support_count
     = ans.total_support_count
   recalculate_dl(ans)
   enqueue(rq, ans)
 ∀ Answer ans ∈ rq
  rederive_ans(ans)

recalculate_dl(targetans)
 spt_max=max{s.dl | s = targetans.support ∧
          s.falsecount == 0}
 espt_max=max{max(S.static_maxdl,ans.dl)+1
  | S ∈ targetans.subgoal.symsupport
  ∧ S.static_falsecount=0
  ∧ ans is a supporting answer of targetans w.r.t S
  ∧ !ans.marked}
 targetans.dl=max(spt_max,espt_max)

rederive_answer(ans)
 ans.marked = false
 ∀ s ∈ ans.uses_of
  rederive_support(s, ans.dl)
 ∀ S ∈ ans.set_uses_of
  rederive_static(S, ans.dl)
 subg=get_subgoal(ans)
 ∀ S ∈ subg.set_uses_of_call
  rederive_dynamic(S, ans)
```

```
rederive_support(s, dlen)
 s.dl = max(s.dl, dlen+1)
 s.falsecount−−
 if (s.falsecount == 0)
  propagate_rederive(s.answer_of, s.dl)

rederive_dynamic(S, sourceans)
 if (S.static_falsecount == 0)
  targetans = supported answer of sourceans w.r.t. S
  dlen = max(S.static_maxdl, sourceans.dl)+1
  propagate_rederive(targetans, dlen)

rederive_static(S, dlen)
 S.static_maxdl = max(S.static_maxdl, dlen)
 S.static_falsecount−−
 if (static_falsecount(S) == 0)
  ∀ sourceans ∈ S.supported_call.answers
   if (!sourceans.marked)
    targetans = supported answer of sourceans w.r.t. S
    dlen' = max(S.static_maxdl, sourceans.dl)+1
    propagate_rederive(targetans, dlen')

propagate_rederive(ans, dlen)
 ans.total_support_count++
 if (ans.acyclic_support_count==0)
  ans.acyclic_support_count = 1
  ans.dl = dlen
  enqueue(rq, ans)
 else
  if (ans.dl ≥ dlen)
   ans.acyclic_support_count++
```

(b)

Fig. 5. Algorithm for Marking (a), Rederivation(b)

from the property that all answers no longer derivable after facts are deleted are marked. The completeness follows from the property that the rederivation removes mark of an answer which has at least one proof independent of the deleted facts. Thus the incremental algorithm can be applied repeatedly applied. The details of the proofs can be found in [24].

5 Space Complexity of Symbolic Support Graphs

In this section we compare the asymptotic size of SSGs with respect to table size and the size of non-symbolic support graphs for a number of useful tabled programs. For purposes of this comparison, we assume that all supports in the SSG are symbolic. The selected programs and the complexity measures are shown in Figure 6. The apparently simple transitive closure programs (`lreach/2` and `rreach/2`) lie at the heart of a remarkable number of applications of tabled logic programming. For instance, verification of safety properties of systems and implementation of inheritance in object-oriented logics reduce to reachability problem. Context-free language reachability, which is the basis for the verification of push-down systems, has rules that resemble the definition of the simpler same-generation (`sg/2`) predicate. A class of useful tabled logic programs not in the figure are those involving negation and aggregation (e.g. dynamic programming problems). In principle, symbolic supports can be used in these cases also, but other aspects of our implementation (e.g. handling of insertions/updates) need extension (see Section 8). Hence we do not include this class in the comparison.

For the graph traversal examples, we assume that the `edge/2` relation defines a graph with v vertices and e edges. We consider a bound-free query to right-recursive transitive closure, say `rreach(`a`,X)`. Tabled evaluation makes $O(v)$ distinct tabled calls to answer this query. Each of these calls can have $O(v)$ answers, and hence the table size is $O(v^2)$. Each answer `rreach(`b`,`c`)` has supports of the form {`edge(`b`,`Y`)`, `rreach(`Y`,`b`)`} where Y ranges over neighbors of b. The number of supports for this answer is bounded by the out-degree of b. Since there are $O(v^2)$ answers, the total number of supports is $O(v * e)$. The symbolic supports associated with call `rreach(`a`,X)` are {`edge(`a`,X)`} and those of the form {`edge(`a`,`Y`)`, `rreach(`Y`,X)`}. Thus there are two symbolic supports for each edge and hence the number of symbolic supports is $O(e)$. Note that SSG grows slower than the tables for this example.

Example programs	Query Modes	Space Complexity		
		Table	SG	SSG
lreach(X,Y):- edge(X,Y). *lreach(X,Y):- lreach(X,Z), edge(Z,Y).*	bb, bf fb, ff	$O(v)$ $O(v^2)$	$O(e)$ $O(v * e)$	$O(v)$ $O(v^2)$
rreach(X,Y):- edge(X,Y). *rreach(X,Y):- edge(X,Z), rreach(Z,Y).*	bf, ff bb, fb	$O(v^2)$ $O(v)$	$O(v * e)$ $O(e)$	$O(e)$ $O(e)$
sg(X,X). *sg(X,Y):- edge(X,Y1),sg(Y1,Y2),edge(Y2,Y).*	all	$O(v^2)$	$O(e^2)$	$O(v * e)$
sg_opt(X,X). *sg_opt(X,Y) :- aux(X,Z),edge(Y,Z).* *aux(X,Y):- edge(X,Z),sg_opt(Z,Y).*	all	$O(v^2)$	$O(v * e)$	$O(v^2)$
Context-Free Language Reachability *N=|nonterms|,G=grammar size*	all	$O(N * v^2)$	$O(G * v^3)$	$O(G * v^2)$

Fig. 6. Space complexity of symbolic support graphs

The asymptotic space complexity for the other examples and queries in Figure 6 are computed along the same lines. The figure shows two versions of the same generation predicate: the naive *sg/2*, and an optimized version *sg_opt/2* obtained by supplementary tabling (i.e. tabling an intermediate join). The latter has better time complexity; observe from the figure that the size of SSG for this program is proportional to table size. For such programs, the space needed for SSG is less than three times the table space in the worst case. In practice the constant factor is close to 1.5 (see next section).

6 Experimental Results

The aim of symbolic support graphs is to make incremental evaluation scale to large applications. To determine the effectiveness of our new data structure and algorithm, we measured their performance on a points-to analyzer for C programs. The analyzer itself is a tabled logic program which encodes Anderson's points-to analysis [2,22]. We measured the performance of the analyzer on programs taken from C benchmarks available with PAF [15] compiler suite and SPEC95 benchmarks. The symbolic support graph based incremental evaluation algorithm was implemented by extending the XSB logic programming system [29] (v2.6). Our incremental points-to analysis system, the benchmarks, and detailed experimental results are available at [24].

We preprocessed the C source code using CIL [14] into Prolog facts representing the primitive assignment statements. Each library function was replaced by a stub representing the data flow between its formal parameters and return value and preprocessed in the same manner. Performance measurements were taken on a PC with 1.4Ghz Pentium M processor with 2GB of physical memory running Linux (Debian) 2.6.7.

We performed All Points-to Analysis (APA), which computes the points-to relation for all program variables. The characteristics of the benchmarks are given in Table 1. In the table, "LOC" refers to the number of lines of source code in the benchmark; "Avg. size" shows the average number of of the points-to tuples per variable. The first four benchmarks in the table are relatively small; to remove noise from the results, we replicated the programs, generating new variable names as appropriate. The remaining three benchmarks are large enough to permit stable measurements without replication. We show the replication factor for each benchmark in the column named "RF".

Space. Table 1 shows the number of supports, and space (in MB) taken by, support graphs [21], partial support graphs with maximum of 2 supports per answer [22], and

Table 1. Comparison of support graph sizes for pointer analysis

			Avg.	Support Graph		Partial Su. Gr.		Symbolic Support Gr.			mem
Benchmark	LOC	RF	size	supports	mem	supports	mem	support	symspt	mem	$\frac{sym}{com}$ %
smail	3850	15	24.5	3,159.4K	92.2	560K	22.8	42.2K	163.0K	15.5	16.8
parser	11391	15	5.8	1,355.7K	44.2	518K	21.8	130.0K	159.2K	17.9	40.5
vpr	17729	15	1.8	213.1K	9.7	172K	8.7	56.2K	51.9K	8.5	86.9
m88ksim	19093	15	6.0	303.5K	11.8	206K	9.2	34.3K	47.9K	7.1	60.5
twmc	24951	1	16.7	5,727.8K	158.5	396K	16.2	90.5K	105.0K	12.6	8.0
nethack	33993	1	35.0	2,074.8K	59.4	269K	11.2	34.9K	60.4K	8.1	13.6
vortex	67110	1	69.8	33,334.5K	912.0	1,714K	65.2	215.3K	361.4K	46.1	5.1

Table 2. Support graph sizes (in MB) for: push-down model checking (a); and synthetic benchmarks from Table 6

Benchmark	Table	SG	SSG
smail	0.8	0.9	0.8
allroots	0.5	0.5	0.4
assembler	47.7	67.6	48.6
compiler	51.1	154.0	53.7
compress	7.0	9.2	7.0
loader	5.9	6.9	6.0

(a)

Programs	Graphs								
	chain			complete			tree		
	2000 nodes			100 nodes			10000 nodes		
	Table	SG	SSG	Table	SG	SSG	Table	SG	SSG
lreach	0.3	0.2	0.2	0.3	0.7	0.4	1.6	0.8	0.9
rreach	47.0	96.0	40.2	0.7	28.5	0.9	5.4	5.7	3.2
sg_opt	1.1	0.4	0.4	1.3	56.6	1.9	5.7	1.8	1.9

(b)

Table 3. Comparison of running times: pointer analysis (a), model checking (b)

Benchmark	from scratch (a)	Incr- Support Graph complete (b)	partial (c)	symbolic (d)	% b/a	% c/a	% d/a
smail	1.45	0.0178	0.1073	0.0433	1.22	7.4	2.98
parser	1.19	0.0025	0.0916	0.0108	0.21	7.7	0.90
vpr	0.37	0.0001	0.0048	0.0005	0.03	1.3	0.14
m88ksim	0.25	0.0005	0.0028	0.0015	0.20	1.1	0.60
twmc	2.49	0.0039	0.2092	0.0125	0.16	8.4	0.50
nethack	0.87	0.0005	0.0487	0.0028	0.06	5.6	0.32
vortex	12.80	0.0040	1.9200	0.0200	0.03	15.0	0.16

(a)

Bench-mark	from scratch	Incremental SG	SSG
smail	.0360	.0005	.0006
allroots	.0050	.0003	.0004
assembler	.0330	.0101	.0131
compiler	.0270	.0113	.0231
compress	.0140	.0023	.0031
loader	.0150	.0017	.0017

(b)

symbolic support graphs for each benchmark. Observe from the table that the symbolic support graph takes the least space among the three. Note that the symbolic support graph may contain non-symbolic supports; while it is possible to make all supports symbolic, we find that it usually increases space requirements by 20%. Finally, the table shows that the symbolic support graph can be considerably smaller than the (non-symbolic) support graph of [21]. Since symbolic supports keep dependencies between calls instead of answers, the reduction in space is proportional to the number of answers per call (the average points-to size).

Table 2(a) shows the sizes of non-symbolic (SG) and symbolic (SSG) support graphs for performing automata-based dead variable analysis of C programs using the push down model checker of [4]. The model checker has few answers per call, consequently we see a reduction in space due to SSGs, but not as much as in the points-to analysis.

Recall from Section 5 the size of the symbolic support graph grows at or near the same rate as the table size for bound-free queries to left-recursive and right-recursive transitive closure and same generation programs (from Figure 6). Table 2(b) shows that not only the growth rates, but the total space requirements of symbolic supports are also close to those of the tables themselves.

Time. The effectiveness of the incremental techniques were evaluated by removing one (source-level) statement from the benchmark programs, and measuring the time and space taken to redo the analysis from scratch and to maintain the points-to

relation incrementally. Deleting one source level assignment statement may delete multiple primitive assignment statements and hence multiple facts.

The results are shown in Table 3. The incremental analysis timings were obtained by repeating the incremental evaluation a number of times to obtain measurable running times. Table 3(a) shows the incremental evaluation times for pointer analysis using the non-symbolic support graph [21] (complete), the partial support graph with at most 2 supports per answer [22] (partial), and the symbolic support graph (symbolic). Observe that the SSG-based algorithm is on average 5 times slower than the complete support graph based one, but is still two orders of magnitude faster than the from-scratch analysis for small changes. For model checking benchmarks SSG-based analysis is on average 1.5 times slower than complete support graph based analysis. The incremental times for synthetic benchmarks are non-measurable and thus omitted here.

7 Related Work

The idea of recording the evaluation process as a graph to guide incremental change propagation has been used in various fields viz. AI, view maintenance, program analysis, model checking, functional programming, and logic programming.

Structures similar to support graph was seen in truth maintenance system [9] (TMS), and later in belief revision systems [3]. With each belief node in TMS a justification set, which represents the reasons for the belief, is kept. This is analogous to the support graph with beliefs as answers and justifications as their supports.

Among the works on materialized view maintenance, we reviewed the relationships between our work and the DRed algorithm [11] in the introduction. The Straight Delete (StDel) [13] algorithm keeps the all the proofs associated with every answer, thereby eliminating the rederivation phase. While this approach may be feasible in constraint databases, its space complexity is worse than the support graph-based algorithm.

The product graph generated during the model checking process is used by incremental model checking algorithm (MCI) of [25]. The complete graph is kept explicitly and the space issues are not addressed. As mentioned in [21] we have adopted MCI's use of counts to efficiently compute truth values of nodes during incremental evaluation.

Incremental attribute grammar evaluation [18] generates an acyclic dependency graph to record the functional dependencies among attribute in the non-circular attribute grammar. In [19] Reps discussed ways to reduce storage space of acyclic dependency graph of attribute grammars. Another instance of an acyclic dependency graph is the *augmented dependency graph* [1] which records dependencies between input and output values in the execution of pure functional programs.

The SSG proposed in this paper grows at or near the rate at which tables grow for many tabled logic programs and queries; in the worst case, however, SSG's size is not bounded by table size. In [22] we proposed an approach to keep a bounded number of supports with each answer, thereby making the support graph size proportional to the table space for arbitrary tabled logic programs.

In [27,26] Binary Decision Diagrams (BDDs) [6] are used to represent transition relation and reachable states of a state transition graph in the context of logic synthesis and formal verification of digital circuits. To incrementally maintain reachable states

in response to changes in the transition relation, a spanning graph is generated during reachability analysis as the evidence for all the reachable states. BDDs are used to represent the edge relation of the spanning graph, thereby making it space efficient. We attempted to keep the support graph using BDDs. However, an inordinate amount of time was taken to build the support graph BDDs: there appears to be no way to use the efficient set-at-a-time operations of the BDDs to construct the support graph when the query evaluation is done tuple at a time.

8 Conclusion

We presented a space-efficient data structure and incremental algorithms for maintaining tables in the presence of deletion of facts. The techniques can be readily extended to handle deletion of rules by keeping rule information in supports as done in [21]. The symbolic support graph maintains dependencies between certain calls, but in such a way that the dependencies between answers can be readily computed whenever needed. The ease of computation is ensured by keeping the symbolic supports in a "join-free" form, keeping only the last literal of a support as a call. This can be easily extended to keeping as calls the right-most literal in a clause that is followed by simple computations (such as comparison operations). This extension of the notion of symbolic supports will permit us to represent programs with aggregation operations using symbolic supports, thereby enabling incremental evaluation of dynamic programming problems.

In this paper, we considered only definite logic programs, where all predicates are either tabled or defined by facts. We can extend this to programs containing a mixture of tabled and non-tabled predicates along the same lines as in [21]: accumulating support information from non-tabled predicates and storing them with answers. In [23] we give a support-graph-based bottom-up algorithm that interleaves insertion and deletion that permits efficient handling of incremental updates, and programs with stratified negation. The idea of symbolic support graph can be used in the algorithm of [23].

References

1. U. A. Acar, G. E. Blelloch, and R. Harper. Adaptive functional programming. In *POPL*, pages 247–259. ACM Press, 2002.
2. L. O. Anderson. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, Unversity of Copenhagen, 1994.
3. K. Apt and J. M. Pugin. Maintenance of stratified databases viewed as a belief revision system. In *Principles of Database Systems*, pages 136–145. ACM Press, 1987.
4. S. Basu, K. N. Kumar, L. R. Pokorny, and C. R. Ramakrishnan. Resource-constrained model checking of recursive programs. In *TACAS*, volume 2280 of *LNCS*, pages 236–250, 2002.
5. R. Bol and L. Degerstadt. Tabulated resolution for well-founded semantics. In *ILPS*, 1993.
6. R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.
7. W. Chen and D. S. Warren. Tabled evaluation with delaying for general logic programs. *JACM*, 43(1):20–74, 1996.
8. S. Dawson, C. R. Ramakrishnan, and D. S. Warren. Practical program analysis using general purpose logic programming systems — a case study. In *ACM PLDI*, pages 117–126, 1996.

9. J. Doyle. A truth maintenance system. *Artificial Intelligence*, 12:231–272, 1979.
10. H. Guo and G. Gupta. A simple scheme for implementing tabled logic programming systems based on dynamic reordering of alternatives. In *ICLP*, pages 181–196. Springer, 2001.
11. A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *SIGMOD*, pages 157–166, 1993.
12. A. Gupta and I.S. Mumick. Maintenance of materialized views: Problems, techniques, and appfications. *IEEE Data Engineering Bulletin*, 18(2):3–18, 1995.
13. J. Lu, G. Moerkotte, J. Schue, and V. S. Subrahmanian. Efficient maintenance of materialized mediated views. In *ACM SIGMOD*, pages 340–351, 1995.
14. G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Compiler Construction*, pages 213–228. Springer-Verlag, 2002.
15. PAF. Prolangs analysis framework. Available at `http://www.prolangs.rutgers.edu/public.html`.
16. L. L. Pollock and M. L. Soffa. An incremental version of iterative data flow analysis. *IEEE Trans. Softw. Eng.*, 15(12):1537–1549, 1989.
17. C. R. Ramakrishnan et al. XMC: A logic-programming-based verification toolset. In *CAV*, number 1855 in LNCS, pages 576–580, 2000.
18. T. Reps, T. Teitelbaum, and A. Demers. Incremental context-dependent analysis for language-based editors. *TOPLAS*, 5(3):449–477, 1983.
19. T. W. Reps. *Generating language-based environments*. MIT Press, 1984.
20. R. Rocha, F. Silva, and V. Santos Costa. YapTab: A Tabling Engine Designed to Support Parallelism. In *Workshop on Tabling in Parsing and Deduction*, 2000.
21. D. Saha and C. R. Ramakrishnan. Incremental evaluation of tabled logic programs. In *ICLP*, volume 2916 of *LNCS*, pages 389–406, 2003.
22. D. Saha and C. R. Ramakrishnan. Incremental and demand-driven points-to analysis using logic programming. In *PPDP*, pages 117–128. ACM Press, 2005.
23. D. Saha and C. R. Ramakrishnan. A local algorithm for efficient incremental evaluation of tabled logic programs, 2005. Available at `http://www.lmc.cs.sunysb.edu/~dsaha/local`.
24. D. Saha and C. R. Ramakrishnan. Symbolic support graph: A space efficient data structure for incremental tabled evaluation, 2005. Available at `http://www.lmc.cs.sunysb.edu/~dsaha/symspt`.
25. O. V. Sokolsky and S. A. Smolka. Incremental model checking in the modal mu-calculus. In *CAV*, volume 818 of *LNCS*, pages 351–363, 1994.
26. G. Swamy. *Incremental Methods for Formal Verification and Logic Synthesis*. PhD thesis, University of California at Berkeley, 1996.
27. G. Swamy, R. K. Brayton, and V. Singhal. Incremental methods for FSM traversal. In *Intl. Conference on Computer Design (ICCD)*. IEEE Computer Society, 1995.
28. H. Tamaki and T. Sato. OLDT resolution with tabulation. In *ICLP*, pages 84–98, 1986.
29. XSB. The XSB logic programming system. Available at `http://xsb.sourceforge.net`.
30. J. Yur, B. G. Ryder, and W. Landi. An incremental flow- and context-sensitive pointer aliasing analysis. In *ICSE*, pages 442–451, 1999.
31. N. Zhou, Y. Shen, L. Yuan, and J. You. Implementation of a linear tabling mechanism. *Journal of Functional and Logic Programming*, 2001(10), October 2001.

Dynamic Mixed-Strategy Evaluation of Tabled Logic Programs

Ricardo Rocha[1], Fernando Silva[1], and Vítor Santos Costa[2]

[1] DCC-FC & LIACC, University of Porto, Portugal,
{ricroc, fds}@ncc.up.pt
[2] COPPE Systems & LIACC, Federal University of Rio de Janeiro, Brazil
vitor@cos.ufrj.br

Abstract. Tabling is an implementation technique that improves the declarativeness and expressiveness of Prolog by reusing answers to subgoals. During tabled execution, several decisions have to be made. These are determined by the scheduling strategy. Whereas a strategy can achieve very good performance for certain applications, for others it might add overheads and even lead to unacceptable inefficiency. The ability of using multiple strategies within the same evaluation can be a means of achieving the best possible performance. In this work, we present how the YapTab system was designed to support dynamic mixed-strategy evaluation of the two most successful tabling scheduling strategies: batched scheduling and local scheduling.

1 Introduction

The past years have seen wide efforts at increasing Prolog's declarativeness, expressiveness and performance. One proposal that has gained popularity is the use of *tabling* (also known as *tabulation* or *memoing*). Tabling based models are able to reduce the search space, avoid looping, and have better termination properties than traditional Prolog based models. Several alternative tabling models have been proposed and implemented [1,2,3,4,5]. The most well-known tabling Prolog system is XSB Prolog [6], which proved the viability of tabling technology in application areas such as Natural Language Processing, Knowledge Based Systems, Model Checking, and Program Analysis.

The basic idea behind tabling is straightforward: programs are evaluated by storing answers for current subgoals in an appropriate data space, called the *table space*. Whenever a repeated call is found, the subgoal's answers are recalled from the table instead of being re-evaluated against the program clauses.

During tabled execution, there are several points where we may have to choose between continuing forward execution, backtracking, consuming answers from the table, or completing subgoals. The decision on which operation to perform is crucial to system performance and is determined by the *scheduling strategy*. Different strategies may have a significant impact on performance, and may lead to a different ordering of solutions to the query goal. Arguably, the two

M. Gabbrielli and G. Gupta (Eds.): ICLP 2005, LNCS 3668, pp. 250–264, 2005.

most successful tabling scheduling strategies are batched scheduling and local scheduling [7].

Batched scheduling favors forward execution first, backtracking next, and consuming answers or completion last. It thus tries to delay the need to move around the search tree by *batching* the return of answers. When new answers are found for a particular tabled subgoal, they are added to the table space and the evaluation continues. On the other hand, *local scheduling* tries to complete subgoals as soon as possible. When new answers are found, they are added to the table space and the evaluation fails. Answers are only returned when all program clauses for the subgoal in hand were resolved.

Empirical work from Freire *et al.* [7,8] showed that, regarding the requirements of an application, the choice of the scheduling strategy can affect the memory usage, execution time and disk access patterns. Freire argues [9] that there is no single best scheduling strategy, and whereas a strategy can achieve very good performance for certain applications, for others it might add overheads and even lead to unacceptable inefficiency. Freire and Warren [10] suggested that using multiple strategies within the same evaluation would be most useful. However, to the best of our knowledge, no such implementation has yet been done.

Our main contribution is a novel approach to supporting dynamic mixed-strategy evaluation of tabled logic programs. We have implemented this approach in the YapTab system, as an elegant extension of the original design [2]. YapTab supports the dynamic intermixing of batched and local scheduling at the subgoal level, that is, it allows one to modify at runtime the strategy to be used to resolve the subsequent subgoal calls of a tabled predicate. We show that YapTab's hybrid approach does indeed return very substantial performance gains. Results were impressive both on artificial applications, and on a complex, real-life, application.

The remainder of the paper is organized as follows. First, we briefly introduce the basic tabling definitions and present the differences between batched and local scheduling. Next, we describe the issues involved in providing engine support for integrating both scheduling strategies at the subgoal level. We then discuss some experimental results and outline some conclusions.

2 Basic Tabling Definitions

Tabling is about storing intermediate answers for subgoals so that they can be reused when a repeated call appears. Whenever a tabled subgoal S is first called, a new entry is allocated in the table space. This entry will collect all the answers found for S. Repeated calls to *variants* of S are resolved by consuming the answers already in the table. Meanwhile, as new answers are found, they are stored into the table and returned to all variant subgoals. Within this model, the nodes in the search space are classified as either: *generator nodes*, corresponding to first calls to tabled subgoals; *consumer nodes*, corresponding to variant calls to tabled subgoals; or *interior nodes*, corresponding to non-tabled subgoals. Tabling based models have four main types of operations for definite programs:

1. The *tabled subgoal call* operation is a call to a tabled subgoal. It checks if the subgoal is in the table. If so, it allocates a consumer node and starts consuming the available answers. If not, it adds a new entry to the table, and allocates a new generator node.
2. The *new answer* operation verifies whether a newly found answer is already in the table, and if not, inserts the answer. Otherwise, the operation fails.
3. The *answer resolution* operation is executed every time the computation reaches a consumer node. It verifies whether extra answers are available for the particular consumer node and, if so, consumes the next one. If no answers are available, it *suspends* the current computation, either by freezing the whole stacks [1], or by copying the execution stacks to separate storage [3], and schedules a possible resolution to continue the execution.
4. The *completion* operation determines whether a tabled subgoal is *completely evaluated*. A subgoal is said to be completely evaluated when all its possible resolutions have been performed, that is, when no more answers can be generated. It executes when we backtrack to a generator node and all of its clauses have been tried. If the subgoal has been completely evaluated, the operation closes the goal's table entry and reclaims stack space. Otherwise, control moves to a consumer with unconsumed answers.

Completion is needed in order to recover space and to support negation. We are most interested on space recovery in this work. Arguably, in this case, we could delay completion until the very end of the execution. Unfortunately, doing so would also mean that we could only recover space for consumers (suspended subgoals) at the very end of the execution. Instead we shall try to achieve *incremental completion* [11] to detect whether a generator node has been fully exploited, and if so to recover space for all its consumers.

Completion is hard because a number of generators may be mutually dependent, thus forming a *Strongly Connected Component* (or *SCC*). Clearly, we can only complete SCCs together. We will usually represent an SCC through the oldest generator. More precisely, the youngest generator node which does not depend on older generators is called the *leader node*. A leader node is also the oldest node for its SCC, and defines the current completion point.

When we call a variant subgoal that is already completed, we can avoid consumer node allocation and perform instead what is called a *completed table optimization* [1]. This optimization allocates a node, similar to an interior node, that will consume the set of found answers executing compiled code directly from the table data structures associated with the completed subgoal [12].

3 Scheduling Strategies

It should be clear that at several points we can choose between continuing forward execution, backtracking to interior nodes, returning answers to consumer nodes, or performing completion. The actual sequence of operations depends on the scheduling strategy. We next discuss in some more detail the batched and local scheduling strategies.

3.1 Batched Scheduling

The batched strategy schedules the program clauses in a depth-first manner as does the WAM. In this strategy, new answers are added to the table space but evaluation continues until it resolves all program clauses for the subgoal in hand. Only when all clauses have been resolved, the newly found answers will be returned to consumer nodes. Hence, when backtracking we may encounter three situations: **(i)** if backtracking to a generator or interior node, we take the next available alternative; **(ii)** if backtracking to a consumer node, we take the next unconsumed answer; **(iii)** if there are no available alternatives or no unconsumed answers, we simply backtrack to the previous node on the current branch. Note however that, if the node without alternatives is a leader generator node, then we must *check for completion*.

In order to perform completion, we must ensure that all answers have been returned to all consumers in the SCC. The process of resuming a consumer node, consuming the available set of answers, suspending and then resuming another consumer node can be seen as an iterative process which repeats until a fixpoint is reached. This fixpoint is reached when the SCC is completely evaluated.

At the engine level, the *fixpoint check procedure* is controlled by the leader of the SCC. Initially, it searches for the younger consumer with unresolved answers, and as long as there new answers, it will consume them. After consuming the available set of answers, the consumer suspends and fails into the next consumer with unresolved answers. This process repeats until it reaches the last consumer, in which case it fails into the leader node in order to allow the re-execution of the fixpoint check procedure. When a fixpoint is reached, all subgoals in the SCC are marked completed and the stack segments belonging to them are released.

3.2 Local Scheduling

Local scheduling is an alternative tabling scheduling strategy that tries to complete subgoals as soon as possible. In this strategy, evaluation is done one SCC at a time. The key idea is that whenever new answers are found, they are added to the table space as usual but execution fails. Thus, execution explores the whole SCC before returning answers outside the SCC. Hence, answers are only returned when all program clauses for the subgoal in hand were resolved.

Figure 1 shows a small example that clarifies the differences between batched and local evaluation. The top sub-figure illustrates the program code and query goal used in the example. Declaration '`:- table t/1.`' indicates that calls to predicate `t/1` should be tabled. The two sub-figures below depict the evaluation sequence for each strategy and how the table space is filled in. In both cases, the leftmost tree represents the evaluation of the query goal `q(X,Y)`. Nodes are numbered according to the evaluation sequence. Generators are depicted by white oval boxes, and consumers by gray oval boxes. For simplicity of presentation, the computation tree for `t(X)` is represented independently at the right.

Both cases begin by resolving the query goal against the unique clause for predicate `q/2`, thus calling the tabled subgoal `t(X)`. As this is the first call to `t(X)`, we create a generator node and insert a new entry in the table space.

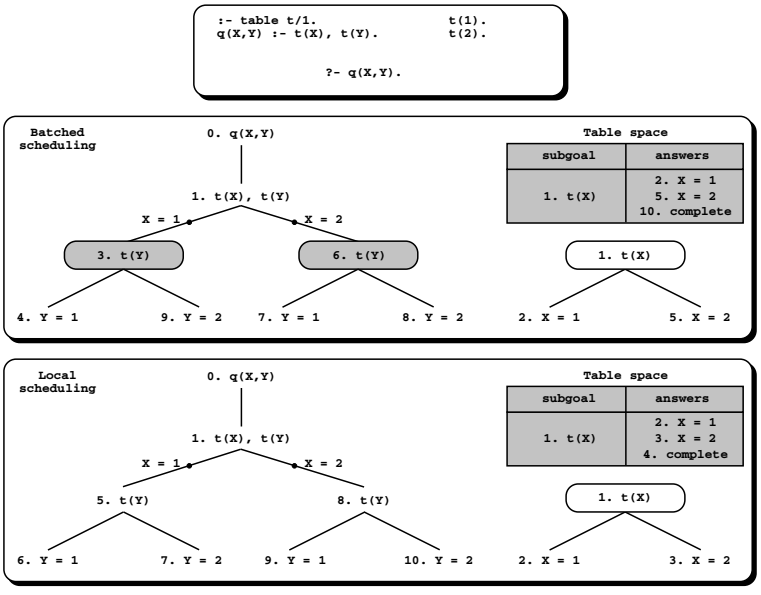


Fig. 1. Batched scheduling versus local scheduling

The first clause for `t(X)` succeeds immediately, obtaining a first answer for `t(X)` that is stored in the table (step 2). The example is most interesting in how both strategies handle the continuation of `t(X)`.

Evaluation with batched scheduling proceeds as in standard Prolog with a call to the second goal, `t(Y)`, therefore creating consumer node 3. Node 3 is a variant call to `t(X)`, so instead of resolving the call against the program clauses, we consume answers from the table space. As we already have one answer stored in the table for this call (`X=1`), we continue by consuming it, which leads to a first solution for the query goal (`X=1,Y=1`). When returning to node 3, we must suspend the consumer node because we cannot collect more answers from the table. We then backtrack to node 1 to try the second clause for `t(X)`, and a new answer is found (`X=2`). The continuation again calls `t(Y)`, and a new consumer is created (node 6). Eventually, two new solutions are found for the query goal (steps 7 and 8). We have consumed all answers from the table. Hence, node 6 suspends and the computation backtracks again to node 1. Node 1 is the leader of the SCC, hence at that point, we can check for completion. However, the generator cannot complete because consumer 3 still has unconsumed answers. The computation thus resumes at node 3 and a new solution for the query goal is found (step 9). No new answers are inserted into the table, and all consumers have exhausted the entries in the table. Thus, when we return to the generator node 1, we can finally complete the tabled subgoal call `t(X)` (step 10).

Evaluation with local scheduling differs from early on, namely, after the first answer was found (step 2). To exhaust the current SCC, we should fail, backtrack to node 1, and execute the second clause for `t(X)`. We thus find a second answer

for it (step 3). Then, we fail again to node 1, and the tabled subgoal call `t(X)` can be completed immediately, as we have found no consumers yet (step 4). From now on, answers are consumed by executing compiled code, as we discussed before. The variant calls to `t(X)` at steps 5 and 8 are thus resolved in this way, and no consumer goals are ever needed.

Choosing the best strategy is hard. The main difference between the two is that in batched scheduling, variable bindings are immediately propagated to the calling environment when an answer is found. For some situations, this behavior may result in creating complex dependencies between consumers. On the other hand, the clear advantage of local scheduling shown in the example does not always hold. Since local scheduling delays answers, it does not benefit from variable propagation, and instead, when explicitly returning the delayed answers, it incurs an extra overhead for copying them out of the table. Freire *et al.* [7] showed that, on average, local scheduling is about 15% slower than batched scheduling in the SLG-WAM [1]. Similar results were also obtained in YapTab [2].

3.3 Defining the Scheduling Strategy

We provide two built-in predicates for defining and controlling the *tabling mode* to be used to evaluate a tabled computation. We extend the standard predicate `yap_flag/2` to define the standard scheduling strategy for the whole computation. Alternatively, we can use the `tabling_mode/2` predicate to define the scheduling strategy of a particular tabled predicate. We next discuss how these predicates can be used to dynamically control the evaluation. Consider, for example, two tabled predicates, `t/1` and `t/2`, and the following query goals:

```
:- t(1).
:- yap_flag(tabling_mode,local), t(2,2).
:- t(3), yap_flag(tabling_mode,default), t(3,3).
:- tabling_mode(t/1,local), t(X),t(X,Y), tabling_mode(t/1,batched), t(Y).
```

In the first example query, `t(1)` evaluates using batched scheduling. This happens because, by default in YapTab, when a predicate is declared as tabled, its initial tabling mode is batched. In the second query, `t(2,2)` evaluates using local scheduling as the call to `yap_flag(tabling_mode,local)` changes the tabling mode of the following computations to local. In the third query, `t(3)` evaluates using local scheduling because the tabling mode for the computation is still local (as a result of the previous `yap_flag/2` declaration in the second query), and `t(3,3)` evaluates using batched. Note that the actual execution tree will have nodes for both strategies: `t(3,3)` might itself call `t(3)`. The call to `yap_flag(tabling_mode,default)` defines that, in what follows, we should use the default strategy of each predicate and the initial tabling mode of `t/2` is batched. Finally, in the fourth query, `t(X)` evaluates using local scheduling and `t(X,Y)` and `t(Y)` evaluates using batched scheduling. The call to `tabling_mode(t/1,local)` initially changes the tabling mode of predicate `t/1` to local and then `tabling_mode(t/1,batched)` changes it back to batched.

4 Implementation

The YapTab design mostly follows the seminal SLG-WAM design [1]: it introduces a new data area to the WAM, the *table space*; a new set of registers, the *freeze registers*; an extension of the standard trail, the *forward trail*; and four new operations: *tabled subgoal call*, *new answer*, *answer resolution*, and *completion*. Tables are implemented using tries as proposed in [12]. The differences between the two designs reside in the data structures and algorithms used to control the process of leader detection and the scheduling of unconsumed answers.

Namely, the original SLG-WAM considers that such control should be done at the level of the data structures corresponding to first calls to tabled subgoals, and does so by associating *completion frames* to generator nodes. The SLG-WAM relies on a *completion stack* of generators to detect completion points. On the other hand, YapTab innovates by considering that the control of leader detection and scheduling of unconsumed answers should be performed through the data structures corresponding to variant calls to tabled subgoals, and it associates a new data structure, the *dependency frame*, to consumer nodes. Dependency frames store information about the last consumed answer; and information to efficiently check for completion points, and to efficiently move across the consumer nodes with unconsumed answers.

In YapTab, applying batched or local scheduling to an evaluation mainly depends on the way generator nodes are handled. At the engine level, this includes minor changes to the operations tabled subgoal call, new answer and completion. All the other tabling extensions are common across both strategies. We claim that, this makes YapTab highly suitable to efficiently support a dynamic mixed-strategy evaluation.

4.1 Tabled Nodes

By combining the two built-in predicates `yap_flag/2` and `tabling_mode/2` we can dynamically define the scheduling strategy to be used to evaluate each tabled subgoal. Thus, when a tabled subgoal is first called, the tabled subgoal call operation starts by consulting the current tabling mode of the computation/predicate in order to decide the strategy to be used by the corresponding generator node.

In our implementation, generator nodes are WAM choice points extended with two extra fields: `CP_DepFr` is a pointer to the corresponding dependency frame (its use is detailed next) and `CP_SgFr` is a pointer to the associated subgoal frame where answers should be stored. Consumer nodes are WAM choice points extended with the `CP_DepFr` field only. Figure 2 details generator and consumer choice points and their relationship with the table and dependency spaces.

The left sub-figure shows a choice point stack with generator nodes for both strategies and with a consumer node. Remember that the key difference between the two strategies is that local scheduling prevents answers from being returned early by backtracking until getting all answers for the leader generator. At the point all answers have been exhausted, the leader must export them to its environment. To do so, it must act like a consumer: consuming answers

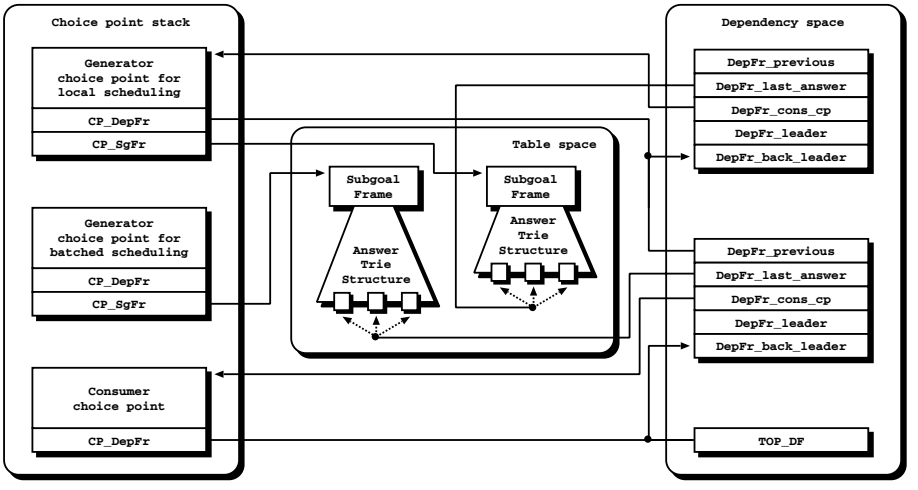


Fig. 2. Generator and consumer choice points in YapTab

and propagating them to caller one by one. In YapTab, this was implemented by having the `CP_DepFr` field in the generators at the same position as for the consumers. This simple extension allows generators being evaluated using local scheduling to easily become consumers. Note that YapTab relies on this field: it is used by the operations new answer and completion as a way to distinguish if a generator is being evaluated using local scheduling (cases where `CP_DepFr` is not `NULL`) or using batched scheduling (cases where `CP_DepFr` is `NULL`).

The right sub-figure shows the dependency frames, the key data structure to synchronize the flow of a tabled evaluation. The `TOP_DF` variable always points to the youngest dependency frame on stack. Frames form a linked list through the `DepFr_previous` field. The `DepFr_last_answer` field points to the last consumed answer in the table space. The `DepFr_cons_cp` field points back to the corresponding consumer choice point. The `DepFr_leader` and the `DepFr_back_leader` fields respectively point to the leader node at creation time and to the leader node where we performed the last unsuccessful completion operation. They are critical in the SCC fixpoint check procedure, that we discuss next.

4.2 Leader Nodes

How does completion change in a mixed environment? Completion takes place when we backtrack to a generator node that **(i)** has exhausted all its alternatives and that **(ii)** is a leader node (remember that the youngest generator which does not depend on older generators is called a leader node). The key idea in our original algorithms is that each dependency frame holds a pointer to the resulting leader node of the SCC that includes the correspondent consumer node. Using the leader node pointer from the dependency frames, a generator can quickly determine whether it is a leader node. We thus rely on the notion of *leader node*:

a generator $\mathcal{L}$ is a leader node when either **(a)** $\mathcal{L}$ is the youngest tabled node, or **(b)** the youngest consumer says that $\mathcal{L}$ is the leader.

Next we show that our algorithm for detecting leader nodes works well in a mixed environment. The algorithm requires computing leader node information whenever creating a new consumer node $\mathcal{C}$. First, we hypothesize that the leader node is $\mathcal{C}$'s generator, say $\mathcal{G}$. Next, for all consumer nodes older than $\mathcal{C}$ and younger than $\mathcal{G}$, we check whether they depend on an older generator node. Consider that there is at least one such node and that the oldest of these nodes is $\mathcal{G}'$. If so then $\mathcal{G}'$ is the leader node. Otherwise, our hypothesis was correct and the leader node is indeed $\mathcal{G}$. Leader node information is implemented as a pointer to the choice point of the newly computed leader node. Figure 3 shows the procedure that computes the leader node information for a new consumer.

```
compute_leader(consumer node CN) {
  leader_cp = generator_for(CN)    // the generator is the default leader
  df = TOP_DF
  while (DepFr_cons_cp(df) is younger than leader_cp ) {
    if (DepFr_leader(df) is older than leader_cp) {   // older dependency
      leader_cp = DepFr_leader(df)
      break
    }
    df = DepFr_previous(df)
  }
  return leader_cp
}
```

Fig. 3. Pseudo-code for `compute_leader()`

The procedure traverses the dependency frames for the consumer nodes between the new consumer and its generator in order to check for older dependencies. As an optimization it only searches until it finds the first dependency frame holding an older reference (the `DepFr_leader` field). The nature of the procedure ensures that the remaining dependency frames cannot hold older references.

Note that for local scheduling, when we store a generator node $\mathcal{G}$ we also allocate a dependency frame. However, we can avoid calling `compute_leader()` because $\mathcal{G}$ itself is the leader node.

4.3 Completion

Next, we show our implementation of completion. Completion is forced as follows. When a generator choice point tries the last program clause, its `CP_AP` (failure continuation program counter) field is updated to the `completion` instruction. From then on, every time we backtrack to the choice point the operation is executed. Figure 4 shows the pseudo-code for completion in YapTab.

First, the procedure checks out if the generator is the current leader node. If a leader, it checks whether all younger consumer nodes have consumed all their answers. To do so, it traverses the chain of dependency frames looking for a frame that has not yet consumed all the generated answers. If there is such a frame, the computation should be resumed to the corresponding consumer node. Otherwise, it can perform completion. This includes **(i)** marking as complete all

```
completion(generator node GN) {
  if (GN is the current leader node) {
    df = TOP_DF
    while (DepFr_cons_cp(df) is younger than GN) {
      if (unconsumed_answers(DepFr_last_answer(df))) {
        DepFr_back_leader(df) = GN        // mark the leader to return to
        move_to(DepFr_cons_cp(df))
      }
      df = DepFr_previous(df)
    }
    perform_completion()
    if (CP_DepFr(GN) != NULL)                          // local scheduling
      completed_table_optimization()
  }
  if (CP_DepFr(GN) != NULL) {                          // local scheduling
    CP_AP(GN) = answer_resolution
    load_first_available_answer_and_proceed()
  } else backtrack()                                  // batched scheduling
}
```

Fig. 4. Pseudo-code for `completion()`

the subgoals in the SCC, and **(ii)** deallocating all younger dependency frames. At the end, if the generator was evaluated using local scheduling, we need to consume the set of answers that have been found. As the subgoal is already completed, we can execute compiled code directly from the trie data structure associated with the completed subgoal.

On the other hand, if the current node is not the leader, the procedure simply backtracks to the previous node, if in batched mode, or starts acting like a consumer node and consumes the first available answer, if in local mode.

4.4 Answer Resolution

Next, we show that our implementation of answer resolution is independent of strategy. The answer resolution operation executes every time the computation fails back to a consumer. In our implementation, a consumer choice point always points to the `answer_resolution` instruction in its `CP_AP` field. Figure 5 shows the pseudo-code for this instruction in YapTab.

Initially, the procedure checks the table space for unconsumed answers. If there are new answers, it loads the next available answer and proceeds. Otherwise, it schedules for a backtracking node. If this is the first time that backtracking from that consumer node takes place, then it is performed as usual to the previous node. This is the case when the `DepFr_back_leader` field is `NULL`. Otherwise, we know that the computation has been resumed from an older leader node $\mathcal{L}$ during an unsuccessful completion operation. Therefore, backtracking must be done to the next consumer node that has unconsumed answers and that is younger than $\mathcal{L}$. We do this by restoring bindings and stack pointers. If no such consumer node can be found, backtracking must be done to node $\mathcal{L}$.

The iterative process of resuming a consumer node, consuming the available set of answers, suspending and then resuming another consumer until a fixpoint is reached, is completely independent of the scheduling strategy being used.

```
answer_resolution(consumer node CN) {
  df = CP_DepFr(CN)                              // dependency frame for CN
  if (unconsumed_answers(DepFr_last_answer(df)))
    load_next_available_answer_and_proceed()
  back_cp = DepFr_back_leader(df)
  if (back_cp == NULL)                                  // first time here
    backtrack()
  df = DepFr_previous(df)
  while (DepFr_cons_cp(df) is younger than back_cp) {
    if (unconsumed_answers(DepFr_last_answer(df))) {
      DepFr_back_leader(df) = back_cp     // mark the leader to return to
      move_to(DepFr_cons_cp(df))
    }
    df = DepFr_previous(df)
  }
  move_to(back_cp)                               // move to last leader node
}
```

Fig. 5. Pseudo-code for `answer_resolution()`

5 Experimental Results

To put the performance results in perspective, we first used a set of common tabled benchmark programs to evaluate the overheads of supporting mixed-strategy evaluation for programs that only require a single-strategy approach. The environment for our experiments was an AMD Athlon XP 2800+ processor with 1 GByte of main memory and running the Linux kernel 2.6.8. YapTab is based on the current development version of Yap, version 4.5.7.

Table 1 shows the running times, in milliseconds, for YapTab supporting a single scheduling strategy (YapTab Single) and supporting the mixed approach (YapTab Mixed). In parentheses, it shows the overhead over YapTab Single. The execution times correspond to the average times obtained in a set of 3 runs. The results indicate that YapTab Mixed introduces insignificant overheads over YapTab Single, both for batched and local scheduling. These overheads are very small. They mainly result from operations that test if a generator is being evaluated using batched or local scheduling.

In the literature, we can find several examples showing that batched scheduling performs better than local scheduling for certain applications and that local scheduling performs better for others [7,10]. However, usually, these examples

Table 1. Overheads of supporting mixed-strategy evaluation

	Batched Scheduling		Local Scheduling	
Program	**YapTab Single**	**YapTab Mixed**	**YapTab Single**	**YapTab Mixed**
mc-iproto	2495	2519 (1.009)	2668	2689 (1.007)
mc-leader	8452	8467 (1.001)	8385	8403 (1.002)
mc-sieve	21568	21325 (0.988)	21797	21217 (0.973)
lgrid	850	870 (1.023)	1012	1031 (1.018)
rgrid	1250	1332 (1.065)	1075	1141 (1.061)
samegen	20	20 (1.000)	21	21 (1.000)
Average		(1.014)		(1.010)

are independent and not part of the same application. To further motivate for the applicability of our mixed-strategy approach, we next present two different examples where we take advantage of YapTab's flexibility.

Our first example is an application in the context of Inductive Logic Programming (ILP) [13]. The fundamental goal of an ILP system is to find a consistent and complete theory (logic program), from a set of examples and prior knowledge, the *background knowledge*, that *explain* all given positive examples, while being consistent with the given negative examples. Since it is not usually obvious which set of hypotheses should be picked as the theory, an ILP system generates many candidate hypotheses (clauses) which have many similarities among them. Usually, these similarities tend to correspond to common prefixes (subgoals) among the hypotheses. Consider, for example, that the system generates an hypothesis '`theory(X):- b1(X),b2(X,Y).`' which obtains a *good coverage quality*, that is, the number of positive examples covered by it is high and the number of negative example is low. Then, it is quite possible that the system will use it to generate more specific clauses like '`theory(X):- b1(X),b2(X,Y),b3(Y).`'.

Computing the coverage of an hypothesis requires, in general, running all positives and negatives examples against the clause. For example, to evaluate if the positive example `theory(p1)` is covered by '`theory(X):- b1(X),b2(X,Y).`', the system executes the goal '`b1(p1),b2(p1,Y)`'. If the same example is then evaluated against the other clause, goal '`b1(p1),b2(p1,Y),b3(Y)`', part of the computation will be repeated. For datasets with a large number of examples, we can arbitrarily do a lot of recomputation. Tabling technology is thus an excellent candidate to significantly reduce the execution time for these kind of problems. Moreover, as we will see, we can benefit from YapTab's mixed-strategy approach to further improve performance.

Assume now that we declared `b2/2` as tabled and that '`b1(p1),b2(p1,Y)`' succeeds. Thus, we can mark `theory(p1)` as covered by the corresponding hypothesis, and we can reclaim space by pruning the search space for the goal in hand. Note that the ILP system is only interested in evaluating the coverage of the examples, and not in finding answers for the subgoals. On the other hand, from the tabling point of view, `b2(p1,Y)` is not completed because it may succeed with other answers for `Y`. A question then arises: should we use batched or local scheduling to table these predicates?

At first, local scheduling seems more attractive because it avoids the pruning problem mentioned above. When the ILP system prunes the search space, the tables are already completed. On the other hand, if the cost of fully generating the complete set of answers is very expensive, then the ILP system may not always benefit from it. Consider, for example a predicate defined by several facts and then by a recursive clause (quite common in some ILP datasets). It can happen that, after completing a subgoal, the subgoal is not called again or when called it succeeds just by using the known facts, thus, turning it useless to compute beforehand the full set of answers.

Note also that, when an example is not covered, all the subgoals in the clause are completed. For example, if in '`b1(p1),b2(p1,Y),b3(Y)`', the subgoal `b3(Y)`

Table 2. Intermixing batched and local scheduling at the predicate level

Predicates	Running Time
Without tabling	> 1 day
All batched (11 predicates)	283779
All local (11 predicates)	147937
Some batched (7 predicates), others local (4 predicates)	127388

never succeeds then, by backtracking, `b2(p1,Y)` will be completely evaluated. For such cases, batched scheduling is better because variable bindings are automatically propagated. We can also benefit from batched when an example is covered in clauses of the form '`b2(p1,Y),b2(p1,Z)`', with the tabled subgoal appearing repeated. Finally, for subgoals that never succeed or that succeed with a yes answer (all arguments ground), batched and local obtain similar results.

We experimented with using both strategies individually and together. Table 2 shows, the running times, in milliseconds, for the April ILP system [14] running a well-known ILP dataset, the *mutagenesis* dataset.

We used four different approaches to evaluate the predicates in the background knowledge: **(i)** without tabling; **(ii)** all predicates being evaluated using batched scheduling; **(iii)** all using local scheduling; and **(iv)** some using batched and others using local (for this approach we show the running time for the best result obtained). Note that the running times include the time to run the whole ILP system and not just the time for computing the coverage of the hypotheses. The results show that tabled evaluation can significantly reduce the execution time for these kind of problems. Moreover, they show that, by using mixed-strategy evaluation, we can further speedup the execution. Better performance is still possible if we use YapTab's flexibility to intermix batched and local scheduling at the subgoal level. However, from the programmer point of view, it is very difficult to define the subgoals to table using one or another strategy. Further work is still needed to study how to use this flexibility to, in runtime, automatically adjust the system to the best approach.

We next show a different application where we take full advantage of the dynamic mixed-strategy of YapTab by intermixing batched and local scheduling at the subgoal level. Consider a 30x30 grid, represented by a number of `edge/2` facts, and the following program code:

```
:- table path/2.
path(X,Y) :- path(X,Z), edge(Z,Y).
path(X,Y) :- edge(X,Y).

reachable(X,Y) :- path(F,X), path(F,Y), !.

go_batched :- tabling_mode(path/2,batched).
go_local :- tabling_mode(path/2,local).
```

Now consider the query goal '`path(X,Y),reachable(X,Y)`' that computes the paths in the grid whose extremities are reachable from, at least, another node. We solve this query using three alternative approaches for tabling the

Table 3. Intermixing batched and local scheduling at the subgoal level

	Query Goal	Running Time
(i)	:- go_batched, path(X,Y), reachable(X,Y), fail.	141962
(ii)	:- go_local, path(X,Y), reachable(X,Y), fail.	60471
(iii)	:- go_local, path(X,Y), go_batched, reachable(X,Y), fail.	19770

`path/2` predicate: **(i)** only batched scheduling; **(ii)** only local scheduling; and **(iii)** local scheduling for the first query subgoal and batched scheduling for the second. Table 3 shows the running times, in milliseconds, for finding all the solutions for the query above using the three approaches. The execution times correspond to the average times obtained in a set of 3 runs.

The results show that by using local scheduling for computing the first subgoal and batched for the second we are able to significantly reduce the execution time and achieve the best performance. This happens because, by using local scheduling to compute the complete set of answers for `path(X,Y)`, we avoid complex dependencies when executing predicate `reachable(X,Y)` with batched. Note that when we call `path(F,X)` in predicate `reachable/2`, `F` is a free variable. Then, when we use the first clause of `path/2` to solve `path(F,X)`, we get a call to `path(F,Z)` (with both variables free), which is a variant call of the initial query subgoal `path(X,Y)`, and thus we must allocate a consumer node.

On the other hand, if we already have the set of answers for the first query subgoal, it is best if we use batched to solve the calls to the `reachable/2` predicate. If we use local scheduling, we will compute all the answers for each particular call to `path(F,X)`, with `X` ground, and this may lead to unnecessary computation. Note that predicate `reachable/2` succeeds by pruning the search space with a cut operation, which makes batched scheduling more appropriate for this particular example.

6 Conclusions

In this work, we presented the design and implementation of YapTab to support dynamic mixed-strategy evaluation of tabled logic programs. Our approach proposes the ability to combine batched scheduling with local scheduling at the subgoal level with minor changes to the tabling engine. These changes introduced insignificant overheads on YapTab's performance. Moreover, our results show that dynamic mixed-strategies can be extremely important to improve the performance of some applications.

The proposed data structures and algorithms can also be easily extended to support dynamic switching from batched to local scheduling and vice versa, while a generator is still producing answers. In particular, we plan to study how such flexibility can be used to design a more *aggressive* approach for applications that do a lot of pruning over the table space, such as ILP applications. We also plan to further investigate the impact of combining both strategies in other application areas.

Acknowledgments

We are very thankful to Nuno Fonseca for his support with the April ILP System. This work has been partially supported by APRIL (POSI/SRI/40749/2001), Myddas (POSC/EIA/59154/2004) and by funds granted to LIACC through the Programa de Financiamento Plurianual, Fundação para a Ciência e Tecnologia and Programa POSC.

References

1. Sagonas, K., Swift, T.: An Abstract Machine for Tabled Execution of Fixed-Order Stratified Logic Programs. ACM Transactions on Programming Languages and Systems **20** (1998) 586–634
2. Rocha, R., Silva, F., Santos Costa, V.: YapTab: A Tabling Engine Designed to Support Parallelism. In: Conference on Tabulation in Parsing and Deduction. (2000) 77–87
3. Demoen, B., Sagonas, K.: CHAT: The Copy-Hybrid Approach to Tabling. Future Generation Computer Systems **16** (2000) 809–830
4. Guo, H.F., Gupta, G.: A Simple Scheme for Implementing Tabled Logic Programming Systems Based on Dynamic Reordering of Alternatives. In: International Conference on Logic Programming. Number 2237 in LNCS, Springer-Verlag (2001) 181–196
5. Zhou, N.F., Shen, Y.D., Yuan, L.Y., You, J.H.: Implementation of a Linear Tabling Mechanism. Journal of Functional and Logic Programming **2001** (2001)
6. Sagonas, K., Swift, T., Warren, D.S.: XSB as an Efficient Deductive Database Engine. In: ACM SIGMOD International Conference on the Management of Data, ACM Press (1994) 442–453
7. Freire, J., Swift, T., Warren, D.S.: Beyond Depth-First: Improving Tabled Logic Programs through Alternative Scheduling Strategies. In: International Symposium on Programming Language Implementation and Logic Programming. Number 1140 in LNCS, Springer-Verlag (1996) 243–258
8. Freire, J., Swift, T., Warren, D.S.: Taking I/O seriously: Resolution reconsidered for disk. In: International Conference on Logic Programming. (1997) 198–212
9. Freire, J.: Scheduling Strategies for Evaluation of Recursive Queries over Memory and Disk-Resident Data. PhD thesis, State University of New York (1997)
10. Freire, J., Warren, D.S.: Combining Scheduling Strategies in Tabled Evaluation. In: Workshop on Parallelism and Implementation Technology for Logic Programming. (1997)
11. Chen, W., Swift, T., Warren, D.S.: Efficient Top-Down Computation of Queries under the Well-Founded Semantics. Journal of Logic Programming **24** (1995) 161–199
12. Ramakrishnan, I.V., Rao, P., Sagonas, K., Swift, T., Warren, D.S.: Efficient Access Mechanisms for Tabled Logic Programs. Journal of Logic Programming **38** (1999) 31–54
13. Muggleton, S.: Inductive Logic Programming. In: Conference on Algorithmic Learning Theory, Ohmsma (1990) 43–62
14. Fonseca, N., Camacho, R., Silva, F., Santos Costa, V.: Induction with April: A Preliminary Report. Technical Report DCC-2003-02, Department of Computer Science, University of Porto (2003)

Nondeterminism Analysis of Functional Logic Programs*

Bernd Braßel and Michael Hanus

Institut für Informatik, CAU Kiel, Olshausenstr. 40,
D-24098 Kiel, Germany
{bbr, mh}@informatik.uni-kiel.de

Abstract. Information about the nondeterminism behavior of a functional logic program is important for various reasons. For instance, a nondeterministic choice in I/O operations results in a run-time error. Thus, it is desirable to ensure at compile time that a given program is not going to crash in this way. Furthermore, knowledge about nondeterminism can be exploited to optimize programs. In particular, if functional logic programs are compiled to target languages without builtin support for nondeterministic computations, the transformation can be much simpler if it is known that the source program is deterministic.

In this paper we present a nondeterminism analysis of functional logic programs in form of a type/effect system. We present a type inferencer to approximate the nondeterminism behavior via nonstandard types and show its correctness w.r.t. the operational semantics of functional logic programs. The type inference is based on a new compact representation of sets of types and effects.

1 Introduction

Functional logic languages [8] aim to integrate the best features of functional and logic languages in order to provide a variety of programming concepts. For instance, the concepts of demand-driven evaluation, higher-order functions, and polymorphic typing from functional programming can be combined with logic programming features like computing with partial information (logic variables), constraint solving, and nondeterministic search for solutions. This combination leads to optimal evaluation strategies [2] and new design patterns [3] that can provide better programming abstractions, e.g., for implementing graphical user interfaces [10] or dynamic web pages [11]. One of the key points in this integration is the treatment of nondeterministic computations. Usually, the top-level of an application written in a functional logic language is a sequence of I/O operations applied to the outside world (e.g., see [25]). Since the outside world (e.g., file system, Internet) cannot be copied in nondeterministic branches, all nondeterminism in logic computations must be encapsulated, as proposed in [4,13] for the declarative multi-paradigm language Curry [15], otherwise a run-time

* The research described in this paper has been partially supported by the German Research Council (DFG) under grant Ha 2457/5-1.

M. Gabbrielli and G. Gupta (Eds.): ICLP 2005, LNCS 3668, pp. 265–279, 2005.

error occurs. Therefore, it is desirable to ensure at compile time that this cannot happen for a given program. Since this is undecidable in general, one can try to approximate the nondeterminism behavior by some program analysis. As a further motivation, the results of such an analysis can also be used for program optimization. For instance, if functional logic programs are compiled to target languages without builtin support for nondeterministic computations (e.g., imperative or functional languages), the compilation process can be considerably simplified for deterministic source programs.

Existing determinism analyses for (functional) logic languages cannot be directly adapted to Curry due to its advanced lazy operational semantics ensuring optimal evaluation for large classes of programs [2]. This demand-driven semantics has the effect that the occurrence of nondeterministic choices depends on the demandedness of argument evaluation (see also [14]). Therefore, analyses for languages like Prolog [24,6], Mercury [16], or HAL [7] do not apply because they do not deal with lazy evaluation. On the other hand, analyses proposed for narrowing-based functional logic languages dealing with lazy evaluation cannot handle residuation, which additionally exists in Curry and is important to connect external operations, and rely on the non-ambiguity condition [20] which is too restrictive in practice. Furthermore, these analyses are either applied during run time (like in Babel [20] and partially in K-Leaf [19]), or are unable to derive groundness information for function calls in arguments (like in K-Leaf).

We present a static analysis of functional logic programs with a demand-driven evaluation strategy. The analysis has the form of a type/effect system [22]. Such systems can be seen as extensions of classical type systems known from functional languages. In our analysis the types represent information about the groundness of the considered expressions, and the effects provide information about the possible source of nondeterministic branches. The inclusion of groundness information is necessary since the same function might evaluate deterministically or not, depending on the instantiation of its arguments. The idea of this type/effect system has been proposed in [14]. In the current paper we propose a slightly modified system and show its correctness w.r.t. a recently developed high-level operational semantics of functional logic programs [1] that covers all operational aspects, in particular, the sharing of subterms which is important in practice but has not been addressed in [14]. Furthermore, we present a new method to *infer* types and effects (which was not covered in [14]) and show the correctness of this inference. In order to make the type/effect inference feasible, we introduce a new compact representation of sets of types and effects.

Due to lack of space, all proofs and details about the implementation are omitted. They can be found in the full version of the paper that is available from `http://www.informatik.uni-kiel.de/~mh`.

2 The Type/Effect Analysis

In this section we define a type/effect system based on the ideas in [14] and show its correctness w.r.t. the operational semantics of functional logic programs

developed in [1]. We assume familiarity with the basic ideas of functional logic programming (see [8] for a survey).

2.1 Flat Functional Logic Programs

Since a determinism analysis of functional logic programs should provide information about nondeterministic branches that might occur during run time, it requires detailed information about the operational behavior of programs. Recently, it has been shown that an intermediate *flat* representation of programs [12] is a good basis to provide this information. In flat programs, the pattern matching strategy (which determines the demand-driven evaluation of goals) is explicitly given by case expressions. This flat representation constitutes the kernel of modern functional logic languages like Curry [9,15] or Toy [21]. Thus, our approach is applicable for general lazy functional logic languages although the examples and implementation are for Curry.

$P ::= D_1 \ldots D_m$	(program)	*Domains*
$D ::= f(x_1, \ldots, x_n) = e$	(function definition)	
$e ::= x$	(variable)	$P_1, P_2, \ldots \in \mathit{Prog}$ (Programs)
$\mid s(e_1, \ldots, e_n)$	(constructor or function call)	$x, y, z, \ldots \in \mathit{Var}$ (Variables)
		$a, b, c, \ldots \in \mathcal{C}$ (Constructors)
$\mid \mathit{let}\ x = e_1\ \mathit{in}\ e_2$	(let binding)	$f, g, h, \ldots \in \mathcal{F}$ (Functions)
$\mid e_1\ \mathit{or}\ e_2$	(disjunction)	$s, t, u, \ldots \in \mathcal{C} \cup \mathcal{F}$
$\mid \mathit{case}\ e\ \mathit{of}\ \{\overline{p_k \to e_k}\}$	(rigid case)	$p_1, p_2, \ldots \in \mathit{Pat}$ (Patterns)
$\mid \mathit{fcase}\ e\ \mathit{of}\ \{\overline{p_k \to e_k}\}$	(flexible case)	$e, e_1, e_2, \ldots \in \mathit{Exp}$ (Expressions)
$p ::= c(x_1, \ldots, x_n)$	(pattern)	

Fig. 1. Syntax of flat programs

The syntax of flat programs is shown in Figure 1. There and in the following we write $\overline{o_k}$ to denote a sequence $o_1, \ldots, o_k$ ($\overline{o_0}$ is empty). A flat program is a set of function definitions, i.e., the arguments are pairwise different variables and the right-hand side consists of variables, constructor/function applications, let bindings, disjunctions to represent nondeterministic choices, and case expressions to represent pattern matching. The difference between *case* and *fcase* corresponds to principles of residuation and narrowing: if the argument is a logic variable, *case* suspends whereas *fcase* proceeds with a nondeterministic binding of the variable in one branch of the case expression (cf. Section 2.1). A flat program is called *normalized* if all arguments of constructor and function calls are variables. Any flat program can be normalized by introducing *let* expressions [1,18]. The operational semantics is defined *only on normalized* programs in order to model sharing, whereas our type-based analysis is defined for flat programs.

Any Curry program can be translated into this flat representation.

Example 1 (Flat Curry representation). The concatenation function on lists

```
app []     ys = ys
app (x:xs) ys = x : app xs ys
```

is represented by the (normalized) flat program

```
app(xs,ys) = fcase xs of {[] -> ys, z:zs -> let a = app(zs,ys) in z:a}
```

Note that all variables occurring in the right-hand side of a function definition must occur in the left-hand side or be introduced by an enclosing let binding. In order to avoid a special declaration for logic variables, they are represented as self-circular let bindings. E.g., the expression "`let xs=xs in app(xs,[])`" introduces the logic variable `xs` in the expression "`app(xs,[])`".

Based on the principles developed in [18], [1] introduces a natural semantics of normalized flat programs. As this semantics adequately resembles the behavior of modern multi-paradigm languages like Curry [9,15] or Toy [21], it is a good reference to show the correctness of program analyses for functional logic languages. There are some special properties of this semantics we have to consider in order to examine our type/effect analysis.

The only difference we have to consider is the treatment of circular data structures which are allowed in [1]. Since the nondeterminism analysis of [14] as well as ours do not consider such structures, we restrict the set of permissible programs to those without circular data structures. This is not a restriction in practice since the current definitions of Curry [9,15] or Toy [21] do not support such structures. Note that the definition of infinite data structures is still possible since they can be defined by functions, e.g., "`repeat x = x : repeat x`".

Definition 1 (Cycle restriction). *The set of programs $P^{\otimes}$ is defined exactly like P except for the definition of let-clauses: For any expression let $x = e$, if x occurs in e then $e = x$.*

This definition allows only non-circular let-expressions with the single exception being logic variables defined by "`let x=x`".

Having defined the set of programs we want to examine, we now turn to the semantics of these programs. In contrast to an operational semantics based on term rewriting (e.g., [2,15]), the semantics considered here correctly models sharing of common subterms as necessary for optimal evaluation and done in implementations. Sharing is modeled by introducing *heaps*. A heap, here denoted by Γ, Δ, or Θ, is a finite partial mapping from variables to expressions (the *empty heap* is denoted by `[]`). The value associated to variable x in heap Γ is denoted by $\Gamma[x]$. $\Gamma[x \mapsto e]$ denotes a heap Γ' with $\Gamma'[x] = e$ (in the rules, this notation is used as a condition as well as an update of a heap). A logic variable x is represented by a circular binding of the form $\Gamma[x] = x$. A *value* v is a constructor-rooted term $c(\overline{x_n})$ (i.e., a term whose outermost function symbol is a constructor symbol) or a logic variable (w.r.t. the associated heap). ρ represents a substitution of variables in expressions by other variables, i.e., ρ is a renaming.

The natural semantics uses judgements of the form "$\Gamma : e \Downarrow \Delta : v$" which are interpreted as: "In the context of heap Γ, the expression e evaluates to value v and produces a new heap Δ." Figure 2 shows the rules defining this semantics (also called big-step semantics) of normalized flat programs, where the current program P is considered as a global constant. The rules VarCons and VarExp

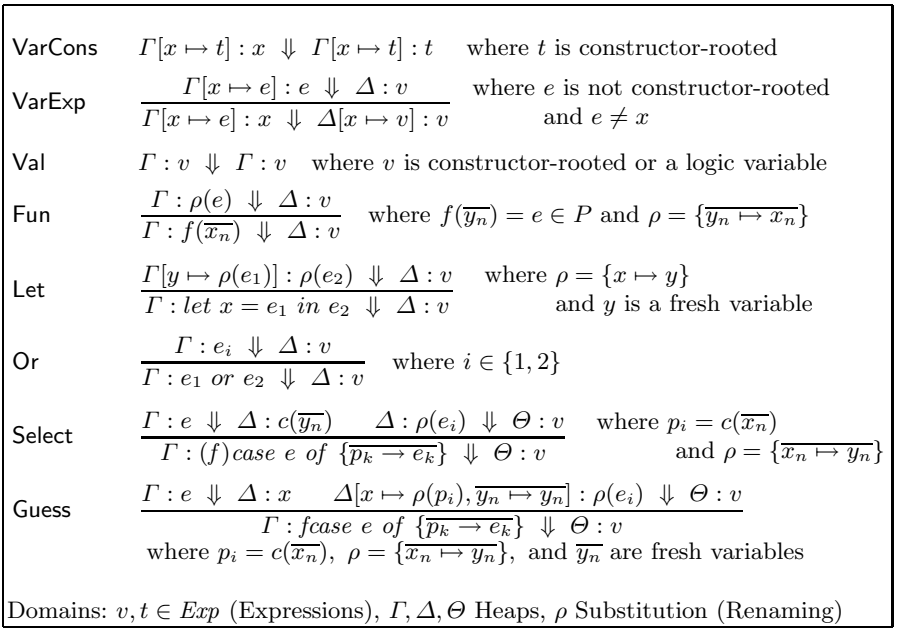


Fig. 2. Natural semantics of *normalized* flat programs [1]

are responsible to retrieve expressions from the heap, the difference being that VarCons retrieves values, whereas the expressions retrieved by VarExp have to be further evaluated. VarCons and Val form the base of proof trees generated by the big-step semantics. They treat values, i.e., expressions which are either logic variables or evaluated to head normal form. VarCons is merely a shortcut for applying VarExp and Val once each. The rule Let introduces a new binding for the heap, Fun is used to unfold function applications, and Or introduces a nondeterministic branching. Select and Guess deal with `case` expressions. Select determines the corresponding branch to continue with, if the first argument of `case` was reduced to a constructor rooted term. Guess treats the case that the first argument evaluates to a logic variable. If so, Guess introduces a nondeterministic branching where the logic variable is bound nondeterministically to one of the patterns of the `case`-expression. Remember that there are two kinds of `case`-expressions in flat programs. Only `fcase` (with `f` for "flexible") can introduce nondeterminism if the number of branches is greater than one. In short, `fcase` models narrowing whereas `case` is used to model the operational behavior of residuation. We often write `(f)case` to denote both kinds of cases.

The restriction to non-circular data structures introduced in Definition 1 implicates that no circular structures are produced during program execution, which is the content of Lemma 1.

Lemma 1 (Well-founded heaps). *Let Γ be a heap occurring in a derivation $[] : e \Downarrow \Delta : v$ w.r.t. a program $P \in P^{\otimes}$, and $\Gamma_0 := \Gamma$, $\Gamma_{n+1} := \hat{\Gamma} \circ \Gamma_n$ for*

$n \geq 0$ where $\hat{\Gamma}$ is the homomorphic extension of Γ. Then there is no non-trivial circular structure in Γ, i.e., there is no natural number n for which a variable x exists with $\Gamma_n(x) = t$ such that x occurs in t and $t \neq x$.

As programs in $P^{\otimes}$ produce only well founded heaps, we can extract a complete substitution from the heap as follows:

Definition 2 (σ_Γ). *For a well-founded heap Γ, Γ^* is defined as the least fixpoint of $\{\Gamma_0 := \Gamma, \Gamma_{n+1} := \hat{\Gamma} \circ \Gamma_n\}$. Then σ_Γ is the substitution with $dom(\sigma_\Gamma) = dom(\Gamma)$ and $\{x \mapsto \Gamma^*(x)\}$.*

Example 2 (Substitution σ_Γ). Consider the following definition:

```
main  = let z = 3 in let y = c(z,z) in let x = f(y) in f(x)
```

Evaluating `main` yields heap $\Gamma := [][\texttt{z'} \mapsto \texttt{3}][\texttt{y'} \mapsto \texttt{c(z',z')}][\texttt{x'} \mapsto \texttt{f(y')}]$. For this heap, $\sigma_\Gamma(\texttt{z'}) = \texttt{3}$, $\sigma_\Gamma(\texttt{y'}) = \texttt{c(3,3)}$ and $\sigma_\Gamma(\texttt{x'}) = \texttt{f(c(3,3))}$.

The main purpose of Definition 1 is to ensure that the substitution σ_Γ is well defined.

2.2 Type/Effect Analysis Revisited

The basic ideas of the type/effect analysis used in this paper were first proposed in [14]. Here we use a slightly different definition (e.g., without a rule for subtyping but `let` clauses to describe sharing that is not covered in [14]) and base it on the natural semantics introduced in the previous section. The analysis uses the idea to attach to expressions and functions two kinds of information: a type to describe the ground status and an effect to describe the nondeterminism behavior. Similarly to standard types in typed functional languages, there are also typing rules that define well-typed expressions w.r.t. this type/effect system. Before defining these rules, two preliminary Definitions are needed. The analysis of a given program is always performed w.r.t. a *type environment* E which associates types/effects to functions, constructors and variables in the given program. Such an association is called *type annotation* and denoted by $s :: \overline{\tau}_n \xrightarrow{\varphi} \tau$ (resp. $s :: \tau/\varphi$ for constants or variables). Note that there may be more than one type annotation for a function or constructor. The purpose of the type inference described in Section 3 is to provide a method to derive appropriate type environments. In this section, we assume that a correct type environment (defined below) is given. In a type annotation $s :: \overline{\tau_n} \xrightarrow{\varphi} \tau$ for a function or constructor s each $\tau_{(i)}$ describes whether the corresponding argument or result of the function is a *ground* value, denoted by G, or if it might contain logic variables, and, hence, is of *any* value, denoted by A. The set of *effects* φ describes the possible causes for nondeterminism which might occur while evaluating s, if s is a function. Effects are either *or* or *guess*. The meaning of these effects is that one of the nondeterministic rules Or or Guess could be applied while evaluating an expression or function.

Definition 3 (Type/Effects, Type Annotation, Type Environment). *The set of types $\mathcal{T}$ is defined as $\mathcal{T} = \{A, G\}$, the set of Effects $\mathcal{E}$ is defined as*

$\mathcal{E}=\{or, guess\}$, *the set of type/effects for arity* n *is defined as* $\mathcal{TE}_n=\{\overline{\tau_n} \xrightarrow{\varphi} \tau \mid \tau, \tau_i \in \mathcal{T}, \varphi \subseteq \mathcal{E}\}$. *For* $n = 0$ *instead of* $\xrightarrow{\varphi} \tau$ *we write* τ/φ. *And, finally, a type environment* E *is a subset of the set of all type annotations* $\mathcal{TA} = \{x :: \chi \mid x$ *is a variable*, $\chi \in \mathcal{TE}_0\} \cup \{s :: \xi \mid s \in \mathcal{F} \cup \mathcal{C}, s$ *is of arity* $n, \xi \in \mathcal{TE}_n\}$.

Before defining the typing rules and giving an example, we have to introduce an ordering on the types to compare different abstract results. In general, an *ordering* is a reflexive, transitive and anti-symmetric relation.

Definition 4 (Type/effect ordering $\leq$, max, min). $\leq$ *denotes an ordering on types and effects that is the least order relation satisfying* $G \leq A$ *and, for effects* $\varphi \leq \varphi'$ *iff* $\varphi \subseteq \varphi'$. *Type/effects are ordered by* $\tau_1 \xrightarrow{\varphi_1} \tau_2 \leq \tau_1' \xrightarrow{\varphi_2} \tau_2'$ *iff* $\tau_1' \leq \tau_1$, $\tau_2 \leq \tau_2'$ *and* $\varphi_1 \leq \varphi_2$. *Furthermore,* $max(\overline{\tau_k})$ *(resp.* $min(\overline{\tau_k})$*) denotes the maximum (minimum) of the* $\overline{\tau_k}$ *with respect to* $\leq$.

Note the difference between argument and result in the definition of $\leq$ for functional types. Informally speaking, for functions with the same result type, it holds: the bigger the argument type, the smaller is the type of the whole function. This makes perfect sense if we think of the type as a grade of nondeterminism. A function of type $A \to G$ is more deterministic than one of type $A \to A$. However, $A \to A$ is still more deterministic than $G \to A$ because a function of the latter type might not merely map logic variables to logic variables but could introduce new ones. We are now ready to define the typing rules as given in Figure 3.

Example 3 ((In)correct type annotation). Consider the (flat) function

