Preface

This volume constitutes the pre-conference proceedings for LOPSTR 2007: those papers selected for presentation at the Seventeenth International Symposium on Logic-Based Program Synthesis and Transformation, that will be held in Kongens Lyngby, Denmark, August, 23-24, 2007. Previous LOPSTR symposia were held in Venice (2007 and 1999), London (2005 and 2000), Verona (2004), Uppsala (2003), Madrid (2002), Paphos (2001), Manchester (1998, 1992 and 1991), Leuven (1997), Stockholm (1996), Arhhem (1995), Pisa (1994) and Louvain-la-Neuve (1993).

The aim of the LOPSTR series is to stimulate and promote international research and collaboration on logic-based program development. LOPSTR thus traditionally solicits papers in the areas of: specification, synthesis, verification, transformation, analysis, optimisation, composition, security, reuse, applications and tools, component-based software development, software architectures, agent-based software development and program refinement. LOPSTR has a reputation for being a lively, friendly forum for presenting and discussing work in progress. Formal proceedings are produced only after the symposium, so that authors can incorporate this feedback in the published papers.

We would like to thank all those who submitted contributions to LOPSTR. Overall, we received 30 submissions, each of which received at least three reviews. The committee decided to accept seven full papers for presentation and for inclusion in the final post-conference proceedings. Nine extended abstracts were also selected for presentation, which describe work that was judged to be mature enough for possible publication in the post-conference volume. We would also like to thank Michael Codish for agreeing to give an invited talk and contributing a paper to the proceedings.

I am very grateful to program committee and the reviewers for their invaluable help and expertise. I would like to thank Andrei Voronkov for his excellent EasyChair paper submission and reviewing system; Michael Hanus for his guidance in using EasyChair; and Andreas Matthias for his pdfpages LaTeX package which simplified the production of the pre-conference proceedings. These pre-proceedings were themselves printed in Denmark, and are available from the LOPSTR 2007 homepage, which is `http://www.cs.kent.ac.uk/events/conf/2007/lopstr/`.

LOPSTR 2007 was co-located with SAS 2007 and my warmest thanks go to Christian W. Probst (Local Chair) who was always willing to help in every aspect of the organisation of LOPSTR 2007. Special thanks also go Hanne Riis Nielson, Flemming Nielson (Treasurer), Sebastian Nanz, Terkel K. Tolstrup, Eva Bing, Elsebeth Strøm who, together with Christian, took care of the overall planning and local organisation of LOPSTR 2007.

August 2007 Andy King

Program Committee

Elvira Albert	Universidad Complutense Madrid, Spain
John Gallagher	University of Roskilde, Denmark
Michael Hanus	Christian-Albrechts-Universität zu Kiel, Germany
Jacob Howe	City University, UK
Andy King	(Program Chair) University of Kent, UK
Michael Leuschel	Heinrich-Heine-Universität Düsseldorf, Germany
Mario Ornaghi	Università degli Studi di Milano, Italy
Étienne Payet	Université de La Réunion, France
Alberto Pettorossi	Università di Roma Tor Vergata, Italy
Carla Piazza	Università degli Studi di Udine, Italy
C. R. Ramakrishnan	SUNY Stony Brook, USA
Abhik Roychoudhury	National University of Singapore, Singapore
Peter Schneider-Kamp	RWTH Aachen, Germany
Alexander Serebrenik	(Publicity Chair) Technische Universiteit Eindhoven
Josep Silva	Technical University of Valencia, Spain
Wim Vanhoof	University of Namur, Belgium

Local Organisation

Christian W. Probst	(Local Chair) Technical University of Denmark
Sebastian Nanz	Technical University of Denmark
Terkel K. Tolstrup	Technical University of Denmark
Eva Bing	Technical University of Denmark
Elsebeth Strøm	Technical University of Denmark
Hanne Riis Nielson	Technical University of Denmark
Flemming Nielson	(Treasurer) Technical University of Denmark

Additional Referees

Slim Abdennadher	Armin Biere	Davide Bresolin
Maurice Bruynooghe	Manuel Carro	Alberto Casagrande
Agostino Dovier	Camillo Fiorentini	Andrea Formisano
Ankit Goel	Gopal Gupta	Frank Huch
Lunjin Lu	Salvador Lucas	Fred Mesnard
Eric Monfroy	José Morales	Rafael Navarro
Alessandro Dal Palu'	David Pearce	Maurizio Proietti
Arend Rensink	Jaime Sanchez	Beata Sarna-Starosta
Valerio Senni	Andrew Sentosa	Jaroslav Sevcik
Axel Simon	Jan-Georg Smaus	Fausto Spoto
German Vidal	Marc Voorhoeve	Tao Wang
Jan Martijn van der Werf		

Thursday 23 August

09:00 Proving Termination with (Boolean) Satisfaction (LOPSTR invited talk), *Michael Codish*, Ben-Gurion University of the Negev, page 5

10:00 Termination Analysis of Logic Programs based on Dependency Graph, *Manh Thang Nguyen*, K. U. Leuven, Belgium, *Jürgen Giesl*, RWTH Aachen, Germany, *Peter Schneider-Kamp*, RWTH Aachen, Germany and *Daniel De Schreye*, K. U. Leuven, Belgium, page 12

10:30 Coffee break

11:00 Typed-based Homeomorphic Embedding for Online Termination (Extended Abstract), *Elvira Albert*, Complutense University of Madrid, Spain, *John Gallagher*, Roskilde University, Denmark, *Miguel Gómez-Zamalloa*, Complutense University of Madrid, Spain and *Germán Puebla*, Technical University of Madrid, Spain, page 27

11:30 Improving Efficiency of Prolog Programs by Fully Automated Transformation (Extended Abstract), *Jiří Vyskočil* and *Petr Štěpánek*, Charles University, Czech Republic, page 38

12:00 Towards a normal form for Mercury programs (Extended Abstract), *Wim Vanhoof* and *François Degrave*, University of Namur, Belgium, page 48

12:30 Lunch

14:00 Aggregates for CHR through Program Transformation, *Peter Van Weert*, *Jon Sneyers* and *Bart Demoen*, K. U. Leuven, Belgium, page 57

14:30 Generation of Rule-based Constraint Solvers: Combined Approach, *Slim Abdennadher* and *Ingi Sobhi*, German University in Cairo, Egypt, page 72

15:00 A Scalable Inclusion Constraint Solver Using Unification (Extended Abstract), *Ye Zhang* and *Flemming Nielson*, Technical University of Denmark, page 87

15:30 Excursion and Conference dinner

Friday 24 August

09:00 Hardware-Oriented Program Properties (SAS invited talk)
Alan Mycroft, University of Cambridge, UK

10:00 Annotation Algorithms for Unrestricted Independent AND-Parallelism in Logic Programs, *Amadeo Casas*, University of New Mexico, *Manuel Carro*, Technical University of Madrid, and *Manuel Hermenegildo*, Technical University of Madrid and the University of New Mexico, page 97

10:30 Coffee break

11:00 A Flexible, CLP-based Approach to the Analysis of Object-Oriented Program, *Mario Mendezi*, *Jorge Navas*, University of New Mexico, and *Manuel Hermenegildo*, Technical University of Madrid and the University of New Mexico, page 112

11:30 Preserving Sharing in the Partial Evaluation of Lazy Functional Programs, *Sebastian Fischer*, University of Kiel, Germany, *Josep Silva*, *Salvador Tamarit* and *German Vidal*, Technical University of Valencia, Spain, page 127

12:00 Denotation by Transformation - Towards Obtaining a Denotational Semantics by Transformation to Point-free Style (Extended Abstract), *Bernd Braßel* and *Jan Christiansen*, University of Kiel, Germany, page 136

12:30 Lunch

14:00 Snapshot Generation in a Constructive Object-oriented Modeling Language, *Mauro Ferrari*, Università degli Studi dell'Insubria, Italy, *Camillo Fiorentini*, *Alberto Momigliano* and *Mario Ornaghi*, Università degli Studi di Milano, Italy, page 145

14:30 Symbolic Generation of Optimal Control Policies for Discrete-Time Systems (Extended Abstract), *Michel Sintzoff*, University of Louvain, Belgium, page 160

15:00 Synthesis of Data Views for Communicating Processes (Extended Abstract), *Iman Poernomo*, King's College, London, page 169

15:30 Coffee break

16:00 A Clausal View for Access Control and XPath Query Evaluation (Extended Abstract), *Barbara Fila* and *Siva Anantharaman*, Université d'Orléans, France, page 178

16:30 Action Refinement in Process Algebra and Security Issues (Extended Abstract), *Annalisa Bossi*, Università Ca' Foscari di Venezia, Italy, *Carla Piazza*, Università di Udine, Italy and *Sabina Rossi*, Università Ca' Foscari di Venezia, Italy, page 187

Proving Termination with (Boolean) Satisfaction

Michael Codish*

Department of Computer Science
Ben-Gurion University of the Negev
Beer-Sheva, Israel mcodish@cs.bgu.ac.il

1 Introduction

At some point there was the Davis-Putnam-Logemann-Loveland (DPLL) algorithm [6]. Forty five years later, research on Boolean satisfiability (SAT) is still ceaselessly generating even better SAT solvers capable of handling even larger SAT instances. Remarkably, the majority of these tools still bear the hallmark of the DPLL algorithm. In sync with the availability of progressively stronger SAT solvers is an accumulating number of applications which demonstrate that real world problems can often be solved by encoding them into SAT. When successful, this circumvents the need to redevelop complex search algorithms from scratch.

This presentation is about the application of Boolean SAT solvers to the problem of determining program termination. Proving termination is all about the search for suitable ranking functions. The key idea in this work is to encode the search for particular forms of ranking functions to Boolean statements which are satisfiable if and only if such ranking functions exist. In this way, proving termination can be performed using a state-of-the-art Boolean satisfaction solver.

2 Encoding Lexicographic Path Orders

In [3] we describe a propositional encoding for lexicographic path orders (LPOs) [15, 8] and the corresponding LPO-termination property of term rewrite systems. In brief, a term rewrite system (TRS) is a set of rules of the form $\ell \to r$ where ℓ and r are terms constructed from given sets of symbols and variables. A lexicographic path order is an order $\succ_{lpo}$ on terms, induced from a partial order $>$ on the symbols occurring in the terms (a so-called *precedence*). A term rewrite system is LPO-terminating if and only if there exists a partial order on the symbols such that the induced LPO orients all of the rules in the system. Namely such that $\ell \succ_{lpo} r$ for each rule $\ell \to r$.

There are two variants of LPO-termination: "strict" and "quasi" depending on if we restrict the precedence to be strict or not. Both imply termination of the corresponding term rewrite system. Quasi-LPO-termination is typically the

* Supported by the Frankel Center for Computer Sciences at Ben-Gurion University

harder problem as the search for a non-strict precedence is more extensive than that for a strict precedence. Both of the corresponding decision problems, strict- and quasi- LPO-termination, are decidable and NP complete [16].

We encode an LPO-termination problem to SAT in two steps: first, a *partial order constraint* on the symbols in the system is derived; then this constraint is solved through an encoding to SAT obtained by viewing each symbol as an integer value corresponding to its index in the partial order. Partial order constraints are propositional formula in which the atoms are statements about a partial order on a finite set of symbols and can be seen as an instance of the more general formulae of separation logic (sometimes called difference logic) described in [23].

Consider an example. To orient a rule $not(or(A, B)) \to and(not(A), not(B))$, is reduced to solving the following partial order constraint on the symbols $\{or, and, not\}$:

$$((or > and) \wedge (or > not)) \vee (not > and).$$

We encode each of the three symbols as an integer in two bits, and each atom in the partial order constraint as a comparison on a pair of integers in bit representation. For instance, numbering the bits with subscripts on the symbols, the encoding of the atom $(or > and)$ works out to:

$$(((\underbrace{or_{[2]} \wedge \neg and_{[2]}}_{or_{[2]} > and_{[2]}}) \vee (\underbrace{or_{[2]} \leftrightarrow and_{[2]}}_{or_{[2]} = and_{[2]}} \wedge \underbrace{or_{[1]} \wedge \neg and_{[1]}}_{or_{[1]} > and_{[1]}}))$$

The experimental results presented in [3] are unequivocal. Our SAT based implementation of LPO-termination surpasses in orders of magnitude the performance of previous implementations such as those provided at the time by the termination proving tools TTT [13] and AProVE [12].

3 Encoding Argument Filterings

Lexicographic path orders on their own are too weak for many interesting termination problems and hence are typically combined with more sophisticated termination proving techniques. One of the most popular and powerful such techniques is the *dependency pair* (DP) method [1]. A main advantage is that this allows the application of *argument filterings* which specify parts of terms that should be ignored when comparing terms. It can be viewed like this: given a set of pairs of terms to orient with an LPO, first decide which parts of the terms to filter away and then orient the filtered pairs in an LPO. The argument filtering specifies for each function symbol f if subterms of the form $f(s_1, \ldots, s_n)$ should be *collapsed* to their i^{th} argument; or if some of the argument positions should be filtered away. Filtering terms can simplify considerably the partial order constraints that need be solved to find an LPO. However, argument filterings represent also a severe bottleneck for the automation of dependency pairs, as the search space for argument filterings is enormous (exponential in the sum of the arities of the symbols).

In [5] we introduce a propositional encoding which combines the search for an LPO with the search for an argument filtering. The key idea is to introduce a small number of additional Boolean variables: one for each symbol to indicate if it is collapsed, and one for each argument position of a symbol to indicate if it is filtered. Then the encoding of LPO is enhanced to consider these new variables. So, there exist an argument filtering and an LPO which orient a set of inequalities if and only if the encoding of the inequalities is satisfiable. Moreover, each model of the encoding corresponds to a suitable argument filtering and a suitable LPO which orient the inequalities. Once again experimental results [5] indicate speedups in orders of magnitude.

4 Encoding Recursive Path Orders

In [22] we introduce two additional extensions which together lead to an encoding of the so-called recursive path order with status (RPO). In the first extension, the lexicographic path order is extended to consider the lexicographic extension, not just from left-to-right, but rather with respect to any fixed order. It can be viewed like this: given a permutation for each symbol in a term rewrite system, first reorder the arguments of every subterm as prescribed by the permutation for its root symbol. Then check if the resulting system is LPO-terminating. So, now to orient a set of rules we seek a partial order on the symbols as well as permutations for each symbol. For the encoding, we introduce a small number of additional Boolean variables to represent for each symbol the order its arguments are permuted to. Then the encoding of LPO is enhanced to consider this order (in terms of these new variables). In the second extension, we consider an encoding of the the multiset path order (MPO) [7] where term arguments are compared with the multiset ordering. Also, in this case, with a small number of additional Boolean variables we can model the multiset order in the encoding. For RPO, each symbol in the system is associated with a *status* (one more Boolean variable per symbol in the encoding) indicating if its arguments are to be compared with a multiset extension or with a lexicographic extension modulo some permutation. By now the reader will not be surprised that we simply encode all of the components for RPO to SAT to obtain an implementation using a SAT solver. The results presented in [22] again leave no doubt that encoding to SAT is the way to go.

5 Experimental Results

Throughout this work we have found Prolog a convenient language for expressing the various encodings to SAT. Prototype analyzers were written in SWI Prolog [25] applying the MiniSAT solver [20] through its Prolog interface described in [4]. Subsequently the approach has been integrated within the termination analyzer AProVE [11], using the SAT4J solver [21].

In [22] we report the following results for the various analyses described in this paper. We tested the implementation on all 865 TRSs from the TPDB

[24]. The TPDB is the collection of examples used in the annual *International Termination Competition* [19]. The experiments were run under AProVE on a 2.2 GHz AMD Athlon 64 with a time-out of 60 seconds (as in the *International Termination Competition* [19]).

In the table below, the first two rows compare our SAT-based approach for application of the various path orders to the previous dedicated solvers for path orders in AProVE 1.2 which did not use SAT solving. The last two rows give a similar comparison for the path orders in combination with the dependency pairs method and argument filterings. The columns contain the data for LPO with strict and non-strict precedence (denoted *lpo*/*qlpo*), for LPO with permutations (*lpos*/*qlpos*), for MPO (*mpo*/*qmpo*), and for RPO with status (*rpo*/*qrpo*). For each encoding we give the number of TRSs which could be proved terminating (with the number of time-outs in brackets) and the analysis time (in seconds) for the full collection (including time-outs). For the SAT based implementation, checking the full collection of 865 TRSs for strict-RPO termination with argument filterings requires about 100 seconds. Allowing non-strict orders takes about 3 times longer.

	Solver	*lpo*	*qlpo*	*lpos*	*qlpos*	*mpo*	*qmpo*	*rpo*	*qrpo*
1	SAT-based	**123** (0)	**127** (0)	**141** (0)	**155** (0)	**92** (0)	**98** (0)	**146** (0)	**162** (0)
	(direct)	**31.0**	**44.7**	**26.1**	**40.6**	**49.4**	**74.2**	**50.0**	**85.3**
2	dedicated	**123** (5)	**127**(16)	**141** (6)	**154**(45)	**92** (7)	**98**(31)	**145**(10)	**158** (65)
	(direct)	**334.4**	**1426.3**	**460.4**	**3291.7**	**653.2**	**2669.1**	**908.6**	**4708.2**
3	SAT-based	**357** (0)	**389** (0)	**362** (0)	**395** (2)	**369** (0)	**408** (1)	**375** (0)	**416** (2)
	(arg. filt.)	**79.3**	**199.6**	**69.0**	**261.1**	**110.9**	**267.8**	**108.8**	**331.4**
4	dedicated	**350**(55)	**374**(79)	**355**(57)	**380**(92)	**359**(69)	**391**(82)	**364**(74)	**394**(102)
	(arg. filt.)	**4039.6**	**5469.4**	**4522.8**	**6476.5**	**5169.7**	**5839.5**	**5536.6**	**7186.1**

The table shows that with our SAT encodings, performance improves by orders of magnitude over existing solvers both for direct analysis with path orders and for the combination of path orders and argument filterings in the DP framework. Note that without a time-out, this effect would be intensified. By using SAT, the number of time-outs reduces dramatically from up to 102 to at most 2. The two remaining SAT examples with time-out have function symbols of high arity and can only be shown terminating by further sophisticated termination techniques in addition to RPO. Apart from these two, for SAT, there are only 15 examples that take longer than two seconds and only 3 of these take longer than 10 seconds. The table also shows that the use of RPO instead of LPO increases the proving power substantially, while in the SAT-based setting, runtimes increase only mildly.

6 Other SAT Based Termination Analyses

The first encoding of a termination problem into propositional logic is presented in [17]. The encoding is different than the one we consider and adopts a BDD-based representation. It does not provide competitive results. However, it makes

an important step. Another BDD-based encoding, this one for size-change termination [18], is described in [2]. Here, sets of size change graphs are viewed as partial order constraints, similar to those considered in this paper for term rewrite systems.

In the past year, several additional papers [9, 10, 14, 26] have illustrated the huge potential in applying SAT solvers for other types of termination proving techniques for term rewrite systems. A common theme in all of these works is to represent (finite domain) integer variables as binary numbers in bit representation and to encode arithmetic constraints as Boolean functions on these representations. Results indicate uniformly that the SAT based approach to proving termination is very attractive.

7 Summary

Lexicographic- and multiset- path orders are about lifting a base order on terms to consider the arguments of terms as sequences or as multisets with corresponding lexicographic or multiset orders. We have introduced a new kind of propositional encoding for reasoning about termination of term rewrite systems based on variants of these path orders. Our results have had a direct impact on the design of several major termination analyzers for term rewrite systems.

Of particular and general interest are the encoding techniques which enable to refine a search algorithm to consider a property of interest for all subsets of objects, instead of for the full set of objects; or to check if a property holds when considering a sequence of objects in any order, instead of in the fixed left-to-right order. The common theme is to represent with a small number of additional Boolean variables the large number of cases which need be considered. For the extensions of LPO-termination considered in this work, the additional cost in analysis time is minor in comparison to the increase in the size of the search space.

Acknowledgment. The author has been lucky to work on this research with friends, old and new. Thankyou coauthors of [3], [5], and [22]: Elena Annov, Jürgen Giesl, Vitaly Lagoon, Peter Schneider–Kamp, Peter J. Stuckey, and René Thiemann.

References

1. T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236(1-2):133–178, 2000.
2. M. Codish, V. Lagoon, P. Schachte, and P. J. Stuckey. Size-change termination analysis in k-bits. In P. Sestoft, editor, *Proceedings of the European Symposium on Programming*, volume 3924 of *Lecture Notes in Computer Science*, pages 230–245. Springer, 2006.
3. M. Codish, V. Lagoon, and P. J. Stuckey. Solving partial order constraints for LPO termination. In *Proc. RTA '06*, volume 4098 of *LNCS*, pages 4–18, 2006.

4. M. Codish, V. Lagoon, and P. J. Stuckey. Logic programming with satisfiability. *Theory and Practice of Logic Programming*, 2007. (To appear) `http://arxiv.org/pdf/cs.PL/0702072`.
5. M. Codish, P. Schneider-Kamp, V. Lagoon, R. Thiemann, and J. Giesl. SAT solving for argument filterings. In *Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, volume 4246 of *Lecture Notes in Artificial Intelligence*, pages 30–44. Springer, November 2006.
6. M. Davis, G. Logemann, and D. W. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962.
7. N. Dershowitz. Orderings for term-rewriting systems. *Theoretical Computer Science*, 17:279–301, 1982.
8. N. Dershowitz. Termination of rewriting. *Journal of Symbolic Computation*, 3(1/2):69–116, 1987.
9. J. Endrullis, J. Waldmann, and H. Zantema. Matrix interpretations for proving termination of term rewriting. In U. Furbach and N. Shankar, editors, *Proceedings of the Third International Joint Conference on Automated Reasoning (IJCAR)*, volume 4130 of *Lecture Notes in Computer Science*, pages 574–588. Springer, 2006.
10. C. Fuhs, J. Giesl, A. Middeldorp, P. Schneider-Kamp, R. Thiemann, and H. Zankl. SAT solving for termination analysis with polynomial interpretations. In *Proceedings of the 10th International Conference on Theory and Applications of Satisfiability Testing (SAT '07)*, volume 4501 of *Lecture Notes in Computer Science*, pages 340–354, 2007.
11. J. Giesl, P. Schneider-Kamp, and R. Thiemann. AProVE 1.2: Automatic termination proofs in the dependency pair framework. In *Proc. IJCAR '06*, volume 4130 of *LNAI*, pages 281–286, 2006.
12. J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke. Automated termination proofs with AProVE. In v. Oostrom, editor, *Proc. RTA '04*, volume 3091 of *LNCS*, pages 210–220, Aachen, Germany, 2004.
13. N. Hirokawa and A. Middeldorp. Tyrolean termination tool. In *Proceedings of the 16th International Conference on Rewriting Techniques and Applications (RTA)*, volume 3467 of *Lecture Notes in Computer Science*, pages 175–184, Nara, Japan, 2005. Springer.
14. D. Hofbauer and J. Waldmann. Termination of string rewriting with matrix interpretations. In *Proceedings of the 17th International Conference on Rewriting Techniques and Applications (RTA)*, volume 4098 of *Lecture Notes in Computer Science*, pages 328–342, 2006.
15. S. Kamin and J.-J. Levy. Two generalizations of the recursive path ordering. Department of Computer Science, University of Illinois, Urbana, IL. Available at `http://www.ens-lyon.fr/LIP/REWRITING/OLD_PUBLICATIONS_ON_TERMINATION` (viewed December 2005), 1980.
16. M. Krishnamoorthy and P. Narendran. On recursive path ordering. *Theoretical Computer Science*, 40:323–328, 1985.
17. M. Kurihara and H. Kondo. Efficient BDD encodings for partial order constraints with application to expert systems in software verification. In *Innovations in Applied Artificial Intelligence, 17th International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems, Proceedings*, volume 3029 of *Lecture Notes in Computer Science*, pages 827–837, Ottawa, Canada, 2004. Springer.
18. 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.

19. C. Marché and H. Zantema. The termination competition. In *Proc. RTA '07*, LNCS, 2007. To appear.
20. MiniSAT solver. `http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat`. Viewed December 2005.
21. SAT4J satisfiability library for `Java`. `http://www.sat4j.org`.
22. P. Schneider-Kamp, R. Thiemann, E. Annov, M. Codish, and J. Giesl. Proving termination using recursive path orders and SAT solving. In F. Wolter, editor, *Proceedings of 6th International Symposium on Frontiers of Combining Systems (FroCoS '07)*, volume 4720 of *Lecture Notes on Artificial Intelligence*, page (to appear). Springer-Verlag, September 2007.
23. M. Talupur, N. Sinha, O. Strichman, and A. Pnueli. Range allocation for separation logic. In *CAV*, pages 148–161, 2004.
24. The termination problem data base. `http://www.lri.fr/~marche/tpdb/`.
25. J. Wielemaker. An overview of the SWI-Prolog programming environment. In F. Mesnard and A. Serebenik, editors, *Proceedings of the 13th International Workshop on Logic Programming Environments*, pages 1–16, Heverlee, Belgium, Dec. 2003. Katholieke Universiteit Leuven. CW 371.
26. H. Zankl and A. Middeldorp. Satisfying KBO constraints. In F. Baader, editor, *Proceedings of the 18th International Conference on Term Rewriting and Applications*, volume 4533 of *Lecture Notes in Computer Science*, pages 389–403, 2007.

Termination Analysis of Logic Programs based on Dependency Graphs

Manh Thang Nguyen[1], Jürgen Giesl[2], Peter Schneider-Kamp[2], and Danny De Schreye[1]

[1] Department of Computer Science, K. U. Leuven, Belgium
{ManhThang.Nguyen, Danny.DeSchreye}@cs.kuleuven.be
[2] LuFG Informatik 2, RWTH Aachen, Germany
{giesl, psk}@informatik.rwth-aachen.de

Abstract. This paper introduces a modular framework for termination analysis of logic programming. To this end, we adapt the notions of dependency pairs and dependency graphs (which were developed for term rewriting) to the logic programming domain. The main idea of the approach is that termination conditions for a program are established based on the decomposition of its dependency graph into its strongly connected components. These conditions can then be analysed separately by possibly different well-founded orders. We propose a constraint-based approach for automating the framework. Then, for example, termination techniques based on polynomial interpretations can be plugged in as a component to generate well-founded orders.

1 Introduction

Termination analysis in logic programming (LP) traditionally aims at proving that a given logic program terminates w.r.t. a specific set of queries. Termination proofs are usually done by finding ranking functions that map the states of the program to a sequence of elements of a well-founded domain such that the sequence is decreasing w.r.t. the well-founded order of the domain. Practically, it is sufficient to consider only the states that are involved in loops of the program.

Techniques in termination analysis of LPs can be divided into two groups: the global versus the local approach [4, 6, 5, 8, 10, 12, 26]. In the global approach, one wants to find only **one ranking function** for all loops [8, 10, 26]. In contrast, techniques in the local approach apply **different ranking functions** for different loops [4, 5, 12]. Some automated techniques in the global approach are based on a constraint-based framework to search for a suitable ranking function. This is done by first generating a set of symbolic constraints from all termination conditions. Then, a constraint solver is used to solve the set of constraints, yielding a suitable ranking function for the proof. In the local approach, most techniques use a given small set of norms, and try to prove that (a combination of) these norms can be applied for the termination proof of the program. It is unclear at this stage whether a search for arbitrary norms in the local approach could also be automated using a constraint-based technique like [10].

While the constraint-based global approach is very suitable for automation, it has some drawbacks. Since it generates the constraints for all termination conditions and solves them at once, it may be very time-consuming, especially for non-terminating programs. This is because the time for solving a set of constraints often increases exponentially with its size. Moreover, if a complex well-founded order is needed for the termination proof (e.g., a lexicographical order), it is often difficult to find such an order using the constraint-based global approach.

Example 1 (ack). Consider a logic program P computing the Ackermann function. We used a variant with a predecessor predicate $p/2$ in order to illustrate how our technique handles local variables. We want to prove termination of this program w.r.t. the set of queries $S = \{ack(t_1, t_2, t_3) \mid t_1$ and t_2 are ground terms, t_3 is an arbitrary term$\}$.

$$p(s(X), X).$$
$$ack(0, X, s(X)).$$
$$ack(X, 0, Z) :- p(X, Y), ack(Y, s(0), Z).$$
$$ack(s(X), s(Y), Z) :- ack(s(X), Y, Z'), ack(X, Z', Z).$$

Proving termination of this example based on the local approach involves two ranking functions: the first one measures the size of the first argument and the other measures the size of the second argument of the predicate $ack/3$. However, with the constraint-based global approach, it is impossible to find a single ranking function for the termination proof (if one is restricted to ranking functions based on polynomial interpretations). As a matter of fact, both tools cTI [25] and Polytool [26, 27] fail to prove termination of this example.

In addition to the local and global approaches which work *directly* on logic programs, there are also several *transformational* approaches which transform logic programs to *term rewrite systems* (TRSs). One of the most recent techniques in this line of work is [31]. However, as demonstrated in [31], it turned out that there remain many LPs whose termination can currently only be proved by tools working with direct approaches. (An example is the "*der*"-program from [9, 26].) On the other hand, there are also many LPs where currently only transformational tools succeed (e.g., the example "*LP/SGST06-shuffle*" from the *Termination Problem Data Base* (TPDB) [32] that is used in the annual *International Competition of Termination Tools* [24]). The present paper tries to solve this problem by porting TRS-techniques so that they can be applied to LPs directly. In this way, we intend to combine the advantages of direct and transformational approaches. Indeed, a first prototypical implementation shows that the new approach of the present paper can handle both the examples "*der*" and "*shuffle*" above as well as other examples that could not be handled by *any* tool up to now (e.g., "*LP/SGST06-snake*" from the TPDB).

More precisely, in this paper we introduce a modular framework for termination analysis of LPs. To this end, the dependency pair technique for termination analysis of TRSs introduced in [1] is adapted to the LP context. With this new technique, termination analysis of programs like Ex. 1 can be done by

decomposing it into several simple sub-problems. Each of them can be solved independently by using any suitable well-founded order.

We also propose a constraint-based approach for automating the approach in which termination techniques based on polynomial interpretations can be plugged in as a component to search for well-founded orders.

The paper is organised as follows. In Sect. 2, we provide some preliminaries. In Sect. 3, we introduce a modular framework for proving termination of LPs based on dependency graphs. In Sect. 4, we present a constraint-based approach to automate the framework. Finally, we end with a conclusion in Sect. 5.

2 Preliminaries

A *quasi-order* on a set S is a reflexive and transitive binary relation $\succsim$ defined on elements of S. In this paper, we use quasi-orders comparing atoms with each other and comparing terms with each other. We define the *associated equivalence relation* $\approx$ as $s \approx t$ iff $s \succsim t$ and $t \succsim s$. A *well-founded order* on S is a transitive relation $\succ$ where there is no infinite sequence $s_0 \succ s_1 \succ \ldots$ with $s_i \in S$. A *reduction pair* $(\succsim, \succ)$ consists of a quasi-order $\succsim$ and a well-founded order $\succ$ that are *compatible* (i.e., $t_1 \succsim t_2 \succ t_3$ implies $t_1 \succ t_3$).[3]

We assume familiarity with standard notions of logic programs. In the paper, P denotes a pure logic program and $Term_P$, $Atom_P$ denote the sets of terms and atoms constructed from P respectively. Given an atom A, $rel(A)$ is the predicate occurring in A. Given two atoms A and B, we denote by $mgu(A, B)$ their most general unifier. A *query* Q is a finite sequence of atoms. We consider termination of P w.r.t. Q using the left-to-right selection rule that is commonly used in implementations of logic programming.[4]

Let 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 $Q \in S$. In this paper, we use ranking functions and reduction pairs built from norms and level mappings [3]. A *norm* is a mapping $\|\cdot\| : Term_P \to \mathbb{N}$. A *level mapping* is a mapping $|\cdot| : Atom_P \to \mathbb{N}$. An *interargument relation* for a predicate p/n is a relation $R_{p/n} = \{p(t_1, \ldots, t_n) \mid t_i \in Term_P \wedge \varphi_p(t_1, \ldots, t_n)\}$, where (1) $\varphi_p(t_1, \ldots, t_n)$ is a formula of an arbitrary boolean combination of inequalities, and (2) each inequality in φ_p is either $s_i \succsim s_j$ or $s_i \succ s_j$, where s_i, s_j are constructed from $t_1, \ldots, t_n$ by applying function symbols of P. $R_{p/n}$ is *valid* iff for every $p(t_1, \ldots, t_n) \in Atom_P$: $P \models p(t_1, \ldots, t_n)$ implies $p(t_1, \ldots, t_n) \in R_{p/n}$. A reduction pair $(\succsim, \succ)$ is *rigid* on a term or an atom A if

[3] In contrast to the definition of "reduction pairs" in term rewriting [21], for the theoretical results in Sect. 3 we do not require $\succsim$ and $\succ$ to be closed under substitutions. But to automate our method, in Sect. 4 we choose relations $\succsim$ and $\succ$ that result from polynomial interpretations and that are closed under substitutions.

[4] By fixing the selection rule, methods for termination analysis can exploit this and become much stronger. This is similar to termination analysis of term rewriting (in particular, when using dependency pairs). Here, termination of *innermost* rewriting is easier to show than termination of full rewriting.

for all substitutions σ, we have $A \approx A\sigma$. A reduction pair $(\succsim, \succ)$ is rigid on a set of terms or atoms if it is rigid on all its elements.

Example 2 (call set, norm, and level mapping for ack). We again regard the program P and the set of queries S in Ex. 1. Then we have $Call(P,S) = S \cup \{\, p(t_1,t_2) \mid t_1$ is a ground term, t_2 is a variable$\,\}$. Consider the reduction pair $(\succsim, \succ)$ which is induced[5] by a norm $\|0\| = 0$, $\|s(t)\| = 1 + \|t\|$, $\|X\| = 0$ for all variables X, and by an associated level mapping $|p(t_1,t_2)| = 0$ and $|ack(t_1,t_2,t_3)| = \|t_1\|$. Thus, we have $s(0) \succ 0$, $ack(s(0),X,Y) \succ ack(0,X,Y)$, and $ack(0,X,Y) \approx ack(0,0,0)$. Note that $(\succsim, \succ)$ is rigid on $Call(P,S)$. An example for a valid interargument relation w.r.t. $(\succsim, \succ)$ is $R_{p/2} = \{p(t_1,t_2) \mid t_1 \succ t_2\}$.

3 Dependency Graphs in Logic Programming

Def. 3 adapts the notion of *dependency pairs* [1] from TRSs to the LP setting.

Definition 3 (dependency triple). *A* dependency triple *is a tuple of three elements $\langle H, I, B\rangle$ in which H and B are atoms and I is a list of atoms. For a logic program P, we define the set $DT(P)$ of all dependency triples as $DT(P) = \{\langle H, I, B\rangle \mid H :\!- I, B, \ldots \in P\}$.*

Given a program, the number of its dependency triples is finite.

Example 4 (dependency triples of ack). Reconsider the program from Ex. 1. The dependency triples $DT(P)$ of the program are:

$$\langle ack(X,0,Z), [\,], p(X,Y)\rangle \tag{1}$$
$$\langle ack(X,0,Z), [p(X,Y)], ack(Y,s(0),Z)\rangle \tag{2}$$
$$\langle ack(s(X),s(Y),Z), [\,], ack(s(X),Y,Z')\rangle \tag{3}$$
$$\langle ack(s(X),s(Y),Z), [ack(s(X),Y,Z')], ack(X,Z',Z)\rangle \tag{4}$$

Now we adapt the notion of the (estimated) *dependency graph* [1] from TRSs to LPs.[6] While "dependency triples" are related to the "binary clauses" of [5], our notion of dependency graphs for LPs is similar to the "atom dependency graph" of [12]. But in contrast to [12], we use dependency graphs to modularize termination proofs such that *several different* reduction pairs can be used in the termination proof of one program.

The nodes of the dependency graph are the dependency triples and there must be an arc from a dependency triple N to a dependency triple M whenever an attempt to solve the "proof goal" N could load to the "proof goal" M. To estimate this, we use the notion of *connectivity*.

[5] So for terms t_1, t_2 we define $t_1 \,{}_{(}\!\succsim_{)}\, t_2$ iff $\|t_1\| \,{}_{(}\!\geq_{)}\, \|t_2\|$ and for atoms A_1, A_2 we define $A_1 \,{}_{(}\!\succsim_{)}\, A_2$ iff $|A_1| \,{}_{(}\!\geq_{)}\, |A_2|$.

[6] Our notion should not be confused with the notion of the "(predicate) dependency graph" from [2, 12, 28] that simply represents the dependencies between different predicate symbols.

Definition 5 (connectivity). *Let $\langle H_1, I_1, B_1\rangle$ and $\langle H_2, I_2, B_2\rangle$ be two dependency triples. $\langle H_1, I_1, B_1\rangle$ is* connectable *to $\langle H_2, I_2, B_2\rangle$ iff B_1 unifies with a renamed apart variant of H_2.*

Example 6 (connectivity for ack's dependency triples). In Ex. 1, dependency triple (2) is connectable to (3) and (4), and both dependency triples (3) and (4) are connectable to all dependency triples (1), (2), (3), and (4).

Definition 7 (dependency graph). *Let DT be a set of dependency triples. The dependency graph associated with DT is a directed graph whose vertices are the dependency triples DT and there is an arc from a vertex N to a vertex M iff N is connectable to M. Let P be a logic program. The dependency graph associated with $DT(P)$ is called the dependency graph of P, denoted as $DG(P)$.*

Example 8 (dependency graph for ack). Fig. 1 shows the dependency graph for the ack-program in Ex. 1.