```
and(x,y) = fcase x of {False -> False; True -> y}
```

Correct types for `and` would be $GA \to A$ and $GG \to G$. The first type can be intuitively read as: "If the first argument is ground and the second possibly

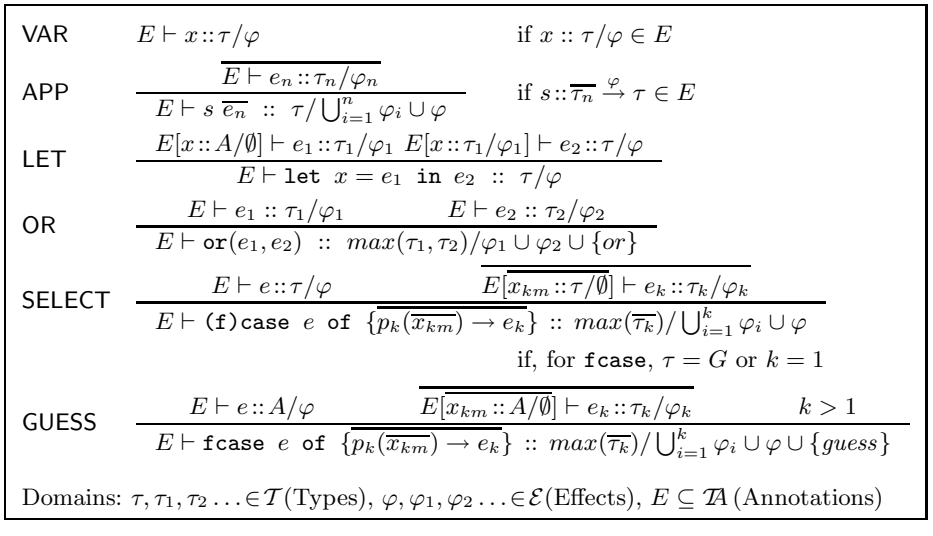


Fig. 3. Typing rules for flat expressions

contains a logic variable, then the result may also contain a logic variable." However, $AG \to A$ is not a valid type. If the first argument is a logic variable, `fcase` will instantiate this variable nondeterministically (cf. Figure 2). Thus, the correct type for these input arguments is $AG \overset{\{guess}{\to} G$. The difference in the actual type check by the rules of Figure 3 is that rule SELECT is applicable for input vector GA, whereas the case AG is covered by rule GUESS.

The correctness of type annotations is now defined in two steps.

Definition 5 (Constructor-correct). *A type environment E is called correct with respect to constructor symbols, or* constructor-correct *for short, iff E contains only the types $c :: \overline{\tau_n} \to max(\overline{\tau_n})$ for any constructor symbol c.*

This definition implies that constructors do not influence the deterministic type of their arguments at all. If any argument is of type A, then the whole term is as well. Furthermore, constructors do never yield any nondeterministic effect. Constructor-correctness is a requirement for our definition of general correctness.

Definition 6 (Correctness). *A type annotation $f :: \overline{\tau_n} \overset{\varphi}{\to} \tau$ contained in a type environment E is* correct *for a definition $f(\overline{x_n}) = e$ if $E[\overline{x_n :: \tau_n/\emptyset}] \vdash e :: \tau/\varphi$. E is* correct *if it is constructor-correct and contains only correct type annotations.*

The aim of this section is to show that correct type environments correctly indicate the nondeterminism caused by the evaluation of a given function. Whenever the evaluation of a function call $f\ \overline{e_n}$ involves a nondeterministic branching by an *or* or a flexible case expression, a correct type environment must contain the corresponding type indicating the effect *or* or *guess*. And whenever the correct type environment indicates that a function f with arguments of a certain type evaluates to a ground term, then no evaluation of f with corresponding arguments yields a result containing a logic variable. The first step towards proving this correctness is the observation that expressions of the same type are indistinguishable by the type/effect system.

Lemma 2 (Substitution Lemma). *Let E be a correct type environment for a flat program. Then for each expression e holds: $E[\overline{x_n :: \tau_n/\emptyset}] \vdash e :: \tau/\varphi$ if and only if replacing each x_i (by a substitution σ) with a term e_i of the same type also yields the same type for e, i.e., $\overline{E \vdash e_n :: \tau_n/\emptyset}$ also implies $E \vdash \sigma(e) :: \tau/\varphi$. Furthermore, if some of the $\overline{e_n}$ have a non-empty effect, i.e., $E \vdash e_i :: \tau_i/\varphi_i$, then $E \vdash \sigma(e) :: \tau/\bigcup_{i=1}^{n} \varphi_i \cup \varphi$, i.e., the type τ of e remains the same but the effect inferred for e is larger.*

Lemma 2 is a typical requirement in type systems. The correctness of the type analysis is mainly based on the following theorem. We use the notation E^{free} for a type environment that extends a type environment E by annotations for free variables, i.e., if $x :: \tau/\varphi \in E$, then $x :: \tau/\varphi \in E^{free}$, otherwise $x :: A/\emptyset \in E^{free}$.

Theorem 1 (Type-descending). *Let E be a correct type environment for a non-circular program P in $P^{\otimes}$, e an expression with $\Gamma : e \Downarrow \Delta : v$ built in a proof tree for an expression $[] : e' \Downarrow \Delta' : v'$, and $E^{free} \vdash \sigma_\Gamma(e) :: \tau/\varphi$ and $E^{free} \vdash \sigma_\Delta(v) :: \tau'/\varphi'$. Then $\tau \geq \tau'$ and $\varphi \supseteq \varphi'$.*

Theorem 1 implies that the type analysis correctly indicates the evaluation of expressions to ground terms:

Corollary 1 (Correctness for ground terms). *Let E be a correct type environment for a non-circular program P in $P^{\otimes}$. If, for some expression e, $E^{free} \vdash e :: G/\varphi$ and e reduces in finitely many steps to a* value *v (i.e., a term without defined function symbols), then v is a ground term.*

The last property to prove is that the analysis is not only decreasing for types but also gathers all effects. This finally leads to the proposition that all potential effects in the evaluation of a given expression are correctly predicted.

Lemma 3 (Gathering of effects). *Let E be a correct type environment for a non-circular program P in $P^{\otimes}$. Let Γ be a well-founded heap, T be a proof tree for $\Gamma : e \Downarrow \Delta : v$ and $E^{free} \vdash \sigma_\Gamma(e) :: \tau/\varphi$. Then, for any $\Gamma' : e' \Downarrow \Delta' : v'$ in T with $E^{free} \vdash \sigma_\Gamma(e') :: \tau'/\varphi'$, $\varphi' \subseteq \varphi$ holds.*

Lemma 3 implies the final important property of the type/effect system:

Corollary 2 (Identification of nondeterminism). *If, for a non-circular program $P \in P^{\otimes}$ and expression e, there are two proof trees T and T' for $[] : e \Downarrow \Delta : v$ and $[] : e \Downarrow \Delta' : v'$ differing in more than variable names, then any type of e w.r.t. a correct type environment for P contains an effect* `or` *or* `guess`*.*

3 Type/Effect Inference

In this section we introduce a method to infer the types and effects introduced in the previous section. In order to obtain a feasible inference method, we introduce *base annotations*, a compact representation of sets of types and effects.

3.1 Base Annotations

The definition of well-typed programs is usually not sufficient. Instead one wants to compute all of the correct type environments for a given program. On a first glance, this problem seems quite hard, as for each n-ary function there are 2^{n+1} possible types even with an empty effect. However, a closer observation shows that one need only to consider $n+1$ types, namely the type where all arguments are ground (G) and the n types where a single argument is any (A) and all others are ground. The remaining types can be deduced by combining these $n+1$ base types, which we also call a *type base*. For instance, the type for $GGAGAG \xrightarrow{\varphi} \tau$ is the result of combining the type for $GGGGAG \xrightarrow{\varphi_1} \tau_1$ and $GGAGGG \xrightarrow{\varphi_2} \tau_2$. Before introducing the compact representation of type/effects, we first show the soundness of this combination of two types.

Definition 7 (Supremum $\sqcup$, $\tau/\varphi_1/\varphi_2$). *For types $\tau_1, \tau_2 \in \mathcal{T}$ the type $\tau_1 \sqcup \tau_2$ is their supremum, i.e. $max(\tau_1, \tau_2)$. For type/effects $\overline{\tau_n} \xrightarrow{\varphi_1} \tau, \overline{u_n} \xrightarrow{\varphi_2} u \in \mathcal{TE}_n$, the type/effect $(\overline{\tau_n} \xrightarrow{\varphi_1} \tau) \sqcup (\overline{u_n} \xrightarrow{\varphi_2} u)$ denotes $\overline{max(\tau_n, u_n)} \xrightarrow{\varphi_1 \xrightarrow{\varphi_2}} max(\tau, u)$. For type environments $E_1, E_2 \subseteq \mathcal{TA}$, $E_1 \sqcup E_2 = \{s :: \xi \sqcup \xi' \mid s \in Var \cup \mathcal{C} \cup F, s :: \xi, s :: \xi' \in E_1 \cup E_2\}$. Finally, $\bigsqcup x$ denotes the supremum of a set x and the notation $\tau/\varphi/\varphi'$ is used to denote $\tau/\varphi \cup \varphi'$.*

Lemma 4 (Compositionality). *If, for any function declaration $f\ \overline{x_n} = e$, there are correct type annotations $\mathcal{A}_1 = \overline{\tau_n} \xrightarrow{\varphi_1} \tau$ and $\mathcal{A}_2 = \overline{u_n} \xrightarrow{\varphi_2} u$ for environments E_1 and E_2, respectively, then $\mathcal{A}_1 \sqcup \mathcal{A}_2$ is also a correct type for f for the environment $E_1 \sqcup E_2$.*

Lemma 4 ensures that every correct type can be easily derived from a correct type base, i.e., a set containing the $n + 1$ basic types as mentioned above. This fact is the basis for the compact representation of correct type environments. Instead of an exponential number of types, it is sufficient to consider only the $n + 1$ elements of a type base. Furthermore, we can pack the information of the type base into a single structure with at most n elements, which we call a *base annotation* for a function. A base annotation for a function f is either $\underline{A}$ (or $\underline{G}$) if the result of f is of type A (or G) regardless of the types of its arguments, or it is a term indicating which arguments influence the type of f. For instance, if f has type A whenever its first argument is of type A, then the base annotation for f is Π^1 (Π denotes a kind of projection). If f has type A whenever either its second or its fourth argument is A, the annotation for f is $\Pi^2 \sqcup \Pi^4$. To determine the type for a given application of f, the Πs are replaced by the actual types of the corresponding arguments. For complex annotations, like $\Pi^2 \sqcup \Pi^4$, the result type is the supremum of the replacements. Therefore, we reuse the symbol $\sqcup$ although it is used here as a term constructor for base annotations. Furthermore, the effect *guess* is extended by a base annotation, e.g., $\underline{guess}(\Pi^1)$. The reason for this will be explained soon.

Definition 8 (Syntax of base annotations). *Let $s \in \mathcal{C} \cup F$ be of arity n. Then the set of well formed base types for s, $\mathcal{BT}_s$, is the smallest set satisfying: $(\{\underline{G}, \underline{A}, \Pi^1, \ldots, \Pi^n\} \subseteq \mathcal{BT}_s) \wedge (\nu, \mu \in \mathcal{BT}_s \Rightarrow \nu \sqcup \mu \in \mathcal{BT}_s)$. The set of well formed base effects $\mathcal{BE}_s$ for s is the smallest set satisfying: $(\{\underline{or}, \underline{guess}\} \subseteq \mathcal{BE}_s) \wedge (\nu \in \mathcal{BT}_s \Rightarrow \underline{guess}(\nu) \in \mathcal{BE}_s)$. The set of well formed base annotations $\mathcal{BA}_s$ is defined as $\mathcal{BA}_s = \{s :: \nu/\varepsilon \mid \nu \in \mathcal{BT}_s, \varepsilon \in \mathcal{BE}_s\}$. We also use $\mathcal{BT}$, $\mathcal{BE}$, $\mathcal{BA}$ (without index) to denote the set of all base types, effects, annotations.*

Example 4 (Some correct base annotations).
-For each n-ary constructor c: $\underline{c} = \Pi^1 \sqcup \ldots \sqcup \Pi^n/\emptyset$ if $n > 0$, otherwise $\underline{c} = \underline{G}/\emptyset$

`-f1 x = 1`	`f1` :: $\underline{G}/\emptyset$
`-f2 = let x=x in x`	`f2` :: $\underline{A}/\emptyset$
`-f3 x y = y`	`f3` :: $\Pi^2/\emptyset$
`-f4 x y = fcase x of {1->1; 2->y}`	`f4` :: $\Pi^2/\{\underline{guess}(\Pi^1)\}$

Function `f4` also illustrates the meaning of a guess effect depending on a type. The rule Guess of the natural semantics (Figure 2) will only be applied if the first argument of `f4` is a logic variable. Therefore, $\underline{guess}(\Pi^1)$ will yield the effect $\underline{guess}$ only if Π^1 is replaced by type $\underline{A}$ and no effect if it is replaced by type $\underline{G}$.

The general meaning of base annotations is best conveyed by defining the set of type/effects each of them represents. In the next section we will show how to compute base annotations for a given program. For both purposes, we need the

notion of a normal form for base annotations as a means to effectively decide the equivalence on base annotations. The normal forms are obtained by rewriting with the following set of confluent and terminating rewrite rules.

Definition 9 (Normal form $\lfloor\nu/\varepsilon\rfloor$). *We denote by $\lfloor\nu\rfloor$ and $\lfloor\varepsilon\rfloor$ the simplification of base type ν and base effect ε, respectively, with the rules*

$$\begin{array}{c} \underline{G} \sqcup \nu \to \nu \qquad \nu \sqcup \underline{G} \to \nu \\ \underline{A} \sqcup \nu \to \underline{A} \qquad \nu \sqcup \underline{A} \to \underline{A} \\ \Pi^i \sqcup \Pi^j \to \Pi^j \sqcup \Pi^i, i > j \\ \nu \sqcup \nu \to \nu \end{array} \qquad \begin{array}{c} \{\underline{guess}(\underline{G})\} \to \{\} \\ \underline{guess}(\underline{A}) \to \underline{guess} \\ \underline{guess}(\nu) \to \underline{guess}(\lfloor\nu\rfloor) \end{array}$$

(the simplification rules for guess become applicable after the transformation shown in the subsequent definition, where the last rule only maintains the sorting of the Π by index). Similarly, $\lfloor\nu/\varepsilon\rfloor$ denotes component-wise simplification.

As motivated above, the base annotations of a given function represents all of its (minimal) types. The following definition describes this representation in detail.

Definition 10 (Base annotations and types). *Let f be an n-ary function. To each base annotation b for f we associate a set of type annotations $types(n, \lfloor b\rfloor)$:*

$$types(n, \underline{G}/\varepsilon) = \{\overline{\tau_n} \xrightarrow{eff(\overline{\tau_n},\varepsilon)} G \mid \overline{\tau_n} \in \mathcal{T}\}$$
$$types(n, \underline{A}/\varepsilon) = \{\overline{\tau_n} \xrightarrow{eff(\overline{\tau_n},\varepsilon)} A \mid \overline{\tau_n} \in \mathcal{T}\}$$
$$types(n, \Pi^{i_1} \sqcup \ldots \sqcup \Pi^{i_j}/\varepsilon) =$$
$$\bigsqcup(\{G^n \xrightarrow{eff(G^n,\varepsilon)} G\} \cup \{\underbrace{G^{k-1}AG^{n-k}}_{\tau_k} \xrightarrow{eff(\tau_k,\varepsilon)} A \mid k \in \{i_1 \ldots i_j\}\})$$

where G^j is the usual notation for a sequence of Gs with length j and $eff(\overline{\tau_n}, \varepsilon) = \lfloor\{\overline{\Pi^n \mapsto \tau_n}\}\varepsilon\rfloor$.[1]

Example 5 (Continuing Example 4). The types associated with the base annotations from Example 4 are:
-For a unary constructor $c : types(1, \Pi^1/\emptyset) = \{G \to G, A \to A\}$
-`f1`: $types(1, \underline{G}/\emptyset) = \{G \to G, A \to G\}$
-`f2`: $types(0, \underline{A}/\emptyset) = \{A/\emptyset\}$
-`f3`: $types(2, \Pi^2/\emptyset) = \{GG \to G, GA \to A, AG \to G, AA \to A\}$
-`f4`: $types(2, \Pi^2/\{\underline{guess}(\Pi^1)\}) = \{GG \to G, GA \to A, AG \xrightarrow{\{guess} G, AA \xrightarrow{\{guess} A\}$

This representation of groundness information has some similarities to the domain *Prop* of propositional formulas used in groundness analysis of logic programs [5]. However, we are interested in covering all sources of nondeterminism which is usually the effect non-ground function arguments (apart from function definitions with overlapping right-hand sides, represented by *or*). Therefore, we

[1] $\{\overline{\Pi^n \mapsto \tau_n}\}\varepsilon$ denotes the replacement of all occurrences of Π^i by τ_i in ε for $i \in \{1, \ldots, n\}$.

$$\underline{\text{VAR}} \qquad B \rhd x :: \nu/\varepsilon \qquad \text{if } x :: \nu/\varepsilon \in B$$

$$\underline{\text{APP}} \qquad \frac{\overline{B \rhd e_n :: \nu_n/\varepsilon_n}}{B \rhd f\ \overline{e_n}\ :: \lfloor\{\overline{\Pi^n \mapsto \nu_n/\varepsilon_n}\}\nu/\{\overline{\Pi^n \mapsto \nu_n}\}\varepsilon\rfloor} \qquad \text{if } f :: \nu/\varepsilon \in B$$

$$\underline{\text{LET}} \qquad \frac{B[x :: \underline{A}/\emptyset] \rhd e_1 :: \nu_1/\varepsilon_1 \qquad B[x :: \nu_1/\varepsilon_1] \rhd e_2 :: \nu/\varepsilon}{B \rhd \texttt{let}\ x = e_1\ \texttt{in}\ e_2\ ::\ \nu/\varepsilon}$$

$$\underline{\text{OR}} \qquad \frac{B \rhd e_1 :: \nu_1/\varepsilon_1 \qquad B \rhd e_2 :: \nu_2/\varepsilon_2}{B \rhd \texttt{or}(e_1, e_2)\ ::\ \nu_1/\varepsilon_1 \sqcup \nu_2/\varepsilon_2 \cup \{or\}}$$

$$\underline{\text{SELECT}} \qquad \frac{B \rhd e :: \nu/\varepsilon \qquad \overline{B[\overline{x_{km}} :: \nu/\emptyset] \rhd e_k :: \nu_k/\varepsilon_k}}{B \rhd \texttt{(f)case}\ e\ \texttt{of}\ \{\overline{p_k(\overline{x_{km}}) \to e_k}\}\ ::\ \bigsqcup_{i=1}^{k} \nu_i/\varepsilon_i \cup \varepsilon}$$

if, for `fcase`, $\nu = \underline{G}$ or $k = 1$

$$\underline{\text{GUESS}} \qquad \frac{B \rhd e :: \nu/\varepsilon \qquad \nu \neq \underline{G} \qquad \overline{B[\overline{x_{km}} :: \nu/\emptyset] \rhd e_k :: \nu_i/\varepsilon_i} \qquad k > 1}{B \rhd \texttt{fcase}\ e\ \texttt{of}\ \{\overline{p_k(\overline{x_{km}}) \to e_k}\}\ ::\ \bigsqcup_{i=1}^{k} \nu_i/\varepsilon_i \cup \varepsilon \cup \lfloor\{\underline{guess}(\nu)\}\rfloor}$$

Domains: $\nu, \nu_1, \nu_2, \ldots \in \mathcal{BT}$ (Base Types), $\varepsilon, \varepsilon_1, \varepsilon_2, \ldots \in \mathcal{BE}$ (Base Effects), $B \subseteq \mathcal{BA}$ (Base Annotations)

Fig. 4. Inference rules

use *projections* Π^i in the base annotations to associate potential nondeterministic behavior to the instantiation of particular arguments.

Finally, we define an ordering on base annotations. This ordering is used to define the type inference in the next section and show its correctness. For the latter purpose, it is important to note that the order is finite.

Definition 11 (Ordering on base annotations $\sqsubseteq$). *The ordering $\sqsubseteq$ is used on base types, base effects, base annotations and sets of base annotations (base environments). It is defined as the least ordering satisfying*

- $\underline{G} \sqsubseteq \nu$ *and* $\nu \sqsubseteq \underline{A}$ *for all* $\nu \in \mathcal{BT}$
- $\Pi^{i_1} \sqcup \ldots \sqcup \Pi^{i_m} \sqsubseteq \Pi^{j_1} \sqcup \ldots \sqcup \Pi^{j_n}$ *if* $\{\Pi^{i_1}, \ldots, \Pi^{i_m}\} \subseteq \{\Pi^{j_1}, \ldots, \Pi^{j_n}\}$
- $\underline{guess}(\nu) \sqsubseteq \underline{guess}$ *and* $\underline{guess}(\nu) \sqsubseteq \underline{guess}(\nu')$ *if* $\nu \sqsubseteq \nu'$ *for all* $\nu, \nu' \in \mathcal{BT}$
- *For* $\varepsilon, \varepsilon' \in \mathcal{BE}$*:* $\varepsilon \sqsubseteq \varepsilon'$ *if* $\forall x \in \varepsilon\ \exists x' \in \varepsilon' : x \sqsubseteq x'$
- *Ordering on base type/effects:* $\nu/\varepsilon \sqsubseteq \nu'/\varepsilon'$ *if* $\lfloor\nu\rfloor \sqsubseteq \lfloor\nu'\rfloor$ *and* $\lfloor\varepsilon\rfloor \sqsubseteq \lfloor\varepsilon'\rfloor$
- *Ordering on base environments:* $B \sqsubseteq B'$ *if* $\forall x \in B\ \exists x' \in B' : x \sqsubseteq x'$

$\sqcup$ *(resp.* $\bigsqcup$*) denotes the* $\sqsubseteq$*-supremum of two (resp. a set of) base annotations.*

3.2 Inferring Base Annotations

After having defined the structure of base annotations, we are ready to define the inference of them. Figure 4 shows the rules to infer base annotations for a given expression. The complete inference is defined as a fix-point iteration on a given flat program. Before we can define the iteration, we need to observe that the inference is monotone, i.e., the inference always computes greater types for greater environments (with respect to $\sqsubseteq$).

Lemma 5 ($\rhd$ respects $\sqsubseteq$). *Let B and B' be two base environments with $B \sqsubseteq B'$. Then, for each e with $B \rhd e :: \nu/\varepsilon$, there is a derivation $B' \rhd e :: \nu'/\varepsilon'$ with $\nu/\varepsilon \sqsubseteq \nu'/\varepsilon'$.*

Because of the monotonicity of $\rhd$, we can define the inference of a base environment as follows:

Definition 12 (Type inference). *The mapping Inf associates to a flat program P a type environment. It is defined by the following fix-point iteration based on the inference system in Figure 4:*

$$\begin{aligned}
Inf_0(P) &= \{c :: \underline{G}/\emptyset \mid c \text{ is a } 0\text{-ary constructor}\} \cup \\
&\quad \{c :: \Pi^1 \sqcup \ldots \sqcup \Pi^n/\emptyset \mid c \text{ is an } n\text{-ary constructor}, n > 0\} \cup \\
&\quad \{f :: \underline{G}/\emptyset \mid f \text{ is a defined function}\} \\
Inf_{i+1}(P) &= \{f :: \lfloor \nu/\varepsilon \rfloor \mid f\ \overline{x_n} = e \in P, Inf_i(P)[\overline{x_n :: \Pi^n/\emptyset}] \rhd e :: \nu/\varepsilon\} \\
Inf(P) &= Inf_j(P), \text{ if } j \in \mathbb{N} \text{ is smallest with } Inf_j(P) = Inf_{j+1}(P)
\end{aligned}$$

After proving that $Inf(P)$ is indeed well defined, we will give examples for inferring types for a given program.

Lemma 6 (Type increase). *Let P be a flat program and f a function defined in P. If $f :: \nu/\varepsilon \in Inf_i(P)$ and $f :: \nu'/\varepsilon' \in Inf_{i+1}(P)$, then $\nu/\varepsilon \sqsubseteq \nu'/\varepsilon'$.*

Corollary 3 ($Inf(P)$ is well defined). *For each finite program P there is a natural number n with $Inf_n(P) = Inf_{n+1}(P)$.*

Corollary 3 states that the iteration of the inference finally terminates.

Example 6 (Type inference). As an example for the type inference, consider the flat program (c_0, c_1 are constructors of arity $0, 1$):

$$P = \begin{cases} f_1(x) &= \texttt{fcase}\ x\ \texttt{of}\ \{c_0 \to g, c_1(y) \to f_1(y)\} \\ f_2(x,y) &= f_1(y) \\ g &= \texttt{let}\ x = x\ \texttt{in}\ x \end{cases}$$

Remember that "$\texttt{let}\ x = x$" defines a logic variable x so that g evaluates to a new logic variable. The type environments are computed by the iterations:

$$\begin{aligned}
Inf_0(P) &= \{c_0 :: \underline{G}/\emptyset, c_1 :: \Pi^1/\emptyset, f_1 :: \underline{G}/\emptyset, f_2 :: \underline{G}/\emptyset, g :: \underline{G}/\emptyset\} \\
Inf_1(P) &= \{c_0 :: \underline{G}/\emptyset, c_1 :: \Pi^1/\emptyset, f_1 :: \underline{G}/\{\underline{guess}(\Pi^1)\}, f_2 :: \underline{G}/\emptyset, g :: \underline{A}/\emptyset\} \\
Inf_2(P) &= \{c_0 :: \underline{G}/\emptyset, c_1 :: \Pi^1/\emptyset, f_1 :: \underline{A}/\{\underline{guess}(\Pi^1)\}, f_2 :: \underline{G}/\{\underline{guess}(\Pi^2)\}, g :: \underline{A}/\emptyset\} \\
Inf_3(P) &= \{c_0 :: \underline{G}/\emptyset, c_1 :: \Pi^1/\emptyset, f_1 :: \underline{A}/\{\underline{guess}(\Pi^1)\}, f_2 :: \underline{A}/\{\underline{guess}(\Pi^2)\}, g :: \underline{A}/\emptyset\} \\
Inf(P) &= Inf_3(P)
\end{aligned}$$

The inference shows that a call to f_2 might produce a non-ground result but causes nondeterministic steps only if the second argument is non-ground.

To complete this section about the type inference, we show that its computed results correctly and completely correspond to the results of the type/effect analysis of Section 2.

Theorem 2 (Correctness of the inference). *Let P be a flat program, $E(P) = \{s :: \overline{\tau_n} \xrightarrow{\varphi} \tau \mid s$ is n-ary, $s :: \nu/\varepsilon \in Inf(P), \overline{\tau_n} \xrightarrow{\varphi} \tau \in types(n, \nu/\varepsilon)\}$, and E be a correct environment for P. Then:*

Soundness: *$E(P)$ is a correct environment in the sense of Definition 6.*
Completeness: *If $\mathcal{A} \in E$ is a type annotation, then $E(P)$ contains a type annotation $\mathcal{A}'$ with $\mathcal{A}' \leq \mathcal{A}$ (cf. Definition 4).*

4 Conclusions

We have presented a program analysis to approximate the nondeterminism behavior of functional logic programs. Unlike existing nondeterminism analyses for logic languages, we have considered a language with a demand-driven evaluation strategy. Such a strategy has good properties for executing (e.g., optimal evaluation [2]) and writing programs (e.g., more modularity due to the use of infinite data structures [17]), it considerably complicates the analysis of programs since, in contrast to logic languages with an eager evaluation model (e.g., Prolog, Mercury, HAL), there is no direct correspondence between the program structure and its evaluation order. Therefore, we have abstracted the information about the run-time behavior of the program in form of a non-standard type and effect system. The program analysis is then an iterative type inference process based on a compact structure to represent sets of types and effects.

For future work we plan to improve the preliminary implementation of the type inference and apply it to larger application programs. Furthermore, we are working on a compilation for the functional logic language Curry [9,15] into the functional language Haskell [23]. This compilation should take great advantage of the presented analysis.

References

1. E. Albert, M. Hanus, F. Huch, J. Oliver, and G. Vidal. Operational Semantics for Declarative Multi-Paradigm Languages. *JSC*, Vol. 40, No. 1, pp. 795–829, 2005.
2. S. Antoy, R. Echahed, and M. Hanus. A Needed Narrowing Strategy. *Journal of the ACM*, Vol. 47, No. 4, pp. 776–822, 2000.
3. S. Antoy and M. Hanus. Functional Logic Design Patterns. In *Proc. of FLOPS 2002*, pp. 67–87. Springer LNCS 2441, 2002.
4. B. Braßel, M. Hanus, and F. Huch. Encapsulating Non-Determinism in Functional Logic Computations. *Journal of Functional and Logic Programming*, No. 6, 2004.
5. A. Cortesi, G. File, and W. Winsborough. *Prop* revisited: Propositional Formula as Abstract Domain for Groundness Analysis. In *Proc. IEEE Symposium on Logic in Computer Science*, pp. 322–327, 1991.
6. S.K. Debray and D.S. Warren. Detection and Optimization of Functional Computations in Prolog. In *Proc. Third International Conference on Logic Programming (London)*, pp. 490–504. Springer LNCS 225, 1986.
7. B. Demoen et al. Herbrand constraint solving in HAL. In *Proc. of ICLP'99*, pp. 260–274. MIT Press, 1999.

8. M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming*, Vol. 19&20, pp. 583–628, 1994.
9. M. Hanus. A Unified Computation Model for Functional and Logic Programming. *Proc. 24th ACM Symp. on Principles of Programming Languages*, pp. 80–93, 1997.
10. M. Hanus. A Functional Logic Programming Approach to Graphical User Interfaces. In *International Workshop on Practical Aspects of Declarative Languages (PADL'00)*, pp. 47–62. Springer LNCS 1753, 2000.
11. M. Hanus. High-Level Server Side Web Scripting in Curry. In *Proc. of the Third International Symposium on Practical Aspects of Declarative Languages (PADL'01)*, pp. 76–92. Springer LNCS 1990, 2001.
12. M. Hanus and C. Prehofer. Higher-Order Narrowing with Definitional Trees. *Journal of Functional Programming*, Vol. 9, No. 1, pp. 33–75, 1999.
13. M. Hanus and F. Steiner. Controlling Search in Declarative Programs. In *Principles of Declarative Programming (Proc. Joint International Symposium PLILP/ALP'98)*, pp. 374–390. Springer LNCS 1490, 1998.
14. M. Hanus and F. Steiner. Type-based Nondeterminism Checking in Functional Logic Programs. In *Proc. of PPDP 2000*, pp. 202–213. ACM Press, 2000.
15. M. Hanus (ed.). Curry: An Integrated Functional Logic Language (Vers. 0.8). Available at `http://www.informatik.uni-kiel.de/~curry`, 2003.
16. F. Henderson, T. Somogyi, Z. Conway. Determinism analysis in the Mercury compiler. In *Proc. 19th Australian Computer Science Conference*, pp. 337–346, 1996.
17. J. Hughes. Why Functional Programming Matters. In D.A. Turner, editor, *Research Topics in Functional Programming*, pp. 17–42. Addison Wesley, 1990.
18. J. Launchbury. A Natural Semantics for Lazy Evaluation. In *Proc. of POPL'93*, pp. 144–154. ACM Press, 1993.
19. F. Liu. Towards lazy evaluation, sharing and non-determinism in resolution based functional logic languages. In *Proc. of FPCA'93*, pp. 201–209. ACM Press, 1993.
20. R. Loogen and S. Winkler. Dynamic Detection of Determinism in Functional Logic Languages. *Theoretical Computer Science 142*, pp. 59–87, 1995.
21. F. López-Fraguas and J. Sánchez-Hernández. TOY: A Multiparadigm Declarative System. In *Proc. of RTA'99*, pp. 244–247. Springer LNCS 1631, 1999.
22. F.Nielson, H.R.Nielson, C.Hankin. *Principles of Program Analysis*. Springer, 1999.
23. S. Peyton Jones, editor. *Haskell 98 Language and Libraries—The Revised Report*. Cambridge University Press, 2003.
24. P. Van Roy, B. Demoen, and Y.D. Willems. Improving the execution speed of compiled Prolog with modes, clause selection, and determinism. In *Proc. of the TAPSOFT '87*, pp. 111–125. Springer LNCS 250, 1987.
25. P. Wadler. How to Declare an Imperative. *ACM Computing Surveys*, Vol. 29, No. 3, pp. 240–263, 1997.

Techniques for Scaling Up Analyses Based on Pre-interpretations*

John P. Gallagher**, Kim S. Henriksen, and Gourinath Banda

Computer Science, Building 42.1, P.O. Box 260,
Roskilde University, DK-4000, Denmark
{jpg, kimsh, gnbanda}@ruc.dk

Abstract. Any finite tree automaton (or regular type) can be used to construct an abstract interpretation of a logic program, by first determinising and completing the automaton to get a pre-interpretation of the language of the program. This has been shown to be a flexible and practical approach to building a variety of analyses, both generic (such as mode analysis) and program-specific (with respect to a type describing some particular property of interest). Previous work demonstrated the approach using pre-interpretations over small domains. In this paper we present techniques that allow the method to be applied to more complex pre-interpretations and larger programs. There are two main techniques presented: the first is a novel algorithm for determinising finite tree automata, yielding a compact "product" form of the transitions of the result automaton, that is often orders of magnitude smaller than an explicit representation of the automaton. Secondly, it is shown how this form (which is a representation of a pre-interpretation) can then be input directly to a BDD-based analyser of Datalog programs. We demonstrate through experiments that much more complex analyses become feasible.

1 Introduction and Motivation

In this paper we investigate the question of the scalability of logic program analyses based on pre-interpretations. This question is raised since pre-interpretations provide a general and flexible approach to specifying a variety of analyses, combining modes, types and other program specific properties. However, previous experiments [1,2,3] were limited to domains containing not more than four or five elements; furthermore for larger programs (especially those with predicates of high arity) experiments were restricted to even smaller domains. We discuss the reasons for this below. It was mentioned in earlier work that efficient representations of relations would be crucial to scalability.

An arbitrary regular type can be used to construct a pre-interpretation [2]. This contributes to the ease of specifying pre-interpretations, but adds another dimension to the complexity problem. A pre-interpretation can be orders of

* Work partially supported by European Framework 5 Project ASAP (IST-2001-38059).

** Partially supported by the CONTROL project funded by the Danish Natural Science Research Council, and the IT University of Copenhagen.

M. Gabbrielli and G. Gupta (Eds.): ICLP 2005, LNCS 3668, pp. 280–296, 2005.

magnitude larger than the regular type from which it is derived, when represented naively. This raises the question of whether the flexibility of this approach can be exploited for more complex analyses.

To summarise our conclusions, we show promising results for both aspects of the scalability problem. We give a new *determinisation algorithm* for finite tree automata, which returns the determinised automaton in a compact form. We then show how this compact form can be used directly in a *BDD*-based analyser for Datalog programs.

2 Preliminaries

In this section we recall those concepts pertaining to pre-interpretations and finite tree automata that concern us. We assume familiarity with standard logical concepts such as *interpretation*, *satisfiable* and *model* [4].

Pre-interpretations. Let P be a definite program and Σ the signature of its underlying language L; Σ is a set of ranked function and predicate symbols. A *pre-interpretation* of L consists of

1. a non-empty domain of interpretation D;
2. an assignment of an n-ary function $D^n \to D$ to each n-ary function symbol in Σ ($n \geq 0$).

A *domain atom* for a pre-interpretation J having domain D is an expression $p(d_1, \ldots, d_n)$ where p is an n-ary predicate symbol in Σ and $d_1, \ldots, d_n$ are elements of D. Let B_P^J be the set of domain atoms for pre-interpretation J and the signature of the language associated with program P. A model of a definite program P, based on pre-interpretation J, is some subset of B_P^J which satisfies P. A definite program has a least model for a given pre-interpretation. In particular, the least model for the Herbrand pre-interpretation is the usual declarative semantics of definite logic programs. The least model for a pre-interpretation J can be computed as the least fixpoint of a function $T_P^J : B_P^J \to B_P^J$.

A pre-interpretation assigns a domain element to each ground term in $\mathsf{Term}(\Sigma)$, its *denotation*. Let $f_J : D^n \to D$ be the function assigned to the n-ary function f by the pre-interpretation J. Then the denotation $\mathsf{Den}_J(t)$ of a term $t \in \mathsf{Term}(\Sigma)$ is defined as $\mathsf{Den}_J(f(t_1, \ldots, t_n)) = f_J(\mathsf{Den}(t_1), \ldots, \mathsf{Den}(t_n))$ ($n > 0$), and $\mathsf{Den}_J(t) = f_J(t)$ if t is a 0-ary function symbol.

Pre-interpretations and term properties. The domain elements of a pre-interpretation J define term properties. A domain element d corresponds to a term property p where for all terms t, $p(t)$ holds if and only if $\mathsf{Den}_J(t) = d$. Note that the properties defined by the elements of a pre-interpretation are disjoint and complete; each term has exactly one of the properties since it denotes exactly one domain element.

Abstract interpretation based on pre-interpretations. Static analysis of definite logic programs using (finite) pre-interpretations was set out in [5,6] and [1]. Earlier related ideas, not mentioning pre-interpretations, were developed by Corsini

et al. [7] and by Codish and Demoen [8]. We briefly summarise the approach; analysis consists of the construction of a pre-interpretation capturing some property of interest, followed by the computation of the least model with respect to that pre-interpretation. The implementation method is in three stages; let P be a program and J a pre-interpretation.

1. Represent J as a set of facts of the form $f(d_1, \ldots, d_n) \rightarrow d$, such that $f_J(d_1, \ldots, d_n) = d$, where f_J is the function assigned to f.
2. Transform P, introducing equalities until every non-variable appears in the left-hand-side of an equality, and no nested functions occur.
3. Convert to an *abstract domain program* by interpreting the introduced equalities as the pre-interpretation function. In practice this just means replacing the $=$ symbol by $\rightarrow$.

The stages of transformation are illustrated for a single clause below.