Now every infinite execution of the program corresponds to a cycle in the dependency graph. In our setting, a set $\mathcal{C} \neq \emptyset$ of dependency triples is called a *cycle* if for all $N, M \in \mathcal{C}$ there is a non-empty path from N to M in the graph which only traverses dependency triples of $\mathcal{C}$. A cycle $\mathcal{C}$ is a *strongly connected component* (SCC) if $\mathcal{C}$ is not a proper subset of another cycle.

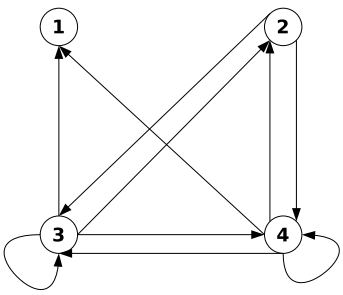


Fig. 1. The dependency graph for the *ack*-program.

Note that in standard graph terminology, a path $N_0 \to N_1 \to \ldots \to N_k$ in a directed graph forms a cycle if $N_0 = N_k$ and $k \geq 1$. In our context we identify cycles with the *set* of elements that occur in it, i.e., we call $\{N_0, N_1, \ldots, N_{k-1}\}$ a cycle, cf. [15]. Since a set never contains multiple occurrences of an element, this results in several cycling paths being identified with the same set. Similarly, an SCC is a graph in standard graph terminology, whereas we identify an SCC with the set of elements occurring in it. Then indeed, SCCs are the same as maximal cycles.

Example 9 (cycles and SCCs for ack). The dependency graph in Fig. 1 has six cycles $\mathcal{C}_1 = \{(3)\}$, $\mathcal{C}_2 = \{(4)\}$, $\mathcal{C}_3 = \{(2), (3)\}$, $\mathcal{C}_4 = \{(2), (4)\}$, $\mathcal{C}_5 = \{(3), (4)\}$, $\mathcal{C}_6 = \{(2), (3), (4)\}$, and one strongly connected component $\mathcal{C}_6 = \{(2), (3), (4)\}$.

Note that each vertex in the dependency graph corresponds to a possible transition from one state to another state in the computational execution of the program. Each loop of the execution corresponds to a cycle in the graph. Intuitively, a program is terminating if there is no cycle in the graph which is traversed infinitely many times.

To use dependency graphs for termination proofs, we proceed as in [1, 16, 19]. The idea is to inspect each SCC of the dependency graph separately and to find a reduction pair $(\succsim, \succ)$ such that *some* dependency triples of the SCC

are *strictly* decreasing (w.r.t. $\succ$) and all others are *weakly* decreasing (w.r.t. $\succsim$). The following definition formalizes when a dependency triple is considered to be "decreasing". It relies on interargument relations for the predicates of the program. Sect. 4 explains how to synthesize such interargument relations and how to find reduction pairs automatically that make dependency triples "decreasing".

Definition 10 (decreasing dependency triples). *Let P be a program. Let $(\succsim, \succ)$ be a reduction pair and $R = \{R_{p_1}, \ldots, R_{p_k}\}$ be a set of interargument relations based on $(\succsim, \succ)$ for the predicates $p_1, \ldots, p_k$ defined in P. Let $N = \langle H, [I_1, \ldots, I_n], B\rangle$ be a dependency triple in $DT(P)$. N is* weakly decreasing *(denoted $(\succsim, R) \models N$) if $H\sigma \succsim B\sigma$ holds for any substitution σ where $(\succsim, \succ)$ is rigid on $H\sigma$ and where $I_1\sigma \in R_{rel(I_1)}, \ldots, I_n\sigma \in R_{rel(I_n)}$. Analogously, N is* strictly decreasing *(denoted $(\succ, R) \models N$) if $H\sigma \succ B\sigma$ holds for any such σ.*

Example 11 (decreasing dependency triples for ack). Consider the reduction pair $(\succsim, \succ)$ from Ex. 2. Let R be the set of valid interargument relations where $R_{ack/3} = \{ack(t_1, t_2, t_3) \mid t_1, t_2, t_3 \in Term_P\}$ and where $R_{p/2}$ is defined as in Ex. 2. Then we have $(\succ, R) \models (2)$. The reason is that for any substitution σ where $(\succsim, \succ)$ is rigid on $ack(X, 0, Z)\sigma$ (i.e., where $X\sigma$ is a ground term) and where $p(X, Y)\sigma \in R_{p/2}$ (i.e., where $X\sigma \succ Y\sigma$), we have $ack(X, 0, Z)\sigma \succ ack(Y, s(0), Z)\sigma$. Similarly, we also have $(\succsim, R) \models (3)$ and $(\succ, R) \models (4)$.

Note that we can restrict ourselves to those SCCs of the dependency graph that can be invoked by calls from $Call(P, S)$. The reason is that only those SCCs can be involved in loops of the execution of the program P, when starting with a query from S. Therefore, we define which SCCs are *reachable* from $Call(P, S)$.

Definition 12 (reachable SCCs). *Let P be a program, S be a set of atomic queries, and $N = \langle H, [I_1, \ldots, I_n], B\rangle$ be a dependency triple. N is* reachable *from $Call(P, S)$ if there is an $A \in Call(P, S)$ such that A unifies with a renamed apart variant of H. An SCC $\mathcal{C}$ in $DG(P)$ is* reachable *from $Call(P, S)$ if there is an $N \in \mathcal{C}$ which is reachable from $Call(P, S)$.*

In the *ack*-example, the only SCC in the dependency graph is reachable from the set $Call(P, S)$ of Ex. 2. But if the *ack*-program contained another clause "$q :\!-\ q$", then the SCC with the resulting dependency triple $\langle q, [\,], q\rangle$ would not be reachable from the call set of Ex. 2. Since it suffices to prove absence of infinite loops only for the *reachable* SCCs, one could then still prove termination of all queries from S. But if one had to regard *all* SCCs, then the termination proof would fail, since the SCC with the dependency triple $\langle q, [\,], q\rangle$ gives rise to an infinite loop. The set of reachable SCCs can easily be (over-)approximated automatically as soon as one has an (over-)approximation of $Call(P, S)$, cf. Sect. 4.

To prove termination, we select an arbitrary reachable SCC $\mathcal{C}$ of the dependency graph. Then, we try to find a reduction pair $(\succsim, \succ)$ such that some dependency triples $\mathcal{C}_\succ \subseteq \mathcal{C}$ are strictly decreasing and all other dependency triples (from $\mathcal{C} \setminus \mathcal{C}_\succ$) are weakly decreasing. This means that the strictly decreasing

dependency triples from $\mathcal{C}_\succ$ can never "occur" *infinitely often* in any execution of the program. Thus, we remove the vertices $\mathcal{C}_\succ$ (and all edges originating or ending in these vertices) from the dependency graph. Afterwards the procedure is repeated (with a possibly different reduction pair). If one finally ends up with a graph without reachable SCCs, then termination of the program is proved.

In this way, our method can use different reduction pairs for different SCCs of the dependency graph. Moreover, one can also use several different reduction pairs in the termination analysis of one single SCC, since SCCs are handled in an incremental way by removing one dependency triple after the other.

However, in our approach we may only use reduction pairs $(\succsim, \succ)$ that are rigid on $\mathit{Call}(P, S)$. This prevents an increase of atoms and terms due to further instantiations in subsequent derivation steps. For details, we refer to [26].

Definition 13 (acceptability). *Let P be a program and S be a set of atomic queries. A subgraph G of the dependency graph $DG(P)$ is called* acceptable *w.r.t. S iff either G has no SCC reachable from $\mathit{Call}(P, S)$ or else, G has such an SCC $\mathcal{C}$ and there is a reduction pair $(\succsim, \succ)$ and a set of valid interargument relations $R = \{R_{p_1}, \ldots, R_{p_k}\}$ based on $(\succsim, \succ)$ for the predicates $p_1, \ldots, p_k$ in P, such that*

- *$(\succsim, \succ)$ is rigid on $\mathit{Call}(P, S)$,*
- *there is a non-empty subset $\mathcal{C}_\succ \subseteq \mathcal{C}$ such that $(\succ, R) \models N$ for all $N \in \mathcal{C}_\succ$ and $(\succsim, R) \models N$ for all $N \in \mathcal{C} \setminus \mathcal{C}_\succ$, and*
- *the graph resulting from G by removing all vertices in $\mathcal{C}_\succ$ is also acceptable.*

Example 14 (termination of ack). The dependency graph of the ack-program in Fig. 1 has only one SCC. First, we select a reduction pair $(\succsim, \succ)$. We re-use the reduction pair from Ex. 2 and the valid interargument relations R from Ex. 11. As shown in Ex. 11, then (2) and (4) are strictly decreasing, whereas (3) is only weakly decreasing. Thus, we remove (2) and (4) from the dependency graph.

The remaining graph has only one vertex (3) and an edge from (3) to itself. Thus, now the only SCC is $\{(3)\}$. We select another reduction pair $(\succsim', \succ')$ which is defined by the same norm $||\cdot||$ as in Ex. 2 and by a new level mapping with $|ack(t_1, t_2, t_3)| = \|t_2\|$. Now we have $(\succ', R) \models (3)$, i.e., (3) can be removed.

The remaining graph is empty and thus, it has no SCC. Hence, termination of the ack-program is proved.

The following theorem states the soundness of our approach.[7]

Theorem 15 (soundness). *A program P is terminating w.r.t. a set of atomic queries S if its dependency graph $DG(P)$ is acceptable w.r.t. S.*

Proof. If P is not terminating w.r.t. S, then there is an $A \in \mathit{Call}(P, S)$, an infinite sequence of (variable renamed) dependency triples $N_0, N_1, \ldots$ with $N_i = \langle H_i, [I_{i1}, \ldots, I_{in_i}], B_i \rangle$, and substitutions $\theta_0, \theta_1, \ldots$ and $\sigma_0, \sigma_1, \ldots$ such that

[7] Note that the proof of Thm. 15 is similar to the one for the dependency pair method in [1]. So in contrast to the "local approaches" [4, 5, 12] for logic programs and the size-change-based methods [23, 29, 33] for other programming paradigms, Thm. 15 does not rely on Ramsey's theorem [6, 30].

- $\theta_0 = mgu(A, H_0)$
- σ_i is a computed answer substitution for the query $(I_{i1}, \ldots, I_{in_i})\theta_i$
- $\theta_{i+1} = mgu(B_i\theta_i\sigma_i, H_{i+1})$

Since there is an edge from N_i to N_{i+1} for all i in the dependency graph, the sequence $N_0, N_1, \ldots$ contains an infinite tail which traverses a cycle of the dependency graph infinitely often.

For any subgraph G of the dependency graph, we show that if this infinite tail is contained in G, then G cannot be acceptable. We use induction on the number of vertices in G. The claim is obviously true if G does not contain any SCC reachable from $Call(P, S)$. Thus, let G contain a reachable SCC $\mathcal{C}$ as in Def. 13. If the infinite tail is still contained in the acceptable subgraph resulting from removing all vertices from $\mathcal{C}_\succ$, the claim follows from the induction hypothesis.

It remains to regard the case where the infinite tail $N_i, N_{i+1}, \ldots$ only traverses dependency triples from $\mathcal{C}$ and where a dependency triple from $\mathcal{C}_\succ$ is traversed infinitely often. Thus, we obtain an infinite sequence

$$\begin{array}{ll}
H_i\theta_i & \approx \text{ (by rigidity, since } H_i\theta_i = B_{i-1}\theta_{i-1}\sigma_{i-1}\theta_i \\
 & \quad \text{and } B_{i-1}\theta_{i-1}\sigma_{i-1} \in Call(P,S)) \\
H_i\theta_i\sigma_i\theta_{i+1} & \succsim \\
B_i\theta_i\sigma_i\theta_{i+1} & = \\
H_{i+1}\theta_{i+1} & \approx \text{ (by rigidity, since } H_{i+1}\theta_{i+1} = B_i\theta_i\sigma_i\theta_{i+1} \\
 & \quad \text{and } B_i\theta_i\sigma_i \in Call(P,S)) \\
H_{i+1}\theta_{i+1}\sigma_{i+1}\theta_{i+2} & \succsim \\
B_{i+1}\theta_{i+1}\sigma_{i+1}\theta_{i+2} & = \\
\ldots &
\end{array}$$

where infinitely many $\succsim$-steps are "strict" (i.e., we can replace infinitely many $\succsim$-steps by "$\succ$"). This is a contradiction to the well-foundedness of $\succ$. □

Thm. 15 can be considered an extension of Thm. 1 in [9], where a strict decrease is required for every (mutually) recursive clause of the program, instead of a decrease on the SCCs as in our theorem above. In particular, Ex. 1 cannot be solved using Thm. 1 of [9].

The converse direction of Thm. 15 does not hold since "acceptability" requires the reduction pair to be rigid on $Call(P, S)$. Hence, the program with the two clauses "$p(X) :- q(X, Y), p(Y)$" and "$q(a, b)$" and the set of queries $S = \{p(X)\}$ from [9] is a counterexample to the completeness direction of Thm. 15.

4 Toward automation

Now we discuss how to automate our approach. In Sect. 4.1, we present a general algorithm to mechanize the technique of Def. 13 and Thm. 15. Then, in Sect. 4.2 we show how to plug in existing approaches for the generation of polynomial interpretations in order to synthesize suitable reduction pairs automatically.

4.1 A general framework

Def. 13 and Thm. 15 provide a method to detect termination of a program P w.r.t. a set of queries S. The method can be automated as follows:

1. Compute the dependency graph $DG(P)$ and remove all vertices which are not reachable from $Call(P, S)$. Decompose the remaining graph into its SCCs.
2. If the set of SCCs is empty, stop with "success" (the program is terminating). Otherwise, select one SCC from the set.
3. If the selected SCC cannot be proved to be acceptable, we stop with "fail" (the program may be non-terminating). If the SCC is acceptable, we delete the strictly decreasing vertices from it and decompose the remaining graph into its SCCs. We add this set of SCCs to the remaining set of SCCs and continue with Step 2.

Step 1 guarantees that all remaining vertices and hence, also all remaining SCCs are reachable from $Call(P, S)$. Therefore, it is obvious that all SCCs decomposed later in Step 3 are also reachable from $Call(P, S)$.

Fig. 2 shows an algorithm based on Step 1-3. In the figure, $reach(G)$ removes all dependency triples from the dependency graph G which are not reachable from $Call(P, S)$, $gcc(G)$ computes the set of SCCs of a graph G, $select(S)$ returns an element selected from the set S, $minus(S_1, S_2)$ returns a set containing all elements that are in the set S_1 but not in S_2, ":=" is the assignment and "=" is the comparison operator. The function $exist(G, O)$ checks if there exists a reduction pair and a set of interargument relations such that G is acceptable. If yes, then the reduction pair is assigned to O. The function $induce(G, O)$ returns a graph which results from G by removing all vertices N where $(\succ, R) \models N$ and their related arcs. Finally, $union(S_1, S_2)$ returns a set that is the union of the sets S_1 and S_2.

Since $Call(P, S)$ can be infinite in general, it is undecidable whether a dependency triple is reachable from $Call(P, S)$. Heuristically, it can be done by first abstracting $Call(P, S)$ to a finite set of call patterns and then checking if there exists a call pattern which unifies with the vertex [26, 27].

The function $exist(G, O)$ is the core of the algorithm. Interestingly, it does not force us to use a fixed type of orders. Therefore,

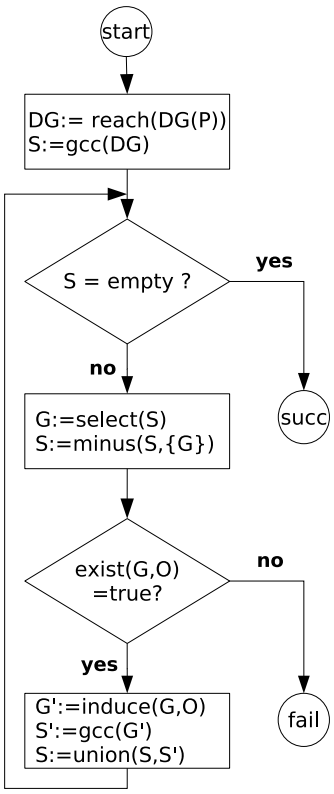


Fig. 2. Our algorithm to verify termination of programs.

the algorithm can be considered a framework where different termination techniques for finding well-founded orders can be plugged in to support the function $exist(G, O)$. In Sect. 4.2, we discuss how the termination analysis technique based on polynomial interpretations from [26, 27] can be applied to the framework.

4.2 Generating well-founded orders

Since arbitrary techniques can be applied to search for reduction pairs required in the function $exist(G, O)$, an obvious option is to use polynomial interpretations, one of the most powerful techniques in termination analysis of logic programming and term rewriting systems [7, 14, 20, 22, 26, 27].[8] The main idea of the technique is to map each function and predicate symbol to a polynomial, under a polynomial interpretation $|\cdot|_I$. The polynomials are considered as functions of type $\mathbb{N} \times \ldots \times \mathbb{N} \to \mathbb{N}$, and the coefficients of the polynomials are also in $\mathbb{N}$. In this way, terms and atoms are mapped to polynomials as well.

Example 16 (polynomial interpretation for ack). The norm and level mapping of Ex. 2 correspond to the polynomial interpretation $|0|_I = 0,\ |s(X)|_I = 1 + X,\ |p(X,Y)|_I = 0,\ |ack(X,Y,Z)|_I = X$. *So we have* $|ack(s(X), s(Y), Z)|_I = |s(X)|_I = 1 + X$ *and* $|ack(X, Z', Z)|_I = |X|_I = X$.

For any polynomial interpretation I, we define a quasi-order $\succsim_I$ on terms and atoms: $t_1 \succsim_I t_2$ iff $|t_1|_I \geq |t_2|_I$ holds for all instantiations of the variables in the polynomials $|t_1|_I$ and $|t_2|_I$ by natural numbers. (It suffices to regard only natural numbers n where $n \geq |c|_I$ for all (constant) function symbols $c/0$ of P.) Similarly, the well-founded order $\succ_I$ is defined as $t_1 \succ_I t_2$ iff $|t_1|_I > |t_2|_I$ holds for all instantiations of the variables in the polynomials $|t_1|_I$ and $|t_2|_I$ by such natural numbers. Obviously, $(\succsim_I, \succ_I)$ is always a reduction pair. Moreover, a term or atom t is rigid w.r.t. $(\succsim_I, \succ_I)$ iff $|t|_I$ contains no variables.

Now, all conditions in Def. 13 can be stated as constraints on polynomials. A reduction pair $(\succsim_I, \succ_I)$ satisfies the conditions in Def. 13 iff the polynomial interpretation $|\cdot|_I$ satisfies the resulting constraints on the polynomials.

Of course, we do not choose a particular polynomial interpretation. Instead, we want to *search* for a suitable one automatically. In the philosophy of the constraint-based approach in [10, 27], we introduce a general symbolic form for the polynomial associated with each predicate and function symbol, and for interargument relations. Since there is no finite symbolic representation for all possible polynomials, we restrict ourselves to fixed types of polynomials. For example, each function and predicate symbol can be associated with a linear polynomial and each interargument relation for a predicate can be expressed in linear form as follows.[9] Here, f_i, p_i^L, and p_i^R are "abstract" symbolic coefficients.

[8] Other possible options would be recursive path orders [11], matrix orders [13], etc.

[9] As already observed for term rewriting, in the vast majority of examples, *linear* polynomial interpretations are already sufficient if they are used in connection with the dependency pair method. But of course, our approach also permits the use of polynomials with higher degree.

In order to complete the termination proof, one has to find suitable instantiations of these coefficients with natural numbers.

- $|f(X_1,\ldots,X_n)|_I = f_0 + \sum_{i=1}^{n} f_i X_i$,
- $R_{p/n} = \{\, p(t_1,\ldots,t_n) \mid p_0^L + \sum_{i=1}^{n} p_i^L |t_i|_I \;\geq\; p_0^R + \sum_{i=1}^{n} p_i^R |t_i|_I \,\}$.

Based on the symbolic forms for polynomial interpretations and interargument relations, all termination conditions expressed in Def. 13 can also be reformulated symbolically. Specifically, the conditions for the function $\mathit{exist}(G,O)$ (which checks whether G is acceptable) are expressed as a set of polynomial constraints with symbolic coefficients (e.g. $f_i, p_i^L, p_i^R, \ldots$). The central question is how to search for an instantiation of these symbolic coefficients such that the set of constraints is satisfied. In [27], we introduced a transformational approach to transform all constraints into a sufficient set of Diophantine constraints on natural numbers where all unknown symbolic coefficients become variables (cf. also [20]). A solution for the Diophantine constraints gives a suitable reduction pair $(\succsim_I, \succ_I)$ and a set of valid interargument relations based on the reduction pair. Finding such a solution can be done by using any available Diophantine constraint solver, e.g. [7, 14]. Finally, the rigidity condition can be symbolised based on the *rigid type graph*. For more details, we refer to [26, 27].

Example 17 (symbolic termination conditions for ack). Reconsider Ex. 1. We define an "abstract" symbolic polynomial interpretation as $|0|_I = c$, $|s(X)|_I = s_0 + s_1 X$, $|p(X,Y)|_I = p_0 + p_1 X + p_2 Y$, $|ack(X,Y,Z)|_I = a_0 + a_1 X + a_2 Y + a_3 Z$, and a set of interargument relations $R = \{R_{p/2}, R_{ack/3}\}$ with

$$\begin{array}{lll} R_{p/2} & = \{p(t_1,t_2) & \mid \; p_0^L + p_1^L|t_1|_I + p_2^L|t_2|_I \;\geq \\ & & \quad\; p_0^R + p_1^R|t_1|_I + p_2^R|t_2|_I \quad \} \\ R_{ack/3} & = \{ack(t_1,t_2,t_3) & \mid \; a_0^L + a_1^L|t_1|_I + a_2^L|t_2|_I + a_3^L|t_3|_I \;\geq \\ & & \quad\; a_0^R + a_1^R|t_1|_I + a_2^R|t_2|_I + a_3^R|t_3|_I \quad \}. \end{array}$$

The conditions for acceptability of the dependency graph can be reformulated as follows:

1. For any dependency triple $N \in \{(2),(3),(4)\}$, we require $(\succsim_I, R) \models N$:

$$\begin{array}{l} \forall X,Y,Z\,[\quad p_0^L + p_1^L X + p_2^L Y \;\geq\; p_0^R + p_1^R X + p_2^R Y \\ \qquad\qquad \Rightarrow a_0 + a_1 X + a_2 c + a_3 Z \;\geq\; a_0 + a_1 Y + a_2(s_0 + s_1 c) + a_3 Z\,] \quad \wedge \\[2ex] \forall X,Y,Z,Z'\,[a_0 + a_1(s_0 + s_1 X) + a_2(s_0 + s_1 Y) + a_3 Z \;\geq \\ \qquad\qquad a_0 + a_1(s_0 + s_1 X) + a_2 Y + a_3 Z'\,] \qquad\qquad\qquad\qquad \wedge \\[2ex] \forall X,Y,Z,Z'\,[\quad a_0^L + a_1^L(s_0 + s_1 X) + a_2^L Y + a_3^L Z' \;\geq \\ \qquad\qquad a_0^R + a_1^R(s_0 + s_1 X) + a_2^R Y + a_3^R Z' \\ \qquad\qquad \Rightarrow a_0 + a_1(s_0 + s_1 X) + a_2(s_0 + s_1 Y) + a_3 Z \;\geq \\ \qquad\qquad a_0 + a_1 X + a_2 Z' + a_3 Z\,] \end{array}$$

2. There exists some dependency triple $N \in \{(2),(3),(4)\}$ with $(\succ_I, R) \models N$:

$$\forall X,Y,Z\,[\quad p_0^L + p_1^L X + p_2^L Y \;\geq\; p_0^R + p_1^R X + p_2^R Y$$
$$\Rightarrow a_0 + a_1 X + a_2 c + a_3 Z \;>\; a_0 + a_1 Y + a_2(s_0 + s_1 c) + a_3 Z\,] \quad \vee$$

$$\forall X,Y,Z,Z'\,[\,a_0 + a_1(s_0 + s_1 X) + a_2(s_0 + s_1 Y) + a_3 Z \;>$$
$$a_0 + a_1(s_0 + s_1 X) + a_2 Y + a_3 Z'\,] \quad \vee$$

$$\forall X,Y,Z,Z'\,[\quad a_0^L + a_1^L(s_0 + s_1 X) + a_2^L Y + a_3^L Z' \;\geq$$
$$a_0^R + a_1^R(s_0 + s_1 X) + a_2^R Y + a_3^R Z'$$
$$\Rightarrow a_0 + a_1(s_0 + s_1 X) + a_2(s_0 + s_1 Y) + a_3 Z \;>$$
$$a_0 + a_1 X + a_2 Z' + a_3 Z\,]$$

3. The valid interargument condition for $p/2$:

$$\forall X\,[\,p_0^L + p_1^L(s_0 + s_1 X) + p_2^L X \;\geq\; p_0^R + p_1^R(s_0 + s_1 X) + p_2^R X\,]$$

4. The valid interargument condition for $ack/3$:

$$\forall X\,[\,a_0^L + a_1^L c + a_2^L X + a_3^L(s_0 + s_1 X) \;\geq\; a_0^R + a_1^R c + a_2^R X + a_3^R(s_0 + s_1 X)\,] \quad \wedge$$

$$\forall X,Y,Z\,[\quad p_0^L + p_1^L X + p_2^L Y \;\geq\; p_0^R + p_1^R X + p_2^R Y$$
$$\wedge\;\; a_0^L + a_1^L Y + a_2^L(s_0 + s_1 c) + a_3^L Z \;\geq$$
$$a_0^R + a_1^R Y + a_2^R(s_0 + s_1 c) + a_3^R Z$$
$$\Rightarrow a_0^L + a_1^L X + a_2^L c + a_3^L Z \;\geq\; a_0^R + a_1^R X + a_2^R c + a_3^R Z\,] \quad \wedge$$

$$\forall X,Y,Z,Z'\,[\quad a_0^L + a_1^L(s_0 + s_1 X) + a_2^L Y + a_3^L Z' \;\geq$$
$$a_0^R + a_1^R(s_0 + s_1 X) + a_2^R Y + a_3^R Z'$$
$$\wedge\;\; a_0^L + a_1^L X + a_2^L Z' + a_3^L Z \;\geq$$
$$a_0^R + a_1^R X + a_2^R Z' + a_3^R Z$$
$$\Rightarrow a_0^L + a_1^L(s_0 + s_1 X) + a_2^L(s_0 + s_1 Y) + a_3^L Z \;\geq$$
$$a_0^R + a_1^R(s_0 + s_1 X) + a_2^R(s_0 + s_1 Y) + a_3^R Z\,]$$

5. The rigidity property for $Call(P,S) = \{ack(t_1,t_2,t_3) \mid t_1$ and t_2 are ground terms, t_3 is an arbitrary term$\} \cup \{p(t_1,t_2) \mid t_1$ is a ground term, t_2 is a variable$\}$:

$$p_2 = 0 \wedge a_3 = 0$$

All the constraints above are satisfied by the following instantiation of the symbolic variables: $c = 0$, $s_0 = s_1 = 1$, $p_0 = p_1 = p_2 = 0$, $a_0 = 0$, $a_1 = 1$, $a_2 = a_3 = 0$, $p_0^L = 0$, $p_1^L = 1$, $p_2^L = 0$, $p_0^R = p_1^R = 1$, $p_2^R = 0$ and $a_i^L = a_i^R = 0$ for all $i \in \{0,1,2,3\}$. This instantiation turns the abstract polynomial interpretation of Ex. 17 into the concrete polynomial interpretation of Ex. 16 (i.e., now it corresponds to the norm and level mapping of Ex. 2). Similarly, the "abstract" interargument relations of of Ex. 17 are turned into the concrete interargument relations of Ex. 2 and Ex. 11 (i.e., $R_{p/2} = \{p(t_1,t_2) \mid t_1 \succ_I t_2\}$ and $R_{ack/3} = \{ack(t_1,t_2,t_3) \mid t_1,t_2,t_3 \in Term_P\}$).

So instead of fixing a polynomial interpretation and interargument relations *before* performing the termination proof, now we only fix the degree of the polynomials used in the polynomial interpretation (e.g., linear or quadratic ones). Then we can automatically generate symbolic constraints and try to solve them afterwards. In this way, suitable polynomial interpretations and interargument relations can be synthesized fully automatically.

5 Conclusion

We have introduced a new framework for termination analysis of LPs based on dependency triples and dependency graphs. Although the notion of dependency pairs and dependency graphs is very popular in the domain of termination analysis of TRS [1, 15, 16, 18, 19], this is the first time that it is applied for LP termination analysis directly. Our contribution is twofold: **(1)** it results in a weaker condition for verifying termination of LPs, where the decrease condition is established for the strongly connected components of the dependency graph, instead of at the clause level as it has been done before; **(2)** it introduces a modular approach in which termination conditions can be separated into different groups, each of which can be treated independently by automatically searching for different suitable well-founded orderings.

A difference between the dependency pair approach for TRSs and our approach is that instead of separating between defined symbols and constructors as for TRSs, we separate between predicate and function symbols of the LP. Another main difference is that in the dependency pair method for TRSs, one requires a weak decrease for the rules of the TRS in order to take the effect of "nested" functions in recursive arguments into account. In the LP-context, these nested functions correspond to body atoms preceding recursive calls. We store these atoms in an additional component of the dependency pair (yielding dependency triples) and take their effect into account by considering interargument relations.

The authors of this paper were involved in the implementation of two of the most powerful automated termination analysers for LPs (Polytool which follows the approach of [26, 27] and AProVE [17] which transforms LPs to TRSs and then tries to prove termination of the resulting TRS [31].) AProVE was the most successful termination prover for logic programs, functional programs, and term rewrite systems in all annual *International Competitions of Termination Tools* 2004 - 2007 [24], where Polytool obtained a close second place for logic programs in the 2007 competition. As mentioned in [31], there exist many LPs where termination can currently only be proved by transformational tools like AProVE and there are also many examples where the termination proof only succeeds with direct tools like Polytool, cf. Sect. 1. Our current work intends to combine the advantages of both approaches by adapting TRS-techniques like dependency pairs to direct termination approaches for LPs. While the present paper only adapted basic concepts of the dependency pair method to the LP setting, in the future we will also try to adapt further more sophisticated "dependency pair processors" [16, 18] as well.

Currently, we are working on an implementation of the results of this paper within Polytool. Here, we try to re-use algorithms from the dependency pair implementation of AProVE. As mentioned in Sect. 1, a first prototypical implementation already shows that in this way one can handle (a) examples that could up to now only be solved with direct tools such as [26, "*der*"], (b) examples that could up to now only be solved with transformational tools based on dependency pairs such as [32, "*LP/SGST06-shuffle*"], as well as (c) exam-

ples like [32, "*LP/SGST06-snake*"] that could not be solved by any tool up to now. Note that the Diophantine constraints resulting from our new approach according to Sect. 4 are usually smaller and simpler than the ones generated by the previous version of Polytool [26, 27]. But already in the previous version of Polytool, solving these constraints automatically was no problem in practice. (To this end, the SAT-based constraint solver of AProVE was used [14].) Thus, this solver will also be used for the automatic generation of the required polynomial interpretations and interargument relations in our new approach.

6 Acknowledgement

We are grateful to the referees for many helpful suggestions. Manh Thang Nguyen is supported by *FWO/2006/09: Termination analysis: Crossing paradigm borders*. Peter Schneider-Kamp and Jürgen Giesl are supported by the *Deutsche Forschungsgemeinschaft (DFG)*, grant *GI 274/5-1*.

References

1. T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236(1-2):133–178, 2000.
2. R. N. Bol, K. R. Apt, and J. W. Klop. An analysis of loop checking mechanisms for logic programs. *Theoretical Computer Science*, 86(1):35–79, 1991.
3. A. Bossi, N. Cocco, and M. Fabris. Norms on terms and their use in proving universal termination of a logic program. *Theoretical Computer Science*, 124(2):297–328, 1994.
4. M. Bruynooghe, M. Codish, J. P. Gallagher, S. Genaim, and W. Vanhoof. Termination analysis of logic programs through combination of type-based norms. *ACM Transactions on Programming Languages and Systems*, 29(2), 2007.
5. M. Codish and C. Taboch. A semantic basis for the termination analysis of logic programs. *Journal of Logic Programming*, 41(1):103–123, 1999.
6. M. Codish and S. Genaim. Proving termination one loop at a time. In *Proc. WLPE '03*, 2003.
7. E. Contejean, C. Marché, A. P. Tomás, and X. Urbain. Mechanically proving termination using polynomial interpretations. *Journal of Automated Reasoning*, 34(4):325–363, 2005.
8. D. De Schreye, K. Verschaetse, and M. Bruynooghe. A framework for analyzing the termination of definite logic programs with respect to call patterns. In *Proc. FGCS '92*, pages 481–488, 1992.
9. D. De Schreye and A. Serebrenik. Acceptability with general orderings. In *Computational Logic: Logic Programming and Beyond*, LNCS 2407, pages 187–210. 2002.
10. S. Decorte, D. De Schreye, and H. Vandecasteele. Constraint-based automatic termination analysis of logic programs. *ACM Transactions on Programming Languages and Systems*, 21(6):1137–1195, 1999.
11. N. Dershowitz. Termination of rewriting. *Journal of Symbolic Computation*, 3(1-2):69–116, 1987.
12. 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.

13. J. Endrullis, J. Waldmann, and H. Zantema. Matrix interpretations for proving termination of term rewriting. In *Proc. IJCAR '06*, LNAI 4130, pages 574–588, 2006.
14. C. Fuhs, J. Giesl, A. Middeldorp, P. Schneider-Kamp, R. Thiemann, and H. Zankl. SAT solving for termination analysis with polynomial interpretations. In *Proc. SAT '07*, LNCS 4501, pages 340–354, 2007.
15. J. Giesl, T. Arts, and E. Ohlebusch. Modular termination proofs for rewriting using dependency pairs. *Journal of Symbolic Computation*, 34(1):21–58, 2002.
16. J. Giesl, R. Thiemann, and P. Schneider-Kamp. The dependency pair framework: Combining techniques for automated termination proofs. In *Proc. LPAR '04*, LNAI 3452, pages 301–331, 2005.
17. J. Giesl, P. Schneider-Kamp, and R. Thiemann. AProVE 1.2: Automatic termination proofs in the dependency pair framework. In *Proc. IJCAR '06*, LNAI 4130, pages 281–286, 2006.
18. J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke. Mechanizing and improving dependency pairs. *Journal of Automated Reasoning*, 37(3):155–203, 2006.
19. N. Hirokawa and A. Middeldorp. Automating the dependency pair method. *Information and Computation*, 199(1-2):172–199, 2005.
20. H. Hong and D. Jakuš. Testing positiveness of polynomials. *Journal of Automated Reasoning*, 21(1):23–38, 1998.
21. K. Kusakari, M. Nakamura, and Y. Toyama. Argument filtering transformation. In *Proc. PPDP '99*, pages 48–62, 1999. LNCS 1702.
22. D. S. Lankford. On proving term rewriting systems are Noetherian. Technical Report MTP-3, Louisiana Technical University, Ruston, LA, USA, 1979.
23. C. S. Lee, N. D. Jones, and A. M. Ben-Amram. The size-change principle for program termination. In *Proc. POPL '01*, pages 81–92, 2001.
24. C. Marché and H. Zantema. The termination competition. In *Proc. RTA '07*, LNCS 4533, pages 303–313, 2007. See also the website `http://www.lri.fr/~marche/termination-competition`.
25. F. Mesnard and R. Bagnara. cTI: A constraint-based termination inference tool for ISO-Prolog. *Theory and Practice of Logic Programming*, 5(1, 2):243–257, 2005.
26. M. T. Nguyen and D. De Schreye. Polynomial interpretations as a basis for termination analysis of logic programs. In *Proc. ICLP '05*, LNCS 3668, pages 311–325, 2005.
27. M. T. Nguyen and D. De Schreye. Polytool: Proving termination automatically based on polynomial interpretations. In *Proc. LOPSTR '06*, LNCS 4407, pages 210–218, 2007. Extended version appeared as Technical report, Department of Computer Science, K. U. Leuven, Belgium.
28. L. Plümer. *Termination Proofs for Logic Programs*. Springer-Verlag, 1990.
29. A. Podelski and A. Rybalchenko. Transition invariants. In *Proc. LICS '04*, pages 32–41, 2004.
30. F. P. Ramsey. On a problem of formal logic. *Proc. London Math. Society*, 30:264–286, 1930.
31. P. Schneider-Kamp, J. Giesl, A. Serebrenik, and R. Thiemann. Automated termination analysis for logic programs by term rewriting. In *Proc. LOPSTR '06*, LNCS 4407, pages 177–193, 2007.
32. The termination problem data base. `http://www.lri.fr/~marche/tpdb`.
33. R. Thiemann and J. Giesl. The size-change principle and dependency pairs for termination of term rewriting. *Applicable Algebra in Engineering, Communication and Computing*, 16(4):229–270, 2005.