```
rev([X|Xs],Zs) :- rev(Xs,Ys), append(Ys,[X],Zs).
rev(U,Zs) :- rev(Xs,Ys), append(Ys,V,Zs), [X|Xs]=U, [X|W]=V, []=W.
rev(U,Zs) :- rev(Xs,Ys), append(Ys,V,Zs), [X|Xs]→U, [X|W]→V, []→W.
```

To continue the example, the pre-interpretation capturing the properties *ground* (`g`) and *non-ground* (`ng`) is given by the following facts defining the relation $\rightarrow$: $\{[] \rightarrow \mathtt{g}, [\mathtt{g}|\mathtt{g}] \rightarrow \mathtt{g}, [\mathtt{g}|\mathtt{ng}] \rightarrow \mathtt{ng}, [\mathtt{ng}|\mathtt{g}] \rightarrow \mathtt{ng}, [\mathtt{ng}|\mathtt{ng}] \rightarrow \mathtt{ng}\}$. The least model of the transformed program together with the facts defining the pre-interpretation is then computed.

Analysis based on pre-interpretations can be presented as an abstract interpretation [9]. The abstract domain $2^{B^J_P}$ is relational, capturing dependencies among arguments of a predicate, and condensing, implying that a bottom-up, goal-independent analysis yields results that lose no information with respect to goal-dependent analyses.

From regular types to pre-interpretations. In [2] it was shown that any term properties expressible by regular types could be transformed to a pre-interpretation, even if the properties were not disjoint. The process of constructing the pre-interpretation uses the algorithm for determinising a finite tree automaton. This means that one can start from term properties and build a pre-interpretation capturing those properties. More specifically, the pre-interpretation captures a set of disjoint properties derived from the original properties; for instance, given properties p_1, p_2 and p_3, the pre-interpretation might for example have domain elements corresponding to $p_1 \wedge p_3$, $p_2 \wedge p_3$, where these two properties were disjoint.

Determinisation of Finite Tree Automata. A *finite tree automaton* (FTA) is defined as a quadruple $\langle Q, Q_f, \Sigma, \Delta \rangle$, where Q is a finite set called *states*, $Q_f \subseteq Q$ is called the set of accepting (or final) states, Σ is a set of ranked function symbols and Δ is a set of *transitions*. Each element of Δ is of the form $f(q_1, \ldots, q_n) \rightarrow q$, where $f \in \Sigma$ and $q, q_1, \ldots, q_n \in Q$. We write f^n to indicate that function symbol f has arity n. and we often write the term $f^0()$ as f and call f a *constant*. Term_Σ is the set of *ground terms* (or *trees*) constructed from Σ in the usual way.

An FTA can be "run" on terms in Term_Σ; the details are omitted here, except to say that a successful run of a term and an FTA is one in which the term is *accepted* by one of the final states the FTA. Implicitly, a tree automaton R defines a set of terms, that is, a tree language, denoted $L(R)$, as the set of all terms that it accepts.

As far as expressiveness is concerned we can limit our attention to FTAs in which the set of transitions Δ contains no two transitions with the same left-hand-side. These are called *bottom-up deterministic* finite tree automata (DFTAs). For every FTA R there exists a bottom-up deterministic FTA R' such that $L(R) = L(R')$. A term can be accepted by at most one final state of a DFTA.

An automaton $R = \langle Q, Q_f, \Sigma, \Delta \rangle$ is called *complete* if for all n-ary functions $f \in \Sigma$ and states $q_1, \ldots, q_n \in Q$, $f(q_1, \ldots, q_n) \to q \in \Delta$.

A complete DFTA in which every state is an accepting state partitions the set of terms into disjoint subsets, one for each state, since every term is accepted by exactly one state.

Example 1. Let $\Sigma = \{[]^0, [\_|\_]^2, 0^0\}$, and let $Q = \{list, listlist, any\}$. We define the set Δ_{any}, for a given Σ, to be the following set of transitions.

$$\{f(\overbrace{any, \ldots, any}^{n \text{ times}}) \to any \mid f^n \in \Sigma\}$$

Let $Q_f = \{list, listlist\}$, $\Delta = \{[] \to list, [any|list] \to list, [] \to listlist, [list|listlist] \to listlist\} \cup \Delta_{any}$. The state (or regular type) *list* accepts terms in the set of lists of any terms, while the state *listlist* accepts terms in the set of lists whose elements are themselves lists. Clearly *listlist* is contained in *list*, which is contained in *any*.

The automaton is not bottom-up deterministic; a determinisation algorithm yields the DFTA $\langle Q', Q'_f, \Sigma, \Delta' \rangle$, where $Q' = \{q_1, q_2, q_3\}$, $Q'_f = \{q_1, q_2\}$ and $\Delta' = \{[] \to q_1, [q_1|q_1] \to q_1, [q_2|q_1] \to q_2, [q_1|q_2] \to q_2, [q_2|q_2] \to q_2, [q_3|q_2] \to q_2, [q_3|q_1] \to q_2, [q_2|q_3] \to q_3, [q_1|q_3] \to q_3, [q_3|q_3] \to q_3, 0 \to q_3\}$. q_1 corresponds

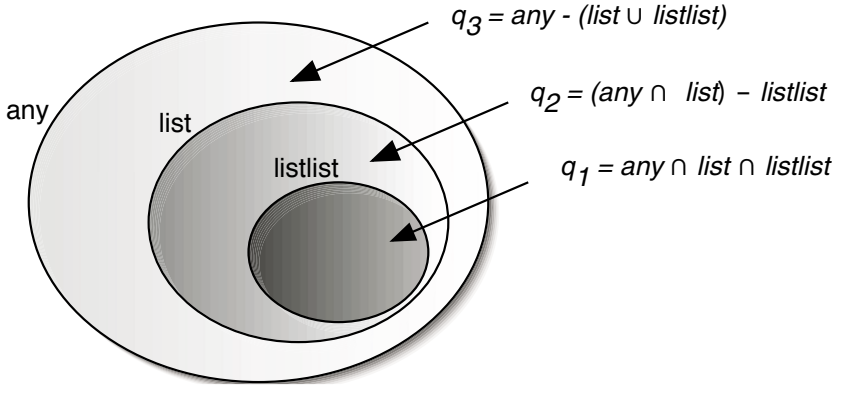


Fig. 1. The original types and the disjoint types from Example 1

to the set $any \cap list \cap listlist$, $q2$ to the set $(list \cap any) - listlist$, and q_3 to $any - (list \cup listlist)$. Thus q_1, q_2 and q_3 accept disjoint sets of terms. The original regular types and the disjoint types are shown in Figure 1. This automaton is also complete. In fact, any DFTA obtained from an FTA whose transitions include Δ_{any} (for the appropriate signature) is complete. □

3 An Algorithm for Determinisation

Product representation sets of transitions. The determinisation algorithm described below generates an automaton whose transitions are represented in product form, as described below, which is a more compact form and leads to a correspondingly more efficient determinisation algorithm. The main difference from the textbook algorithm is the form of the output, and in the explicit use of indices for efficient searching of the set of transitions. A *product transition* is of the form $f(Q_1, \ldots, Q_n) \to q$ where $Q_1, \ldots, Q_n$ are sets of states and q is a state. This product transition denotes the set of transitions $\{f(q_1, \ldots, q_n) \to q \mid q_1 \in Q_1, \ldots, q_n \in Q_n\}$. Thus $\prod_{i=1\ldots n} |Q_i|$ transitions are represented by a single product transition.

Example 2. The transitions of the DFTA generated in Example 1 can be represented in product transition form as follows. $\Delta' = \{[] \to q_1, 0 \to q_3, [\{q_1, q_2, q_3\}|\{q_3\}] \to q_3, [\{q_1, q_2\}|\{q_2\}] \to q_2, [\{q_1, q_2, q_3\}|\{q_1\}] \to q_1, [\{q_3\}|\{q_2\}] \to q_2\}$. Thus 4 product transitions replace the 9 transitions for $[\_|\_]^2$ shown in Example 1. There are other equivalent sets of product transitions, for example, $\Delta' = \{0 \to q_3, [\{q_1, q_2\}|\{q_3\}] \to q_3, [\{q_3\}|\{q_3\}] \to q_3, [\{q_1, q_2\}|\{q_2\}] \to q_2, [\{q_3\}|\{q_2\}] \to q_2 [\{q_1, q_2\}|\{q_1\}] \to q_1, [\{q_3\}|\{q_1\}] \to q_1, [] \to q_1\}$.

3.1 A Determinisation Algorithm Generating Product Form

The algorithm developed in this section was based initially on the classical textbook algorithm [10]. It differs firstly by introducing an index structure to avoid traversing the complete set of transitions in each iteration of the algorithm, and secondly by noting that the algorithm only needs to compute explicitly the set of states of the determinised automaton. The set of transitions can be represented implicitly in the algorithm and generated later if required from the determinised states and the implicit form. However, in our approach the implicit form is close to product transition form and we will use this form directly. Hence, we never need to compute the full set of transitions and this is a major saving of computation. Let $\langle Q, Q_f, \Sigma, \Delta \rangle$ be an FTA. Consider the following functions.

- $\mathsf{qmap}_\Delta : (Q \times \Sigma \times \mathcal{N}) \to 2^\Delta$
  $\mathsf{qmap}_\Delta(q, f^n, j) = \{f(q_1, \ldots, q_n) \to q_0 \in \Delta \mid q = q_j\}$ for $1 \leq j \leq n$.
- $\mathsf{Qmap}_\Delta : (2^Q \times \Sigma \times \mathcal{N}) \to 2^\Delta$
  $\mathsf{Qmap}_\Delta(Q', f^n, j) = \bigcup\{\mathsf{qmap}_\Delta(q, f^n, j) \mid q \in Q'\}$.
- $\mathsf{states}_\Delta : 2^\Delta \to 2^Q$
  $\mathsf{states}_\Delta(\Delta') = \{q_0 \mid f(q_1, \ldots, q_n) \to q_0 \in \Delta'\}$.

- $\mathsf{fmap}_\Delta : \Sigma \times \mathcal{N} \times 2^{2^Q} \to 2^{2^\Delta}$
  $\mathsf{fmap}_\Delta(f^n, j, \mathcal{D}) = \{\mathsf{Qmap}_\Delta(Q', f^n, j) \mid Q' \in \mathcal{D}\} \setminus \emptyset$, for $1 \leq j \leq n$.
- $\mathsf{C} : 2^Q$
  $\mathsf{C} = \{\{q \mid f^0 \to q \in \Delta\} \mid f^0 \in \Sigma\}$
- $\mathsf{F}_\Delta : 2^{2^Q} \to 2^{2^Q}$
  $\mathsf{F}_\Delta(\mathcal{D}') = \mathsf{C} \cup \{\mathsf{states}_\Delta(\Delta_1 \cap \cdots \cap \Delta_n) \mid f^n \in \Sigma, \Delta_1 \in \mathsf{fmap}_\Delta(f^n, 1, \mathcal{D}'), \ldots, \Delta_n \in \mathsf{fmap}_\Delta(f^n, n, \mathcal{D}')\} \setminus \emptyset$

The subscript Δ is omitted in the context of some fixed FTA. The function qmap_Δ is an index on Δ, recording the set of transitions that contain a given state q at a given position in its left-hand-side. Qmap_Δ is the same index lifted to sets of states.

The algorithm finds the least set $\mathcal{D} \in 2^{2^Q}$ such that $\mathcal{D} = \mathsf{F}(\mathcal{D})$. The set $\mathcal{D}$ is computed by a fixpoint iteration as follows.

initialise $i = 0$; $\mathcal{D}_0 = \emptyset$
repeat $\mathcal{D}_{i+1} = \mathsf{F}(\mathcal{D}_i)$; $i = i + 1$ **until** $\mathcal{D}_i = \mathcal{D}_{i-1}$

It can be shown that the sequence $\mathcal{D}_0, \mathcal{D}_1, \mathcal{D}_2, \ldots$ increases monotonically (with respect to the subset ordering on 2^{2^Q}) and clearly there exists some i such that $\mathcal{D}_{i-1} = \mathcal{D}_i$ since Q is finite.

Example 3. Consider the following regular types (FTA transitions), in which each transition has been labelled to identify it conveniently. We have $Q = \{any, list\}$ and $\Delta = \{t_1, \ldots, t_5\}$.

$$t_1 : [] \to list \qquad t_3 : [] \to any$$
$$t_2 : [any|list] \to list \qquad t_4 : [any|any] \to any$$
$$t_5 : f(any, any) \to any$$

The qmap function is as follows:

$\mathsf{qmap}(list, cons, 1) = \emptyset$ $\mathsf{qmap}(list, cons, 2) = \{t_2\}$ $\mathsf{qmap}(list, f, 1) = \emptyset$
$\mathsf{qmap}(list, f, 2) = \emptyset$ $\mathsf{qmap}(any, cons, 1) = \{t_2, t_4\}$ $\mathsf{qmap}(any, cons, 2) = \{t_4\}$
$\mathsf{qmap}(any, f, 1) = \{t_5\}$ $\mathsf{qmap}(any, f, 2) = \{t_5\}$

There is only one constant, [], and $\mathsf{C} = \{\{any, list\}\}$. Initialise $\mathcal{D}_0 = \emptyset$; the iterations of the algorithm produce the following values.

1. $\mathcal{D}_1 = \{\{any, list\}\}$
2. $\mathcal{D}_2 = \{\{any, list\}, \{any\}\}$
3. $\mathcal{D}_2 = \mathcal{D}_3$ □

The determinised automaton can be constructed from the fixpoint $\mathcal{D}$ and Qmap. The set of states $\mathcal{Q}$ is $\mathcal{D}$ itself. The set of final states $\mathcal{Q}_f$ is $\{Q' \mid Q' \in \mathcal{Q}, Q' \cap Q_f \neq \emptyset\}$. The set of transitions is

$$\{f(Q_1, \ldots, Q_n) \to \mathsf{states}(\mathsf{Qmap}(Q_1, f, 1) \cap \cdots \cap \mathsf{Qmap}(Q_n, f, n)) \mid f^n \in \Sigma, Q_1 \in \mathcal{Q}, \ldots, Q_n \in \mathcal{Q}\}$$

The transition for each constant f^0 is $f^0 \rightarrow \{q \mid f^0 \rightarrow q \in \Delta\}$. Continuing Example 3, we obtain

$[] \rightarrow \{any, list\}$
$[\{any\}|\{any, list\}] \rightarrow \mathsf{states}(\mathsf{Qmap}(\{any\}, cons, 1) \cap \mathsf{Qmap}(\{any, list\}, cons, 2))$
$\quad \rightarrow \mathsf{states}(\{t_2, t_4\} \cap \{t_2, t_4\})$
$\quad \rightarrow \{any, list\}$
$[\{any\} \mid \{any\}] \rightarrow \mathsf{states}(\mathsf{Qmap}(\{any\}, cons, 1) \cap \mathsf{Qmap}(\{any\}, cons, 2))$
$\quad \rightarrow \mathsf{states}(\{t_2, t_4\} \cap \{t_4\})$
$\quad \rightarrow \{any\}$
$f(\{any\}, \{any\}) \rightarrow \mathsf{states}(\mathsf{Qmap}(\{any\}, f, 1) \cap \mathsf{Qmap}(\{any\}, f, 2))$
$\quad \rightarrow \mathsf{states}(\{t_5\} \cap \{t_5\})$
$\quad \rightarrow \{any\}$
and so on.

There are nine transitions in this small example. As we will see we can also obtain a more compact representation as a set of product transitions.

Implementation of the Algorithm. The function qmap is computed once at the start of the algorithm in time $O(|\Delta|)$, and it can be stored as a hash-table, which allows the computation of $\mathsf{qmap}(q, f, j)$ in constant time. The value of $\mathsf{Qmap}(Q', f, j)$ can thus be computed in $O(|Q|)$. $\mathsf{states}(\Delta')$ can be computed in $O(|\Delta|)$ after construction of a suitable index to the transitions.

The function fmap is maintained as a table, called ftable. As described above, the algorithm computes a sequence $\emptyset, \mathsf{F}(\emptyset), \mathsf{F}^2(\emptyset), \ldots$, where $\mathcal{D}_i = \mathsf{F}^i(\emptyset)$. Let $\mathcal{D}_i$ and $\mathcal{D}_{i+1}$ be successive values of the sequence. At the $i + 1^{th}$ stage of the algorithm values of the form $\mathsf{fmap}(f, j, \mathcal{D}_{i+1})$ are computed for each f and j. We use the property that $\mathsf{fmap}(f, j, \mathcal{D}_{i+1}) = \mathsf{fmap}(f, j, \mathcal{D}_i) \cup \mathsf{fmap}(f, j, (\mathcal{D}_{i+1} \setminus \mathcal{D}_i))$. The table entry $\mathsf{ftable}(f^n, j)$ holds the values of $\mathsf{fmap}(f, j, \mathcal{D}_i)$ on the i^{th} iteration of the algorithm. Hence on the next iteration only the new values of fmap, that is, $\mathsf{fmap}(f, j, (\mathcal{D}_{i+1} \setminus \mathcal{D}_i))$, need to be added to $\mathsf{ftable}(f, j)$.

The evaluation of the function F can also be optimised taking into account the newly computed values of fmap. Assuming the existence of the ftable, define a function F' as

$$\mathsf{F}'(\mathcal{D}_{new}) = \{\mathsf{states}(\Delta_1 \cap \cdots \cap \Delta_n) \mid \begin{array}{l} f^n \in \Sigma, \\ \Delta_1 \in \mathsf{ftable}(f^n, 1), \\ \ldots, \\ \Delta_j \in \mathsf{fmap}(f^n, j, \mathcal{D}_{new}), \\ \ldots, \\ \Delta_n \in \mathsf{ftable}(f^n, n), \\ 1 \leq j \leq n\} \setminus \emptyset \end{array}$$

Thus for each tuple $\Delta_1, \ldots, \Delta_n$, at least one component of the tuple must be chosen from $\mathcal{D}_{new}$, ensuring that each tuple $\Delta_1, \ldots, \Delta_n$ needs to be considered only once for each f^n during the execution of the algorithm. After performing these optimisations the algorithm can be summarised as follows.

$\mathcal{D} = \mathsf{C};\ \mathcal{D}_{new} = \mathcal{D};$
for $f^n \in \Sigma$
  for $j = 1$ **to** n
    $\mathsf{ftable}(f^n, j) = \emptyset$
  endfor
endfor
repeat
  $\mathcal{D}_{old} = \mathcal{D};$
  for $f^n \in \Sigma$
   for $j = 1$ **to** n
      $\mathsf{ftable}(f^n, j) = \mathsf{ftable}(f^n, j) \cup \mathsf{fmap}(f^n, j, \mathcal{D}_{new})$
   endfor
  endfor
  $\mathcal{D} = \mathcal{D} \cup \mathsf{F}'(\mathcal{D}_{new});$
  $\mathcal{D}_{new} = \mathcal{D} \setminus \mathcal{D}_{old}$
until $\mathcal{D}_{new} = \emptyset$

Complexity. For each $f^n \in \Sigma$, the computation time is dominated by the number of tuples $Q_1, \ldots, Q_n$ that have to be considered during the computation of F. This is $\prod_{i=1\ldots n} |\mathsf{fmap}(f, i, \mathcal{D})|$. The maximum size of $|\mathsf{fmap}(f, i, \mathcal{D})|$ is the number of possible right-hand-sides in the determinised transitions for a f, say k_f. This is 2^Q in the worst case, but in practice it is often much smaller. The number of tuples is in fact closely related to the set of product transitions generated as follows. As can be seen from Figure 2, this is usually much smaller than the set of transitions in the DFTA.

Let $f^n \in \Sigma$ and let $\mathcal{D}$ be the set of sets of states computed as the fixpoint in the algorithm. Then the set of product transitions for f^n $(n > 0)$ is

$$\{f(\mathsf{fmap}^{-1}(\Delta_1, f^n, 1), \ldots, \mathsf{fmap}^{-1}(\Delta_n, f^n, n)) \to \mathsf{states}(\Delta_1 \cap \cdots \cap \Delta_n) \mid \Delta_1 \in \mathsf{fmap}(f^n, 1, \mathcal{D}), \ldots, \Delta_n \in \mathsf{fmap}(f^n, n, \mathcal{D})\}$$

where $\mathsf{fmap}^{-1}(\Delta', f^n, i) = \{Q' \mid \mathsf{Qmap}(Q', f^n, i) = \Delta', Q' \in \mathcal{D}\}$. $\mathsf{fmap}^{-1}(\Delta', f^n, i)$ can be computed and stored during the evaluation of $\mathsf{fmap}(f^n, i, \mathcal{D})$. For the example above, the final values of the fmap function are

$$\mathsf{fmap}(cons, 1, \mathcal{D}) = \{\{t_2, t_4\}\} \qquad \mathsf{fmap}(cons, 2, \mathcal{D}) = \{\{t_2, t_4\}, \{t_4\}\}$$
$$\mathsf{fmap}(f, 1, \mathcal{D}) = \{\{t_5\}\} \qquad \mathsf{fmap}(f, 2, \mathcal{D}) = \{\{t_5\}\}$$

The values of fmap^{-1} are:

$$\mathsf{fmap}^{-1}(\{t_2, t_4\}, cons, 1) = \{\{any, list\}, \{list\}\} \quad \mathsf{fmap}^{-1}(\{t_2, t_4\}, cons, 2) = \{\{any, list\}\}$$
$$\mathsf{fmap}^{-1}(\{t_4\}, cons, 2) = \{\{any\}\} \quad \mathsf{fmap}^{-1}(\{t_5\}, f, 1) = \{\{any, list\}, \{list\}\}$$

From these values we obtain the following product transitions (including the transition for the constant []).

$$[\{\{any\}, \{any, list\}\} | \{\{any, list\}\}] \to \{any, list\}$$
$$[\{\{any\}, \{any, list\}\} | \{\{any\}\}] \to \{any\}$$
$$f(\{\{any\}, \{any, list\}\}, \{\{any\}, \{any, list\}\}) \to \{any\}$$
$$[] \to \{any, list\}$$

The two states $\{any\}$ and $\{any, list\}$ denote non-lists and lists respectively. The determinised automaton is a pre-interpretation over this two-element domain. In general, a state $\{q_1, \ldots, q_k\}$ in a determinised automaton represents those terms in the intersection of the original states $q_1, \ldots, q_k$, and not in any other state. Thus $\{any\}$ always stands for terms that are of type any that are not of some other type.

4 Computing Models of Datalog Programs

The essential task in performing an analysis using pre-interpretations is to compute the minimal Herbrand model of a (definite) Datalog program [11]. A definite Datalog program is a set of Horn clauses containing no function symbols with arity greater than zero. The Herbrand models of such programs are finite. In the abstract domain programs defined in Section 2, a pre-interpretation was represented by a set of facts (unit clauses) of the form $(f(d_1, \ldots, d_n) \rightarrow d) \leftarrow true$. Although there are function symbols occuring in such facts, we can easily represent the facts using a separate predicate for each function symbol; say pre_f is the relation corresponding to f. Then all atoms of form $f(d_1, \ldots, d_n) \rightarrow d$ would be represented as the function-free atom $pre_f(d_1, \ldots, d_n, d)$ instead. Since function symbols occur nowhere else in the abstract domain program, we are left with a Datalog program.

Efficient techniques for computing Datalog models have been studied extensively in research on deductive database systems [11], and indeed, many techniques (especially algorithms for computing joins) from the field of relational databases are also relevant. In the logic programming context, facts containing variables are also allowed; tabulation and subsumption techniques have been applied in a Datalog model evaluation system for program analysis [12].

The analysis method based on pre-interpretations is of course independent of which technique is used for computing the model of the Datalog program. Having transformed the analysis task to that of computing a Datalog program model, we are free to choose the best method available. We do not give a detailed account of the various techniques here, but remark only that current techniques allow very large Datalog programs to be handled [13].

Our previous experiments [2] used a Prolog implementation, which though it incorporated many optimisations such as computing SCCs and the semi-naive strategy, did not scale well in certain dimensions. In particular, programs containing predicates of high arity (such as the Aquarius compiler benchmark, which has some predicates with arity greater than 25) could not be analysed for domains with size greater than three. The number of possible tuples of arity n with a domain of size m is m^n, so this limitation is almost certain to apply to any tuple-based representation. It was pointed out in [2] that improved representations of finite relations was a key factor in scaling up to larger domains.

Computing Datalog models using BDDs. Our current work uses the BDD-based solver `bddbddb` developed by Whaley [14]. This tool computes the model of a Datalog program, and provides facilities for querying Datalog programs. It

is written in Java and can link to established BDD libraries using the Java Native Interface (JNI). Our experiments were conducted using `bddbddb` linked to the BuDDy package [15]. We wrote a front end to translate our abstract logic programs and pre-interpretations into the form required by `bddbddb`.

The possibility of using Boolean functions to represent finite relations[1] was exploited in model-checking [16]. Assume that a relation over D^n is to be represented, where D contains m elements. Then we code the m elements using $k = \lceil log_2(m) \rceil$ bits and introduce $n.k$ Boolean variables $x_{1,1}, \ldots, x_{1,k}, x_{2,1}, \ldots, x_{n,1}, \ldots, x_{n,k}$. A tuple in the relation is then a conjunction $x_{1,1} = b_{1,1} \wedge \ldots \wedge, x_{n,k} = b_{n,k}$ where $b_{i,1} \cdots b_{i,k}$ is the encoding of the i^{th} component of the tuple. A finite relation is thus a disjunction of such conjunctions. BDDs allow very large relations, translated in this way into Boolean formulas, to be represented compactly (though variable ordering is critical, and there are some relations that admit no compact representation).

In a BDD-based evaluation of a Datalog program, the solution of each predicate is thus represented as a Boolean formula (in BDD form) and the relational operations required to compute the model can be translated into operations on BDDs. For example, if we are solving the conjunction $p(A, B), q(B, C)$ we take the Boolean formulas representing the current solutions of p and q, say F_p and F_q and build a new BDD representing the formula $F_p \wedge F_q \wedge x_{2,1} = y_{1,1} \wedge \ldots \wedge x_{2,k} = y_{1,k}$ where $x_{1,1}, \ldots, x_{1,k}, x_{2,1}, \ldots, x_{2,k}$ and $y_{1,1}, \ldots, y_{1,k}, y_{2,1}, \ldots, y_{2,k}$ are the Boolean variables representing the respective arguments of p and q.

Representing and manipulating Boolean formulas is a very active research field and there are other techniques besides BDDs that are competitive. In logic-program analyses, multi-headed clauses have demonstrated good performance when compared to BDDs, for example [17].

5 From Product Representations to Datalog

The determinisation algorithm in Section 3 returns transitions in product form. Though this saves computation, we still need to represent the product form as a Datalog program, so that we can exploit techniques such as BDD-based evaluation of the model.

Consider a product transition $f(\{a, b\}, \{c, d, e\}) \rightarrow q$. As before, we can introduce a predicate for each function to replace the arrow relation, obtaining $pre_f(\{a, b\}, \{c, d, e\}, q)$. To represent this as a clause we could write the following.

$$pre_f(X, Y, q) \leftarrow member(X, [a, b]), member(Y, [c, d, e]).$$

To convert to Datalog we need only introduce a specialised *member* predicate for each set that occurs as an argument in a product transition. In the above case we obtain:

[1] We are indebted to Peter Stuckey for drawing our attention to the fact that BDD-based approaches could be applied to arbitrary Datalog programs.

$$pre_f(X,Y,q) \leftarrow m_1(X), m_2(Y). \quad m_2(c) \leftarrow true.$$
$$m_1(a) \leftarrow true. \quad m_2(d) \leftarrow true.$$
$$m_1(b) \leftarrow true. \quad m_2(e) \leftarrow true.$$

As a further optimisation, if some product transition has for some argument a set containing all of the determinised states, we may simply replace that argument by an anonymous variable (a "don't care" argument). Also, singleton sets $\{q\}$ can be replaced by q instead of introducing a deterministic *member* call. For the transitions produced from Example 3, the set of determinised states was $\{\{any\}, \{any, list\}\}$. (We can write these states as constants q_1, q_2 respectively). The product transitions are

$$[\{q_1, q_2\}|\{q_2\}] \rightarrow q_2$$
$$[\{q_1, q_2\}|\{q_1\}] \rightarrow q_1$$
$$f(\{q_1, q_2\}, \{q_1, q_2\}) \rightarrow q_1$$
$$[] \rightarrow q_2$$

The Datalog program is thus

$$pre_{cons}(\_, q_2, q_2) \leftarrow true.$$
$$pre_{cons}(\_, q_1, q_1) \leftarrow true.$$
$$pre_f(\_, \_, q_1) \leftarrow true.$$
$$pre_{nil}(q_2) \leftarrow true.$$

Introduction of don't care arguments is certainly important for tuple-based representations but probably not for BDD-based approaches. In any case it does no harm in the latter case.

6 Experiments

We now summarise the analysis procedure. The procedure takes two inputs: a program P to be analysed and a set of regular type definitions R expressing term properties of interest. The procedure then follows these steps.

1. Augment the types with a standard type *any* over the signature of the program, and determinise yielding transitions R_d in product form.
2. Transform P to an abstract domain program P_a (using flattened predicates pre_f to denote the pre-interpretation of function f as explained in the previous section).
3. Transform R_d to a suitable Datalog representation R_{dat}, again using the pre_f representation, together with the specialised *member* predicates for the product transitions (and optionally introduce don't care arguments).
4. Transform $P_a \cup R_{dat}$ to the syntax required by `bddbddb` and compute its least model.

`bddbddb` provides facilities for querying specific predicates rather than computing the whole model, which may be more useful in certain applications, especially

those where we are simply interested in whether a predicate has any solution at all. However, we simply computed the whole model in the experiments. All the experiments were carried out using a machine equipped with a Pentium IV 2.8GHz processor with Hyper Threading enabled, 512MB RAM, with Linux installed. The determinisation algorithm is implemented in Ciao-Prolog, and the `bddbddb` tool is implemented in Java, with a JNI interface to the BuDDy BDD package, which is implemented in C.

Experiments on determinisation. Figure 2 shows a few experimental results just illustrating the effect of the determinisation algorithm. For each input FTA, the table shows the number of states Q and transitions Δ, followed by the number of states in the output DFTA, Q_d. Three measures of the set of transitions are shown. First the total number of transitions Δ_d, followed by the size of the set of product transitions generated by the algorithm $\Delta_\prod$. Thirdly we show the size of another set of product transitions Δ_{dc} that is generated by locating "don't care" arguments. The final column is the time in seconds to compute the product form Δ_{dc} (which is almost identical to the time to compute $\Delta_\prod$).

The most important observation is the significant reduction in size of $\Delta_\prod$ and Δ_{dc} compared to Δ_d. Note also that the set of states in the DFTA can actually be less than the set of states in the input FTA, as in the `dnf` example. This is because, as is typical in automatically generated FTAs, there are many equivalent states in the input, and this redundancy is removed in the DFTA.

	FTA		DFTA				
Name	Q	Δ	Q_d	Δ_d	$\Delta_\prod$	Δ_{dc}	secs
`chr`	21	64	57	118837	242	86	0.09
`dnf`	105	803	46	6567	168	141	0.57
`mat1`	6	10	6	39	8	8	0.01
`mat2`	3	8	3	12	9	7	0.01
`ring`	5	12	5	30	14	11	0.01
`pic`	8	270	8	4989	274	280	0.15

Fig. 2. Determinisation results

The input FTAs are `chr`, a set of regular types for analysing a CHR transition system; `dnf`, the regular type inferred automatically by the abstract interpretation over DFTAs described in [18]; `mat1`, a set of types for an off-line binding time analysis of a matrix transposition program; `mat2`, the regular types from Example 1 augmented by two extra function symbols; `ring`, the regular types describing states in the token-ring analysis problem [2]; and `pic`, a set of regular types expressing properties of a PIC processor emulator. We were unable to determinise the `chr`, `dnf` or `pic` examples using an available toolkit for handling tree automata, Timbuk[2] [19].

[2] The author of Timbuk confirmed that the implementation followed the textbook algorithm and no special effort to optimise it had been made.

Experiments on model computation. We now describe some experiments with analyses that use both determinisation and model computation.

We performed three general kinds of experiment. Firstly, we analysed two larger standard benchmarks using general-purpose domains including groundness, and list types. The results shown are for the Aquarius compiler and the Chat parser. One domain (`dom1`) has four elements (ground-lists, non-ground-lists, ground-non-lists, non-ground-non-lists) but this is more complex than the two-element domains (such as POS [20]) reported previously for analysis of these programs [21,17]. Another (`dom2`) includes a fifth element (variable) as well as the ones mentioned above (and therefore the binary encoding requires three bits per element). This caused a much more complex analysis for the Aquarius compiler (see Figure 3). Secondly, we took an example of automatically generated regular types from a program (`dnf`) using the type inference system described in [18] and re-analysed the program with a pre-interpretation based on those types `dnftype`. The point of doing this is that further precision can be gained, since the type inference analysis is not relational, but derives an independent type for each variable of the program. Using analysis with a pre-interpretation, dependencies among the arguments can be derived. Thirdly, we analysed a program using some program-specific types `colours` written by the user. The purpose is to check that required properties hold. In our case the program analysed is a Coloured Petri Net emulator, implemented in Prolog, for the task scheduler of an operating system kernel for real-time embedded systems [22]. The user types describe the types (colours) of the tokens in the net.

None of these examples could be handled by our previous analyser employing a tuple-representation of the least model. In the case of the larger pre-interpretations, the results show that the product representation allows pre-interpretations that would have enormous numbers of transitions if written out in full.

For each experiment in Figure 3, the following information is reported: the name of the program (Prog) and the number of clauses it contains (Clauses); the name of the pre-interpretation (Domain); the number of states in the original FTA (Q); the number of transitions in the FTA (Δ); the number of states in the determinised automaton (Q_d); the number of transitions in the full determinised automaton (Δ_d), which is shown in brackets as this is not actually computed - it is just shown to underline the impracticality of computing this; the number of product transitions ($\Delta_\prod$); and finally the time taken, split into the pre-processing time and the actual model computation. The pre-processing is shown separately since `bddbddb` can be considerably optimised in this respect[3], and should in fact be linear in the size of the program.

Variable ordering can be critical to the effectiveness of BDDs. In the experiments we used the default textual order of variables occurring in the program, and this was satisfactory except for the `aquarius` program with `dom2`, which was unable to complete in one hour. `bddbddb` has various heuristics for selecting variable order but we have not yet succeeded in exploiting these effectively. An-

[3] Personal communication from the developer of `bddbddb`.

Prog	Clauses	Domain	Q	Δ	Q_d	(Δ_d)	$\Delta_\prod$	Pre-Process	Analyse
`aquarius`	4192	`dom1`	3	1933	4	(1130118)	1951	68.8s	3.0s
`aquarius`	4192	`dom2`	4	1934	5	(10054302)	1951	70.0s	1h+
`chat`	515	`dom1`	3	655	4	(20067)	433	1.6s	0.2s
`chat`	515	`dom2`	4	656	5	(86803)	433	1.6s	2.8s
`dnf`	33	`dnftype`	105	803	46	(6567)	141	0.5s	58.0s
`petri`	66	`colours`	16	65	16	(268436271)	89	1.2s	1.5s

Fig. 3. Experimental results for Model Computation

other aspect of the variable ordering issue is the binary encoding of the domain elements. For instance,[4] given domain elements $\{a, b, c, d\}$, with the encoding $a = 00, b = 01, c = 10, d = 11$, the relation $\{p(b), p(c)\}$ requires two BDD nodes, while the relation $\{p(a), p(b)\}$ can be represented with a single node. The situation is reversed with the encoding $a = 10, b = 00, c = 01, d = 11$.

7 Related Work and Conclusions

Analysis based on pre-interpretations was introduced some time ago [5,6,1]. Earlier related approaches were put forward [7,8]. Scalability of these approaches was not really investigated, except in the case of Boolean domains, where BDDs [21] and other representations [17] were applied.

Tree automata are increasingly being applied in static analysis e.g. [23,24,25,26,18,19]. It is well known that an arbitrary finite tree automaton (FTA) can be transformed to an equivalent bottom-up deterministic tree automaton (DFTA). Many important operations and properties of tree automata are stated in terms of DFTAs [10]. However, the transformation to deterministic form can result in an explosion of states and transitions, and so some previous attempts to use DFTAs directly in static analysis reported problems with scalability [25,27]. The possibility of using a product representation does not seem to have been investigated before, though other means of compressing tree automata have been studied [28].

Dawson *et al.* [12] described an approach to program analysis (for various target languages) using logic programs to express semantic properties. Computation in a Datalog program is fundamental to the approach. Their implementation uses optimisations such as tabling and subsumption, but presumably relies on a tuple-based representation of the model and hence scalability for large relations must be an issue. Whaley *et al.* [14] obtained very promising results, with more evidence of scalability, again using Datalog programs to represent properties, but using BDDs to represent relations. Iwaihara *et al.* [29] presented two different approaches for using BDDs to compute models of Datalog programs, including the one used in `bddbddb`. In future work we plan to compare other binary encodings of relations.

[4] This example was provided by one of the anonymous referees.

Conclusions. We have described two techniques for handling larger pre-interpretations and applying them to analyse larger programs. Firstly, we presented a novel determinisation algorithm for finite tree automata, which yields a compact representation of the result. This makes it possible to build pre-interpretations from regular types, that are much more complex than those described previously [2]. Secondly, we showed how analysis based on pre-interpretations can be computed using BDD-based methods (or any other technique able to compute models of Datalog programs). Such methods have proven their scalability in other domains, especially model-checking, and there is a reasonable hope of achieving greater scalability for logic program analysis using these techniques.

Much work is required, especially in investigating strategies for improving BDD-based computations, particularly variable orderings, but also strategies for solving clause bodies, where the order of solution of body atoms, and the early elimination of local variables, can have a significant effect.

Acknowledgements

We wish to thank Peter Stuckey for suggesting the use of BDDs for computing models of Datalog programs, and for other related discussions. John Whaley provided great assistance with the `bddbddb` tool. We also thank the partners in the ASAP project for discussions and feedback on related topics. An abstract presenting the determinisation algorithm was presented at the NSAD Workshop in Paris, January 2005, and useful comments were received from Laurent Mauborgne and other attendees at the workshop. The ICLP referees gave valuable suggestions for improving the paper.

References

1. Gallagher, J.P., Boulanger, D., Sağlam, H.: Practical model-based static analysis for definite logic programs. In Lloyd, J.W., ed.: Proc. of International Logic Programming Symposium, MIT Press (1995) 351–365
2. Gallagher, J.P., Henriksen, K.S.: Abstract domains based on regular types. In Lifschitz, V., Demoen, B., eds.: Proceedings of the International Conference on Logic Programming (ICLP'2004). Volume 3132 of Springer-Verlag Lecture Notes in Computer Science. (2004) 27–42
3. Craig, S., Gallagher, J.P., Leuschel, M., Henriksen, K.S.: Fully automatic binding time analysis for Prolog. In Etalle, S., ed.: Pre-Proceedings, 14th International Workshop on Logic-Based Program Synthesis and Transformation, LOPSTR 2004, Verona, August 2004. (2004) 61–70
4. Lloyd, J.: Foundations of Logic Programming: 2nd Edition. Springer-Verlag (1987)
5. Boulanger, D., Bruynooghe, M., Denecker, M.: Abstracting *s*-semantics using a model-theoretic approach. In Hermenegildo, M., Penjam, J., eds.: Proc. 6$^{\text{th}}$ International Symposium on Programming Language Implementation and Logic Programming, PLILP'94. Volume 844 of Springer-Verlag Lecture Notes in Computer Science. (1994) 432–446

6. Boulanger, D., Bruynooghe, M.: A systematic construction of abstract domains. In Le Charlier, B., ed.: Proc. First International Static Analysis Symposium, SAS'94. Volume 864 of Springer-Verlag Lecture Notes in Computer Science. (1994) 61–77
7. Corsini, M.M., Musumbu, K., Rauzy, A., Le Charlier, B.: Efficient bottom-up abstract interpretation of prolog by means of constraint solving over symbolic finite domains. In Bruynooghe, M., Penjam, J., eds.: Programming Language Implementation and Logic Programming, 5th International Symposium, PLILP'93. Volume 714 of Springer-Verlag Lecture Notes in Computer Science. (1994) 75 – 91
8. Codish, M., Demoen, B.: Analysing logic programs using "Prop"-ositional logic programs and a magic wand. In Miller, D., ed.: Proceedings of the 1993 International Symposium on Logic Programming, Vancouver, MIT Press (1993)
9. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proceedings of the 4th ACM Symposium on Principles of Programming Languages, Los Angeles. (1977) 238–252
10. Comon, H., Dauchet, M., Gilleron, R., Jacquemard, F., Lugiez, D., Tison, S., Tommasi, M.: Tree Automata Techniques and Applications. http://www.grappa.univ-lille3.fr/tata (1999)
11. Ullman, J.: Principles of Knowledge and Database Systems; Volume 1. Computer Science Press (1988)
12. Dawson, S., Ramakrishnan, C.R., Warren, D.S.: Practical program analysis using general purpose logic programming systemsa case study. In: Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation. (May 1996) 17–126
13. Whaley, J., Lam, M.S.: Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In Pugh, W., Chambers, C., eds.: PLDI, ACM (2004) 131–144
14. Whaley, J., Unkel, C., Lam, M.S.: A bdd-based deductive database for program analysis (2004) `http://bddbddb.sourceforge.net/`.
15. Lind-Nielsen, J.: BuDDy, a binary decision diagram package (2004) `http://sourceforge.net/projects/buddy`.
16. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press (2000)
17. Howe, J.M., King, A.: Positive Boolean Functions as Multiheaded Clauses. In Codognet, P., ed.: International Conference on Logic Programming. Volume 2237 of LNCS. (2001) 120–134
18. Gallagher, J.P., Puebla, G.: Abstract Interpretation over Non-Deterministic Finite Tree Automata for Set-Based Analysis of Logic Programs. In: Fourth International Symposium on Practical Aspects of Declarative Languages (PADL'02). LNCS (2002)
19. Genet, T., Tong, V.V.T.: Reachability analysis of term rewriting systems with Timbuk. In Nieuwenhuis, R., Voronkov, A., eds.: LPAR. Volume 2250 of Lecture Notes in Computer Science., Springer (2001) 695–706
20. Marriott, K., Søndergaard, H.: Bottom-up abstract interpretation of logic programs. In: Proceedings of the Fifth International Conference and Symposium on Logic Programming, Washington. (1988)
21. Schachte, P.: Precise and Efficient Static Analysis of Logic Programs. PhD thesis, Dept. of Computer Science, The University of Melbourne, Australia (1999)
22. Banda, G.: Scalable real-time kernel for small embedded systems. Master's thesis, Southern Univ. of Denmark, Sønderborg (2003)

23. Charatonik, W., Podelski, A.: Set-based analysis of reactive infinite-state systems. In Steffen, B., ed.: Proc. of TACAS'98, Tools and Algorithms for Construction and Analysis of Systems, 4th International Conference, TACAS '98. Volume 1384 of Springer-Verlag Lecture Notes in Computer Science. (1998)
24. Goubault-Larrecq, J.: A method for automatic cryptographic protocol verification. In Rolim, J.D.P., ed.: 15 IPDPS 2000 Workshops, Cancun, Mexico, May 1-5, 2000, Proceedings. Volume 1800 of Springer-Verlag Lecture Notes in Computer Science., Springer (2000) 977–984
25. Monniaux, D.: Abstracting cryptographic protocols with tree automata. Sci. Comput. Program. **47(2-3)** (2003) 177–202
26. Comon, H., Kozen, D., Seidl, H., Vardi, M.: Applications of Tree Automata in Rewriting, Logic and Programming. Schloß Dagstuhl Seminar 9743, `http://www.informatik.uni-trier.de/~seidl/Trees.html` (October 20-24, 1997)
27. Heintze, N.: Using bottom-up tree automaton to solve definite set constraints. Unpublished. Presentation at Schloß Dagstuhl Seminar 9743, `http://www.informatik.uni-trier.de/~seidl/Trees.html` (1997)
28. Börstler, J., Möncke, U., Wilhelm, R.: Table compression for tree automata. ACM Trans. Program. Lang. Syst. **13** (1991) 295–314
29. Iwaihara, M., Inoue, Y.: Bottom-up evaluation of logic programs using binary decision diagrams. In Yu, P.S., Chen, A.L.P., eds.: ICDE, IEEE Computer Society (1995) 467–474

Deductive Multi-valued Model Checking

Ajay Mallya

Department of Computer Science,
University of Texas at Dallas
axm011500@utdallas.edu

Abstract. Model checking is a widely used technique for verifying complex concurrent systems. The models used in classical model checking methods are assumed to be complete and consistent. However, a recent body of work has shown that this is not always the case, and multi-valued logics have been proposed to represent such models, spawning an extension of classical model checking, known as, multi-valued model checking. In this paper, we define a multi-valued set based semantics for the multi-valued modal μ-calculus and present a novel interpretation of logic programs to support multi-valued sets as first-class entities, that can be used as a practical deductive multi-valued model checking framework. This framework provides a semantics preserving encoding of multi-valued transition systems, and allows verification of arbitrary multi-valued modal μ-calculus properties. A prototype implementation of this framework has also been realized.

1 Introduction

Model checking [8] is a widely used technique for the verification of complex concurrent systems such as digital circuits, communication protocols and more recently software systems. It involves verifying whether a model M of a system has a property ϕ, i.e. $M \models \phi$. If the model does not satisfy the property, the model checker then produces a *counterexample* for the checked property, which is a run of the system that causes the property to be falsified. The property ϕ is formally represented, usually as a temporal logic formula. A number of temporal logics have been used to specify properties, almost all of which are fragments of the modal μ-calculus [20]. The verification of the temporal logic formula ϕ over M is performed automatically. The complexity of model checking depends on the transition system M used to model the system and on the property ϕ that is to be verified.

The models used for representing concurrent systems in classical model checking are assumed to be complete and consistent. Therefore, two-valued transition systems and the two-valued modal μ-calculus are sufficient for verifying properties of such systems. However, a recent body of work shows that this assumption does not always hold and there are problems where the models may be incomplete or inconsistent. Incomplete models arise when information about the system is not known or has been abstracted away [6,9].

M. Gabbrielli and G. Gupta (Eds.): ICLP 2005, LNCS 3668, pp. 297–310, 2005.

Typically, at system design time, even though some implementation specific details are still unknown, it is still beneficial to verify the design to ensure that design errors are not propagated into the implementation. Also when dealing with systems that have very large state spaces, a common technique is to select a part of the state space hoping to find errors there [2]. Since the selection is usually done on an ad-hoc basis, it is essential to have a modeling representation that takes the partiality of the state space into consideration. Models may contain inconsistency because they combine conflicting points of view or contain components that have been developed independently by different designers [14].

Multi-valued temporal logics are an extension of their classical two-valued counterparts to verify properties of systems that are represented by incomplete and/or inconsistent models. The set of truth values is usually chosen to be a DeMorgan (or quasi-boolean) lattice, so that the operations of conjunction, disjunction and negation are well-defined.

There are two different approaches to model checking multi-valued logics: the direct approach and the reduction approach [2,19]. The reduction approach exploits truth values that form a distributive DeMorgan lattice and reduce the model checking problem to j instances of classical two-valued model checking, where j is the number of *join-irreducible* elements of the lattice of truth values and then combining the individual results to obtain the result for the multi-valued model checking problem. The advantage of this approach is that it can leverage existing two-valued model checking tools.

The direct approach involves the use of special-purpose model checking algorithms, which are extensions of the classical two-valued model checking algorithms. Techniques based on the direct approach are applicable even when the lattice is not distributive. Therefore these techniques are more general than those based on the reduction approach. For example, the rightmost lattice shown in Figure 1 is quasi-boolean, but not distributive. Consequently, reduction based techniques are unable to handle such lattices. However, direct techniques are also harder to implement and proving correctness is cumbersome.

In this paper, we present a technique for direct multi-valued model checking, using logic programming. Our main contributions are: (1) We propose a novel interpretation of logic programming predicates as multi-valued sets that allows us to represent the multi- valued semantics of the modal μ-calculus in terms of logic programs and prove the correctness of this approach (2) Since logic programs are executable, our technique gives rise to a logic based executable multi-valued model checking framework for the full modal μ-calculus, and for all DeMorgan lattices; as far as we are aware this is the first such executable framework.

2 Preliminaries

We will take the notions of logic programming [24], lattice theory [11] and denotational semantics [23].

2.1 DeMorgan Lattices

DeMorgan Lattices. A *DeMorgan lattice* is a tuple $(L, \sqsubseteq, \sqcup, \sqcap, \neg)$, such that:

- $(L, \sqsubseteq)$ is a complete lattice.
- $\sqcup$ and $\sqcap$ are the join and meet operators of $(L, \sqsubseteq)$
- The negation operator $\neg$ satisfies DeMorgan's laws:

$$\neg(a \sqcup b) = \neg a \sqcap \neg b$$
$$\neg(a \sqcap b) = \neg a \sqcup \neg b$$

  for $a, b \in L$.
- Negation is *involutive*, i.e. $\neg\neg a = a$ for $a \in L$.
- Negation is *anti-monotonic*, i.e., $a \sqsubseteq b$ iff $\neg b \sqsubseteq \neg a$, for $a, b \in L$.
  In the rest of the paper, we will use the term *lattice* to refer to DeMorgan lattices.

DeMorgan lattices exhibit horizontal symmetry, which makes it possible to define the complement operation in an unambiguous manner. Therefore, every element has a unique complement. The complement of an element is simply its image under horizontal symmetry. In addition to the join and meet operation which are defined over individual elements, the operators $\bigvee$ and $\bigwedge$ extend join and meet to subsets of L.

Some examples of DeMorgan lattices are shown in Figure 1. The lattice **2** represents the truth values of classical logic i.e., *true* and *false*. The lattice **3**, has the additional truth value M, which represents *unknown truth values*. This lattice corresponds to the truth values of Kleene's three-valued logic [17]. The lattice **3x3**, which is the product of the lattice **3** with itself, is used to denote inconsistencies in models caused by merging differing viewpoints. The element FM of **3x3** indicates that a proposition is *false* in the first viewpoint and unknown in the second viewpoint, whereas the element FT, indicates that a proposition is *false* in the first viewpoint and *true* in the second viewpoint.

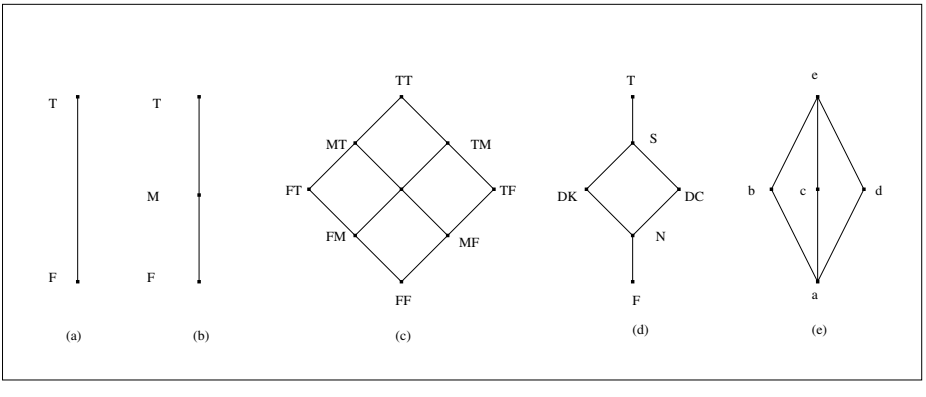


Fig. 1. Some DeMorgan lattices

2.2 Multi-valued Sets

In classical set theory, sets are defined by a boolean predicate, also known as a *characteristic function*, that maps elements of some universe to *true*, if that element belongs to the set and to *false* otherwise. For example, if a predicate P defines a set S, then we write $S = \{x|P(x)\}$. If this intensional definition of sets is extended such that the characteristic function, instead of being a boolean predicate, maps the elements of the universe to the elements of a lattice L, then we obtain the notion of *multi-valued sets*, or L-*valued* sets.

Definition 1. *Given a lattice* $\langle L, \sqcap, \sqcup, \neg \rangle$*, and a classical set,* U*, a* L*-valued set on* U *is a total function* $U \rightarrow L$*.*

Consider the multi-valued transition system shown in Figure 2, which uses the quasi-boolean lattice $2 \times 2 + 2$ shown in Figure 1. Each state variable induces a multi-valued set over the state space of the multi-valued transition system. For example, the state variable *cup* induces the multi-valued set $\{\langle idle, dc\rangle, \langle ready, t\rangle, \langle off, dc\rangle, \langle del\_coffee, t\rangle\}$.

2.3 Representing DeMorgan Lattices as Logic Programs

Consider the lattice $\mathbf{2} \times \mathbf{2} + \mathbf{2}$ shown in Fig. 1(d).The partial order relation of the lattice is represented by a predicate leq_L, which is defined as follows:

$leq_L(f, n).$ $\quad leq_L(n, dk).$ $\quad leq_L(n, dc).$
$leq_L(dk, s).$ $\quad leq_L(dc, s).$ $\quad leq_L(s, t).$
$leq_L(x, x).$
$leq_L(x, z) \leftarrow leq_L(x, y), leq_L(y, z).$

The ground facts model the immediate precedence relation of the lattice and the rules model the transitive and reflexive closure of the immediate precedence relation. The set of all ground instances of the predicate leq_L represents the partial order relation $\sqsubseteq$ of the lattice. The definition of leq_L can be used to define the predicates $join_L$ and $meet_L$, which model the $\sqcup$ and $\sqcap$ operations of the lattice. The predicates $join\_set_L$ and $meet\_set_L$ model the operations $\bigvee$ and $\bigwedge$ respectively. The definition of these predicates is obvious and we will not discuss it further.

The complement operation is defined in terms of the horizontal symmetry exhibited by DeMorgan lattices. For example, in the lattice $\mathbf{2} \times \mathbf{2} + \mathbf{2}$, we have $\neg T = F$, $\neg S = N$, $\neg DC = DC$ and $\neg DK = DK$. This is represented by the following logical predicates:

$comp_L(f, t).$ $\quad comp_L(dc, dc).$
$comp_L(n, s).$ $\quad comp_L(dk, dk).$
$comp_L(x, y) \leftarrow comp_L(y, x).$

The predicate $comp_L$ captures the horizontal symmetry of the DeMorgan lattice. We add a rule that computes the symmetric closure of the complement

relation. Unlike classical two-valued logic, where, the two truth values are complements of each other, in a multi-valued logic, the complement of a given truth value is not always obvious. The predicate $comp_L$ is therefore a definition of the complement operation of the lattice that returns the complement of a given truth value. The definition of the predicates leq_L,$join_L$,$meet_L$ and $comp_L$ provides a structure for treating DeMorgan lattices as first-class entities in our model checking procedure.

3 Multi-valued Transition Systems

Formally, a multi-valued transition system over a lattice L, with respect to a set of atomic propositions AP, is a tuple $\langle \mathcal{S}, s_0, \mathcal{R}, \theta, L, AP \rangle$, where:

1. AP is a set of atomic propositions.
2. $\mathcal{S}$ is a finite set of states.
3. $s_0 \in \mathcal{S}$ is the start state.
4. $\theta : S \rightarrow AP \rightarrow L$ is a function that maps states to a function mapping atomic propositions to elements of L.
5. $\mathcal{R} : (\mathcal{S} \times \mathcal{S}) \rightarrow L$ is a transition labelling function, mapping transitions to elements of L.
6. L is a DeMorgan lattice.

3.1 Multi-valued Transition Systems as Logic Programs

A multi-valued transition system M is transformed into a logic program P_M by representing the states and transitions of M by ground facts in P_M. For each state s of M, a predicate $theta_s$ is generated and the atom $theta_s(p, t)$ denotes that the value of the atomic proposition p at state s is the truth value t. For every outgoing multi-valued transition from state s, a predicate $trans_s$ is generated, and the atom $trans_s(s', t)$ denotes that the transition from state s to s' has truth value t.

We use an example multi-valued model checking problem adapted from [19] to introduce the concepts of multi-valued model checking. Figure 2 shows a multi-valued transition system for a coffee machine. Propositional state variables and transitions of the transition system are labeled with truth values, from the lattice $\mathbf{2} \times \mathbf{2} + \mathbf{2}$, shown in Figure 2.

The lattice elements T, S, N, F, DK, DC represent truth values as indicated in Figure 2. The element DK denotes that the truth value is unknown for variables controlled by the system and DC denotes that the truth value is unknown for variables controlled by the environment.Transitions between states also have a similar notion of "truth" associated with them. Transitions labeled with F have been omitted from the figure, since these transitions are "impossible".

The initial state of the coffee machine is the state *OFF*. In this state, it does not matter whether a cup has been provided or not and the proposition *cup* is labeled *DC*. The truth value *DK* is assigned to the transition from state *OFF*

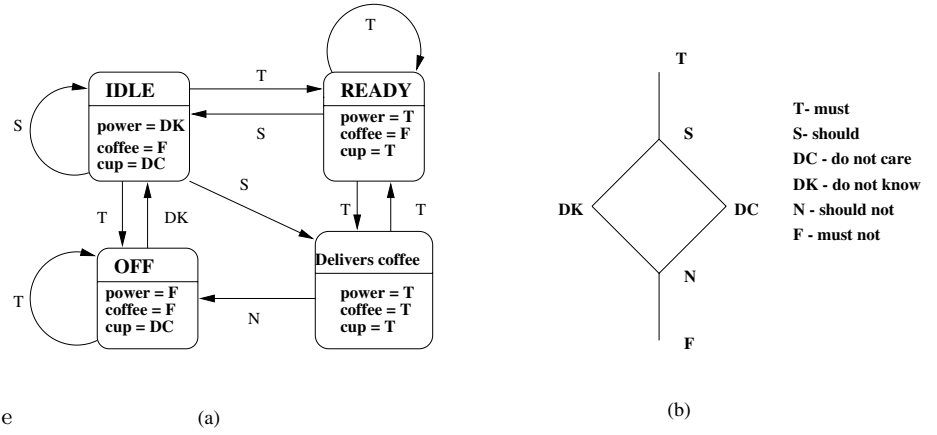


Fig. 2. Multi-valued model for a coffee machine

to *IDLE* to indicate that it is unknown whether the system can go directly from *OFF* to *IDLE* or not. Although the system itself disallows transitions from the state *Delivers coffee* to the state *OFF*, this transition is still lablelled by the truth value *N*, since it depends on the environment and cannot be explicitly prohibited by the system.

For the transition system in the coffee machine example above, we obtain the following predicates:

$$
\begin{array}{ll}
theta_{idle}(power, dk). & theta_{off}(power, f) \\
theta_{idle}(coffee, f). & theta_{off}(coffee, f). \\
theta_{idle}(cup, dc). & theta_{off}(cup, dc). \\
\ldots & \\
theta_{del\_coffee}(power, t). & \\
theta_{del\_coffee}(coffee, t). & \\
theta_{del\_coffee}(cup, t). &
\end{array}
$$

Each state s is represented by the predicate $theta_s$, which maps atomic propositions to their value in state s. Multi-valued transitions are represented by the following logical predicates:

$$
\begin{array}{ll}
trans_{idle}(idle, s). & trans_{del\_coffee}(off, n). \\
trans_{idle}(ready, t). & trans_{del\_coffee}(ready, t). \\
trans_{idle}(off, t). & \\
trans_{idle}(del\_coffee, s) & \\
\ldots &
\end{array}
$$

The definitions of the predicates $theta_s$ and $trans_s$ together comprise a transformation of the transition system M into a logic program P_M.

Definition 2. *Given a multi-valued transition system* $M = \langle \mathcal{S}, s_0, \mathcal{R}, \theta, L, AP \rangle$, *the logic program* P_M *is defined as*

$P_M = \{theta_s(p,l) \mid s \epsilon \mathcal{S}, p \epsilon AP, l \epsilon L, \theta(s)(p) = l\} \cup \{trans_s(s',t) \mid s, s' \epsilon \mathcal{S}, t \epsilon L - \{\bot\}, R(s,s') = t\}$

The predicates $theta_s$ and $trans_s$ represent the states and transitions of the multi-valued transition system respectively. The meaning of the predicate $theta_s$, written as $[\![theta_s]\!]$ is the set of all ground instances of $theta_s(P,T)$ that can be inferred from P_M, i.e.,

$$[\![theta_s]\!] = \{(p,t) | P_M \models theta_s(p,t)\}$$

Similarly,

$$[\![trans_s]\!] = \{(s',t) | P_M \models trans_s(s',t)\}$$

The logical consequence operator T_P of a logic program P is a function that when applied to a set A of ground atoms, yields the set of all atoms that are direct logical consequences of the atoms in A, i.e., for the program P_M, we have,

$T_P(A) = \{q(\widetilde{d}) | q(\widetilde{d}) \leftarrow b_1, \ldots, b_n$ is a ground instance of a clause in P, $b_1, \ldots, b_n \epsilon A\}$

where $\widetilde{d}$ refers to the tuple of values $\langle d_1, \ldots, d_n \rangle$. Since a ground fact $q(d)$ is in fact, a Horn clause of the form $q(d) \leftarrow true$, it follows that all ground facts in P are always included in the set $T_P(A)$, for all A. Therefore, the set of ground facts of the logic program P is exactly the set obtained by applying T_P to the empty set ϕ of atoms:

$$T_P(\phi) = \{q(\widetilde{d}) | q(\widetilde{d}) \text{ is a ground fact in P}\}$$

$$[\![theta_s]\!] = \{(p,t) | theta_s(p,t) \epsilon T_{P_M}(\phi)\}$$

Similarly, we have

$$[\![trans_s]\!] = \{(s',t) | trans_s(s',t) \epsilon T_{P_M}(\phi)\}$$

If the system cannot transition from state s to state s', then $\mathcal{R}(s,s') = \bot_L$, as *false* is the bottom value of L. In other words, impossible transitions are mapped onto the bottom value of the multi-valued logic used to represent the system. According to Definition 2, such transitions are not represented by facts in P_M and hence they are not logical consequences of P_M. All transitions that are not impossible are represented by facts in P_M. For each of these transitions, the set of predicates $trans_s$ associates the transition with its corresponding truth value.

The following theorem demonstrates the correctness of the above transformation of a multi-valued transition system into a classical two-valued logic program.

Theorem 1. *Given a multi-valued transition system M, and a transformation of M into a logic program, P_M, as shown above, then the tuple $\langle [\![theta_{s_0}]\!], \ldots, [\![theta_{s_{|\mathcal{S}|\ 1}}]\!] \rangle$ is the definition of the function θ of M and the tuple $\langle [\![trans_{s_0}]\!], \ldots, [\![trans_{s_{|\mathcal{S}|\ 1}}]\!] \rangle$ is the definition of the multi-valued transition relation $\mathcal{R}$ of M and these definitions are effectively computable from the definition of P_M.*

Proof. By construction, for each predicate $theta_s$, $[\![theta_s]\!]$ is the tuple of pairs $\langle (p_1, t_1), \ldots, (p_k, t_k) \rangle$, mapping atomic propositions p_i in state s to truth value t_i. But, this is exactly the definition of $\theta(s)$, the function θ, instantiated to the state s. Therefore, tuple of instantiations of θ over all $s \,\epsilon\, \mathcal{S}$, $[\![], \ldots, [\![theta_{s_{|\mathcal{S}\ 1|}}]\!] \rangle$, is the required definition of θ. By similar reasoning, it can be shown that the tuple $\langle [\![trans_{s_0}]\!], \ldots, [\![trans_{s_{|\mathcal{S}|\ 1}}]\!] \rangle$ is the definition of $\mathcal{R}$.

The definitions of θ and $\mathcal{R}$ are computable, because, for finite state systems, the transformation yields a finite number of ground facts of the form $theta_s$ and $trans_s$. $[\![theta_s]\!]$ and $[\![trans_s]\!]$ can therefore be computed by iterating T_{P_M} over the empty set ϕ of atoms, until a fixpoint is reached. Since, P_M only consists of ground facts, the least fixpoint is obtained in a single iteration and the computation always terminates. □

4 The Modal μ-Calculus

In classical model checking, μ-calculus formulas are associated with sets of states of a transition system. In the multi-valued scenario, this notion is extended to associate with each μ-calculus formula Φ, a L-valued set $\mathcal{S} \rightarrow L$, over the states, $\mathcal{S}$ of a multi-valued transition system, M, with respect to a lattice L of truth values, such that Φ is the characteristic function of the L-valued set. The truth value of the formula is then obtained by applying the characteristic function to the start state s_0 of the multi-valued transition system translation. A μ-calculus formula is a set of fixpoint equations, whose variables range over L-valued sets. Given a formula Φ, we associate with each variable Z in Φ, a predicate m_Z, such that $[\![m_Z]\!]$ is the set of tuples $\{\langle s, t \rangle \mid s \,\epsilon \mathcal{S},\, t \,\epsilon\, L\}$, that form ground instances of $m_Z(S, T)$, that satisfy the formula Φ. In other words, $[\![m_Z]\!]$ implicitly represents the L-valued set that is assigned to Z in a solution of Φ. The truth value of Φ is then given by applying the characteristic function of the L-valued set induced by Φ to the initial state s_0 of the multi-valued transition system.

The formula Φ consists of a sequence of fixpoint operators μZ or νZ, which denote least and greatest fixpoint operations respectively, applied to a set of equations of the form $Z = \psi$, where ψ is defined by the grammar

$$\psi ::= p \mid \neg p \mid Z_1 \vee Z_2 \mid Z_1 \wedge Z_2 \mid \Diamond Z \mid \Box Z.$$

where $p \,\epsilon\, AP$ and Z, Z_1 and $Z_2 \,\epsilon\, Var$, a set of logical variables, that range over μ-calculus formulas. We assume that the formula Φ is in positive normal form, i.e. negation is only applied to atomic propositions and that it is syntactically monotone in the fixpoint variables, i.e. all occurrences of fixpoint variables in Φ fall under an even number of negations.

4.1 Translating the Multi-valued Semantics of μ-Calculus Formulas into Logic Programs

For each equation in a μ-calculus formula Φ, a Horn clause is obtained as follows:

$$
\begin{array}{ll}
m_Z(s,t) \leftarrow theta_s(p,t) & \text{for } Z = p \\
m_Z(s,t) \leftarrow theta_s(p,t'), t = \neg_L t' & \text{for } Z = \neg p \\
m_Z(s,t) \leftarrow m_{Z_1}(s,t'), m_{Z_2}(s,t''), t = t' \sqcap t'' & \text{for } Z = Z_1 \wedge Z_2 \\
m_Z(s,t) \leftarrow m_{Z_1}(s,t'), m_{Z_2}(s,t''), t = t' \sqcup t'' & \text{for } Z = Z_1 \vee Z_2 \\
m_Z(s,t) \leftarrow t = \bigvee_{s\ \epsilon\mathcal{S}} \{t' \mid trans_s(s',l), m_Z\ (s',l'), t' = l \sqcap l'\} & \text{for } Z = \Diamond Z' \\
m_Z(s,t) \leftarrow t = \bigwedge_{s\ \epsilon\mathcal{S}} \{t' \mid trans_s(s',l), m_Z\ (s',l'), t' = l \sqcap l'\} & \text{for } Z = \Box Z'
\end{array}
$$

The symbols $\neg_L, \sqcap, \sqcup, \bigvee, \bigwedge$ are syntactic sugar for the predicates $comp_L$, $meet_L$, $join_L$, $join\_set_L$ and $meet\_set_L$ respectively, defined in Section 2.3.

The last two rules differ from the usual notion of a Horn clause, since, in the body of the clause, there is an implicit universal quantification over sets of tuples, i.e., relations, rather than over an individual tuple, which is an instance of a relation. This is because the transition with the greatest (resp. least) truth value has to be chosen non-deterministically for the $\Diamond$ (resp. $\Box$) operator. This is achieved by means of second order logic programming predicates (our implementation uses the *findall* predicate in XSB); however, the logical consequence operator is still defined and computable for programs with finite models. We will call the logic program that is obtained by the above transformation P_Φ.

From the programs P_M and P_Φ, we obtain the program P_μ, by taking the union of their Horn clauses:

$$P_\mu = P_M \cup P_\Phi$$

The predicate symbols $\langle m_{Z_1}, \ldots, m_{Z_n} \rangle$ of P_μ denote L-valued sets, and the semantics of P_μ is the tuple of L-valued sets $\langle S_{m_{Z_1}}, \ldots, S_{m_{Z_n}} \rangle$. Each L-valued set is of the form $\{\langle s_i, t_i \rangle \mid s_i\, \epsilon\, \mathcal{S}, t_i\, \epsilon\, L\}$.

Definition 3. *Assuming that Φ is of the form*

$$\Phi \equiv \sigma_1 Z_1 \ldots \sigma_n Z_n \{Z_i = \psi_i \mid i = 1, \ldots, n\}.$$

where each σ_i is μ or ν, the semantics $\langle [\![m_{Z_1}]\!], \ldots, [\![m_{Z_n}]\!] \rangle$ of P_μ according to the fixpoint prefix of the logical consequence operator T_{P_μ} is:

$$\langle [\![m_{Z_1}]\!], \ldots, [\![m_{Z_n}]\!] \rangle = \sigma_1 m_{Z_1} \ldots \sigma_n m_{Z_n} T_{P_\mu}(\langle m_{Z_1}, \ldots, m_{Z_n} \rangle).$$

Theorem 2. *Given a multi-valued transition system M and a property in the form of a μ-calculus formula Φ, the L-valued set induced on the states of M by Φ is the value $[\![m_{Z_1}]\!]$ of the predicate m_{Z_1} under the solution of P_μ specified by the fixpoint prefix of Φ.*

Proof . We extend the logical consequence operator of logic programs to a operator T_{P_μ} over tuples $\langle S_{m_{Z_1}}, \ldots, S_{m_{Z_n}} \rangle$ of L-valued sets of states. Formally, T_{P_μ} is defined as:

$$T_{P_\mu}(\langle S_{m_{Z_1}}, \ldots, S_{m_{Z_n}} \rangle) = (\langle S'_{m_{Z_1}}, \ldots, S'_{m_{Z_n}} \rangle)$$

where $S'_{m_{Z_j}} = \{\langle s,t\rangle \mid P_\mu \cup m_{Z_1}(S_{m_{Z_1}}) \cup \ldots m_{Z_n}(S_{m_{Z_n}}) \vdash m_{Z_j}(\langle s,t\rangle)\}$ and $m_{Z_j}(S_{m_{Z_j}})$ stands for the conjunction of ground atoms $m_{Z_j}(\langle s,t\rangle)$, where $\langle s,t\rangle \epsilon\, S_{m_{Z_j}}$. From the definition of Φ and the construction of P_μ, it follows that by applying the fixpoint prefix of Φ to $T_{P_\mu}(\langle m_{Z_1},\ldots,m_{Z_n}\rangle)$, the value of m_{Z_1} is the required L-valued set. □

4.2 Multi-valued Model Checking Temporal Properties

We consider some examples of multi-valued temporal properties with respect to the coffee machine example to demonstrate the details of multi-valued model checking. Consider the μ-calculus formula

$$\Phi = \mu Z \mu Z_1 \mu Z_2 \left\{ \begin{array}{l} Z = Z_1 \vee Z_2, \\ Z_1 = \mathit{coffee}, \\ Z_2 = \Diamond Z \end{array} \right\}$$

which states that "eventually, a state is reached where coffee is produced". This formula is transformed into the logic program

$$P_\Phi = \left\{ \begin{array}{l} m_Z(s,t) \leftarrow m_{Z_1}(s,t'), m_{Z_2}(s,t''), t = t' \sqcup t'' \\ m_{Z_1}(s,t) \leftarrow \mathit{theta}_s(\mathit{coffee},t) \\ m_{Z_2}(s,t) \leftarrow t = \bigvee_{s\ \epsilon\, \mathcal{S}} \{t' \mid \mathit{trans}_s(s',l), m'_Z(s',l'), t' = l \sqcap l'\} \end{array} \right\}$$

Assuming that we construct P_μ in the usual way, then, by Theorem 2, the L-valued set corresponding to Φ is given by the value $[\![m_Z]\!]$ of the predicate m_Z, obtained by computing the least fixpoint of T_{P_μ}. The L-valued set is:

$$\{\langle \mathit{idle},t\rangle, \langle \mathit{ready},t\rangle, \langle \mathit{off},\mathit{dk}\rangle, \langle \mathit{del\_coffee},t\rangle\}.$$

The truth value of the formula Φ is the value that the initial state is mapped to in the L-valued set, i.e dk.

Another property that is of interest is the *safety* property "coffee is always delivered in a cup", which is specified by the formula $\nu Z.(\neg \mathit{coffee} \vee \mathit{cup}) \wedge \Box Z$, whose corresponding L-valued set is

$$\{\langle \mathit{idle},t\rangle, \langle \mathit{ready},t\rangle, \langle \mathit{off},t\rangle, \langle \mathit{del\_coffee},t\rangle\}.$$

The truth value of this safety property is therefore t.

5 Implementation and Performance

We have developed an implementation of our framework and have verified several multi-valued model checking problems found in the literature. The tabled logic programming engine XSB has been used for the implementation. The system takes a description of a DeMorgan lattice, a multi-valued transition system and a μ-calculus formula and computes the multi-valued semantics of the formula.

Preliminary experiments demonstrate that performance of the tool is comparable to that of other multi-valued model checkers developed using MBTDDs, MDDs, ADDs and BDDs. These other multi-valued model checkers can only verify subsets of the modal μ-calculus. Our implementation can perform multi-valued model checking of the full modal μ-calculus and as far as we are aware, no other such implementation has been described in the liteature. Moreover, since our implementation is declarative, it is considerably easier to realize, compared to other existing techniques. At the same time experimental results demonstrate that this ease of implementation can be achieved without paying a penalty in terms of performance.

We have obtained results by running our prototype implementation on several benchmarks described in [5]. The benchmark problems are a 3-floor elevator, a 5-floor elevator, a phone system modelled using 4-valued logic and using 9-valued logic. Various example properties of these systems have also been described Their framework was implemented separately using MDDs, MBTDDs, ADDs and BDDs. The execution times of each of these implementations on the benchmarks has been taken from [5]. A comparison of these implementations against our XSB implementation is shown in Table 1. It is obvious that our implementation is comparable to theirs and in some cases performs considerably better. The reason for this is their technique uses the reduction approach, which reduces the multi-valued model checking problem to a number of classical two-value model checking sub-problems, which is equal to the number of join-irreducible elements of the lattice of truth values. Solving all these instances can significantly deteriorate the performance of the system. Our technique on the other hand is based on the direct approach and only requires us to perform the model checking once. The results for our implementation were obtained on a Sun

Table 1. Multi-valued model checking results

Model	Property	Result	MDD	MBTDD	ADD	BDD	XSB
3-floor	1.	F	0.505 s	0.423 s	1.222 s	1.007 s	0.680 s
	2.	T	0.194 s	0.125 s	1.306 s	0.23 s	0.730 s
	3.	T	0.197 s	0.122 s	1.171 s	0.233 s	0.430 s
	4.	T	0.497 s	0.591 s	1.406 s	1.196 s	1.020 s
	5.	M	0.202 s	0.125 s	0.596 s	0.23 s	0.020 s
5-floor	1.	F	19.009 s	21.251 s	30.89 s	66.713 s	8.230 s
	2.	T	2.254 s	2.407 s	6.978 s	5.156 s	3.730 s
	3.	T	2.331 s	2.406 s	5.210 s	4.929 s	4.150 s
	4.	T	14.853 s	19.394 s	79.753 s	63.978 s	7.550 s
	5.	M	1.017 s	0.862 s	4.263 s	1.725 s	0.050 s
Phone system	1.	MM	0.048 s	0.066 s	2.3 s	0.048 s	0.070 s
(9-valued)	2.	MF	0.044 s	0.074 s	2.135 s	0.052 s	0.320 s
	3.	MF	0.046 s	0.068 s	2.613 s	0.037 s	0.320 s
Phone system	1.	TT	0.031 s	0.023 s	0.024 s	0.029 s	0.040 s
(4-valued)	2.	FF	0.031 s	0.027 s	0.042 s	0.036 s	0.240 s
	3.	FF	0.031 s	0.027 s	0.041 s	0.031 s	0.240 s

Sparc machine with two 750 MHz processors and 4GB of RAM. Each individual execution, however, was carried out on a single processor with about 1 GB of available RAM.

Model checking of the full modal μ-calculus with Horn logic, requires that the Horn clauses be interpreted according to the answer set semantics [18]. The tabled resolution strategy used by XSB implements the well-founded semantics and therefore, it might be unable to verify certain kinds of μ-calculus formulas with alternating least and greatest fixed point operators. In such cases, XSB generates a *residual program*, which captures the dependencies between predicates with unknown truth values in the well-founded semantics. The stable models generator *smodels* is then invoked to compute the stable models of the residual program via the XNMR interface provided by XSB. The details of interfacing XSB with *smodels* can be found in the XSB manual [25].

6 Related Work

Several methods based on both the direct and the reduction approaches have been studied in the literature on multi-valued model checking. [2] describes a reduction from the 3-valued μ-calculus to standard model checking. [6] defines an algorithm based on the direct approach, for model checking multi-valued CTL using multi-valued extensions of BDDs. [5] presents a reduction algorithm for distributive quasi-boolean lattices. In [7] an algorithm for model checking multi-valued LTL is presented,based on a transformation to multi-valued Büchi automata, under the restriction that truth values form a totally ordered set. [19] defines a transformation from multi-valued CTL* to standard CTL*, based on the reduction approach and is limited to handling distributive quasi-boolean lattices. [16] demonstrates how multi-valued model checking can be performed for the full modal μ-calculus based on a reduction to standard model checking. Like all reduction based techniques, it can only deal with truth values that form a distributive lattice. [3] demonstrates multi-valued model checking for the full modal μ-calculus for arbitrary quasi-boolean lattices. Therefore, our technique is equivalent to theirs. However, their method involves a transformation of the denotational semantics of the multi-valued modal μ-calculus to Extended Alternating Automata [3], which is quite complex and proving correctness is also quite involved. Also, the resulting automata-theoretic semantics is not directly executable and its implementation can be quite complex. Our technique on on the other hand, uses Horn logic to directly specify the formal semantics of the multi-valued modal μ-calculus and avoids the additional step of converting the denotational semantics into an automata-theoretic formalism. Also these Horn logic definitions are executable, unlike the automata-theoretic definitions.

Horn Logic-based approaches have been widely used in classical two-valued model checking, although we are not aware of any that are targeted towards multi-valued model checking. In [4] transition systems are encoded as logic programs and the logical consequence operator computes the set of predecessor states for a given set of states. [22] uses the tabled logic programming engine

XSB to compute fixpoints of temporal logic formulas. Constraint Logic Programming based methods have also been used for the verification of discrete infinite state [12] and real-time systems [15,13,21].

7 Conclusions and Future Work

In this paper, we have formalized the multi-valued semantics of μ-calculus formulas in terms of L-valued sets. Each formula induces a L-valued set over the state space of a multi-valued transition system, and this L-valued set is the semantics of the formula. We formulate a representation for L-valued sets, using the two-valued predicates of classical logic programming, which yields an executable framework for multi-valued model checking of the modal μ-calculus. We prove the correctness of our representation and demonstrate a mechanism to compute fixpoints of multi-valued temporal logic formulas, via the formal semantics of logic programming. Even though our method encompasses the most general multi-valued model checking techniques described in the literature, it has an elegant intuitive semantics due to the declarative nature of logic programming, while comparing favorably to various existing ad-hoc techniques in terms of performance. Future work includes exploring the possibility of extending the logic programming based approach by using constraint systems, especially ones defined over lattice domains. Another direction for future research is to study the use of multi-valued techniques in the verification of real-time systems.

References

1. L. Bolc and P. Borowik *Many-Valued Logics*. Springer-Verlag, 1992.
2. G. Bruns and P. Godefroid: Generalized model checking:reasoning about partial state spaces. In *Proc. of CONCUR'00*, *LNCS*, pp. 168-182. Springer-Verlag.
3. G. Bruns and P. Godefroid: Model checking with Multi-Valued Logics In *Proc. of ICALP'04*, *LNCS*, pp. 281-293. Springer-Verlag.
4. W. Charatonik and A. Podelski: Set based analysis of reactive infinite-state systems. In *Proc. TACAS'98, LNCS*. Springer-Verlag.
5. M. Chechik,A. Gurfinkel,B. Deverereux, A. Lai, S. Easterbrook: Symbolic data structures for multi-valued model checking. CSRG Technical Report 446, University of Toronto, 2002.
6. M.Chechik, B. Devereux, S. Easterbrook and A. Gurfinkel: Multi-valued Symbolic Model Checking ACM TOSEM, ACM, 2003.
7. M. Chechik, B. Devereux, A. Gurfinkel: Model checking infinite state-space systems with fine-grained abstractions using SPIN. In *Proc. SPIN Workshop on Model-Checking software*, 2001.
8. E.M.Clarke and E.A.Emerson: Design and Synthesis of synchronization skeletons using branching time temporal logic logic. In *Logic of Programs: Workshop, Yorktown Heights,NY,May 1981, LNS 131.*
9. D. Dams, Rob Gerth,Gert Dohmen, Ronald Herrmann,Peter Kelb,Hergen Pargmann: Model checking using adaptive state and data abstraction. In *Proc. CAV'94*, *LNCS*, pp. 455-467. Springer-Verlag.

10. E. Dantsin, Thomas Eiter, Georg Gottlob, Andrei Voronkov: Complexity and Expressive Power of Logic Programming ACM Computing Surveys, ACM, 2001.
11. B.A. Davey and H.A. Priestley: *Introduction to Lattices and Order.* Cambridge University Press, 1990.
12. G. Delzanno and A. Podelski: Model Checking in CLP In *Proc. TACAS'99*, *LNCS*, pp.223-239. Springer-Verlag.
13. X. Du, C.R. Ramakrishnan, Scott A. Smolka: Tabled Resolution + Constraints: A Recipe for Model Checking Real-Time Systems In *Proc. RTSS'00*, pp. 175-184. IEEE.
14. S. Easterbrook and M. Chechik: A framework for multi-valued reasoning with over inconsistent viewpoints In *Proc. of ICSE'01*, pp. 411-420.
15. G. Gupta and E. Pontelli: A Constraint Based Approach for Specification and Verification of Real-Time Systems In *Proc. RTSS'97*, IEEE.
16. A. Gurfinkel and M. Chechik: Multi-valued model checking via classical model checking. In *Proc. of CONCUR'03*, *LNCS*, pp. 411-420. Springer-Verlag.
17. S.C. Kleene: Introduction to Metamathematics : D. Van Nostrand Company Inc., Princeton, New Jersey, 1952.
18. X. Liu, C. Ramakrishnan,S.A. Smolka: Fully Local and Efficient Evaluation of Alternating Fixed Points In *Proc. TACAS '98*, LNCS 1384,pp 5-19, Springer-Verlag, 1998.
19. B. Konikowska and W. Penczek: Reducing model checking from multi-valued CTL* to CTL*. In *Proc. of CONCUR'02,LNCS*. Springer-Verlag.
20. D.Kozen: Results on the propositional mu-calculus *Theoretical Computer Science* 27:333-354.
21. S. Mukhopadhyay and A. Podelski: Model checking for timed logic processes. In *Proc. Computational Logic'00*, *LNCS*, pp. 598-612. Springer-Verlag.
22. Y.S. Ramakrishna, C.R. Ramakrishnan, I.V. Ramakrishnan, Scott A. Smolka, Terrance Swift and David S. Warren: Efficient Model Checking Using Tabled Resolution In *Proc. CAV'97*, *LNCS*, pp 143-154. Springer-Verlag.
23. D. Schmidt: *Denotational Semantics: A Methodology for Language Development.* W.C. Brown Publishers, 1986.
24. L. Sterling and S. Shapiro: The Art of Prolog. MIT Press, 1994.
25. www.xsb.sourceforge.net

Polynomial Interpretations as a Basis for Termination Analysis of Logic Programs

Manh Thang Nguyen and Danny De Schreye

Department of Computer Science, K.U. Leuven
Celestijnenlaan 200A, B-3001, Heverlee, Belgium
{ManhThang.Nguyen, Danny.DeSchreye}@cs.kuleuven.ac.be

Abstract. This paper introduces a new technique for termination analysis of definite logic programs based on polynomial interpretations. The principle of this technique is to map each function and predicate symbol to a polynomial over some domain of natural numbers, like it has been done in proving termination of term rewriting systems. Such polynomial interpretations can be seen as a direct generalisation of the traditional techniques in termination analysis of LPs, where (semi-) linear norms and level mappings are used. Our extension generalises these to arbitrary polynomials. We extend a number of standard concepts and results on termination analysis to the context of polynomial interpretations. We propose a constraint based approach for automatically generating polynomial interpretations that satisfy termination conditions.

Keywords: Termination analysis, acceptability, polynomial interpretations.

1 Introduction

In the last 20 years, the work on termination analysis has been most active for declarative programming languages, with an emphasis on two specific paradigms: logic programming (LP) and term rewriting systems (TRSs). In both areas, the work has been extensive and successful, with many powerful techniques developed and automated tools for these techniques available. However, termination analysis research has evolved very independently for these two paradigms. This has led to two collections of techniques and tools that co-exist without a reasonable level of cross-fertilization between them, nor an acceptable understanding of the portability of these techniques from one paradigm to the other.

Independent of the paradigm, almost every termination analysis is based on a mapping from computational states to some well-founded ordered set. A main difference between LP and TRS is the class of well-founded orderings that are being considered as a basis for the termination proof. For LP, computational states are usually mapped to a well-founded order on the natural numbers. This is usually done through "norms" and "level mappings", that respectively map terms and atoms to corresponding natural numbers (see [4]). In TRSs, a considerably wider range of well-ordered sets is being considered in the literature,

M. Gabbrielli and G. Gupta (Eds.): ICLP 2005, LNCS 3668, pp. 311–325, 2005.

including polynomial interpretations, recursive and lexicographic path orders, and Knuth-Bendix orders (see [7,18]).

On a most general, methodological level, cross-fertilization of techniques could be organized using two alternative routes: a transformational approach or a direct approach. A disadvantage of the transformational approach is that it somewhat obscures the intuitions regarding the termination argument. Often, one is not merely interested in finding a proof of termination as such, but it is more helpful if the proof - or the absence of it - helps us in better understanding the behaviour of the program. Another disadvantage is that this approach is only applicable to a class of transformable programs (i.e. well-moded logic programs).

In this paper we focus on a direct approach of porting techniques, in our case from TRS to LP. Within this context, an initial result to allow porting of more general orderings to the LP setting is presented in [5]. This work provides a new termination condition for definite logic programs based on general term orders. As such, it can be used as a framework in which different orderings considered in TRSs could be ported to LP directly and be evaluated. The current paper provides a first step in this study: the use of polynomial interpretations for LP termination analysis. Using polynomial interpretations as a basis for ordering terms in TRSs was first introduced by Lankford in [12]. It is currently one of the best known and most widely used techniques in TRS termination analysis.

In this paper, we develop the approach within an LP context. We redefine and extend several known concepts and results from LP termination analysis to polynomial interpretations. We show how polynomial interpretations can be seen as a direct generalisation of currently used techniques in LP termination based on (semi-) linear norms and linear level-mappings. As one would expect, the generalisation is a move from linear polynomial functions to arbitrary polynomials, while the concepts that link the two approaches are those of the "abstract norm" and "abstract level mapping" [17]. The paper is organised as follows. In the next section, we present some preliminaries. In section 3, we introduce basic definitions of polynomial interpretations and show how this approach can be used to prove termination with some examples. In section 4, we discuss the automation of the approach. We end with a conclusion in section 5.

2 Preliminaries

2.1 Notations and Terminology

We assume familiarity with logic programming concepts and with the main results of logic programming [1,13]. In the following, L denotes the language underlying a definite logic program P. We use Var_P, $Const_P$, Fun_P and $Pred_P$ to denote the set of variables, constant, function, and predicate symbols of L. Given an atom A, $rel(A)$ denotes the predicate occurring in A. Let p, q be predicates occurring in the program P, we say that p *refers to* q if there is a clause in P such that p is in its head and q is in its body. We say that p *depends on* q if (p, q) is in the transitive, reflexive closure of the relation refer to. If p depends on q and vice versa, p and q are called *mutually recursive*, denoted by $p \simeq q$. Let

$Term_P$ and $Atom_P$ denote, respectively, the sets of all terms and atoms that can be constructed from L. The *extended Herbrand Universe* U_P^E, and the *extended Herbrand Base* B_P^E are the quotient sets of $Term_P$, and $Atom_P$ modulo the variant relation [9]. Given two expressions E and F (terms, atoms, n-tuples of terms or n-tuples of atoms), we denote by $mgu(E, F)$ their most general unifier.

In this paper, we focus our attention only on definite logic programs and SLD-derivations where the left-to-right selection rule is used. Such derivations are referred to as LD-derivations; the corresponding derivation tree as the LD-tree. We say that a query Q *LD-terminates* for a program P, if the LD-tree for $Q \cup P$ is finite (left-termination [13]).

2.2 Norms and Level Mappings

Definition 1 (norm, level mapping). *A* norm *is a mapping* $\|.\| : U_P^E \to \mathbb{N}$. *A* level-mapping *is a mapping* $|.| : B_P^E \to \mathbb{N}$.

Several examples of norms can be found in literature [2]. One of the most commonly used norms is the *list-length norm* which maps lists to their lengths and any other term to 0. Another frequently used norm is *term-size* which counts the number of function symbols in the tree representation of a term.

Definition 2 (linear norm). *[15] A norm* $\|.\|$ *is a* linear norm *if it is recursively defined by means of the following schema:*

- $\|X\| = 0$ *for any variable* X,
- $\|f(t_1, ..., t_n)\| = f_0 + \sum_{i \in I_f} f_i \|t_i\|$ *where* $f_i \in \mathbb{N}$ *and the index set* $I_f \subseteq \{1, ..., n\}$ *depend only on the n-ary function symbol* f/n, $n \geq 0$.

2.3 Conditions for Termination w.r.t. General Orderings

A *quasi-ordering on a set* S is a reflexive and transitive binary relation $\succeq$ defined on elements of S. We define the *associated equivalence relation* $\preceq\succeq$ as $s \preceq\succeq t$ if and only if $s \succeq t$ and $t \succeq s$. If neither $s \succeq t$, nor $t \succeq s$ we write $\|_{\succeq}$. To each quasi-ordering $\succeq$ on S, we can associate a *strict ordering* $\succ$ *on* S as $s \succ t$ if and only if $s \succeq t$ and it is not the case that $t \succeq s$. A strict ordering $\succ$ is called *well-founded* if there is no infinite sequence $s_0 \succ s_1 \succ ...$ with $s_i \in S$. Let T be a set such that $S \subseteq T$. A quasi-ordering $\trianglerighteq$ defined on T is called a *proper extension of* $\succeq$ if

- $t_1 \succeq t_2$ implies $t_1 \trianglerighteq t_2$ for all $t_1, t_2 \in S$.
- $t_1 \succ t_2$ implies $t_1 \triangleright t_2$ for all $t_1, t_2 \in S$, where $\triangleright$ is the strict ordering associated with $\trianglerighteq$.

We also need the following notion of a call set.

Definition 3 (call set). *Let* P *be a program and* S *be a set of atomic queries. The* call set, $Call(P, S)$, *is the set of all atoms* A, *such that a variant of* A *is the selected atom in some derivation for* (P, Q), *for some* $Q \in S$ *and under the left-to-right selection rule.*

In practice, the query set S is specified as a call pattern. The set $Call(P, S)$ can be computed by using a type inference technique (e.g.[11]).

Definition 4 (order-acceptability w.r.t. a set). *[5] Let S be a set of atomic queries and P be a program. P is* order-acceptable w.r.t. *S if there exists a well-founded ordering $\succ$ such that*

- *for any $A \in Call(P, S)$,*
- *for any clause $A' \leftarrow B_1, ..., B_n$, such that $mgu(A, A') = \theta$ exists,*
- *for any atom B_i, such that $rel(B_i) \preceq rel(A)$,*
- *for any computed answer substitution σ for $\leftarrow (B_1, ..., B_{i-1})\theta$:*

$$A \succ B_i\theta\sigma$$

The following theorem establishes the link between order-acceptability w.r.t. a set and LD-termination of a program.

Theorem 1. *[5] A program P LD-terminates under the left-to-right selection rule for any query in S if and only if P is order-acceptable w.r.t. S.*

Definition 5 (interargument relation). *Let P be a program, p/n be a predicate in P and $\succ$ be an ordering on U_P^E. An* interargument relation for *p is a relation $R_p = \{(t_1, ..., t_n) | t_i \in Term_P \wedge \varphi_p(t_1, ..., t_n)\}$, where:*

- *$\varphi_p(t_1, ..., t_n)$ is a formula in a disjunctive normal form,*
- *each conjunct in φ_p is either $s_i \succeq s_j$, $s_i \succ s_j$, $s_i \preceq \succeq s_j$ or $s_i \|_{\succeq} s_j$, where s_i, s_j are constructed from $t_1, ..., t_n$ by applying functors of P.*

R_p is a valid interargument relation for *p/n w.r.t. the ordering $\succ$ if and only if for every $p(t_1, ..., t_n) \in Atom_P$: $P \models p(t_1, ..., t_n)$ implies $p(t_1, ..., t_n) \in R_p$.*

The concept of rigidity is also generalized to general orderings.

Definition 6 (rigidity). *[5] A term or atom $A \in U_P^E \cup B_P^E$ is called* rigid w.r.t. a quasi-ordering $\succeq$ *if $\forall \sigma \in Subs$, $A \preceq \succeq A\sigma$. In this case, $\succeq$ is said to be* rigid on *A. A set of terms (or atoms) S is called* rigid w.r.t. a quasi-ordering $\succeq$ *if all its elements are rigid w.r.t. $\succeq$.*

Example 1. The list $[X|t]$ (X is a variable, t is a ground term) is rigid w.r.t. the quasi-ordering $\succeq$ imposed by the list-length norm $\|.\|_l$, i.e. $t_1 \succeq t_2$ if and only if $\|t_1\|_l \geq \|t_2\|_l$, $t_1 \succ t_2$ if and only if $\|t_1\|_l > \|t_2\|_l$. For any substitution σ, $\|[X|t]\sigma\|_l = 1 + \|t\|_l = \|[X|t]\|_l$. Therefore, $[X|t]\sigma \preceq \succeq [X|t]$. However, this list is not rigid w.r.t. the quasi-ordering $\trianglerighteq$ imposed by the term-size norm $\|.\|_t$, i.e. $t_1 \trianglerighteq t_2$ if and only if $\|t_1\|_t \geq \|t_2\|_t$, $t_1 \triangleright t_2$ if and only if $\|t_1\|_t > \|t_2\|_t$. For instance, with $\sigma_1 = \{X/a_1\}$, a_1 is a constant, $\|[X|t]\sigma_1\|_t = 1 + \|t\|_t$, while with $\sigma_2 = \{X/[a_1, a_2]\}$ a_1, a_2 are constants, $\|[X|t]\sigma_2\|_t = 3 + \|t\|_t$. That implies $[X|t]\sigma_2 \triangleright [X|t]\sigma_1$. □

The following notion of rigid order-acceptability w.r.t. a set of atoms no longer forces us to reason on $Call(P, S)$. Instead, we only need to consider the rigidity of the call set. Furthermore, the condition in this notion is fully at the clause level and the condition on computed answer substitution is replaced by one on valid interargument relations.

Definition 7 (rigid order-acceptability w.r.t. a set). *[5] Let S be a set of atomic queries and P be a program. Let $\succeq$ be a well-founded quasi-ordering on U_P^E and for each predicate p in P, let R_p be a valid interargument relation for p w.r.t. $\succeq$. P is* rigid order-acceptable w.r.t. *S if there exists a proper extension $\trianglerighteq$ of $\succeq$ on $U_P^E \cup B_P^E$, which is rigid on $Call(P,S)$ such that*

- *for any clause $H \leftarrow B_1, B_2, ..., B_n$,*
- *for any atom B_i in its body such that $rel(B_i) \trianglerighteq rel(H)$,*
- *for any substitution θ such that the arguments of the atoms in $(B_1, ..., B_{i-1})\theta$ all satisfy their associated interargument relations $R_{rel(B_1)}, ..., R_{rel(B_{i-1})}$:*

$$H\theta \triangleright B_i\theta$$

Theorem 2. *[5] If P is rigid order-acceptable w.r.t. S, then P is order-acceptable w.r.t. S.*

The stated condition of rigid order-acceptability is sufficient for acceptability, but is not necessary for it (see [5]).

3 Polynomial Interpretations in Logic Programming

Recall that our objective is to develop and discuss the basic definitions and properties of polynomial interpretations, and apply them to prove termination of a program. Here terms and atoms are mapped to polynomials, instead of natural numbers. This will allow to solve a class of problems that the traditional approach can not solve. To illustrate this point, consider the following program, *Der*, that formulates rules for computing the repeated derivative of a function in some variable u. This example was introduced in [5] (see also [8]).

Example 2 (Der).

$$d(der(u), 1).$$
$$d(der(A), 0) : -number(A).$$
$$d(der(X + Y), DX + DY) : -d(der(X), DX), d(der(Y), DY).$$
$$d(der(X * Y), X * DY + Y * DX) : -d(der(X), DX), d(der(Y), DY).$$
$$d(der(der(X)), DDX) : -d(der(X), DX), d(der(DX), DDX).$$

We are interested in proving termination of this program w.r.t. the query set $S = \{d(t_1, t_2) | t_1$ is a ground term, and t_2 is a free variable$\}$. We consider the first argument of $d/2$ as an input argument and the second as an output.

Doing this on the basis of a linear norm and level mapping is impossible. The function symbol $der/1$ expresses a non-linear relation between the input and output of the original derivative function. In particular, assume that there exists such a linear norm $\|.\|$ and level mapping $|.|$ of general forms such that: $\|u\| = 0$, $\|t_1 + t_2\| = f_0^+ + f_1^+\|t_1\| + f_2^+\|t_2\|$, $\|t_1 * t_2\| = f_0^* + f_1^*\|t_1\| + f_2^*\|t_2\|$, $\|der(t)\| = f_0^d + f_1^d\|t\|$, $|d(t_1, t_2)| = d_0 + d_1\|t_1\| + d_2\|t_2\|$, $|number(t)| = n_0 + n_1\|t\|$ where

t, t_1, t_2 are terms and f_0^+, f_1^+, f_2^+, f_0^*, f_1^*, f_2^*, f_0^d, f_1^d, d_0, d_1, d_2, n_0 and n_1 are non-negative integers. Applying the general constraint based method in [6] shows a contradiction: the system of inequalities that is set up from the acceptability condition is unsolvable. A complete proof can be found in [14]. Of course this only proves that one particular approach is unable to prove termination on the basis of linear mappings. □

3.1 Polynomial Interpretations

Let $\mathbb{N}$ be the set of all natural numbers and $A \subseteq \mathbb{N}$. We denote by $\mathbb{P}^A_{Var_P}$ the set of all polynomials in Var_P over A, with coefficients in $\mathbb{N}$. The following definition establishes an ordering on $\mathbb{P}^A_{Var_P}$.

Definition 8 (polynomial ordering). *Let P be a program and $A \subseteq \mathbb{N}$. Let $\mathbb{P}^A_{Var_P}$ be a set of all polynomials in Var_P over A. For polynomials $H, Q \in \mathbb{P}^A_{Var_P}$ let $X_1, ..., X_n$ be the variables occurring in H or Q. We define an ordering $\geq_A$ on $\mathbb{P}^A_{Var_P}$ as $H \geq_A Q$ if and only if $H - Q \geq 0$ for all instantiations $a_1, ..., a_n \in A$ of $X_1, ..., X_n$ respectively. A strict ordering $>_A$ associated with $\geq_A$ is defined as $H >_A Q$ if and only if $H - Q > 0$ for all $a_1, ..., a_n \in A$. If $H - Q = 0$ for all $a_1, ..., a_n \in A$, we write $H \leq \geq_A Q$. For any other cases, $H \|_{\geq_A} Q$.*

We usually require that A is an infinite set. Under this condition, $H \leq \geq_A Q$ if and only if the two polynomials are identical, denoted by $H \equiv Q$, i.e. all their corresponding coefficients are equal.

Example 3. Let H, Q be two polynomials in $Var_P = \{X_1, X_2, X_3\}$ over A such that: $H = 2X_1^2 + 3X_2X_3 + 5X_3$ and $Q = X_1^2 + 3X_2 + 2X_3 + 4$. We define a function $F(X_1, X_2, X_3) = H - Q = X_1^2 + 3X_2X_3 - 3X_2 + 3X_3 - 4$. Consider the following cases:

- $A = \mathbb{N} \backslash \{0\}$. For all $a_1, a_2, a_3 \in A$, $F(a_1, a_2, a_3) \geq 0$. Hence, $H \geq_A Q$.
- $A = \mathbb{N} \backslash \{0, 1\}$. For all $a_1, a_2, a_3 \in A$, $F(a_1, a_2, a_3) > 0$. Hence, $H >_A Q$.
- $A = \mathbb{N}$. For $a_1 = a_2 = a_3 = 0$, $F(a_1, a_2, a_3) = -4 < 0$. For $a_1 = a_2 = a_3 = 2$, $F(a_1, a_2, a_3) = 12 > 0$. Hence, $H \|_{\geq_A} Q$. □

Theorem 3. *Let $A \neq \emptyset$. The ordering $>_A$ on $\mathbb{P}^A_{Var_P}$ defined in definition 8 is a well-founded ordering.*

Proof. See [14].

Definition 9 (polynomial pre-interpretation).
A polynomial pre-interpretation J for a language of terms L *consists of:*

- *a set of natural numbers A, $A \subseteq \mathbb{N}$,*
- *an assignment that associates each n-ary function symbol f, $n \geq 0$, in L with a polynomial $P_f(X_{f_1}, ..., X_{f_m})$ from A^m to A, where the coefficients of the polynomial P_f/m are in $\mathbb{N}$ and the index set $I_f = \{f_1, ..., f_m\} \subseteq \{1, ..., n\}$ is determined by f/n.*

Note that each constant c in L can be considered an 0-ary function symbol and is assigned to an element c_I of A. Another issue is that the set A should be *closed under evaluating the polynomials*, i.e. for all $f \in Fun_P$ and $a_1, ..., a_n \in A$, $P_f(a_1, ..., a_n) \in A$. This extra condition is required in the definition because of the fact that terms are recursively defined from their subterms. Thus, when selecting a polynomial pre-interpretation, we not only select an appropriate polynomial associated with each function symbol but also an appropriate set A such that the above closure property is guaranteed.

Definition 10 (polynomial norm). *The* polynomial norm associated with a polynomial pre-interpretation J *is a mapping* $\|.\|_J : Term_P \rightarrow \mathbb{P}^A_{Var_P}$ *which is defined recursively as:*

- $\|X\|_J = X$ *if* X *is a variable,*
- $\|f(t_1, ..., t_n)\|_J = P_f(\|t_{f_1}\|_J, ..., \|t_{f_m}\|_J)$,

where $P_f(X_1, ..., X_m)$ *and* $I_f = \{f_1, ..., f_m\}$ *are the same as in the definition of the polynomial pre-interpretation* J.

Similarly, we define the notion of a polynomial interpretation that sets up an abstract version of each predicate symbol.

Definition 11 (polynomial interpretation). *A* polynomial interpretation I for a language L underlying a program P *consists of a polynomial pre-interpretation* J *for the language of terms defined by* L *extended by*

- *an assignment to each predicate symbol* p/n, $n \geq 0$, *in* L *of a polynomial* $P_p(X_{p_1}, ..., X_{p_m})$ *from* A^m *to* A, *where the coefficients of the polynomial* P_p/m *are in* $\mathbb{N}$ *and the index set* $I_p = \{p_1, ..., p_m\} \subseteq \{1, ..., n\}$ *is determined by* p/n.

Definition 12 (polynomial level-mapping). *The* polynomial level-mapping associated with a polynomial interpretation I *is a mapping* $|.|_I : Atom_P \rightarrow \mathbb{P}^A_{Var_P}$ *which is defined as:* $|p(t_1, ..., t_n)|_I = P_p(\|t_{p_1}\|_J, ..., \|t_{p_m}\|_J)$ *where* $P_p(X_1, ..., X_m)$ *and* $I_p = \{p_1, ..., p_m\}$ *are as in the definition of the polynomial interpretation* I.

For each term t and atom A, we denote by $P_t = \|t\|_J$ and $P_A = |A|_I$ as the polynomial interpretations of respectively t and A in I.

Example 4 (Dist). Consider the following distributive program *Dist*. This example was introduced in [5] (see also [18]):

$$dist(x, x).$$
$$dist(x * x, x * x).$$
$$dist(X + Y, U + V) : -dist(X, U), dist(Y, V). \quad (1)$$
$$dist(X * (Y + Z), T) : -dist(X * Y + X * Z, T). \quad (2)$$
$$dist((X + Y) * Z, T) : -dist(X * Z + Y * Z, T). \quad (3)$$

Let I be a polynomial interpretation that consists of a set $A \subseteq \mathbb{N}$, an assignment that associates the function symbol $*/2$ with the polynomial $P_* = X_1 * X_2$, the function symbol $+/2$ with the polynomial $P_+ = X_1 + X_2 + 1$, the constant x with a constant $c_x \in A$, and an assignment that associates the predicate symbol $dist/2$ with the polynomial $P_{dist} = X$, where the variable X corresponds to the first argument position of *dist/2*. The polynomial interpretation of the atom $A = dist(U*(X+Y), T)$ in I is: $P_A = |dist(U*(X+Y), T))|_I = \|U*(X+Y)\|_J = P_*(\|U\|_J, \|X+Y\|_J) = P_*(\|U\|_J, P_+(\|X\|_J, \|Y\|_J)) = \|U\|_J * (\|X\|_J + \|Y\|_J + 1) = U * (X + Y + 1)$. □

We define a quasi-ordering on $U_P^E \cup B_P^E$ imposed by the ordering $>_A$ on $\mathbb{P}^A_{Var_P}$ as follows:

Definition 13 (ordering on terms and atoms). *Let P be a program and I be a polynomial interpretation. We define $\succeq_I$ a quasi-ordering on U_P^E such that:*

- *$t \succ_I s$ if and only if $P_t >_A P_s$ for any $t, s \in U_P^E$,*
- *$t \preceq \succeq_I s$ if and only if $P_t \leq \geq_A P_s$ for any $t, s \in U_P^E$,*

and $\trianglerighteq_I$ a proper extension of $\succeq_I$ on $U_P^E \cup B_P^E$ such that:

- *$B \triangleright_I C$ if and only if $P_B >_A P_C$ for any $B, C \in B_P^E$,*
- *$B \trianglelefteq \trianglerighteq_I C$ if and only if $P_B \leq \geq_A P_C$ for any $B, C \in B_P^E$,*

where P_t, P_s, P_B, P_C are polynomial interpretations of t, s, B and C.

Theorem 4. *The strict orderings $\succ_I$ and $\triangleright_I$ are well-founded orderings on U_P^E and $U_P^E \cup B_P^E$ respectively.*

Integrated with definition 4 and theorem 1 we obtain:

Proposition 1. *Let P be a program and S be a set of atomic queries. If there exists a polynomial interpretation I such that*

- *for any $A \in Call(P, S)$,*
- *for any clause $A' \leftarrow B_1, ..., B_n$ in P, such that $mgu(A, A') = \theta$ exists,*
- *for any atom B_i, such that $rel(B_i) \trianglerighteq rel(A)$,*
- *for any computed answer substitution σ for $\leftarrow (B_1, ..., B_{i-1})\theta$:*

$$P_A >_A P_{B_i\theta\sigma}$$

where P_A denotes the polynomial interpretation of the atom A,

then P left-terminates w.r.t. S.

Example 5. Reconsider example 4. We prove termination of the program with the following set of queries $S = \{dist(t_1, t_2) | t_1$ is a ground term and t_2 is a free variable$\}$. We choose the polynomial interpretation I of example 4 except that $A = \mathbb{N} \setminus \{0, 1\}$. Then, $\forall t \in Term_P$, $\|t\|_J >_A 1$. Observe that the set $Call(P, S) = S$. Suppose $A = dist(t, s)$ is a selected atom in *Call(P,S)*. There are 3 cases to consider: clauses (1), (2) and (3). We present only the last one:

- $A = dist((t_1+t_2)*t_3, s)$ (t_1, t_2, t_3 are ground terms) and clause (3) is selected. There exists a substitution θ such that $\theta = mgu(A, dist((X_1+Y_1)*Z_1, T_1))$. That implies $X_1\theta = t_1, Y_1\theta = t_2, Z_1\theta = t_3$. Therefore, $|dist((t_1+t_2)*t_3, s)|_I = \|(t_1+t_2)*t_3\|_J = \|t_1+t_2\|_J * \|t_3\|_J = \|t_1\|_J * \|t_3\|_J + \|t_2\|_J * \|t_3\|_J + \|t_3\|_J >_A \|t_1\|_J * \|t_3\|_J + \|t_2\|_J * \|t_3\|_J + 1 = \|t_1 * t_3 + t_2 * t_3\|_J = |dist(X_1 * Z_1 + Y_1 * Z_1, T_1)\theta|_I$.

With a similar verification for clauses (1) and (2), P is order-acceptable w.r.t. S and P terminates on S. □

Next, we study rigidity of a call set w.r.t. a polynomial interpretation and use it to verify rigid order acceptability.

3.2 Rigidity

First we present the classical notion of strictly monotone polynomials. This class of polynomials is discussed in [18]. Next we study the rigidity of a set of (terms) atoms w.r.t. a polynomial (pre-)interpretation that maps (terms) atoms to polynomials.

Definition 14 (strictly monotone polynomials). *Let $A \subseteq \mathbb{N}$. A polynomial $P(X_1, ..., X_n)$, $n > 0$, over A is called* strictly monotone *if and only if $t > s \Rightarrow P(a_1, ..., a_{i-1}, t, a_{i+1}..., a_n) > P(a_1, ..., a_{i-1}, s, a_{i+1}..., a_n)$ holds for all* i, $1 \leq i \leq n$, *and all $s, t, a_1, ..., a_{i-1}, a_{i+1}..., a_n \in A \setminus \{0\}$.*

Example 6. Reconsider example 3. Let $A = \mathbb{N}\setminus\{0\}$. Obviously, both H and Q are monotone polynomials. □

Definition 15 (monotone polynomial (pre-)interpretation). *A polynomial pre-interpretation is called* monotone *if it associates each function symbol f/n, $n > 0$ in Fun_P with a strictly monotone polynomial. A polynomial interpretation is* monotone *if it consists of a* monotone polynomial pre-interpretation *and an assignment that associates each predicate symbol p/n, $n > 0$, in $Pred_P$ with a strictly monotone polynomial.*

Usually, when talking about rigidity, we are only interested in rigidity of a set of terms (or atoms) w.r.t. a particular norm (or level mapping). In [2], Bossi, Cocco and Fabris discussed rigidity of *Call(P,S)* w.r.t. a semi-linear norm and a level mapping for some P and S. It is then generally extended to the case of rigidity of *Call(P,S)* w.r.t. a general term ordering in [5]. In this paper, we discuss rigidity of terms (or atoms) w.r.t. a polynomial interpretation and show that it is also an extension of [2]. Let us recall and extend some basic notions defined in [2].

Definition 16 (rigidity w.r.t. a polynomial (pre-)interpretation). *A term $t \in U_P^E$ is called* rigid w.r.t. a polynomial pre-interpretation J *if and only if for any substitution θ, $\|t\|_J \leq \geq_A \|t\theta\|_J$. An atom $A \in B_P^E$ is called* rigid w.r.t. a polynomial interpretation I *if and only if for any substitution θ, $|A|_I \leq \geq_A |A\theta|_I$. In this case, J and I are said to be rigid on, respectively, t and A.*

The notion of rigidity on a term or an atom is naturally extended to the notion of rigidity on a set of terms or atoms. In particular, we are interested in polynomial interpretations that are rigid on a call set $Call(P, S)$ for some P and S.

Definition 17. *Let J be a polynomial pre-interpretation and t be a term. The i^{th} occurrence $X_{(i)}$ of a variable X in t is called* relevant w.r.t. J *if there exists a replacement $\{s \rightarrow X_{(i)}\}$ of a term s for $X_{(i)}$ such that $\|t\{s \rightarrow X_{(i)}\}\|_J \not\equiv \|t\|_J$. We call $VREL(t)$ the set of all relevant occurrences of variables in t.*

Obviously from definition 17, if a term t is not rigid w.r.t. J, there must be some relevant occurrence of some variable in t.

Example 7. Let $t = [X|X]$ and J be the polynomial pre-interpretation imposed by the list-length norm $\|.\|_l$, $P_{[H|T]} = 1 + P_T$. Then, $VREL(t) = \{X_{(2)}\}$. □

Proposition 2. *Let J be a polynomial pre-interpretation and t be a term. If $VREL(t) = \emptyset$, then t is rigid w.r.t. J. For the reverse direction, if J is monotone and t is rigid w.r.t. J, then $VREL(t) = \emptyset$.*

Proof. See [14].

The following proposition shows that monotone polynomial pre-interpretations characterize relevant subterms in a purely syntactic way.

Proposition 3. *For any polynomial pre-interpretation J, for any term t, the following property holds:*

(i) $VREL(t) = \{t\}$ *if t is a variable,*
(ii) $VREL(t) \subseteq \cup_{j=1,...,m} VREL(t_{fj})$, *if $t = f(t_1, ..., t_n)$ and $P_t = P_f(P_{t_{f_1}}, ..., P_{t_{f_m}})$ is the polynomial interpretation of t ($t_{fj}, 1 \leq j \leq m$, are the selected subterms of t under J),*
(iii) If J is monotone, then the inclusion in the conclusion of ii) becomes an equality.

Proof. The proof is similar to the proof in [2] except that it is extended to the case of polynomial pre-interpretations. □

The major advantage of monotone polynomial pre-interpretations is that we can check the rigidity of a term t w.r.t. a given monotone polynomial pre-interpretation in a syntactic way: namely to verifying emptiness of $VREL(t)$. In principle, another way of verifying that t is rigid under J is to compute P_t and check that it is variable-free. However, this is computationally more expensive.

3.3 Applying Rigid Order Acceptability to Polynomial Interpretations

First, we need the following notion of polynomial interargument relations.

Definition 18 (polynomial interargument relation). *Let P be a program, p/n be a predicate in P and I be a polynomial interpretation for the language*

L underlying P. A polynomial interargument relation for p *is a relation* $R_p = \{(t_1, ..., t_n) | t_i \in Term_P \wedge \varphi_p(P_{t_1}, ..., P_{t_n})\}$, *where:*

- $\varphi_p(P_{t_1}, ..., P_{t_n})$ *is a formula in a disjunctive normal form,*
- *each conjunct in* φ_p *is either* $P_{s_i} \geq_A P_{s_j}$, $P_{s_i} >_A P_{s_j}$, $P_{s_i} \leq \geq_A P_{s_j}$ *or* $P_{s_i} \|_{\geq_A} P_{s_j}$, *where* s_i, s_j *are constructed from* $t_1, ..., t_n$ *by applying functors of P.*

R_p *is a* valid polynomial interargument relation for p/n *w.r.t. I if and only if for every* $p(t_1, ..., t_n) \in Atom_P$: $P \models p(t_1, ..., t_n)$ *implies* $(t_1, ..., t_n) \in R_p$.

Using the notions of rigidity and polynomial interargument relations w.r.t. a polynomial interpretation integrated with definition 7, theorem 2 and definition 13 we obtain:

Proposition 4. *Let S be a set of atomic queries, P be a program and I be a polynomial interpretation for the language L underlying P. For each predicate p in P, let* R_p *be a valid polynomial interargument relation for p w.r.t. I. If I is rigid on* $Call(P, S)$ *such that*

- *for any clause* $H \leftarrow B_1, ..., B_n$,
- *for any atom* B_i *in its body, such that* $rel(B_i) \backsimeq rel(H)$,
- *for any substitution* θ, *such that the arguments of the atoms in* $(B_1, ..., B_{i-1})\theta$ *satisfy their associated polynomial interargument relations* $R_{rel(B_1)}, ...,$ $R_{rel(B_{i\ 1})}$,

$$P_{H\theta} >_A P_{B_i\theta},$$

then P left-terminates w.r.t. S.

Example 8. Reconsider example 2. We are interested in proving termination of the program w.r.t. the query set $S = \{d(t_1, t_2) | t_1$ is a ground term and t_2 is a free variable$\}$. Observe that $Call(P, S)$ coincides with S.

Let I be a polynomial interpretation that consists of a set $A = N \backslash \{0, 1\}$, an assignment that associates the function symbol $der/1$ with the polynomial $P_{der} = X^2$, $+/2$ with $P_+ = X_1 + X_2$, $*/2$ with $P_* = X_1 * X_2$, the constant u with a constant $c_u \in A$ and an assignment that associates the predicate symbol $d/2$ with $P_d = X$, where the variable X corresponds to the first argument position of $d/2$. Let $R_d = \{(t_1, t_2) | t_1, t_2 \in Term_P$ and $P_{t_1} \geq_A P_{t_2}\}$ be a polynomial interargument relation w.r.t. the predicate $d/2$.

It is easy to verify that I is rigid on $Call(P, S)$ and R_d is valid w.r.t. I. Then, the program terminates if the following holds:

$$|d(der(X + Y), DX + DY)\theta|_I >_A |d(der(X), DX)\theta|_I$$
$$d(der(X), DX)\theta \text{ satisfies } R_d \text{ implies}$$
$$|d(der(X + Y), DX + DY)\theta|_I >_A |d(der(Y), DY)\theta|_I$$
$$|d(der(X * Y), X * DY + Y * DX)\theta|_I >_A |d(der(X), DX)\theta|_I$$
$$d(der(X), DX)\theta \text{ satisfies } R_d \text{ implies}$$
$$|d(der(X * Y), X * DY + Y * DX)\theta|_I >_A |d(der(Y), DY)\theta|_I$$

$$|d(der(der(X)), DDX)\theta|_I >_A |d(der(X), DX)\theta|_I$$
$$d(der(X), DX)\theta \text{ satisfies } R_d \text{ implies}$$
$$|d(der(der(X)), DDX)\theta|_I >_A |d(der(DX), DDX)\theta|_I$$

They are equivalent to the following inequalities on $X, Y, DX \in Var_p$:

$$(X+Y)^2 >_A X^2 \qquad X^2 >_A DX \Rightarrow (X*Y)^2 >_A Y^2$$
$$X^2 >_A DX \Rightarrow (X+Y)^2 >_A Y^2 \qquad X^4 >_A X^2$$
$$(X*Y)^2 >_A X^2 \qquad X^2 >_A DX \Rightarrow X^4 >_A DX^2$$

Since $A = \mathbb{N}\backslash\{0,1\}$, the above inequalities are easily verified and the program left-terminates. □

4 Automation: The General Idea

For automation of the approach, two sources of ideas and techniques are important:

- the generalisation of the constraint-based approach to termination analysis of [6] from linear norms and level mappings to polynomials.
- the integration of a number of useful results and heuristics from TRSs ([3,7,10,12,16]).

The idea of the approach in [6] is to set up a symbolic form for all concepts involved in the termination conditions: in our case, the polynomial interpretation of each function and predicate symbol, the polynomial interargument relations and polynomial ordering conditions in proposition 4. Note that if we do not put a limit on the maximal degree of the polynomial, then there can be no finite general form of the polynomial associated with a term (there are infinitely many monomials $a_i X^{i_1} X^{i_2} ... X^{i_k}$ to consider). This is why we associate each function and predicate symbol with a *simple-mixed* polynomial, which is either a multivariate polynomial with all variables of at most degree 1 or a unary polynomial of at most degree 2.

From TRSs we borrow a sufficient condition for monotonicity of the polynomials:

Proposition 5. *(see also [18]) Let* $P = \sum_{i=1}^{r} a_i X_1^{k_{i,1}} X_2^{k_{i,2}} ... X_m^{k_{i,m}}$ *be a polynomial from* A^m *to* A *for which* $A = \mathbb{N}\backslash\{0\}, m > 0$ *and* $a_i \geq 0$ *for all* $i = 1, ..., r, r > 0$. P *is strictly monotone if* $\sum_{i=1}^{r} a_i k_{i,j} > 0$ *for every* $j = 1, ..., m$.

Example 9. Reconsider example 4. Let the first and the second argument positions of the predicate $dist/2$ be, respectively, the input and output positions. For all other function symbols, let all arguments be the input arguments. Let I be a polynomial interpretation such that the constant x is associated with $x_I \in A$, the function symbol $+/2$ is associated with the polynomial $P_+(X,Y) = f_0^+ + f_1^+ X + f_2^+ Y + f_3^+ XY$, the function symbol $*/2$ is associated with the polynomial $P_*(X,Y) = f_0^* + f_1^* X + f_2^* Y + f_3^* XY$, and the predicate symbol $dist/2$

is associated with the polynomial $P_{dist}(X) = f_0^d + f_1^d X + f_2^d X^2$. I is monotone if $f_1^d + f_2^d * 2 > 0$, $f_1^+ + f_3^+ > 0$, $f_2^+ + f_3^+ > 0$, $f_1^* + f_3^* > 0$, $f_2^* + f_3^* > 0$. □

For the interargument relations, we only allow the linear interargument relations of [6], i.e. $R_{p/n} = \{(t_1, ..., t_n) | \sum_{i \in p_{inp}} p_i^e P_{t_i} \geq_A \sum_{j \in p_{out}} p_j^e P_{t_j} + p_0^e\}$, with $p_i^e \in \mathbb{N}$, $i \in \{1, ..., n\}$, p_{inp} and p_{out} respectively the sets of input and output argument positions of p/n. But because these are applied to polynomial interpretations of terms, they still give rise to non-linear conditions in general.

Example 9 (continued). As an example, the condition for a valid interargument relation R_{dist} applied to clause 1 is of the following form:

$$(d_1^e X \geq_A d_2^e U + d_0^e) \wedge (d_1^e Y \geq_A d_2^e V + d_0^e) \Rightarrow$$
$$d_1^e(f_0^+ + f_1^+ X + f_2^+ Y + f_3^+ XY) \geq_A d_2^e(f_0^+ + f_1^+ U + f_2^+ V + f_3^+ UV) + d_0^e. \quad \square$$

Next all other polynomial inequality conditions from proposition 4 are translated into constraints on the introduced symbols.

Example 9(continued). As an example, for recursive clause 1, the following constraints are imposed:

$$f_0^d + f_1^d(f_0^+ + f_1^+ X + f_2^+ Y + f_3^+ XY)$$
$$+ f_2^d(f_0^+ + f_1^+ X + f_2^+ Y + f_3^+ XY)^2 >_A f_0^d + f_1^d X + f_2^d X^2$$
$$d_1^e X \geq d_2^e U + d_0^e \Rightarrow f_0^d + f_1^d(f_0^+ + f_1^+ X + f_2^+ Y + f_3^+ XY)$$
$$+ f_2^d(f_0^+ + f_1^+ X + f_2^+ Y + f_3^+ XY)^2 >_A f_0^d + f_1^d Y + f_2^d Y^2 \quad \square$$

After normalisation, all the above constraints are transformed into the form: $P(X_1, ..., X_n) \geq_A 0 \Rightarrow Q(X_1, ..., X_m) \geq_A 0$ or the form $P(X_1, ..., X_n) \geq_A 0$. In [6] techniques are proposed to transform the constraints of the first type into constraints of the second type. These can be extended to polynomials.

The following step is to transform all those constraints into constraints which contain only coefficients as variables. It can be done by applying one of the following approaches from TRS:

In the first approach of [10], the first step is to move from $A \subseteq \mathbb{N}$ to $\mathbb{R}^+$. Let a be $min\{c_I | c_I \in A$ is a polynomial interpretation of a constant c$\}$. Then instead of demanding that any of these constraints should hold (i.e. $P(X_1, ..., X_n) \geq 0$ for all $X_1, ..., X_n \in A$), it is sufficient to prove that $P(X_1, ..., X_n) \geq 0$ for all $X_1, ..., X_n \in A_R$, $A_R = \mathbb{R}^+ \setminus [0, a)$. The next step is to apply repeatedly the following differentiation rules to transform all polynomial constraints to constraints containing only coefficients as variables:

$$\frac{P(..., X_i, ...) > 0}{P(..., a, ...) > 0, \frac{\partial P(..., X_i, ...)}{\partial X_i} \geq 0} \qquad \frac{P(..., X_i, ...) \geq 0}{P(..., a, ...) \geq 0, \frac{\partial P(..., X_i, ...)}{\partial X_i} \geq 0}$$

Note the introduction of the inequations on the derivatives, which are actually extra constraints. Within TRS it has been argued that imposing these extra constraints is most often reasonable as it allows to eliminate all variables X_i and because, if a solution to the original problem exists, the solution space is usually

large enough to also contain an element that respects the extra constraints. There are a number of heuristics that can be applied to solve these constraints.

In the second approach of [3], all constraints are transformed to Diophantine inequalities. Then, if we put an arbitrary bound on the values of variable coefficients (e.g., $[0, B]$), the problem becomes solving a *finite domain constraint satisfaction problem* for a finite set of variables. Here finite-domain constraint solvers provide a variety of techniques to solve the remaining inequalities.

5 Conclusions

Since a few years ago, the LP termination analysis community and the TRS termination analysis community jointly organize the "International Workshop on Termination" (WST). These workshops have raised a considerable interest in gaining a better understanding of each others approaches. It soon became clear that there has to be a close relationship between one of the most popular techniques in TRS, polynomial interpretations, and one of the key techniques in LP, acceptability with linear norms and level mappings. However, partly because of the distinction between orderings over the natural numbers (LP) versus orderings over polynomials (TRS), the actual relation between the approaches was unclear.

One main conclusion of the research that led to this paper is that the distinction is a superficial one. Although termination conditions in LP are formulated in terms of mappings to natural numbers, the actual termination proofs do not reason on natural numbers. They are formulated in terms of linear inequalities. In fact, LP termination analysis systems never work on the basis of the norm and the level mapping; they work on the level of the *abstract norm* and *abstract level mapping* (see [17]). As such, one outcome of the work is that, indeed, the polynomial interpretations of TRS are a direct generalization of the current LP practice.

On the more technical level, the contribution of this paper is that we provide a complete theoretical framework for polynomial interpretations in LP termination analysis. Part of this builds strongly on the results in [5] on order acceptability, another part extends the results of Bossi et al. [2] on syntactic characterization of rigidity.

In the paper we only provide two examples of the class of programs for which the extension from linear to polynomial interpretations is important. Note that typical examples in LP termination analysis are often deliberately chosen to be linear, to remain in the scope of the designed techniques. Non-linear polynomial functions are present in many real world problems and programs encoding these problems are bound to require polynomial interpretations for their termination proofs.

It remains to be studied how we can benefit from the huge amount of work that people in TRS termination analysis have spent on automating proofs with polynomial interpretations and how integration of these techniques with the best approaches of LP termination analysis can lead to even more powerful techniques. We expect that this will lead to the development of a powerful new termination analyzer in the near future.

Acknowledgements

Manh Thang Nguyen is supported by GOA/2003/08. We thank the referees for useful comments.

References

1. K. R. Apt. Logic programming. In *Handbook of theoretical computer science (vol. B): formal models and semantics*, pages 493–574. MIT Press, 1990.
2. A. Bossi, N. Cocco, and M. Fabris. Proving termination of logic programs by exploiting term properties. In *TAPSOFT, Vol.2*, pages 153–180, 1991.
3. E. Contejean, C. Marché, A. P. Tomás, and X. Urbain. Mechanically proving termination using polynomial interpretations. *J. Auto. Reason.*, 2005.
4. D. De Schreye and S. Decorte. Termination of logic programs: the never-ending story. *J. Log. Program.*, 19-20:199–260, 1994.
5. D. De Schreye and A. Serebrenik. Acceptability with general orderings. In *Computational Logic: Logic Programming and Beyond*, pages 187–210. Springer Verlag, 2002.
6. S. Decorte, D. De Schreye, and H. Vandecasteele. Constraint based automatic termination analysis of logic programs. *ACM Trans. Program. Lang. Syst*, 21(6):1137–1195, November 1999.
7. N. Dershowitz. Termination of rewriting. *J. Symb. Comput.*, 3(1-2):69–116, 1987.
8. N. Dershowitz. 33 examples of termination. *LNCS*, 909:16–26, 1995.
9. M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. Declarative modeling of the operational behaviour of logic languages. *Theor. Comput. Sci.*, 63(3):289–318, 1989.
10. J. Giesl. Generating polynomial orderings for termination proofs. In *RTA*, pages 426–431, 1995.
11. G. Janssen and M. Bruynooghe. Deriving descriptions of possible values of program variables by means of abstract interpretation. *J. Log. Program.*, 13(2&3):205–258, 1992.
12. D. S. Lankford. On proving term rewriting systems are noetherian. Technical report, Mathematics Department, Louisiana Tech. University, Ruston, LA, 1979.
13. J. W. Lloyd. *Foundations of Logic Programming*. Springer Verlag, Berlin, 1987.
14. M. T. Nguyen and D. De Schreye. Polynomial interpretations as a basis for termination analysis of logic programs. Technical report, Department of Computer Science, K.U.Leuven, Belgium, 2005.
15. A. Serebrenik. *Termination Analysis of Logic Programs*. PhD thesis, Department of Computer Science, K.U.Leuven, Belgium, 2003.
16. J. Steinbach. Generating polynomial orderings. *Inf. Process. Lett.*, 49(2):85–93, 1994.
17. K. Verschaetse and D. De Schreye. Deriving termination proofs for logic programs, using abstract procedures. In *Proceedings 8th ICLP*, pages 301–315, 1991.
18. H. Zantema. *Termination, In Terese, Term Rewriting Systems*, chapter 6. Cambridge Univ. Press, 2003.

Testing for Termination with Monotonicity Constraints

Michael Codish[1,⋆], Vitaly Lagoon[2], and Peter J. Stuckey[2,3]

[1] Department of Computer Science, Ben-Gurion University, Israel
[2] Department of Computer Science and Software Engineering, The University of Melbourne, Australia
[3] NICTA Victoria Laboratory
mcodish@cs.bgu.ac.il,
{lagoon, pjs}@cs.mu.oz.au

Abstract. Termination analysis is often performed over the abstract domains of monotonicity constraints or of size change graphs. First, the transition relation for a given program is approximated by a set of descriptions. Then, this set is closed under a composition operation. Finally, termination is determined if all of the idempotent loop descriptions in this closure have (possibly different) ranking functions. This approach is complete for size change graphs: An idempotent loop description has a ranking function if and only if it has one which indicates that some specific argument decreases in size. In this paper we generalize the size change criteria for size change graphs which are not idempotent. We also illustrate that proving termination with monotonicity constraints is more powerful than with size change graphs and demonstrate that the size change criteria is incomplete for monotonicity constraints. Finally, we provide a complete termination test for monotonicity constraints.

1 Introduction

Termination analysis is often performed by approximating the transition relation induced by a program. For logic programs this is a relation on the calls to predicates encountered during computation. A transition from call $p(\bar{t})$ to a subsequent call $q(\bar{s})$ in some computation can be represented as a binary clause of the form $p(\bar{t}) \leftarrow q(\bar{s})$. A semantics which specifies this transition relation is introduced and shown to make calls observable in [6]. It is shown to make termination observable in [2]. The TerminWeb termination analyzer for logic programs [2] is basically, a meta-interpreter for an abstraction of this semantics with transitions approximated by monotonicity constraints [1].

Size change graphs were introduced in [7] and are similar to monotonicity constraints. These two domains are used by an increasing number of termination analyzers for a variety of languages including: TermiLog [9,8] and TerminWeb [2] for logic programs, implementations for simple first-order functional languages [13,4] and the AProVE analyzer for term rewrite systems [12].

⋆ Research performed at the University of Melbourne.

M. Gabbrielli and G. Gupta (Eds.): ICLP 2005, LNCS 3668, pp. 326–340, 2005.

Monotonicity constraints and size change graphs can be represented as (abstract) binary clauses of the form $p(\bar{x}) \leftarrow \mu(\bar{x}, \bar{y}), q(\bar{y})$ where $\bar{x}$ and $\bar{y}$ are tuples of distinct variables and $\mu(\bar{x}, \bar{y})$ is a conjunction of binary constraints of the form $u > v$,$u \geq v$ on the sizes of the data before and after a corresponding concrete transition. For monotonicity constraints u and v are any variables among $\bar{x}$ and $\bar{y}$. Size change graphs are more restricted with u in $\bar{x}$ and v in $\bar{y}$. When p and q are the same symbol the (abstract) binary clause is recursive and describes a loop.

Example 1. *Consider the predicate* `ackerman/3` *below (on the left) which computes Ackerman's function. The size change graphs (on the right) describe the induced transition relation. In subsequent calls to this predicate, either the first argument decreases in size or else it does not increase in size and the second argument decreases in size.*

$ackerman(0, N, s(N)).$
$ackerman(s(M), 0, Res) \leftarrow$
$\quad ackerman(M, s(0), Res).$
$ackerman(s(M), s(N), Res) \leftarrow$
$\quad ackerman(s(M), N, Res1),$
$\quad ackerman(M, Res1, Res).$

$ackermann(x_1, x_2, x_3) \leftarrow$
$\quad x_1 > y_1, ackermann(y_1, y_2, y_3).$
$ackermann(x_1, x_2, x_3) \leftarrow$
$\quad x_1 \geq y_1, x_2 > y_2, ackermann(y_1, y_2, y_3).$

This paper is not concerned with how approximations of transition relations are obtained, but rather with the question of how termination is proven given such an approximation. Existing analyzers provide the approximations as a starting point for this paper.

In the classic approach, to prove termination of a program P one should identify a ranking function f from program states to the elements of a well founded domain and show that f decreases as execution proceeds through all of the loops in P. For example, one might show that the function $f(u_1, u_2, u_3) = \langle u_1, u_2 \rangle$ decreases with respect to the lexicographic ordering for both of the loop descriptions in Example 1. This is a *global* ranking function — it is shown to decrease for all loop descriptions in the analysis.

An alternative approach is based on the application of *local* ranking functions. In this approach, under the condition that the set of loop descriptions is "closed under composition" (resolution of abstract binary clauses), termination is guaranteed if for each individual loop description a (possibly different) ranking function is shown to decrease when execution goes through that loop.

The main advantage in applying local ranking functions is that they take a simpler form than corresponding global ranking functions and are easy to find automatically. Moreover, it is sufficient to find ranking functions only for those descriptions which are idempotent (a description is idempotent if it remains invariant when composed with itself) [7]. There is also a disadvantage: as illustrated in [7], for size change graphs, there is a worst case exponential growth factor (in the number of arguments) associated with the computation of closure under composition. The following example illustrates the advantage. The disadvantage is the topic of another paper.

Example 2. *Consider the following three loop descriptions (which are idempotent and closed under composition)*[1].

$$p(x_1, x_2, x_3) \leftarrow x_1 > y_2, x_2 \geq y_2, x_3 > y_3, p(y_1, y_2, y_3).$$
$$p(x_1, x_2, x_3) \leftarrow x_1 > y_1, x_2 \geq y_1, p(y_1, y_2, y_3).$$
$$p(x_1, x_2, x_3) \leftarrow x_1 > y_2, x_2 > y_2, p(y_1, y_2, y_3).$$

Local ranking functions are respectively $f_1(u_1, u_2, u_3) = u_3$, $f_2(u_1, u_2, u_3) = u_1$, *and* $f_3(u_1, u_2, u_3) = u_2$. *The function* $min(u_1, u_2)$ *decreases for the third loop description and does not increase for the first two descriptions. The functions* $\langle min(u_1, u_2), u_3 \rangle$ *and* $\langle min(u_1, u_2), u_1 \rangle$ *decrease (with respect to the lexicographic ordering) for the first two and last two loop descriptions respectively. One can verify that there does not exist any function based on lexicographic ordering of linear functions (even allowing minimum and maximum functions) that is a global ranking function for this example.*

The correctness of the local approach is first given by Dershowitz *et al.* [3] and is based on the application of Ramsey's Theorem [11]. The approach is also complete [7] in the sense that an idempotent size change graph has a ranking function if and only if it has one of the form $f(\bar{u}) = u_i$. Hence an algorithm to decide termination for size change graphs is obtained.

The first contribution of this paper is to generalize the completeness result for size change graphs which are not necessarily idempotent. Here, if there exists any ranking function then there exists one of the form $f(u_1, \ldots, u_n) = \Sigma a_i u_i$ with all coefficients $a_i \in \{0, 1\}$. In [8] the authors suggest a termination test for monotonicity constraints which is the one implemented in TermiLog and TerminWeb. We show that this test provides a simple decision procedure for the existence of a ranking function for a size change graph (idempotent or not).

We proceed to illustrate that size change termination is incomplete for monotonicity constraints which are not size change graphs. This fact has been overlooked until now. Both TermiLog and TerminWeb implement for monotonicity constraints the test which is complete for size change graphs. The second contribution of this paper is to provide completeness results for monotonicity constraints: for an idempotent monotonicity constraint, if there exists any ranking function then there exists one of the form $f(u_1, \ldots, u_n) = u_i$ or of the form $f(u_1, \ldots, u_n) = u_i - u_j$. For arbitrary monotonicity constraints if there exists a ranking function then there exists one which is linear.

In [10], the authors present an efficient test for termination for loop descriptions for a domain which is more general than monotonicity constraints. Their approach is complete with respect to linear ranking functions: if there exists a linear ranking function then the proposed procedure will succeed (and synthesize it). However, if the procedure fails, it could be the case that there exists a ranking function which is non-linear. Our result implies that the test presented in [10] is complete for monotonicity constraints.

The remainder of this paper is structured as follows: Section 2 introduces monotonicity constraints and size change graphs. Section 3 describes the com-

[1] This example was suggested by Amir Ben Amram.

pleteness result for idempotent size change graphs and extends it for arbitrary size change graphs. Section 4 illustrates that size change termination is incomplete for monotonicity constraints and provides two completeness results: first for idempotent descriptions and second for descriptions which are not necessarily idempotent. Section 5 concludes.

2 Monotonicity Constraints and Size Change Graphs

Let $\bar{x} = \langle x_1, \ldots, x_n \rangle$ and $\bar{y} = \langle y_1, \ldots, y_n \rangle$ denote n-tuples of variables taking non-negative integer values. When clear from the context we let these denote the corresponding sets of variables. Intuitively, these values correspond to the sizes of terms in a computation, with respect to a given norm function.

Definition 1 (monotonicity constraint, size change graph). *A monotonicity constraint is a binary clause of the form* $p(\bar{x}) \leftarrow \mu(\bar{x}, \bar{y}), q(\bar{y})$ *where* $\mu(\bar{x}, \bar{y})$ *is a conjunction of constraints of the form* $u \geq v + b$*, denoted also* $u \succ^b v$*, with* $u, v \in \bar{x} \cup \bar{y}$ *and* $b \in \{0, 1\}$*. We write also* $u > v$ *and* $u \geq v$ *when respectively* $b = 1$ *or* $b = 0$ *or* $u \succ v$ *when not distinguishing between the two cases. If constraints are restricted so that* $u \in \bar{x}$*,* $v \in \bar{y}$ *then* $\mu(\bar{x}, \bar{y})$ *is called a size change graph. When clear from the context we refer to* $\mu(\bar{x}, \bar{y})$ *as the monotonicity constraint (or size change graph).*

A monotonicity constraint $\mu(\bar{x}, \bar{y})$ can be viewed as a directed graph with nodes $\bar{x} \cup \bar{y}$ and an edge labeled by b from u to v if and only if $\mu(\bar{x}, \bar{y}) \models u \succ^b v$. For size change graphs this view gives a directed bipartite graph. In the examples, graphs are depicted with solid and dashed arrows representing edges of the form $u > v$ and $u \geq v$ respectively. We often omit edges which can be inferred from those drawn. A monotonicity constraint is satisfiable if and only if its graph representation has no cycle with a solid edge. Note that a size change graph is always satisfiable. The size change graphs from Example 2 are depicted in Figure 1.

Monotonicity constraints induce corresponding transition relations.

Definition 2 (transition relation).
A monotonicity constraint $p(\bar{x}) \leftarrow \mu(\bar{x}, \bar{y}), q(\bar{y})$ *induces a transition relation* $\sqsubseteq_\mu$ *on labeled vectors of non-negative integers given by* $p(\bar{a}) \sqsubseteq_\mu q(\bar{b})$ *if and only if* $\mu(\bar{a}, \bar{b})$ *is valid. When clear from the context we drop the labels* p *and* q*.*

A derivation for a set of monotonicity constraints is a chain in the corresponding transition relations. For the completeness results of this paper it is sufficient to consider derivations induced by a single recursive monotonicity constraint.

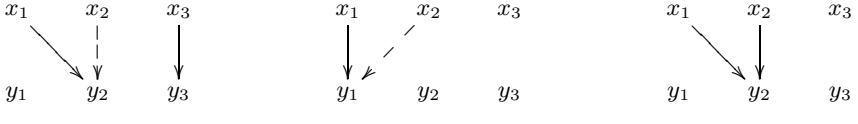


Fig. 1. Size change graphs from Example 2

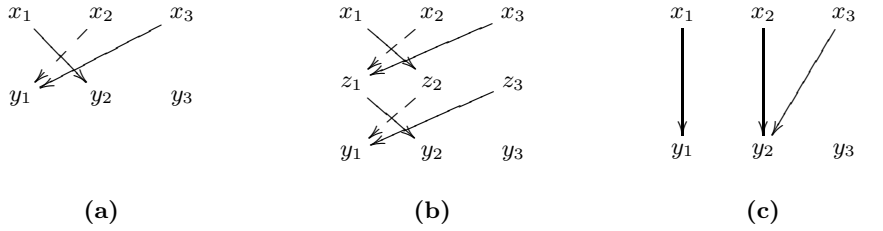


Fig. 2. Constraints of Example 3: **(a)** monotonicity constraint $\mu(\bar{x}, \bar{y})$, **(b)** derivation constraint $\mu(\bar{x}, \bar{z}, \bar{y})$, and **(c)** self composition $\mu^2(\bar{x}, \bar{y})$

Definition 3 (derivation, derivation constraint). *Let $p(\bar{x}) \leftarrow \mu(\bar{x}, \bar{y}), p(\bar{y})$ be a recursive monotonicity constraint. A derivation of $\mu(\bar{x}, \bar{y})$ is a chain of non-negative integer vectors of the form $\bar{a}^0 \sqsubseteq_\mu \bar{a}^1 \sqsubseteq_\mu \cdots \sqsubseteq_\mu \bar{a}^k$ which may also be infinite. For a finite derivation, there is a corresponding derivation constraint $\mu(\bar{x}^0, \ldots, \bar{x}^k) = \mu(\bar{x}^0, \bar{x}^1) \wedge \cdots \wedge \mu(\bar{x}^{k-1}, \bar{x}^k)$. A derivation constraint can also be viewed as a directed graph.*

The composition of monotonicity constraints is defined through renaming of variables (graph nodes), conjunction and entailment. Derivation constraints and composition are illustrated in Figure 2.

Definition 4 (composition of monotonicity constraints). *The composition of monotonicity constraints $\mu_1(\bar{x}, \bar{y})$ and $\mu_2(\bar{x}, \bar{y})$ is given by*

$$\mu_1(\bar{x}, \bar{y}) \circ \mu_2(\bar{x}, \bar{y}) = \bigwedge \left\{ \; u \succ^b v \;\middle|\; u, v \in \bar{x} \cup \bar{y}, \mu_1(\bar{x}, \bar{z}) \wedge \mu_2(\bar{z}, \bar{y}) \models u \succ^b v \; \right\}.$$

We denote by $\mu^k(\bar{x}, \bar{y})$ the composition of $\mu(\bar{x}, \bar{y})$ with itself k times.

Definition 5 (idempotent monotonicity constraints). *A monotonicity constraint, $\mu(\bar{x}, \bar{y})$, is* idempotent *if and only if $\mu^2(\bar{x}, \bar{y}) = \mu(\bar{x}, \bar{y})$.*

Example 3. *The monotonicity constraint $\mu(\bar{x}, \bar{y}) = x_1 > y_2 \wedge x_2 \geq y_1 \wedge x_3 > y_1$ is depicted as Figure 2(a). The derivation constraint $\mu(\bar{x}, \bar{z}, \bar{y})$ is shown in Figure 2(b) and consists of 2 copies of $\mu(\bar{x}, \bar{y})$. The self composition $\mu^2(\bar{x}, \bar{y})$ is depicted in Figure 2(c). The constraint of Figure 2(a) is not idempotent but the constraint of Figure 2(c) is. Note that $\mu(\bar{x}, \bar{z}, \bar{y}) \models x_3 \geq y_2 + 2$ while $\mu^2(\bar{x}, \bar{y}) \models x_3 \geq y_2 + 1$. This illustrates that (projections of) derivation constraints are not monotonicity constraints.*

Definition 6 (closure under composition). *Let $\mathcal{G}$ be a set of monotonicity constraints. We denote by $\mathcal{G}^*$ the closure of $\mathcal{G}$ under composition. This is the smallest superset of $\mathcal{G}$ such that if $\mu_1(\bar{x}, \bar{y}) \in \mathcal{G}^*$ and $\mu_2(\bar{x}, \bar{y}) \in \mathcal{G}^*$ then also $\mu_1(\bar{x}, \bar{y}) \circ \mu_2(\bar{x}, \bar{y}) \in \mathcal{G}^*$.*

A central notion when proving termination is that of a ranking function. We focus on ranking functions for individual monotonicity constraints.

Definition 7 (ranking function). *A ranking function for a monotonicity constraint $\mu(\bar{x}, \bar{y})$ is a mapping f from tuples of non-negative integers to a well founded domain $(D, \prec_D)$ such that $\mu(\bar{x}, \bar{y}) \models f(\bar{y}) \prec_D f(\bar{x})$.*

In this paper we often choose ranking functions mapping to the domain of natural numbers $(\mathcal{N}, <)$ with the standard ordering.

Example 4. *Consider the size change graph $\mu(\bar{x}, \bar{y})$ from Figure 2(a). The function $f(\bar{u}) = u_1 + u_2$ on the natural numbers is a ranking function for $\mu(\bar{x}, \bar{y})$.*

Theorem 1 (correctness [3,7]). *Let $\mathcal{G}$ be a finite set of monotonicity constraints which approximates the loops in the transition relation of a program P. If for each of the idempotent monotonicity constraints in $\mathcal{G}^*$ there exists a (possibly different) ranking function then P terminates.*

The proofs in [3] and [7] are for monotonicity constraints and size change graphs respectively. The proofs are essentially the same and presented as applications of Ramsey's Theorem [11].

3 Completeness for Size Change Graphs

In [7] the authors present a completeness result for size change graphs. We rephrase this result in terms of ranking functions.

Theorem 2 (completeness – idempotent size change graphs). *Let $\mu(\bar{x}, \bar{y})$ be an idempotent size change graph. If there exists a ranking function for $\mu(\bar{x}, \bar{y})$ then there exists one mapping to $(\mathcal{N}, <)$ of the form $f(u_1, \ldots, u_n) = u_i$.*

We present a proof, different from the one in [7]. Following this proof will help understand its generalizations in the remainder of this paper. The proof relies on the following two lemmata.

Lemma 1. *If $\mu(\bar{x}, \bar{y})$ is an idempotent monotonicity constraint which implies $x_i \succ^{b_1} y_j$ and $x_j \succ^{b_2} y_k$ (or $y_i \succ^{b_1} x_j$ and $y_j \succ^{b_2} x_k$) then it implies also $x_i \succ^{b_1 \vee b_2} y_k$ (or $y_i \succ^{b_1 \vee b_2} x_k$).*

Proof. By definitions of composition and idempotence.

Lemma 2. *If $\mu(\bar{x}, \bar{y})$ is an idempotent size change graph, then either for some argument i, $\mu(\bar{x}, \bar{y}) \models x_i > y_i$; or $\mu(\bar{x}, \bar{y}) \wedge \bar{x} = \bar{y}$ is satisfiable.*

Proof. Let $\mu(\bar{x}, \bar{y})$ be an idempotent size change graph and assume that the second condition does not hold. Since $\mu(\bar{x}, \bar{y})$ is a size change graph, $\mu(\bar{x}, \bar{y})$ must be satisfiable and there must be an alternating sequence of constraints implied by $\mu(\bar{x}, \bar{y})$ and $\bar{x} = \bar{y}$, forming a simple cycle, as depicted in Figure 3 and of the form: $x_{i_1} \succ^{b_1} y_{i_2} = x_{i_2} \succ^{b_2} y_{i_3} \ldots y_{i_k} = x_{i_k} \succ^{b_k} y_{i_1} = x_{i_1}$ with $b_1 \vee \cdots \vee b_k = 1$. From idempotence using Lemma 1 since the constraints $x_{i_1} \succ^{b_1} y_{i_2}$, $x_{i_2} \succ^{b_2} y_{i_3}$, …, $x_{i_k} \succ^{b_k} y_{i_1}$ are implied by $\mu(\bar{x}, \bar{y})$ then so is the constraint $x_{i_1} > y_{i_1}$. Hence the first condition must hold.

For the proof of Theorem 2, we know only that $\mu(\bar{x}, \bar{y})$ has some ranking function of an unknown form. This is sufficient to guarantee that the transition relation induced by $\mu(\bar{x}, \bar{y})$ has no infinite derivations.

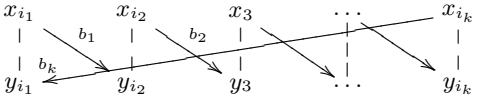


Fig. 3. An alternating cycle of inconsistent constraints from $\mu(\bar{x}, \bar{y})$ and $\bar{x} = \bar{y}$

Proof. (of Theorem 2) Assume $\mu(\bar{x}, \bar{y})$ has a ranking function. So $\mu(\bar{x}, \bar{y}) \wedge \bar{x} = \bar{y}$ is not satisfiable. Otherwise there would be a vector $\bar{a}$ such that $\bar{a} \sqsubseteq_\mu \bar{a}$ giving an infinite derivation. By Lemma 2, $\mu(\bar{x}, \bar{y}) \models x_i > y_i$ and so $f(u_1, \ldots, u_n) = u_i$ is a ranking function.

Theorems 1 and 2 provide the basis for a decision procedure [7]. For a set of size change graphs $\mathcal{G}$, first compute $\mathcal{G}^*$ and then check for each $\mu(\bar{x}, \bar{y}) \in \mathcal{G}^*$ if it is idempotent and if: $\bigvee_{1 \le i \le n} (\mu(\bar{x}, \bar{y}) \models x_i > y_i)$. We can strengthen this statement checking instead for each idempotent size graph the condition:

$$\mu(\bar{x}, \bar{y}) \models \bigvee_{1 \le i \le n} (x_i > y_i) \tag{1}$$

This is justified by the following result.

Lemma 3. *Let $\mu(\bar{x}, \bar{y})$ be an idempotent size change graph. Then*

$$\bigvee_{1 \le i \le n} (\mu(\bar{x}, \bar{y}) \models x_i > y_i) \quad \Leftrightarrow \quad \mu(\bar{x}, \bar{y}) \models \bigvee_{1 \le i \le n} (x_i > y_i).$$

The graph in Figure 2(a) demonstrates that this result does not hold for non-idempotent size change graphs.

Proof. (of Lemma 3) **(⇒)** Obvious. **(⇐)** If $\mu(\bar{x}, \bar{y}) \models \bigvee_i x_i > y_i$ then $\mu(\bar{x}, \bar{y}) \wedge \bigwedge_i (x_i = y_i)$ is not satisfiable. Hence, by Lemma 2, $\mu(\bar{x}, \bar{y})$ implies a constraint of the form $x_i > y_i$ as required.

It is interesting to note that the condition of Equation (1) is precisely that implemented in TermiLog and TerminWeb where no test for idempotence is applied. We proceed to generalize the completeness result of Theorem 2 to apply for size change graphs which are not necessarily idempotent. We show the condition of Equation (1) is complete for arbitrary size change graphs, idempotent or not. It means that we need not test for idempotence in an implementation.

Theorem 3 (Completeness – arbitrary size change graphs). *Let $\mu(\bar{x}, \bar{y})$ be a size change graph (not necessarily idempotent). If there exists any ranking function for $\mu(\bar{x}, \bar{y})$ then there exists one mapping to the non-negative integers of the form $f(u_1, \ldots, u_n) = \sum a_i u_i$ with all coefficients $a_i \in \{0, 1\}$.*

The proof follows from the observation that a set of size change graphs closed under composition is a finite semigroup with composition as the operator.

Lemma 4 (idempotent self composition). *Let $\mu(\bar{x}, \bar{y})$ be a monotonicity constraint. Then there exists a positive integer k such that $\mu^k(\bar{x}, \bar{y})$ is idempotent.*

Proof. A finite non-empty semigroup of the form $\{ a^k \mid k \in \mathcal{Z}^+ \}$ contains precisely one idempotent element [5].

Proof. (of Theorem 3) Let f mapping to $(D, \prec_D)$ be a ranking function for size change graph $\mu(\bar{x}, \bar{y})$. By Lemma 4, there exists a positive k such that $\mu^k(\bar{x}, \bar{y})$ is idempotent. By transitivity of $\prec_D$, f is also a ranking function for $\mu^k(\bar{x}, \bar{y})$. By Theorem 2, $\mu^k(\bar{x}, \bar{y})$ has a ranking function of the form $f'(u_1, \ldots, u_n) = u_i$ indicating the presence of a corresponding strict down arrow $\mu^k(\bar{x}, \bar{y}) \models x_i > y_i$. It follows that $\mu(\bar{x}^0, \bar{x}^1), \ldots, \mu(\bar{x}^{k-1}, \bar{x}^k)$ imply respectively constraints forming a chain of the form $x^0_{i_0} \succ^{b_1} x^1_{i_1} \succ^{b_2} x^2_{i_2} \succ \cdots \succ x^{k-1}_{i_{k\ 1}} \succ^{b_k} x^k_{i_k}$ with $i_0 = i_k = i$ such that at least one of $b_1, \ldots, b_k$ is 1 (i.e., strict). It follows that $\mu(\bar{x}, \bar{y})$ implies corresponding constraints $x_{i_0} \succ^{b_1} y_{i_1}$, $x_{i_1} \succ^{b_2} y_{i_2}$, ..., $x_{i_{k\ 2}} \succ^{b_{k\ 1}} y_{i_{k\ 1}}$, $x_{i_{k\ 1}} \succ^{b_k} y_{i_k}$ with $i_0 = i_k = i$. Summing these constraints we get $x_{i_0} + \cdots + x_{i_{k\ 1}} > y_{i_0} + \cdots + y_{i_{k\ 1}}$.

We can assume without loss of generality that there are no repeated indices among $i_0, \ldots, i_{k-1}$. Hence we obtain the required result taking coefficients $a_{i_0}, \ldots, a_{i_{k\ 1}}$ equal to one and all others equal to zero.

If there were a repeated index $i_\ell = i_{\ell\ } = i'$ then the sequence would be of the form $x^0_{i_0} \succ x^1_{i_1} \succ \cdots \succ x^\ell_{i_\ell} \succ \cdots \succ x^\ell_{i_\ell\ } \succ \cdots \succ x^k_{i_k}$. At least one of the shorter sequences: that starting and ending in argument position i without the segment from i' to i', or that starting and ending in argument position i' must contain a strict relation and can be chosen instead.

Theorem 3 does indicate an efficient test for termination and it would seem to require checking all possible combinations of coefficients $a_i \in \{0, 1\}$. We show that the, easy to implement, condition of Equation (1) is a complete test for non-idempotent graphs.

Corollary 1 (detecting ranking functions). *A size change graph $\mu(\bar{x}, \bar{y})$ has a ranking function if and only if*

$$\mu(\bar{x}, \bar{y}) \models \bigvee_{1 \le i \le n} x_i > y_i.$$

Proof. **(⇒)** Assume to the contrary that $\mu(\bar{x}, \bar{y})$ has a ranking function and $\mu(\bar{x}, \bar{y}) \wedge \neg \bigvee_i (x_i > y_i)$ has a solution. So $\mu(\bar{x}, \bar{y}) \wedge \bigwedge_i (x_i \le y_i)$ has a solution which implies that $\mu(\bar{x}, \bar{y}) \wedge (\Sigma_i a_i x_i \le \Sigma_i a_i y_i)$ has a solution for any coeeficients a_i. This is a contradiction because by Theorem 3, $\mu(\bar{x}, \bar{y}) \models \Sigma_i a_i x_i > \Sigma_i a_i y_i$ for some coefficients $a_i \in \{0, 1\}$. **(⇐)** If $\mu(\bar{x}, \bar{y}) \models \bigvee_i (x_i > y_i)$ then $\mu(\bar{x}, \bar{y}) \wedge \bar{x} = \bar{y}$ is not satisfiable and follow the proof of Theorem 2 up till the point when we get a simple cycle of constraints of the form depicted in Figure 3 (this part does not rely on idempotence). Summing these constraints gives a ranking function of the form $f(\bar{u}) = \Sigma_i a_i u_i$ with $a_i \in \{0, 1\}$.

4 Completeness for Monotonicity Constraints

Monotonicity constraints are more expressive than size change graphs. They may contain relations of the form $y_i \succ x_j$, going "up" in the graph representation,

and also "horizontal" *loop invariants* of the form $x_i \succ x_j$ or of the form $y_i \succ y_j$. An analyzer based on size change graphs cannot prove termination when the size of an argument is increasing in a loop towards an upper bound. The following generic example illustrates that size change termination is incomplete for monotonicity constraints.

Example 5. *Consider a program involving a loop of the form* `while (a1 < a2) a1 := a1 + 1`. *A corresponding loop description involves a monotonicity constraint of the form* $\mu(\bar{x}, \bar{y}) = x_1 < x_2, x_1 < y_1, x_2 = y_2$ *which is idempotent but not a size change graph. While the loop clearly terminates, neither* $f(u_1, u_2) = u_1$ *nor* $f(u_1, u_2) = u_2$ *is a ranking function. There is however a ranking function of the form* $f(u_1, u_2) = u_2 - u_1$.

We provide a completeness result for monotonicity constraints. If $\mu(\bar{x}, \bar{y})$ is an idempotent monotonicity constraint and has a ranking function, then it has a ranking function of the form $f(u_1, \ldots, u_n) = u_i$ or of the form $f(u_1, \ldots, u_n) = u_i - u_j$ for $1 \leq i, j \leq n$. If $\mu(\bar{x}, \bar{y})$ is not idempotent, then it has a linear ranking function.

The extra expressiveness of monotonicity constraints introduces several problems. First, a monotonicity constraint $\mu(\bar{x}, \bar{y})$ or one of its derivation constraints may be unsatisfiable and hence have no infinite derivations. For example, $\mu(\bar{x}, \bar{y}) = x_1 > x_2 \wedge y_1 \leq y_2$ is satisfiable but $\mu(\bar{x}, \bar{z}, \bar{y})$ is not. A second problem is illustrated in Figure 4. The constraint in Figure 4(a) is idempotent and has no infinite derivations because the value in its first argument is strictly decreasing in any such derivation. However there is no direct down arc in $\mu(\bar{x}, \bar{y})$. If we restrict attention to constraints with *balanced* invariants we avoid both problems. This is not a limitation for termination analysis as every postcondition of a loop is the precondition for the next time around.

Definition 8 (balanced constraint). *A monotonicity constraint* $\mu(\bar{x}, \bar{y})$ *is* balanced *if* $\mu(\bar{x}, \bar{y}) \models x_i \succ^b x_j \;\Leftrightarrow\; \mu(\bar{x}, \bar{y}) \models y_i \succ^b y_j$. *The* balanced extension $\mu_B(\bar{x}, \bar{y})$ *of* $\mu(\bar{x}, \bar{y})$ *is the smallest monotonicity constraint which includes* $\mu(\bar{x}, \bar{y})$ *and is balanced. We define* $bal(\mu)(\bar{x}, \bar{y})) = \mu(\bar{x}, \bar{y}) \wedge \{x_i \succ^b x_j \mid \mu(\bar{x}, \bar{y}) \models y_i \succ^b y_j\} \wedge \{y_i \succ^b y_j \mid \mu(\bar{x}, \bar{y}) \models x_i \succ^b x_j\}$. *Clearly* $\mu(\bar{x}, \bar{y})_B = bal^{4n(n-1)}(\mu)(\bar{x}, \bar{y}))$ *since there are at most* $4n(n-1)$ *constraints that can be added by bal. There are tighter bounds but that will suffice for our purposes.*

The balanced extension of the constraint in Figure 4(a) is shown in Figure 4(b) and with transitive closure in Figure 4(c). The downwards paths are now explicit. The balanced extension of a constraint is almost equivalent to the original, particularly in its powers. Figures 4(d) and (e) illustrate the similarity. For termination analysis we can restrict our attention to balanced extensions because of the following two Lemmata.

Lemma 5. *If monotonicity constraint* $\mu(\bar{x}, \bar{y})$ *is balanced, then either the derivation constraint* $\mu(\bar{x}^0, \ldots, \bar{x}^m)$ *is satisfiable for all* $m > 0$ *or* $\mu(\bar{x}, \bar{y})$ *is unsatisfiable.*

Proof. Assume that $\mu(\bar{x}, \bar{y})$ *is balanced and let* $m > 0$ *be such that* $\mu(\bar{x}^0, \ldots, \bar{x}^m)$ *is not satisfiable. Hence for some variable* x_i^k *(at level* k *in argument* i*) there*

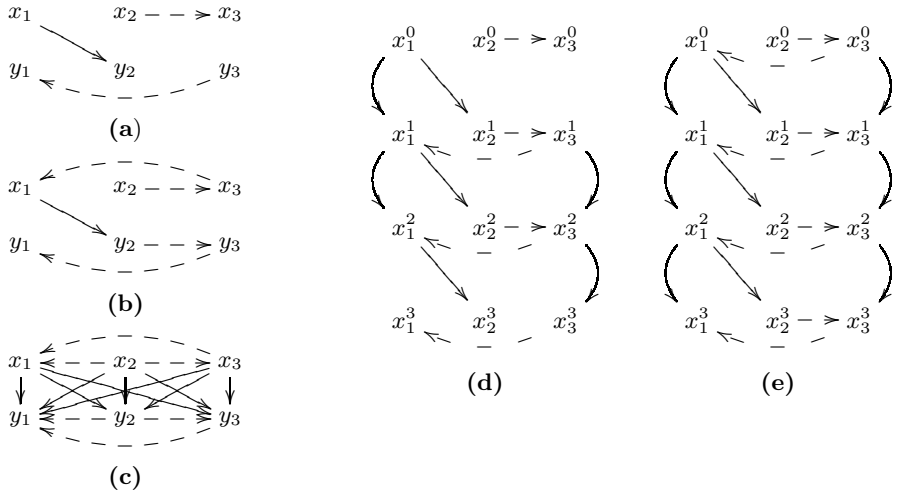


Fig. 4. (a) An unbalanced but idempotent constraint $\mu(\bar{x}, \bar{y})$, (b) its balanced extension $\mu_B(\bar{x}, \bar{y})$, and (c) $\mu_B(\bar{x}, \bar{y})$ indicating also transitive paths, (d) derivation constraint $\mu(\bar{x}^0, \bar{x}^1, \bar{x}^3, \bar{x}^4)$, and (e) $\mu_B(\bar{x}^0, \bar{x}^1, \bar{x}^3, \bar{x}^4)$

is a strict cycle of constraints implied by $\mu(\bar{x}^0, \ldots, \bar{x}^m)$ *of the form* $x_i^k \succ x_j^k \succ x_j^k \succ x_i^k \succ x_i^k$ *such that: (i) if* $k = k'$ *then* $i \neq j$, $i = i'$ *and* $j = j'$*; or (ii)* $k' = k \pm 1$*. Thus, as* $\mu(\bar{x}, \bar{y})$ *is balanced it must also imply a strict cycle of the form* $x_i \succ x_j \succ x_i$ *(if* $k = k'$*) or of the form* $x_i \succ y_j \succ y_j \succ x_i \succ x_i$ *(if* $k' = k+1$*) or of the form* $y_i \succ x_j \succ x_j \succ y_i \succ y_i$ *(if* $k' = k-1$*). Hence* $\mu(\bar{x}, \bar{y})$ *is not satisfiable.*

Lemma 6. *Monotonicity constraint* $\mu(\bar{x}, \bar{y})$ *has an infinite derivation if and only if its balanced extension* $\mu_B(\bar{x}, \bar{y})$ *has an infinite derivation.*

Proof. (sketch) **(⇐)** *Let* $\bar{b}^0 \sqsubseteq_{\mu_B} \bar{b}^1 \sqsubseteq_{\mu_B} \bar{b}^2 \sqsubseteq_{\mu_B} \cdots$ *be an infinite derivation for the balanced extension. Since* $\mu_B(\bar{x}, \bar{y}) \models \mu(\bar{x}, \bar{y})$*, the infinite derivation* $\bar{b}^0 \sqsubseteq_\mu \bar{b}^1 \sqsubseteq_\mu \bar{b}^2 \sqsubseteq_\mu \cdots$ *exists.* **(⇒)** *Let* $\bar{a}^0 \sqsubseteq_\mu \cdots \bar{a}^k \sqsubseteq_\mu \bar{a}^{k+1} \sqsubseteq_\mu \cdots$ *be an infinite derivation. One can show by induction that for any* k *and* ℓ *such that* $0 \leq \ell \leq k$*,* $\bar{a}^k \sqsubseteq_{bal^\ell(\mu)} \bar{a}^{k+1}$*. Now given that* $\mu_B(\bar{x}, \bar{y}) = bal^{4n(n-1)}(\mu(\bar{x}, \bar{y}))$ *we have the infinite derivation* $\bar{a}^{4n(n-1)} \sqsubseteq_{\mu_B} \bar{a}^{4n(n-1)+1} \sqsubseteq_{\mu_B} a^{4n(n-1)+2} \sqsubseteq_{\mu_B} \cdots$.

Theorem 4 (Completeness for idempotent monotonicity constraints). *Let* $\mu(\bar{x}, \bar{y})$ *be a balanced idempotent monotonicity constraint. If there exists any ranking function for* $\mu(\bar{x}, \bar{y})$ *then there exists one mapping to* $(\mathcal{N}, <)$ *of the form* $f(u_1, \ldots, u_n) = u_i$ *or of the form* $f(u_1, \ldots, u_n) = u_i - u_j$ *for some* $1 \leq i, j \leq n$*.*

The proof strategy for Theorem 4 is similar to that for Theorem 2. We will show that if there exists no ranking function for $\mu(\bar{x}, \bar{y})$ of the prescribed form then there is an infinite chain in $\sqsubseteq_\mu$ implying that there exists no ranking function of any form for $\mu(\bar{x}, \bar{y})$. We will need the following lemma.

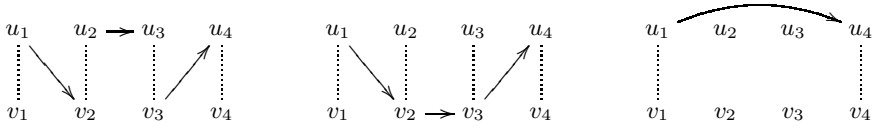


Fig. 5. Illustrating the proof of Lemma 7: **(a)** An alternating path from v_1 to v_4 with horizontal relation $u_2 > u_3$ **(b)** The graph is balanced so it contains also $v_2 > v_3$ **(c)** By transitivity it contains also $u_1 > u_4$ giving a shorter alternating path from v_1 to v_4

Lemma 7. *If $\mu(\bar{x}, \bar{y})$ is a satisfiable balanced idempotent monotonicity constraint then either: (a) $\mu(\bar{x}, \bar{y})$ implies a constraint of the form $x_i > y_i$ or of the form $y_i > x_i$ or (b) $\mu(\bar{x}, \bar{y}) \wedge \bar{x} = \bar{y}$ is satisfiable.*

Proof. *Let $\mu(\bar{x}, \bar{y})$ be a satisfiable balanced idempotent monotonicity constraint and assume that condition (b) does not hold. Given that $\mu(\bar{x}, \bar{y})$ is satisfiable and since $\mu(\bar{x}, \bar{y}) \wedge \bar{x} = \bar{y}$ is not satisfiable, it must be the case that there is a sequence of constraints from $\mu(\bar{x}, \bar{y})$ and from $\bar{x} = \bar{y}$ giving a contradiction. Without loss of generality, otherwise applying transitivity, we may assume the sequence is alternating and hence of the form:*

$$u_{i_1} \succ^{b_1} v_{i_2} = u_{i_2} \succ^{b_2} v_{i_3} \ldots = u_{i_k} \succ^{b_k} v_{i_1} = u_{i_1}$$

with $b_1 \vee \cdots \vee b_k = 1$. Given that $\mu(\bar{x}, \bar{y})$ is balanced, we may also assume without loss of generality that the sequence does not involve "horizontal" relations as these could be removed by transitivity. See Figure 5. It follows that all of the constraints $u_{i_j} \succ^{b_1} v_{i_{j+1}}$ are in the same direction (downwards or upwards). Namely, that for all $1 \leq j \leq k$ either $u_{i_j} \in \bar{x}$ and $v_{i_j} \in \bar{y}$ or $u_{i_j} \in \bar{y}$ and $v_{i_j} \in \bar{x}$. From idempotence using Lemma 1 we get that $\mu(\bar{x}, \bar{y})$ contains a constraint of the form $x_i > y_i$ or of the form $y_i > x_i$.

Lemma 8. *If $\mu(\bar{x}, \bar{y})$ is a balanced idempotent monotonicity constraint where for all i $\mu(\bar{x}, \bar{y}) \not\models x_i \succ y_i$, then for all i, j, $\mu(\bar{x}, \bar{y}) \not\models x_i \succ y_j$.*

Proof. Assume to the contrary that $\mu(\bar{x}, \bar{y}) \models x_i \succ y_j$. Then since $\mu(\bar{x}, \bar{y})$ is idempotent and balanced there must exist two constraints $x_i \succ y_{k_1}$ and $x_{k_1} \succ y_j$ implied by $\mu(\bar{x}, \bar{y})$ to ensure that $x_i \succ y_j$ is in the self composition. If $k_1 \in \{i, j\}$ we have a contradiction. So $k_1 \notin \{i, j\}$. Now consider the constraint $x_i \succ y_{k_1}$ implied by $\mu(\bar{x}, \bar{y})$. Using the same reasoning there must be constraints $x_i \succ y_{k_2}$ and $x_{k_2} \succ y_{k_1}$ implied by $\mu(\bar{x}, \bar{y})$. If $k_2 = i$ or $k_2 = k_1$ we immediately have a contradiction. If $k_2 = j$, then we have $x_j = x_{k_2} \succ y_{k_1}$ and $x_{k_1} \succ y_j$ implied by $\mu(\bar{x}, \bar{y})$ and hence by Lemma 1 also $x_j \succ y_j$ implied by $\mu(\bar{x}, \bar{y})$. Contradiction. Hence $k_2 \notin \{i, j, k_1\}$. We can now consider the constraint $x_i \succ y_{k_2}$ to generate $x_i \succ y_{k_3}$ and $x_{k_3} \succ y_{k_2}$, where $k_3 \notin \{i, j, k_1, k_2\}$. Following the same reasoning we eventually run out of argument positions. Contradiction.

Lemma 9. *For satisfiable, balanced and idempotent monotonicity constraint $\mu(\bar{x}, \bar{y})$, if for all $1 \leq i, j \leq n$, $\mu(\bar{x}, \bar{y}) \not\models x_i > y_i$ and $\mu(\bar{x}, \bar{y}) \not\models x_i \geq y_i \wedge x_i \geq x_j \wedge y_j > x_j$ then there is an infinite derivation using $\mu(\bar{x}, \bar{y})$.*

Proof. (Sketch)
Construction: *Let μ be the set of binary relations of the form $u \succ v$ implied by $\mu(\bar{x}, \bar{y})$. Let $U \subseteq \{1, \ldots, n\}$ be the set of arguments j which have a strict up arrow $y_j > x_j \in \mu$ and arguments i such that $j \in U$ and $u_i \succ v_j \in \mu$. Let $E = \{1, \ldots, n\} - U$ be the rest of the arguments. Let $V_E = \cup\{\{x_i, y_i\} \mid i \in E\}$ and $V_U = \cup\{\{x_i, y_i\} \mid i \in U\}$. We partition μ into three disjoint sets (conjunctions) of constraints: μ_U — the restriction of μ to the arguments U, μ_E — its restriction to the arguments in E, and μ_{EU} — the rest. So, $\mu(\bar{x}, \bar{y}) = \mu_U \wedge \mu_E \wedge \mu_{EU}$. This partitioning is given by: $\mu_E = \{u \succ v \in \mu \mid \{u, v\} \subseteq V_E\}$, $\mu_U = \{u \succ v \in \mu \mid \{u, v\} \subseteq V_U\}$ and $\mu_{EU} = \mu - \mu_E - \mu_U$.*

The "equals" part: *First we show that $\mu_E \wedge \bigwedge_{i \in E} x_i = y_i$ is satisfiable. By Lemma 7 either this holds or there exists $x_i > y_i$ in μ contradicting the assumption of the Lemma or $y_i > x_i$ in μ for $i \in E$ contradicting the definition of E. Hence there is a solution $\bar{a}_E$ of $\mu_E(\bar{x}, \bar{y}) \wedge \bigwedge_{i \in E} x_i = y_i$ and so $\bar{a}_E \sqsubseteq_{\mu_E} \bar{a}_E$.*

The "up" part: *Now let us consider μ_U. From the assumption of the Lemma there can be no $i \in U$ with $x_i > y_i \in \mu_U$. We show by the construction and the preconditions that there is no $i \in U$ with $x_i \geq y_i \in \mu_U$.*

First we show that for each $i \in U$ either $y_i > x_i$ or there exists k where $y_k > x_k$ and $x_i \succ x_k$ or $y_i \succ x_k$. The first case is straightforward from the definition of U. For the second, suppose i is added to U because j is already in U. Then either (a) $x_i \succ x_j$ (and $y_i \succ y_j$), (b) $x_i \succ y_j$ or (c) $y_i \succ x_j$. If $y_j > x_j$ then in all three cases we get the result. Otherwise by induction, we have $x_j \succ x_k$ or $y_j \succ x_k$. For case (a) if $x_j \succ x_k$ then we have by transitivity $x_i \succ x_k$ or if $y_j \succ x_k$ we have $y_i \succ x_k$. For case (b) if $x_j \succ x_k$ then by balance we have that $y_j \succ y_k$ and by transitivity ($x_i \succ y_j, y_j \succ y_k, y_k > x_k$) we have $x_i \succ x_k$, or if $y_j \succ x_k$ then by transitivity we have $x_i \succ x_k$. For (c) if $x_j \succ x_k$ then by transitivity we have $y_i \succ x_k$, and if $y_j \succ x_k$ we have by idempotence that $y_i \succ x_k$.

Suppose that $x_i \geq y_i$ for some $i \in U$. Then for some j with $y_j > x_j$ we have either $x_i \succ x_j$ or $y_i \succ x_j$. In the first case this contradicts the preconditions of the lemma. In the second case since $x_i \geq y_i$ and $y_i \succ x_j$ by transitivity we have $x_i \succ x_j$ again contradicting the preconditions of the lemma. So we have that there are no directly down arcs in U. By Lemma 8 we have there are no downwards arcs (direct or indirect) amongst arguments in U (E cannot be involved since there are no arcs from arguments in E to arguments in U).

Ordering the "up" part: *We now partition U into (disjoint) sets $U = U_1 \cup \cdots \cup U_l$ where for each $1 \leq k \leq l$ and $\{i, j\} \subseteq U_k$ we have $\mu_U \models x_i = x_j$ and for each $1 \leq k_1 < k_2 \leq l$, $i_1 \in U_{k_1}$ and $i_2 \in U_{k_2}$ we do not have $\mu_U \models x_{i_1} \succ x_{i_2}$. This is possible since μ is satisfiable. This provides a total order on equivalence classes of arguments such that variables in U_k are not constrained from above by any arguments in E or $U_1 \cup \cdots \cup U_{k-1}$.*

An infinite derivation: *We can now build an infinite derivation for μ. To build $\bar{a}^0$ set each position in E to the value in $\bar{a}_E$. Then for U_1 set all positions $j_1 \in U_1$ to the least integer satisfying all the constraints with respect to arguments in E.*

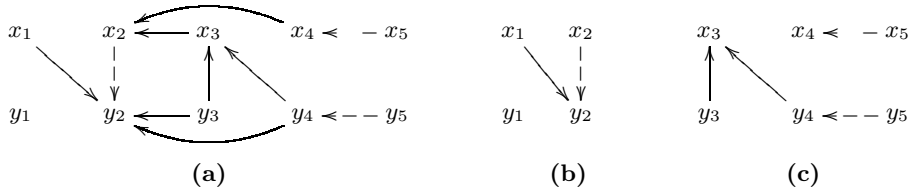


Fig. 6. The monotonicity constraints from Example 6: **(a)** the monotonicity constraint $\mu(\bar{x}, \bar{y})$, **(b)** the "equals" part μ_E, and **(c)** the "up" part μ_U

For $k = 2, \ldots l$ set all positions j_k to the least integer satisfying all constraints with respect to arguments in $E \cup U_1 \cup \cdots \cup U_{k-1}$.

To build $\bar{a}^{k+1}$ from a^k is similar but also taking into account all (lower bounding) constraints with respect to $\bar{a}^k$.

Example 6. *Consider the satisfiable balanced idempotent monotonicity constraint depicted in Figure 6***(a)**. *Building $U = \{3, 4, 5\}$: $3 \in U$ because of the constraint $y_3 > x_3$, $4 \in U$ because of 3 and the constraint $y_4 \geq x_3$ and $5 \in U$ because of 4 and the constraint $x_5 \geq x_4$. The remaining indices are $E = \{1, 2\}$. Partitioning U: we can take $U_1 = \{3\}$, $U_2 = \{4\}$ and $U_3 = \{5\}$ (U_3 must be last in the ordering). The constraints μ_E and μ_U are depicted as Figures 6***(b)** and **(c)**. *The constraint $\mu_{EU} = \{u > v \mid u \in \{x_3, x_4, x_5, y_3, y_4, y_5\}, v \in \{x_2, y_2\}\}$.*

We build an infinite derivation as follows. Pick a solution for $\mu_E \wedge x_1 = y_1 \wedge x_2 = y_2$, say $x_1 = y_1 = 1$, $x_2 = y_2 = 0$. $\bar{a}_E = (1, 0)$. Create $\bar{a}_0$ starting from $\bar{a}_E$, and filling in the argument positions in U_1, U_2, and U_3 with the least value satisfying constraints in filled in positions. Since there are no arcs from an argument position to an unfilled position this is always possible. We find $\bar{a}_0 = (1, 0, 1, 1, 1)$, $\bar{a}_1 = (1, 0, 2, 2, 2)$, $\bar{a}_2 = (1, 0, 3, 3, 3), \ldots$