Typed-based Homeomorphic Embedding for Online Termination

Elvira Albert[1], John Gallagher[2], Miguel Gómez-Zamalloa[1], and Germán Puebla[3]

[1] DSIC, Complutense University of Madrid, E-28040 Madrid, Spain
[2] Computer Science, Roskilde University, DK-4000 Roskilde, Denmark
[3] CLIP, Technical University of Madrid, E-28660 Boadilla del Monte, Madrid, Spain

Abstract. We introduce the *type-based homeomorphic embedding* relation as an extension of the standard, untyped homeomorphic embedding which allows us to obtain more precise results in the presence of infinite signatures (e.g., the integers). In particular, we show how our type-based relation can be used to improve the accuracy of *online partial evaluation*. For this purpose, we outline an approach to constructing suitable types for partial evaluation automatically, given an untyped program and a goal or set of goals. Our approach is based on existing analysis tools for constraint logic programs: (i) inference of a well-typing of a program and goal, and (ii) bounds analysis for numerical values. We argue that our work improves the state of the practice of online termination and it is very relevant for instance in the context of the specialization of interpreters.

1 Introduction

The *homeomorphic embedding* (HEm) relation [10–12] has become very popular to ensure online termination of *symbolic* transformation and specialization methods and it is essential to obtain powerful optimizations, for instance, in the context of online Partial Evaluation (PE) [9]. Intuitively, HEm is a structural ordering under which an expression t_1 *embeds* expression t_2, written as $t_2 \trianglelefteq t_1$, if t_2 can be obtained from t_1 by deleting some operators, e.g., $\underline{\mathtt{s}}(\mathtt{s}(\underline{\mathtt{U}}+\mathtt{W})\times(\underline{\mathtt{U}}+\mathtt{s}(\underline{\mathtt{V}})))$ embeds $\mathtt{s}(\mathtt{U}\times(\mathtt{U}+\mathtt{V}))$.

The HEm relation can be used to guarantee termination because, assuming that the set of constants and functors is finite, every infinite sequence of expressions $t_1, t_2, \ldots$, contains at least a pair of elements t_i and t_j with $i < j$ s.t. $t_i \trianglelefteq t_j$. Therefore, when iteratively computing a sequence $t_1, t_2, \ldots, t_n$, finiteness of the sequence can be guaranteed by using HEm as a "whistle". Whenever a new expression t_{n+1} is to be added to the sequence, we first check whether $t_i \not\trianglelefteq t_{n+1}$ for all i s.t. $1 \leq i \leq n$. If that is the case, finiteness is guaranteed and computation can proceed. Otherwise, HEm is not capable of guaranteeing finiteness and the computation has to be stopped. The intuition is that computation can proceed as long as the new expression is not larger than any of the previously computed ones since that is a sign of potential non-termination. The success of HEm is due to the fact that sequences can usually grow considerably large before the whistle blows, when compared to other online approaches to guaranteeing termination.

While HEm has been proved very powerful for symbolic computations, some difficulties remain in the presence of infinite signatures such as the numbers. In the case of logic programs, infinite signatures appear as soon as certain Prolog built-ins

such `is/2`, `functor/3` and `name/2` are used. HEm relations over infinite signatures have been defined (e.g. [11, 2]), but they tend to be too conservative in practice ("whistling" too early).

A starting point of our work is the observation that, even if an expression is defined over an infinite signature, it might then only take a finite set of values over such domain for each computation. In this paper, we introduce the *type-based homeomorphic embedding* (TbHEm) relation on typed atoms and typed terms, which by taking context information into account provides more precise results in the presence of infinite signatures. For this, our typed relation is defined on types structured into a (possibly empty) finite part and a (possibly empty) infinite partition. Intuitively, TbHEm allows expanding sequences as long as, whenever we compare sub-terms from an infinite type, the concrete values which appear in the expression remain within the finite part of the type.

The benefits of TbHEm are illustrated in the context of online Partial Evaluation (PE) [9]. In particular, we use a simplified bytecode interpreter in Prolog whose specialization (if successful) allows decompiling simple bytecode programs to Prolog. For the interpreter, we show how to automatically construct typings which are appropriate to be combined with TbHEm. They are inferred by relying on existing analysis techniques, namely on the inference of well-typings [5]. Moreover, we outline how analysis of numeric bounds can also be used to infer useful information for TbHEm. Such analysis makes over-approximations of the set of values that the program arguments can have. Intuitively, when we can prove that such set of values is bounded, then we know that the infinite partition of the type is empty and, hence, we can safely apply traditional HEm (and improve the effectiveness of PE). Although further experimentation is required, we believe that the examples we present already show the benefits of our approach for the specialization of logic programs with infinite signatures.

2 Embedding in Partial Evaluation with Infinite Signatures

This section intends to illustrate the challenges that infinite signatures pose to online termination based on HEm. For the sake of concreteness, we present our ideas in the context of online PE, but they can be also applied to other online transformation and specialization methods (see [11]). We start by recalling the definition of HEm, which can be found for instance in Leuschel's work [13].

Definition 1 ($\trianglelefteq$). *Given two atoms $A = p(t_1, \ldots, t_n)$ and $B = p(s_1, \ldots, s_n)$, we say that A is embedded by B, written $A \trianglelefteq B$, if $t_i \trianglelefteq s_i$ for all i s.t. $1 \leq i \leq n$. The embedding relation over terms, also written $\trianglelefteq$ is defined by the following rules:*

1. *$Y \trianglelefteq X$ for all variables X, Y.*
2. *$s \trianglelefteq f(t_1, \ldots, t_n)$ if $s \trianglelefteq t_i$ for some i.*
3. *$f(s_1, \ldots, s_n) \trianglelefteq f(t_1, \ldots, t_n)$ if $s_i \trianglelefteq t_i$ for all i, $1 \leq i \leq n$.*

Online PE [9] is a semantics-based program transformation technique which specializes a program w.r.t. a given input data, hence, it is often called program specialization. Essentially, partial evaluators are non-standard interpreters which evaluate expressions while termination is guaranteed and specialization is considered profitable. In PE of logic programs, such evaluation basically consists in building a

```
main(InArgs,Top) :-
    build_init_state(InArgs,S0),
    execute(S0,st(_,[Top|_],_)).
execute(S,S):-
    S = st(PC,_,_),
    bytecode(PC,return,_).
execute(S1,Sf) :-
    S1 = st(PC,_,_),
    bytecode(PC,Inst,_),
    step(Inst,S1,S2),
    execute(S2,Sf).

step(const(_T,Z),st(PC,S,L),S2) :-
    next(PC,PCp),
    S2 = st(PCp,[Z|S],L).
```

```
step(istore(X),st(PC,[I|S],L),S2) :-
    next(PC,PCp),
    localVar_update(L,X,I,Lb),
    S2 = st(PCp,S,Lb).

step(goto(O),st(PC,S,L),S2) :-
    PCp is PC+O,
    S2 = st(PCp,S,L).
....
....
next(PC,PCp) :-
    bytecode(PC,_,N),
    PCp is PC + N.
```

Fig. 1. Fragment of simplified bytecode interpreter

partial SLD tree for a given atom. How to construct the evaluations and when to stop them is determined by the *local control* (also referred to as *unfolding rule*). In state-of-the-art partial evaluators, HEm is used to guarantee termination by ensuring that the sequence of covering ancestors of the atom selected for further unfolding remains finite (see, e.g., [15]). When the embedding whistle blows, evaluation is terminated and the selected atom is passed to the *global control*, whose role is to ensure that we do not try to specialize an infinite number of atoms. Here again, HEm can be applied to guarantee finiteness of the set of atoms which are specialized. Now, if the whistle blows, the atom is generalized so that it no longer embeds any of the previous atoms.

As an example, in Fig. 1 we show a fragment of a simplified imperative bytecode interpreter implemented in Prolog. If the partial evaluator is powerful enough, given a bytecode program we can obtain a decompiled version of it in Prolog (see e.g. [1]). For brevity, we omit the code of some predicates like `build_init_state/2` (whose purpose is explained below) and `localVar_update/4` which simply updates the value of a local variable. We only show the definition of `step/3` for a reduced set of instructions. Furthermore, we have removed the frame stack and therefore only intra-procedural executions are considered. The bytecode to be decompiled is represented as a set of facts `bytecode(PC,Inst,NumBytes)` where `PC` contains the program counter position, `Inst` the particular bytecode instruction, and `NumBytes` the number of bytes the instruction takes up. A state is of the form `st(PC,Stack,Local)` where `Stack` represents the operand stack and `Local` the list of local variables. The predicate `main/2`, given the input method arguments `InArgs`, first builds the initial state by means of predicate `build_init_state/2` and then calls predicate `execute/2`. In turn, `execute/2` first calls predicate `step/3`, which produces `S2`, the state after executing the corresponding bytecode, and then calls predicate `execute/3` recursively with `S2` until we reach a `return` instruction.

Now, we want to decompile a method which receives an integer and executes a loop where a counter (initialized to "0") is incremented by one at each iteration until the counter reaches the value of the input parameter. For this, we partially

evaluate the interpreter w.r.t. the bytecode of this method by specializing the atom: `main([N],I)`, where `N` is the input parameter and `I` represents the returned value (i.e. the top of the stack at the end of the computation).

Let us first consider an online partial evaluator[4] which uses HEm to control termination both at the local and global control levels. We do not show the SLD trees built by the partial evaluator nor the decompilation due to space limitations. However, it suffices to know that in the bytecode program, the `PC` value "2" corresponds to the loop entry. By applying HEm, the evaluation contains a subsequence of atoms of the form: $\texttt{execute}(\texttt{st}(2,[],[\texttt{N},0]),\texttt{S}_\texttt{f})$, $\texttt{execute}(\texttt{st}(2,[],[\texttt{N},1]),\texttt{S}_\texttt{f})$, $\texttt{execute}(\texttt{st}(2,[],[\texttt{N},2]),\texttt{S}_\texttt{f})\ldots$, which correspond to consecutive iterations of the loop in which the control returns to the loop head with a value for the loop counter (local variable at the second position in the resulting state) increased by one. This sequence can grow infinitely, as the HEm does not flag it as potentially dangerous. In order to get a quality decompilation we need to filter the value of the counter (local variable) but not that of the PC. This would result in stopping the derivation when we hit the atom $\texttt{execute}(\texttt{st}(2,[],[\texttt{N},1]),\texttt{S}_\texttt{f})$ and its generalization into $\texttt{execute}(\texttt{st}(2,[],[\texttt{N},\texttt{X}]),\texttt{S}_\texttt{f})$.

A possible relatively straightforward solution in this case is to use the relation $\trianglelefteq_{num}$ which is a slight adaptation of HEm which filters numeric values, i.e., any number embeds any other number. Under this relation, the atom $\texttt{execute}(\texttt{st}(2,[],[\texttt{N},1]),\texttt{S}_\texttt{f})$ embeds $\texttt{execute}(\texttt{st}(2,[],[\texttt{N},0]),\texttt{S}_\texttt{f})$ and therefore we avoid non-termination. Unfortunately, this modification to HEm, though is too conservative and leads to excessive precision loss. For instance, at the beginning of the specialization process we have the atom $\texttt{execute}(\texttt{st}(0,[],[\texttt{N},0]),\texttt{S}_\texttt{f})$ and, after one unfolding step, we obtain the atom $\texttt{execute}(\texttt{st}(1,[0],[\texttt{N},0]),\texttt{S}_\texttt{f})$. By using $\trianglelefteq_{num}$, the whistle blows at this point and unfolding has to stop. Furthermore, the latter atom is generalized into $\texttt{execute}(\texttt{st}(\texttt{X},\texttt{Y},[\texttt{N},0]),\texttt{S}_\texttt{f})$ before proceeding with the specialization. This turns out not to be acceptable for specialization of our interpreter, since we lose track of what is the next instruction to execute, which avoids eliminating the interpretation layer and in many cases the residual program ends up containing the original interpreter.

Another solution is to use an extension of the embedding relation, as explained in [11], which is based on a distinction between the finite number of symbols actually occurring in the program and goal. Under this relation, the atom $\texttt{execute}(\texttt{st}(1,[0],[\texttt{N},0]),\texttt{S}_\texttt{f})$ does not embed $\texttt{execute}(\texttt{st}(0,[],[\texttt{N},0]),\texttt{S}_\texttt{f})$, as the numbers `0` and `1` are different static symbols which occur in the program. Hence, we are not forced to generalize them and lose the `PC` value. However, this extended embedding turns out not to be optimal either since we have that $\texttt{execute}(\texttt{st}(2,[],[\texttt{N},1]),\texttt{S}_\texttt{f})$ does not embed $\texttt{execute}(\texttt{st}(2,[],[\texttt{N},0]),\texttt{S}_\texttt{f})$. This means that we will not stop the unfolding process after evaluating one iteration of the loop, i.e., we proceed with a second iteration of the loop and so on. Although the process terminates once we have unfolded as many iterations of the loop as distinct numbers appear in the program, we are not able to achieve a quality decompilation. For obtaining a good decompilation, we need to generalize the loop counter, i.e., the atom $\texttt{execute}(\texttt{st}(2,[],[\texttt{N},1]),\texttt{S}_\texttt{f})$ has to embed $\texttt{execute}(\texttt{st}(2,[],[\texttt{N},0]),\texttt{S}_\texttt{f})$.

[4] We assume that we have a partial evaluator which is able to accurately handle built-in predicates and to safely perform non-leftmost unfolding [3].

This suggests that embeddings that take context into account are needed: an appropriate embedding handling PC values has to be different from one handling numeric values in program variables such as the loop counter.

3 Type-based Homeomorphic Embedding

In the presence of infinite signatures, a general method of defining homeomorphic embedding relations exists; an *extended homeomorphic embedding relation* is defined in [11] based on previous results by Kruskal [10] and by Dershowitz [6]. This solution defines a family of embedding relations, where a subsidiary ordering on function symbols plays an essential role. However, we argue that this does not really solve the practical problem of finding an effective embedding relation, since there is no automated mechanism for finding the "right" ordering relation on the functions in the signature.

In this section, we propose *typed-based homeomorphic embedding* (TbHEm for short), a relation which improves HEm by making use of additional information provided in the form of types. We outline how this approach can be seen as a way of generating program-specific instances of extended HEm as defined by Leuschel. Such additional information is program-dependent and might also be goal-dependent; it could be provided manually or be automatically inferred by program analysis, as we will see in Section 4.

3.1 Types: preliminaries and notation

In the following, let P be a program and Σ_P be a (possibly infinite) signature including the functions and constants appearing in P and goals for P as well as in computations of P. We adopt the syntax of Mercury [16] for type definitions. *Type expressions (types)*, elements of $\mathcal{T}$, are constructed from an infinite set of type variables (parameters) $\mathcal{V}_\mathcal{T}$ and an alphabet of ranked type symbols $\Sigma_\mathcal{T}$; these are disjoint from the set of variables V and the alphabet of functors $\Sigma_P^\mathcal{N}$ of a given program P respectively.

Definition 2 (type definition). *A* type rule *for a type symbol $h/n \in \Sigma_\mathcal{T}$ is of the form $h(\bar{T}) \longrightarrow f_1(\bar{\tau}_1); \ldots; f_k(\bar{\tau}_k); \ldots \quad (k \geq 1)$ where $\bar{T}$ is a n-tuple of distinct type variables, $f_1, \ldots, f_k, \ldots$ are distinct function symbols from Σ_P, $\bar{\tau}_i$ $(i \geq 1)$ are tuples of corresponding arity from $\mathcal{T}$, and type variables in the right hand side, if any, are from $\bar{T}$.*[5] *A* type definition *is a finite set of type rules where no two rules contain the same type symbol on the left hand side, and there is a rule for each type symbol occurring in the type rules.*

As in Mercury [16], a function symbol can occur in several type rules. In the definition above we allow type rules containing an infinite number of cases. Thus, standard infinite types such as *integer* are permitted, defined by a rule with an infinite number of cases containing the numeric constants. In order to define TbHEm we introduce some extra annotation into type rules. We consider the right hand side

[5] The last condition is known as *transparency* and is necessary to ensure that well-typed programs cannot go wrong [14, 8].

of each type rule to consist of two disjoint partitions, each possibly empty. More precisely, we will structure a type rule as $h(\bar{T}) \longrightarrow F; I$, where the union $F \cup I$ are the cases in the type rule, $F \cup I$ is non-empty, F is either empty or finite and I is either empty or infinite. A type $\tau \in \mathcal{T}$ is labelled (when necessary) with ∞ denoting *infinite* if I is non-empty in the rule defining τ. If a type τ is written with no label then it could be either finite or infinite. Note that there could be different partitions of the same type in different type definitions; for example $nat \longrightarrow F; I$ where $F = \emptyset$ and $I = \mathbb{N}$, or $F = \{0, 1, 2\}$ and $I = \mathbb{N} \setminus \{0, 1, 2\}$, etc.

A *predicate signature* for an n-ary predicate p is of the form $p(\bar{\tau})$ and declares a type $\tau_i \in \mathcal{T}$ for each argument of the predicate p/n. The standard concept of a *well-typed program* is assumed, restricted to be *monomorphic* in the sense that the atoms in a clause, and their sub-terms, can be assigned types such that the type assigned to each head and body atom is a variant of the signature for its predicate, and multiple occurrences of the same variable in the clause are assigned the same type. A more general well-typing allows the types of the body atoms to be instances of the signatures rather than variants. It suffices for our purpose to state that, given a well-typed program and a well-typed atomic goal, then each atom arising in computations of the goal (that is, in an SLD tree for the program and goal) has a type that is a variant of its respective signature. In short, a well-typed program and goal generate only well-typed atoms in computations. Furthermore, the monomorphic assumption implies that only a finite number of types arises during computation.

3.2 Type-based Homeomorphic Embedding

We now define TbHEm ($\trianglelefteq_T$). We first define a subsidiary relation on function symbols paired with their associated types.

Definition 3. *Let $\preceq_F$ be the following relation on the set of pairs $\Sigma_P \times \mathcal{T}$; we assume that $\mathcal{T}$ is finite and there is a set of type rules defining the types. Σ_P is possibly infinite, but we assume that the arity of the function symbols is bounded. $(f_1, \tau_1) \preceq_F (f_2, \tau_2)$ iff either $f_1 = f_2 \wedge \tau_1 = \tau_2$ or f_1 and f_2 have the same arity, the rule defining τ_2 is of form $h(\bar{V}) \longrightarrow F; I$, and f_2 is in the infinite partition I.*

Definition 4 ($\trianglelefteq_T$). *We write $t{:}\tau$ to mean that term t is of type τ. Given two typed atoms $A = p(t_1, \ldots, t_n)$ and $B = p(s_1, \ldots, s_n)$, with predicate signature $p(\tau_1, \ldots, \tau_n)$, we say that A is embedded by B, written $A \trianglelefteq_T B$, if $t_i : \tau_i \trianglelefteq_T s_i : \tau_i$ for all i s.t. $1 \leq i \leq n$. The embedding relation over typed terms, also written $\trianglelefteq_T$, is defined by the following rules:*

1. *$Y{:}\tau_Y \trianglelefteq_T X{:}\tau_X$ for all variables X, Y.*
2. *$s : \tau' \trianglelefteq_T f(t_1, \ldots, t_n) : \tau$ if $s : \tau' \trianglelefteq_T t_i : \tau_i$ for some i, where $\tau_1, \ldots, \tau_n$ are the respective types of $t_1, \ldots, t_n$.*
3. *$f(s_1, \ldots, s_n) : \tau_1 \trianglelefteq_T g(t_1, \ldots, t_m) : \tau_2$ if $(f, \tau_1) \preceq_F (g, \tau_2)$, and $s_i : \tau_i \trianglelefteq_T t_i : \tau_i'$ for all i, $1 \leq i \leq n$, where $\tau_1, \ldots, \tau_n, \tau_1', \ldots, \tau_n'$ are the respective types of $s_1, \ldots s_n, t_1, \ldots, t_n$.*

Referring to Definition 3, rule 3 specifies that embedding can occur between terms with different function symbols, where the function symbol of the "larger" term is from the I partition of its type. However, as long as we compare distinct

terms from an infinite type and remain within the finite part F of the type, no embedding (using rule 3) occurs since the condition $(f, \tau_1) \preceq_F (g, \tau_2)$ does not hold. For instance, consider the following predicate signature and type definition, $p(\tau)$ and $\tau \longrightarrow F; I$. We have that $p(1) \trianglelefteq_T p(2)$ if $F = \emptyset$ and $I = \mathbb{N}$. However, $p(1) \not\trianglelefteq_T p(2)$ if $F = \{0, 1, 2\}$ and $I = \mathbb{N} \setminus \{0, 1, 2\}$.

Proposition 1. *Given a type definition and set of signatures, there is no infinite sequence of well-typed atoms $A_1, A_2, \ldots$ such that for all i, j where $i < j$, $A_i \not\trianglelefteq_T A_j$.*

Proof. (Outline). The ordering defined above can be seen as a special case of the "extended homeomorphic embedding" $\trianglelefteq^*$ [11], which is defined for terms over infinite signatures. The detailed proof shows that the relation $\preceq_F$ is a *well binary relation* on the set $\Sigma_P \times \mathcal{T}$.

We remark that this could be seen as a refinement of the idea sketched in [11] to build an extended homeomorphic embedding based on a distinction between the finite number of symbols actually occurring in the program and goal (the *static* symbols), and the rest (the *dynamic* symbols). However, the types allow a more fine-grained control over the embedding than is possible with that approach. Also, in Definition 3 functions have to have the same arity in order for the $\preceq_F$ to hold. This restriction could be relaxed, using an ordering on sequences as in the definition of extended homeomorphic embedding [11].

Note that, if we assume an embedding relation based on a given set of types and signatures that is a well-typing for a program, we are assured that the embedding relation is well-defined for all pairs of atoms arising in computations of that program.

4 Automatic Inference of Well-Typings

In this section we outline an approach to constructing in an automatic way suitable types to be used in online partial evaluation in combination with TbHEm, given an untyped program and a goal or set of goals. The approach is based on existing analysis tools for constraint logic programs.

We note first that the problem does not allow a precise, computable solution. Determining the exact set of symbols that can appear at run-time at a specific program point, and in particular determining whether the set is finite, is closely related to termination detection and is thus undecidable. However, the better the derived types are, the more aggressive partial evaluation can be without risking non-termination. If the derived types have finite components that are too small, the over-generalization is likely to result; if they are too large, then specialization might be over-aggressive, producing unnecessary versions.

A procedure for constructing a monomorphic well-typing of an arbitrary logic program was described by Bruynooghe *et al.* [5][6]. The procedure scales well (roughly linear in program size) and is robust, in that every program has a well-typing, and the procedure works with partial programs (modules).

In the original type inference procedure, an externally defined predicate such as `is/2` is treated as if defined by a clause `X is Y :- true` and is thus implicitly assumed not to generate any symbols not occurring elsewhere in the program. In

[6] available on-line at `http://wagner.ruc.dk/Tattoo/`

deriving types for partial evaluation, we provide a type for such built-ins in the form of a dummy additional "fact" for `is/2`, namely `num is num :- true`. The constant `num` (assumed not to occur elsewhere in the program) will thus propagate during type inference into those types that unify with the types of the `is` predicate arguments. In the resulting inferred types, we interpret occurrences of the constant `num` as being an abbreviation for an infinite set of cases.

Example 1. A type is inferred for the bytecode interpreter sketched in Figure 1, together with a particular bytecode program. Note that the program counter is sometimes computed in the interpreter using the predicate `is/2` as an offset from the current program counter value and hence its type is in principle any number.

When the extra fact `num is num :- true` is added to the program, the inferred type for the program counter argument `PC` is as follows.

```
t51 --> -8; 0; 1; 2; 3; 4; 5; 6; 7; 8; 9; 10; 11; 12; 13; 14; num
```

This type can be naturally interpreted as consisting of a finite part (the named constants) and an infinite part (the numbers other than the named constants). In other words, the partition F of the rule is $\{-8, 0, 1, 2, \ldots, 14\}$ and $I = \texttt{num} \setminus F$. Using the rule structured in this way, the typed-homeomorphic embedding ensures that the program counter is never abstracted away during partial evaluation, so long as its value remains in the expected range (the named constants). In particular, the atom $\texttt{execute}(\texttt{st}(1, [0], [\texttt{N}, 0]))$ does not embed $\texttt{execute}(\texttt{st}(0, [], [\texttt{N}, 0]))$ by using the type definition above, thus, the derivation can proceed. This avoids the need for generalizing the `PC` what would prevent us from having a quality specialization (decompilation) as explained in Sect. 2. The derivation will either eventually end or the `PC` value will be repeated due to a backwards jump in the code (loops). In this case, $\trianglelefteq_T$ will flag the relevant atom as dangerous, e.g., $\texttt{execute}(\texttt{st}(2, [], [\texttt{N}, 0])) \trianglelefteq_{\texttt{T}} \texttt{execute}(\texttt{st}(2, [], [\texttt{N}, 1]))$. If however, a different value arose, perhaps due to an addressing error, the infinite part of the type rule `num` is encountered and embedding (followed by generalization of the program counter argument) would take place.

5 Analysis of Numeric Bounds

It is important to note that TbHEm allows us to distinguish a finite set of functors (the F component of the type rules) even in the case of infinite signatures. A non-empty I component in type rules often arises during inference of well-typings. We now consider performing additional dataflow analysis in order to infer that the I component in type rules is empty. Indeed, we would like to infer whether a type τ is a *bounded interval*, i.e., if the type rule for τ is of the form $\tau \longrightarrow F; \emptyset$ and F is a finite set of values.

Given a logic program processing numeric values, analyses exist that make over-approximations of the set of values that the program arguments can have. Polyhedral analyses are perhaps the most widely known of these and they have successfully been applied to constraint logic programs [4]. When we can prove that the set of values that all program arguments can have is bounded, then we know that its infinite partition is empty and, hence, we can safely apply traditional HEm (and improve the effectiveness of PE).

Let us assume for the sake of this discussion that a polyhedral analysis can return, for a given program and goal, an approximation to the set of calls to each n-ary predicate p, in the form:

$$p(X_1, \ldots, X_n) \leftarrow c(X_1, \ldots, X_n).$$

where the expression $c(X_1, \ldots, X_n)$ is a set of linear constraints (describing a closed polyhedron). From this information it can be determined whether each argument X_i is bounded or not by projecting $c(X_1, \ldots, X_n)$ onto X_i. If it is bounded (from above and below), and it is known that the ith argument takes on integral values, then it can take only a finite set of values.

Example 2. Consider the following clauses defining a procedure for computing an exponential.

```
exp(Base,Exp,Res) :- exp_(Base,Exp,1,Res).
exp_(_,0,Ac,Ac).
exp_(Base,Exp,Ac,Res) :-  Exp > 0, Exp' is Exp-1, Ac' is Ac*Base,
                          exp_(Base,Exp',Ac',Res)
```

Type inference yields the following signature for the predicate `exp_/4`.

```
exp_(t24,t24,t24,t24)
```

with the type `t24 --> 0; 1; num`. A polyhedral analysis of the same program with respect to the goal `exp(Base,10,Res)` yields the following approximation to the queries to `exp_/4`.

```
exp_(Base,Exp,Ac,Res) :- Exp > -1, Exp =< 10.
```

The second argument is thus bounded. Combining this with the inferred type, we obtain the signature `exp_(t24,s,t24,t24)` with the types `t24 --> 0; 1; num` and `s --> 0..10` (we use the interval notation `0..10` as a shortcut to `0; 1; .. ; 10`). Here we assume that the second argument can take on only integer values. The finite type `0..10` implies that the typed-homeomorphic embedding will not abstract away the value of the second argument of `exp_/4` and this allow maximum specialization to be achieved.

6 Discussion and Related Work

Guaranteeing termination is essential in a number of tasks which have to deal with possibly infinite computations. These tasks include partial evaluation, abstract model checking, rewriting, etc. Broadly speaking, guaranteeing termination can be tackled in an *offline* or an *online* fashion. The main difference between these two perspectives is that in offline termination we aim at statically determining termination. This means that we do not have the concrete values of arguments at each point of the computation but rather just *abstractions* of such values. Traditionally, these abstractions refer to the *size* of values under some measure such as list length, term size, numeric value for natural numbers, etc. In contrast, in online termination, we aim at dynamically guaranteeing termination by supervising the computation in such a way that it is not allowed to proceed as soon as we can no longer guarantee termination. The main advantage of the offline approach is that if we can prove termination statically, there is no longer any need to supervise the computation for

termination, which results in important performance gains. On the other hand, the online approach is potentially more precise, since we have the concrete values at hand, but also more expensive, because of the overhead introduced by the termination supervision.

In the context of partial evaluation, our problem in the online setting is similar to offline termination in that we have to find conditions for ensuring local and global termination. In offline PE, the problem of termination of local unfolding has been tackled by annotating arguments as "bounded static". The work of Glenstrup and Jones [7] is the main reference, though the idea of bounded static variation goes back a long way. To detect bounded static arguments it is necessary to prove some decrease in well-founded ordering (e.g. using size-change techniques). Quasi-termination is a bit weaker than standard termination but still quite hard to prove. There is Vidal's recent work on this [17] as well as Glenstrup-Jones [7]. On the other hand, ensuring termination in online PE is easier because we can use "dynamic" termination detection based on supervisors of the computations such as for example embeddings. This means that we do not need any well-founded orderings but only well-quasi-orderings. In effect, in our technique it is only necessary to show boundedness of an argument's values instead of decrease.

References

1. E. Albert, M. Gómez-Zamalloa, L. Hubert, and G. Puebla. Verification of Java Bytecode using Analysis and Transformation of Logic Programs. In *Proc. PADL*, number 4354 in LNCS, pages 124–139. Springer-Verlag, 2007.
2. E. Albert, M. Hanus, and G. Vidal. A practical partial evaluation scheme for multi-paradigm declarative languages. *Journal of Functional and Logic Programming*, 2002(1), 2002.
3. E. Albert, G. Puebla, and J. Gallagher. Non-Leftmost Unfolding in Partial Evaluation of Logic Programs with Impure Predicates. In *Proc. of LOPSTR'05*. Springer LNCS 3901, pages 115–132, April 2006.
4. F. Benoy and A. King. Inferring argument size relationships with CLP(R). In John P. Gallagher, editor, *Logic-Based Program Synthesis and Transformation (LOPSTR'96)*, volume 1207 of *Springer-Verlag LNCS*, pages 204–223, August 1996.
5. Maurice Bruynooghe, John Gallagher, and Wouter Van Humbeeck. Inference of well-typings for logic programs with application to termination analysis. LNCS 3672, pages 35–51, 2005.
6. N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. B*, pages 243–320. Elsevier, 1990.
7. A. J. Glenstrup and N. D. Jones. Termination analysis and specialization-point insertion in offline partial evaluation. *ACM Trans. Program. Lang. Syst.*, 27(6):1147–1215, 2005.
8. Patricia M. Hill and Rodney W. Topor. A semantics for typed logic programs. In Frank Pfenning, editor, *Types in Logic Programming*, pages 1–62. MIT Press, 1992.
9. N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, New York, 1993.
10. J.B. Kruskal. Well-quasi-ordering, the tree theorem, and Vazsonyi's conjecture. *Transactions of the American Mathematical Society*, 95:210–225, 1960.
11. M. Leuschel. Homeomorphic embedding for online termination of symbolic methods. In *The Essence of Computation*, volume 2566 of *LNCS*, pages 379–403. Springer, 2002.

12. M. Leuschel and M. Bruynooghe. Logic program specialisation through partial deduction: Control issues. *Theory and Practice of Logic Programming*, 2(4 & 5):461–515, July & September 2002.
13. Michael Leuschel. On the power of homeomorphic embedding for online termination. In Giorgio Levi, editor, Static Analysis. *Proceedings of SAS'98*, LNCS 1503, pages 230–245, Pisa, Italy, September 1998. Springer-Verlag.
14. Alan Mycroft and Richard A. O'Keefe. A polymorphic type system for Prolog. *Artif. Intell.*, 23(3):295–307, 1984.
15. G. Puebla, E. Albert, and M. Hermenegildo. Efficient Local Unfolding with Ancestor Stacks for Full Prolog. In *Proc. of LOPSTR'04*, pages 149–165. Springer LNCS 3573, 2005.
16. Z. Somogyi, F. Henderson, and T. Conway. The Execution Algorithm of Mercury: an Efficient Purely Declarative Logic Programming Language. *JLP*, 29(1–3), October 1996.
17. G. Vidal. Quasi-Terminating Logic Programs for Ensuring the Termination of Partial Evaluation. In *ACM PEPM'07*, pages 51–60, 2007.

Improving Efficiency of Prolog Programs by Fully Automated Transformation

Jiří Vyskočil and Petr Štěpánek

Charles University, Malostranské náměstí 25, Czech Republic

1 Introduction

Two important methods for transformation of logic programs are Unfold/Fold transformations [12] and Partial Deduction (partial evaluation). In [11] an algorithm based on the Unfold/Fold transformation was designed to get speedup by eliminating so called unnecessary variables in logic programs. The authors call it Elimination Procedure and identify two classes of definite programs on which the Elimination Procedure terminates and eliminates all unnecessary variables. A variant of this strategy is proposed in the present paper which terminates for all definite programs, but in general, does not eliminates all unnecessary variables. The Unfold/Fold transformations can be used for program specialization, and there are several fully automated specialization methods using Unfold/Fold transformations (e.g. the *Paddy* and *Mixtus* systems). On the other hand, Conjunctive Partial Deduction (CPD) [4] can be used to eliminate intermediate structures (e.g. [4,9]). The comparison of Unfold/Fold and CPD methods is rather complicated (see [4]). It is due to the fact that not all corresponding algorithms are fully automated and that there are restrictions of input programs which almost each such method needs.

In the present paper, we describe a transformation motivated by Unfold-Definition-Fold method (UDF) [11] and by Partial Deduction. It terminates for all definite logic programs and is fully automated[1]. The proof of its termination proof is obtained by homeomorphic embedding used first by Dershowitz [3][2] and some techniques similar to those used in the termination proof of the Conjunctive Partial Deduction [4]. In most cases, the transformed programs have better computational behaviour than the original ones. Experimental results: measuring time and a number of inferences are included. The obtained results are compared with the results of Conjunctive Partial Deduction implemented in the ECCE system. Although the present method is not optimized and has no pre-processing and only weak post-processing, the results are encouraging.

We suppose, that the reader is well equinted with the standard of concept of unfolding and folding. In the paper, we shall adopt the notation used in Apt [1].

[1] It means that the transformation algorithm presented in this paper terminates and it is applicable to all definite logic programs (which are given as input). It does not mean that the algorithm converts non-terminating programs into terminating ones.

[2] But the use is of a different nature than the one for CPD and this paper.

2 Eliminating Unnecessary Variables - Motivation

Existential variables are often used in logic programs for storing intermediate results. Multiple variables are used for multiple structure traversals. Thus in many cases, eliminating them improves the efficiency of the program.

Definition 1. *(Unnecessary variables)*

Let c be a clause, and P be a program.

- We say that variables occurring only in the body of c are *existential* variables.
- Variables that occur in the body of c more than once are called *multiple* variables.
- Both *existential* and *multiple* variables in the clauses of P are called *unnecessary variables* of P.

As shown in [11] there are two classes of definite programs for which all unnecessary variables can be eliminated by repeating UDF (unfolding-definition-folding) transformation steps in this order. The authors of [11] showed that in general it is undecidable whether, for any input, this elimination procedure terminates[3].

We present a new transformation that is motivated by the above UDF method which terminates for all definite Prolog programs. However, it may not eliminate all unnecessary variables. The algorithm has been implemented as fully automated and in most cases the resulting programs are computationally more efficient than the original ones.

First we illustrate the transformation process on an example.

Example 1.

Consider a naive but intuitive version of `DOUBLEAPPEND` used in [4] which concatenates three lists.

```
(1)   doubleapp(X,Y,Z,XYZ)  :- append(X,Y,XY), append(XY,Z,XYZ).
(2)   append([],L,L).
(3)   append([H|X],Y,[H|Z]) :- append(X,Y,Z).
```

In the first clause, there are two occurrences of the unnecessary variable `XY`. Using the strategy Unfold-Definition-Fold, we remove this variable and obtain a less intuitive but more efficient version of `DOUBLEAPPEND`. After several unfolding, definition and folding steps we get a following program (for more detail see the algorithm in Section 4), which is a more efficient version of the original program without double traversing of the intermediate list.

[3] The authors of [11] also defined so called extended elimination procedure which terminates for all definite programs but may not eliminate all unnecessary variables. As it is obvious, that our algorithm uses a different control and termination mechanism than it is proposed in [11] which is one of the contributions of the present paper.

```
(1a)  doubleapp([],Y,Z,XYZ)          :- append(Y,Z,XYZ).
(1b)  doubleapp([H|X],Y,Z,[H|XYZ]) :- doubleapp(X,Y,Z,XYZ)
(2)   append([],L,L).
(3)   append([H|X],Y,[H|Z]) :- append(X,Y,Z).
```

The above example shows that we need an instrument for the choice of atoms for unfolding (it is the first and the second unfolding step if you follow the algorithm in Section 4) and another instrument for proving the termination of the algorithm. To solve the latter requirement, we use a version of homeomorphic embedding [3,4,8], which was used for a termination proof of Conjunctive partial deduction. To solve the former one we use a syntactic method.

3 Homeomorphic Embedding, Linking Variables and Safe Selection

The idea of homeomorphic embedding on sets of terms goes back to Kruskal [7] and Nash-Williams [10].

Definition 2. *(The homeomorphic embedding)*

Assume that we have a language with finite special symbols and the potentially infinite set of variables. Note that every successful computation uses only finitely many variables.
The homeomorphic embedding $\trianglelefteq$ on the set of all atomic formulas and terms is defined inductively as follows:

- $X \trianglelefteq Y$ for all variables,
- (diving rule) $s \trianglelefteq f(t_1, \ldots, t_n)$ if $s \trianglelefteq t_i$ for some $i \in \{1...n\}$,
- (coupling rule) $f(s_1, \ldots, s_n) \trianglelefteq f(t_1, \ldots, t_n)$ if $s_i \trianglelefteq t_i$ for every $i \in \{1...n\}$,

where s, s_i, t_i are terms, and in diving and coupling rule $n = 0$ is allowed.