Proof. (of Theorem 4) If $\mu(\bar{x}, \bar{y})$ is unsatisfiable then any ranking function is suitable (in particular one the form required by the theorem). Otherwise the conditions of Lemma 9 hold and since there exists a ranking function there can be no infinite derivation. Hence either (a) for some i, $\mu(\bar{x}, \bar{y}) \models x_i > y_i$ and the function $f(\bar{u}) = u_i$ is thus a ranking function, or (b) for some i, j, $\mu(\bar{x}, \bar{y}) \models x_i \geq y_i \wedge x_i \geq x_j \wedge y_j > x_j$ from which it follows that $f(\bar{u}) = u_i - u_j$ is a ranking function.

Theorem 5 (Completeness for arbitrary monotonicity constraints). *Let $\mu(\bar{x}, \bar{y})$ be a balanced monotonicity constraint (not necessarily idempotent). If there exists any ranking function for $\mu(\bar{x}, \bar{y})$ then there exists a linear ranking function for $\mu(\bar{x}, \bar{y})$.*

Proof. Assume balanced $\mu(\bar{x}, \bar{y})$ has a ranking function. Assume $\mu(\bar{x}, \bar{y})$ is satisfiable otherwise the result is trivial. It follows that $\mu^k(\bar{x}, \bar{y})$ is satisfiable, balanced and has a ranking function for any positive k. By Lemma 4 there exists a k such that $\mu^k(\bar{x}, \bar{y})$ is idempotent and hence by Theorem 4, $\mu^k(\bar{x}, \bar{y})$ has a ranking

function of the form $f(u_1, \ldots, u_n) = u_i$ or of the form $f(u_1, \ldots, u_n) = u_i - u_j$. If the first case then the proof is identical to that of Theorem 3. If the second case then $\mu^k(\bar{x}, \bar{y}) \models x_i \geq y_i \wedge x_i \geq x_j \wedge y_j > x_j$ and similar to the proof of Theorem 3 the following two sequences of "down" and "up" constraints are implied by $\mu(\bar{x}, \bar{y})$:

$$\begin{array}{llllll} x_{i_0} \geq y_{i_1}, & x_{i_1} \geq y_{i_2}, & \ldots, & x_{i_{k\ 2}} \geq y_{i_{k\ 1}}, & x_{i_{k\ 1}} \geq y_{i_k} \\ y_{j_1} \succ x_{j_0}, & y_{j_2} \succ x_{j_1}, & \ldots, & y_{j_{k\ 1}} \succ x_{j_{k\ 2}}, & y_{j_k} \succ x_{j_{k\ 1}} \end{array}$$

where at least one of the inequalities in the second ("up") sequence is strict. Adding these inequalities pairwise we get the sequence:

$$x_{i_0} - x_{j_0} \succ y_{i_1} - y_{j_1}, \quad x_{i_1} - x_{j_1} \succ y_{i_2} - y_{j_2}, \quad \ldots, \quad x_{i_{k\ 1}} - x_{j_{k\ 1}} \succ y_{i_k} - y_{j_k}$$

with at least one strict inequality. Summing this sequence and observing that $i_0 = i_k = i$ and $j_0 = j_k = j$ we obtain $\Sigma_{\ell=1}^k (x_{i_\ell} - x_{j_\ell}) > \Sigma_{\ell=1}^k (y_{i_\ell} - y_{j_\ell})$ which is of the form $\Sigma a_i x_i > \Sigma a_i y_i$ with coefficients determined by the number of repetitions of the constraints in the two sequences. Positive coefficients originate from "downwards" constraints and negative coefficients from the "upwards". We take $f(\bar{u}) = \Sigma a_i u_i$.

We now show that $\mu(\bar{x}, \bar{y}) \models f(\bar{x}) \geq 0$. We have $\mu^k(\bar{x}, \bar{y}) \models x_i \geq x_j$ which implies that $\mu(\bar{x}, \bar{y}) \models x_i \geq x_j$ and from balance $\mu(\bar{x}, \bar{y}) \models y_i \geq y_j$. Recalling that $i = i_k$ and $j = j_k$ we have $\mu(\bar{x}, \bar{y}) \models y_{i_k} \geq y_{j_k}$ and $\mu(\bar{x}, \bar{y}) \models x_{i_k} \geq x_{j_k}$. From transitivity (with the last constraints in the "down" and "up" sequences) that $\mu(\bar{x}, \bar{y}) \models x_{i_{k\ 1}} \geq x_{j_{k\ 1}}$ and from balance $\mu(\bar{x}, \bar{y}) \models y_{i_{k\ 1}} \geq y_{j_{k\ 1}}$. In a similar way we obtain that $\mu(\bar{x}, \bar{y})$ implies the constraints $x_{i_\ell} \geq x_{j_\ell}$ for $\ell \in \{1, \ldots, k\}$. Summing these constraints gives $f(\bar{x}) = \Sigma_{\ell=1}^k (x_{i_\ell} - x_{j_\ell}) \geq 0$.

Example 7. *Consider the (balanced extension of) monotonicity constraint* $x_1 \geq y_2, x_2 \geq y_1, x_2 \geq x_3, y_3 > x_3$. *The "down" and "up" sequences from the proof of Theorem 5 are respectively* $x_1 \geq y_2, x_2 \geq y_1$ *and* $y_3 < x_3, y_3 < x_3$. *Summing these gives* $x_1 + x_2 - 2x_3 > y_1 + y_2 - 2y_3$. *A ranking function of the form* $f(\bar{u}) = u_1 + u_2 - 2u_3$ *exists. The constraints* $x_1 \geq x_3$ *and* $x_2 \geq x_3$ *imply that* $f(\bar{x}) = x_1 + x_2 - 2x_3 \geq 0$.

5 Conclusion

This paper makes two contributions. For size change graphs we establish that the termination test implemented in analyzers such as TermiLog and TerminWeb is complete for size change graphs and incomplete for monotonicity constraints. In particular there is no loss of precision when not checking for idempotence. For idempotent monotonicity constraints, we prove that if there exists any ranking function for a loop description then there exists one of a simple form: a single argument or the difference between two arguments is decreasing. Moreover, for loop descriptions which are not idempotent if there exists a ranking function then there exists one which is linear.

References

1. A. Brodsky and Y. Sagiv. Inference of monotonicity constraints in Datalog programs. In *Proceedings of the Eighth ACM SIGACT-SIGART-SIGMOD Symposium on Principles of Database Systems*, pages 190–199, 1989.
2. M. Codish and C. Taboch. A semantic basis for the termination analysis of logic programs. *The Journal of Logic Programming*, 41(1):103–123, 1999.
3. N. Dershowitz, N. Lindenstrauss, Y. Sagiv, and A. Serebrenik. A general framework for automatic termination analysis of logic programs. *Applicable Algebra in Engineering, Communication and Computing*, 12(1-2):117–156, 2001.
4. C. C. Frederiksen. A simple implementation of the size-change termination principle. Technical Report D-442, DIKU, Copenhagen University, Denmark, 2001.
5. G. Frobenius. Über endliche gruppen. *Sitzungsber. Preuss. Akad. Wiss. Berlin*, pages 163–194, 1895.
6. M. Gabbrielli and R. Giacobazzi. Goal independency and call patterns in the analysis of logic programs. In *Proceedings of the Ninth ACM Symposium on Applied Computing*, pages 394–399. ACM Press, 1994.
7. C. S. Lee, N. D. Jones, and A. M. Ben-Amram. The size-change principle for program termination. *ACM SIGPLAN Notices*, 36(3):81–92, 2001. Proceedings of POPL'01.
8. N. Lindenstrauss and Y. Sagiv. Automatic termination analysis of logic programs. In L. Naish, editor, *Proceedings of the Fourteenth International Conference on Logic Programming*, pages 63–77, Leuven, Belgium, 1997. The MIT Press.
9. N. Lindenstrauss, Y. Sagiv, and A. Serebrenik. Termilog: A system for checking termination of queries to logic programs. In O. Grumberg, editor, *CAV*, volume 1254 of *Lecture Notes in Computer Science*, pages 444–447. Springer, 1997.
10. A. Podelski and A. Rybalchenko. A complete method for the synthesis of linear ranking functions. In B. Steffen and G. Levi, editors, *Verification, Model Checking, and Abstract Interpretation, 5th International Conference*, volume 2937 of *Lecture Notes in Computer Science*, pages 239–251. Springer, 2004.
11. F. Ramsey. On a problem of formal logic. In *Proceedings London Mathematical Society*, volume 30, pages 264–286, 1930.
12. R. Thiemann and J. Giesl. Size-change termination for term rewriting. In *Proceedings of the 14th International Conference on Rewriting Techniques and Applications (RTA-03)*, volume 2706 of *Lecture Notes in Computer Science*, pages 264–278. Springer-Verlag, 2003.
13. D. Wahlstedt. Detecting termination using size-change in parameter values. Master's thesis, Göteborgs Universitet, 2000. `http://www.cs.chalmers.se/~davidw/`.

A Well-Founded Semantics with Disjunction

João Alcântara, Carlos Viegas Damásio, and Luís Moniz Pereira

Centro de Inteligência Artificial (CENTRIA),
Departamento de Informática, Universidade Nova de Lisboa,
2829-516 Caparica, Portugal
{jfla, cd, lmp}@di.fct.unl.pt

Abstract. In this paper we develop a new semantics for disjunctive logic programs, called *Well-Founded Semantics with Disjunction* (WFS_d), by resorting to a fixed point-based operator. Coinciding with the Well-Founded Semantics (WFS) for normal logic programs, our semantics is uniquely defined for every disjunctive logic program. By exploring examples, we show WFS_d does not agree with any other semantics we have studied, such as Brass and Dix's *D-WFS*, Przymusinski's Static, Baral *et al*'s *GDWFS*, Wang's *WFDS*, and van Gelder *et al*'s *SWFS*. Despite that, we ensure WFS_d is strictly stronger than *D-WFS* by guaranteing WFS_d allows the five, desirable, program transformations proposed by Brass and Dix: unfolding, elimination of tautologies and non-minimal rules, and positive and negative reduction.

1 Introduction

To be, or not to be: that is the question
William Shakespeare in Hamlet (III, I)

Not just Hamlet has been involved in disquisitions on disjunctions; since the early eighties the Logic Programming Community too has devoted great effort to provide reasonable answers to disjunctive forms of reasoning. The pioneer work in the field is [6], where Minker presents the *Generalized Closed World Assumption (GCWA)* to define model-theoretically the default negation in disjunctive theories. For Horn theories [12], *GCWA* reduces to the *Closed World Assumption*.

For a Horn theory extended to allow disjunction in the head of its rules, named positive-disjunctive logic program (*PDLP*), we can easily match its semantics to the set of *minimal Herbrand models* or to the *minimal model state*, which is the collection of all positive disjunctions holding in every minimal Herbrand Model. Both paths can equivalently yield the expected meaning. In [7], Minker and Rajasekar defined the operator $\mathcal{T}_P^s$, whose least fixed point represents the minimal model state of a *PDLP*. Regarding the path based on minimal Herbrand models, in [3], for *PDLP*s without function symbols, Fernández and Minker defined the operator $\mathcal{T}_P^M$ on sets of Herbrand interpretations whose least fixed point is the set of minimal Herbrand models of the program. Afterwards, in [14], the authors extended $\mathcal{T}_P^M$ in terms of $\mathcal{T}_P^{INT}$ in order to deal also with function symbols in such programs.

M. Gabbrielli and G. Gupta (Eds.): ICLP 2005, LNCS 3668, pp. 341–355, 2005.

Whereas the minimal Herbrand models (or minimal model state) are largely accepted as capturing the meaning of *PDLP*s, the same cannot be said for *disjunctive logic programs*, i.e. normal logic programs whose syntax allows disjunction in the head of their rules. Indeed, as well as for normal logic programs, there are two major approaches when considering the semantics of disjunctive logic programs: one extends stable models semantics [4], and the other, more skeptical, extends well-founded models semantics (*WFS*) [18].

Unlike stable model semantics, *WFS* is defined for every normal logic program. Indeed, for each there is a unique *WFM* intended to capture its meaning or declarative semantics. This is attained by abandoning classical two-valued interpretations and permitting partial interpretations, wherein a third value is introduced to represent undefinedness. The result is an elegant semantics, advantageously compared, from a computational viewpoint, to stable models.

When moving to disjunctive logic programming one would hope that such properties be preserved. Notwithstanding, given the numerous proposals (for a non-exhaustive list, see [13,1,9,2,19]), we would agree that a well-founded semantics for disjunctive logic programs representing their intended meaning is not consensual. The issues arise from the start in identifying which meaning someone assigns to disjunction in logic programming: should it be exclusive? inclusive? or should it be the so-called epistemic disjunction, designed to be exclusive as far as possible? This barrier surmounted, which properties should endorse the relation between disjunction and default negation? Such concerns are not only formidable, but they involve aspects so diverse that even the choice of a touchstone to appraise disjunctive well-founded semantics is hardly achievable.

Prospectively, we propound that a good well-founded semantics for disjunctive logic programs should, at least:

- Coincide with *WFS* on normal logic programs.
- Be uniquely defined for every disjunctive logic program.
- Be definable by both a fixed point operator and a model-theoretically one.
- Comply with the five program transformations proposed by Brass and Dix: unfolding, elimination of tautologies and non-minimal rules, positive and negative reduction. In other words, their *D-WFS* should be the weakest well-founded semantics for disjunctive logic programs.
- Allow further extensions to assimilate explicit negation, both in its explosive as well as in its paraconsistent varieties.

In our studies about the applicability of logic programming as a suitable tool to represent knowledge and reasoning we have observed that by generalising to a set of interpretations the Ω_P operator [8], used to define *WFS*, and by positing a new domain to deal specifically with default negation, we are able to define a new well-founded semantics for disjunctive logic programs. This semantics, dubbed *Well-Founded Semantics with Disjunction* (*WFS*$_d$), shows to be worthy enough and to satisfy the conditions just enrolled.

The remainder of this work is spent as follows. First, by recalling in Section 2 how to determine the minimal models of a *PDLP* via the $\mathcal{T}_P^{INT}$ operator [14]. We

then reserve Section 3 to present our main contribution: the definition of WFS_d established by a fixed point operator. Section 4 is devoted to guarantee, by means of examples, that WFS_d generally does not agree with any of the semantics for disjunctive logic programs we have studied. Going into more detail, we then show that WFS_d is strictly stronger than *D-WFS* [2]. The paper terminates with conclusions and future work.

2 Minimal Models of Positive-Disjunctive Logic Programs

In this section we recall [14] in showing how to capture the minimal models of a *PDLP* via the fixed point operator $\mathcal{T}_P^{INT}$. As the authors emphasise, although non-continuous it converges to its least fixed point in at most ω iterations. We also present basic concepts employed throughout this paper.

Let us now present disjunctive (and positive-disjunctive) logic programs in more formal terms. Given a first order language $\mathfrak{L}$, a disjunctive logic program is a set of logical inference rules r of the form

$$r = A_1 \vee \ldots \vee A_l \leftarrow B_1 \wedge \ldots \wedge B_m \wedge not\ C_1 \ldots \wedge not\ C_n \tag{1}$$

in which the A_i, B_j, C_k are atoms in $\mathfrak{L}$, and $l, m, n \in \mathbb{N}$. The set of all *ground instances* of the rules in P is denoted by $gnd(P)$. If $m = n = 0$ the rule is said a *fact*, and the symbol $\leftarrow$ may be dropped.

In particular, P is defined as a positive-disjunctive logic program (*PDLP*) if $\forall r \in P$ we have that $k = 0$, i.e. P is deprived of default negation.

By *Herbrand base* H_{B_P} *of a disjunctive logic program* P, we mean the set of all ground atoms over the language of P. In order to study the meaning of these programs, the notion of interpretation is introduced:

Definition 1 (Interpretations and coins). *Given a program P, we define the following terms:*

- *By* Herbrand interpretation *(or just interpretation) we mean a set $I \subseteq H_{B_P}$. The set of all interpretations is denoted by $\mathcal{H}_{\mathcal{I}_P}$.*
- *For conciseness, by* coin *we mean a collection of interpretations. In particular, $\mathcal{H}_{\mathcal{I}_P}$ is a coin.*

When dealing with a *coin* $\mathcal{I}$, we will frequently refer to the operators *min*, *exp*, and *imp* for $\mathcal{I}$:

$min(\mathcal{I}) = \{I \in \mathcal{I} \mid \nexists J \in \mathcal{I} : J \subset I\}$,
$exp(\mathcal{I}) = \{I \in \mathcal{H}_{\mathcal{I}_\mathcal{P}} \mid \exists J \in \mathcal{I} : J \subseteq I\}$,
$imp(\mathcal{I}) = \{I \in \mathcal{H}_{\mathcal{I}_\mathcal{P}} \mid \exists J \in \mathcal{I} : I \subseteq J\}$.

A coin $\mathcal{I}$ is called *expanded* if, for each interpretation $I \in \mathcal{I}$, $\mathcal{I}$ contains all interpretations that are supersets of I, i.e. $\mathcal{I} = exp(\mathcal{I})$. Similarly, $\mathcal{I}$ is called *self-contained* if, for each interpretation $I \in \mathcal{I}$, $\mathcal{I}$ contains all interpretations

that are subsets of I, i.e. $\mathcal{I} = imp(\mathcal{I})$. In order to illustrate applications of these operators, let us choose the Herbrand base $H_{B_P} = \{a, b, c\}$ and the coin $\mathcal{I} = \{\{a\}, \{a, b\}, \{b, c\}\}$; we have:

$min(\mathcal{I}) = \{\{a\}, \{b, c\}\}$,
$exp(\mathcal{I}) = \mathcal{I} \cup \{\{a, c\}, \{a, b, c\}\}$,
$imp(\mathcal{I}) = \mathcal{I} \cup \{\{\}, \{b\}, \{c\}\}$.

For *PDLP*s, we can distinguish the models as per below:

Definition 2 (Satisfaction and model). *An interpretation I satisfies a rule $r = A_1 \vee \ldots \vee A_l \leftarrow B_1, \ldots, B_m$ of a PDLP P iff it holds that $\{B_i \mid 1 \leq i \leq m\} \subseteq I$ implies that there exists $A_j \in I$ such that $1 \leq j \leq l$. I is a model of P iff for every $r \in P$, I satisfies r. The set of all models of P is denoted by $\mathcal{M}_{\mathcal{OD}}(P)$.*

We say a model M of P is *minimal* if there is no model M' for which $M' \subset M$. In order to reason with *PDLP*s, we consider the set $\mathcal{MM}_P$ of all minimal models of P. For Horn theories, the unique minimal (least) model can be obtained by iteratively applying an immediate consequences operator introduced by van Emden and Kowalski [17]. This operator can be generalised to deal with *PDLP*s:

Definition 3 (T_P mapping). *Let P be a PDLP and I be an interpretation. Then the immediate consequences mapping $T_P : \mathcal{H}_{\mathcal{I}_P} \rightarrow 2^{\mathcal{H}_{\mathcal{I}_P}}$ is defined as follows:*

$T_P(I) = \{J \mid J \cap \{a_1, \ldots, a_l\} \neq \emptyset$ for every $r = a_1 \vee \ldots \vee a_l \leftarrow b_1, \ldots, b_m \in P$ such that $\{b_1, \ldots, b_m\} \subseteq I, I \subseteq J\}$.

Thus, from each single interpretation, various interpretations may be derived through the T_P mapping. Then the $\mathcal{T}_P^{INT}$ operator can be defined by applying T_P to each interpretation of a coin. Formally:

Definition 4 ($\mathcal{T}_P^{INT}$ Operator). *Let P be a PDLP and $\mathcal{I}$ be a coin. We define $\mathcal{T}_P^{INT} : 2^{\mathcal{H}_{\mathcal{I}_P}} \rightarrow 2^{\mathcal{H}_{\mathcal{I}_P}}$ such that*

$$\mathcal{T}_P^{INT}(\mathcal{I}) = \bigcup_{I \in \mathcal{I}} T_P(I)$$

In their studies to define an immediate consequences operator for *PDLP*s, Fernández and Minker [3] explored the subsumption relation $\sqsubseteq$ established for coins $\mathcal{I}, \mathcal{J} \in 2^{\mathcal{H}_{\mathcal{I}_P}}$:

$$\mathcal{I} \sqsubseteq \mathcal{J} \text{ iff } \forall J \in \mathcal{J} : \exists I \in \mathcal{I} : I \subseteq J$$

In fact, $\sqsubseteq$ corresponds to the Hoare ordering (see [15]) associated with the partially ordered set $(\mathcal{H}_{\mathcal{I}_P}, \subseteq)$. For example, letting $\mathcal{I} = \{\{a\}, \{c\}\}$ and $\mathcal{J} = \{\{a\}, \{a, b\}, \{b, c\}\}$, we have $\mathcal{I} \sqsubseteq \mathcal{J}$.

They also define an equivalence class with respect to $\sqsubseteq$, denoted by $\equiv_{\sqsubseteq}$, when coins subsume one another, i.e.

$$\mathcal{I} \equiv_{\sqsubseteq} \mathcal{J} \text{ iff } \mathcal{I} \sqsubseteq \mathcal{J} \text{ and } \mathcal{J} \sqsubseteq \mathcal{I}$$

Nonetheless, as it stands, the characterisation of $\mathcal{T}_P^{INT}$ may present some problems. The reason is that $\sqsubseteq$ on $2^{\mathcal{H}_{\mathcal{I}_P}}$ is reflexive and transitive, but not anti-symmetric; the pair $\mathcal{O} =< 2^{\mathcal{H}_{\mathcal{I}_P}}, \sqsubseteq >$ is not a (complete) lattice. This jeopardises making use of the results supported by the Knaster-Tarski theorem [16].

In order to overcome this, extending Fernández and Minker's work [3], Seipel *et al* [14] proposed the lattice $\mathcal{O}_{exp} =< 2^{\mathcal{H}_{\mathcal{I}_\mathcal{P}}}_{exp}, \sqsubseteq >$, in which $2^{\mathcal{H}_{\mathcal{I}_\mathcal{P}}}_{exp}$ corresponds to the set of all expanded coins. In $\mathcal{O}_{exp}$ the relation $\sqsubseteq$ reduces to superset inclusion, i.e. for every $\mathcal{I}, \mathcal{J} \in 2^{\mathcal{H}_{\mathcal{I}_P}}$ it holds that

$$\mathcal{I} \sqsubseteq \mathcal{J} \text{ iff } exp(\mathcal{J}) \subseteq exp(\mathcal{I}) \tag{2}$$

This suffices to ensure $\mathcal{O}_{exp}$ is *complete* and that the least upper bound and the greatest lower bound of a set $X \subseteq 2^{\mathcal{H}_{\mathcal{I}_\mathcal{P}}}_{exp}$ of expanded coins are respectively given by

$$lub_{exp}(X) = \bigcap_{\mathcal{I} \in X} \mathcal{I}, \qquad glb_{exp}(X) = \bigcup_{\mathcal{I} \in X} \mathcal{I}$$

The bottom element of $\mathcal{O}_{exp}$ is the coin $\bot_{exp} = \mathcal{H}_{\mathcal{I}_P}$. Notice that both the union and the intersection of a set X of expanded interpretations is an expanded interpretation.

The $\mathcal{T}_P^{INT}$ operator is guaranteed monotonic with respect to $\sqsubseteq$, i.e. if $\mathcal{I}_1 \sqsubseteq \mathcal{I}_2$ then $\mathcal{T}_P^{INT}(\mathcal{I}_1) \sqsubseteq \mathcal{T}_P^{INT}(\mathcal{I}_2)$, so, according to the Knaster-Tarski theorem, a least fixed point of $\mathcal{T}_P^{INT}$ on $< 2^{\mathcal{H}_{\mathcal{I}_\mathcal{P}}}_{exp}, \sqsubseteq >$ is guaranteed to exist. Hence we can employ the *ordinal powers* of $\mathcal{T}_P^{INT}$ defined below to reach this least fixed point:

Definition 5 (Ordinal powers of $\mathcal{T}_P^{INT}$). *Let P be a PDLP.*

1. *For $\mathcal{I} \in 2^{\mathcal{H}_{\mathcal{I}_\mathcal{P}}}_{exp}$, the ordinal powers $\mathcal{T}_P^{INT\ \uparrow\alpha}$ are defined by*

   $\mathcal{T}_P^{INT\ \uparrow 0} = \bot_{exp} = \mathcal{H}_{\mathcal{I}_P}$.

   $\mathcal{T}_P^{INT\ \uparrow\alpha} = \mathcal{T}_P^{INT}(\mathcal{T}_P^{INT\ \uparrow\alpha-1})$, *for successor ordinal* α.

   $\mathcal{T}_P^{INT\ \uparrow\alpha} = lub_{exp}(\{\mathcal{T}_P^{INT\ \uparrow\beta} \mid \beta < \alpha\}) = \bigcap_{\beta<\alpha} \mathcal{T}_P^{INT\ \uparrow\beta}$, *for limit ordinal* α.

2. *The least fixed point of $\mathcal{T}_P^{INT}$, denoted by $\mathcal{T}_P^{coin}$, is given by $\mathcal{T}_P^{INT\ \uparrow\alpha}$, in which α is the smallest ordinal such that $\mathcal{T}_P^{INT\ \uparrow\alpha} = \mathcal{T}_P^{INT\ \uparrow\alpha+1}$.*

The authors show in [14] that even though $\mathcal{T}_P^{INT}$ is *not continuous*, it reaches its least fixed point in at most ω iterations. They also guarantee that $\mathcal{T}_P^{INT}$ generalises the characterisation of the minimal model semantics that Fernández and Minker obtained for disjunctive deductive databases to arbitrary *PDLP*s, namely that

$$\mathcal{MM}_P \equiv_{\sqsubseteq} exp(\mathcal{MM}_P) = \mathcal{T}_P^{coin} = \mathcal{T}_P^{INT\ \uparrow\omega} \tag{3}$$

In the next section, we will investigate how to explore the results exhibited here in order to define a well-founded semantics for disjunctive logic programs.

3 A Fixed Point Based Definition of WFS_d

The definition of an adequate semantics for default negation in disjunctive logic programs has puzzled many researchers [10,13,1,11,9,2,19] employed to define well-founded semantics [8] to a set of interpretations, we propose a new disjunctive version of the well-founded semantics. A fixed point-based definition is then presented to characterise it.

In normal logic programs, well-founded semantics may be defined by introducing partial interpretations, which are a pair of interpretations $I^p =< I^t, I^{tu} >$; the I^t component is designed to determine what is true in I^p, whilst I^{tu} what is "non-false", i.e. true or undefined, in I^p. As is suggestive, I^t is employed to evaluate atoms, and I^{tu} default negated atoms, the later being true if and only if their atoms are not in I^{tu}.

Our idea is to exploit this notion of pair of interpretations to characterise well-founded semantics for disjunctive logic programs, but now considering the pair $\mathcal{I}^p =< \mathcal{I}^t, \mathcal{I}^{tu} >$, in which both $\mathcal{I}^t$ and $\mathcal{I}^{tu}$ are coins. Pairs of coins will be used to evaluate disjunctive clauses as follows:

A *disjunctive ground clause* $a_1 \vee \ldots \vee a_m \vee not\ b_1 \vee \ldots \vee not\ b_n$ is true in $\mathcal{I}^p =< \mathcal{I}^t, \mathcal{I}^{tu} >$ iff

D_1 For each $I_1 \in \mathcal{I}^t$, $\exists a_i \in \{a_1, \ldots, a_m\}$, such that $a_i \in I_1$
or
D_2 For each $I_2 \in \mathcal{I}^{tu}$, $\exists b_j \in \{b_1, \ldots, b_n\}$, such that $b_j \notin I_2$.

As we shall unveil, conditions D_1 and D_2 generalise the evaluation mechanism of positive-disjunctive and negative-disjunctive ground clauses in *PDLP*s. Condition D_1 is motivated by the way positive-disjunctive ground clauses are characterised in terms of minimal models of *PDLP*, as Minker proved [6]:

Theorem 1. *Let P be a PDLP. A positive-disjunctive ground clause $a_1 \vee \ldots \vee a_n$ is a logical consequence of P if and only if $a_1 \vee \ldots \vee a_n$ is* true *in every minimal model of P.*

Then we have the following result, linking condition D_1 to the notion of logical consequence of a positive-disjunctive logic program:

Proposition 1. *Let P be a PDLP, $a_1 \vee \ldots \vee a_n$ be a positive-disjunctive ground clause such that $a_i \in H_{B_P}$, $1 \leq i \leq n$, and $exp(\mathcal{M}^t) = exp(\mathcal{M}\mathcal{M}_P)$[1]. We have*

$$a_1 \vee \ldots \vee a_n \text{ is a logical consequence of } P$$

iff

$$a_1 \vee \ldots \vee a_n \text{ is true in } \mathcal{M}^p =< \mathcal{M}^t, \mathcal{M}^{tu} >$$

in which $\mathcal{M}^{tu}$ is any set of interpretations.

Regarding D_2, it provides a mechanism to evaluate negative-disjunctive ground clauses based on a principle known as Extended Generalised Closed-World Assumption (*EGCWA*), that we recall below:

[1] Note that the theorem holds for $\mathcal{M}^t = \mathcal{M}\mathcal{M}_P$.

Definition 6 (Semantic definition of EGCWA). *[5] Let P be a PDLP and $\mathcal{C}$ be a negative-disjunctive ground clause not $a_1 \vee \ldots \vee$ not a_n such that $a_i \in H_{B_P}$, $1 \leq i \leq n$. Then the* model-theoretic definition of *EGCWA is given by*

$$EGCWA(P) = \{\mathcal{C} \mid \mathcal{C} \text{ is true in all minimal models of } P\}.$$

The relation between *EGCWA* and how to evaluate negative-disjunctive ground clauses is described by Proposition 2:

Proposition 2. *Let P be a PDLP, $\mathcal{C}$ be a negative-disjunctive ground clause not $A_1 \vee \ldots \vee$ not A_n such that $A_i \in H_{B_P}$, $1 \leq i \leq n$, $imp(\mathcal{M}^{tu}) = imp(\mathcal{MM}_P)$, and $\mathcal{MM}_P$ is the set of minimal models of P. We have*

$$not\ A_1 \vee \ldots \vee not\ A_n \in EGCWA(P)$$

iff

$$not\ A_1 \vee \ldots \vee not\ A_n \text{ is true in } \mathcal{M}^p =< \mathcal{M}^t, \mathcal{M}^{tu} >$$

in which $\mathcal{M}^t$ is any coin.

Proposition 2 ensures that D_2 corresponds to evaluating negative-disjunctive ground clauses according to *EGCWA*. It emphasises as well, like in Proposition 1, the role played by minimal models in determining what follows from a *PDLP*. So, if we wish to provide meaning for the more complex syntax of disjunctive logic programs, determining them is fundamental.

As expounded in Section 2, the ordinal powers for the $\mathcal{T}_P^{INT}$ operator on the lattice $\mathcal{O}_{exp} =< 2^{\mathcal{H}_{\mathcal{I}_\mathcal{P}}}_{exp}, \sqsubseteq >$ are suitable to express what is true in a disjunctive positive logic program. Ideally, a similar mathematical equipment would give us what is false by default in a *PDLP*. The problem is that the relation $\sqsubseteq$ does not preserve the notion of falsity by default, as shown below:

Example 1. Let $\mathcal{I}_1^p =< \mathcal{I}_1^t, \mathcal{I}_1^{tu} >$ and $\mathcal{I}_2^p =< \mathcal{I}_2^t, \mathcal{I}_2^{tu} >$, in which $\mathcal{I}_1^{tu} = \{\{a\}\}$, $\mathcal{I}_2^{tu} = \{\{a\}, \{a, b\}\}$, and $\mathcal{I}_1^t$ and $\mathcal{I}_2^t$ are arbitrary coins.

If $\sqsubseteq$ was suitable to deal with default negation, equivalent elements of $2^{\mathcal{H}_{\mathcal{I}_P}}$ according to $\sqsubseteq$ would correspond to equivalent evaluations for default negated atoms. That is not the case in Example 1, wherein $\mathcal{I}_1^{tu} \equiv_\sqsubseteq \mathcal{I}_2^{tu}$, but intuitively b is false by default in $\mathcal{I}_1^p$, whilst it is not false by default in $\mathcal{I}_2^p$.

In conclusion, the notion of equivalence suggested by $\sqsubseteq$ does not correspond generally to equivalent evaluations for default negated atoms, we thence introduce the new relation $\preceq$, which corresponds to the Smyth ordering [15] associated with the partially ordered set $(\mathcal{H}_{\mathcal{I}_P}, \subseteq)$ amongst coins:

$$\mathcal{I} \preceq \mathcal{J} \text{ iff } \forall I \in \mathcal{I}, \exists J \in \mathcal{J} : I \subseteq J \tag{4}$$

The relation $\preceq$ works as the dual of $\sqsubseteq$. Like for $\sqsubseteq$, the relation $\preceq$ on $2^{\mathcal{H}_{\mathcal{I}_P}}$ does not represent a partial ordering because the anti-symmetry property does not hold. In any case, considering the domain $2^{\mathcal{H}_{\mathcal{I}_\mathcal{P}}}_{imp}$, defined as the set of all coins $\mathcal{I} \in 2^{\mathcal{H}_{\mathcal{I}_P}}$ such that $imp(\mathcal{I}) = \mathcal{I}$, we know that $\mathcal{O}_{min} =< 2^{\mathcal{H}_{\mathcal{I}_\mathcal{P}}}_{imp}, \preceq >$ is a complete lattice. Let us start by guaranteeing the following result:

Theorem 2. *For every $\mathcal{I}, \mathcal{J} \in 2^{\mathcal{H}_{\mathcal{I}_P}}$ it holds that $\mathcal{I} \preceq \mathcal{J}$ iff $imp(\mathcal{I}) \subseteq imp(\mathcal{J})$.*

Hence if $\mathcal{I}, \mathcal{J} \in 2^{\mathcal{H}_{\mathcal{I}_\mathcal{P}}}_{imp}$ we have $\mathcal{I} \preceq \mathcal{J}$ iff $I \subseteq J$. This is enough to guarantee that $\mathcal{O}_{imp} =< 2^{\mathcal{H}_{\mathcal{I}_\mathcal{P}}}_{imp}, \preceq >$ is a complete lattice in which $\preceq$ reduces to subset inclusion, whereas the least upper bound and the greatest lower bound of a set $X \subseteq 2^{\mathcal{H}_{\mathcal{I}_\mathcal{P}}}_{imp}$ are given respectively by

$$lub_{imp}(X) = \bigcup_{\mathcal{I} \in X} \mathcal{I}, \qquad glb_{imp}(X) = \bigcap_{\mathcal{I} \in X} \mathcal{I}$$

The bottom element of $\mathcal{O}_{imp}$ is the coin $\bot_{imp} = \{\{\}\}$, and the top one is $\top_{imp} = \mathcal{H}_{\mathcal{I}_P}$. Notice that both the union and the intersection of a set X of coins $\mathcal{I} \in 2^{\mathcal{H}_{\mathcal{I}_\mathcal{P}}}_{imp}$ is also an element of $2^{\mathcal{H}_{\mathcal{I}_\mathcal{P}}}_{imp}$.

We are going to show in the sequel that the lattice $\mathcal{O}_{imp}$ complies with our objective of providing a suitable means to interpret default negation in a disjunctive logic program. For the moment, let us reconsider Example 1:

Example 2. Let $\mathcal{O}_{imp} =< \{\{a, b\}, \{a\}, \{b\}, \{\}\}, \preceq >$, and both $\mathcal{I}^{tu}_1 = \{\{a\}\}$ and $\mathcal{I}^{tu}_2 = \{\{a\}, \{a, b\}\}$ be coins.

As we can see, $\mathcal{I}^{tu}_1$ and $\mathcal{I}^{tu}_2$ are not equivalent according to $\preceq$, reflecting that b is false by default in $\mathcal{I}^{p}_1$, whilst it is not false by default in $\mathcal{I}^{p}_2$.

We prove now that in $2^{\mathcal{H}_{\mathcal{I}_\mathcal{P}}}_{imp}$ the truth-value assigned to a negative-disjunctive clause is preserved in the equivalence class created by $\preceq$:

Proposition 3. *If a negative-disjunctive clause $\mathcal{C} = not\ A_1 \vee \ldots \vee not\ A_n$ is true in the pair of coins $\mathcal{I}^p =< \mathcal{I}^t, \mathcal{I}^{tu} >$, in which $\mathcal{I}^{tu} \in \mathcal{H}_{\mathcal{I}_P}$, then $\mathcal{C}$ is true in any pair of coins $\mathcal{J}^p =< \mathcal{J}^t, \mathcal{J}^{tu} >$ such that $imp(\mathcal{J}^{tu}) = imp(\mathcal{I}^{tu})$.*

Because in $2^{\mathcal{H}_{\mathcal{I}_\mathcal{P}}}_{imp}$ the equivalence classes preserve the truth-value assigned to negative-disjunctive clauses, then, instead of using the domain $\mathcal{H}_{\mathcal{I}_\mathcal{P}}$ the subdomain $2^{\mathcal{H}_{\mathcal{I}_\mathcal{P}}}_{imp}$ rather would be suitable to provide semantics for negative-disjunctive clauses in disjunctive logic programs, whereas $2^{\mathcal{H}_{\mathcal{I}_\mathcal{P}}}_{exp}$ would be suitable to deal with positive-disjunctive clauses. With these ideas in mind, we define partial coins as follows:

Definition 7 (Partial coins). *A partial coin is a pair $\mathcal{I}^p =< \mathcal{I}^t, \mathcal{I}^{tu} >$ such that $\mathcal{I}^t \in 2^{\mathcal{H}_{\mathcal{I}_\mathcal{P}}}_{exp}$ and $\mathcal{I}^{tu} \in 2^{\mathcal{H}_{\mathcal{I}_\mathcal{P}}}_{imp}$.*

Accordingly, partial coins are designed to evaluate positive and negative-disjunctive rules, and can be ordered in the following way:

Definition 8 (Truth and knowledge ordering). *Let $\mathcal{I}^p_1$ and $\mathcal{I}^p_2$ be two partial coins. The truth and knowledge orderings among them are defined by*

Truth ordering: $\mathcal{I}^p_1 \sqsubseteq_t \mathcal{I}^p_2$ *iff* $\mathcal{I}^t_1 \sqsubseteq \mathcal{I}^t_2$ *and* $\mathcal{I}^{tu}_1 \preceq \mathcal{I}^{tu}_2$.
Knowledge ordering: $\mathcal{I}^p_1 \sqsubseteq_k \mathcal{I}^p_2$ *iff* $\mathcal{I}^t_1 \sqsubseteq \mathcal{I}^t_2$ *and* $\mathcal{I}^{tu}_2 \preceq \mathcal{I}^{tu}_1$.

Associated with partial coins, we propose the following notions of models:

Definition 9 (T and TU-models). *Let P be a PDLP, and $\mathcal{M}^p = <\mathcal{M}^t, \mathcal{M}^{tu}>$ be a partial coin. We say that*

- *$\mathcal{M}^t$ is a T-model of P iff, for every $M \in \mathcal{M}^t$, M is a model of P.*
- *$\mathcal{M}^{tu}$ is a TU-model of P iff, for every $M \in \mathcal{M}^{tu}$, there exists a model $M' \in \mathcal{M}^{tu}$ of P such that $M \subseteq M'$.*

We refer to $\mathcal{M}^t$ as minimal T-model of P whenever there is no T-model $\mathcal{M}$ of P such that $\mathcal{M} \sqsubset \mathcal{M}^t$. Similarly, we refer to $\mathcal{M}^{tu}$ as minimal TU-model of P whenever there is no TU-model $\mathcal{M}'$ of P such that $\mathcal{M}' \prec \mathcal{M}^{tu}$.

In point of fact, we saw in Section 2 that, for each *PDLP* P, there is a unique minimal (least) T-model $\mathcal{MM}_P \in \mathcal{M}_{\mathcal{OD}}(P)$ [14] captured by the $\mathcal{T}_P^{INT}$ operator, i.e. $\mathcal{MM}_P \equiv_{\sqsubseteq} \mathcal{T}_P^{coin} = \mathcal{T}_P^{INT\ \uparrow\omega}$.

Before continuing the study of TU-models, we introduce the $\mathcal{TU}_P^{coin}$ mapping:

Definition 10 ($\mathcal{TU}_P^{coin}$ mapping). *Let P be a PDLP. We define*

$$\mathcal{TU}_P^{coin} = imp(\mathcal{MM}_P) = imp(min(\mathcal{T}_P^{INT\ \uparrow\omega}))$$

The $\mathcal{TU}_P^{coin}$ mapping will be used to obtain the set of interpretations in $2^{\mathcal{H}_{\mathcal{I}_\mathcal{P}}}_{imp}$ corresponding to the minimal models of a *PDLP*. Concerning its meaning, unlike for the $\mathcal{T}_P^{INT}$ operator, from which one can obtain the least T-model of a positive-disjunctive definite logic program P (see Equation (3)), one cannot draw out the least TU-model of P, under $\preceq$, with the operator $\mathcal{TU}_P^{coin}$.

Example 3. Let $P = \{a \vee b\}$ be a *PDLP*. Although $\mathcal{TU}_P^{coin} = \{\{a\}, \{b\}, \{\}\}$, P has two minimal TU-models: $\{\{a\}, \{\}\}$ *and* $\{\{b\}, \{\}\}$.

Notwithstanding, as we are about to prove, $\mathcal{TU}_P^{coin}$ corresponds to the least upper bound in $2^{\mathcal{H}_{\mathcal{I}_\mathcal{P}}}_{imp}$ of the minimal *TU*-models of a *PDLP* P:

Theorem 3. *Let P be a PDLP, and $\mathcal{M} = \bigcup\{\mathcal{M}^{tu} \mid \mathcal{M}^{tu}$ is a minimal TU-model of $P\}$. We have that $\mathcal{M} = \mathcal{TU}_P^{coin}$.*

Even though $\mathcal{T}_P^{coin}$ and $\mathcal{TU}_P^{coin}$ mappings were defined in different domains in order to attain different objectives, they do not disagree with each other in what concerns the set of minimal models of a *PDLP*:

Theorem 4. *Let P be a PDLP. We have*

$$\mathcal{T}_P^{coin} \cap \mathcal{TU}_P^{coin} = \mathcal{MM}_P(P).$$

The $\mathcal{T}_P^{coin}$ and $\mathcal{TU}_P^{coin}$ mappings therefore share the core meaning of a *PDLP* P represented by its sets of minimal models. The difference in these mappings is focused on the way that such minimal models are interpreted. In order to capture what is true in P the sets of minimal models of P are closed upward (expanded) in $\mathcal{T}_P^{coin}$, whereas this very set is closed downward in $\mathcal{TU}_P^{coin}$ to capture what is non-false in P.

Example 4. Let $P = \{a \vee b, b \vee c\}$ be a *PDLP*. We have

$$\mathcal{M}_{\mathcal{OD}}(P) = \{\{b\}, \{a, c\}, \{a, b\}, \{b, c\}, \{a, b, c\}\}$$

$$\mathcal{MM}_P(P) = \{\{b\}, \{a, c\}\}$$

and

$$\mathcal{T}_P^{INT} \uparrow^0 = \{\{\}, \{a\}, \{b\}, \{c\}, \{a, b\}, \{a, c\}\{b, c\}, \{a, b, c\}\}$$

$$\mathcal{T}_P^{INT} \uparrow^1 = \mathcal{T}_P^{INT}(\mathcal{T}_P^{INT} \uparrow^0) = \{\{b\}, \{a, c\}, \{a, b\}, \{b, c\}, \{a, b, c\}\}$$

$$\mathcal{T}_P^{INT} \uparrow^\omega = \mathcal{T}_P^{INT} \uparrow^2 = \mathcal{T}_P^{INT}(\mathcal{T}_P^{INT} \uparrow^1) = \mathcal{T}_P^{INT} \uparrow^1$$

From which we obtain $\mathcal{T}_P^{coin} = \mathcal{T}_P^{INT} \uparrow^\omega = \{\{b\}, \{a, c\}, \{a, b\}, \{b, c\}, \{a, b, c\}\}$ and $\mathcal{TU}_P^{coin} = \{\{b\}, \{a, c\}, \{a\}, \{c\}, \{\}\}$. As expected according to Theorem 4, $\mathcal{T}_P^{coin} \cap \mathcal{TU}_P^{coin} = \mathcal{MM}_P(P)$.

In order to provide semantics for disjunctive logic programs we will resort to a Gelfond-Lifschitz like division operator [4], which transforms disjunctive logic programs into *PDLP*s. Then, to these programs, we can apply $\mathcal{T}_P^{coin}$ or $\mathcal{TU}_P^{coin}$ to determine, respectively, what is true and what is false by default.

Definition 11 (Division program). *Consider a disjunctive logic program P and an interpretation I. The division of program $\frac{P}{I}$ is the PDLP obtained by following the procedure below:*

- *For each rule $a_1 \vee \ldots \vee a_l \leftarrow b_1, \ldots, b_m, not\ c_1, \ldots, not\ c_n \in P$ such that $\{c_1, \ldots, c_n\} \cap I = \emptyset$, add $\frac{P}{I}$ the rule $a_1 \vee \ldots \vee a_l \leftarrow b_1, \ldots, b_m$.*
- *No other rule is in $\frac{P}{I}$.*

In a normal logic program P, the program division is employed to define the operators Γ_P^t and Γ_P^{tu} to obtain, respectively, what is true and what is not false by default in P, i.e. true or undefined. In the sequel, we enroll this notion and apply it to coins. However, in order to guarantee that $2_{exp}^{\mathcal{H}_{\mathcal{I}_P}}$ and $2_{imp}^{\mathcal{H}_{\mathcal{I}_P}}$ are, respectively, closed under Γ_P^t and under Γ_P^{tu}, we impose $\mathcal{H}_{\mathcal{I}_{\frac{P}{I}}}$ equal to $\mathcal{H}_{\mathcal{I}_P}$ by assuming P and $\frac{P}{I}$ defined over the same language.

Definition 12 (Γ_P^t and Γ_P^{tu}). *Let P be a disjunctive logic program and $\mathcal{I}^p =< \mathcal{I}^t, \mathcal{I}^{tu} >$ a partial coin. We define $\Gamma_P^t : 2_{imp}^{\mathcal{H}_{\mathcal{I}_P}} \rightarrow 2_{exp}^{\mathcal{H}_{\mathcal{I}_P}}$ and $\Gamma_P^{tu} : 2_{exp}^{\mathcal{H}_{\mathcal{I}_P}} \rightarrow 2_{imp}^{\mathcal{H}_{\mathcal{I}_P}}$ thus:*

$$\Gamma_P^t(\mathcal{I}^{tu}) = \bigcup_{I \in \mathcal{I}^{tu}} \mathcal{T}_{\frac{P}{I}}^{coin} \text{ and } \Gamma_P^{tu}(\mathcal{I}^t) = \bigcup_{I \in \mathcal{I}^t} \mathcal{TU}_{\frac{P}{I}}^{coin}$$

Casting attention to the definition of well founded semantics for normal logic programs [8], we now introduce the operator Φ_P to establish what follows immediately from a disjunctive logic program given a partial coin:

Definition 13 (Φ_P operator). *Consider a disjunctive logic program P and a partial coin $\mathcal{I}^p =< \mathcal{I}^t, \mathcal{I}^{tu} >$, where*

$$\mathfrak{F}(\mathcal{I}^{tu}) = \{A \in H_{B_P} \mid \forall I \in \mathcal{I}^{tu}, A \notin I\} \text{ and } \mathfrak{T}(\mathcal{I}^p) = \{I \in \mathcal{I}^t \mid \mathfrak{F}(\mathcal{I}^{tu}) \cap I = \emptyset\}.$$

We define

$$\Phi_P(\mathcal{I}^p) =< \Gamma_P^t(\mathcal{I}^{tu}), \Gamma_P^{tu}(\mathfrak{T}(\mathcal{I}^p)) >$$

Conceiving the partial coin $\mathcal{I}^p =< \mathcal{I}^t, \mathcal{I}^{tu} >$ as representing the safe knowledge so far, the rationale behind the Φ_P operator is that from $\mathcal{I}^{tu}$ one can conclude what definitely does not hold; so we can use this set of interpretations for deriving what surely holds. On the other hand, from $\mathcal{I}^t$ one knows what is true; then, by ignoring those interpretations containing atoms false by default in $\mathcal{I}^t$, one can use the remaining interpretations for deriving what may possibly hold. The Φ_P operator is assured monotonic with respect to $\sqsubseteq_k$:

Theorem 5 (Monotonicity of operator Φ_P according to $\sqsubseteq_k$). *Consider a disjunctive logic program P plus two partial coins $\mathcal{I}^p =< \mathcal{I}^t, \mathcal{I}^{tu} >$ and $\mathcal{J}^p =< \mathcal{J}^t, \mathcal{J}^{tu} >$. If $\mathcal{I}^p \sqsubseteq_k \mathcal{J}^p$ then $\Phi_P(\mathcal{I}^p) \sqsubseteq_k \Phi_P(\mathcal{J}^p)$.*

We ascertain forthwith, by the Knaster-Tarski theorem, that, for every disjunctive logic program P, Φ_P possesses a least fixed point under $\sqsubseteq_k$, thereby establishing the well-founded model with disjunction of P.

Definition 14 (Well-founded semantics with disjunction). *Let P be a disjunctive logic program. Each fixed point of the Φ_P operator corresponds to a partial stable model with disjunction of P. In particular, the least one under the $\sqsubseteq_k$ ordering is its well-founded model. It can be obtained by the following transfinite sequence of partial coins $\mathcal{I}_P^{\uparrow\alpha} =< \mathcal{I}_\alpha^t, \mathcal{I}_\alpha^{tu} >$:*

$$\begin{array}{l}
\mathcal{I}_P^{\uparrow 0} =< \mathcal{H}_{\mathcal{I}_P}, \mathcal{H}_{\mathcal{I}_P} > \\
\mathcal{I}_P^{\uparrow\alpha} = \Phi_P(\mathcal{I}_P^{\uparrow\alpha-1}) =< \Gamma_P^t(\mathcal{I}_{\alpha-1}^{tu}), \Gamma_P^{tu}(\mathfrak{T}(\mathcal{I}_P^{\uparrow\alpha-1})) >, \text{ for successor ordinal } \alpha \\
\mathcal{I}_P^{\uparrow\alpha} =< lub_{exp}(\{\mathcal{I}_\beta^t \mid \beta < \alpha\}), glb_{imp}(\{\mathcal{I}_\beta^{tu} \mid \beta < \alpha\}) >= \\
\quad < \bigcap_{\beta<\alpha} \mathcal{I}_\beta^t, \bigcap_{\beta<\alpha} \mathcal{I}_\beta^{tu} >, \text{for limit ordinal } \alpha.
\end{array}$$

The Well-Founded Semantics with Disjunction for a disjunctive logic program P ($WFS_d(P)$) is defined by $\mathcal{I}_P^{\uparrow\alpha}$, where α is the smallest ordinal such that $\mathcal{I}_P^{\uparrow\alpha} = \Phi_P(\mathcal{I}_P^{\uparrow\alpha})$.

It is noteworthy stressing that, for normal logic programs, WFS_d and WFS are isomorphic:

Theorem 6. *Let P be a normal logic program, $\mathcal{W} =< \mathcal{W}^t, \mathcal{W}^{tu} >$ be the WFS_d of P, and $W =< W^t, W^{tu} >$ be the WFS of P.*

- *An atom A is true in $\mathcal{W}$ iff it is also true in W.*
- *not A is true in $\mathcal{W}$ iff it is also true in W.*

Hence, for normal logic programs, WFS_d may be understood as an alternative characterisation of WFS, satisfying one condition we exacted for evaluating a good well-founded semantics for disjunctive logic programs.

One may wonder why not define Φ_P just as $< \Gamma_P^t(\mathcal{I}^{tu}), \Gamma_P^{tu}(\mathcal{I}^t)) >$. The justification is simple; posed in that manner, WFS_d becomes ineligible for one of the program transformations presented in [2]: *positive reduction*. We illustrate this in the next example:

Example 5. Let $P = \{ a \vee b, b \leftarrow not\ c\}$

$\mathcal{I}_P^{\uparrow 0} = < \mathcal{H}_{\mathcal{I}_P}, \mathcal{H}_{\mathcal{I}_P} >$
$\mathcal{I}_P^{\uparrow 1} = < \{\{a\},\{b\},\{a,b\},\{a,c\},\{b,c\},\{a,b,c\}\},\{\{a\},\{b\},\{\}\} >$
$\mathcal{I}_P^{\uparrow 2} = < \{\{b\},\{a,b\},\{b,c\},\{a,b,c\}\},\{\{b\},\{\}\} >$
$WFS_d(P) = < \mathcal{W}^t, \mathcal{W}^{tu} > = \mathcal{I}_P^{\uparrow 3} = \mathcal{I}_P^{\uparrow 2}$

As *not* c is *true* in $WFS_d(P)$, $c \in \mathfrak{F}(\mathcal{W}^{tu})$, only interpretations in $\mathcal{W}^t$ deprived of c are considered to determine $\mathcal{W}^{tu}$, i.e.

$$\mathcal{W}^{tu} = \bigcup_{W_1 \in \mathfrak{T}(WFS_d(P))} \mathcal{TU}^{coin}_{\frac{P}{W_1}} \tag{5}$$

Considering that for every $W_1 \in \mathfrak{T}(WFS_d(P))$, $\frac{P}{W_1} = \{a \vee b, b\}$, we have $imp(min(\mathcal{T}^{coin}_{\frac{P}{W_1}})) = \{\{b\},\{\}\}$, leading us to conclude *not* a is in $WFS_d(P)$.

Indeed, had we considered every interpretation in $\mathcal{W}^t$ to characterise $\mathcal{W}^{tu}$ in (5), then whether from $\{b,c\}$ or $\{a,b,c\}$ we would obtain $a \in \mathcal{TU}^{coin}_{\frac{P}{\{b,c}}}$ or $a \in \mathcal{TU}^{coin}_{\frac{P}{\{a,b,c}}}$, because the rule $b \leftarrow not\ c \in P$ is eliminated in $\frac{P}{\{b,c\}}$ as well as in $\frac{P}{\{a,b,c\}}$. Under such a scheme, we would infer the unexpected result $\{a\} \in \mathcal{W}^{tu}$, dismissing any possibility of concluding *not* a in $WFS_d(P)$.

4 Examples

By exhibiting appropriate examples, we are about to show that WFS_d does not coincide with the most prominent semantics for disjunctive logic programs [10,13,1,11,9,2,19].

Example 6. Let $P_1 = \{a \vee b \vee c, a \leftarrow not\ b, b \leftarrow not\ c, c \leftarrow not\ a\}$.

$\mathcal{I}_{P_1}^{\uparrow 0} = < \mathcal{H}_{\mathcal{I}_{P_1}}, \mathcal{H}_{\mathcal{I}_{P_1}} >$
$\mathcal{I}_{P_1}^{\uparrow 1} = < \{\{a\},\{b\},\{c\},\{a,b\},\{a,c\},\{b,c\},\{a,b,c\},\mathcal{H}_{\mathcal{I}_P} >$
$\mathcal{I}_{P_1}^{\uparrow 2} = < \{\{a\},\{b\},\{c\},\{a,b\},\{a,c\},\{b,c\},\{a,b,c\}\},$
$\{\{a,b\},\{b,c\},\{a,c\},\{a\},\{b\},\{c\},\{\}\} >$
$WFS_d(P_1) = \mathcal{I}_{P_1}^{\uparrow 3} = \mathcal{I}_{P_1}^{\uparrow 2}.$

Program P_1 is not assigned any meaning according to Partial Disjunctive Stable Models [11]. In contradistinction, WFS_d enables us to say that both $a \vee b \vee c$ and *not* $a \vee$ *not* $b \vee$ *not* c are true.

Indeed, even for stratified programs, where Partial Disjunctive Stable Models and Perfect Models [10] are always defined, we obtain distinct results via WFS_d:

Example 7. Let $P_2 = \{a \vee b, b \vee c, d \leftarrow not\ a, d \vee e \leftarrow not\ c\}$.

$$\begin{aligned}
&\mathcal{I}^{\uparrow 0}_{P_2} = < \mathcal{H}_{\mathcal{I}_{P_2}}, \mathcal{H}_{\mathcal{I}_{P_2}} > \\
&\mathcal{I}^{\uparrow 1}_{P_2} = < \{\{b\},\{a,c\},\{a,b\},\{b,c\},\{b,d\},\{b,e\},\{a,b,c\},\{a,b,d\},\{a,b,e\},\{a,c,d\}, \\
&\quad \{a,c,e\},\{b,c,d\},\{b,c,e\},\{b,d,e\},\{a,b,c,d\},\{a,b,c,e\},\{a,b,d,e\},\{b,c,d,e\}, \\
&\quad \{a,c,d,e\},\{a,b,c,d,e\}\}, \\
&\quad \{\{a,c,d\},\{a,c,e\},\{a,c\},\{a,d\},\{a,e\},\{c,d\},\{c,e\},\{b,d\},\{b,e\},\{a\},\{b\}, \\
&\quad \{c\},\{d\},\{e\},\{\}\} > \\
&WFS_d(P_2) = \mathcal{I}^{\uparrow 2}_{P_2} = \mathcal{I}^{\uparrow 1}_{P_2}.
\end{aligned}$$

P_2 has the partial stable models (perfect models): $< \{a,c\},\{a,c\} >$ and $< \{b,d\},\{b,d\} >$. Hence *not e* is obtained according to disjunctive partial stable models and perfect models; that same conclusion is verified under static [9] and stationary semantics, whilst it stands *undefined* in WFS_d.

In the next example we compare WFS_d to *GDWFS* [1], *D-WFS* [2], *WFDS* [19], and *SWFS* [13]:

Example 8. Let $P_3 = \{a \vee b \leftarrow not\ b, b \leftarrow not\ b\}$.

$$\begin{aligned}
&\mathcal{I}^{\uparrow 0}_{P_3} = < \mathcal{H}_{\mathcal{I}_{P_3}}, \mathcal{H}_{\mathcal{I}_{P_3}} > \\
&\mathcal{I}^{\uparrow 1}_{P_3} = < \{\{\},\{a\},\{b\},\{a,b\},\{a,b\}\},\{\{b\},\{\}\} > \\
&WFS_d(P_3) = \mathcal{I}^{\uparrow 2}_{P_3} = \mathcal{I}^{\uparrow 1}_{P_3}.
\end{aligned}$$

In P_3, WFS_d assigns *false* to a, whilst b remains *undefined*. Given that *D-WFS* deletes the first clause by elimination of non-minimal clauses, a is obtained as *false* and b continues *undefined* because there is no rule left with a in the head. *SWFS* assigns *undefined* to both a and b. In *GDWFS* b is *true* and a is *false*. Finally, for *WFDS* a and b are undefined. Then we have that in P_3 *GDWFS* derives the most information, followed by WFS_d and *D-WFS*. The weakest semantics for this program are *SWFS* and *WFDS*.

So far WFS_d and *D-WFS* have presented the same results. However this is not the case generally:

Example 9. Let $P_4 = \{a \vee b, c \leftarrow not\ a, c \leftarrow not\ b\}$.

$$\begin{aligned}
&\mathcal{I}^{\uparrow 0}_{P_4} = < \mathcal{H}_{\mathcal{I}_{P_4}}, \mathcal{H}_{\mathcal{I}_{P_4}} > \\
&\mathcal{I}^{\uparrow 1}_{P_4} = < \{\{a\},\{b\},\{a,b\},\{a,c\},\{b,c\},\{a,b,c\}\}, \\
&\quad \{\{a,c\},\{b,c\},\{a\},\{b\},\{c\},\{\}\} > \\
&\mathcal{I}^{\uparrow 2}_{P_4} = < \{\{a,c\},\{b,c\},\{a,b,c\}\},\{\{a,c\},\{b,c\},\{a\},\{b\},\{c\},\{\}\} > \\
&WFS_d(P_4) = \mathcal{I}^{\uparrow 3}_{P_4} = \mathcal{I}^{\uparrow 2}_{P_4}.
\end{aligned}$$

According to WFS_d, c is *true* in P_4 but *undefined* in *D-WFS*. Though *D-WFS* and WFS_d do not coincide generally, yet *D-WFS* is strictly weaker than WFS_d.

Theorem 7. *Let P_1 and P_2 be disjunctive logic programs such that P_2 results from P_1 by unfolding, elimination of tautologies and nonminimal rules, and positive and negative reduction. We have $WFS_d(P_1) = WFS_d(P_2)$.*

This guarantees that *D-WFS* is strictly weaker than WFS_d, since *D-WFS* is known to be the weakest semantics allowing unfolding, elimination of tautologies and nonminimal rules, and positive and negative reduction.

5 Conclusions and Future Work

When studying disjunctive logic programs, one of the first obstacles is not to be at a loss in keeping track of the alphabet soup of their different semantics. Despite the jocular tone of this affirmation, it reveals the great attention the logic programming community has devoted to this matter since 1982. Collaterally, it also reveals the difficulty in reaching a consensual semantics.

Bearing in mind that scores of semantics have already been presented to treat disjunction in logic programming, it might appear unreasonable to consider new proposals. Indeed, finding a semantics to capture the meaning of disjunctive logic programs is a conundrum sufficiently hard to dissuade simple solutions.

Notwithstanding, we have developed a new semantics for disjunctive logic programs, christened *Well-Founded Semantics with Disjunction* (WFS_d), by generalising to a set of interpretations the fixed point operator Ω_P defined in [8] to characterise *WFS*. Our semantics does not coincide generally with any other we have studied. It surmises *D-WFS*, Static, *GDWFS*, *WFDS* and *SWFS*. Yet we have shown WFS_d is uniquely defined for every disjunctive logic program, generalises *WFS*, and is strictly stronger than *D-WFS*.

Furthermore, preliminary results authorise us to state that WFS_d can be extended to deal with explicit negation, either explosive or paraconsistent. Again we postpone analyses of the algorithmic complexity of WFS_d, along with its model-theoretical counterpart, apropos of which we can promote a detailed scenario exploring the logical properties of WFS_d. In particular, it empowers us to identify the meaning assigned to disjunction and the properties supporting the relation between disjunction and default negation in our semantics.

Concomitantly we are working in another well-founded semantics for disjunctive logic programs we will present in a future work. Although preserving good properties of WFS_d, this new semantics is simpler and in some cases more credulous. For instance, in the program shown in Example 7, it concludes that *not e* is *true*, instead of standing *undefined* as in WFS_d.

Scaffolded on the aforementioned reasons, while we do not claim WFS_d is *the* best semantics for disjunctive logic programs, we steadfastly uphold that WFS_d is not just a new semantics for disjunctive logic programs, but that it is at least a worthwhile one: a semantics whose adequacy rend it a possible choice for representing knowledge and reasoning in logic programming.

Acknowledgments. João Alcântara is a PhD student supported by CAPES - Brasília, Brazil. The authors also thank the REWERSE project.

References

1. C. Baral, J. Lobo, and J. Minker. Generalized well-founded semantics for logic programs. In *10th International Conference on Automated Deduction*, 1990.
2. S. Brass and J. Dix. Semantics of (Disjunctive) Logic Programs Based on Partial Evaluation. *Journal of Logic Programming*, 38(3):167–213, 1999.
3. J. A. Fernández and J. Minker. Bottom-up computation of perfect models for disjunctive theories. *Journal of Logic Programming*, 25(1):33–51, 1995.
4. M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *5th Int. Conf. on Logic Programming*, pages 1070–1080. MIT Press, 1988.
5. J. Lobo, J. Minker, and A. Rajasekar. *Foundations of Disjunctive Logic Programming*. MIT Press, 1992.
6. J. Minker. On indefinite databases and the closed world assumption. In *6th Conference on Automated Deduction*, volume 138, pages 292–308. Springer-Verlag, 1982.
7. J. Minker and A. Rajasekar. A fixpoint semantics for disjunctive logic programs. *Journal of Logic Programming*, 9(1):45–74, 1990.
8. H. Przymusinska and T. Przymusinski. Semantic issues in deductive databases and logic. In R. Banerji, editor, *Formal Techniques in Artificial Intelligence - a Sourcebook*, pages 321–367. North Holland, 1990.
9. T. Przymusinski. Static semantics for normal and disjunctive logic programs. *Annals of Mathematics and Artificial Intelligence*, Special Issue on Disjunctive Programs:323–357, 1995.
10. T. C. Przymusinski. On the declarative semantics of stratified deductive databases and logic programs. In J. Minker, editor, *Foundation of Deductive Databases and Logic Programming*, pages 193–216. Morgan Kaufmann, 1988.
11. T. C. Przymusinski. Stable semantics for disjunctive programs. *New Generation Computing Journal*, 9:401–424, 1991.
12. R. Reiter. On closed world data bases. In H. Gallaire and J. Minker, editors, *Logic and Databases*, pages 55–76. Plenum Press, 1978.
13. K. Ross. The well founded semantics for disjunctive logic programs. In *First Int. Conf. on Deductive and Object Oriented Databases*, pages 352–369, 1989.
14. D. Seipel, J. Minker, and C. Ruiz. Model generation and state generation for disjunctive logic programs. *Journal of Logic Programming*, 32(1):49–69, 1997.
15. M. Smyth. Power domains. *J. Computer and System Sciences*, 16(1):23–36, 1978.
16. A. Tarski. Lattice-theoretic fixpoint theorem and its applications. *Pacific Journal of Mathematics*, pages 285–309, 1955.
17. M. van Emden and R. Kowalski. The semantics of predicate logic as a programming language. *Journal of ACM*, 4(23):733–742, 1976.
18. A. van Gelder, K. Ross, and J. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, 1991.
19. K. Wang. Argumentation-based abduction in disjunctive logic programming. *Journal of Logic Programming*, 2001.

Semantics of Framed Temporal Logic Programs*

Zhenhua Duan[1], Xiaoxiao Yang[1], and Maciej Koutny[2]

[1] School of Computer Science and Engineering, Xidian University,
Xi'an 710071, P.R. China
{zhhduan, xxyang}@mail.xidian.edu.cn
[2] Department of Computing Science, University of Newcastle,
Newcastle upon Tyne NE1 7RU, UK
maciej.koutny@ncl.ac.uk

Abstract. This paper investigates semantics of framed temporal logic programs. To this end, a projection temporal logic and its executable subset are presented. Based on this language, a framing technique is introduced. The semantics of a non-framed program is well interpreted by the canonical model. However, since introducing a framing operator destroys monotonicity, a canonical model may no longer capture the intended meaning of a program. Hence, a minimal model theory is developed. Within this model, negation by default is used to manipulate frame operator. Further, the temporal semantics of framed programs is captured by means of the minimal models. The existence of a minimal model for a given framed program is also proved. An example is given to illustrate how the semantics of framed programs can be captured.

Keywords: Temporal logic programming, framing, minimal model, monotonicity, semantics.

1 Introduction

Framing [6,3] is concerned with how the value of a variable from one state can be carried to the next. Temporal logic offers no solution in this respect; no value from a previous state is assumed to be carried along. Framing techniques have been employed by conventional imperative languages for many years. However, framing in conventional languages has been taken for granted and there is no conscious effort to consider it explicitly. However, within a temporal logic programming language such as Tempura [8,3], XYZ/E [11] a program is executed over a sequence of states and the values of variables are not inherited automatically. Thus, for improving the efficiency of a program and synchronizing communication for parallel processes, we have to consider the framing techniques carefully in temporal logic programming. To synchronize communication between parallel processes in a concurrent program with the shared variable

* This research is supported by the NSFC Grant No. 60373103 and 60433010, the SRFDP Grant 20030701015, and Grant SYSKF0407 from Lab. Computer Science, ISCAS.

M. Gabbrielli and G. Gupta (Eds.): ICLP 2005, LNCS 3668, pp. 356–370, 2005.

model, a synchronization construct, *await*(c) is required, similarly as in many concurrent programming languages [10]. Defining *await*(c) is difficult without some kind of framing construct since the values of variables are not inherited automatically from one state to another. But one requires some kind of indefinite stability, since it cannot be known at the point of use how long the waiting will last. At the same time one must also allow variables to change, so that an external process can modify the boolean parameter and it can eventually become true.

To capture the temporal semantics of non-framed programs in Tempura, the canonical model has been introduced to interpret programs [3]. Within this model, the semantics of a non-framed program is well captured. However, since introducing a framing operator destroys monotonicity, a canonical model may no longer capture the intended meaning of a program. A program, therefore, can have different meanings under different models. To interpret a framed program faithfully, minimal models will be employed in this paper. Within this model, negation by default is used to manipulate the frame operator. Furthermore, the existence of a minimal model for a satisfiable program is proved by means of fix-point theory.

This paper is organized as follows. In the following section, a Projection Temporal Logic (PTL) is briefly introduced. Based on this logic, an executable temporal logic programming language called Tempura is formalized in Section 3. Section 4 formalizes a framing technique. Section 5 presents the temporal semantics of framed programs by means of minimal models. Finally, in Section 6, an example is given to illustrate how the minimal model can be used to capture the meaning of a framed program. Conclusions are drawn in Section 7.

2 Projection Temporal Logic

Our underlying logic PTL is the first order temporal logic [7,10] with projection [2,3,5]. It is an extension of ITL [8].

2.1 Syntax

Let Π be a countable set of *propositions*, and V be a countable set of typed static and dynamic *variables*. The *terms* e and *formulas* p of the logic are given by the following grammar:

$$e ::= x \mid u \mid \bigcirc e \mid \ominus e \mid \mathit{beg}(e) \mid \mathit{end}(e) \mid f(e_1, \ldots, e_n)$$

$$p ::= \pi \mid e_1 = e_2 \mid P(e_1, \ldots, e_n) \mid \neg p \mid p_1 \wedge p_2 \mid \exists x : p \mid \bigcirc p \mid \ominus p \mid (p_1, \ldots, p_m)\ \mathit{prj}\ p$$

where π is a proposition, x is a dynamic variable and u is a static variable. In $f(e_1, \ldots, e_n)$ and $P(e_1, \ldots, e_n)$, where f is a function and P is a predicate. It is assumed that the types of the terms are compatible with those of the arguments of f and P. A formula (term) is called a state formula (term) if it does not contain any temporal operators (i.e. $\bigcirc$, $\ominus$, *beg*(.), *end*(.) and *prj*); otherwise it is a temporal formula (term).

2.2 Semantics

A *state* s is a pair of assignments (I_v, I_p) which for each variable $v \in V$ defines $s[v] = I_v[v]$, and for each proposition $\pi \in \Pi$ defines $s[\pi] = I_p[\pi]$. $I_v[v]$ is a value of the appropriate type or *nil* (undefined), whereas $I_p[\pi] \in \{true, false\}$. An *interval* $\sigma = \langle s_0, s_1, \ldots \rangle$ is a non-empty (possibly infinite) sequence of states. The length of σ, denoted by $|\sigma|$, is defined as ω if σ is infinite; otherwise it is the number of states in σ minus one. To have a uniform notation for both finite and infinite intervals, we will use *extended integers* as indices. That is, we consider the set N_0 of non-negative integers and ω, $N_\omega = N_0 \cup \{\omega\}$ and extend the comparison operators, $=, <, \leq$, to N_ω by considering $\omega = \omega$, and for all $i \in N_0$, $i < \omega$. Moreover, we define $\preceq$ as $\leq - \{(\omega, \omega)\}$. For $0 \leq i, j \leq |\sigma|$ we will use $\sigma_{(i..j)}$ to denote the subinterval $\langle s_i, s_{i+1}, \ldots, s_j \rangle$.[1] It is assumed that each static variable is assigned the same value in all the states in σ. To define the semantics of the projection operator we need an auxiliary operator for intervals.

Let $\sigma = \langle s_0, s_1, \ldots \rangle$ be an interval and $r_1, \ldots, r_h$ be integers ($h \geq 1$) such that $0 \leq r_1 \leq r_2 \leq \ldots \leq r_h \leq |\sigma|$. The *projection* of σ onto $r_1, \ldots, r_h$ is the interval, $\sigma{\downarrow}(r_1, \ldots, r_h) = \langle s_{t_1}, s_{t_2}, \ldots, s_{t_l} \rangle$, where $t_1, \ldots, t_l$ is obtained from $r_1, \ldots, r_h$ by deleting all duplicates. For example, $\langle s_0, s_1, s_2, s_3, s_4 \rangle{\downarrow}(0, 0, 2, 2, 2, 3) = \langle s_0, s_2, s_3 \rangle$.

An *interpretation* for a PTL term or formula is a tuple $\mathcal{I} = (\sigma, i, k, j)$, where $\sigma = \langle s_0, s_1, \ldots \rangle$ is an interval, i and k are non-negative integers, and j is an integer or ω, such that $i \leq k \preceq j \leq |\sigma|$. We use (σ, i, k, j) to mean that a term or formula is interpreted over a subinterval $\sigma_{(i..j)}$ with the current state being s_k. For every term e, the evaluation of e relative to interpretation $\mathcal{I} = (\sigma, i, k, j)$ is defined as $\mathcal{I}[e]$, by induction on the structure of a term, as shown in Fig. 1, where v is a variable and $e_1, \ldots, e_m$ are terms. The satisfaction relation for formulas $\models$ is defined as the least relation satisfying the following.

$$
\begin{aligned}
\mathcal{I}[a] &= s_k[a] \;=\; I_v^k[a] \;=\; I_v^i[a] \quad \text{if } a \text{ is a static variable.} \\
\mathcal{I}[x] &= s_k[x] \;=\; I_v^k[x] \text{ if } x \text{ is a dynamic variable.} \\
\mathcal{I}[f(e_1, \ldots, e_m)] &= \begin{cases} f(\mathcal{I}[e_1], \ldots, \mathcal{I}[e_m]) & \text{if } \mathcal{I}[e_h] \neq nil \text{ for all } h \\ nil & \text{otherwise} \end{cases} \\
\mathcal{I}[\bigcirc e] &= \begin{cases} (\sigma, i, k+1, j)[e] & \text{if } k < j \\ nil & \text{otherwise} \end{cases} \\
\mathcal{I}[\ominus e] &= \begin{cases} (\sigma, i, k-1, j)[e] & \text{if } i < k \\ nil & \text{otherwise} \end{cases} \\
\mathcal{I}[beg(e)] &= (\sigma, i, i, j)[e] \\
\mathcal{I}[end(e)] &= \begin{cases} (\sigma, i, j, j)[e] & \text{if } j \neq \omega \\ nil & \text{otherwise} \end{cases}
\end{aligned}
$$

Fig. 1. Interpretation of PTL terms

[1] When $i > j$, $\sigma_{(i..j)}$ is the empty string, and if $j = \omega$ then $\sigma_{(i..j)} = \langle s_i, s_{i+1}, \ldots \rangle$.

1. $\mathcal{I} \models \pi$ if $s_k[\pi] = I_p^k[\pi] = true$.
2. $\mathcal{I} \models P(e_1, \ldots, e_m)$ if $P(\mathcal{I}[e_1], \ldots, \mathcal{I}[e_m]) = true$ and $\mathcal{I}[e_h] \neq nil$, for all h.
3. $\mathcal{I} \models e = e'$ if $\mathcal{I}[e] = \mathcal{I}[e']$.
4. $\mathcal{I} \models \neg p$ if $\mathcal{I} \not\models p$.
5. $\mathcal{I} \models p \wedge q$ if $\mathcal{I} \models p$ and $\mathcal{I} \models q$.
6. $\mathcal{I} \models \bigcirc p$ if $k < j$ and $(\sigma, i, k+1, j) \models p$.
7. $\mathcal{I} \models \ominus p$ if $i < k$ and $(\sigma, i, k-1, j) \models p$.
8. $\mathcal{I} \models \exists x : p$ if for some interval σ' which has the same length as σ, $(\sigma', i, k, j) \models p$ and the only difference between σ and σ' can be in the values assigned to variable x.
9. $\mathcal{I} \models (p_1, \ldots, p_m)\ prj\ q$ if there exist integers $k = r_0 \leq r_1 \leq \ldots \leq r_m \preceq j$ such that $(\sigma, i, r_0, r_1) \models p_1$, $(\sigma, r_{l-1}, r_{l-1}, r_l) \models p_l$ (for $1 < l \leq m$), and $(\sigma', 0, 0, |\sigma'|) \models q$ for one of the following σ':
   (a) $r_m < j$ and $\sigma' = \sigma{\downarrow}(r_0, \ldots, r_m) \cdot \sigma_{(r_m+1..j)}$
   (b) $r_m = j$ and $\sigma' = \sigma{\downarrow}(r_0, \ldots, r_h)$ for some $0 \leq h \leq m$.

A formula p is said to be:

- *satisfied* by an interval σ, denoted $\sigma \models p$, if $(\sigma, 0, 0, |\sigma|) \models p$.
- *satisfiable* if $\sigma \models p$ for some σ.
- *valid*, denoted $\models p$, if $\sigma \models p$ for all σ.
- *left end closed* (lec-formula) if $(\sigma, k, k, j) \models p \Leftrightarrow (\sigma, i, k, j) \models p$ for any interpretation (σ, i, k, j).
- *equivalent* to another formula q, denoted $p \equiv q$, if $\models \Box(p \leftrightarrow q)$.

Projection. To ensure smooth synchronization between $p_1, \ldots, p_m$ and q, the previous operator is not allowed within q appearing in $(p_1, \ldots, p_m)\ prj\ q$. The projection construct is executable, and to interpret $(p_1, \ldots, p_m)\ prj\ q$ we need *two* sequences of clocks (states) running on different time scales: one is a local state sequence, over which $p_1, \ldots, p_m$ are executed, while the other is a global state sequence over which q is executed in parallel with the sequence of processes $p_1, \ldots, p_m$. The execution proceeds as follows: First, q and p_1 start at the first global state and p_1 is executed over a sequence of local states until its termination. Then (the remaining part of) q and p_2 are executed at the second global state. Subsequently, p_2 is continuously executed over a sequence of local states until its termination, and so on. Although q and p_1 start at the same time, $p_1, \ldots, p_m$ and q may terminate at different time points. E.g., if q terminates before some p_{h+1}, then, subsequently, $p_{h+1}, \ldots, p_m$ are executed sequentially.

2.3 Other Formulas

The derived connectives, $\vee$, $\rightarrow$ and $\leftrightarrow$, as well as the logic formulas, *true* and *false*, are defined as usual. We also use the following derived formulas:

Simple Temporal Formulas

$$Prj\ (p_1,\ldots,p_m) \stackrel{\text{def}}{=} (p_1,\ldots,p_m)\ prj\ empty \qquad empty \stackrel{\text{def}}{=} \neg \bigcirc true$$

$$\odot p \stackrel{\text{def}}{=} \neg\ominus\neg p \qquad more \stackrel{\text{def}}{=} \neg empty$$

$$first \stackrel{\text{def}}{=} \neg\ominus true \qquad skip \stackrel{\text{def}}{=} len(1)$$

$$\bigodot p \stackrel{\text{def}}{=} empty \vee \bigcirc p \qquad \Diamond p \stackrel{\text{def}}{=} Prj\ (true, p)$$

$$len(n) \stackrel{\text{def}}{=} \begin{cases} empty & n = 0 \\ \bigcirc len(n-1) & n > 1 \end{cases} \qquad \Box p \stackrel{\text{def}}{=} \neg\Diamond\neg p$$

The chop operator (;), which is a central operator in ITL [8], can be expressed by the projection operator of PTL, as follows: $p; q \stackrel{\text{def}}{=} Prj\ (p, q)$. The chop star operator ($*$) of [9] can also be defined.

Theorem 1. Let p, q, w be formulas, and e, e_1, e_2 terms, then the following formulas hold:

$FCH1$	$\bigcirc p; q \equiv \bigcirc (p; q)$	$FDU1$	$\neg\bigodot p \equiv \bigcirc\neg p$
$FCH2$	$w \wedge (p; q) \equiv (w \wedge p; q)$	$FDU2$	$\neg\bigcirc p \equiv \bigodot\neg p$
$FD3$	$\bigcirc(p \wedge q) \equiv \bigcirc p \wedge \bigcirc q$	$FE1$	$\Diamond p \equiv p \vee \bigcirc\Diamond p$
$FD4$	$\bigcirc(p \vee q) \equiv \bigcirc p \vee \bigcirc q$	$FE2$	$\Box p \equiv p \wedge \bigodot\Box p$
$FD9$	$(w; p \vee q) \equiv (w; p) \vee (w; q)$	NFE	$\neg first \wedge more \supset (\bigcirc\ominus p \leftrightarrow \ominus\bigcirc p)$
$FD10$	$(p \vee q; w) \equiv (p; w) \vee (q; w)$	$FQT1$	$\bigcirc(\exists x : p) \equiv \exists x : \bigcirc p$
$FW2$	$\bigodot p \equiv \neg\bigcirc\neg p$	$EQ3$	$\neg first \wedge more \supset (\bigcirc\ominus e = \ominus\bigcirc e)$
$FW1$	$\bigcirc p \equiv \bigodot p \wedge more$	$EQ1$	$more \supset (\bigcirc e_1 = \bigcirc e_2 \leftrightarrow \bigcirc(e_1 = e_2))$
$FUN3$	$\bigcirc e_1 + \bigcirc e_2 = \bigcirc(e_1 + e_2)$	$FST1$	$p^* \equiv empty \vee (p; p^*) \vee p \wedge \Box more$

These logic laws are useful in the reduction of programs and the proofs of them can be found in [3].

3 Temporal Logic Programming Language

The programming language we use is an executable subset of PTL. It is an extension of Tempura [8,6]. We augment Tempura with frame, new projection, and await operators [2,3,4,5]. In addition, variables within a program can also refer to their previous values. In the following, we first introduce the basic constructs of Tempura. Later, we formalize the frame and await constructs.

3.1 Syntax

The basic statements of Tempura are as follows.

- **Assignment**: $x = e$
- **Conjunction**: $p \wedge q$
- **Conditional statement**: $if\ b\ then\ p\ else\ q \stackrel{\text{def}}{=} (b \rightarrow p) \wedge (\neg b \rightarrow q)$
- **Local variable**: $\exists x : p$
- **Next statement**: $\bigcirc p$

- **Always statement**: $\Box p$
- **Sequential statement**: $p; q$
- **While statement**: $while\ b\ do\ p \stackrel{\text{def}}{=} (p \wedge b)^* \wedge \Box(empty \rightarrow \neg b)$
- **Projection statement**: $(p_1, ..., p_m)\ prj\ q$
- **Parallel statement**: $p||q \stackrel{\text{def}}{=} p \wedge (q; true) \vee q \wedge (p; true)$
- **Termination**: $empty$

where b is a state boolean expression consisting of propositions, variables, and boolean connectives.

The following formulas are derived from PTL and can be used in programs.

1. **Assignment Operators**
   Let x be a variable, u a static variable, and e an expression (term).
   1) Next assignment: $x\ o{=}\ e \stackrel{\text{def}}{=} \bigcirc x = e$
   2) Unit assignment: $x\ :=\ e \stackrel{\text{def}}{=} skip \wedge x\ o{=}\ e$
   The next assignment specifies the value of x to be e at the next state, while the unit assignment assigns value e to x at the next state, the same function as the next assignment, but, in the meantime, it specifies the length of the interval over which the assignment takes place to be 1.

2. **Termination and the Final State**
   1) $fin(p) \stackrel{\text{def}}{=} \Box(empty \rightarrow p)$
   2) $keep(p) \stackrel{\text{def}}{=} \Box(\neg empty \rightarrow p)$
   3) $halt(p) \stackrel{\text{def}}{=} \Box(empty \leftrightarrow p)$
   $fin(p)$ holds over an interval as long as p holds at the final state, whereas $keep(p)$ holds over an interval if p holds at every state, ignoring the final one. $halt(p)$ holds over an interval if and only if p holds at the final state.

3.2 Semantics of Programs

An expression e can be treated as a term and a program P can be viewed as a formula in PTL. Therefore, the evaluation of e and the interpretation of P can be done as in PTL. However, since the programming language is a subset of the underlying logic, a program may have its own characteristics and may be interpreted in a simple and manageable way.

In order to interpret temporal logic programs, we assume that a program P contains a finite set S of variables and a finite set Φ of propositions. We interpret propositions over B and variables over $D' = D \cup \{nil\}$, where nil is undefined and D denotes all data needed by us including integers, lists, sets etc. For a program P, there are three ways to interpret propositions contained in P, namely canonical, complete, and partial interpretations as defined for the semantics of logic programming language [1]. Here, we use the canonical interpretation only on propositions. That is, in a model $\sigma =< (I_v^0, I_p^0), ... >$, I_v^k is used as in the logic but I_p^k is changed to the canonical interpretation.

A canonical interpretation on propositions is a subset $I_p \subseteq \Phi$. Implicitly, propositions not in I_p are false. Note that I_p^k in the interpretation of the logic

framework is an assignment of a truth value in B to each proposition $\pi \in \Pi$ at state s_k; whereas in a canonical interpretation, I_p^k is a set of propositions, each of them has truth value *true* in B at s_k. Clearly, the two definitions are equivalent except that they refer to different sets of variables and propositions. Using canonical interpretation is necessary for easy manipulation of minimal models. Let $\sigma =< (I_v^0, I_p^0), ... >$ be a model. We denote the sequence of interpretation on propositions of σ by $\sigma_p =< I_p^0, ... >$. σ_p is said to be canonical if each $I_p^i (i \geq 0)$ is a canonical interpretation on propositions.

If there exists a model σ with σ_p being a canonical interpretation sequence on propositions and $\sigma \models P$ as in the logic, then program P is said to be satisfiable under the canonical interpretation on propositions, denoted by $\sigma \models_c P$; and σ_p is said to be a canonical interpretation sequence (on propositions) of program P. If for all σ with σ_p being a canonical interpretation sequence, $\sigma \models P$, then program P is said to be valid under the canonical interpretation on propositions, denoted by $\models_c P$.

Note that the definition of the canonical interpretation of program P is independent of its syntax in the sense that the definition does not refer to the structure of the program. So the definition can be extended so that it can be applied to non-deterministic programs and temporal formulas.

Example 1. For the propositional formula, P_1: $\neg A \leftrightarrow \bigcirc B$, which can be treated as a non-deterministic program, we have $\Phi = \{A, B\}$, and P_1 has the following canonical interpretation sequences of length 2, $< \phi, \{B\} >$, $< \phi, \{A, B\} >$, $< \{B\}, \{B\} >$, $< \{B\}, \{A, B\} >$, $< \{A\}, \phi >$, $< \{A\}, \{A\} >$, $< \{A, B\}, \phi >$, and $< \{A, B\}, \{A\} >$.

P_1 is satisfiable but not valid under the canonical interpretation on propositions because a canonical interpretation sequence, $< \phi, \phi >$, does not satisfy it.

Note that a program P can be satisfied by several different canonical models on propositions so program P has, possibly, different meanings under different models. Therefore, it is important to choose a model which satisfies the intended meaning of a program P, and this is the topic of Section 5.

Since the canonical model is basically equivalent to the basic model except that the latter acts on the fixed set V of variables and the fixed set Π of propositions, whereas the former acts on the set of variables and the set of propositions within a concrete program. $\exists x : p(x)$ can be renamed as a formula $p(y)$ (or $p[y/x]$) with a free variable y by renaming x as y.

Lemma 2. Let $p(y)$ be a renamed formula of $\exists x : p(x)$. Then, $\exists x : p(x)$ is satisfiable if and only if $p(y)$ is satisfiable. Furthermore, any model of $p(y)$ is a model of $\exists x : p(x)$.

4 Framing

In this section, we first define some new assignments which are required by framing, then we define frame operators; and finally, we present a minimal model-based approach for framing.

Suppose $S = \{x_1, ..., x_n\}(S \subset V)$ is a set of state variables within a program P. Note that variables bound by quantifiers can always be given distinct names by renaming them as necessary.

Definition 1. *(new assignments)*

(1) $x_i \Leftarrow e \stackrel{\text{def}}{=} x_i = e \wedge p_i \quad (0 \leq i \leq n, e \neq nil)$
(2) $x_i \ o{=}^+ \ e \stackrel{\text{def}}{=} \bigcirc x_i = e \wedge \bigcirc p_i$
(3) $x_i :=^+ e \stackrel{\text{def}}{=} x_i \ o{=}^+ e \wedge skip$

where p_i is an atomic proposition associated with state variable x_i $(0 \leq i \leq n)$ and cannot be used for other purposes.

The meanings of these assignment operators are similar to those presented in Section 3, but they render some propositions *true* besides assigning some values to variables in the same unit of time. It is now time to define the assignment flag

$$af(x_i) \stackrel{\text{def}}{=} p_i$$

where proposition p_i associated with variable x_i is the same as in Definition 1, and cannot be used for other purposes. As expected, whenever $x_i \Leftarrow b$ is encountered, p_i is set to be *true*, hence $af(x_i)$ is *true* whereas if no assignment to x_i takes place, p_i is unspecified. In this case, we will use a minimal model to force it to be $false$.

Armed with the assignment flag, we can define state frame and interval frame operators. Intuitively, when a variable is framed at a state, its value remains unchanged if no assignment is encountered at that state. A variable is framed over an interval if it is framed at every state over the interval.

Definition 2. *(looking back framing)*

(1) $lbf(x_k) \stackrel{\text{def}}{=} \neg af(x_k) \rightarrow \exists b : (\ominus x_k = b \wedge x_k = b)$
(2) $frame(x_k) \stackrel{\text{def}}{=} \Box(more \rightarrow \bigcirc lbf(x_k))$
(3) $frame(x_1, ..., x_n) \stackrel{\text{def}}{=} frame(x_1) \wedge ... \wedge frame(x_n)$

We interpret programs using minimal models. Let $\sigma =< s_0, ... >$ be an interval, and $s_i = (I_v^i, I_p^i)$; I_v^i is defined as in Section 2.2 and I_p^i is the canonical interpretation defined as in Section 3.2 but the sequence of interpretations on propositions of σ, $\sigma_p =< I_p^0, ..., >$, is required to be a minimal canonical sequence, as defined in the next section. Armed with framing operator, the synchronized communication construct $await(c)$ can be defined as follows:

Definition 3. $await(c) \stackrel{\text{def}}{=} frame(V_c) \wedge halt(c)$ whereV_c represents all dynamic variables contained in c.

5 Minimal Model

5.1 The Minimal Satisfaction Relation

In this section, we discuss semantics of framed programs. As before, let V denote the set of all variables. A dynamic variable $x \in V$ is said to be framed in a

program p if $frame(x)$ or $lbf(x)$ is contained in p. A program p is said to be framed if p contains at least one framed variable. In general, a framed program is non-deterministic under the canonical model. Consequently, a framed program can inductively be defined, as follows

- For any variable $x \in V$ and any well-formed expression e, $x = e$, $x \Leftarrow e$, and $empty$ are framed programs.
- $lbf(x)$, and $frame(x)$ are framed programs.
- If $p, q, p_1, ..., p_m$ are framed programs, then so are the followings:

  $\bigcirc p$, $\Box p$, $p \wedge q$, $p; q$, $if\ b\ then\ p\ else\ q$, $while\ b\ do\ p$, $p \| q$, $(p_1, ..., p_m)\ prj\ q$, and $\exists x : p$.

Fact 1.

$$\begin{array}{rl} EQFR & x_i = e_i \equiv p_{x_i} \wedge x_i = e_i \vee \neg p_{x_i} \wedge x_i = e_i \\ LBF & lbf(x_i) \equiv p_{x_i} \vee \neg p_{x_i} \wedge x_i = \ominus x_i \end{array}$$

Proof:

EQFR is obviously true. We only prove LBF.

$$\begin{array}{rll} lbf(x_i) & \equiv \neg af x_i \rightarrow \exists b : \ominus x_i = b \wedge x_i = b & \\ & \equiv \neg p_{x_i} \rightarrow \exists b : \ominus x_i = b \wedge x_i = b & \\ & \equiv \neg p_{x_i} \rightarrow \ominus x_i = a \wedge x_i = a & \text{Lemma 2} \\ & \equiv \neg p_{x_i} \rightarrow x_i = \ominus x_i\ (\neq nil) & \\ & \equiv p_{x_i} \vee \neg p_{x_i} \wedge x_i = \ominus x_i & \end{array}$$

By EQFR and LBF, when we reduce a framed program p, whenever $x_i = e_i$ occurs in p, it is replaced by $p_{x_i} \wedge x_i = e_i \vee \neg p_{x_i} \wedge x_i = e_i$; whereas whenever $lbf(x_i)$ occurs in p, it is replaced by $p_{x_i} \vee \neg p_{x_i} \wedge x_i = \ominus x_i$. Then we can reduce p under the canonical model as usual.

A framed program p can be a non-deterministic program. There may be several models satisfying the program under the canonical models.

Example 2.

$$\begin{array}{l} \quad frame(x) \wedge x = 1 \wedge len(1) \\ \equiv \Box(more \rightarrow \bigcirc lbf(x)) \wedge x = 1 \wedge \bigcirc(empty) \\ \equiv (more \rightarrow \bigcirc lbf(x)) \wedge \odot \Box(more \rightarrow \bigcirc lbf(x)) \wedge x = 1 \wedge more \wedge \bigcirc(empty) \\ \equiv \bigcirc lbf(x) \wedge \bigcirc \Box(more \rightarrow \bigcirc lbf(x)) \wedge x = 1 \wedge \bigcirc(empty) \\ \equiv (p_x \wedge x = 1 \vee \neg p_x \wedge x = 1) \wedge \bigcirc(lbf(x) \wedge empty) \end{array}$$

Thus,

$$p_c \equiv p_x \wedge x = 1 \vee \neg p_x \wedge x = 1$$

$$\begin{array}{rl} p_f & \equiv lbf(x) \wedge empty \\ & \equiv (p_x \vee \neg p_x \wedge x = 1) \wedge empty \\ & \equiv p_x \wedge empty \vee \neg p_x \wedge x = 1 \wedge empty \end{array}$$

Hence, four models given below can satisfy the program.

$$\sigma_1 =< (\{p_x\}, \{x:1\}), (\{p_x\}, \phi) >, \sigma_2 =< (\{p_x\}, \{x:1\}), (\phi, \{x:1\}) >$$

$$\sigma_3 =< (\phi, \{x:1\}), (\{p_x\}, \phi) >, \sigma_4 =< (\phi, \{x:1\}), (\phi, \{x:1\}) >$$

As seen, a framed program can have a number of canonical models. Thus, a problem we have to face is how to choose a model to satisfy the intended meaning of a program. We interpret framed programs using minimal models.

Definition 4. Let p be a framed program, and $\Sigma_p = \{\sigma | \sigma \models_c p\}$. Let $\sigma_p =< I_p^0, I_p^1, ... >$, $\sigma_1, \sigma_2 \in \Sigma_p$. We define

- $\sigma_{1p} \sqsubseteq \sigma_{2p}$ iff $I_{1p}^i \subseteq I_{2p}^i$ and $|\sigma_1| = |\sigma_2|$ for all $i, 0 \leq i \leq |\sigma_1|$
- $\sigma_1 \sqsubseteq \sigma_2$ iff $\sigma_{1p} \sqsubseteq \sigma_{2p}$
- $\sigma_1 \sqsubset \sigma_2$ iff $\sigma_1 \sqsubseteq \sigma_2$ and $\sigma_2 \not\sqsubseteq \sigma_1$

Example 3.

$$< (\phi, \{x:1\}) >=< (\phi, \{x:1\}) >, < (\{p_x\}, \phi) > \sqsupset < (\phi, \{x:1\}) >$$

Definition 5. (the minimal satisfaction relation)
Let p be a program, and (σ, i, k, j) be an interpretation. Then the minimal satisfaction relation $\models_m$ is defined as

$(\sigma, i, k, j) \models_m p$ iff $(\sigma, i, k, j) \models_c p$ and there is no σ' such that $\sigma' \sqsubset \sigma$ and $(\sigma', i, k, j) \models_c p$.

A program p is satisfied by a model σ under relation $\models_m$, denoted by $\sigma \models_m p$, if $(\sigma, 0, 0, |\sigma|) \models_m p$. A model σ is a minimal model of program p if $\sigma \models_m p$.

The relations $\equiv_m$ and $\approx_m$ can be defined similarly to the relations $\equiv$ and $\approx$. $p \equiv_m q$ iff for all σ, all k, $0 \leq k \preceq |\sigma|$, $(\sigma, 0, k, |\sigma|) \models_m p \Leftrightarrow (\sigma, 0, k, |\sigma|) \models_m q$. $p \approx_m q$ iff for all σ, $\sigma \models_m p \Leftrightarrow \sigma \models_m q$. The relations $\equiv_m$ and $\approx_m$ are also equivalence relations over the set of programs. That is, they are reflexive, symmetric and transitive.

Note that the definition of the minimal model of a program p is also independent of its syntax in the sense that the definition does not refer to the structure of the program, and can be applied to temporal formulas.

Example 4. The program p in Example 2 has only one minimal model $\sigma_4 = (< \phi, \{x:1\}), (\phi, \{x:1\}) >$. The formula P_1 in Example 1 has only two minimal models, namely, $< \phi, \{B\} >$ and $< \{A\}, \phi >$.

The intended meaning of a program p is captured by its minimal model. For instance, if p is $x_1 \Leftarrow 1 \wedge frame(x_1) \wedge len(1)$ then under the minimal model, $x_1 = 1$ defined at both state s_0 and s_1, this is the intended meaning of p. However, within only the canonical model, p_{x_1} is unspecified at state s_1, so it could be *true* at s_1. This causes x_1 to be unspecified at state s_1. Therefore, x_1 could be any value from its domain.

5.2 Normal Form

Definition 6. A framed program q is in normal form if

$$q \stackrel{\text{def}}{=} \bigvee_{i=1}^{k} q_{ei} \wedge empty \vee \bigvee_{j=1}^{h} q_{cj} \wedge \bigcirc q_{fj} \tag{5.1}$$

where $k, h \geq 0$ $(k + h \geq 1)$ and

- for all $1 \leq j \leq h$, $\bigcirc q_{fj}$ are lec-formulas and q_{fj} are programs.
- q_{cj} $(j \leq h)$ and q_{ei} $(i \leq k)$ are *true* or all state formulas of the form:

$$(x_1 = e_1) \wedge ... \wedge (x_l = e_l) \wedge \dot{p}_{x_1} \wedge ... \wedge \dot{p}_{x_m}$$

  where $e_i \in D$ $(1 \leq i \leq l)$ and $\dot{p}_x$ denotes p_x or $\neg p_x$ and $l \geq 0$ and $m \geq 0$ and $k + h \geq 1$. Notice that, $q_{cj_1} \not\equiv q_{cj_2}$ if $j_1 \neq j_2$, otherwise they can be merged into one by taking the common factor.

In some circumstances, we simply write $q_e \wedge empty$ instead of $\bigvee_{i=1}^{k} q_{ei} \wedge empty$. Also, we call conjuncts, $q_{ei} \wedge empty$, $q_{cj} \wedge \bigcirc q_{fj}$, basic products; the former is called terminal product whereas the latter is called future products. Further, we call q_{ei}, q_{cj} present components, $\bigcirc q_{fj}$ future components of basic products.

Theorem 3. If p is a framed program, then there is a program q as defined in (5.1) such that

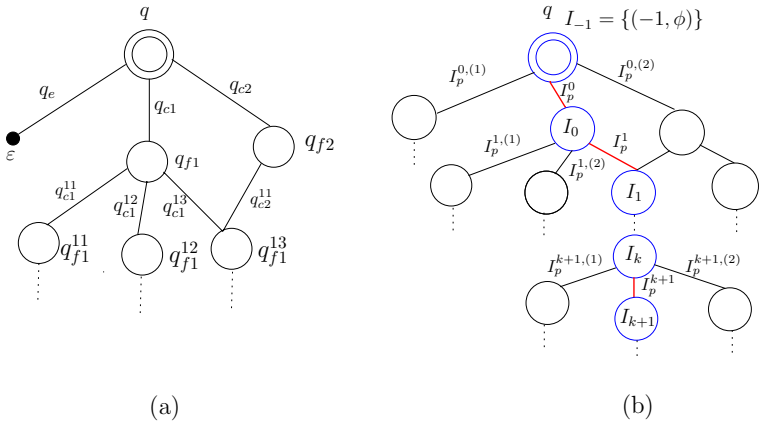
$$p \equiv q$$

Theorem 4. Let $q \equiv \bigvee_{i=1}^{k} q_{ei} \wedge empty \vee \bigvee_{j=1}^{h} q_{cj} \wedge \bigcirc q_{fj}$ be the normal form of a framed program q. If p_x and $x = e'$, where $e' \neq e$ $(e', e \in D)$, are not contained in q_{ei} $(1 \leq i \leq k)$ and q_{cj} $(1 \leq j \leq h)$, then

$$(p_x \vee \neg p_x \wedge x = e) \wedge q \equiv_m \neg p_x \wedge x = e \wedge q$$

The proofs of the above two theorems can be found in [3]. Armed with the normal form, a program q can be decomposed to a so called Normal Form Graph(NFG) as follows:

Initially, the root (denoted by a small double circle) of the Graph is labelled by program q, each basic product in the normal form of q becomes a son of q. With the terminal product, the edge labelled by present component q_e and a terminal vertex (a small black dot) labelled by ε without appearing of empty; and with the future product, the edge labelled by q_{cj} and the next vertex (a small circle) labelled by next component q_{fj}. Then, q_{fj} can further be reduced to a sub-graph of q and so on. If two vertices are identical, we merge them into one. It is clear that if q has only finite models, its NFG is also finite. A normal form graph is shown in Fig.2(a). Note that a sup-script of a present component denotes the reduction level on a path in NFG.



(a) (b)

Fig. 2. NFG

5.3 Existence Theorems of Minimal Models

In this section, we investigate the existence of minimal models of a satisfiable framed program. Two theorems are proved. The first is the one with sufficiency conditions whereas the second is the one with necessity and sufficiency conditions.

Theorem 5. Let p be a satisfiable framed program (which may be non-terminating, and/or non-deterministic). If, (1) p has at least one finite model or (2) p has finitely many models, then p has at least one minimal model on propositions.

The proof of the theorem can be found in [3] and is omitted here.

Theorem 6. Let p be a satisfiable framed program (which can be non-terminating, and/or non-deterministic), then p has at least one minimal model on propositions.

Proof:
In order to distinguish operations between sequences and sets, we denote a finite canonical interpretation sequence $(I_p^0, ..., I_p^k)$ by $\overline{I}_k$, and its corresponding coded set $\{(0, I_p^0), ..., (k, I_p^k)\}$ by I_k. Thus, for an arbitrary canonical interpretation sequence $\overline{I} = (I_p^0, ..., I_p^h)$ $(h \in N_0)$, its corresponding coded set is $I = \{(0, I_p^0), ..., (h, I_p^h)\}$.

For convenience, we need to add extra information to the NFG of a program p. First, the label of each edge, i.e, present component in the normal form of p, e.g. p_{ej} or p_{ci} is changed to corresponding canonical interpretation on propositions I_p^h for some $h \geq 0$ ($(h-1)^{th}$ edge from root on a path) ignoring program variables. For instance, if $p_{ci} \equiv p_1 \wedge x_1 = 1 \wedge p_2 \wedge p_3 \wedge x_3 = 2$, then $I_p^h = \{p_1, p_2, p_3\}$. Second, a node is given an extra label I_k. The initial node is labelled by $I_{-1} = \{(-1, \phi)\}$. With a node I_k ($(k-2)^{th}$ node from root), to find out the next edge with minimal canonical interpretation, we define a function $e - min$ as follows.

$e-min(I_k) = min\{I_p^{k+1,(1)}, ..., I_p^{k+1,(h)}\} = I_p^{k+1,(i)}$ (I_p^{k+1} for short), if $I_p^{k+1,(i)} \subset I_p^{k+1,(j)}$ or $I_p^{k+1,(i)}$ is not comparable with $I_p^{k+1,(j)}$, for $\forall j, i \neq j, 0 \leq i, j \leq h$

where $I_p^{k+1,(1)}, ..., I_p^{k+1,(h)}$ are all canonical interpretations associated with edges departing from node I_k.

By Theorem 3, a framed program p can be reduced to its normal form. Since p is satisfiable, so p has at least one canonical model. Thus, we can construct its NFG as shown in Fig.2(b). Based on NFG, we can construct a calculus T on a canonical interpretation coded set I, $T(I)$ is defined as:

$$T(I) = \{(n, I_p^n) | \exists I_{n-1} \subseteq I, I_p^n = e - min(I_{n-1}), n \geq 0\}$$

$e - min(I_{n-1})$ is a function returning the minimal interpretation among all canonical interpretations associated with edges departing from node I_{n-1}.

Initially I_{-1}=$\{(-1, \emptyset)\}$, then we repeatedly apply calculus T to sets $I_{-1}, I_0, ...$ Thus, we got,
$I_0 = T(I_{-1}) = \{(0, I_p^0)\}$, $I_1 = T(I_0) = \{(0, I_p^0), (1, I_p^1)\}$, $I_2 = T(I_1) = T^2(I_0) = \{(0, I_p^0), (1, I_p^1), (2, I_p^2)\}$,..., $I_n = T(I_{n-1}) = T^n(I_0) = \{(0, I_p^0), (1, I_p^1), ..., (n, I_p^n)\}$. Thus, $I_0 \subseteq I_1 \subseteq I_2 \subseteq \ldots \subseteq I_n \subseteq \ldots$. It is readily to see that T is monotonic.
Let $I = \bigcup_{n=0}^{\infty} I_n$, $\overline{I}_n$ stands for the prefix of minimal interpretation sequence $\overline{I}$.
We now prove the following conclusions.
1. $\overline{I}$ is a canonical interpretation sequence of p.
We first prove $T(I) = I$.
(1) $T(I) \subseteq I$. For $\forall (n+1, I_p^{n+1}) \in T(I)$, by definition, $\exists I_n, I_n \subseteq I, I_p^{n+1} = e - min(I_n)$, so $(n+1, I_p^{n+1}) \in T(I_n)$, i.e. $(n+1, I_p^{n+1}) \in I_{n+1}$. Since $I_{n+1} \subseteq I$, we have $(n+1, I_p^{n+1}) \in I$. Hence $T(I) \subseteq I$.

(2) $I \subseteq T(I)$. That is to prove $\bigcup_{n=0}^{\infty} I_n \subseteq T(I)$. First, we prove $\forall n, n \in N, I_n \subseteq T(I)$. Suppose $(n, I_p^n) \in I_n$, then $(n, I_p^n) \in T(I_{n-1})$. Since $I_{n-1} \subseteq I$, by definition, $I_p^n = e - min(I_{n-1})$, so $(n, I_p^n) \in T(I)$, hence $I_n \subseteq T(I)$. In addition, for all $i \in N$, if $(i, I_p^i) \in \bigcup_{n=0}^{\infty} I_n$, then $\exists n \in N$ such that $(i, I_p^i) \in I_n$. Since $I_n \subseteq T(I)$, so $(i, I_p^i) \in T(I)$, $\bigcup_{n=0}^{\infty} I_n \subseteq T(I)$, therefore $I \subseteq T(I)$.
By the above, we have $T(I) = I$. Thus, I is a fix-point of T and $\overline{I}$ is a canonical interpretation sequence of program p.

2. Let $M = \{\sigma | \sigma \models_c p\}$ and $\sigma \in M$, $\sigma_p = \overline{I}$. Then σ is a minimal model of p.
Suppose $\exists \sigma' \in M, \sigma' \sqsubseteq \sigma$. We prove $\sigma = \sigma'$ by induction on $I_p^n = {I_p}^n$.
(1) Since $\sigma' \sqsubseteq \sigma$, so $\sigma_p \sqsubseteq \sigma_p$, i.e. ${I_p}^i \subseteq I_p^i$ for all i, $0 \leq i \leq |\sigma|$. Thus ${I_p}^0 \subseteq I_p^0$, by definition of T, $I_p^0 \subseteq {I_p}^0$, therefore $I_p^0 = {I_p}^0$.
(2) Suppose for $n \leq k$ ($0 \leq k \leq |\sigma|$), $I_p^h = {I_p}^h$ ($0 \leq h \leq k$). Let $n = k+1$. Since ${I_p}^{k+1} \subseteq I_p^{k+1}$, on the other hand, by definition of T, $I_p^{k+1} \subseteq {I_p}^{k+1}$, so $I_p^{k+1} = {I_p}^{k+1}$. Therefore $\sigma = \sigma'$.
In conclusion, σ is a minimal model of program p.

6 Example

In this section, an example is given to show how to apply the minimal model to interpret a framed program. Let $p \equiv frame(x) \wedge x = 1 \wedge if\ p_x\ then\ y \circ= 1\ else\ y \circ=^+ 2 \wedge len(1)$. The following is a complete reduction process of program p. The sup-scripts of components denote the reduction levels and positions.

$$
\begin{aligned}
p &\equiv frame(x) \wedge x = 1 \wedge if\ p_x\ then\ y \circ= 1\ else\ y \circ=^+ 2 \wedge len(1) \\
&\equiv \Box(more \rightarrow \bigcirc lbf(x)) \wedge x = 1 \wedge (p_x \wedge y \circ= 1 \ \vee\ \neg p_x \wedge y \circ=^+ 2) \wedge \\
&\quad \bigcirc(empty) && \text{definition 2} \\
&\equiv (more \rightarrow \bigcirc lbf(x)) \wedge \odot \Box(more \rightarrow \bigcirc lbf(x)) \wedge x = 1 \wedge \\
&\quad (p_x \wedge y \circ= 1 \ \vee\ \neg p_x \wedge y \circ=^+ 2 \wedge \bigcirc(empty) \wedge more && \text{FE2} \\
&\equiv \bigcirc lbf(x) \wedge \bigcirc \Box(more \rightarrow \bigcirc lbf(x)) \wedge x = 1 \wedge \\
&\quad (p_x \wedge y \circ= 1 \ \vee\ \neg p_x \wedge y \circ=^+ 2) \wedge \bigcirc(empty) && \text{FW1} \\
&\equiv x = 1 \wedge \bigcirc(lbf(x) \wedge empty) \wedge (p_x \wedge y \circ= 1 \ \vee\ \neg p_x \wedge y \circ=^+ 2) \\
&\equiv x = 1 \wedge \bigcirc(lbf(x) \wedge empty) \wedge p_x \wedge y \circ= 1 \vee \\
&\quad x = 1 \wedge \bigcirc(lbf(x) \wedge empty) \wedge \neg p_x \wedge y \circ=^+ 2 \\
&\equiv x = 1 \wedge \bigcirc(lbf(x) \wedge empty) \wedge p_x \wedge \bigcirc y = 1 \vee \\
&\quad x = 1 \wedge \bigcirc(lbf(x) \wedge empty) \wedge \neg p_x \wedge \bigcirc y \Leftarrow 2 && \text{Next assignment,def1(3)} \\
&\equiv x = 1 \wedge p_x \wedge \bigcirc(lbf(x) \wedge y = 1 \wedge empty) \vee x = 1 \wedge \neg p_x \wedge \bigcirc(lbf(x) \wedge y \Leftarrow 2 \wedge empty)
\end{aligned}
$$

The last formula shows that p is in a well-reduced (i.e. normal) form $p_c^{0(1)} \wedge \bigcirc p_f^{0(1)} \vee p_c^{0(2)} \wedge \bigcirc p_f^{0(2)}$ at state s_0. Then $p_f^{0(1)}$ and $p_f^{0(2)}$ are continuously re-reduced as follows:

$$
\begin{aligned}
p_c^{0(1)} &\equiv x = 1 \wedge p_x \\
p_f^{0(1)} &\equiv lbf(x) \wedge y = 1 \wedge empty \\
&\equiv (p_x \vee \neg p_x \wedge x = 1) \wedge (p_y \wedge y = 1 \vee \neg p_y \wedge y = 1) \wedge empty && \text{fact1} \\
&\equiv_m \neg p_x \wedge x = 1 \wedge \neg p_y \wedge y = 1 \wedge empty && \text{theorem4} \\
p_c^{0(2)} &\equiv x = 1 \wedge \neg p_x \\
p_f^{0(2)} &\equiv lbf(x) \wedge y \Leftarrow 2 \wedge empty \\
&\equiv (p_x \vee \neg p_x \wedge x = 1) \wedge y \Leftarrow 2 \wedge empty && \text{fact 1,LBF} \\
&\equiv_m \neg p_x \wedge x = 1 \wedge p_y \wedge y = 2 \wedge empty && \text{theorem4, def1(1)}
\end{aligned}
$$

Finally, $p_f^{0(1)}$ is reduced to the form $p_{e(1)} \equiv p_c^{1(1)} \wedge empty$, $p_f^{0(2)}$ is reduced to the form $p_{e(2)} \equiv p_c^{1(2)} \wedge empty$,which indicate that the reduction process of p are successfully completed. Where

$$
\begin{aligned}
p_c^{1(1)} &\equiv \neg p_x \wedge x = 1 \wedge \neg p_y \wedge y = 1, \quad p_f^{1(1)} \equiv empty \\
p_c^{1(2)} &\equiv \neg p_x \wedge x = 1 \wedge p_y \wedge y = 2, \quad p_f^{1(2)} \equiv empty
\end{aligned}
$$

Hence, six models given below can satisfy the program.

$$\sigma_1 =< (\{p_x\}, \{x:1\}), (\{p_x\}, \{y:1\}) >, \sigma_2 =< (\{p_x\}, \{x:1\}), (\{p_x, p_y\}, \{y:1\}) >$$

$$\sigma_3 =< (\{p_x\}, \{x:1\}), (\{p_y\}, \{x:1, y:1\}) >, \sigma_4 =< (\phi, \{x:1\}), (\{p_x, p_y\}, \{y:2\}) >$$

$$\sigma_5 =< (\{p_x\}, \{x:1\}), (\phi, \{x:1, y:1\}) >, \sigma_6 =< (\phi, \{x:1\}), (\{p_y\}, \{x:1, y:2\}) >$$

However, only two minimal models, σ_5 and σ_6, can capture the meaning of program p. This example also shows us that a framed program might have more than one minimal models.

7 Conclusion

This paper presented a framing technique based on an explicit frame operator. A framed program is interpreted by a minimal model. An interpreter was also developed using SICSTUS Prolog for the Framed Tempura. The interpreter employed the framing technique we presented in this paper. It is a workable and useful technique. Because of the space limitation, we cannot introduce the interpreter and the reduction technique in this paper. It will be discussed elsewhere.

References

1. N.Bidoit: *Negation in Rule-based Data Base Languages: A Survey.* Theoretical Computer Science 78 (1991) 3-83, North-Holland.
2. Z.Duan, M.Koutny M., and C.Holt: *Projection in temporal logic programming.* In F. Pfenning, editor, Proceeding of Logic Programming and Automatic Reasoning, Lecture Notes in Artificial Intelligence, a subseries of LNAI, Vol. 822, pp333-344, Springer Verlag, July, 1994.
3. Z.Duan: *An Extended Interval Temporal Logic and A Framing Technique for Temporal Logic Programming.* Ph.D Thesis (Technical Report No.556), University of Newcastle upon Tyne, May 1996.
4. Z.Duan, M.Holcombe, and A.Bell: *A Logic for Biological Systems.* BioSystems 55 (2000) 93-105, Elsevier, 2000.
5. Z.Duan, M.Koutny: *A Framed Temporal Logic Programming Language.* Journal of Computer Science and Technology, Vol.19, No.3, pp.341-351, May, 2004.
6. R.Hale: *Programming in Temporal Logic.* Ph.D. Thesis, 173, (1989) Trini College Computer Laboratory, Cambridge University, Cambridge, England, October, 1988
7. F.Kröger: *Temporal logic of programs.* Springer-Verlag (1987).
8. B.Moszkowski: *Executing temporal logic programs.* Cambridge University Press Cambridge (1986).
9. B. Moszkowski: *Some very compositional temporal properties.* Programming Concepts, Methods, and Calculi, 307–326. Elsevier Science B.V. (North-Holland), 1994.
10. Z.Manner and A.Pnueli: *The temporal logic of reactive and concurrent systems.* Springer-Verlag (1992).
11. H.xie, J.Gong, C.S.Tang: *A Structured Temporal Logic Language XYZ/SE.* J.of Comp.Sci.& Tech.,1991.1

Practical Higher-Order Pattern Unification with On-the-Fly Raising

Gopalan Nadathur and Natalie Linnell

Digital Technology Center and Department of Computer Science and Engineering, Univ. of Minnesota, 4-192 EE/CS Bldg, 200 Union St SE, Minneapolis, MN 55455

Abstract. Higher-order pattern unification problems arise often in computations within systems such as Twelf, λProlog and Isabelle. An important characteristic of such problems is that they are given by equations appearing under a prefix of alternating universal and existential quantifiers. Most existing algorithms for solving these problems assume that such prefixes are simplified to a $\forall\exists\forall$ form by an a priori application of a transformation known as raising. There are drawbacks to this approach. Mixed quantifier prefixes typically manifest themselves in the course of computation, thereby requiring a dynamic form of preprocessing that is difficult to support in low-level implementations. Moreover, raising may be redundant in many cases and its effect may have to be undone by a subsequent pruning transformation. We propose a method to overcome these difficulties. In particular, a unification algorithm is described that proceeds by recursively descending through the structures of terms, performing raising and other transformations on-the-fly and only as needed.

1 Introduction

Higher-order unification, or the unification of typed lambda terms modulo the rules of lambda conversion, is a problem that appears to have poor computational properties: most general unifiers may not exist in relevant instances, complete sets of unifiers may be infinite, the search for such unifiers cannot always be nonredundant and unifiability itself is, in general, undecidable. It seems somewhat of an anomaly, therefore, that effective use has been made of this operation in a variety of applications within metalanguages, logical frameworks and proof assistants such as λProlog [9], Twelf [16] and Isabelle [13]. The answer to this puzzle seems to lie in the fact that good programming practice avoids exercising the pathological cases for this form of unification. A discovery of this kind was made by Dale Miller who observed that occurrences of instantiatable (existential) variables in λProlog programs usually satisfy static conditions that lead to unification computations belonging to what is known as the L_λ or higher-order pattern class [6,12]. For problems in this class, unifiability is decidable and most general unifiers can be provided. Moreover, even though the syntactic restrictions may not be satisfied by all useful programs [5], the unification problems that arise dynamically still usually lie within the L_λ class.

M. Gabbrielli and G. Gupta (Eds.): ICLP 2005, LNCS 3668, pp. 371–386, 2005.

In light of these observations, higher-order pattern unification has adopted a special practical significance. At a coarse level, the unification procedure for simply typed lambda terms that was invented by Huet [3] behaves well on these special problems: it converges on exactly one successful path for each solvable problem and can also be made to terminate in every case. Nevertheless, this procedure has finer-grained characteristics that can be improved: each local step within it still involves examining several competing substitutions and a successful computation may offer only a pre-unifier, conditioned by a solvable but as yet unsolved set of constraints. Both deficiencies can be addressed. Miller has proposed what is ultimately a refinement of Huet's procedure that, for each problem of the L_λ kind, either determines non-unifiability or yields a most general unifier at the end of a non-branching computation [6]. The ideas underlying this procedure has been extended to dependently typed lambda calculi [14,15] and higher-order rewrite systems [11]. A version of the procedure that has a time and space complexity that is linear in the size of the input terms has been developed [18] and it has also been adapted to use explicit substitutions relative to a special grafting interpretation of instantiatable variables [2].

This work is motivated by a desire to exploit higher-order pattern unification in low-level implementations; in particular, in the *Teyjus* implementation of λProlog [10]. While many variants of the original procedure have been described, none of them seems quite suited to this task. In such a setting, it is important that the processing be driven entirely by a recursive examination of the structures of terms. The original procedure that is given by transformation rules has two rules—pruning and raising—that do not possess this character. Another important property of practical unification problems is that they need to be solved under a mixed prefix of quantifiers [7] that are created in the course of computation. Most procedures other than the original one seem to finesse this issue by assuming that existential quantifiers appear only at the top-level, embedded at most under universal quantifiers corresponding to global constants.[1] Now, it is possible to transform arbitrary prefixes into this specialized form by initially applying a raising transformation to existential quantifiers. However, this preprocessing must be done dynamically and it requires at least some additional bookkeeping at runtime. Moreover, much of this kind of raising may be unnecessary and performing it has the potential of making other necessary steps more expensive than they need to be.

This paper is intended to redress this situation. We describe in it a procedure within which unification takes place relative to a mixed quantifier prefix. However, such prefixes are rendered implicit by tagging existential and universal variables with numbers that count the quantifier alternations prior to the ones binding these variables. The processing is oriented around a recursive traversal of the terms to be unified and consists essentially of simplification and variable binding phases. The latter phase is characterized by an on-the-fly application of the pruning and raising rules in which the numeric tags are used to recognize quantifier orders. The algorithm we describe is meant for use with lambda terms

[1] See Section 7 for a further discussion of this matter.

that are polymorphically typed. In this setting it may sometimes be necessary to treat η-convertibility dynamically. Our algorithm supports this capability.

The rest of this paper is organized as follows. The next two sections describe the higher-order pattern unification problem and present a naive procedure for solving it. This procedure is then refined into a more sophisticated form. One refinement, described in Section 4, makes the quantifier prefix implicit. Another refinement, developed in Section 5, factors the algorithm into simplification and variable binding phases. In Section 6 we discuss some aspects relevant to the practical realization of the procedure. We illustrate our procedure in Section 7 and also contrast its behaviour with previously described ones for the same problem. We conclude the paper with a brief discussion of continuing work.

2 Logical Preliminaries

The lambda terms that are of interest to us may contain universal, existential and lambda bound variables. We initially use the symbol u and x, possibly with subscripts, to denote variables of the first two kinds; later the status of such variables will be determined by an explicit quantifier prefix. A variable occurrence that is bound by an abstraction will be represented, following the scheme due to de Bruijn, by a positive number that counts the abstractions up to and including the one binding that particular occurrence. We bunch together a sequence of abstractions and, likewise the arguments in a sequence of applications.

Formally, our terms are given by the syntax rule

$$t ::= x \mid u \mid i \mid \lambda(n, t) \mid t(\overline{t})$$

in which i represents positive integers, n represents natural numbers and $\overline{t}$ represents a sequence of comma separated terms. In an expression of the form $\lambda(i, t)$, i denotes the number of abstractions. In a schematic presentation, we shall allow this number also to be 0, in which case the term is identical to t. Applications are written in a manner reminiscent of first-order syntax rather than in the usual curried form: thus, the term $t_1(t_2, t_3)$ is equivalent to what we would ordinarily write as $((t_1\ t_2)\ t_3)$ in a higher-order language. We actually think of $\overline{t}$ as a vector, writing $|t|$ to denote its length and $\overline{t}[i]$ to refer to its i-th argument. Once again, in a schematic presentation, we shall let $\overline{t}$ be an empty sequence; the expression $t(\overline{t})$ in this case matches with whatever matches with t.

Two terms are considered equal if they can be β-converted to each other. Testing for such equality is based on head normal forms. A term in this form has the structure $\lambda(n, a(\overline{t}))$ where a, called the head of the term, is a universal or existential variable or a de Bruijn index. Although we do not display types explicitly anywhere, we assume that our terms are typed in an ML-like sense. A consequence of this assumption is that every term in fact reduces to a head normal form. A discussion of procedures for producing such a form that exploit explicit substitution notations for the lambda calculus may be found in [4].

Our operative notion of equality also includes η-conversion. This makes it necessary to sometimes consider the η-expansion of terms. We write $t \uparrow j$ to

denote the 'lifting' of the term t over j (new) abstractions; this operation, in effect, increments the free variables in t by j. For atomic terms, the computation actually has a simple form: if t is a universal or existential variable then $t\uparrow j = t$ and if it is a de Bruijn index then $t\uparrow j = t + j$. For existential variables, this definition reflects a logical interpretation as opposed to the grafting one in [1]; in particular, these variables cannot be instantiated by terms containing de Bruijn indices bound by external abstractions. The lifting operation is extended to a sequence of terms: if $|\overline{t}| = n$ then $\overline{t}\uparrow j = \overline{t'}$ where $\overline{t'}$ is a sequence of length $n + j$ with $\overline{t'}[i] = \overline{t}[i]\uparrow j$ for $1 \leq i \leq n$ and $\overline{t'}[i] = j - (i - (n+1))$ for $n < i \leq n+j$. This definition applies even when $|\overline{t}| = 0$ and we allow the expressions $t\uparrow j$ and $\overline{t}\uparrow j$ to be used also when j is 0 in which case they are identical to t and $\overline{t}$, respectively. Using these operations, the j-fold η-expansion of the term $\lambda(n, a(\overline{t}))$ in head-normal form is given by the term $\lambda(n + j, (a\uparrow j)(\overline{t}\uparrow j))$, also in head-normal form. Such an η-expansion is sensible only if the type of a allows for it.

A unification problem is defined by two components: a list of equations such that the two terms in each equation have the same type and a quantifier prefix that scopes over the list. In depicting equation lists, we shall use *nil* to denote the empty list and :: to denote an infix, right associative operator that allows a new equation to be added to the front of an existing list. Universal and existential variables may appear in the equations and these are expected to be captured by a quantifier of the corresponding force appearing in the prefix. Intuitively, universal variables correspond to constants whereas existential variables may be instantiated towards solving the equations in a situation where an equation is considered solved when it relates two (closed) terms that are equal modulo β- and η-conversion. A solution to or a unifier for a given problem is a substitution for the existential variables that reduces the list of equations to a solved form.

However, the prefix structure restricts what substitutions may legitimately be made for a given existential variable: these must be closed terms, *i.e.*, terms with no existential variables or unbound de Bruijn indices, in which the only constants that are allowed to appear are ones whose quantifier scope includes that of the quantifier governing the existential variable. This constraint determines, for example, that the unification problem given by $\exists x \forall u((x = u) :: nil)$ has no solutions whereas the problem $\forall u \exists x((x = u) :: nil)$ has the solution $\{\langle x, u\rangle\}$.

In the description above, solutions are required to be closed substitutions. We relax this requirement to permit an existential variable to appear in a substitution term provided its quantifier scope includes all the universal quantifiers within the scope of the quantifier governing the variable being substituted for. Thus, the unification problem $\forall u \exists x_1 \exists x_2((x_1 = x_2) :: nil)$ now has $\{\langle x_1, x_2\rangle\}$ as a solution. We further allow a prefix to be extended by the introduction of existential quantifiers over variables not occurring in the equation list and consider solutions to such modified problems to be solutions to the original problem. For instance, the earlier problem may be modified to $\forall u \exists x_1 \exists x_2 \exists x_3((x_1 = x_2) :: nil)$ and then has the solution $\{\langle x_1, x_3\rangle, \langle x_2, x_3\rangle\}$. Substitutions with existential variables may be thought of as schemas for generating (legitimate) closed solutions. Towards making this idea precise, we first note that substitutions are given by

a finite set of variable-term pairs where each pair pertains to a distinct variable and where the term obeys the constraints imposed by a relevant quantifier prefix and that the application of a substitution θ to a term t is denoted by $\theta(t)$. Further, the composition of two substitutions θ_1 and θ_2, written $\theta_1 \circ \theta_2$, is given as follows: $\langle x, t\rangle$ belongs to $\theta_1 \circ \theta_2$ just in case $\langle x, s\rangle \in \theta_2$ and $t = \theta_1(s)$ or there is no pair pertaining to x in θ_2 and $\langle x, t\rangle \in \theta_1$. Now, it is easily seen that if θ is a solution to a unification problem $\mathcal{Q}(E)$ that is legitimate with respect to a prefix $\mathcal{Q}'$ that extends $\mathcal{Q}$ in the permitted fashion and ρ is another substitution that is legitimate with respect to $\mathcal{Q}'$, then $\rho \circ \theta$ is also a solution to $\mathcal{Q}(E)$. The substitution θ then constitutes a schema in the sense that it represents all the closed solutions that can be obtained from it by such a composition. Our interest is eventually only in closed substitutions pertaining to the existential variables appearing in $\mathcal{Q}$, the original prefix. Letting $\theta_1 =_{\mathcal{Q}} \theta_2$ represent the proposition that θ_1 and θ_2 agree on these variables, we say that θ is a most general unifier for $\mathcal{Q}(E)$ just in case it is a solution to this problem that is legitimate with respect an extended prefix $\mathcal{Q}'$ and for every closed solution ρ_1 of $\mathcal{Q}(E)$ there is a substitution ρ_2 that is legitimate with respect to $\mathcal{Q}'$ and such that $\rho_1 =_{\mathcal{Q}} \rho_2 \circ \theta$.

Unification problems may be higher-order in the sense that function variables may by existentially quantified in the prefix. A particular problem illustrating this facet is the following: $\forall u_1 \forall u_2 \exists x((x(u_2) = u_1(u_2)) :: nil)$. This problem has the two incomparable solutions $\{\langle x, \lambda(1, u_1(1))\rangle\}$ and $\{\langle x, \lambda(1, u_1(u_2))\rangle\}$. A *higher-order pattern unification* problem is one where each occurrence of an existential variable in the equation list satisfies the following syntactic constraint: if it appears applied to arguments, then each of these arguments is a distinct de Bruijn index or a distinct universal variable whose quantifier appears within the scope of the quantifier binding the existential variable. The expression $x(u_2)$ in the unification problem just considered does not satisfy this constraint. However, it does obey this requirement in the problem $\forall u_1 \exists x \forall u_2((x(u_2) = u_1(u_2)) :: nil)$ with a modified prefix. The result of this change is that the problem now has only one solution: the substitution $\{\langle x, \lambda(1, u_1(1))\rangle\}$. The existence of most general solutions is a general property of higher-order pattern unification problems [6].

The quantifier prefixes governing the list of equations in a unification problem usually arise from reasoning over predicate formulas in a larger logical system. While our presentation appears to portray these prefixes as fixed entities, it is important to bear in mind that they evolve during computation in practice.

3 Unification via Transformations

We present the first version of our unification procedure in the form of rewrite rules that transform tuples of the form $\langle \mathcal{Q}(E), \theta\rangle$ where $\mathcal{Q}(E)$ is a unification problem and θ is a substitution. In the initial configuration, the first component of the tuple is the problem that we want solved and θ is the empty substitution. The purpose of the rewrite rules is to reduce the differences between the terms in the equations. They may postulate substitutions towards this end and these are accumulated in the second component of the tuple. New existential variables may

(1) $\langle \mathcal{Q}((\lambda(n,t) = \lambda(n,s)) :: E), \theta\rangle \longrightarrow \langle \mathcal{Q}((t = s) :: E), \theta\rangle$, provided $n > 0$.

(2) $\langle \mathcal{Q}((\lambda(n,t) = \lambda(m,s)) :: E), \theta\rangle \longrightarrow \langle \mathcal{Q}((t = \lambda(m-n,s)) :: E), \theta\rangle$,
provided $n > 0$ and $m > n$.

(3) $\langle \mathcal{Q}((a(\overline{t}) = \lambda(m,s)) :: E), \theta\rangle \longrightarrow \langle \mathcal{Q}(((a{\uparrow}m)(\overline{t}{\uparrow}m) = s) :: E), \theta\rangle$,
provided a is a de Bruijn index or a universal variable and $m > 0$.

(4) $\langle \mathcal{Q}((f(\overline{t}) = \lambda(n, g(\overline{s}))) :: E), \theta\rangle \longrightarrow \langle \mathcal{Q}((f(\overline{t}{\uparrow}n) = g(\overline{s})) :: E), \theta\rangle$,
provided f and g are existential variables and $n > 0$.

(5) $\langle \mathcal{Q}((a(\overline{t}) = a(\overline{s})) :: E), \theta\rangle \longrightarrow \langle \mathcal{Q}((\overline{t}[1] = \overline{s}[1]) :: \ldots :: (\overline{t}[n] = \overline{s}[n]) :: E), \theta\rangle$,
where $|\overline{t}| = n$, provided a is a de Bruijn index or a universal variable.

(6) $\langle \mathcal{Q}_1 \exists f \mathcal{Q}_2((f(\overline{y}) = \lambda(n, a(t_1, \ldots, t_m))) :: E), \theta\rangle \longrightarrow$
$\langle \mathcal{Q}_1 \exists h_1 \ldots \exists h_m \exists f \mathcal{Q}_2((h_1(\overline{y}{\uparrow}n) = t_1) :: \ldots :: (h_m(\overline{y}{\uparrow}n) = t_m) :: \theta'(E)), \theta' \circ \theta\rangle$,
where $\theta' = \{\langle f, \lambda(|\overline{y}| + n, a(h_1(|\overline{y}| + n, \ldots, 1), \ldots, h_m(|\overline{y}| + n, \ldots, 1)))\rangle\}$,
provided a is universally quantified in $\mathcal{Q}_1$ and f does not appear in $\overline{t}$.

(7) $\langle \mathcal{Q}_1 \exists f \mathcal{Q}_2((f(\overline{y}) = \lambda(n, a(t_1, \ldots, t_m))) :: E), \theta\rangle \longrightarrow$
$\langle \mathcal{Q}_1 \exists h_1 \ldots \exists h_m \exists f \mathcal{Q}_2((h_1(\overline{y}{\uparrow}n) = t_1) :: \ldots :: (h_m(\overline{y}{\uparrow}n) = t_m) :: \theta'(E)), \theta' \circ \theta\rangle$,
where $\theta' = \{\langle f, \lambda(|\overline{y}| + n, a'(h_1(|\overline{y}| + n, \ldots, 1), \ldots h_m(|\overline{y}| + n, \ldots, 1)))\rangle\}$
for $a' = a{\downarrow}(\overline{y}{\uparrow}n)$, provided a appears in $\overline{y}{\uparrow}n$ and f does not appear in $\overline{t}$.

(8) $\langle \mathcal{Q}_1 \exists f \mathcal{Q}_2((f(\overline{y}) = f(\overline{z})) :: E), \theta\rangle \longrightarrow \langle \mathcal{Q}_1 \exists h \exists f \mathcal{Q}_2(\theta'(E)), \theta' \circ \theta\rangle$,
for $\theta' = \{\langle f, \lambda(m, h(\overline{w}))\rangle\}$, where $m = |\overline{y}|$ and $\overline{w} = \overline{\{m - i \mid i \leq m \text{ and } \overline{y}[i] = \overline{z}[i]\}}$.

(9) $\langle \mathcal{Q}_1 \exists f \mathcal{Q}_2 \exists g \mathcal{Q}_3((f(\overline{y}) = g(\overline{z})) :: E), \theta\rangle \longrightarrow$
$\langle \mathcal{Q}_1 \exists f \exists h \mathcal{Q}_2 \exists g \mathcal{Q}_3((f(\overline{y}) = h(\overline{w} + \overline{z})) :: \theta'(E)), \theta' \circ \theta\rangle$,
where $\overline{w} = \overline{\{u \mid \forall u \text{ appears in } \mathcal{Q}_2\}}$ and $\theta' = \{\langle g, h(\overline{w})\rangle\}$,
provided $\mathcal{Q}_2$ contains at least one universal quantifier.

(10) $\langle \mathcal{Q}_1 \exists f \mathcal{Q}_2 \exists g \mathcal{Q}_3((f(\overline{y}) = g(\overline{z})) :: E), \theta\rangle \longrightarrow \langle \mathcal{Q}_1 \exists h \exists f \mathcal{Q}_2 \exists g \mathcal{Q}_3(\theta'(E)), \theta' \circ \theta\rangle$,
for $\theta' = \{\langle f, \lambda(m, h(\overline{u}))\rangle, \langle g, \lambda(n, h(\overline{v}))\rangle\}$,
where $m = |\overline{y}|$, $n = |\overline{z}|$ and $\overline{u} = \overline{w}{\downarrow}\overline{y}$ and $\overline{v} = \overline{w}{\downarrow}\overline{z}$ for $\overline{w} = \overline{y} \cap \overline{z}$,
provided no universal quantifiers appear in $\mathcal{Q}_2$.

Fig. 1. Transformation rules for higher-order pattern unification

be introduced in the process and the quantifier prefix is, in this case, modified to accommodate them. If a reduction sequence succeeds in transforming the equation list to an empty one in this way, then the substitution component is intended to be a most general solution to the original unification problem.

The specific rules defining the procedure appear in Figure 1. As is evident from their lefthand sides, the action carried out by these rules is based on the form of the first equation in a non-empty list. Prior to attempting a match, the two terms in such an equation must be reduced to their head normal forms. The matching process assumes that the equality symbol is symmetric, *i.e.*, it will attempt a match by also interchanging the left and right sides of the equations shown in the patterns. An explanation is warranted with respect to some of the notation for terms used in these patterns. The symbols $\overline{t}$ and $\overline{s}$ used in rules (3)-(7) are intended to match with (possibly empty) sequences of arguments in actual terms and the rules may refer to the lengths of these sequences as well as their components subsequent to the match. The symbols $\overline{y}$ and $\overline{z}$ that are used in rules (6)-(10) match with actual argument sequences only if they further satisfy

the higher-order pattern restriction: each element of such a sequence must be a distinct de Bruijn index or universal variable that is quantified within the scope of the existential quantifier binding the variable that appears as the head of the term. That this condition is satisfied needs to be ascertained only if the problem is not already known to be of the higher-order pattern unification kind and, in this case, the arguments will also have to be reduced to head-normal form prior to the attempted match. Some explanation pertaining to the action parts of the rules is also relevant. The rules (6)-(10) introduce existential quantifiers over variables shown schematically as h, possibly with subscripts. These variables must be different from ones already appearing in the quantifier prefix. In rule (9), the notation $\overline{S}$ where S is a set is used to denote a sequence created from the elements of S and the expression $\overline{w} + \overline{z}$ is used to represent the sequence obtained from the concatenation of two given ones. In rule (10), we write $\overline{y} \cap \overline{z}$ to represent a sequence of the elements common to $\overline{y}$ and $\overline{z}$. The notation $a \downarrow \overline{z}$ where $\overline{z}$ is a sequence of distinct universal variables and de Bruijn indices and a is an element of this sequence that is used in rule (7) corresponds to the de Bruijn index given by $(|\overline{z}| + 1 - i)$ where $\overline{z}[i] = a$. The expression $\overline{w} \downarrow \overline{z}$ used in rule (10) extends this notation to the situation where $\overline{w}$ is a sequence of elements appearing in $\overline{z}$ as follows: if $|\overline{w}| = n$ then $\overline{w} \downarrow \overline{z} = \overline{w}[1] \downarrow \overline{z}, \ldots, \overline{w}[n] \downarrow \overline{z}$.

The procedure that we have described here is essentially a deterministic adaptation of the one in [6] to a situation where the de Bruijn notation is used and where η-expansion is done on demand. Determinism is obtained by imposing a processing order that is based on a recursive traversal of the structures of the terms to be unified. In this context the rules in Figure 1 may be understood as follows. Rules (1) and (2) implement a descent through abstractions. If the number of abstractions at the top-level in the two terms are mismatched, an η-expansion may be needed. This is done explicitly when the head of the non-abstraction term is *rigid* or unchangeable under substitution (rule (3)) or when both terms have existential variables as their heads, *i.e.*, have heads that are *flexible* (rule (4)). In the only remaining case, η-expansion is folded into the substitution generation process (rules (6) and (7)). Once past abstractions, rule (5) reduces the task of unifying two terms with the same rigid head to that of unifying their (possibly empty) sequence of arguments in a pairwise fashion; typing constraints ensure that the lengths of these sequences are the same. Rule (8) solves an equation between two terms with the same flexible head; if these are applications, typing constraints again ensure that they have the same number of arguments. Rule (10) solves such an equation when the two heads are not identical but are bound by existential quantifiers that scope over the same universal quantifiers. Rule (9) applies the *raising* transformation to prepare the ground for rule (10) in case the proviso on quantifier scopes is not met initially. The only remaining case relative to higher-order patterns is that where one of the terms is an application with a flexible head and the other term has a rigid head. Rules (6) and (7) attempt to solve this problem at the root using imitation and projection substitutions, respectively, adding new equations to realize the needed recursion over the arguments of the term with the rigid head.

The formal properties of the procedure are given by the following theorem whose proof is to be found in an extended version of this paper:

Theorem 1. *If $\mathcal{Q}(E)$ represents a unification problem, then the sequence of rule applications starting from $\langle \mathcal{Q}(E), \emptyset \rangle$ must terminate. Further, if the last tuple has the form $\langle \mathcal{Q}'(nil), \theta \rangle$, then θ represents a most general solution to $\mathcal{Q}(E)$. Finally, if $\mathcal{Q}(E)$ is a solvable higher-order pattern unification problem, then the tuple at the end must have such a form.*

Theorem 1 shows that the procedure we have outlined in this section is complete for higher-order pattern unification problems. In a more general situation, however, the procedure may terminate because the proviso on the forms of arguments for terms with flexible heads is not satisfied. This can happen even when the problem embodied in the state has a solution. Our procedure is therefore seen not to be complete for general higher-order unification.

4 Eliminating the Quantifier Prefix

The explicit treatment of quantifier prefixes poses practical difficulties: Prefixes grow and shrink as the result of other logical computations and maintaining them therefore requires run-time effort. Using the prefixes also requires that contextual information be examined in the recursive descent through term structure. It is preferable that such a descent be predicated entirely on local information.

Towards understanding how quantifier prefixes may be obviated, we examine the manner in which they are utilized in the unification procedure. These prefixes are relevant to three tasks: (i) determining whether given variable occurrences are of the existential or universal kind, (ii) ascertaining that the arguments of a flexible term satisfy the scoping requirements of higher-order patterns, and (iii) realizing the raising transformation embodied in rule (9). The first of these tasks can also be accomplished by labelling each variable with its associated type. The only additional information the prefix supplies for the second task is the relative order of quantification. However, this information can be maintained more succinctly by associating with each variable a numeric tag that records the number of times an existential quantifier is immediately followed by a universal one in the prefix up to and including the quantifier binding that occurrence. We assume henceforth that this is done and that the tag for a variable y is given by $l(y)$. The test for the satisfaction of the higher-order pattern constraint becomes a local one with these tags: a universal variable u is quantified within the scope of the quantifier for an existential variable x just in case $l(x) < l(u)$.

The information needed for the raising transformation seems more difficult to encode in a local fashion at the outset: rule (9) requires knowledge of the universal quantifiers that intervene between two existential quantifiers in the prefix and this is information that appears not to be available simply from looking at the two (flexible) terms that are to be unified. There is, however, an important observation to be made about the role of rule (9) in the unification procedure. The purpose of this rule is essentially to prepare the stage for an application

of rule (10) that solves an equation between two flexible terms with distinct heads. From this perspective, the universal variables over which g, the flexible head of one of the pertinent terms in rule (9), is raised can be factored into two kinds: those that appear as arguments of f, the flexible head of the other term, and those that do not so appear. While rule (9) raises g over the latter kind of variables as well, this raising is redundant since the subsequent application of rule (10) prunes them away. Thus, the collection of variables over which g really needs to be raised can be determined simply by looking at the universal variables that appear as arguments of f and checking if they are quantified outside the scope of the quantifier for g. The last aspect, as we have already noted, can be decided by looking at the tags associated with the two variables.

We introduce notation to represent the operation of raising over a restricted collection of variables: If g is an existential variable and $\overline{y}$ is a sequence of distinct universal variables and de Bruijn indices, then $\overline{y}{\Uparrow}g$ will denote the sequence

$$\overline{\{u \mid u \text{ is a universal variable occurring in } \overline{y} \text{ such that } l(u) \leq l(g)\}}.$$

Now, the modified version of rule (9) and rule (10) both involve traversals over the arguments of the flexible terms that can potentially be carried out simultaneously in an implementation. Towards facilitating this possibility, we combine these rules into one new rule labelled (9'):

(9') $\langle (f(\overline{y}) = g(\overline{z})) :: E, \theta\rangle \longrightarrow \langle \theta'(E), \theta' \circ \theta\rangle$,
for $\theta' = \{\langle f, \lambda(m, h(\overline{q} + \overline{v}))\rangle, \langle g, \lambda(n, h(\overline{p} + \overline{u}))\rangle\}$, where $m = |\overline{y}|$, $n = |\overline{z}|$, h is a new existential variable such that $l(h) = l(f)$, $\overline{p} = \overline{y}{\Uparrow}g$, $\overline{q} = \overline{p}{\downarrow}\overline{y}$, and $\overline{v} = \overline{w}{\downarrow}\overline{y}$ and $\overline{u} = \overline{w}{\downarrow}\overline{z}$ for $\overline{w} = \overline{\{a \mid a \text{ appears in both } \overline{y} \text{ and } \overline{z}\}}$, assuming $f \neq g$ and $l(f) \leq l(g)$.

In this rule, $\overline{p}$ collects the universal variables over which g must eventually be raised by looking at the arguments of f and $\overline{q}$ represents a sequence of projections over the arguments of f that is needed to match $\overline{p}$. The calculation of $\overline{p}$ and $\overline{q}$ captures the cumulative effect of raising as per rule (9) and the application of rule (10) relative to the added arguments. The effect of rule (10) corresponding to the original arguments of g and their counterparts in f is reflected in the calculation of $\overline{v}$ and $\overline{u}$, respectively.

The discussions of this section result in a unification procedure that is obtained by modifying the rules in Figure 1 as follows: First, quantifier prefixes are eliminated from unification problems and numeric and type tags are associated with all universal and existential variables. Second, the new existential variables introduced in rules (6)-(8) are all accorded the same numeric tag as f. Third, the choice between rules (7) and (8) when a is a universal variable is made by comparing the numeric tags of a and f, picking (7) if $l(a) \leq l(f)$ and attempting to use (8) otherwise. Finally, rules (9) and (10) are replaced by rule (9').

Let $\mathcal{Q}(E)$ be a unification problem that is presented with an explicit quantifier prefix and let E' be a version of the equations in E in which universal and existential variables are distinguished and labelled with numeric tags consistent with the prefix. We shall then say that E' is obtained from $\mathcal{Q}(E)$ by prefix

erasure. The important property of the unification procedure described in this section is the content of the following theorem whose proof involves using the intended correspondence between tags and prefixes and the relationship between rule (9') in the present system and the earlier rules (9) and (10).

Theorem 2. *Let E' be obtained from $\mathcal{Q}(E)$ by prefix erasure. Then the sequence of rule applications starting from $\langle E', \emptyset\rangle$ must terminate. Further, if the last tuple has the form $\langle nil, \theta\rangle$, then θ is a most general unifier for $\mathcal{Q}(E)$. Finally, if $\mathcal{Q}(E)$ is a solvable higher-order pattern unification problem, then the tuple at the end must have such a form.*

5 Combining Variable Substitution Steps

The procedure we have at this point solves equations involving two flexible terms immediately. However, it resorts to an incremental process when one term is flexible and the other is rigid and has arguments. This character is manifest in the structure of rules (6) and (7) in Figure 1 that solve the problem if possible at the 'root' and introduce new variables and equations towards solving the rest of the problem in subsequent steps. There is a bookkeeping overhead to this approach that can be avoided by combining the sequence of steps into a mechanism that generates a single composite substitution for solving the entire equation. Such a mechanism would obviously involve a traversal of the structure of the rigid term. Rules (6) and (7) already require such a traversal towards ascertaining that the head of the flexible term does not appear in the arguments of the rigid one. Ideally, these two traversals should be folded into one.

Figure 2 presents a set of rewrite rules that embody a realization of the substitution generation process of the kind desired. These rewrite rules have a pseudo-procedural character in that some of them have as side conditions additional computations using the same set of rules; the symbol $\stackrel{*}{\longrightarrow}$ is to be interpreted in them as a sequence of rewritings. In interpreting these rules, all the notational conventions described in conjunction with Figure 1 are to be utilized. We also assume the following additional conventions: $\overline{s}^R$ represents the reverse of the sequence $\overline{s}$, ε matches with the empty sequence and $\varphi(\overline{t})$ denotes the result of applying the substitution φ to each term in the sequence $\overline{t}$. Further, in attempting to match with the first two rules, *i.e.*, the ones for *mksubst*, we require that the second argument be head normalized and, similarly, before matching with the next four rules, the second argument of *bnd* should be put in head normal form. Finally, the satisfaction of side conditions that involve rewriting requires also that the results of the rewriting have the forms shown for the righthand sides.

The rules in Figure 2 are intended for solving an equation of the form $f(\overline{y}) = t$ where f is an existential variable, $\overline{y}$ is a sequence of arguments satisfying the higher-order pattern restriction and t is an arbitrary term. Their usage begins with an attempt to rewrite the expression $mksubst(f, t, \overline{y}, |\overline{y}|)$. The outcome of a successful rewriting will be a substitution with a binding for f. The substitution may also bind other variables: t may contain occurrences of existential variables and the solution to the equation may require that substitutions be made for these

$mksubst(f, \lambda(n, f(\overline{z})), \overline{y}, m) \longrightarrow \{\langle f, \lambda(n+m, h(\overline{w}))\rangle\}$,
where $\overline{w} = \{m+n-i \mid i \leq n+m \text{ and } (\overline{y}\uparrow n)[i] = \overline{z}[i]\}$ and
h is a new existential variable such that $l(h) = l(f)$.

$mksubst(f, t, \overline{y}, m) \longrightarrow \{\langle f, \lambda(m, s)\rangle\} \circ \theta$,
if the head of t is not f and $bnd(f, t, \overline{y}, 0) \xrightarrow{*} \langle \theta, s\rangle$.

$bnd(f, \lambda(n, t), \overline{y}, l) \longrightarrow \langle \theta, \lambda(n, s)\rangle$,
if $n > 0$ and $bnd(f, t, \overline{y}, l+n) \xrightarrow{*} \langle \theta, s\rangle$.

$bnd(f, a(\overline{t}), \overline{y}, l) \longrightarrow \langle \theta, b(\overline{s}^R)\rangle$,
provided $foldbnd(f, \langle \theta, \varepsilon\rangle, \overline{t}, \overline{y}, l) \xrightarrow{*} \langle \theta, \overline{s}\rangle$ and
either a is a universal variable such that $l(a) \leq l(f)$ and $b = a$
or a appears in $\overline{y}\uparrow l$ and $b = a\downarrow(\overline{y}\uparrow l)$.

$bnd(f, g(\overline{z}), \overline{y}, l) \longrightarrow \langle \{\langle g, \lambda(|\overline{z}|, h(\overline{p} + \overline{u}))\rangle\}, h(\overline{q} + \overline{v})\rangle$,
where h is a new existential variable such that $l(h) = l(f)$,
$\overline{p} = (\overline{y}\uparrow l)\Uparrow g$, $\overline{q} = \overline{p}\downarrow(\overline{y}\uparrow l)$, and $\overline{u} = \overline{w}\downarrow\overline{z}$ and $\overline{v} = \overline{w}\downarrow(\overline{y}\uparrow l)$ for $\overline{w} = (\overline{y}\uparrow l)\cap\overline{z}$,
provided f and g are distinct existential variables such that $l(f) < l(g)$.

$bnd(f, g(\overline{z}), \overline{y}, l) \longrightarrow \langle \{\langle g, \lambda(|\overline{z}|, h(\overline{q} + \overline{v}))\rangle\}, h(\overline{p} + \overline{u})\rangle$,
where h is a new existential variable such that $l(h) = l(g)$,
$\overline{p} = \overline{z}\Uparrow f$, $\overline{q} = \overline{p}\downarrow\overline{z}$, and $\overline{v} = \overline{w}\downarrow\overline{z}$ and $\overline{u} = \overline{w}\downarrow(\overline{y}\uparrow l)$ for $\overline{w} = (\overline{y}\uparrow l)\cap\overline{z}$,
provided f and g are distinct existential variables such that $l(g) \leq l(f)$.

$foldbnd(f, \langle \theta, \overline{s}\rangle, \varepsilon, \overline{y}, l) \longrightarrow \langle \theta, \overline{s}\rangle$.

$foldbnd(f, \langle \theta, \overline{s}\rangle, (t_1, \overline{t}), \overline{y}, l) \longrightarrow foldbnd(f, \langle \varphi \circ \theta, (s_1, \overline{s})\rangle, \varphi(\overline{t}), \overline{y}, l)$,
provided $bnd(f, t_1, \overline{y}, l) \xrightarrow{*} \langle \varphi, s_1\rangle$.

Fig. 2. Calculating variable bindings

as well. Of course, not every such equation will be solved: this may happen, as in Section 3, because t violates the higher-order pattern restriction or because $f(\overline{y})$ and t are not unifiable. Such an effect would be manifest in our system by the inability to rewrite $mksubst(f, t, \overline{y}, |\overline{y}|)$ to a substitution. A failure of this kind would arise, in turn, out of the inability to use any rule to rewrite an intermediate expression either because it does not match with the lefthand side of the rule or because of the violation of a side condition.

We comment briefly on the content of the rules in Figure 2, explaining this with reference to the rules in Figure 1. Borrowing imagery from [2], the substitution for f that solves the equation $f(\overline{y}) = t$ may be viewed as an 'inversion' of t relative to $\overline{y}$. That such an inversion has to be performed is an essential difference from first-order unification and its reflection is the third argument to *mksubst*. Now, the only situation in which the equation with f occurring in t has a solution is when this occurrence is at the head. The first rule for *mksubst* treats this case, leaving all others to be handled by *bnd*. In the treatment of the other cases, actual η-expansion of the flexible term is delayed—this is more efficient and also necessary to treat embedded abstractions. The fourth argument of *bnd* provides the information for this expansion when needed. The first rule for *bnd* reflects this treatment of η-expansion. The second rule treats the case when a rigid structure is encountered in the traversal, building rules (6) and (7) into this process. These rules naturally lead to the examination of the arguments of

(1') $\langle(\lambda(n,t) = \lambda(n,s)) :: E, \theta\rangle \longrightarrow \langle(t = s) :: E, \theta\rangle$, provided $n > 0$.

(2') $\langle(\lambda(n,t) = \lambda(m,s)) :: E, \theta\rangle \longrightarrow \langle((t = \lambda(m-n,s)) :: E), \theta\rangle$, provided $n > 0$ and $m > n$.

(3') $\langle(a(\overline{t}) = \lambda(m,s)) :: E, \theta\rangle \longrightarrow \langle((a{\uparrow}m)(\overline{t}{\uparrow}m) = s) :: E, \theta\rangle$, provided a is a de Bruijn index or a universal variable and $m > 0$.

(4') $\langle(a(\overline{t}) = a(\overline{s})) :: E, \theta\rangle \longrightarrow \langle(\overline{t}[1] = \overline{s}[1]) :: \ldots :: (\overline{t}[n] = \overline{s}[n]) :: E, \theta\rangle$, where $|\overline{t}| = n$, provided a is a de Bruijn index or a universal variable.

(5') $\langle(f(\overline{y}) = t) :: E, \theta\rangle \longrightarrow \langle\varphi(E), \varphi \circ \theta\rangle$ provided f is an existential variable and $mksubst(f, t, \overline{y}, |\overline{y}|) \longrightarrow \varphi$.

Fig. 3. Simplified higher-order pattern unification rules

the rigid term. This process is compiled into the definition of *bnd* and is realized through *foldbnd* that 'maps' *bnd* over the arguments. The third and fourth rules for *bnd* use the idea embodied in rule (9') to treat the situation where a flexible term is encountered. The analysis splits into two cases here because, unlike in rule (9'), we cannot guarantee that $l(f) \leq l(g)$. A final aspect to observe is the effect the delaying of η-expansion has on the last three rules for *bnd*.

Our higher-order pattern unification procedure can be simplified based on the substitution computation rules. The changed set of rules are shown in Figure 3. These rules are to be used in the manner already described and the correctness of the resulting procedure follows from Theorem 2 by formalizing the explanation provided of *mksubst* earlier in this section. We omit the details.

6 Some Aspects Relevant to Actual Implementation

The presentation of the procedure of the last section is still at a high-level and some optimizations are possible in an actual implementation. One such aspect concerns the treatment of the lifting transformation on an argument sequence that is needed in the course of generating a substitution term. Although our presentation suggests that this is done at the relevant points in an eager fashion, a delayed realization that folds lifting into the specific computation that needs it is usually possible. Thus consider the calculation of the sequence

$$\overline{\{m + n - i \mid i \leq n + m \text{ and } (\overline{y}{\uparrow}n)[i] = \overline{z}[i]\}}$$

that appears in the first rule for *mksubst*. Finding this sequence requires an iteration over $\overline{z}$ and $\overline{y}{\uparrow}n$. However, rather than adjusting $\overline{y}$ at the outset, the iteration may be driven by $\overline{z}$ and the lifting operation can be performed on demand. A similar observation also applies to the places where this operation is needed in the definition of *bnd*.

Another observation concerns the computation of the raising and pruning sequences of arguments in the last two rules for *bnd*. The presentation of these rules may suggest that the sequences $\overline{p}$, $\overline{q}$, $\overline{u}$, $\overline{v}$ and $\overline{w}$ are to be calculated separately. In reality, however, the calculation of at least $\overline{p}$ and $\overline{q}$ on the one hand and that of $\overline{u}$, $\overline{v}$ and $\overline{w}$ on the other can be coordinated to yield two

essential iterations. The last rule for *bnd* presents an even better situation where the entire computation can be carried out in one iteration assuming that the elements of $\overline{q} + \overline{v}$ can be shuffled so long as the elements of $\overline{p} + \overline{u}$ are shuffled in the same way: we sweep through the elements of $\overline{z}$ determining which ones to keep in $\overline{q} + \overline{v}$ either because of raising or because they also occur in $\overline{y}{\uparrow}l$ and we simultaneously determine the corresponding elements in $\overline{p} + \overline{u}$.

We have treated substitution non-destructively up to this point. This may be changed in an implementation, thereby obviating the explicit application of substitutions to argument sequences and equation lists. In a different direction, our procedure currently fails when it discovers terms that are not higher-order patterns. This failure may be treated by simply deferring the unification problem. This is probably the best decision at the top-level but a different treatment is possible within the substitution computation process. If it is discovered that t is a flexible term that does not have the structure of a higher-order pattern when attempting to rewrite the expression $bnd(f, t, \overline{y}, l)$, a substitution term of the form $h(\overline{y}{\uparrow}l)$ where h is a new existential variable such that $l(h) = l(f)$ may be returned and the equation $h(\overline{y}{\uparrow}l) = t$ may be deferred.

7 An Example

We illustrate our procedure relative to the unification problem

$$\exists x \forall a \forall b \forall c \exists y \forall d((b(x(a, d)) = b(\lambda(2, (y(1))))) :: nil).$$

Associating the tags 0 with x, 1 with a, b, c and y and 2 with d allows the quantifier prefix to be eliminated, reducing the problem representation to

$$(b(x(a, d)) = b(\lambda(2, (y(1))))) :: nil.$$

Rule (4') of Figure 3 simplifies this to $(x(a, d) = \lambda(2, (y(1)))) :: nil$ and rule (5') then calls for the rewriting of the expression $mksubst(x, \lambda(2, (y(1))), (a, d), 2)$ to a substitution that solves the sole equation in this list.

The rule for *mksubst* in Figure 2 that is relevant to this rewriting task is the second one. This rule requires the expression $bnd(x, \lambda(2, (y(1))), (a, d), 0)$ to be transformed into the form $\langle \theta, s \rangle$ where θ composed with the substitution $\lambda(2, s)$ for x is intended to be a solution to the original equation. The first rule for *bnd* applies to this case, leading to the attempt to rewrite the expression $bnd(x, y(1), (a, d), 2)$; observe that $(a, d) \uparrow 2 = (a, d, 2, 1)$ represents the arguments of x after an η-expansion. The evaluation of this last expression actually represents the heart of the entire calculation. The rule relevant to its rewriting is the third one for *bnd*. Conceptually, one component of this rule determines the arguments of x over which y needs to be raised. This part is given by $((a, d){\uparrow}2){\Uparrow}y$ that evaluates to (a). The projection over $(a, d){\uparrow}2$ of this sequence, notated as $(a){\downarrow}((a, d){\uparrow}2)$, yields (4) that represents the corresponding part of the substitution term being constructed for x. Another component of the rule finds that part of y's arguments that must not be pruned. This is calculated as $((a, d){\uparrow}2){\cap}(1)$, yielding the sequence (1). The projection over the arguments of y and x of this

sequence have an identical result, both being given by (1). Combining the two parts gives us the substitution $\lambda(1, h(a, 1))$ for y and the term $h(4, 1)$ that represents the corresponding part of the substitution for x; h is a new existential variable with tag 0 here. Embedding the substitution term for x in the right context eventually yields the substitution $\{\langle x, \lambda(4, h(4, 1))\rangle, \langle y, \lambda(1, h(a, 1))\rangle\}$ that is a most general unifier for the given problem.

It is instructive to contrast the calculation described above with the one that results under a "blind" raising that first moves all existential variables to the outermost level. Such a raising transforms the problem into the form

$$\exists x \exists y' \forall a \forall b \forall c \forall d((b(x(a, d)) = b(\lambda(2, (y'(a, b, c, 1))))) :: nil)$$

by substituting $y'(a, b, c)$ for y. The subsequent substitution generation process must then take responsibility for pruning away the unnecessary arguments introduced during the initial raising. The advantages of our approach become evident from this especially when we note that, in the typical setting, the initial raising must be performed dynamically, the quantifier prefixes can get long and, finally, most of the dependencies that are introduced during the indiscriminate raising eventually have to be pruned away.

Most previously described procedures for higher-order pattern unification assume that existential quantifiers have an outermost scope. The comparison of our ideas with these is therefore obvious. The contrast with the approach presented in [2] is more subtle. In the statement of the problem, this work also assumes that all existential variables are quantified at the outermost level. However, by exploiting properties of explicit substitutions and a special interpretation for instantiatable variables, this requirement can be eliminated and, diverging from the underlying theory, the eventual procedure presented in [2] seems to actually allow for mixed quantifier prefixes. Moreover, this procedure does not explicitly utilize quantifier prefixes, basing its behaviour mainly on the manipulation of explicit substitutions. However, even given this, the computation that results can manifest a character that is akin to redundant raising complemented by pruning. In the particular example considered here, the behaviour will, in fact, be quite similar to that seen under an initial blind raising. A detailed discussion of this matter requires an exposition also of the explicit substitution approach and is, for this reason, beyond the scope of this paper.

8 Conclusion

We have presented in this paper a procedure that allows for the treatment of higher-order pattern unification in the context of a mixed prefix of quantifiers. This procedure has actually been implemented in C and incorporated into the *Teyjus* system, leading to what appear to be significant performance improvements over full general higher-order unification. We have also realized these ideas in an SML program [8] that has been used in a meta-proof system built by Alwen Tiu and Dale Miller.

This work can be extended in a few different directions. First, there are similarities in the dynamics of our procedure and the one presented in [2] even

though we have not utilized explicit substitutions. We would like to understand these connections better since this is likely to shed light on the question of whether the explicit substitutions based approach of [2] is really useful in the higher-order pattern unification setting. In a different direction, we have found it beneficial to employ explicit substitutions implicitly in reduction procedures [4] and we would like to extend this approach also to the unification context. Finally, higher-order pattern unification offers promising possibilities for compilation. Some work has already been done on this topic [17] and we also have recently started a systematic study oriented around a redesign of the λProlog abstract machine that exploits the procedure of this paper.

Acknowledgements

This work began while the first author was on a sabbatical visit to the Protheo group at LORIA and INRIA, Nancy and the Comete and Parsifal groups at Ecole Polytechnique and INRIA, Saclay. Support has also been derived from the NSF through a Graduate Fellowship and the grant numbered CCR-0429572 and from the Digital Technology Center at the University of Minnesota.

References

1. G. Dowek, T. Hardin, and C. Kirchner. Higher-order unification via explicit substitutions. *Information and Computation*, 157:183–235, 2000.
2. G. Dowek, T. Hardin, C. Kirchner, and F. Pfenning. Unification via explicit substitutions: The case for higher-order patterns. Technical Report 3591, INRIA, December 1998.
3. G. Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27–57, 1975.
4. C. Liang, G. Nadathur, and X. Qi. Choices in representation and reduction strategies for lambda terms in intensional contexts. *Journal of Automated Reasoning*, 33:89–132, 2005.
5. S. Michaylov and F. Pfenning. An empirical study of the runtime behavior of higher-order logic programs. In *Conference Record of the Workshop on the λProlog Programming Language*, Philadelphia, July-August 1992.
6. D. Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1(4):497–536, 1991.
7. D. Miller. Unification under a mixed prefix. *Journal of Symbolic Computation*, 14:321–358, 1992.
8. G. Nadathur and N. Linnell. An SML implementation of higher-order pattern unification, January 2004. Source code available from the first author's web page.
9. G. Nadathur and D. Miller. An overview of λProlog. In K. A. Bowen and R. A. Kowalski, editors, *Fifth International Logic Programming Conference*, pages 810–827. MIT Press, August 1988.
10. G. Nadathur and D.J. Mitchell. System description: Teyjus—a compiler and abstract machine based implementation of λProlog. In H. Ganzinger, editor, *Automated Deduction–CADE-16*, number 1632 in LNAI, pages 287–291. Springer, 1999.

11. T. Nipkow. Higher-order critical pairs. In *Proc. 6th IEEE Symp. Logic in Computer Science*, pages 342–349. IEEE Press, 1991.
12. T. Nipkow. Functional unification of higher-order patterns. In *Eighth Annual IEEE Symposium on Logic in Computer Science*, pages 64–74. IEEE Computer Society Press, June 1993.
13. T. Nipkow, L.C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
14. F. Pfenning. Logic programming in the LF logical framework. In G. Huet and G. D. Plotkin, editors, *Logical Frameworks*. Cambridge University Press, 1991.
15. F. Pfenning. Unification and anti-unification in the Calculus of Constructions. In *Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 74–85, 1991.
16. F. Pfenning and C. Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In H. Ganzinger, editor, *Automated Deduction–CADE-16*, number 1632 in LNAI, pages 202–206. Springer, July 1999.
17. B. Pientka and F. Pfenning. Optimizing higher-order pattern unification. In F. Baader, editor, *Proceedings of the 19th Conference on Automated Deduction (CADE-19)*, LNCS 2741, pages 473–487. Springer-Verlag, July 2003.
18. Z. Qian. Unification of higher-order patterns in linear time and space. *Journal of Logic and Computation*, 6(3):315–341, 1996.

Small Proof Witnesses for LF

Susmit Sarkar[1,⋆], Brigitte Pientka[2], and Karl Crary[1]

[1] Carnegie Mellon University, Pittsburgh, USA
[2] McGill University, Montréal, Canada

Abstract. We instrument a higher-order logic programming search procedure to generate and check small proof witnesses for the Twelf system, an implementation of the logical framework LF. In particular, we extend and generalize ideas from Necula and Rahul [16] in two main ways: 1) We consider the full fragment of LF including dependent types and higher-order terms and 2) We study the use of caching of sub-proofs to further compact proof representations. Our experimental results demonstrate that many of the restrictions in previous work can be overcome and generating and checking small witnesses within Twelf provides valuable addition to its general safety infrastructure.

1 Introduction

Proof-carrying code applications establish trust by verifying compliance of the code with safety and security policies. A code producer verifies that the program is safe to run according to some predetermined safety policy, and supplies a binary executable together with its safety proof. Before executing the program, the code consumer then quickly checks the code's safety proof against the binary.

The Twelf system [22], an implementation of the logical framework LF [12], provides a general safety infrastructure to represent and execute safety policies via a higher-order logic program interpretation and has been employed in several proof-carrying code projects [4,8,3,9]. Higher-order logic programming extends first order logic programming along two orthogonal dimensions: First, dynamic assumptions may be generated and used during proof search. Second, first-order terms are replaced with dependently typed λ-terms, thereby directly supporting encodings via higher-order abstract syntax.

One of the benefits of using Twelf is that the execution of a query will not only produce a yes or no answer, but produce a proof term as a certificate that can be checked independently. This increases the confidence in the overall correctness of the higher-order logic programming engine, and the certificate can be sent to the code consumer where compliance with the code is checked.

Unfortunately, the proof terms produced by Twelf are quite big in size. This creates problems in a proof carrying code setting where proof terms are sent across the network. We would like to produce small proof witnesses and check

⋆ This work was supported by NSF ITR Grant 0121633:ITR/SY+SI: "Language Technology for Trustless Software Dissemination".

M. Gabbrielli and G. Gupta (Eds.): ICLP 2005, LNCS 3668, pp. 387–401, 2005.

them. Our approach to this problem is to instrument the higher-order logic programming interpreter by extending and generalizing ideas by Rahul and Necula [16]. To obtain small proof witnesses, they propose to only record the non-deterministic choices during logic programming execution as a bit-string. We can check such a proof witness by guiding a deterministic logic programming interpreter using the bit-string and re-running the proof. This simple idea has been proven to be effective in many practical examples. We observe a minimum compression of a factor of 70 in proof size in our experiments, increasing up to a factor of almost 700 for larger proofs. This idea has also been used by Wu *et. al* [30] for creating a foundational proof checker with small witnesses.

Previous approaches restricted themselves to a fragment of LF excluding higher-order terms and dependent types thereby trading the expressive power of the logical framework LF against simplicity of implementation to generate and check proof witnesses. As a consequence, these systems do not support higher-order abstract syntax in practice, but each particular system now has to use encoding tricks to encode their variable binding constructs together with substitution operations. For example, Wu *et al.* [30] encode the explicit substitution calculus [1] together with the necessary proofs about substitutions for their foundational certified code implementation. As the technology of certified code evolves, we will move to more powerful and expressive safety policies and type systems and the use of higher-order abstract syntax will become crucial for achieving a simple, compact encoding of these systems.

In this paper, we describe the design of generating and checking of small proof witnesses for the full logical framework LF. This work continues where Necula and Rahul [16] left off saying "more experimental results are needed especially in the higher-order setting". Our work has been implemented and evaluated within the Twelf system [22] making it unnecessary to build separate proof checking engines. To obtain a practical scalable implementation, we use higher-order substitution tree indexing [24]. Furthermore, we improve on the size of proof witnesses by caching common sub-proofs[1] .

This paper is structured as follows. We give background on higher-order logic programming in Twelf in Section 2. In Section 3, we present our approach to generating and checking small proof witnesses. In Section 4.1 we explain higher-order term indexing and in Section 4.2, we discuss caching techniques for factoring out common subproofs. We conclude with a discussion of some experimental results within Twelf and related work.

2 Higher-Order Logic Programming

The theoretical foundation underlying higher-order logic programming within Twelf is the LF type theory, a dependently typed lambda calculus [20]. In this setting types are interpreted as clauses and goals and typing context represents

[1] Eliminating common sub-proofs is an orthogonal problem to eliminating redundant implicit type information, as is proposed in [17].

the store of program clauses available. We will use types and formulas interchangeably. Types and programs are defined as follows:

$$\begin{array}{ll} \text{Types } A & ::= P \mid A_1 \to A_2 \mid \Pi x : A_1.A_2 \qquad \text{Programs } \Gamma ::= \cdot \mid \Gamma, x : A \\ \text{Terms } M & ::= c \cdot S \mid x \cdot S \mid \lambda x.M \qquad\quad \text{Spines } S \quad ::= \mathsf{nil} \mid M; S \end{array}$$

We present terms and types using the spine notation [6]. We use meta-variables x to range over term level variables. There are constants at both the term level, denoted by c, and at type level, denoted by a. P ranges over atomic formulas such as $a \cdot S$, *i.e.* type constants applied to spines. We interpret the function arrow $A_1 \to A_2$ as implication and the Π-quantifier, denoting dependent function type, corresponds to the universal $\forall$-quantifier. Types, which are goals and clauses, are inhabited by corresponding proof terms M, and we assume that all proof terms are in normal form.

Other higher-order logic programming languages of a similar flavor are λ-Prolog [15] or Isabelle [19]. To illustrate the notation and explain the problem of small proof witnesses, we will first give an example of encoding the natural deduction calculus in the logical framework LF using higher-order logic programming following the methodology in Harper *et al.* [12]. For more information on how to encode formal systems in LF, see for example Pfenning [21]. Using this example, we will explain generating and checking of small proof witnesses.

2.1 Representing Logics

As a running example, we will consider a fragment of intuitionistic natural deduction calculus consisting of implications and universal quantifiers. Propositions can be then described as follows:

$$\begin{array}{ll} \text{Propositions } A, B, C & := \ldots \mid A \supset B \mid \forall x.A \\ \text{Context} \qquad \Gamma & := . \mid \Gamma, A \end{array}$$

Inference rules describing natural deduction are presented next.

$$\frac{\Gamma \vdash [a/x]A \quad a \text{ is new}}{\Gamma \vdash \forall x.A}\ \mathsf{allI} \qquad \frac{\Gamma \vdash \forall x.A}{\Gamma \vdash [T/x]A}\ \mathsf{allE} \qquad \frac{}{\Gamma, A \vdash A}\ \mathsf{hyp}$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \supset B}\ \mathsf{impI} \qquad \frac{\Gamma \vdash A \supset B \quad \Gamma \vdash A}{\Gamma \vdash B}\ \mathsf{impE}$$

To represent this system in LF, we first need formation rules to construct terms for propositions. We intend that terms belonging to `prop` represent well-formed propositions and `i` represents individuals.

The connective for implication has a type that takes in two propositions and returns a proposition, hence the constructor `imp` has type `prop -> prop -> prop.` To represent the forall-quantifier, we will use higher-order abstract syntax. The crucial idea is to represent bound variables in the object language (logic)

with bound variables in the meta-language (higher-order logic programming). Hence the type of `forall` is `(i -> prop) -> prop`.

Next we turn our attention to the inference rules. The judgment for provability within this logic is denoted by the type family `prov`. Each clause will correspond to an inference rule in the object logic. For convenience, we give the constructors descriptive names.