Intuitively, an expression A is homeomorphically embedded into an expression B iff B is more complex than A in the following sense: A can be obtained from B by "simplifying" some subexpressions of B. Note that all variables are treated in the same way. It was shown by Kruskal [7] and Nash-Williams [10] that the relation $\trianglelefteq$ of homeomorphic embedding is a well quasi-ordering on the set of all terms and atomic formulas. De Schreye et al. [4] used this result to prove termination of their algorithm of conjunctive partial deduction. We use a similar approach (but not the same) in proving termination of our Unfold/Fold algorithm.

Definition 3. *(Partition of the body of a clause) Given a set of atoms, we define a binary relation* $\sim$ *as follows. For two atoms* A, B, *we put*

$$A \sim B \text{ iff } Vars(A) \cap Vars(B) \neq \varnothing$$

It means that $A \sim B$ *iff* A *and* B *have at least one variable in common. Obviously,* $\sim$ *is symmetric and reflexive, thus the transitive closure* $\approx$ *of* $\sim$

is an equivalence relation. Given a clause c

$$H \leftarrow A_1, \ldots, A_n$$

We denote by $PartB(c)$ [4] *the partition of the set* $\{A_1, \ldots, A_n\}$ *of the atoms of the body of* c *to disjoint subsets induced by the equivalence relation* $\approx$ *. The elements of the partition are called the blocks of the clause* c*.*

Definition 4. *(Linking variables of a block) Let* c *be a clause* $H \leftarrow A_1, \ldots, A_n$ *and let* $B \in PartB(c)$ *be a block in the body of* c*. The set of the linking variables of* B *is defined as follows*

$$LinkVars_c(B) = Vars(H) \cap Vars(B)$$

Note that linking variables are not existential variables of the clause c*.*

Definition 5. *(Faithful variant of a block) A Block* B_1 *of a clause* c_1 *is called a faithful variant of a block* B_2 *of a clause* c_2 *iff there exists a renaming substitution* θ *such that the following conditions hold*
(i) $B_1 = B_2\theta$ *(*B_1 *is a variant of* B_2*)*
(ii) $(\forall X \in Vars(B_2))\ (X \in LinkVars_{c_2}(B_2) \leftrightarrow X\theta \in LinkVars_{c_1}(B_1))$

Definition 6. *(Safe selected atom) Let* $\trianglelefteq$ *be the homeomorphic embedding on the set of atoms and terms defined in Definition 2. Let* c *be a clause*

$$H \leftarrow A_1, \ldots, A_k, \ldots, A_n$$

which arises from an original clause by sequential unfolding via selected atoms $B_1, \ldots, B_m$*. We say that the atom* A_k *is safe selected for unfolding iff* $A_k \not\trianglelefteq B_i$ *for each atom* B_i *.*

4 The Algorithm

The algorithm is motivated by the Unfold-Definition-Fold strategy presented in [11]. The key idea in this new algorithm is the selection of a clause and of one of its atoms for Unfolding. The selection rule used in the presented algorithm makes the transformation applicable to all definite logic programs. In most cases, the transformed programs are more effective than original ones (see Section 5). The transformation is fully automated.

Our algorithm improves the original elimination procedure [11] which limits the class of programs for which the algorithm is applicable and has no fully automated implementation. The complete elimination of unnecessary variables in the algorithm is not guaranteed for all definite logic program (more about necessary conditions for elimination can be found in [11]).

[4] The $PartB(c)$ is implemented in our algorithm as follows. An input is a list of atoms with an ordering from the original clause c. An output is a list of lists, where each element from the output list corresponds to each block from $PartB(c)$ with a preserved ordering of atoms from the original clause c inside each block.

Algorithm 1 Procedure for elimination of unnecessary variables

Input: A definite logic program $\boldsymbol{P}$.

Output: A set ***TransfP*** of clauses such that for each goal $\boldsymbol{G}$ consisting of predicates of $\boldsymbol{P}$, the answer set of $\boldsymbol{P}$ via $\boldsymbol{G}$ is equal to answer set of ***TransfP*** via $\boldsymbol{G}$ (as shown in Example 1, the set of clauses ***TransfP*** can be interpreted as the resulting transformed program).

1. *let* $\boldsymbol{Q}$ be an empty queue of pairs ¡*Clause*, *History*¿ where *History* is a set of atoms and *Clause* is a definite clause;

2. $\boldsymbol{TransfCs,D,Cs} := \varnothing$;

3. *for each* clause $\boldsymbol{C} \in \boldsymbol{P}$:

(i) ***if*** $\boldsymbol{C}$ contains at least one unnecessary variable, then
$\boldsymbol{Cs} := \boldsymbol{Cs} \cup \{\boldsymbol{C}\}$;
(ii) if the head in $\boldsymbol{C}$ occurs only once as the head in $\boldsymbol{P}$, then
$\boldsymbol{D} := \boldsymbol{D} \cup \{\boldsymbol{C}\}$;

4. *for each* clause $\boldsymbol{C} \in \boldsymbol{Cs}$ *push* pair ¡$\boldsymbol{C}$, $\varnothing$¿ into $\boldsymbol{Q}$;

5. *while* $\boldsymbol{Q} \neq \varnothing$ *do*

Unfolding steps:

- ¡$\boldsymbol{C}$,$\boldsymbol{H}$¿ := *pop* $\boldsymbol{Q}$;
- select the first *safe selected atom* $\boldsymbol{A}$ from the body of $\boldsymbol{C}$, that is:
  for each $\boldsymbol{h} \in \boldsymbol{H}$ $\boldsymbol{A} \not\trianglelefteq \boldsymbol{h}$;
- *if* no such $\boldsymbol{A}$ exists *then*
  $\boldsymbol{TransfCs} := \boldsymbol{TransfCs} \cup \{\boldsymbol{C}\}$; *goto* 5;
  else unfold atom $\boldsymbol{A}$ in the body of $\boldsymbol{C}$ using clauses in $\boldsymbol{P}$ and store resulting unfolded clauses in the set $\boldsymbol{Us}$;

Definition steps:

- *for each* clause $\boldsymbol{E} \in \boldsymbol{Us}$
  for each block $\boldsymbol{B} \in$ PartB($\boldsymbol{E}$) such that:
  (i) $\boldsymbol{B}$ contains at least one unnecessary variable, and
  (ii) $\boldsymbol{B}$ is not a *faithful variant* of the body of any clause which is in $\boldsymbol{D}$ then
  let $\boldsymbol{F} := newp(\boldsymbol{X}_1, \ldots, \boldsymbol{X}_n) \text{ :- } \boldsymbol{B}$,
  where *newp* is a fresh predicate symbol and $\boldsymbol{X}_1, \ldots, \boldsymbol{X}_n$ are *linking variables* of $\boldsymbol{B}$ with respect to the head of $\boldsymbol{E}$,
  push the pair ¡$\boldsymbol{F}$, $\boldsymbol{H} \cup \{\boldsymbol{A}\}$¿ into $\boldsymbol{Q}$,
  $\boldsymbol{D} := \boldsymbol{D} \cup \{\boldsymbol{F}\}$;

Folding steps:

- *for each* clause $\boldsymbol{E} \in \boldsymbol{Us}$
  begin
  for each block $\boldsymbol{B} \in$ PartB($\boldsymbol{E}$) such that:
  $\boldsymbol{B}$ is a *faithful variant* of body of a clause $\boldsymbol{N}$ from $\boldsymbol{D}$,
  then fold $\boldsymbol{B}$ in $\boldsymbol{E}$ using $\boldsymbol{N}$ to obtain $\boldsymbol{E}$;

  $\boldsymbol{TransfCs} := \boldsymbol{TransfCs} \cup \{\boldsymbol{E}\}$
  end;

end while

6. *for each* clause $\boldsymbol{E} \in \boldsymbol{TransfCs}$ such that:

$\boldsymbol{E}$ contains atoms $\boldsymbol{As}$ in its body which are defined by at most one clause from $(\boldsymbol{P} \setminus \boldsymbol{Cs}) \cup \boldsymbol{TransfCs}$ (it means that there is at most one clause in $(\boldsymbol{P} \setminus \boldsymbol{Cs}) \cup \boldsymbol{TransfCs}$ with the head which is unifiable with an atom in the body of $\boldsymbol{E}$),
then replace $\boldsymbol{E}$ in $\boldsymbol{TransfCs}$ by $\boldsymbol{E}$ with all atoms in $\boldsymbol{As}$ unfolded;

7. $\boldsymbol{TransfP} := (\boldsymbol{P} \setminus \boldsymbol{Cs}) \cup \boldsymbol{TransfCs}$;

8. *end.*

The termination proof uses the concepts of safe selection and homeomorphic embedding. Numbers 1 - 5 are the main part of the algorithm. Number 6 is a post-processing part. Proofs of termination and correctness of the presented algorithm are beyond the scope of this paper and can be found in [16].

5 Benchmarks

The following table compares several comparative benchmarks for original Prolog sources[5], sources transformed by ECCE 1.1 [9], sources transformed by Algorithm 1 1[6] and sources transformed by a simple combination of both algorithms. The comparison was made on one representative predefined goal for each tested program. The original Prolog program is denoted by $\boldsymbol{P}$ in the input for Algorithm 1 (UDF).

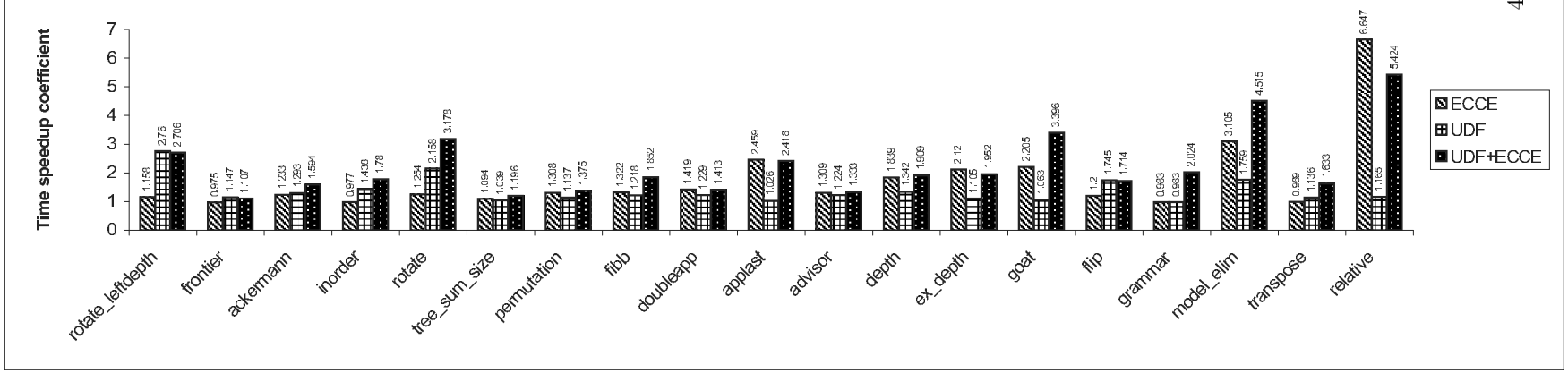
To make a comparison of the above algorithm we put to ECCE the original Prolog source $\boldsymbol{P}$ and a specialisation goal for partial evaluation. This goal is a predicate from the predefined test goal with free variables for all its arguments. The following flags were enabled in ECCE: `RAF Filtering`, `FAR Filtering`, `Dead Code Elimination`, `Remove Redundant Calls`, `Determinate Post Unfolding`, `Reduce Polyvariants`. The combination of both algorithms works in the following way. First, the original source was transformed by Algorithm 1 (UDF) and then the output was put as an input for ECCE.

Because the ECCE system is nowadays probably the best fully automated partial evaluation system based on Conjunctive Partial Deduction (CPD) it is very interesting to compare it with our system which is to the best of our knowledge the first fully automated algorithm based on Unfold/Fold transformations.

Measurements of the number of inferences[7], of the inference speedup and of the number of clauses have been performed on SWI-Prolog 5.2.13. These measured parameters (with the exception of time) do not depend on any specific operating system or hardware architecture. Measurements of CPU runtime and CPU runtime speedup have been performed on SWI-Prolog 5.12.3 with default settings (without any options). The testing machine had following parameters: Processor: Intel Pentium 4 - 3.4GHz, RAM: 2 GB, Operating system: Debian GNU/Linux system with kernel 2.6.14.3. The results are shown in the table.

[5] In the table this program is marked as none transformation.
[6] In the table this program is marked as UDF (Unfold-Definition-Fold) transformation.
[7] The exact definition of number of inferences can be found in the manual of SWI-Prolog. It roughly corresponds to the number of resolution steps.

Alg.		rotate_leftdepth	frontier	ackermann	inorder	rotate	tree_sum_size	permutation	fibb	doubleapp	applast	advisor	depth	ex_depth	goat	flip	grammar	model_elim	transpose	relative
none	# clauses	8	6	5	6	5	11	6	7	5	7	29	11	12	13	5	50	20	8	17
	# infs.	28003072	22003072	24343072	54003072	29103072	36503072	32884472	31603072	15004172	26403072	26003072	27003072	40603072	42074172	29603072	10314072	19603072	24403072	28203072
	Time [s]	8.28	7.02	10.01	18.01	8.2	11.3	21.65	10.85	6.06	10.01	11.68	12.98	16.98	18.68	10.61	3.38	10.34	9.13	6.78
	Inf. speedup	1.00	1.00	1.00	1.00	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
	T. speedup	1.00	1.00	1.00	1.00	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
ECCE	# clauses	10	6	13	6	8	20	8	13	8	6	31	15	26	110	8	48	28	8	42
	# infs.	25003072	23003072	23783072	56003072	20723072	32503072	21924572	26553072	10004172	13603072	20003072	16003072	17003072	18124172	23603072	10314072	9303072	24803072	4403072
	Time [s]	7.15	7.2	8.12	18.43	6.54	10.33	16.55	8.21	4.27	4.07	8.92	7.06	8.01	8.47	8.84	3.44	3.33	9.23	1.02
	Inf. speedup	1.12	0.957	1.024	0.964	1.404	1.123	1.5	1.19	1.5	1.941	1.3	1.687	2.388	2.321	1.254	1	2.107	0.984	6.405
	T. speedup	1.158	0.975	1.233	0.977	1.254	1.094	1.308	1.322	1.419	2.459	1.309	1.839	2.12	2.205	1.2	0.983	3.105	0.989	6.647
UDF	# clauses	10	11	8	9	10	17	7	10	6	8	56	40	19	24	6	237	31	11	53
	# infs.	10003072	17003072	17633072	26003072	12003072	30503072	25956372	24753072	10004072	25803072	16003072	14003072	21603072	31054172	14403072	4434072	12303072	18803072	24303072
	Time [s]	3	6.12	7.74	12.52	3.8	10.88	19.04	8.91	4.93	9.76	9.54	9.67	15.37	17.58	6.08	3.44	5.88	8.04	5.82
	Inf. speedup	2.799	1.294	1.381	2.077	2.425	1.197	1.267	1.277	1.5	1.023	1.625	1.928	1.88	1.355	2.055	2.326	1.593	1.298	1.16
	T. speedup	2.76	1.147	1.293	1.438	2.158	1.039	1.137	1.218	1.229	1.026	1.224	1.342	1.105	1.063	1.745	0.983	1.759	1.136	1.165
UDF+ECCE	# clauses	8	14	36	11	16	110	9	18	6	6	38	185	105	105	4	488	55	17	57
	# infs.	11003072	18003072	16563072	23003072	8103072	24503072	19028372	17203072	10004172	13603072	20003072	15003072	15203072	13044172	16003072	2744072	5203072	11603072	5103072
	Time [s]	3.06	6.34	6.28	10.12	2.58	9.45	15.74	5.86	4.29	4.14	8.76	6.8	8.7	5.5	6.19	1.67	2.29	5.59	1.25
	Inf. speedup	2.545	1.222	1.47	2.348	3.592	1.49	1.728	1.837	1.5	1.941	1.3	1.8	2.671	3.226	1.85	3.759	3.768	2.103	5.527
	T. speedup	2.706	1.107	1.594	1.78	3.178	1.196	1.375	1.852	1.413	2.418	1.333	1.909	1.952	3.396	1.714	2.024	4.515	1.633	5.424



As it can be seen from the table, the speed (inference speedup and CPU runtime speedup) of UDF on the tested program increased after transformation in most cases. In many cases the speed was almost the same and in only one case the speed of UDF was considerably lower than that of the original Prolog program. The average inference speedup on tested programs was approximately 69% (ECCE had 67%). Average CPU runtime speedup was approximately 31% (ECCE had 72%).

On one hand the measurement of CPU runtime is not ideal because the measured speedup can differ on various machines with the same Prolog system and the same operating system up to $\pm 10\%$[7]. When a different Prolog system is used, the situation is even worse. But on the other hand, such measurements represent the real run of the tested program while the number of inferences is a more theoretical measure.

As it can be seen from the table, there are many cases where both methods ECCE and UDF were comparable (small negative differences of UDF are caused by a much better post-processing of ECCE). But there are some cases where UDF method was better and vice versa. The UDF was significantly better than ECCE in seven cases, significantly worse than ECCE in seven cases and almost equivalent with ECCE in five cases. These experimental results may indicate that both methods are orthogonal.

The logical conclusion of the previous results was to combine both methods together. The result is very interesting and it shows that the combination of both methods produces almost always significantly better results (in comparison with previous two methods). The average inference speedup of the combined method on tested programs was approximately 136% and average CPU runtime speedup was approximately 111% (both numbers are compared with the original Prolog programs).

The size of programs transformed by UDF did not increase more than 6 times, the average code size coefficient of all tested programs was approximately 2.4 times in comparison with the original Prolog programs. The ECCE had in most cases less number of clauses because of its better post-processing such as `Dead Code Elimination`. The combination did not increased the code size more than 11 times.

We have to say that it was very surprising for us that the first attempt of fully automated Unfold/Fold transformation method is quite comparable to the most advanced CPD ECCE system. Moreover, more than one half of test programs belonged to the DPPD library which was originally designed for testing ECCE, and even for these examples UDF had quite comparable results. The combination of both methods gave significantly better results than each method applied individually. Moreover, this combination represents, to our best knowledge, the most effective transformation program for definite logic programs.

6 Conclusions and Future Work

In the paper we have described a fully automated algorithm for eliminating unnecessary variables in definite programs. We have implemented this algorithm and applied it to several benchmarks of programs. The results showed that the transformed programs have approximately 31% speedup (measured in CPU runtime) compared to non-transformed originals.