```
alli: prov (forall λx. A x)          impi: prov (imp A B)
      <- Πx. prov (A x)                    <- (prov A -> prov B).
alle: prov (A T)                     impe: prov B
      <- prov (forall λx. A x).            <- prov (imp A B)
                                           <- prov A.
```

A, B, C denote existential or logic variables which are instantiated during proof search. Throughout the example we reverse the arrow `A -> B` writing instead `B <- A`. This way, goals appear in the order in which they are processed during proof search. From a logic programming view, it might be more intuitive to think of the clause `H <- A1 <- A2 <- ... <- An` as `H <- A1, A2, ..., An`. There are two key ideas which make the encoding of the logic calculus elegant and direct. First, we use and manipulate dynamic assumptions which higher-order logic programming provides, to eliminate the need to manage assumptions in a list explicitly. To illustrate, we consider the clause `impI`. To prove `prov (imp A B)`, we prove `prov B` assuming `prov A`. In other words, the proof for `prov B` may use the dynamic assumption `prov A`. Second, we use higher-order abstract syntax to encode the bound variables in the universal quantifier. As a consequence substitution in the object language can be reduced to application and β-reduction in the meta-language (higher-order logic programming). Consider the rule for all-elimination. If we have a proof of $\forall x.A$, then we know that $[T/x]A$ is true for any term T. The substitution $[T/x]A$ in the object language is achieved via application in the meta-language `(A T)`.

2.2 Proof Search in Higher-Order Logic Programming

Higher-order logic programming is similar to a Prolog interpreter in that it performs essentially a depth-first search over all the program clauses. The key challenges in moving to a higher order setting are twofold: First, we may have dynamic assumptions which may be used within a certain scope. Second, since we allow higher-order terms (i.e. terms may contain λ-abstraction), higher-order unification is used to unify clause heads with current goal.

In this section, we briefly describe the depth-first proof search procedure of the higher-order logic programming interpreter. Computation in logic programming is achieved through proof search. Given a goal (or query) G and a program Γ, we derive G by successive application of clauses of the program Γ. To solve a goal G from a set of clauses Γ, we decompose the compound goal G until it is atomic and then resolved it with a program clause. We have the following three possible actions (for a more detailed description see Miller *et al.* [14]):

Select. $\Gamma \vdash G \Rightarrow c_i \cdot S$
Given an atomic goal G and clauses Γ:
Focus on a clause $c_i : A_i$ from Γ by unifying the head of A_i with the current goal G. Solve the subgoals of the clause, yielding a proof spine S. The proof term established for G is $c_i \cdot S$.

Augment. $\Gamma \vdash G_1 \rightarrow G_2 \Rightarrow \lambda u.M$ if $\Gamma, u{:}G_1 \vdash G_2 \Rightarrow M$
Augment the clauses in Γ with the dynamic assumption $u{:}G_1$ and establish a proof M for the goal G_2 from the extended program $\Gamma, u{:}G_1$.

Universal. $\Gamma \vdash \Pi x.G \Rightarrow \lambda x.M$ if $\Gamma \vdash [a/x]G \Rightarrow [a/x]M$ where a is a new parameter
Given a universally quantified goal $\Pi x.G$, we generate a new parameter a, and establish a proof $[a/x]M$ for the goal $[a/x]G$ in the program context Γ.

Once the goal is atomic, we need to select a clause from the program context Γ to establish a proof for G. In a logic programming interpreter, we consider all the clauses in Γ in order. First, we will consider the dynamic assumptions, and then we will try the static program clauses one after the other. Let us assume, we picked a clause A from the program context Γ. We now need to establish a proof for G, by unifying the head of the clause A with G and solving the subgoals of A. We will illustrate proof search by considering the following example:

```
prov (forall λy. (imp (forall λ x. p x) (p y)))
```

which corresponds to $(\forall y.(\forall x.p(x)) \supset p(y))$. where `p` is a defined predicate. To prove the query, we will start by unifying the head of the clause (`allI`) with the query, which results in subgoal:

$$\Pi a.\texttt{prov}(\texttt{imp}\ (\texttt{forall}\ \lambda\texttt{y.p}\ \texttt{y})\ (\texttt{pa}))$$

In the Universal step, we introduce a new parameter a yielding the subgoal:

$$\texttt{prov}(\texttt{imp}\ (\texttt{forall}\ \lambda\texttt{y.p}\ \texttt{y})\ (\texttt{pa})).$$

To prove this subgoal we will again inspect our clauses. Three of them will be applicable, namely `allE`, `impI`, and `impE`. This time we will pick the second clause `impI`. Hence we will introduce the dynamic assumption `u:prov(forall λ y.p y)` and show `prov (p a)` using the dynamic assumption `u`. In the third step, again two clauses are applicable, `allE`, and `impE`. Using the first one, `allE`, we need to show that we can prove `prov(forall λy.P y)`. There are four possible clauses whose clause head will unify: the dynamic clause `u` and the three program clauses `alli`, `alle`, and `impe`. Using the dynamic assumption `u`, we can finish the proof. Twelf's higher-order logic programming engine will generate the following proof term in explicit form:

```
(alli (λx. ((forall λy. p y) imp p x))
   λa. (impi (forall λy.p y) (p a)
           λu. (alle (λy.p y) a u))).
```

The final proof term not only tracks the rules which have been used in every step of the proof, but also tracks the instantiations for the logic variables in each steps. In the proof term above we show the instantiations in gray.

As shown in Necula [17], the instantiations of existential variables need not be recorded in the explicit proof terms but can be reconstructed as long as we only concentrate on a fragment of LF, called LF_i. This can lead to substantial savings in proof checking and proof size. Proofs are roughly $\mathrm{O}(\sqrt{n})$, where n is the size of the query. However, extending this idea to full LF has been difficult [29]. Maybe more importantly, proofs in LF_i are still several times as big as the overall program they certify.

Our goal is to produce smaller proof witnesses by reducing the proof evidence to the choices we make while constructing the proof. In the previous example, it suffices to know that in the first step, three possible rules apply, namely `alli`, `alle`, and `impe` and we want to follow the first possibility. In the second step, again three possible rules apply, namely `alle`, `impi`, and `impe`, and we want to follow the second possibility. In the final step, we have four potential candidates, the dynamic assumption `u:forall` (λ`y.p y`), and the rules `allI`, `allE`, and `impE`. Hence it would suffice to store only a list of the choices made in the proof. In this example, the choices can be characterized by the following sequence: 1/3, 2/3, 1/2, 1/4, keeping in mind that dynamic assumptions are tried first by proof search procedures. This sequence will constitute our compact proof witness and is all that needs to be generated and sent to the verifier. In the remainder of the paper, we show how to incorporate this technique into Twelf.

3 Generating and Checking Small Proof Witnesses

3.1 Proof Compression

In this section, we describe the modifications to the proof search procedure needed to generate a compact proof witness in the form of a bit-string. We assume that we already have the full proof term, which in certifying code systems is typically generated by a compiler. The bit-string encodes the non-deterministic choices within the proof, namely picking the right clause $c_i{:}A$ from the program context Γ to establish a proof P for G, once the goal G is atomic, by unifying the head of A with the atomic goal G. Potentially, there is more than one clause whose head unifies with G, and hence a proof search procedure would need to try all the possible choices in order. The proof witness just needs to keep track of which possibility was successful.

In our approach, generating and checking witnesses essentially perform the same overall proof search. The only difference is that in proof search we would likely explore multiple fruitless paths and backtrack until we find the right path. When generating and checking witnesses, we will consult the proof term or witness respectively to know which choice to consider, and thus eliminate backtracking. We modify the proof search steps presented earlier as follows:

Select. $\Gamma \vdash c_i \cdot S : G \Rightarrow \underbrace{0 \ldots 0}_{1 \ldots (i-1)} 1W$

Given an atomic goal G and clauses Γ:
Let $c_i : A_i$ be the $i-$th clause from Γ whose head unifies with goal G.
Focus on clause $c_i : A_i$ from Γ. Use the proof spine S to guide the solving of subgoals, yielding witness W.

Augment. $\Gamma \vdash \lambda u.M : G_1 \rightarrow G_2 \Rightarrow W$ if $\Gamma, u{:}G_1 \vdash M : G_2 \Rightarrow W$
Augment the clauses in Γ with the dynamic assumption $u{:}G_1$ and compress a proof M for G_2 within the extended program $\Gamma, u{:}G_1$ to obtain the witness W.

Universal. $\Gamma \vdash \lambda x.M : \Pi x.G \Rightarrow W$ if $\Gamma \vdash [a/x]M : [a/x]G \Rightarrow W$
Given a universally quantified goal $\Pi x.G$, we generate a new parameter a, and compress a proof $[a/x]M$ for $[a/x]G$ in the program context Γ to W.

Note that the Select step is deterministic as the proof term determines which choice will be successful. It should be intuitively clear that we do not necessarily have to pass in the full proof term, but could directly produce a proof witness in form of a bit-string if our proof search is powerful enough that it will eventually find a proof.

3.2 Checking Small Proof Witnesses

In this section, we modify the previous search procedure, in such a way that it is not parameterized by the proof term M, but rather by the compact proof witness W encoded as a bit-string. We are given goal G in a program context Γ, together with a proof witness W. The procedure is the dual of the compression case, and we show the important **Select** case.

Select $\Gamma \vdash \underbrace{0 \ldots 0}_{1 \ldots (i-1)} 1W : G$
Given an atomic goal G and clauses Γ:
Let k be the number of clauses whose head unifies with the current goal G, then inspect up to k bits, and find the i-th bit which is one.
Focus on clauses $c_i : A_i$ from Γ to establish a proof for the atomic goal G from Γ using remaining proof witness W.

In Select step, we first generate the k possible candidates whose head will unify with the current goal G. If k is greater than 1, we will examine up to k bits from the witness to see which choice to take. If a bit 1 occurs at position i of these k bits, we will pick the i-th candidate. For this idea to work, it is crucial that the order of choices during witness checking is same as during witness generation.

In order to check the proof witnesses, we re-run the prover guided with the advice encoded in the bit-string. The witness checker is then a deterministic search procedure. No backtracking is necessary, since all the non-deterministic choices are resolved. Note that the proof term does not need to be reconstructed.

3.3 Bit-String Encodings for Proof Witnesses

The choices as described above are choice sequences of the form $i_1/k_1, i_2/k_2, \ldots$, where at the j^{th} stage we have k_j choices, and we want to pick i_j $(1 \leq i_j \leq k_j)$. With the tight coupling of the witness generation and checking phases, both phases agree on the number of choices (k_j) as well as the ordering of those choices,i.e. both producer and checker agree on which choice is to be considered the i_j-th one.

We can now see that the separator between choices is unnecessary. We can decide on a encoding scheme, and pull only the required number of bits from the oracle. The witness checker will always know how many bits to extract.

We have experimented with two simple encoding schemes, though more complex coding schemes can be imagined. The original proposal by Necula and Rahul proposed what we call the binary scheme, in that the number would be encoded in binary. If k choices apply, this will require $\lceil \log k \rceil$ bits. We discover that a scheme we call unary encoding works better in practice. In this scheme, the choice number i is encoded as 000 . . . (i-1 zeros)1. This takes i bits.

The binary scheme will work better when we habitually have a large number of choices, and we take one of the later choices in the ordering considered by the producer/checker. The unary scheme will work better precisely in the other cases. In all our examples, we have observed that only a few choices typically apply. Further, logic programmers usually write their programs so that the more common choices are tried first. With these observations, unary encodings should outperform binary encodings, as indeed they do in experimental studies. This is a configurable option in our engine, and can be set depending on the particular proof or logic.

4 Optimizations

4.1 Higher-Order Term Indexing

In the **Select** step of our algorithms, we need to retrieve clauses which may unify with the goal. To avoid redundant computations most first-order logic programming interpreter use efficient term indexing strategies such as automata driven indexing [28]. Indexing strategies for higher-order terms are more difficult, since in general retrieval and insertion operations rely on computing the most general unifier or the most specific generalization. However, in the higher-order case, unification is in general undecidable and the most general unifier does not necessarily exist. The same holds for computing the most specific generalization of two terms.

We will adopt higher-order substitution trees [24,26] as our indexing mechanism. Substitution tree indexing has been successfully used in a first-order setting [11] and allows the sharing of common sub-expressions via substitutions. This is unlike other non-adaptive term indexing methods which only allow sharing of common term prefixes. To extend substitution tree indexing to the higher-order setting, we use linear higher-order patterns [27]. Higher-order patterns [13] are terms where all existential variables must be applied to distinct bound variables. Linear higher-order patterns further restrict existential variables to occur only once and to be applied to all distinct bound variables.

The construction of a substitution tree in the higher-order setting follows the overall algorithm described in Ramakrishnan *et al* [28]. We will illustrate higher-order substitution trees by an example. Assume we have the following clauses which allow quantifier manipulation for first-order logic:

$$\texttt{e1} : \texttt{eq}\ (\texttt{imp}\ (\texttt{exists}\ \lambda\texttt{x.A x})\ \texttt{B})\quad (\texttt{forall}\ \lambda\texttt{x.}(\texttt{imp}\ (\texttt{A x})\ \texttt{B})).$$
$$\texttt{e2} : \texttt{eq}\ (\texttt{imp}\ \texttt{A}\ (\texttt{forall}\ \lambda\texttt{x.B x}))\quad (\texttt{forall}\ \lambda\texttt{x.}(\texttt{imp A}\ (\texttt{B x}))).$$
$$\texttt{e3} : \texttt{eq}\ (\texttt{and A}\ (\texttt{forall}\ \lambda\texttt{x.B x}))\quad (\texttt{forall}\ \lambda\texttt{x.}(\texttt{and A}\ (\texttt{B x}))).$$

Although all the terms in these clauses fall into the pattern fragment, not all of them are linear patterns. We linearize them by eliminating any duplicate occurrences of existential variables, and replacing any existential variable which is not fully applied with one which is. The linearized program is given next:

$$\texttt{e1} : \texttt{eq}\ (\texttt{imp}\ (\texttt{exists}\ \lambda\texttt{x.A x})\ \texttt{B})\ (\texttt{forall}\ \lambda\texttt{x.}(\texttt{imp}\ (\texttt{A}'\ \texttt{x})\ (\texttt{B}'\ \texttt{x}))).$$
$$\forall x.(A'\ x)\ \doteq\ (A\ x)\ \texttt{and}\ B'\ x\ \doteq\ B$$

$$\texttt{e2} : \texttt{eq}\ (\texttt{imp A}\ (\texttt{forall}\ \lambda\texttt{x.B x}))\quad (\texttt{forall}\ \lambda\texttt{x.}(\texttt{imp}\ (\texttt{A}'\ \texttt{x})\ (\texttt{B}'\ \texttt{x}))).$$
$$\forall x.(A'\ x)\ \doteq\ A\ \texttt{and}\ B'\ x\ \doteq\ (B\ x)$$

$$\texttt{e3} : \texttt{eq}\ (\texttt{and A}\ (\texttt{forall}\ \lambda\texttt{x.B x}))\quad (\texttt{forall}\ \lambda\texttt{x.and}\ (\texttt{A}'\ \texttt{x})\ (\texttt{B}'\texttt{x})).$$
$$\forall x.(A'\ x)\ \doteq\ A\ \texttt{and}\ B'\ x\ \doteq\ (B\ x)$$

We now compute the most specific generalization between these clauses, and can build up a substitution tree. The algorithm for computing the most specific generalization is given in [26,24].

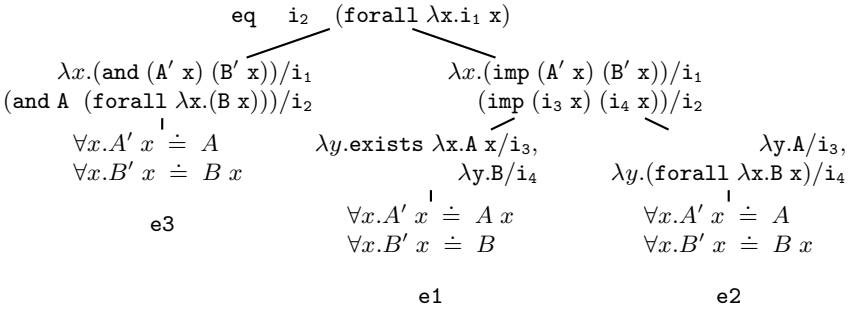


Fig. 1. Substitution tree

By composing the substitutions along a path, we will obtain a clause head. By composing the substitutions in the right-most branch, we obtain the clause head e2. In contrast to other indexing techniques such as discrimination tries, substitution trees allows the sharing of common sub-expressions instead of common term prefixes. This is often very useful, as we can see in this example, since the most sharing is done in the second argument.

We have chosen to index only the static set of program clauses. In theory, substitution tree indexing can be used for dynamic clauses generated during proof search also. However, it is not clear how useful this will be, since creating the tree itself is time-consuming. It is also noted by Necula and Rahul [16] that indexing dynamic assumptions imposes a performance penalty.

4.2 Caching Results

Since large proofs often have identical subproofs, there is often potential for sharing subproofs, particularly in machine-generated proofs which tend to have

repeated proofs of simple facts. This was also pointed out by Necula and Lee [18]: "... it is very common for the proofs to have repeated sub-proofs that should be hoisted out and proved only once ...".

When generating and checking small proof witnesses, this leads to two problems. First, the proof witnesses become larger, thus increasing transmission costs. Second, the performance of the witness checker may degrade, since it spends its time uselessly verifying the same fact over and over again. Ideally we would like to cache intermediate results and re-use them later.

We use ideas from and also infrastructure developed for tabled higher order logic programming [26,23]. Since caching everything may be too costly in practice, we support selective caching. The user can declare certain predicates to be cached. We modify the Select step in our previous search procedure to allow for caching. Assume our subgoal is $\Gamma \vdash G$. We check whether the current subgoal is an instance of a previous table entry. If there exists a table entry $\Gamma' \vdash G'$ s.t. $\Gamma \vdash G$ is a variant (or instance) of the already existing entry, then a pointer to the corresponding answer list is returned. If no such entry exists, $\Gamma \vdash G$ is added to the table and a pointer to an empty answer list is returned. In this case, we will continue to focus on a clause c_i as usual to solve the goal $\Gamma \vdash G$. When we are done, we will add the answer substitution for the existential variables in $\Gamma \vdash G$ together with its proof term $c_i \cdot S$ to its answer list.

If a table entry for the goal already exists, there are two possible situations:

1. If the answer list contains an answer substitution θ_k that leads to the proof $c_i \cdot S$ we are compressing, then we will just re-use the answer substitution θ_k.
2. If the answer list does not contain an answer that would lead to the proof $c_i \cdot S$, then we need to use a program clause c_j to focus on and solve the goal $\Gamma \vdash G$.

The generation and checking of witnesses will follow similar algorithms, so both have identical caches and consider the same number of choices.

5 Experimental Results

In this section, we give an experimental evaluation of generating and checking compact proof witnesses. In particular, the results discuss the trade off between witness size and the time it takes to construct or check witnesses. Thus, we will consider three cases: asking the checker to perform proof search, proof checking of explicit proofs, and our approach of small proof witnesses. The first two represent two extreme cases, the first one with zero witness size but large proof search time, and the second one with large witness size but fast checking times. We will also discuss the trade-offs of caching subproofs. Finally, we will compare different encoding schemes for describing the non-deterministic choices and see how this affects the size of the proof witness.

Our experiments are run on a Pentium 4 machine running at 2GHz with 1 GB of memory size. The machine runs Twelf compiled by SML of New Jersey version 110.0.7, and runs it on the Redhat Linux 7.1 operating system, with no

programs running on the background. We present a representative selection of results from an extensive suite of experiments we have run.

5.1 Time and Size Trade-Offs

Our first example suite is an implementation of a sequent calculus for intuitionistic propositional logic where invertible rules are chained together thereby eliminating some non-determinism in the overall proof search.

Sequent Calculus: Times with Caching of User-Selected Predicates

Example	PST	PCT	WV	(PST/WV)	PS	PSN	WS	(PS/WS)
$(A \supset B) \wedge (A \supset C) \Rightarrow A \supset B \wedge C$	0.47	< 0.01	< 0.01	∞	361	43	5	72.2
$A \vee C \wedge (B \supset C) \Rightarrow (A \supset B) \supset C$	1.70	< 0.01	0.01	170	570	50	6	95.0
$(A \supset C) \wedge (B \supset C) \Rightarrow A \vee B \supset C$	2.18	< 0.01	< 0.01	∞	561	56	6	93.5
$\Rightarrow (A \supset C) \wedge (B \supset C) \supset A \vee B \supset C$	2.43	< 0.01	0.01	243	792	57	6	132.0

Key: PST =Proof Search time (s) PS =Proof Size in bytes
PCT=Proof Checking time (s) PSN=Proof Size in Number of tokens
WV =Witness Verification time (s) WS =Witness Size in bytes

We study the time to find a proof and contrast it against the proof checking times. We use the tabled higher-order logic programming engine [25,26] to find proofs for the propositional logic. The proof compression and the verification procedures provide significant time speedups, since in these procedures, we already know the proof. Next, we turn our attention to questions of proof size. The original proof is measured both by number of bytes as well as the number of tokens, and the compact proof witness is produced using the unary encoding described earlier.

Our second example is an advanced type system for a high-level call-by-value functional language using refinement types [10]. The language has functions, a fix-point construct, booleans and bit-strings. In particular, the type of bit-strings is refined by zero and strictly positive number representations.

Refinement Type System : Proof Compression Times with Caching

Example	PST	PCT	WV	(PST / WV)	PS	PSN	WS	(PS / WS)
mult-pos-nat	5.81	0.05	1.10	5.3	15,654	1,159	169	92.6
mult	0.39	0.02	0.13	3.0	6,074	509	47	129.2
square-pos-nat	12.55	0.06	1.85	6.8	25,303	1,587	242	104.6

The experimental results demonstrates that proof checking yields a speedup between three and six times. This figure is achieved if we are caching subgoals to get maximum compressions. As we see later, even more time gains can be achieved by turning off caching, since we do not explore unproductive branches in the proof tree. Finally, we notice that the compact oracle is about 1% of the size of the proof term.

Our last example suite is an implementation from the Foundational Proof Carrying Code project at Princeton [2]. This is a large program that type checks

SPARC object code with the help of annotations produced by a compiler. The type system used is a low-level type system known as LTAL [7].

FPCC: Times without Caching

Example	PST	PCT	WV	(PST / WV)	PS	PSN	WS	(PS / WS)
clos	12.26	2.505	0.47	26.1	201,910	16,502	638	316.5
mid	10.29	2.246	0.45	22.9	398,589	34,250	528	754.9
inc	11.55	2.310	0.47	24.6	410,600	35,724	579	709.2
lint	12.84	2.591	0.70	18.3	441,965	38,416	703	628.7

In the proof carrying code scenario, asking the consumer to verify our compact proof witness as opposed to doing proof search gives a speedup of about 20 times. Also important is the size of the proof that must be sent to the consumer. Our proof witnesses are between 300 and 700 times smaller than the corresponding proof terms. Finally, we notice that as proof sizes become bigger, our mechanisms perform better at compressing proofs. The space savings go from a factor of about 70 in the smallest examples all the way to about 700 times for our largest examples. The gains in time are from a factor of about 5 to a factor of about 25 for the larger examples.

5.2 Caching: Time vs Space

Next we investigate the practicality of caching subgoals. Caching is a fairly expensive operation, in terms of both time for stores and lookups and the memory required to maintain the table and we investigate this trade-off next. As our result show, caching results in a speed penalty of between three and fifteen times. The gain from this is that the size of the oracle is smaller for the cached version in every experiment. Disappointingly, the gain is usually small.

Refinement Type System: Caching during proof compression

Example	Compression Time			Witness Size			Table
	Cached (sec) (A)	Uncached (sec) (B)	Slowdown (A / B)	Cached (bytes) (C)	Uncached (bytes) (D)	Saving (D - C)/D	Size
mult-pos-nat	1.18	0.11	10.7	169	171	1.2 %	579
mult	0.14	0.05	2.8	47	67	29.9 %	164
square-pos-nat	2.31	0.16	14.4	242	247	2.0 %	794

5.3 Encoding Schemes

Finally, we study the issue of unary versus binary encodings of the choices. A representative study with examples from multiple example suites is given next. We notice that binary encodings always increase the size of the oracle, by between 7% and 115%. As we discussed before, logic programmers usually write their programs so that the first few clauses are the ones that are used more commonly, in which case unary encodings are better.

Unary versus Binary Encodings: no Caching

Example	WSU	WSB	(WSB - WSU/ WSU)
clos	638	715	12.0 %
mid	528	652	23.5 %
lint	703	754	7.3 %
mult-pos-nat	171	338	97.7 %
mult	67	144	114.9 %

Key WSU = Witness Size (Unary Encoded)
WSB = Witness Size (Binary Encoded)

6 Related Work

The idea of compact proof witnesses that encode the non-deterministic choice in a logic programming interpreter was first proposed by Necula and Rahul [16] for a fragment of the logical framework LF, that excludes the use of higher-order terms and significantly limits the use of dependent types in practice. Their main goal was to design a practical method for current proof-carrying code applications to reduce the size of proofs sent to a consumer. To achieve an efficient implementation, they propose the use of automata-driven indexing, where any higher-order features are ignored. Their indexing algorithm will generate a set of potential candidates from which unsound candidates need to be weeded out by calling higher-order unification based on Huet's algorithm. This is clearly wasteful and expensive in the general higher-order case, since we will traverse higher-order terms at least twice. Moreover, since Huet's unification algorithm is non-deterministic itself, their proof witnesses also record the choices made during unification. To avoid these problems in practice, their realization and their experimental evaluation does not consider terms defined via λ-abstraction.

The idea of using oracles was also explored in Wu *et al* [30]. The main difference between their and the previous approach is that the proof rules are proven correct independently thereby minimizing the trusted computing base. Trust is not our concern here, rather we aim at extending the safety infrastructure already provided by Twelf with capabilities of generating and checking small proof witnesses. This step, we believe, will provide the developers of safety policies in Twelf with new insights about the relationship of safety rules and size of proofs.

As in Necula and Rahul's work, Wu *et al.*'s system does not support higher-order abstract syntax, which drastically limits its usefulness. Wu *et al.*[30] encode the explicit substitution calculus [1] together with the necessary proofs about substitutions for their foundational implementation of LTAL. Although the overhead in this setting is still manageable, it is not general enough to handle richer safety polices.

7 Conclusion

In this paper, we extended the logical framework LF with small proof witnesses. Witness generation and checking within the logical framework LF constitutes a valuable addition to the general safety infrastructure already provided. This can provide insights into the relationship between safety policies and small safety

proofs and allows for experiments with different kinds of encoding schemes. Given the potential of proof-carrying code methods and their new applications to proof-carrying authorization [3,5], this will provide a comprehensive guide for future implementations of proof checkers which need not be restricted to first-order Prolog-like systems.

References

1. Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lèvy. Explicit substitutions. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages, San Francisco, California*, pages 31–46. ACM, 1990.
2. Andrew Appel. Foundational proof-carrying code. In J. Halpern, editor, *Proceedings of the 16th Annual Symposium on Logic in Computer Science (LICS'01)*, pages 247–256. IEEE Computer Society Press, June 2001. Invited Talk.
3. Andrew W. Appel and Edward W. Felten. Proof-carrying authentication. In *ACM Conference on Computer and Communications Security*, pages 52–62, 1999.
4. W. Appel and Amy P. Felty. A semantic model of types and machine instructions for proof-carrying code. In *27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '00)*, pages 243–253, Jan. 2000.
5. Lujo Bauer. *Access Control for the Web via Proof-carrying Authorization*. PhD thesis, Princeton University, November 2003.
6. Iliano Cervesato and Frank Pfenning. A linear spine calculus. *Journal of Logic and Computation*, 13(5):639–688, 2003.
7. Juan Chen, Dinghao Wu, Andrew W. Appel, and Hai Fang. A provably sound TAL for back-end optimization. In *PLDI '03 : Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 208–219. ACM Press, June 2003.
8. Karl Crary. Toward a foundational typed assembly language. In *30th ACM Symposiumn on Principles of Programming Languages (POPL)*, pages 198–212, New Orleans, Louisisana, January 2003. ACM-Press.
9. Karl Crary and Susmit Sarkar. Foundational certified code in a metalogical framework. In *19th International Conference on Automated Deduction*, Miami, Florida, USA, 2003. Extended version published as CMU technical report CMU-CS-03-108.
10. Rowan Davies and Frank Pfenning. Intersection types and computational effects. In P. Wadler, editor, *Proceedings of the Fifth International Conference on Functional Programming*, pages 198–208. ACM Press, 2000.
11. Peter Graf. *Term Indexing*, volume 1053 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 1995.
12. Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993.
13. Dale Miller. Unification of simply typed lambda-terms as logic programming. In *Eighth International Logic Programming Conference*, pages 255–269, Paris, France, June 1991. MIT Press.
14. Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

15. Gopalan Nadathur and Dustin J. Mitchell. System description: Teyjus – a compiler and abstract machine based implementation of Lambda Prolog. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 287–291, Trento, Italy, July 1999. Springer-Verlag LNCS.
16. G. Necula and S. Rahul. Oracle-based checking of untrusted software. In *28th ACM Symposium on Principles of Programming Languages (POPL'01)*, pages 142–154, 2001.
17. George C. Necula and Peter Lee. Efficient representation and validation of logical proofs. In Vaughan Pratt, editor, *Proceedings of the 13th Annual Symposium on Logic in Computer Science (LICS'98)*, pages 93–104, Indianapolis, Indiana, June 1998. IEEE Computer Society Press.
18. George C. Necula and Peter Lee. Safe, untrusted agents using proof-carrying code. In Giovanni Vigna, editor, *Mobile Agents and Security*, pages 61–91. Springer-Verlag LNCS 1419, August 1998.
19. Lawrence C. Paulson. Natural deduction as higher-order resolution. *Journal of Logic Programming*, 3:237–258, 1986.
20. Frank Pfenning. Logic programming in the LF logical framework. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 149–181. Cambridge University Press, 1991.
21. Frank Pfenning. Computation and deduction, 1997.
22. Frank Pfenning and Carsten Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 202–206, Trento, Italy, July 1999. Springer-Verlag Lecture Notes in Artificial Intelligence (LNAI) 1632.
23. Brigitte Pientka. A proof-theoretic foundation for tabled higher-order logic programming. In P. Stuckey, editor, *18th International Conference on Logic Programming, Copenhagen, Denmark*, Lecture Notes in Computer Science (LNCS), 2401, pages 271 –286. Springer-Verlag, 2002.
24. Brigitte Pientka. Higher-order substitution tree indexing. In C. Palamidessi, editor, *19th International Conference on Logic Programming, Mumbai, India*, Lecture Notes in Computer Science (LNCS 2916), pages 377–391. Springer-Verlag, 2003.
25. Brigitte Pientka. Tabling for higher-order logic programming. In R. Nieuwenhuis, editor, *20th International Conference on Automated Deduction, Talinn, Estonia*, Lecture Notes in Computer Science (LNCS) (to appear). Springer-Verlag, 2005.
26. Brigitte Pientka. *Tabled higher-order logic programming.* PhD thesis, Dept. of Computer Sciences, Carnegie Mellon University, Dec 2003. CMU-CS-03-185.
27. Brigitte Pientka and Frank Pfennning. Optimizing higher-order pattern unification. In F. Baader, editor, *19th International Conference on Automated Deduction, Miami, USA*, Lecture Notes in Artificial Intelligence (LNAI) 2741, pages 473–487. Springer-Verlag, July 2003.
28. I. V. Ramakrishnan, R. Sekar, and A. Voronkov. Term indexing. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, volume 2, pages 1853–1962. Elsevier Science Publishers B.V., 2001.
29. Jason Reed. Redundancy Elimination for LF. In Carsten Schuermann, editor, *Proceedings of the Fourth Workshop on Logical Frameworks and Meta-languages — LFM'04*, Cork, Ireland, 5 July 2004.
30. Dinghao Wu, Andrew W. Appel, and Aaron Stump. Foundational proof checkers with small witnesses. In *PPDP '03: Proceedings of the 5th ACM SIGPLAN international conference on Principles and practice of declaritive programming*, pages 264–274. ACM Press, 2003.

A Type System for CHR

Emmanuel Coquery[1,2] and François Fages[1]

[1] INRIA Rocquencourt, France
[2] Conservatoire National des Arts et Métiers, Paris, France

The language of *Constraint Handling Rules* (CHR) of T. Frühwirth [1] is a successful rule-based language for implementing constraint solvers in a wide variety of domains. It is an extension of a host language, such as Prolog [2], Java or Haskell [3], allowing the introduction of new constraints in a declarative way. One peculiarity of CHR is that it allows multiple heads in rules. For the sake of simplicity, we consider only simplification rules, since the distinction of propagation and simpagation rules [1] is not needed for typing purposes. A simplification rule is of the form $H_1, \ldots, H_i$ `<=>` $G_1, \ldots, G_j \mid B_1, \ldots, B_k$, where $H_1, \ldots, H_i$ is a nonempty sequence of CHR constraints, the guard $G_1, \ldots, G_j$ is a sequence of native constraints and the body $B_1, \ldots, B_k$ is a sequence of CHR and native constraints.

In this paper, we describe a type system for CHRs. Chin, Sulzmann and Wang [3] proposed a monomorphic type system for their type-safe embedding of CHR into Haskell. Here, we propose a generic type system for CHR inspired by the TCLP type system for constraint logic programs [4]. As CHR is an extension of a host language, the type system is parameterized by the type system of the host language. We assume that the host type system, noted $\vdash_N$, is based on some algebra of types $\mathcal{T}$. *Typing environments*, noted Γ, associate types to program variables. Given an expression t and a typing environment Γ, $\vdash_N$ is used to deduce typing judgments of the form $\Gamma \vdash_N t : \tau$, which means that the expression t has type τ in Γ, or of the form $\Gamma \vdash_N C\ Atom$, which means that C is a *well-typed* constraint in Γ.

The type system for CHR defines a notion of well-typedness for CHR rules. To each CHR constraint symbol c/n is associated a set of types $types(c/n)$, each type being of the form $\tau_1 \times \ldots \times \tau_n$. This set of types is assumed to be fixed, for example using some declarations provided by the programmer. This framework allows the use of parametric polymorphism [5]. A parametric type scheme $\forall \alpha_1 \ldots \alpha_k.\ \tau_1 \times \ldots \times \tau_n$ is represented by the set of all its possible instanciations. For example, the declaration $types(\texttt{append}/3) = \{list(\tau) \times list(\tau) \times list(\tau) \mid \tau \in \mathcal{T}\}$ allows one to give the type $\forall \alpha.\ list(\alpha) \times list(\alpha) \times list(\alpha)$ to the constraint `append`/3.

In the typing rules given below, σ represents a type for a CHR constraint and S represents a set of such types. The rule (*CHR Rule*) requires that, for each combination $\sigma_1, \ldots, \sigma_n$ of the types of the different occurences of the CHR constraints in the head of a CHR rule, the head, the guard and the body of the CHR rule are well-typed in some typing environment Γ. This can be seen as a generalization of the *definitional genericity* [6] condition in logic programming.

M. Gabbrielli and G. Gupta (Eds.): ICLP 2005, LNCS 3668, pp. 402–403, 2005.

$$(Native) \quad \frac{\Gamma \vdash_N C\ Atom}{\Gamma \vdash C\ Atom} \qquad \text{if } C \text{ is a native constraint}$$

$$(Atom) \quad \frac{\Gamma \vdash_N t_1 : \tau_1 \quad \ldots \quad \Gamma \vdash_N t_n : \tau_n}{\Gamma \vdash c(t_1, \ldots, t_n)\ Atom} \qquad \begin{array}{l}\text{where } c/n \text{ a CHR constraint} \\ \text{and } \tau_1 \times \ldots \times \tau_n \in types(c/n)\end{array}$$

$$(Goal) \quad \frac{\Gamma \vdash B_1\ Atom \quad \ldots \quad \Gamma \vdash B_n\ Atom}{\Gamma \vdash B_1, \ldots, B_n\ Goal}$$

$$(Head) \quad \frac{\Gamma \vdash_N t_1 : \tau_1 \quad \ldots \quad \Gamma \vdash_N t_n : \tau_n}{\Gamma \vdash c(t_1, \ldots, t_n)\ Head_{\tau_1 \times \ldots \times \tau_n}} \qquad \begin{array}{l}\text{where } c/n \text{ a CHR constraint} \\ \text{and } \tau_1 \times \ldots \times \tau_n \in types(c/n)\end{array}$$

$$(MultiHead) \quad \frac{\Gamma \vdash H_1\ Head_{\sigma_1} \qquad \ldots \qquad \Gamma \vdash H_i\ Head_{\sigma_i}}{\Gamma \vdash H_1, \ldots, H_i\ MHead_{\sigma_1, \ldots, \sigma_i}}$$

$$(CHR\ Rule) \quad \frac{\begin{array}{ll} & \Gamma \vdash H_1, \ldots, H_n\ MHead_{\sigma_1, \ldots, \sigma_n} \\ \forall \sigma_1 \in S_1, \ldots, \forall \sigma_n \in S_n \exists \Gamma & \Gamma \vdash G_1, \ldots, G_r\ Goal \\ & \Gamma \vdash B_1, \ldots, B_q\ Goal \end{array}}{\vdash H_1, \ldots, H_n\ \texttt{<=>}\ G_1, \ldots, G_r \mid B_1, \ldots, B_q\ Rule}$$

where $H_i = c_i(t_1^i, \ldots, t_{m_i}^i)$ and $S_i = types(c_i/m_i)$ for all $1 \leq i \leq n$

This type system has been implemented with the TCLP type system [4] for constraint logic languages. In this implementation, the type system is extended to allow the combined use of predicates both in CHR and CLP, following the CLP+CHR framework [1]. We refer to [7] for more details. This implemention is part of the TCLP software[1] and experiments, including the typing of TCLP itself, show that the system is usable with large programs and useful in practice.

References

1. Frühwirth, T.: Theory and practice of constraint handling rules. Journal of Logic Programming, Special Issue on Constraint Logic Programming **37** (1998) 95–138
2. Holzbaur, C., Frühwirth, T.: A Prolog Constraint Handling Rules compiler and runtime system. Special Issue Journal of Applied Artificial Intelligence on Constraint Handling Rules **14** (2000)
3. Chin, W.N., Sulzmann, M., Wang, M.: A type-safe embedding of constraint handling rules into Haskell. Technical report, National University of Singapore (2003) `http://www.comp.nus.edu.sg/~sulzmann/chr/hchr/hchr-tr.ps`.
4. Fages, F., Coquery, E.: Typing constraint logic programs. Journal of Theory and Practice of Logic Programming **1** (2001) 751–777
5. Mycroft, A., O'Keefe, R.: A polymorphic type system for Prolog. Artificial Intelligence **23** (1984) 295–307
6. Lakshman, T., Reddy, U.: Typed Prolog: A semantic reconstruction of the Mycroft-O'Keefe type system. In Saraswat, V., Ueda, K., eds.: Proceedings of the 1991 International Symposium on Logic Programming, MIT Press (1991) 202–217
7. Coquery, E., Fages, F.: A type system for CHR. Technical Report RR-5565, INRIA (2005)

[1] `http://contraintes.inria.fr/~coquery/tclp`

Decision Support for Personalization on Mobile Devices

Thomas Kleemann and Alex Sinner

University of Koblenz-Landau, Department of Computer Science, Germany
{tomkl, sinner}@uni-koblenz.de

During the past few years mobile phones have become an ubiquitous companion. Starting as a handset to cellular networks today's mobile phones are devices capable of intelligent services itself. The IASON[1] project aims at providing mobile users with location-aware personalized information. Motivated by the development of powerful mobile devices and the semantic web, we define a *Semantic Mobile Environment*. In such an environment, so-called service nodes are installed at chosen points of interest. These service nodes broadcast semantically annotated messages to nearby mobile users using bluetooth ad-hoc wireless technology. Location-awareness is given implicitly by being in the wireless range of a service node and comes free of costs just as the communication itself. A *Semantic User Profile* describing the users interests and disinterests is managed on her mobile device. This user profile is used to sort out unwanted messages by performing matchmaking between the semantic annotation of the messages and the user profile (see [8]). The protection of privacy requires the profile to stay on the phone and to perform all necessary reasoning on the phone.

Semantic Personalization

Personalization is essential for all kinds of mobile services. The most obvious reason is that spam messages are generally not wanted. Another less obvious reason is that if mobile users want information, they only want information that is of interest to them. In our approach we use the description logic $\mathcal{SHI}$ [2] as a semantic language. $\mathcal{SHI}$ is the standard DL $\mathcal{ALC}$ extended with role hierarchies and inverse, transitive roles, it covers most of OWL-lite [1] enabling the use of tools like protege to edit ontologies.

Every message is accompanied by a concept-expression that defines the semantics of the message. In order to distinguish different roles like producer/consumer in a commercial environment or generic preferences in a peer2peer setting all annotations (1) feature an existentially quantified role R of the service ontology (2,3) and a concept C based on the vocabulary of an ontology available to all participants.

The profile (4) is a collection of positively or negatively marked annotation (5,6). It is initialized through a short questionaire and is updated upon the

[1] http://www.uni-koblenz.de/~iason, funded by "Stiftung Rheinland-Pfalz für Innovation"

M. Gabbrielli and G. Gupta (Eds.): ICLP 2005, LNCS 3668, pp. 404–406, 2005.

user's responses to previously received messages. All subsequent messages are evaluated wrt. the profile. This matchmaking is a refinement of [6] providing graded matches (mismatch, compatible, match) with only two reasoning steps (7,8) even for complex profiles defining many interests and disinterests including distinguished business-roles.

$$annotation_j \equiv \exists R.C \quad with\ R \sqsubseteq share \tag{1}$$

$$request \sqsubseteq share \tag{2}$$

$$offer \sqsubseteq share \tag{3}$$

$$profile \equiv interests \sqcap \overline{disinterests} \tag{4}$$

$$interests \equiv \bigsqcup_{i=1}^{n} positive - marked - annotation_i \tag{5}$$

$$disinterests \equiv \bigsqcup_{i=1}^{m} negative - marked - annotation_i \tag{6}$$

$$profile \sqcap annotation \not\equiv \bot \qquad compatibility - test \tag{7}$$

$$annotation \sqsubseteq profile \quad match - test \tag{8}$$

To enable the Pocket KRHyper [7] to evaluate the matchmaking all DL axioms are translated into a disjunctive logic program. This translation (similar to [3]) preserves the decidability for acyclic general TBoxes and extends the common subset of DL and LP (DLP) [5] significantly.

To cope with the restrictions on resources of the mobile phones a segmentation of the clausal knowledgebase limits the number of translations from DL to LP and numerous reductions of equivalences to implications reduces the number of generated clauses. As a result the matchmaking turned out to be fast enough to be accepted by a user. Distinguishing between offers and requests was an additional benefit for users. Different from other approaches [4] all decisions are kept on the mobile device to protect the user's profile. Our J2ME based implementation of the reasoner and the matchmaking library enables the development of a new class of applications on independant mobile devices.

References

1. G. Antoniou and F. van Harmelen. Web ontology language: Owl. In S. Staab and R. Studer, editors, *Handbook on Ontologies in Information Systems*. Springer, 2003.
2. F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, and P. F. Patel-Schneider. *The description logic handbook: theory, implementation, and applications*. Cambridge University Press, 2003.
3. P. Baumgartner, U. Furbach, M. Gross-Hardt, and T. Kleemann. Model based deduction for database schema reasoning. In S. Biundo, T. Frühwirth, and G. Palm, editors, *KI 2004: Advances in Artificial Intelligence*, volume 3238, pages 168–182. Springer Verlag, Berlin, Heidelberg, New-York, 2004.
4. F. L. Gandon and N. M. Sadeh. Semantic web technologies to reconcile privacy and context awareness. *J. Web Sem.*, 1(3):241–260, 2004.

5. B. N. Grosof, I. Horrocks, R. Volz, and S. Decker. Description logic programs: Combining logic programs with description logic. In *Proc. of the Twelfth International World Wide Web Conference (WWW 2003)*, pages 48–57. ACM, 2003.
6. L. Li and I. Horrocks. A software framework for matchmaking based on semantic web technology. In *Proc. of the 12th Int. World Wide Web Conf.* ACM, 2003.
7. A. Sinner and T. Kleemann. Krhyper - in your pocket, system description. In *proc. of Conference on Automated Deduction, CADE-20*, 2005.
8. A. Sinner, T. Kleemann, and A. v. Hessling. Semantic user profiles and their applications in a mobile environment. In *Artificial Intelligence in Mobile Systems*, 2004.

A Generic Framework for the Analysis and Specialization of Logic Programs*

Germán Puebla[1], Elvira Albert[2], and Manuel Hermenegildo[1,3]

[1] School of Computer Science, Technical U. of Madrid
{german, herme}@fi.upm.es
[2] School of Computer Science, Complutense U. of Madrid
elvira@sip.ucm.es
[3] Depts. of Comp. Sci. and El. and Comp. Eng., U. of New Mexico
herme@unm.edu

The relationship between abstract interpretation [2] and partial evaluation [5] has received considerable attention and (partial) integrations have been proposed starting from both the partial deduction (see e.g. [6] and its references) and abstract interpretation perspectives. Abstract interpretation-based analyzers (such as the CiaoPP analyzer [9,4]) generally compute a *program analysis graph* [1] in order to propagate (abstract) call and success information by performing fixpoint computations when needed. On the other hand, partial deduction methods [7] incorporate powerful techniques for on-line specialization including (concrete) call propagation and unfolding.

In this work we propose what we argue is the first generic framework for the efficient and precise integration of abstract interpretation and partial deduction from an abstract interpretation perspective, and which combines the best of both worlds. As starting point, we consider state-of-the-art algorithms for context-sensitive, polyvariant abstract interpretation [9,4]. The central idea in this novel framework is to extend such algorithms, which already incorporate success propagation, such that calls which appear dynamically in the analysis graph are not analyzed w.r.t. the definition of the procedure in the original program but w.r.t. possibly *new, specialized definitions* of these procedures. These specialized definitions are obtained by applying powerful techniques for on-line program specialization, including unfolding and abstract executability [10]. Abstract executability allows exploiting analysis information in order to (abstractly) execute certain atoms, which in turn may allow unfolding of other atoms. Also, performing unfolding steps allows us to prune away useless branches, which will result in improved success information. Furthermore, propagating (abstract) success information simultaneously will result in an improved

* This work was funded in part by the Information Society Technologies programme of the European Commission, Future and Emerging Technologies under the IST-2001-38059 *ASAP* project and by the Spanish Ministry of Science and Education under the MCYT TIC 2002-0055 *CUBICO* project. Manuel Hermenegildo is also supported by the Prince of Asturias Chair in Information Science and Technology at UNM.

M. Gabbrielli and G. Gupta (Eds.): ICLP 2005, LNCS 3668, pp. 407–409, 2005.

unfolding. Therefore, key ingredients of our proposal include the accurate success propagation inherent to context-sensitive abstract interpretation and the powerful constant propagation and program transformations achievable by partial deduction.

It should be noted that existing proposals for such integration use abstract interpretation as a *means* for improving partial evaluation rather than as a *goal* at the same level as producing a specialized program. This implies that, as a result, their objective is to yield a set of atoms which determines a partial evaluation rather than to compute a safe approximation of its success. In contrast, a fundamental objective of our work is to improve success information by analyzing the specialized code, rather than the original one. We achieve this objective by smoothly *interleaving* both techniques and this, on one hand, improves success information, even for abstract domains which are not related directly to partial evaluation. On the other hand, with more accurate success information we can improve further the quality of partial evaluation. The overall method thus yields not only a specialized program but also a safe approximation of its behavior.

Our framework is parametric w.r.t. different control strategies (both for local and global control [3]) and abstract domains (including non *downwards-closed* properties). Different combinations of such parameters correspond to existing algorithms for program analysis and specialization. Simultaneously, our approach opens the door to strictly more precise results than those achievable by each of the individual techniques. The framework has been implemented in the context of the `CiaoPP` analysis and specialization system. A complete description of the method (and related techniques) can be found in [8].

References

1. M. Bruynooghe. A Practical Framework for the Abstract Interpretation of Logic Programs. *Journal of Logic Programming*, 10:91–124, 1991.
2. P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. of POPL'77*, pages 238–252, 1977.
3. J.P. Gallagher. Tutorial on specialisation of logic programs. In *Proceedings of PEPM'93, the ACM Sigplan Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 88–98. ACM Press, 1993.
4. M. Hermenegildo, G. Puebla, K. Marriott, and P. Stuckey. Incremental Analysis of Constraint Logic Programs. *ACM Transactions on Programming Languages and Systems*, 22(2):187–223, March 2000.
5. N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, New York, 1993.
6. M. Leuschel. A framework for the integration of partial evaluation and abstract interpretation of logic programs. *ACM Transactions on Programming Languages and Systems*, 26(3):413 – 463, May 2004.
7. J. W. Lloyd and J. C. Shepherdson. Partial evaluation in logic programming. *The Journal of Logic Programming*, 11:217–242, 1991.

8. G. Puebla, E. Albert, and M. Hermenegildo. Abstract Interpretation with Specialized Definitions. Technical Report CLIP6/2005.0, Technical University of Madrid, School of Computer Science, UPM, July 2005.
9. G. Puebla and M. Hermenegildo. Optimized Algorithms for the Incremental Analysis of Logic Programs. In *Proc. of SAS'96*, pages 270–284. Springer LNCS 1145, 1996.
10. G. Puebla and M. Hermenegildo. Abstract Multiple Specialization and its Application to Program Parallelization. *J. of Logic Programming.*, 41(2&3):279–316, November 1999.

The Need for Ancestor Resolution When Answering Queries in Horn Clause Logic

Oliver Ray

Department of Computing, Imperial College London, UK
or@doc.ic.ac.uk

Prolog Technology Theorem Proving. (PTTP) [5] is a well known extension of Prolog for answering queries in first-order logic. PTTP is based on the idea that '*Prolog can be viewed as an "almost complete" theorem prover, which has to be extended by only a few ingredients in order to handle the non-Horn case*' [1]. As explained in [5], PTTP is an efficient realisation of the Model Elimination (ME) calculus [2] that utilises five extensions of standard Prolog:

1. It uses a *sound unification* algorithm with the occur-check;
2. It uses a *complete search* strategy based on iterative-deepening;
3. It adds *contrapositives* of the clauses in the theory in order to provide entry points for all of the literals in those clauses;
4. It uses *ancestor resolution* when unfolding the query in order to overcome the incompleteness of Prolog's SLD resolution;
5. It adds contrapositives for the negation of the query in order to extract *indefinite answers* from successful computations.

Although it is equipped for full clausal reasoning, PTTP has been tailored to some notable Horn clause applications by removing those features deemed unnecessary in the Horn case. For example, the Inductive Logic Programming (ILP) system Progol5 [3] includes a simplified PTTP technique that uses contrapositives in order to query the negative literals entailed by a given Horn theory, but does not support ancestor resolution or indefinite answers.

This paper shows all five extensions above are needed for the soundness and completeness of PTTP in the Horn case (where the theory is a set of Horn clauses and the query is a set of literals — implicitly read as universally and existentially quantified conjunctions, respectively). It also shows how the omission of ancestor resolution is responsible for a recently discovered incompleteness [4] of Progol5 and how indefinite answers can be used to enhance Progol5's learning ability.

Evidently, sound unification and complete search are needed in the Horn case just as much as the general case (even if they are often omitted for efficiency). Contrapositives are also needed in the Horn case to solve queries with negative literals. For example, the negative literal $\neg q(0)$ is entailed by the Horn clauses $p(x) \leftarrow q(x)$ and $\leftarrow p(0)$, but the query $\neg q(y)$ only succeeds with answer $y/0$ if the contrapositives $\neg q(x) \leftarrow \neg p(x)$ and $\neg p(0)$ are also provided.

At first sight, it might appear as if ancestor resolution and indefinite answers are made redundant in the Horn case by the completeness of SLD resolution.

M. Gabbrielli and G. Gupta (Eds.): ICLP 2005, LNCS 3668, pp. 410–411, 2005.

However, the following examples show that they are both in fact needed for the completeness of PTTP even when the theory comprises a single Horn clause:

• First, consider the Horn clause $\leftarrow q(0), q(x)$ and note that although this clause entails $\neg q(0)$, the query $\neg q(y)$ only succeeds with the answer $y/0$ from the contrapositives $\neg q(0) \leftarrow q(x)$ and $\neg q(x) \leftarrow q(0)$ if ancestor resolution is used (i.e. resolved upon literals are retained as framed literals and allowed to resolve away complementary unframed literals to their left).

• Then, consider the Horn clause $\leftarrow q(0), q(1)$ and note that although this clause entails $\exists y(\neg q(y))$, the query $\neg q(y)$ only succeeds from the contrapositives $\neg q(0) \leftarrow q(1)$ and $\neg q(1) \leftarrow q(0)$ if indefinite answers are supported (i.e. contrapositives are added for the negation of the query and any resulting input bindings are used to identify indefinite answers: $y/(0 \text{ or } 1)$, in this case).

Progol5 [3] is a prominent ILP system for generalising sets of positive and negative examples with respect to prior background knowledge. It is based on a method for computing Horn clauses h (hypothesis) that entail a ground atom e (example) relative to a Horn theory B (background). In a nutshell, this method searches for those clauses $h = A_0 \leftarrow A_1, \ldots, A_n$ having a ground instance $h\theta$ such that $B \cup \{\leftarrow e\} \models \neg h\theta$ where $\neg h\theta \equiv \neg A_0\theta \wedge A_1\theta \wedge \ldots \wedge A_n\theta$.

To find these hypotheses, Progol5 first attempts to compute the atom $\neg A_0\theta$ using a Prolog engine supplemented with contrapositives to identify the ground instances of the negative literal $\neg A_0$ entailed by the Horn theory $B \cup \{\leftarrow e\}$. But, without ancestor resolution, the incompleteness of this method is immediate from the examples above. For example, if $B = \{p \leftarrow q(0), q(x)\}$ and $e = p$, then Progol5 cannot compute the hypothesis $h = q(0)$ as the query $\neg q(0)$ only succeeds from the theory $B \cup \{\leftarrow e\}$ using both contrapositives and ancestor resolution.

Although ancestor resolution overcomes the incompleteness of Progol5 noted above, indefinite answers can be used to further enhance its learning ability. In particular, by simply Skolemising any variables in $\neg A_0$ that return disjunctive answers, the Progol5 proof procedure can easily learn sound ILP hypotheses outside its original semantics. For example, if $B = \{p \leftarrow q(0), q(1)\}$ and $e = p$, then the extended procedure could compute the hypothesis $h = q(y)$ as the query $\neg q(y)$ effectively succeeds with the answer y/k for some Skolem constant k.

References

1. P.Baumgartner and U.Furbach. Model Elimination without Contrapositives and its Application to PTTP. *Journal of Automated Reasoning*, 13:339–359, 1994.
2. D. Loveland. *Automated Theorem Proving: A Logical Basis.* North Holland, 1978.
3. S. Muggleton and C. Bryant. Theory Completion Using Inverse Entailment. In *Proc. 10th Int. Conf. on ILP*, LNCS 1866: 130–146, 2000.
4. O. Ray, K. Broda and A. Russo. Hybrid Abductive Inductive Learning: a Generalisation of Progol. In *Proc. 13th Int. Conf. on ILP*, LNAI 2835: 311–328, 2003.
5. M. Stickel. A Prolog technology theorem prover: Implementation by an extended Prolog compiler. *Journal of Automated Reasoning*, 4(4):353–380, 1988.

Modeling Systems in CLP

Joxan Jaffar, Andrew E. Santosa, and Răzvan Voicu

School of Computing, National University of Singapore
{joxan, andrews, razvan}@comp.nus.edu.sg

Constraint Logic Programming has been successful as a programming language, and more recently, as a model of executable specifications. There have been numerous works which use CLP to model programs and which use an adaptation of the CLP proof system for proving certain properties, for example, the XMC system [3] uses SLG resolution on alternation-free μ-calculus formulas, and the work on deductive model checking [1] model for CTL properties on transition systems represented as CLP rules. These, amongst other works, cover a limited class of programs and use specialized proof methods. In our work, we present a systematic method to model a general class of programs, and provide adaptations of the CLP proof systems in order to provide a systematic and general proof method.

We start by representing the behavior of a concurrent program, possibly with a parameterized number of processes, or a high-level specification, in the form a predicate transformer. The method is intuitive: we use, for a program p, a predicate $\mathtt{p}(k, \tilde{X}, \tilde{X}^t)$, where the logic variables $\tilde{X}$ and $\tilde{X}^t$ represent values of program variables at program point k, and at a target program point, respectively. We can model *bottom up* behavior, in which case we capture the strongest postcondition transform, or *top-down* behavior, where we capture the weakest precondition transform. Most importantly, we specify the program property in a rely-guarantee or Hoare-triple style, relating the initial program values to those at target program points, and hence our proof method is *compositional*.

We can model, in addition to traditional sequential and concurrent programs:

- **Parameterized Systems.** Program counters are described symbolically as an array of counters, and an array theory is added to the proof method to handle constraints over arrays.
- **Synchronous and Asynchronous Concurrency.** We consider both non-blocking, as well as blocking, **await** statement-based concurrency, and model both asynchronous and synchronous (e.g. [2]) composition of processes.
- **High-level specifications.**
  - *Timed automata.* We can mix clock and non-clock variables in (possibly non-linear) constraints. Moreover, we do not require region-based analysis for verification. See [2] for more details.
  - *Statecharts.* Different semantics of Statecharts (e.g., Statemate, UML, step, superstep) can be easily encoded and then verified (including history states — often ignored by Statechart verification tools).
  - The list can be extended to any behavioral specifications, e.g., MSC and LSC.

Our model captures the trace semantics, reflecting not only properties of variables, but also properties of runtime behavior, e.g. the value of a variable is monotonically increasing. Further, our model elegantly captures properties of the underlying *runtime system*, thus providing a foundation for resource (e.g. time and space) reasoning. For example:

M. Gabbrielli and G. Gupta (Eds.): ICLP 2005, LNCS 3668, pp. 412–413, 2005.

- **Scheduling.** In a concurrent setting, we explicitly model the policy of choosing which process should execute next. Such modeling can be either very detailed (e.g. round robin), or high level (e.g. a generic fair scheduler).
- **Micro architecture.** Here we model the execution of programs on hardware with instruction and data caches, pipelining, etc.

A safety proof is established by executing a particular goal representing a state or set of states, against the CLP model of the program. Informally, an assertion A of the form $G \models \Psi$ where G is a goal and Ψ a constraint, we perform unfolding toward the objective that each derived goal G' is either

- *directly provable*, ie of the form $p(k,\tilde{X},\tilde{X}^t),\Psi_1 \models \Psi_2$ where $\Psi_1 \models \Psi_2$ can be directly validated (and typically this is done when G' is terminal); or
- *subsumed*, ie G' is an instance of another goal in another derivation sequence, or
- *coinductive*, ie G' can be proved using the assumption that some parent goal is true.

To implement the proof method, we extend CLP with *coinductive tabling*. This allows the use of proof obligations that are assumed but not yet proven according to the principle of the coinductive proof above. This is akin to tabling in logic programming, and a main purpose is termination. A main difference is that coinductive tabling stores proof obligations instead of calls and answers. While there have been similar uses of induction, eg. for program transformation [4], our formulation is new for the purpose of proving general safety assertions. We have implemented a prototype of coinductive tabling as a regular CLP($\mathcal{R}$) program.

In general we need to combine abstraction techniques with our proof method to verify programs with large or infinite state space. We have augmented the proof method above with an abstraction mechanism where any goal in a derivation sequence may be abstracted by replacing the constraint Ψ in G with a more general one Ψ_1, that is, $\Psi \models \Psi_1$. Essentially, a proof obligation $G' \models \Psi$ is correct implies that $G \models \Psi$, where G' is an abstraction of G, is also correct. Note that here we do not require Ψ_1 to be of finite domain.

We have also implemented an abstraction method whose key feature is that it is *intermittent*, that is, abstraction is performed at arbitrarily selected points. The challenge here is to deal with an unbounded number of variables because the relationship between the initial and target values of a program fragment require auxiliary variables in general. A main advantage is performance: in general, performing abstraction entails the cost of theorem-proving. A second advantage is accuracy: we need in general a less detailed abstract description because we only perform abstraction intermittently.

Finally, ongoing work is on automatic discovery of abstraction functions, and on progress properties.

References

1. G. Delzanno, A. Podelski. Constraint-Based Deductive Model Checking. *Int. J. STTT*, 2001.
2. J. Jaffar, A. Santosa, R. Voicu, A CLP Proof Method for Timed Automata, 25^{th} *RTSS*, 2004.
3. Y. S. Ramakrishna *et al*, Efficient Model Checking Using Tabled Resolution, CAV'97.
4. A. Roychoudhury *et al*, An Unfold/Fold Transformation Framework for Definite Logic Programs, TOPLAS 26/3.

A Sufficient Condition for Strong Equivalence Under the Well-Founded Semantics*

Christos Nomikos[1], Panos Rondogiannis[2], and William W. Wadge[3]

[1] Department of Computer Science, University of Ioannina,
P.O. Box 1186, 45 110 Ioannina, Greece
cnomikos@cs.uoi.gr
[2] Department of Informatics & Telecommunications, University of Athens,
Panepistimiopolis, 157 84 Athens, Greece
prondo@di.uoa.gr
[3] Department of Computer Science, University of Victoria,
PO Box 3055, STN CSC, Victoria, BC, Canada V8W 3P6
wwadge@csr.uvic.ca

Abstract. We consider the problem of strong equivalence [1] under the infinite-valued semantics [2] (which is a purely model-theoretic version of the well-founded semantics). We demonstrate that two programs are now strongly equivalent if and only if they are logically equivalent under the infinite-valued logic of [2]. In particular, we show that for propositional programs strong equivalence is decidable but coNP-complete. Our results have a direct practical implication for the well-founded semantics since, as we demonstrate, if two programs are strongly equivalent under the infinite-valued semantics, then they are also strongly equivalent under the well-founded semantics.

1 Introduction

The notion of strong equivalence of logic programs was recently introduced in [1]. Two logic programs P_1 and P_2 are termed strongly equivalent under the answer set semantics if for all logic programs P, $P_1 \cup P$ has the same answer sets as $P_2 \cup P$. Obviously, when two logic programs are strongly equivalent, we can replace one for the other inside a bigger program without any change in the observable behaviour of this program. In [1] it is demonstrated that two programs are strongly equivalent under the answer set semantics if and only if they are equivalent in the logic of here-and-there.

We consider the problem of characterizing the notion of strong equivalence under the infinite-valued semantics of [2] (which is actually a refined version of the well-founded semantics). Our characterization of strong equivalence under the infinite-valued semantics is actually very simple: two programs are now

* This research is supported by ΕΠΕΑΕΚ ΙΙ under the task "ΠΥΘΑΓΟΡΑΣ-ΙΙ: ΕΝΙΣΧΥΣΗ ΕΡΕΥΝΗΤΙΚΩΝ ΟΜΑΔΩΝ ΣΤΑ ΠΑΝΕΠΙΣΤΗΜΙΑ", Project title: *Applications of Computational Logic to the Semantic Web*, funded by the European Social Fund (75%) and the Greek Ministry of Education (25%).

M. Gabbrielli and G. Gupta (Eds.): ICLP 2005, LNCS 3668, pp. 414–415, 2005.

strongly equivalent if and only if they are logically equivalent under the infinite-valued logic of [2]. Moreover, although the underlying logic is based on a truth domain with an infinite number of truth values, we demonstrate that for propositional programs strong equivalence is decidable (but coNP-complete). As an immediate consequence of our characterization, we get a sufficient condition for strong equivalence under the well-founded semantics.

2 Main Technical Results

Our first result states that the notion of strong equivalence under the infinite-valued semantics, coincides with the notion of logical equivalence under the infinite-valued logic that has been developed in [2]:

Theorem 1. *Two programs are strongly equivalent under the infinite-valued semantics if and only if they are logically equivalent under the infinite-valued logic of [2].*

The above theorem gives as an immediate corollary the following:

Corollary 1. *If two programs are logically equivalent in the infinite-valued logic of [2], then they are strongly equivalent under the well-founded semantics.*

The following result demonstrates that the problem of strong equivalence under the infinite-valued semantics is actually decidable. This fact is not obvious since the underlying logic has an infinite number of truth values. The proof is based on the observation that in order to decide if two programs are strongly equivalent, we need to examine only a finite set of interpretations. On the negative side, we also demonstrate that the problem of strong equivalence is co-NP complete.

Theorem 2. *The problem of whether two programs are strongly equivalent under the infinite-valued semantics is decidable but co-NP complete.*

The main open problem of this work is the strengthening of Corollary 1: can the infinite-valued approach be used so as to provide a necessary and sufficient condition for strong equivalence under the well-founded semantics? We believe that a further investigation of this issue would give a better understanding of strong equivalence under canonical model semantics.

References

1. V. Lifschitz, D. Pearce, and A. Valverde. Strongly Equivalent Logic Programs. *ACM Transactions on Computational Logic*, 2(4):526–541, 2001.
2. P. Rondogiannis and W.W. Wadge. Minimum Model Semantics for Logic Programs with Negation-as-Failure. *ACM Transactions on Computational Logic*, 6(2):441–467, 2005.

IMPACT: Innovative Models for Prolog with Advanced Control and Tabling

(Extended Abstract)

Ricardo Rocha[1], Ricardo Lopes[1], Fernando Silva[1], and Vítor Santos Costa[2,⋆]

[1] DCC-FC & LIACC, University of Porto, Portugal
{ricroc, rslopes, fds}@ncc.up.pt
[2] COPPE Systems & LIACC, Federal University of Rio de Janeiro, Brazil
vitor@cos.ufrj.br

During the past years, our research group has been working in the design and implementation of Logic Programming Systems. In previous work, we have produced systems to support sequential, parallel and distributed execution of Prolog; to support novel techniques and models, such as tabling, through the YapTab system [1], or the Extended Andorra Model (EAM), through the BEAM [2]; and to support the combination of the above, such as parallel tabling [3]. With the IMPACT project we want to combine the power of tabling with that of EAM in order to produce an execution model with advanced control strategies that guarantees termination, avoids looping, reduces the search space, and is less sensitive to goal ordering. Ultimately, we believe such a system will allow novel logic programming applications.

To the best of our knowledge, YapTab is the only proposal that compares favorably with current versions of XSB [4], the most successful and well-known tabling Prolog system. YapTab has been designed taking into account the major purpose of further integration with parallelism to achieve the first available parallel tabling computational system, the OPTYap [3].

The Andorra based execution models were designed to explore more advanced control strategies aiming at reducing the search space and maximizing available parallelism. To study whether the EAM with implicit control was practical, we have developed the BEAM, a novel system that refines Warren's original design. Performance results showed that our model is less sensitive to goal ordering and quite effective at reducing the search space.

We have noticed that both EAM and tabling have common characteristics that seem to fit naturally. First, tabling contributes to one of the main premises of the EAM, reducing search space through the reuse of goals. Second, both paradigms are less sensible to goal ordering. Third, tabling avoids looping thus guaranteeing termination for programs with recursive computations. Is thus our belief that BEAM with tabling opens up a design space for novel mechanisms to

⋆ This work has been partially supported by APRIL (POSI/SRI/40749/2001), Myddas (POSC/EIA/59154/2004) and by funds granted to LIACC through the Programa de Financiamento Plurianual, Fundação para a Ciência e Tecnologia and Programa POSC.

M. Gabbrielli and G. Gupta (Eds.): ICLP 2005, LNCS 3668, pp. 416–417, 2005.

improve both control and performance of existing applications and to achieve a more declarative formulation of search and database applications.

To address the integration between tabling and the EAM we have identified the following tasks for the IMPACT project. **(1)** Build an external module in C implementing tabling primitives that provide direct control over the search strategy. This module will allow tabling to be easily incorporated into other Prolog systems. **(2)** Use the external module with BEAM to study how tabling interacts with the EAM. This involves running and validating the system with a set of benchmark applications. **(3)** Design a new model to extend the BEAM system to support tabling running within the EAM environment. There are several major problems that need to be tackled. **(i)** Integrate the basic operations of tabling evaluation, such as checking for variants, checking for new answers, and performing completion, with the BEAM execution rules. This includes studying and defining how to start the execution of tabled subgoals, when new answers should be allowed to be exported, and how completion should be done. Note that whereas these operations are already complex for Prolog, they are even more intricate within the EAM. **(ii)** Study the implications of tabling evaluation on the BEAM rewrite rules. Some of the BEAM rewrite rules will need to be modified to support tabling. We will address the implications on the reduction, splitting, eager-splitting, promotion, propagation, and-compression, deterministic-reduce-and-promote, success propagation, and failure propagation rules. **(iii)** Study how pruning should be implemented on BEAM with tabling. BEAM allows both implicit and explicit pruning, but aggressive pruning may interfere with tabling and result in incomplete tables. **(4)** Implement the combined system. The proposed work will be implemented on top of the existing systems and will profit from the expertise of our research group in the area.

In this project we focus on combining tabling with the EAM. Our ultimate goal is to develop a system that supports simultaneously and efficiently parallelism, tabling, and the EAM model.

References

1. Rocha, R., Silva, F., Santos Costa, V.: YapTab: A Tabling Engine Designed to Support Parallelism. In: Conference on Tabulation in Parsing and Deduction. (2000) 77–87
2. R. Lopes, V.S., Silva, F.: A Novel Implementation of the Extended Andorra Model. In: International Symposium on Pratical Aspects of Declarative Languages. Number 1990 in LNCS, Springer-Verlag (2001) 199–213
3. Rocha, R., Silva, F., Santos Costa, V.: On applying or-parallelism and tabling to logic programs. Journal of Theory and Practice of Logic Programming **5** (2005) 161–205
4. Sagonas, K., Swift, T., Warren, D.S.: XSB as an Efficient Deductive Database Engine. In: ACM SIGMOD International Conference on the Management of Data, ACM Press (1994) 442–453

Using CLP to Characterise Linguistic Lattice Boundaries in a Text Mining Process

Alexandre S. Saidi

LIRIS & Ecole Centrale de Lyon, B.P. 163, 69134 Ecully - France
Alexandre.Saidi@ec-lyon.fr

Abstract. In this paper, we expose the use of CLP in a Textual Data Mining Task. Text Mining process is here applied to a corpus of semi-structured documents like seminary and job announcement. Such documents contain semi-structured sections each of which will be recognised by an automaton whose language is characterised by a set of CLP rules.

1 Introduction and Objectives

The textual data bases constitute the major part of available information. Hence, significant research work concentrate on the Information Extraction (IE) from these databases. Given a corpus, the Information Extraction process applied by *Text Mining* techniques (see e.g. [FAY 96], [HEA 97], [GRI 97]) consists on the search for non explicit information in these texts. Text Mining aims to extract significative informations like the general research directions of a university, location or the subject of a (untagged) seminar in a conference announcement.

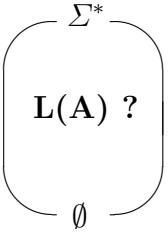


Fig. 1. The search lattice

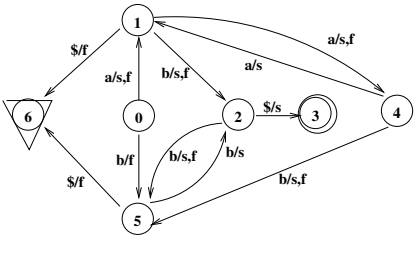


Fig. 2. The inferred automaton A

2 The Task

Given a set of positive (e.g. seminar announcements) and negative samples, the task is to characterise the language that generalises these examples.

Hence, given an alphabet Σ, the search space can be characterised by a (language inclusion) lattice (see Fig. 1) where the top element Σ^* is the set of all possible constructions over Σ and the bottom element is the empty language.

M. Gabbrielli and G. Gupta (Eds.): ICLP 2005, LNCS 3668, pp. 418–420, 2005.

Let $I_+ \in \Sigma^*$ a finite set of positive sentences (to be accepted and generalised) and $I_- \in \Sigma^*$ a finite set of negative sentences (to be rejected), we aim to infer **A** a minimal generalised *deterministic finite state automaton* (DFA) that accepts all sentences belonging to its (possibly infinite) language L(A), and rejects those that do not belong to L(A). Note that $I_+ \subseteq L(A)$ and $I_- \nsubseteq L(A)$.

An example: being interested to the language L = $\{\mathbf{a}^n\ \mathbf{b}^m \mid n, m$ both even or both odd$\}$[1], we have (Fig 2 is a minimal generalised DFA for L):

Σ={a, b}, I_+ = {ab, aabb, aaabbb, ... } , I_- = {a, b, aab, abb, ... }.

3 Equivalence Classes and Rule Inference by CLP

The following set of constraints (implemented by GNU Prolog) characterises the lattice i which we will search a solution that minimises the final automaton. Elements of $I = I_+ \cup I_-$ are first described by a set of automata giving a tree T of automata. Then every pair of transition like ($r_1 : \alpha \times s'_1 \rightarrow s_1$) and ($r_2 : \beta \times s'_2 \rightarrow s_2$) in T is considered by the following predicate. $[x]$ denotes the equivalence class of the lexeme $x \in \Sigma^+$. F denotes the set of final states in T.

Predicate Congruence(r_1, r_2) adds constraints to the constraint store θ
Let r_1 and r_2 be the above rules (transitions) with α,$\beta \in \Sigma$

$r_1 : [\alpha] \times s'_1 \rightarrow s_1$ $\qquad$ $r_2 : [\beta] \times s'_2 \rightarrow s_2$

(1) if s_1 and s_2 are different final states in ($F_+ \times F_-$) then tell $[s_1] \neq [s_2]$.
(2) if $[\alpha] = [\beta]$ then tell ($[s'_1] = [s'_2] \Rightarrow [s_1] = [s_2]$) *(DFA condition)*
(3) if $[\alpha] \neq [\beta]$ then tell $[s_1] \neq [s_2]$

4 Conclusion

In the current realisation, an announcement (e.g. seminar or job) contains several sections (name, address, salary, job description or seminary object, date, etc.) each of which is translated to an automata. Then a (prefix-) tree T of automata is constructed for each section before applying the above predicate that creates equivalence classes for the DFA states. Having generalised and learned how to recognise each section of the announcement corpus, template slots are filled to summarise (new) announcements. Theses templates can then be used by classical database engines. Note that for some nontrivial sections, a preliminary partial but time consuming syntactical analysis may be required (thanks to DCG).

A system totally based on CLP is realized and applied to corpora of various announcement texts. We plan to translate parts of this system in some imperative language (Java, C++, ADA) and asset by applying it to real world databases.

[1] Visually, the set I_+ may be thought to describe words in $\{a^n b^n \mid n \geq 1\}$ which is a Context-Free language that we do not treat (being limited to regular languages).

References

[FAY 96] U. fayyad & all, *From Data Mining to KD : An overview.* in Advances in Knowledge Discovery and Data Mining, MIT Press, Cambridge, Mass 1996.

[GRI 97] R. Grishman, *Information Extraction: Techniques and Challenges*, http://citeseer.nj.nec.com/grishman97information.html.

[HEA 97] M.A. Hearst, *Text Data Mining : Issues, Techniques and Relationship to Information Access*, Presentation Notes for UW/MS Workshop on data mining, July 1997.

Hybridization of Genetic Algorithms and Constraint Propagation for the BACP

Tony Lambert[1,2], Carlos Castro[3], Eric Monfroy[1,3],
María Cristina Riff[3], and Frédéric Saubion[2]

[1] LINA, Université de Nantes, France
Firstname.Name@lina.univ-nantes.fr
[2] LERIA, Université d'Angers, France
Firstname.Name@univ-angers.fr
[3] Universidad Santa María, Valparaíso, Chile
Firstname.Name@inf.utfsm.cl

1 Introduction

Constraint Satisfaction Problems (CSP) provide a modelling framework for many computer aided decision making problems. Many of these problems are associated to an optimization criterion. Solving a CSP consists in finding an assignment of values to the variables that satisfies the constraints and optimizes a given objective function (in case of an optimization problem). In this paper, we extend our framework for genetic algorithms (GA) as suggested by the reviewers of our previous ICLP paper [5]. Our purpose is not to solve efficiently the Balanced Academic Curriculum Problem (BACP) [2] but to combine a genetic algorithm with constraint programming techniques and to propose a general modelling framework to precisely design such hybrid resolution process and highlight their characteristics and properties.

2 Computational Frameworks: CP vs GA

Constraint propagation, one of the most famous techniques for solving CSP consists in iteratively reducing domains of variables by removing values that do not satisfy the constraints. These reductions must be interleaved with a splitting mechanism in order to obtain a complete solver. Constraint optimization problems, although similar to constraint solving, is comparatively harder because it only accepts solutions that minimize or maximize a given objective function while satisfying the constraints. The key principle of Genetic Algorithms approach states that, species evolve through adaptations to a changing environment and that the gained knowledge is embedded in the structure of the population and its members, encoded in their chromosomes. Applying a genetic algorithm consists in iteratively generating better and better individuals w.r.t. an evaluation function.

In the context of GA, for the resolution of a given CSP (X, D, C), the search space can be usually defined with the set of tuples $D = D_1 \times \cdots \times D_n$. We

M. Gabbrielli and G. Gupta (Eds.): ICLP 2005, LNCS 3668, pp. 421–423, 2005.

consider populations g of size i, $g \subseteq D$ such as $|g| = i$. An element $s \in g$ is an individual and represents a potential solution to the problem.

In order to handle the different data structures associated to each side of the resolution (GA & CP), we add a particular genetic factor to the search space to each CSP, on which GA will work and where optimization will be done. A CSP with genetic factor (ogCSP) for optimization is defined by a sequence (D, C, p) where $p = (g_1, \ldots, g_n)$ corresponds to sequence of generations.

Based on the framework of chaotic iterations [1], resolution will be achieved according to the generic algorithm through a partial ordering where the set of functions is instantiated by reduction, split and GA functions.

3 Experimentation

The purpose of this section is to highlight the benefit of our framework for hybridization. We consider the bacp8, bacp10 and bacp12 problems issued from the CSPlib [3]. We control the rates of each family of functions *reduction*, *split* and *ga* by giving as strategy a tuple $(\%_{red}, \%_{sp}, \%_{ga})$ of application rates. These values correspond indeed to a probability of application of a function of each family but, in practice, we measure in Fig 1 the real rate of participation during the objective function evolution.

The most interesting in such an hybridization is the completeness of the association GA-CP, and the roles played by GA and CP in the search process : GA optimizes the solutions in a search space progressively being locally consistant (and thus smaller and smaller) using constraint propagation and split.

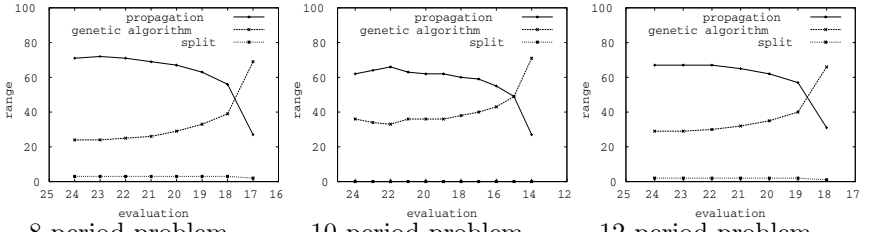


Fig. 1. Evolution of CP vs GA during the optimization process

Concerning strategies using *GA* **and** *CP* **alone**, in this implementation, CP is unable to find a feasible solution in 10 minutes cpu time. GA is able to find alone the optimal but is 10 times slower w.r.t. the hybrid.

4 Perspectives and Conclusion

In this paper, we have used a more suitable general framework to model hybrid optimization solving algorithms. The results over the BACP show the benefits of

our framework and of hybridization. They also allow us to identify the interaction between the different resolution mechanisms. Such studies could be used to tune general purpose hybrid solvers in the future.

References

1. K. R. Apt. From chaotic iteration to constraint propagation. In *Proceedings of ICALP'97*, pages 36–55. Springer-Verlag, 1997.
2. C. Castro and S. Manzano. Variable and value ordering when solving balanced academic curriculum problems. In *Proceedings of 6th Workshop of the ERCIM WG on Constraints. CoRR cs.PL/0110007*, 2001.
3. I. Gent, T. Walsh, and B. Selman. http://www.csplib.org, funded by the UK Network of Constraints.
4. J. H. Holland. *Adaptation in Natural and Artificial Systems*. 1975.
5. E. Monfroy, F. Saubion, and T. Lambert. On hybridization of local search and constraint propagation. *Proc. of ICLP'04*, *LNCS 3132*, pp 299–313. Springer, 2004.

The MYDDAS Project: Using a Deductive Database for Traffic Characterization

Michel Ferreira

DCC-FC & LIACC, University of Porto,
Rua do Campo Alegre, 823, 4150-180 Porto, Portugal
michel@ncc.up.pt

The MYDDAS[1] project (*Mysql/Yap Deductive DAtabase System*) is developing a deductive database system by coupling Yap Prolog with MySQL [1]. Although this coupling approach between a logic system and a relational database management system has been quite explored [5], our system tries to go a step further in terms of the degree of *tightness* in the interface architecture between the two systems. Examples of this improved integration include the representation of relational tuples directly in choice-points, with a transparent support for *cuts* over EDB predicates; the extended use of the tabling engine of Yap [3], with the ability to persistently store the table data structure in MySQL relations; and the development of automatic view-level transformations using information from existing MySQL indexes and MySQL query optimizer. The MYDDAS system should also be able to explore the combination of tabling with or-parallelism provided by the OPTYap engine [4] of Yap in the concurrent evaluation of database goals.

Together with the development of the deductive database engine, the MYDDAS project also proposed to develop a real-world application supported by this deductive engine. The application being developed aims at real-time traffic characterization based on a mobile sensors network. In the next lines we concisely describe the main features of this application and justify its appropriateness for support by a deductive database system.

Traffic characterization based on the dynamic mobility pattern of vehicles is an interesting problem for modelling using a deductive database, for two main reasons. First, because of the dimension of the data involved, which is very large (not because of static geographic information but because of GPS logs collected simultaneously by a large number of vehicles). Second, because the mobility pattern of vehicles follows consistent rules that we can try to model in a *clausal* form using Prolog rules. For instance, we can have road classification, road signs identification (stop signs, traffic lights) and daily congestion hours on roads, represented by Prolog rules, instead of by static geographic information.

In our application the mobility sensors are installed on vehicles and can consist of a laptop computer, generic PDA or Linux gumstixs, coupled with a GPS receiver and wi-fi capability. Each sensor has an instance of the

[1] This work has been partially supported by MYDDAS (POSC/EIA/59154/2004) and by funds granted to LIACC through the Programa de Financiamento Plurianual, Fundação para a Ciência e Tecnologia and Programa POSC.

M. Gabbrielli and G. Gupta (Eds.): ICLP 2005, LNCS 3668, pp. 424–426, 2005.

deductive database, with limited information contextualized for the current location. Vectorial road maps are stored in MySQL relations using the spatial extensions available since version 4.1 of MySQL. Our coupling interface provides the representation as Yap terms of the several geometry objects, such as points and lines.

The definition of rules for the different traffic features based only in mobility is one of the most interesting aspects in this application. There are two different approaches that can be taken: programmer defined rules, based on the knowledge of the mobility pattern of each traffic feature; or definition based on *induction*, using examples of mobility and an inductive logic programming system, such as APRIL [2], for learning the rules. Rules for identifying, as soon as possible, momentaneous traffic jams caused by accidents or similar events, are also fundamental for the dynamic calculation of *fastest* route.

Another interesting aspect in this application is the communication model between the different sensors. Clearly, the existence of stationary sensors with higher processing power, which collect information from the mobile sensors and dynamically update road traffic conditions sending back this updated information, is the obvious communication model. Nevertheless, a vehicle-to-vehicle communication in areas with no stationary sensors can be interesting. Consider a traffic jam at a particular location in a highway. This information can be transported by vehicles on the opposite lane to vehicles which are approaching the traffic jam. Clearly, the time lag of data is a crucial factor in this traffic information system. A scaled evaluation of this communication model will be based on traffic simulators.

The obvious and straightforward function of such a traffic information system is the ability to calculate routes based on real-time traffic characterization, avoiding congested roads and finding the *fastest* route for a given destination. The ability to automatically complete maps with traffic controlling features and road congestion characterization is also interesting.

The MYDDAS project is a three years project and is currently in the first year. Implemented work so far is mainly on the interfacing between Yap and MySQL, particularly to provide spatial extensions to Yap, and in the modelling of the communication scheme between sensors. The project joins together people from the areas of logic programming implementation, and communication and information networks. For more information and current prototypes, visit MYDDAS's homepage at `http://www.ncc.up.pt/~michel/MYDDAS/`.

References

[1] M. Ferreira and R. Rocha. The MyYapDB Deductive Database System. In J. J. Alferes and J. Leite, editors, *Proceedings of the 9th European Conference on Logics in Artificial Intelligence, JELIA'2004*, number 3229 in LNAI, pages 710–713, Lisbon, Portugal, September 2004. Springer-Verlag.

[2] N. Fonseca, F.Silva, R. Camacho, and V. S. Costa. Induction with April - A preliminary report. Technical Report DCC-2003-02, DCC-FC & LIACC, Universidade do Porto, 2003.

[3] R. Rocha, F. Silva, and V. Santos Costa. YapTab: A Tabling Engine Designed to Support Parallelism. In *Conference on Tabulation in Parsing and Deduction*, pages 77–87, 2000.

[4] R. Rocha, F. Silva, and V. Santos Costa. On a Tabling Engine That Can Exploit Or-Parallelism. In *International Conference on Logic Programming*, number 2237 in LNCS, pages 43–58. Springer-Verlag, 2001.

[5] K. Sagonas, T. Swift, and D. S. Warren. XSB as an Efficient Deductive Database Engine. In *ACM SIGMOD International Conference on the Management of Data*, pages 442–453. ACM Press, 1994.

Open World Reasoning in Datalog*

Gergely Lukácsy and Zsolt Nagy

Budapest University of Technology and Economics,
Department of Computer Science and Information Theory,
1117 Budapest, Magyar tudósok körútja 2., Hungary
{lukacsy, zsnagy}@cs.bme.hu

Introduction and Background. This work is part of the SILK project which aims at supporting *semantic integration* using logic programming [2]. The main idea of our approach is to collect and manage meta-information on the sources to be integrated. These pieces of information are stored in the model warehouse of the SILK system in the form of models, constraints and mappings. By using logic, all these are represented in a uniform way.

Our modelling language incorporates UML-like constructs as well as description logic (DL) formulae. In the latter case we can build composite concepts, e.g. $\mathsf{Human} \sqcap \neg\mathsf{Male} \sqcap (\geqslant 3\,\mathsf{hasChild})$ represents mothers with at least 3 children.

The process of querying the models is called *mediation*. Mediation decomposes complex queries formulated in terms of simple ones answerable by individual information sources.

We have already dealt with the problem of querying the object oriented models [2]. Our current task is to investigate the possibilities how to do the same for the description logic based models. Querying these models normally involves using so called *ABox* reasoning techniques. The existing ABox algorithms however turn out to be very slow when dealing with large amount of instances, or restrict the structure of the ABox significantly [1].

We are working on designing efficient algorithms which transform a description logic concept description into a Datalog query, where individual subgoals access the databases which store the instances.

Open World Reasoning. A challenge is that we have to consider the so called *open world assumption (OWA)*, where the absence of information only indicates the lack of knowledge.

For instance, let us examine the well-known example about the family of Iocaste and Oedipus [3]. There are four instances in the ABox: I, O, P and T. O is known to be patricide, while T is known not to be patricide. We do not know anything about the other two instances, regarding them being patricide or not. We also know that the children of I are O and P, P is a child of O, and T is a child of P.

We will look for the answer to the following query: is there a person who has a patricide child who, in turn, has a non-patricide child? The answer is yes, since assuming that P is either patricide or non-patricide, I is found to be a solution.

* This work is part of the SINTAGMA project which is supported by the Hungarian NKFP programme under grant no. 2/052/2004.

M. Gabbrielli and G. Gupta (Eds.): ICLP 2005, LNCS 3668, pp. 427–428, 2005.

Thus the OWA may require case analysis. Note that the need for case analysis can be detected by analyzing the structure of the query.

Possible Approaches. The most obvious approach is using brute force to check all the possible cases. A more sophisticated solution would be to partially evaluate the existing ABox inference algorithms. In this paper we present our ideas about a technique based on retrieving so-called supersets. We use the description logic language $\mathcal{ALC}$ over an empty terminology store (TBox).

Current Stage of Research. Our algorithm is based on the idea of first calculating a ,,superset" of the answer-set together with n-tuples supporting candidate solutions (superset table). We decompose the concept in question to so-called path queries of form $\exists R_1.(C_1 \sqcap \exists R_2.(C_2 \sqcap \ldots \exists R_n.C_n))$. We then weaken this query and calculate the superset table. This is examplified by the following a Datalog program (the ABox contains Datalog facts of form `patricide(i)`, `not_patricide(i)` and `hasChild(i,j)`, where `i` and `j` are instances):

```
superset(X, Y, Z) :- hasChild(X, Y),  \+ not_patricide(Y),
                     hasChild(Y, Z),  \+ patricide(Z).
| ?- findall(E, (superset(X, Y, Z), E = X-Y-Z), Superset).
Superset = ['I'-'O'-'P', 'I'-'P'-'T', 'O'-'P'-'T']
```

The query above enumerates all the grandfather-father-child triples that contain candidate solutions for the query. After finding the `Superset`, the solutions that answer the query are derived from it using resolution-based techniques.

In the superset query above, the instances matching the logic variable `Y` are candidates for patricide instances. If an instance is known to be `not_patricide`, it will not be included in the `Superset`.

It can also be seen from the code that the ABox is only accessed when retrieving the `patricide` and `not_patricide` instances, and the pairs of instances that are in `hasChild` relation with each other. If the ABox contained information about other domains, we would not need to access them.

Future Work. We actually consider this technique as only the first step, namely, we are investigating the possibilities to extend the algorithm to more powerful description logics as well. If the TBox contained axioms, the termination of the Datalog programs would have to be assured by blocking techniques.

References

1. I. Horrocks, L. Li, D. Turi, S. Bechhofer: The instance store: DL reasoning with large numbers of individuals. Proc. of the 2004 Description Logic Workshop 31-40
2. Tamás Benkő, Péter Krauth, and Péter Szeredi. A logic-based system for application integration. In *Proceedings of the International Conference on Logic Programming*, volume 2401 of *Lecture Notes in Computer Science*, pages 452–466. Springer, 2002.
3. F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press, 2003.

Optimizing Queries for Heterogeneous Information Sources

András G. Békés

Budapest University of Technology and Economics,
Műegyetem rkp. 3-9. 1111, Budapest, Hungary
bekesa@sch.bme.hu

1 Introduction

Integration of heterogeneous information sources requires building an infrastructure for accessing several different information sources together. One task of the integration is to provide a uniform interface to the different information sources (databases, directory servers, web services, web sites, XML files). The other task is the semantic integration: the meaning of the stored data is also different in the different sources.

In the SILK [1] system (Semantic Integration with Knowledge and Logic), semantic integration is carried out by providing a high-level model and the mappings between the high-level model and the models of the sources. When executing a query of the high-level model, the query has to be transformed to queries of the sources and to the code performing the semantic transformation of the data. The component which translates a high-level query to a low level query is the Query Planner. The Query Planner builds the set of low-level queries, and a further component is needed to create an efficient query plan from it.[1] This component in the SILK system is called the Query Optimizer.

2 Optimizing Query Plans

In the SILK system, the query plan, the result of query planning is a piece of Prolog code, which when executed, enumerates the answers to the high-level query. The Query Optimizer transforms this Prolog code to a piece of code which has the same solution set, while it has better characteristics in some ways (e.g. shorter execution time).

The input query plan is a Prolog predicate body in a disjunctive normal form. This Prolog code requires call reordering to be executable: while some sources can only be called with arbitrary argument instantiations (predicates representing SQL tables), some predicates can be called with a certain subset of their arguments instantiated (predicate representing web services, etc).

The main and compulsory goal of the optimization step is to make the query callable. The available modes of the predicates are given, and with this information at hand it is decidable whether a sequence of goals is actually callable. The

[1] This can be compared to writing a Prolog Program with considering only the declarative semantics, and modifying it to fit the execution mechanism.

M. Gabbrielli and G. Gupta (Eds.): ICLP 2005, LNCS 3668, pp. 429–430, 2005.

secondary goal of the optimizer is to lower the total cost of calling the query, which is basically the estimated execution time of it. For this, some statistical information is available on the average execution time of the predicates, and also on the number of their solutions.

With this information at hand, the optimizer not just rearranges the order of goals in the disjunctive branches, but it does other manipulations on the code in the hope of obtaining a piece of code (i.e. a query) with better performance characteristics. The possible optimizations examined so far are the following:

Reordering: when considering a sequence of predicates, there can be several orders which are callable, but they can have different total costs. Consider the predicates `a` and `b` where expectedly `a` will have less solutions than `b`, then the sequence `a,b` is considered cheaper than `b,a`, assuming that `a` and `b` themselves have the same cost. If the goals are not independent, then the difference between the costs of the two query plans can be even more significant.

Disjunction: a query with similar goals (unifiable with some constraints) in the disjunctive branches can be transformed to have more levels of disjunctions by moving the common goals in front of an inner disjunction. For example, the input query `( b,a ; c,a ; d,e )` can be transformed to `( a,(b ; c) ; d,e )`[2]. The former leads to a cheaper query if the call to `a` does not depend on the instantiations made by `b` and `c`, and a more expensive query if the call to `a` without the instantiations made by `b` and `c` has many solutions, which leads to a larger search space.

Collection of Common-Source predicates: in case of source predicates which will call a common information source, sometimes it may be beneficial to submit a compound query to the source, rather than the simple ones. For example, the query `sql_table1(A,B), sql_table2(B,C)` would lead to the execution of two separate SQL queries, while the SQL engine can perform the join operation if given a compound query, which leads to potentially better performance. In the SILK system, common-source predicates are wrapped in a meta-predicate, which generates the compound source query at run-time.

3 Status of the Research

The current implementation of the Query Optimizer supports the three mentioned optimizations, besides that it generates callable queries. The development of the SILK system continues, we are in the course of exploring further optimization possibilities.

Reference

1. Benkő, T., Krauth, P., Szeredi, P.: A logic-based system for application integration. In: Proceedings of the International Conference on Logic Programming. Volume 2401 of Lecture Notes in Computer Science., Springer (2002) 452–466

[2] Note that the goals in the disjunctive branches of the input query are considered sets, rather than sequences.

Denotational Semantics Using Horn Concurrent Transaction Logic

Marcus Vinicius Santos

Department of Computer Science,
Ryerson University, Toronto, Canada
m3santos@ryerson.ca

In this work we propose to use a Horn fragment of Concurrent Transaction Logic ($\mathcal{CTR}$) as an intuitive logic framework to specify semantics of concurrent programming languages.

Using Horn logic to specify a programming language has been suggested before [5,6,2]. By specifying a programming language we mean writing semantics, all three semantics–operational, denotational, and axiomatic–in Horn logic, which is also executable. Slonneger convincingly demonstrated that, for the specification of denotational semantics, Prolog can be regarded as superior to imperative languages. Gupta [2] expanded on the idea and showed how Horn logic denotations lead to some interesting practical applications, such as automatic program verification and automatic generation of compilers. The work presented here builds on the aforementioned works and extends that of [2] by providing a Horn logic denotational semantics for concurrent programming languages.

Using logic for expressing concepts of parallelism and mobility in computation has been proposed by Milner in [3]. There he presented a calculus, further developed in [4] and called π-calculus. The communication paradigm within $\mathcal{CTR}$ is inspired by the π-calculus. However, $\mathcal{CTR}$ is a *programming* logic, while π-calculus is an algebra for *specifying* and *verifying* finite-state concurrent systems (which databases and logic programs are not). $\mathcal{CTR}$ [1] is an extension of classical logic that seamlessly integrates concurrency and communication with queries and updates. It has a purely logic semantics, including a natural model theory and a sound-and-complete proof theory. Like classical logic, $\mathcal{CTR}$ has a Horn fragment, here called $\mathcal{CTR}_{\mathcal{H}}$, with a procedural interpretation, in which programs can be specified and executed. Next are some examples of $\mathcal{CTR}_{\mathcal{H}}$ formulas: a *concurrent serial goal* is a formula of the form $\phi_1 \otimes \phi_2 \mid \varphi_1 \otimes \odot\varphi_2$, which means: "execute concurrently the predicates $\phi_1 \otimes \phi_2$ and $\varphi_1 \otimes \odot\varphi_2$. To execute $\phi_1 \otimes \phi_2$, first do ϕ_1 then ϕ_2, and similarly for $\varphi_1 \otimes \odot\varphi_2$, except that φ_2 is executed atomically, *i.e.*, without interleaving with other predicates"; *concurrent Horn rules* are expressions of the form $p \leftarrow q$, where q is a concurrent serial goal.

To provide logical denotations in $\mathcal{CTR}_{\mathcal{H}}$ for a typical stateful, imperative programming language, we follow an approach similar to those based on plain Horn logic (*e.g.*, [2]). More specifically, *syntax* and *valuation functions (predicates)* are specified analogously to plain Horn logic approaches. To specify the *semantic algebra* we use $\mathcal{CTR}_{\mathcal{H}}$'s elementary database update operations, which resemble Prolog's predicates *assert* and *retract*. However, unlike Prolog, $\mathcal{CTR}_{\mathcal{H}}$'s elementary operations are integrated into a complete logical system, which enable us to implement the notion of scope and store

M. Gabbrielli and G. Gupta (Eds.): ICLP 2005, LNCS 3668, pp. 431–432, 2005.

within the confines of the logic programming tradition. To avoid using database update predicates, an alternative implementation technique would be to represent the notion of store and scope using lists, usually thread through the valuation predicates. However, we deem such approach renders the valuation predicates unwieldy.

As seen above, $\mathcal{CTR}_{\mathcal{H}}$ provides logic connectives for modeling the concurrent execution of complex processes. Such processes execute in an interleaved fashion and can communicate and synchronize themselves. To provide logical denotations for shared-state concurrent languages we use the same semantic algebra of stateful, imperative languages. We model the semantics of explicit or implicit locking mechanisms, which coordinate the execution of concurrent parts of a program, via $\mathcal{CTR}_{\mathcal{H}}$'s modal operator, $\odot$, for atomic execution. For example, in a sequential language, the valuation predicate for an assignment statement can be specified as follows: $comm(assign(I, E)) \leftarrow expr(E, Val) \otimes update(I, Val)$. In a concurrent language, we must ensure an assignment statement executes "atomically", *i.e.*, non interleaved with other concurrent parts. Formally: $comm(assign(I, E)) \leftarrow \odot(expr(E, Val) \otimes update(I, Val))$.

To provide Logical denotations for data-driven concurrent languages we use a different semantic algebra. The algebra specifies a store which consists of a set of variables, in which each variable is initially unbound, but once bound it stays bound throughout the computation and is indistinguishable from its value. Hence variables in a data-driven language support *dataflow execution*, *i.e.*, an operation waits until all arguments are bound before executing. If an argument is required for a computation but a value is never bound to it, then the execution will result in failure. To model this sort of communication through dataflow synchronization we resort to $\mathcal{CTR}_{\mathcal{H}}$'s inference system. In $\mathcal{CTR}_{\mathcal{H}}$, if an execution path fails, the inference system backtracks and attempts another path of execution by picking another concurrent part for execution. This amounts to synchronizing the execution with the availability of data.

Our results have shown that the semantics of programming languages in general, and concurrent languages in particular, can be fully specified in the $\mathcal{CTR}$ logic framework. We were also successful in demonstrating that $\mathcal{CTR}$-based logical denotations provide a simple, unified formal semantics for a such languages, which can also serve as a prototyping tool for the language developer.

References

1. A.J. Bonner and M. Kifer. Concurrency and communication in transaction logic. In *Joint Intl. Conference and Symposium on Logic Programming*, pages 142–156, Bonn, Germany, September 1996. MIT Press.
2. G. Gupta. Horn logic denotations and their applications. In *Workshop on Current trends and Future Directions in Logic Programming Research*, 1998.
3. R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.
4. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, II. *Information and Computation*, 100(1):41–77, September 1992.
5. Chris D. Moss. How to define a language using prolog. In *Acm Symposoium on Lisp and Functional Programming*, pages 67–73, 1982.
6. Ken Slonneger. Implementing denotational semantics with logic programming. In *CSC '92: Proceedings of the 1992 ACM annual conference on Communications*, pages 337–344, New York, NY, USA, 1992. ACM Press.

Gentra4cp: A Generic Trace Format for Constraint Programming*

Ludovic Langevine** and The French RNTL OADymPPaC Team***

INRIA Rocquencourt, France
langevin@sics.se

Introduction. Several debugging tools have been designed for constraint programming (CP). There is no ultimate tool, that satisfies all the needs, but rather a set of complementary tools. Most of them are dynamic tools. They collect data from the execution and produce abstract views of this execution, for instance a search-tree, the evolution of some domains, or an application-specific display.

So far, there are two issues concerning CP debugging tools. Firstly, each tool is dedicated to a given platform: there is no sharing of tools among the CP platforms, whereas most of solvers are based on the same techniques. Secondly, the extraction of execution data requires the solver to be instrumented. Such instrumentation is tedious and needs to access the solver code. We propose to address those two issues by means of a generic trace format which allows the execution to be described as a sequence of elementary events reflecting the behavior of the search procedure and the propagation process. The tools can then pick in the trace the data they need.

The genericity of the trace format is twofold. It is generic with respect to the tools since the trace contains all the data they need. It is also generic with respect to the solvers, as the very same set of events can reflect the executions of many solvers. The difficult and tedious work of instrumenting the solver is made only once. The efficiency of this scheme is ensured by an adaptation of the trace to the needs of a specific tool. There exist four tracers implementing Gentra4cp, namely for CHIP, Choco, PaLM and GNU-Prolog. Another tracer is currently under implementation within SICStus Prolog. Several tools are freely available.

A Semantics to Specify the Trace. The trace is based on an abstraction of the solver state, including the domains and the constraint store. This abstract state specifies what in the execution state can be observed by a tool. The trace format defines a set of possible events that are elementary modifications of this state. Each one of these events is specified by a state transition rule [1]. This semantics helps interpret the trace.

A Trace of Search and Propagation. The execution of a constraint program is often viewed as the overlapping of two levels: search and propagation. The trace format follows the same distinction.

* This work has been partly supported by the French RNTL OADymPPaC project.

** Currently ERCIM fellow at SICS, Uppsala, Sweden.

*** Pierre Deransart, François Fages (INRIA), Jean-Daniel Fekete, Mohammad Ghoniem, Narendra Jussien (EMN), Mireille Ducassé, Erwan Jahier (IRISA), Alexandre Tessier, Willy Lesaint, Gerard Ferrand, Ali Ed-Dbali (LIFO).

M. Gabbrielli and G. Gupta (Eds.): ICLP 2005, LNCS 3668, pp. 433–434, 2005.

The search level is described by 9 different events. Firstly, three events deal with the creation of entities (creation of a variable, of a constraint, adding of a constraint into the store). Three specific events are used to trace solutions, failures and choice-points. Most of search-strategies can be modeled as the traversal of a search-tree. The *back-to* event aims at tracing the restoration of a former choice-point. The latter four events (solution, failure, choice-point and back-to) can trace any tree-based search strategy, such as chronological backtracking associated to depth-first search, branch-and-bound, or dynamic backjumping. Some search strategies cannot be described as a search-tree, for instance repair techniques such as MAC-DBT. Two additional events, *relax* and *restore* allow tracing the relaxation of a constraint and an elementary restoration of a domain. It is thus possible to trace a large variety of search strategies [1].

Different solvers exhibit different propagation strategies (e.g. variable- or constraint-oriented propagation queues, and some priorities). The trace format models the common behaviors while allowing solver specific extensions. Six different events have been defined to trace the propagation process. They capture the common concepts of the solvers we studied. Most of the differences of these solvers are reflected by the order in which those events occur. The *reduce* event traces an elementary domain reduction. Four events describe the propagation loop: the awakening of a constraint, its suspension, the detection of its entailment or of its unsatisfiability. Those five events are generic: they can be found in many constraint solvers, whatever the exact propagation strategy is. The sixth event, *schedule*, is used to trace solver-specific aspects of the propagation.

Easy Development of Tools Thanks to XML. The trace format is an XML document type. As a widely-used standard, XML enables dozens of libraries to be used to parse the trace. XML answers the needs of trace structuring thanks to attributes and nested elements: an execution event is an XML tag that encloses all its attached data. WBXML, a binary representation using a table of symbols, copes with the verbosity of XML and speeds up the parsing of the trace.

Trace Content Negotiation. The trace format makes for a potentially very detailed description of the execution. This potential trace is not meant to be exhaustively generated. The format provides a protocol between the debugging tool and the tracer. This protocol is used to set the actual level of details. This level of details can even be modified during the execution. This protocol is flexible enough to cope with the versatility of the tools and the evolution of their needs.

Evolution of the Trace Format. The OADymPPaC project is now finished, but the Gentra4cp format is still under development by instrumentation of new solvers, (e.g. SICStus Prolog). In order to take advantage of these experiences, a new SourceForge project has been set up (see http://tra4cp.sf.net).

Reference

1. The OADymPPaC Project. Generic trace format for constraint programming. http://contraintes.inria.fr/OADymPPaC/Public/Trace/, May 2004.

Analyses, Optimizations and Extensions of Constraint Handling Rules: Ph.D. Summary

Tom Schrijvers*

Dept. of Computer Science,
K.U.Leuven, Belgium

Abstract. This is a summary of the Ph.D. thesis of Tom Schrijvers [4].

Constraint Handling Rules (CHR) [3] is a rule-based language commonly embedded in a host language. It combines elements of Constraint Logic Programming and term rewriting. Several implementations of CHR exist: in Prolog, Haskell, Java and HAL. Typical applications of CHR are in the area of constraint solving, but currently CHR is also used in a wider range of applications, such as type checking, natural language processing and multi-agent systems.

In this work we contribute program analyses, program optimizations and extensions of the CHR language.

For the optimized compilation of CHR we present several new optimizations: code specialization for ground constraints, anti-monotonic delay avoidance, hashtable constraint stores and a new late storage optimization. These and other optimizations have been implemented in a new state-of-the-art CHR system: the K.U.Leuven CHR system [5] which is currently available in SWI-Prolog [10], XSB [9] and hProlog [2].

Abstract interpretation is a general technique for program analysis [1]. We propose a framework of abstract interpretation for the CHR language [7], in particular for the formulation of analyses that drive program optimization. This frameworks allows for the uniform formulation of program analyses as well as easier improvements and combinations of existing analyses. We also evaluate analyses for theoretical properties, confluence and time complexity, on a practical case study to investigate their relevance.

We contribute two extensions to the expressivity of CHR. The first extension comprises the integration of CHR with tabled execution [8]. Tabled execution avoids many forms of non-termination and is useful for automatic program optimization through the dynamic reuse of previous computations. The second extension automatically provides implication checking functionality to constraint solvers written in CHR [6]. Implication checking is an essential building block for formulating complex constraints in terms of basic constraints and for composing constraint solvers.

* Research Assistant of the Fund for Scientific Research - Flanders (Belgium)(F.W.O. - Vlaanderen).

M. Gabbrielli and G. Gupta (Eds.): ICLP 2005, LNCS 3668, pp. 435–436, 2005.

Acknowledgements

I would like to thank my supervisor Bart Demoen, the jury members of my doctoral committee and all my co-authors for their help and support.

References

1. Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unifed Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252, Los Angeles, California, 1977. ACM Press.
2. Bart Demoen. hProlog. http://www.cs.kuleuven.ac.be/ bmd/hProlog/.
3. Thom Frühwirth. Theory and practice of constraint handling rules. *Journal of Logic Programming*, 37(1–3):95–138, October 1998.
4. Tom Schrijvers. *Analyses, Optimizations and Extensions of Constraint Handling Rules*. PhD thesis, K.U.Leuven, Leuven, Belgium, June 2005.
5. Tom Schrijvers and Bart Demoen. The K.U.Leuven CHR system: Implementation and application. In Thom Frühwirth and Marc Meister, editors, *First Workshop on Constraint Handling Rules: Selected Contributions*, pages 1–5, Ulm, Germany, May 2004.
6. Tom Schrijvers, Bart Demoen, Gregory Duck, Peter Stuckey, and Thom Frühwirth. Automatic Implication Checking for CHR Constraints. In Horatiu Cirstea and Narciso Martí-Oliet, editors, *RULE'05: Proceedings of the 6th International Workshop on Rule-Based Programming*, Nara, Japan, April 2005.
7. Tom Schrijvers, Peter Stuckey, and Gregory Duck. Abstract Interpretation for Constraint Handling Rules. In *PPDP'05: Proceedings of the 7th International Symposium on Principles and Practice of Declarative Programming*, Lisbon, Portugal, July 2005. ACM Press.
8. Tom Schrijvers and David S. Warren. Constraint handling rules and tabled execution. In Bart Demoen and Vladimir Lifschitz, editors, *ICLP'04: Proceedings of the 20th International Conference on Logic Programming*, volume 3132 of *Lecture Notes in Computer Science*, pages 120–136, St-Malo, France, September 2004. Springer Verlag.
9. David S. Warren et al. The XSB Programmer's Manual: version 2.7, vols. 1 and 2, January 2005. http://xsb.sf.net.
10. Jan Wielemaker. SWI-Prolog release 5.4.0, 2004. http://www.swi-prolog.org/.

Formalization and Verification of Interaction Protocols

Federico Chesani

DEIS – University of Bologna, viale Risorgimento 2,
40136 Bologna, Italy
fchesani@deis.unibo.it

1 Introduction

In recent years, the study of protocols and their properties has been one of the most investigated issues in distributed and multi-process systems research, and they are indeed one of the key component of Multi-Agent Systems. Several formal languages for defining protocols and properties have been proposed within different research communities. Some of the most common objectives of such languages include the ability to: *formalize the protocols* in an easy and clear way for human users; *define the protocols abstracting away from the internal architecture* of the participating peers; be able to specify and investigate *properties*, and help the *implementation* of the peers.

Most of the current research on protocols falls into one of the following four main areas of interest: *protocol formalization*, where languages for specifying protocol has been intensively studied not only in MAS research [3,5,9], but also in the broader community of distributed and multi-process systems [6]; *standardization*, aimed at guaranteeing interoperability between heterogeneous agents in open computing environment [2,5]; *protocol properties*, where tools for proving properties are of utmost importance in the MAS community [4] and in the security protocols community [1]; and finally *specific application domain protocols*, where argumentation and negotiation are examples of domains where the study of protocols is driven by the need to address specific features [8].

2 Goal and Current Status of the Research

Interaction protocols are a necessary component of open and heterogeneous systems. Definition languages, formal semantics, verification tools and proof of properties are the main issues that must be considered to achieve effective interaction protocols. Logic Programming can greatly contribute to tackle these issues, due to its declarative character, as well as its possibility of automatically proving theorems. My doctoral research programme aims to adopt Logic Programming, and in particular Abduction, for solving these problems. To this end, I envisage to pursue the following research directions:

M. Gabbrielli and G. Gupta (Eds.): ICLP 2005, LNCS 3668, pp. 437–438, 2005.

- study of the *state-of-the-art* for protocol definition languages;
- definition of a framework where concepts like protocol, property, compliance and interaction are defined in a coherent and unified way;
- formulation of a language for protocol specification;
- development and implementation of tools for the verification of interaction and the proof of properties;
- development of a methodology for multi agent systems design.

My research activity on protocols has started within the SOCS project, with its definition of a general model for societies of agents, and of a protocol specification language based on *Social Integrity Constraints* (IC_s). An abductive proof-procedure, called SCIFF, has been developed and proven to be sound and complete with respect to its declarative semantics. Using SCIFF it is possible to verify if a certain interaction is *compliant* with a protocol specified using IC_s [7].

My research activity, in particular, has focussed on studying the automatic translation of other protocol definition languages such as AUML into IC_s, and on studying different methods for extending SCIFF in order to prove protocol properties automatically. In particular, we are considering several SCIFF extensions for generating a proof based on refutation. Up to now, we are able to (dis)prove a property only in certain cases. Next, I intend to investigate completeness so as to able to rely on SCIFF for refutation-based property proving.

References

1. David A. Basin, S. Mödersheim, and L. Viganò. An on-the-fly model-checker for security protocol analysis. In *ESORICS*, pages 253–270, 2003.
2. FIPA: Foundation for Intelligent Physical Agents. `http://www.fipa.org/`
3. N. Fornara and M. Colombetti. Operational specification of a commitment-based agent communication language. In C. Castelfranchi and W. L. Johnson, editors, *Proc. of AAMAS-2002*, pages 535–542. ACM Press.
4. F. Guerin and J. Pitt. Proving properties of open agent systems. In C. Castelfranchi and W. L. Johnson, editors, *Proc. AAMAS-2002*, pages 557–558. ACM Press.
5. M. P. Huget. Agent uml notation for multiagent system design. *Internet Computing, IEEE*, Vol. 8(4):63–71, July-Aug. 2004.
6. K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use.*, vol. 1 of *Mon. in Theor. Computer Science. An EATCS Series.* Springer, 2 edition, X 1997.
7. Alberti M., Chesani F., Gavanelli M., Lamma E., Mello P., and Torroni P., Compliance verification of agent interaction: a logic-based software tool. *Applied Artificial Intelligence*, 2005. To appear.
8. P. J. McBurney. *Rational Interaction.* PhD thesis, University of Liverpool, 2002.
9. P. Yolum and M.P. Singh. Flexible protocol specification and execution: applying event calculus planning using commitments. In C. Castelfranchi and W. L. Johnson, editors, *Proc. AAMAS-2002*, pages 527–534. ACM Press.

$\mathcal{PS}$-LTL for Constraint-Based Security Protocol Analysis

Ricardo Corin, Ari Saptawijaya, and Sandro Etalle

Department of Computer Science, University of Twente, The Netherlands

Abstract. Several formal approaches have been proposed to analyse security protocols, e.g. [2,7,11,1,6,12]. Recently, a great interest has been growing on the use of constraint solving approach. Initially proposed by Millen and Shmatikov [9], this approach allows analysis of a finite number of protocol sessions. Yet, the representation of protocol runs by *symbolic* traces (as opposed to concrete traces) captures the possibility of having unbounded message space, allowing analysis over an infinite state space. A constraint is defined as a pair consisting of a message M and a set of messages K that represents the intruder's knowledge. Millen and Shmatikov present a procedure to *solve* a set of constraints, i.e. that in each constraint, M can be built from K. When a set of constraints is solved, then a concrete trace representing an attack over the protocol can be extracted.

Corin and Etalle [4] has improved the work of Millen and Shmatikov by presenting a more efficient procedure. However, none of these constraint-based systems provide enough flexibility and expresiveness in specifying security properties. For example, to check secrecy an artificial protocol role is added to simulate whether a secret can be learned by an intruder. Authentication cannot also be checked directly. Moreover, only a built-in notion of authentication is implemented by Millen and Shmatikov in his Prolog implementation [10]. This problem motivates our current work.

A logical formalism is considered to be an appropriate solution to improve the flexibility and expresiveness in specifying security properties. A preliminary attempt to use logic for specifying local security properties in a constraint-based setting has been carried out [3]. Inspired by this work and the successful NPATRL [11,8], we currently explores a variant of linear temporal logic (LTL) over finite traces, $\mathcal{PS}$-LTL , standing for pure-past security LTL [5]. In contrast to standard LTL, this logic deals only with past events in a trace. In our current work, a protocol is modelled as in previous works [9,4,3], viz. by protocol roles. A protocol role is a sequence of *send* and *receive* events, together with status events to indicate, e.g. that a protocol role has completed her protocol run. A *scenario* is then used to deal with the number of sessions and protocol roles considered in the analysis.

Integrating $\mathcal{PS}$-LTL into our constraint solving approach presents a challenge, since we need to develop a sound and complete decision procedure against *symbolic* traces, instead of concrete traces. Our idea to address this problem is by concretizing symbolic traces incrementally while deciding a formula. Basically, the decision procedure consists of two steps: *transform* and *decide*. The former step transforms a $\mathcal{PS}$-LTL

M. Gabbrielli and G. Gupta (Eds.): ICLP 2005, LNCS 3668, pp. 439–440, 2005.

formula with respect to the current trace into a so-called *elementary* formula that is built from constraints and equalities using logical connectives and quantifiers. The decision is then performed by the latter step through solving the constraints and checking the equalities.

Although we define a decision procedure for a fragment of $\mathcal{PS}$-LTL , this fragment is expressive enough to specify several security properties, like various notions of secrecy and authentication, and also data freshness. We provide a Prolog implementation and have analysed several security protocols.

There are many directions for improvement. From the implementation point of view, the efficiency of the decision procedure can still be improved. I would also like to investigate the expressiveness of the logic for speficying other security properties. This may result in an extension of the decision procedure for a larger fragment of the logic. Another direction is to characterize the expressivity power of $\mathcal{PS}$-LTL compared to other security requirement languages.

References

1. M. Abadi and A. Gordon. A calculus for cryptographic protocols: The spi calculus. In *Fourth ACM Conference on Computer and Communications Security*, pages 36–47. ACM Press, 1997.
2. M. Burrows, M. Abadi, and R. Needham. A logic of authentication. *ACM Transactions on Computer Systems*, 8(1):18–36, 1990.
3. R. Corin, A. Durante, S. Etalle, and P. H. Hartel. A trace logic for local security properties. In *Int. Workshop on Software Verification and Validation (SVV)*, volume 118, pages 129–143, Mumbai, India, Dec 2003. Elsevier Science in Electronic Notes in Theoretical Computer Science.
4. R. Corin and S. Etalle. An improved constraint-based system for the verification of security protocols. In M. V. Hermenegildo and G. Puebla, editors, *9th Int. Static Analysis Symp. (SAS)*, volume LNCS 2477, pages 326–341, Madrid, Spain, Sep 2002. Springer-Verlag, Berlin.
5. R. Corin, A. Saptawijaya, and S. Etalle. $\mathcal{PS}$-LTL for constraint-based security protocol analysis. `http://www.cs.utwente.nl/~corin/ces05long.ps`, 2005.
6. G. Lowe. Casper: A compiler for the analysis of security protocols. In *Proc. 10th IEEE Computer Security Foundations Workshop (CSFW '97)*, pages 18–30. IEEE, 1997.
7. C. Meadows. The NRL protocol analyzer: An overview. *Journal of Logic Programming*, 26(2):113–131, 1996.
8. C. Meadows, P. Syverson, and I. Cervesato. Formalizing GDOI group key management requirements in NPATRL. In *Proceedings of the ACM Conference on Computer and Communications*, pages 235–244. ACM, November 2001.
9. J. Millen and V. Shmatikov. Constraint solving for bounded-process cryptographic protocol analysis. In *8th ACM Conference on Computer and Communication Security*, pages 166–175. ACM SIGSAC, November 2001.
10. J. K. Millen. CAPSL web site. `http://www.csl.sri.com/~millen/capsl`, 2000.
11. P. Syverson and C. Meadows. A formal language for cryptographic protocol requirements. *Designs, Codes, and Cryptography*, 7(1/2):27–59, 1996.
12. F. J. Thayer, J. Herzog, , and J. D. Guttman. Strand spaces: Proving security protocols correct. *Journal of Computer Security*, 7:191–230, 1999.

Concurrent Methodologies for Global Optimization

Luca Bortolussi

Dept. of Maths and Computer Science, University of Udine,
Via delle Scienze 206, 33100 Udine, Italy
bortolussi@dimi.uniud.it

Introduction. My research interests are mainly focused on concurrent approaches to global optimization problems. Optimization tasks have two main conflicting features: they are both very difficult and central to a lot of the applications computer science faces daily. The problems I'm most interested in stem out from biology, protein folding being the principal one[1].

The protein folding, or protein structure prediction problem, is the task of identifying the characteristic (native) spatial configuration of a protein (a polymeric chain built up from 20 different aminoacids), given the sequence of aminoacids composing it. This problem can be modeled as the search of the configuration of minimum free energy, and even very coarse abstractions of it are known to be NP-hard. Needless to say, it is far from being solved in a satisfactory way, even if there is a huge amount of research on it, due to its enormous importance in biological and pharmaceutical research.

The most striking feature, however, is not its mathematical difficulty, but the fact that Nature is able to fold correctly a protein in a very short time (milliseconds), "exploring" a very small portion of the search space, even if the main forces behind the process (i.e. protein-solvent interaction) have a "stochastic" nature (in the sense of statistical mechanics).[2] In addition, the native configuration is reached simply by the interaction among the atoms constituting the protein. In some sense, it's the concurrent interaction between these "simple" agents, that obey "simple" rules (the laws of physics), that determines both the protein's native structure and the dynamics of the process for reaching it. Therefore, one of the key ingredient that enables Nature to be so efficient could be this total concurrency itself.

Concurrent Simulation of Protein Folding. The previous reflections are the starting point of the attempts we made so far in modelling the protein structure prediction problem in a concurrent setting. In [1], we associated an independent process to each aminoacid, modeled here in a simplified way as a single centre of interaction. These agents interact by exchanging information about their spatial position, using this knowledge to move in the space, trying to reach the configuration of minimum free energy. The moving strategy used is a Monte-Carlo one: moves lowering the energy are always performed, while moves rising the energy are executed with a certain probability depending on the difference of potential. The whole simulation was written in SICStus-LINDA. The results were encouraging, even if the coarseness of the energy model used

[1] Pointers to references describing the problems and the techniques mentioned here can be found in the papers cited in the reference section. They are all available on-line at http:www.dimi.uniud.it/bortolus

[2] This is known as the Leventhal Paradox.

M. Gabbrielli and G. Gupta (Eds.): ICLP 2005, LNCS 3668, pp. 441–443, 2005.

and the slowness of communications between LINDA processes forbade to obtain decent predictions for whole proteins.

To tackle these problems, in [2] we embedded the previous framework in a multi-agent scheme designed for optimization tasks, implementing the whole system both in SICStus-LINDA and in C (in a multi-threading version). In this new model we adopted a different potential, still representing the aminoacids as a single center of interaction, and identifying them with a concurrent process. In addition, we introduced some higher level agents, which have the task of both coordinating the exploration of the state space performed by the aminoacid agents and introducing some form of cooperation among them. This cooperative action helps the aminoacids to form some local patterns, like helices and sheets, that are very common in proteins. The enhancements introduced here improve a lot the results obtained in terms of both the stability of the simulation and the quality of the predictions of the native structures.

A Language for Concurrent Optimization Algorithms. The development of concurrent simulations of protein folding induced us a more general interest in concurrent optimization algorithms. This field is very prolific, and a lot of work on parallel optimization heuristics has been done in the last 20 years. However, this area is also very fragmented: every single family of algorithms (parallel simulated annealing, parallel genetic algorithms, and so on) has its own conferences, and even its own language and metaphors. It's only recently that some unifying approaches have been put forward, like the multi-agent framework MAGMA. Despite this, to our knowledge there is still lacking of proposals of a unifying language in which describing and reasoning about these optimization algorithms in general, and their concurrent features in particular.

We have identified some characterizing features that such a language should have: it should be declarative (this permits to reason easily on it) and it should allow an easy description of the optimization problems, it must be concurrent and capable to express some form of localization, and it should also have probabilistic features. The description of an optimization problem generally consists of both the objective function and its domain, usually expressed by means of constraints on the variables into play. Therefore, a constraint based language seems a reasonable choice. Concurrency can be obtained by the mechanisms of concurrent constraint programming, while localization can be achieved by adding to it some features of distributed languages. Finally, the last request can be fulfilled by extending the language with a probabilistic semantics. Actually, it turned out that a probabilistic and distributed version of CCP has never been developed, so this was the first step in our investigation of general concurrent optimization problems.

The language we have developed [3] casts CCP in a distributed setting, where there is a network of computational resources, each one running CCP programs independently. While local communications follow the rules of CCP and are asynchronous, the exchange of information between different nodes is performed synchronously using a π-calculus scheme. In addition, the language is provided with a probabilistic structural operational semantics, where time is discrete both at the local and at the global level. Currently, we are working on an implementation of the language, and on some straightforward generalizations, like modeling the transitions at the network level as contin-

uous stochastic processes and improving the dynamic reconfiguration of the network topology.

Future Directions. In the future, we plan to develop methodologies for performing some kind of analysis of programs written in this language. One direction can be that of doing a probabilistic model checking of a suitable (probabilistic) abstraction of the semantics of programs. This would enable us to study properties like the average cost of execution, or the average quality of the solutions found. Another interesting approach can be that of studying the programs using techniques proper of statistical mechanics (like ergodicity analysis) or of complex systems analysis.

In the meanwhile, we want also to perfection the concurrent simulation of protein folding, by developing a new, more realistic, energy function and by improving the interaction strategies between the agents. In particular, we aim at defining enhanced strategic coordination and cooperation features, in order to achieve a more efficient exploration of the search space. In addition, we want to encode this optimization algorithm in our new language, in order to test the analysis techniques.

References

1. L. Bortolussi, A. Dal Palù, A. Dovier, and F. Fogolari. Agent-based protein folding simulation. *Intelligenza Artificiale*, January 2005.
2. L. Bortolussi, A. Dovier, and F. Fogolari. Multi-agent simulation of protein folding. In *To be presented at MAS-BIOMED 2005*, 2005.
3. L. Bortolussi and H. Wiklicky. A distriuted and probabilistic concurrent constraint programming language. *To be presented at ICLP 2005*, 2005.

A Temporal Programming Language for Heterogeneous Information Systems

Vitor Nogueira

Departamento de Informática, Universidade de Évora, Portugal
vbn@di.uevora.pt

In the last decades the number of information centers that receive data from different origins or technologies has increased enormously. Representing and reasoning with temporal data is an important issue in the majority of those centers. Nevertheless, most of the solutions to solve such issue are rather limited and specific to a certain domain.

Constraint–based frameworks are widely used to perform temporal reasoning. There is even a specialisation of the Constraint Satisfaction Problem (basically is a tuple $< V, D, C >$, where V is the set of variables, D their domains and C the set of constraints to be satisfied) called Temporal Constraint Satisfaction Problem (TCSP) where variables represent time and constraints stand for sets of allowed temporal relations between the variables. According to the time entity represented by the variables (such as time points, time intervals, durations) and the class of constraints (qualitative, metric or both) different TCSP's are defined. For a survey on this subject see for instance [1].

Contextual Logic Programming [2] (CxLP) is a simple and powerful language that extends logic programming with mechanisms for modularisation. Recent work not only presented a revised specification of CxLP together with a new implementation for it but also explained how this language could be seen as a shift into the Object-Oriented Programming paradigm [3]. Finally, CxLP was shown to be a powerful paradigm in which to design and implement Organizational Information Systems (the real-world application: Universidade de Évora's second generation Academic Information System [4], is an example of such).

From the reasons above one is lead assume that the combination of a constraint framework with CxLP is a promising tool for building a temporal programming language. There is already some preliminary work on that direction [5,6,7] where CxLP is used not only in associating temporal information to the propositions but also to build interesting concepts such as "implicit time" (given a computation with temporal elements, and even if there is no explicit mention of time, it is assumed that all the elements share a "common time"). In this work the constraint language chosen was Constraint Logic Programming since it combines the power of constraint solving with the one of logic programming: the former allows us to do the temporal reasoning and the later to establish a richer connection to the CxLP layer. CLP was also used to represent the temporal elements since a satisfaction problem was regarded as way of defining implicitly a set of time points, i.e. as defining the *basic temporal elements* of the framework.

M. Gabbrielli and G. Gupta (Eds.): ICLP 2005, LNCS 3668, pp. 444–445, 2005.

With this research we endeavour to further develop the framework briefly described in the previous paragraph. We intend to give a formal definition of the temporal data model along with the relation algebra proposed; to show how this framework can represent qualitative and metric constraints; to exemplify that it can work with elements having diferent units; to further explore the use of CxLP paradigm in the temporal reasoning.

In the end its ourl goal to have a full blown language that can work as the *backbone* for constructing and handling of Heterogeneous Information Systems.

References

1. Schwalb, E., Vila, L.: Temporal constraints: A survey. Constraints **3** (1998) 129–149
2. Porto, A., Monteiro, L.: Contextual logic programming. In Levi, G., Martelli, M., eds.: Proceedings 6th Intl. Conference on Logic Programming, Lisbon, Portugal , 19–23 June 1989. The MIT Press, Cambridge, MA (1989) 284–299
3. Abreu, S., Diaz, D.: Objective: in minimum context. In: Proc. Nineteenth International Conference on Logic Programming. (2003)
4. Abreu, S., Diaz, D., Nogueira, V.: Organizational information systems design and implementation with contextual constraint logic programming. In: IT Innovation in a Changing World – The 10^{th} International Conference of European University Information Systems, Ljubljana, Slovenia (2004)
5. Nogueira, V.B., Abreu, S., David, G.: Towards temporal reasoning in constraint contextual logic programming. In: Proceedings of the 3^{rd} International Workshop on Multiparadigm Constraint Programming Languages MultiCPL'04 associated to ICLP'04, Saint–Malo, France (2004)
6. Nogueira, V., Abreu, S., David, G.: Using contextual logic programming for temporal reasoning. In Ernesto Pimentel, Nieves J. Brisaboa, J.G., ed.: Proceedings of the VIII Jornadas de Ingeniería del Software y Bases de Datos, Alicante, Spain (2003) 479–489
7. Nogueira, V., Abreu, S., David, G.: Towards temporal reasoning on isco. In Juan José Moreno-Navarro, J.M.n.C., ed.: Proceedings of the Joint Conference on Declarative Programming APPIA-GULPE-PRODE, Madrid, Spain (2002) 311–324

Nonmonotonic Logic Programs for the Semantic Web*

Roman Schindlauer

Institut für Informationssysteme, Technische Universität Wien,
Favoritenstraße 9-11, A-1040 Vienna, Austria
roman@kr.tuwien.ac.at

Motivation. The next step in the development of the Semantic Web is the realisation of the Rules, Logic, and Proof layers, which will be developed on top of the Ontology layer, and which should offer sophisticated representation and reasoning capabilities. The aim of this project is to investigate methods and techniques how nonmonotonic logic programming can be successfully and efficiently applied for integrating the Rules and the Ontology layer. In particular, it is crucial to allow for building rules on top of ontologies, that is, for rule-based systems that use vocabulary specified in ontology knowledge bases. Another type of combination is to build ontologies on top of rules, which means that ontological definitions are supplemented by rules or imported from rules. Our approach combines both kinds of couplings in one single framework. At the same time, we want to enrich Semantic Web reasoning by nonmonotonic reasoning methods, focussing on those closely related to Answer Set Programming. Among these are the use of defaults, closed-world reasoning and multiple model generation.

Current State. In [3], a combination of logic programs under the answer-set semantics with description logics was proposed, introducing *description logic programs* (or *dl-programs*), which are of the form $KB = (L, P)$, where L is a knowledge base in a description logic (e.g., an OWL ontology), and P is a finite set of description logic rules (or *dl-rules*). Such dl-rules are similar to usual rules in logic programs with negation as failure, but may also contain *queries to* L in their bodies which are given by special atoms (on which possibly default negation may apply).

These queries to L also allow for specifying an input from P, and thus for a *flow of information from* P *to* L, besides the flow of information from L to P, given by any query to L. Hence, dl-programs allow for building rules on top of ontologies, but also (to some extent) building ontologies on top of rules. This is achieved by dynamic update operators through which the extensional part of L can be modified for subjunctive querying. For example, the rule

$$paperArea(P, A) \leftarrow DL[keyword \uplus kw; inArea](P, A)$$

intuitively says that paper P is in area A, if P is in A according to the description logic knowledge base L, where the extensional part of the *keyword* role in L (which is known to influence *inArea*) is augmented by the facts of a binary predicate *kw* from the program. In this way, additional knowledge (gained in the program) can be supplied

* This work was partially supported by the Austrian Science Fund (FWF) under grant P17212-N04, and by the European Commission through the IST NoE REWERSE (IST-2003-506779).

M. Gabbrielli and G. Gupta (Eds.): ICLP 2005, LNCS 3668, pp. 446–447, 2005.

to L before querying. Using this mechanism, also more involved relationships between concepts and/or roles in L can be defined and exploited.

Two basic types of semantics have been defined for dl-programs: A generalization of the answer-set semantics [3] adopts the model-generation paradigm, while the well-founded semantics, which has been generalized for dl-programs in [4], represents a skeptical approximation of the answer-set semantics, i.e., every well-founded consequence of a given ordinary normal program P is contained in every answer set of P.

For both semantics, a prototype system, called NLP-DL, has been implemented (`http://www.kr.tuwien.ac.at/staff/roman/semweblp`), exploiting the two state-of-the-art solvers DLV [6] and RACER [5]. A major issue in this respect is an efficient interfacing between the two reasoning systems, for which we devised special methods [1]. It supports cautious and brave reasoning from answer sets, which enables default and minimal model reasoning on top of ontologies, as discussed in [2].

Open Tasks. In [2], we defined a more general class of logic programs, called HEX-*programs*, which embeds new important features aimed at dealing with practical applications and, in particular, with Semantic Web applications. Firstly, the notion of a dl-atom is generalized to a generic *external atom*, which imports external knowledge of various nature into the context of a logic program. This creates a versatile interface under well-defined semantics to any application that can serve as an external knowledge source; for instance, the atom $\#reach[edge, a](X)$ computes all the reachable nodes in the graph $edge$ from node a, delegating this task to an external computational source. Secondly, the language is extended by *higher-order atoms*, allowing to quantify values over predicate names, and to freely exchange predicate symbols with constant symbols.

The computation of HEX-programs will be carried out by a reasoner, which is based on the current prototype system. Currently, we are researching methods for a refined program evaluation, based on extended notions of stratification and rule dependency, reducing time and space complexity of the current iterative computation.

References

1. T. Eiter, G. Ianni, R. Schindlauer, and H. Tompits: Nonmonotonic Description Logic Programs: Implementation and Experiments. In: *Proc. LPAR 2004*. *LNCS* 3452, (2005), 511–517.
2. T. Eiter, G. Ianni, R. Schindlauer, and H. Tompits: A Uniform Integration of Higher-Order Reasoning and External Evaluations in Answer Set Programming. In: *Proc. IJCAI-05*, Morgan Kaufmann (2005). To appear.
3. T. Eiter, T. Lukasiewicz, R. Schindlauer, and H. Tompits: Combining Answer Set Programming with Description Logics for the Semantic Web. In: *Proc. KR-2004*. (2004), 141–151.
4. T. Eiter, T. Lukasiewicz, R. Schindlauer, and H. Tompits: Well-founded Semantics for Description Logic Programs in the Semantic Web. In: *Proc. RuleML 2004*. *LNCS* 3323, Springer (2004), 81–97.
5. V. Haarslev and R. Möller: RACER System Description. In *Proc. IJCAR-2001*. *LNCS* 2083, Springer (2001), 701–705.
6. N. Leone, *et al*: The DLV System for Knowledge Representation and Reasoning. *ACM Transactions on Computational Logic* (2005). To appear.

ICLP 2005 Doctoral Consortium

SiLCC Is Linear Concurrent Constraint Programming

Rémy Haemmerlé

Project Contraintes, INRIA Rocquencourt, France
Remy.Haemmerle@inria.fr

Introduction. The Concurrent Constraint (CC) [8] languages allow to model a large number of constraint programming systems, form Prolog coroutinings to constraints propagation mechanisms such these implemented in the Finite Domain (FD) solver of GNU-Prolog [2]. These languages are an extension of CLP obtained by an addition of a synchronization primitive based on constraint implication. The Linear Concurrent Constraint (LCC) [9] languages are generalizations of CC languages that consists of considerating constraint systems on Girard's linear logic [5] instead of the classical logic. This new kind of languages offer an unified framework combining constraints with state change and allow us to express the semantics of constraint programming together with concurrency and imperative features such as multiple assignment variables. It is, then, possible to defined solvers as pure libraries written in the same languages as the system.

Objectives. The prime objective of this thesis is to design and implement a LCC language called **SiLCC**, for "SiLCC is Linear Concurrent Constraint programming". The conception of this language should respected imperatives of efficiency and of programming ease. The main idea is to define a rich language combining constraints, concurrence and state change. The second objective is the realization of a completely bootstrapped implementation of a constraint system.

Hence any parts of the system could be modify or extended by any usual programmers. For this purpose we need to develop a kernel as small as possible provided with a robust modules system.

Related Works. This thesis falls under the continuity of previous works of our team. On the one hand there are the theoretical works of Fages, Ruet and Soliman [4] on semantics of LCC languages. Using refined versions of observable, they have extended Saraswat's results to obtain more precise and more general semantics. On the other hand there are the works of Diaz [2] on compilation to native code of LP programs. We plan to use his compilation technology to give efficient behavior to our system. The closest works to ours seem to be the Cabeza's ones [1]. In fact he has tried to develop "an extensible [and] global analysis friendly LP System" (Ciao Prolog). For this purpose he proposed a syntactic module system and designed a functional system in which the most of the code is written in Prolog. Nonetheless he did not provide to his modules system any formal definition and did not try to limit the size of his kernel.

Accomplish Results and Current Status. After one and half a year, the main result of this thesis concerns the module system. We have proposed in [6] a

M. Gabbrielli and G. Gupta (Eds.): ICLP 2005, LNCS 3668, pp. 448–449, 2005.

formal definition of closed syntactic module system. By closed we mean the property that the module system is able to prevent any call to the private predicates of a module from the other modules, in particular through meta-programming predicates. We show that this property necessitates to distinguish the execution of a term (meta-programming predicate `call`) from the execution of a closure (higher order). The system we propose is close to the one of CIAO Prolog in its implementation, but has the advantage of revealing the need for closures, and of positioning them w.r.t. meta-programming predicates in the framework of formal operational semantics. We have also developed a first prototype consisting of a layer, written in Prolog, on the top of GNU Prolog RH [3]. This layer is composed of a preprocessor that converts modular code into non-modular code and a new top-level, that transparently interprets modular programs and is able to dynamically compile and load modules and their dependencies.

Current Status. We are actively working on the bootstrapping of our first prototype. We think, hence, distinguish what must be native in our system from what could be rewritten as pure SiLCC libraries.

Expected Achievements and Open Issues. At the end of this thesis, we plan to have a complete implementation of SiLCC. The language will be based on a small kernel, we call LCC($\mathcal{K}$) combining a basic constraints system, concurrence and imperative features. We expect the language to be efficient and extensible, thanks to its compilation to native code, to its modular architecture and to the expressive power of its kernel. Moreover our team also plan to distribute this system with a large number of libraries such as CHR, constraints solvers, lexing/parsing, prescriptive typer. As future works, we can mention the possible implementation of a more advanced module system such as the Sanella and Wallen's one [7] extended with meta-calls and closures.

References

1. Daniel Cabeza. *An Extensible, Global Analysis Friendly Logic Programming System.* PhD thesis, Universidad Politécnica de Madrid, August 2004.
2. Daniel Diaz and Philipe Codognet. Design and implementation of the GNU Prolog system. *Journal of Functional and Logic Programming*, 6, October 2001.
3. Daniel Diaz and Rémy Haemmerlé. *GNU Prolog RH user's manual*, 1999–2004.
4. François Fages, Paul Ruet, and Sylvain Soliman. Linear concurrent constraint programming: operational and phase semantics. *Information and Computation*, 165(1):14–41, February 2001.
5. Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50(1), 1987.
6. Rémy Haemmerlé and François Fages. Closures are needed for closed module systems. Technical Report RR-5575, INRIA, 2005.
7. D. T. Sannella and L. A. Wallen. a calculus for the construction of modular Prolog programs. *Journal of Logic Programming*, pages 147–177, 1992.
8. Vijay A. Saraswat. *Concurrent constraint programming.* ACM Doctoral Dissertation Awards. MIT Press, 1993.
9. Vijay A. Saraswat and Patrick Lincoln. Higher-order linear concurrent constraint programming. Technical report, Xerox Parc, 1992.

Analysis and Optimization of CHR Programs

Jon Sneyers*

Dept. of Computer Science, K.U.Leuven, Belgium
jon@cs.kuleuven.be

Introduction. Constraint Handling Rules (CHR) [2] is a high-level, powerful, yet relatively simple *"no box"* CLP language, embedded in a host language, commonly Prolog. It is based on multi-headed committed-choice rules. Recent implementations of CHR consist of a compiler which translates a CHR program to host language code, and a run-time system implementing the constraint store. Originally, CHR was designed for rapid prototyping of user-defined constraint solvers. In the early years of CHR limited attention went to optimized compilation. As a consequence, the reference implementation of CHR [4] comprises a general compilation schema, with only a small number of optimizations. Currently, CHR is increasingly used as a general-purpose programming language in a wide range of applications. Therefore, performance becomes more important, and recently, more advanced compilation optimizations have been proposed [3].

Several implementations of CHR exist [10]. The K.U.Leuven CHR system [11] includes a state-of-the-art CHR compiler for popular Prolog systems like SWI-Prolog and XSB. The formulation of the refined operational semantics of CHR [1], and subsequently the formulation of the call-based refined operational semantics [5], have captured the essentials of current implementations on a formal level. Optimizations can now be defined formally and their correctness w.r.t. the operational semantics can be proved.

The main goals of my research are to improve the practical usability of CHR, to allow a more declarative use of CHR, and to compile CHR programs to more efficient host language code. These goals are achieved by developping, implementing, and evaluating analyses and optimizations for CHR programs.

Current Results. In [8], a new optimization called *Guard Simplification* is introduced, which uses reasoning about the refined operational semantics to eliminate parts of rule guards that are entailed by the conjunction of the negations of guards in earlier subrules. The general compilation schema [4] translates every head constraint occurrence to a Prolog clause. The *Occurrence Subsumption* optimization [7] detects occurrences for which the generated clause is redundant. Both optimizations are implemented in the K.U.Leuven CHR compiler. We also extended CHR to allow optional mode and type declarations, which further improve the optimizations. We developed a stronger optimization called *Guard and Continuation Optimization* [6] which unifies and extends the above two optimizations. It is defined formally using a new call-based refined operational semantics

* This work was partly supported by project G.0144.03 funded by the Research Foundation - Flanders (F.W.O.-Vlaanderen).

M. Gabbrielli and G. Gupta (Eds.): ICLP 2005, LNCS 3668, pp. 450–451, 2005.

for *Occurrence Representations*, a more expressive representation of CHR programs. Correctness results have been proved. Performance improvements of 20 to 40% have been measured [6,7].

Ongoing and Future Work. Future work regarding the guard and continuation optimization includes: enhancing the entailment reasoning knowledge base to increase the strength of optimizations; improving the scalability of our approach (improving compilation times); adding support for declarations of intended patterns of initial queries, allowing more accurate analyses and stronger optimizations. More ideas for future work related to the guard and continuation optimizations are given in [6].

In [9], we show that every algorithm can be implemented in CHR with the best known time and space complexity. However, it remains a challenge to implement classical algorithms in a natural and elegant way. My current work focusses on implementing and comparing shortest path algorithms in CHR. Identifying and eliminating performance bottlenecks in these programs is a great inspiration for new optimizations.

References

1. Gregory J. Duck, Peter J. Stuckey, María García de la Banda, and Christian Holzbaur. The Refined Operational Semantics of Constraint Handling Rules. In *Proceedings of the 20th Intl. Conference on Logic Programming (ICLP'04)*, 2004.
2. Thom Frühwirth. Theory and Practice of Constraint Handling Rules. In *Special Issue on CLP, Journal of Logic Programming*, volume 37 (1–3), October 1998.
3. Christian Holzbaur, María García de la Banda, Peter J. Stuckey, and Gregory J. Duck. Optimizing compilation of Constraint Handling Rules in HAL. *Special Issue of Theory and Practice of Logic Programming on CHR*, 5, 2005. To appear.
4. Christian Holzbaur and Thom Frühwirth. A Prolog Constraint Handling Rules Compiler and Runtime System. *Special Issue Journal of Applied Artificial Intelligence on Constraint Handling Rules*, 14(4), April 2000.
5. Tom Schrijvers, Peter Stuckey, and Gregory Duck. Abstract Interpretation for Constraint Handling Rules. In *Proceedings of the 7th Intl. Conference on Principles and Practice of Declarative Programming (PPDP'05)*, Lisbon, Portugal, July 2005.
6. Jon Sneyers, Tom Schrijvers, and Bart Demoen. Guard and Continuation Optimization for Occurrence Representations of CHR. In *21st International Conference on Logic Programming (ICLP'05)*, Sitges, Spain, October 2005.
7. Jon Sneyers, Tom Schrijvers, and Bart Demoen. Guard Reasoning for CHR Optimization. Technical Report CW 411, K.U.Leuven, Dept. CS, May 2005.
8. Jon Sneyers, Tom Schrijvers, and Bart Demoen. Guard simplification in CHR programs. In *19th Workshop on (Constraint) Logic Programming*, Germany, 2005.
9. Jon Sneyers, Tom Schrijvers, and Bart Demoen. The Computational Power and Complexity of Constraint Handling Rules. In *2nd Workshop on Constraint Handling Rules (CHR'05)*, Sitges, Spain, October 2005. Submitted.
10. CHR homepage. http://www.cs.kuleuven.be/~dtai/projects/CHR.
11. K.U.Leuven CHR system. http://www.cs.kuleuven.ac.be/~toms/Research/CHR.

Author Index