But there still exist programs for which UDF has worse results after transformation than their originals. In the future work we would like to identify the class of programs that have worse results and improve the behaviour of the algorithm on these programs. We believe that this method has a comparable asymptotic time complexity as Conjunctive Partial Deduction presented in [4,8] since we used a similar Homeomorphic Embedding termination control mechanism. The tranformation times of UDF did never exceed the times of ECCE on the benchmarks. To describe the exact time complexity is out of the scope of this paper.

We also presented a new algorithm that simply combines UDF and CPD methods and produces significantly better results than any of them. The average CPU runtime speedup of the combined method is 111% and this algorithm seems to be the most effective transformation algorithm for definite logic programs.

As the described algorithm is fully automated, it would be valuable to extend it to all general logic programs (and possibly to full Prolog).

References

1. Apt, K.R.: *From Logic Programming to Prolog.* Prentice Hall, 1996
2. Demoen, B.: *On the transformation of a Prolog program to a more efficient binary program.* in: K.-K.Lau and Clement, T. (Eds) Proceedings of LOPSTR 92, International Workshop on Logic Program Synthesis and Transformation, University of Manchester, Workshops in Computing series, 1992, pp. 242 - 252. Springer-Verlag
3. Dershowitz, N.: *Termination in rewriting*, Journal of Symbolic Computation, 3, 1987, pp. 69 - 116
4. De Schreye, D., Glück, R., Jørgensen J., Leuschel M., Martens B., Sørensen M., H.: *Conjunctive partial deduction: foundations, control, algorithms and experiments.* The Journal of Logic Programming 41 (1999), pp. 231 - 277
5. Glück, R., Jørgensen J., Martens B., Sørensen H.,M. : *Controlling Conjunctive Partial Deduction*, in: Programming Languages, Implementation, Logic and Programs, (Kuchen, E. and Swiestra, D., editors), LNCS Springer-Verlag 1996, pp. 137 - 151
6. Komorowski, H. J.: *A Specification of an Abstract Prolog Machine and its Application to Partial Evaluation.* Technical Report LSST 69, Linköpping University, 1981
7. Kruskal, J. B.: *Well-quasi-ordering, the Tree Theorem, and Varsonyi's conjecture*, Trans.Amer. Math. Society 95 (1960), pp. 210 - 225
8. Leuschel, M.: *Improving Homeomorphic Embeddings for On line Termination*, in: Logic Program Synthesis and Transformation, Proc. of LOPSTR'98, LNCS 1559, P. Flener (editor) Springer-Verlag, 1998, pp. 199 - 218

9. Leuschel, M.: *Ecce partial deduction system*, www.ecs.soton.ac.uk/~mal/systems/ecce.html
10. Nash-Williams, C.: *On well quasi ordering finite trees*, Proc. Camb. Phil. Society, 59 (1963), pp. 833 - 835
11. Proietti, M. and Pettorossi, A.: *Unfolding-definition-folding, in this order, for avoiding unnecessary variables in logic programs*, Theoretical Computer Science 142 (1995), pp. 89 - 124
12. Pettorossi, A. and Proietti M.: *Program specialization via algorithmic unfold/fold transformations*, ACM Computing Surveys, 30(3es), (September 1998).
13. Prestwich, S. D.: *The PADDY partial deduction system*, Technical Report ECRC92 -6, ECRC, Munich, Germany (1992)
14. Sahlin, D.: *Mixtus: an automatic partial evaluator for full Prolog*, New Generation Computing, 12(1):7-51, (1993)
15. Tamaki, H., and Sato, T.: *Unfold/Fold Transformation of Logic Programs*, in: Proc. of ICLP 84, Uppsala University, Sweeden, 1984, pp. 127 - 138
16. Vyskočil, J., Štěpánek, P., M. Halama: *Speedup by Fully Automated Unfold/Fold Transformation.* Technical Report TR No 2006/1 Department of KTIML MFF, Charles University in Prague.

Towards a normal form for Mercury programs

Wim Vanhoof and François Degrave

University of Namur,
Faculty of Computer Science,
Rue Grangagnage 21, B-5000 Namur, Belgium
email:{fde,wva}@info.fundp.ac.be

Abstract. In this work in progress we define a program transformation that normalises a Mercury program by reordering clauses, body goals, and possibly predicate arguments. The transformation, which preserves the well-modedness and determinism characteristics of the program, aims at reducing the complexity of performing a search for duplicated or similar code fragments between programs. In previous work, we have defined an analysis that searches for such duplicated functionality basically by pairwise comparing atoms and goals. While feasible in theory, the number of permutations to perform during the search renders it hard if not impossible to use in practice. We conjecture that the transformation to normal form, defined in this work, allows to substantially reduce the number of permutations, and hence the complexity of the search.

1 Introduction and motivation

The problem of deciding whether two code fragments are equivalent, in the sense that they implement the same functionality, is well-known to be undecidable. Nevertheless, there seems to be an interest in developing analyses that are capable to detect such equivalence under particular circumstances and within a certain error margin [3, 1, 11]. Applications can be found in plagiarism detection and tools for program refactoring. Work in this area can be based on parametrised string matching, an example being the MOSS system [7], or perform a more involved analysis on a graph representation of a program [2, 12]. Most of these latter works, including the more recent [10], concentrate on finding behavioral *differences* between strongly related programs and are often limited to (subsets of) imperative programs.

In a recent work [9], we have studied the conditions under which two (fragments of) logic programs can be considered equivalent. The main motivation of that work was to develop an analysis capable of detecting program fragments that are susceptible for refactoring, aiming in particular to the removal of duplicated code or to the generalisation of two related predicates into a new (higher-order) one. The basic idea is as follows: two code fragments (be they goals, clauses or complete predicate definitions) are equivalent if they are *isomorphic* in the sense that one can be considered to be a renaming of the other

modulo a permutation of the body goals and the arguments of the predicate. Take for example the definitions of `app1` and `conc1` below:

```
app1([],Y,Y).
app1([Xe|Xs],Y,[XN|Zs]):- XN is Xe + 1, app1(Xs,Y,Zs).

conc1(A,[],A).
conc1([NB|As],[Be|Bs],C):- conc1(As,Bs,C), NB is Be + 1.
```

Both definitions basically implement the same ternary relation in which one argument is the result of concatenating both other arguments and incrementing each element by one. This can easily be deduced from the source code, since the definition of `conc1` can be obtained from that of `app1` by variable renaming, goal reordering and a permutation of the argument positions. Note that our notion of equivalence is limited to the syntactical equivalence of predicates. Other characteristics like computational complexity etc. are not taken into account. In [9] we have defined an analysis that basically searches for such isomorphisms between each possible pair of subgoals in a given program. While the analysis *can* be used to search for duplication within two predicate definitions, its complexity – mainly due to the fact that one needs to consider every possible permutation of the predicate's body atoms – renders it hard if not impossible to use in practice.

This abstract reports on work in progress motivated by the desire to port the concepts and the analysis of [9] to the functional/logic programming language Mercury while, at the same time, rendering the analysis more practical. The basic idea is define a program transformation that reorders clauses, body atoms and possibly predicate arguments in a *unique* and predefined way such that 1) the operational characteristics (well-modedness and determinism) of the program remain unchanged, but 2) the number of permutations to peform during predicate comparison is substantially reduced.

2 Mercury programs in normal form

Mercury [8] is a strongly typed and moded functional/logic programming language. Although it is an expressive and syntactically rich language, its core syntax can be defined as follows:

Definition 1.

$$\begin{aligned} Goal ::= {} & Y = X \mid Y = f(\overline{X}) \mid Y = p(\overline{X}) \mid Z = Y(\overline{X}) \mid p(\overline{X}) \mid Y(\overline{X}) \mid \\ & (G_1, \ldots, G_n) \mid (G_1; \ldots; G_n) \mid not(G) \mid if(G_1, G_2, G_3) \mid \\ & \exists \overline{X} G \mid \forall \overline{X} G \mid true \mid fail \\ Pred ::= {} & p(\overline{X}) \texttt{:-}\ G. \end{aligned}$$

where G, $G_i (\forall i) \in Goal$, and X, Y, Z represent variables, $\overline{X}$ a sequence of distinct variables, and f and p respectively a functor and predicate symbol.

The syntax defined in Definition 1 corresponds to the so-called superhomogeneous form, which is an intermediate form used by the Mercury compiler. It defines a program as a set of predicate definitions. Each such predicate definition consists of a single clause (usually a disjunction) in which the arguments in the head of the clause and in predicate calls in the body are all distinct variables. Explicit unifications are generated for these variables in the body, and complex unifications are broken down into several simpler ones. Among these unifications we differ between term construction and matching

($X = Y$ and $X = f(\overline{Y})$) on the one hand and closure construction ($Y = p(\overline{X})$ and $Z = Y(\overline{X})$) on the other. Other goals include first-order and higher-order predicate calls ($p(\overline{X})$ and $Y(\overline{X})$ respectively), conjunction, disjunction, negation, if-then-else, existential and universal quantification and the special goals *true* and *fail*.

The full Mercury language contains a number of aditionnal constructs, such as function definitions, record syntax, state variables, DCG notation, etc. [5]. However, each of these constructions can be translated into the above syntax by introducing new predicates, adding arguments to existing predicates and introducing new unifications [5]. Note that these transformations are in principle reversible.

Example 1. Let us reconsider the example from above, this time in superhomogeneous Mercury syntax:

```
app1(X,Y,Z):- ( X=[], Z=Y ;
                X=[Xe|Xs], Z=[XN|Zs], E=1, XN=(Xe + E), app1(Xs,Y,Zs)).

conc1(A,B,C):- ( B=[], A=C ;
                 A=[NB|As], B=[Be|Bs], conc1(As,Bs,C), E=1, NB=(Be + E)).
```

From a programmer's point of view, the order in which the individual goals in a conjunction are written is of no importance. While this is one of the main characteristics that makes the language more declarative than other (logic) programming languages, it clearly renders the search for code isomorphisms in the sense outlined above even more dependent on the need to consider all permutations of the goals within a conjunction.

The fact that Mercury is a strongly moded language provides us with a starting point for our transformation into normal form. In Mercury, each predicate has an associated mode declaration[1] that classifies each argument as either input to the call, denoted by `in` (the argument is a ground term before and after the call) or output by the call which is denoted by `out` (the argument is a free variable that will be instantiated to a ground term at the end of the call). Given a predicate's mode declaration it is possible to derive how the instantiation of each variable changes over the execution of each individual goal in the predicate's body. In what follows we will use $\mathbf{in}(G)$ and $\mathbf{out}(G)$ to denote, for a goal G, the set of its input, respectively output variables. As such $\mathbf{in}(G)$ refers to the variables whose values are *consumed* by the goal G, whereas $\mathbf{out}(G)$ refers to the variables whose values are *produced* by G.

Example 2. If we consider the `app1` predicate (Example 1) for the mode `app1(in,in,out)` – reflecting the fact that the two first arguments are considered input whereas the third is considered output – we have:

G	$\mathbf{in}(G)$	$\mathbf{out}(G)$
`X = []`	$\{X\}$	$\emptyset$
`Z = Y`	$\{Y\}$	$\{Z\}$
`X = [Xe|Xs]`	$\{X\}$	$\{Xe, Xs\}$
`E = 1`	$\emptyset$	$\{E\}$

G	$\mathbf{in}(G)$	$\mathbf{out}(G)$
`XN = Xe + E`	$\{Xe, E\}$	$\{XN\}$
`app1(Xs,Y,Zs)`	$\{Xs, Y\}$	$\{Zs\}$
`Z = [XN|Zs]`	$\{XN, Zs\}$	$\{Z\}$

In order to be accepted by the compiler, Mercury programs must be *well-moded*. Intuitively, this means that the goals in a predicate's body can be rearranged in such

[1] In general, a predicate may have more than one mode declaration, but these can easily be converted into separate predicate (or, in Mercury terminology, *procedure* definitions.

a way that values are produced before they are consumed when the predicate is executed by a left-to-right selection rule [6]. In case of a conjunction, the well-modedness constraint could be formalised as follows:

Definition 2. *A conjunction* $(G_1, ..., G_n)$ *verifies the well-modedness constraint if*

$$\forall 1 \leq i \leq n, \forall k > i : \mathbf{in}(G_i) \cap \mathbf{out}(G_k) = \emptyset.$$

Example 3. The second disjunct of the `app1` definition in Example 1 does not verify the well-modedness constraint since the goal `Z=[Xn|Zs]` consumes variables `Xn` and `Zs`, which are both produced by goals further to the right in the conjunction. However, the following reordering *does*:

```
app1(X,Y,Z):- ( X=[], Z=Y ;
                X=[Xe|Xs], E=1, XN=(Xe + E), app1(Xs,Y,Zs), Z=[XN|Zs]).
```

It is the task of the compiler to rearrange conjunctions in a program such that they verify the well-modedness constraint, thanks to the information provided by the mode analyser. Note however that well-modedness in itself does not suffice to obtain a *unique* reordering. In the example above one could, e.g. switch the atoms `XN=(Xe + E)` and `app1(Xs,Y,Zs)` while the conjunction would remain well-moded. Consequently, well-modedness can be used as a starting point for our normalisation, but it needs to be further constrained in order to obtain a unique reordering. As a first step, the following definition rephrases and reinforces the well-modedness constraint.

Definition 3. *We define a* proper rearrangement *of a conjunction* $G_1, \ldots, G_n$ *to be a sequence of multisets* $\langle S_1, \ldots, S_k \rangle$ *such that*

$$\bigcup_{i \in \{1,\ldots,k\}} S_i = \{G_1, \ldots, G_n\}$$

and such that $\forall S_i$ *we have*

1. $\forall G, G' \in S_i : \mathbf{in}(G) \cap \mathbf{out}(G') = \emptyset$.
2. $\forall G \in S_i, \forall G' \in S_k$ *for* $k > i : \mathbf{in}(G) \cap \mathbf{out}(G') = \emptyset$.
3. $\forall G \in S_i,\ i > 0 : \exists G' \in S_{i-1} : \mathbf{in}(G) \cap \mathbf{out}(G') \neq \emptyset$.

Intuitively, a conjunction is properly arranged if its components can be partitioned into a sequence of sets of goals such that: (1) there are no dataflow dependencies between the goals in a single set; (2) a goal belonging to a set S_i does not consume values that are produced by a goal belonging to a set S_k that is placed *after* S_i in the sequence; and (3) each goal in a set S_i consumes at least one value that was produced by a goal placed in the previous set S_{i-1}. There are two main points of difference between our notion of a proper arrangement and that of well-modedness. First, we impose an order between *sets* of independent goals and, secondly and more importantly, consumers are pushed forward in the sequence as much as possible.

Example 4. Consider the definition of `app1` of Example 1. We have that

$$\langle \{\mathtt{X} = [], \mathtt{Z} = \mathtt{Y}\} \rangle$$

is a proper rearrangement of the first disjunct, whereas

$$\langle \{\mathtt{X} = [\mathtt{Xe}|\mathtt{Xs}], \mathtt{E} = 1\}, \{\mathtt{XN} = (\mathtt{Xe} + \mathtt{E}), \mathtt{app1}(\mathtt{Xs}, \mathtt{Y}, \mathtt{Zs})\}, \{\mathtt{Z} = [\mathtt{XN}|\mathtt{Zs}]\} \rangle$$

is a proper rearrangement of the second disjunct.

Note that there always exists a proper rearrangement of a well-moded conjunction. Also note that the required partitioning into sets is unique. Intuitively, this means that conjunctions that would classify as being isomorphic have similar proper rearrangements and, consequently, that a search for isomorphisms can be limited to a pairwise comparison of the corresponding sets of goals.

Example 5.

As such our notion of proper rearrangement seems a good starting point for a transformation that aims at rearranging predicate definitions in a unique way. All that remains, is to impose an order on the goals within the individual sets of a proper rearrangement. Since these goals share no dataflow dependencies, we can use any order without influencing well-modedness. We choose lexicographic ordering on goals in tree representation. Formally:

Definition 4. *Given a goal G, we define its tree representation, denoted $tr(G)$ as a tree over strings defined as follows:*

$$\begin{array}{ll} tr((G_1,\ldots,G_n)) = (conj, tr(G_1),\ldots,tr(G_n)) & tr(\exists \overline{X} G) = (\exists, \overline{X}, tr(G)) \\ tr((G_1;\ldots;G_n)) = (disj, tr(G_1),\ldots,tr(G_n)) & tr(\forall \overline{X} G) = (\forall, \overline{X}, tr(G)) \\ tr(if(G_1,G_2,G_3)) = (if, tr(G_1), tr(G_2), tr(G_3)) & tr(true) = (true) \\ tr(not(G)) = (not, tr(G)) & tr(fail) = (fail) \end{array}$$

$$\begin{array}{ll} tr(Y = X) = (unifv, Y, X) & tr(Z = Y(\overline{X})) = (closv, Y, Z, \overline{X}) \\ tr(Y = f(\overline{X})) = (unifc, f, Y, \overline{X}) & tr(Y = p(\overline{X})) = (closc, p, Y, \overline{X}) \\ tr(p(\overline{X}) = (call, p, \overline{X}) & tr(Y(\overline{X}) = (hocall, Y, \overline{X}) \end{array}$$

Given two goals G and G', we will write $G < G'$ if and only if $tr(G) <_l tr(G')$ where $<_l$ represents the lexicographic ordering over trees of strings.

Example 6. Since $tr(\texttt{X} = []) = (unifc, [], X)$ and $tr(\texttt{Z} = \texttt{Y}) = (unifv, Z, Y)$ We have $\texttt{X} = [] < \texttt{Z} = \texttt{Y}$. Likewise, one can easily verify that we have $\texttt{X} = [\texttt{Xe}|\texttt{Xs}] < \texttt{E} = \texttt{1}$ and $\texttt{app1}(\texttt{Xs}, \texttt{Y}, \texttt{Zs}) < \texttt{XN} = (\texttt{Xe} + \texttt{E})$.

Proper rearrangement and lexicographic ordering are the main ingredients of our transformation to normal form, which we can now easily define as follows:

Definition 5. *The transformation of a goal G into normal form, denoted by $nf(G)$, is defined as follows:*

$nf(\exists \overline{X} G) = \exists \overline{X} nf(G) \quad nf(\forall \overline{X} G) = \forall \overline{X} nf(G)$
$nf(not(G)) = not(nf(G))$
$nf(if(G_1,G_2,G_3)) = if(nf(G_1), nf(G_2), nf(G_3))$
$nf((G_1,\ldots,G_n)) = C$ *where*
$\quad C = (nf(G_1^1),\ldots,nf(G_{l_1}^1), nf(G_1^2),\ldots,nf(G_{l_2}^2),\ldots,nf(G_1^k),\ldots,nf(G_{l_k}^k))$
$\quad$ *if* $\langle S_1,\ldots,S_k\rangle$ *is a proper arrangement of* $(G_1,\ldots,G_n)$.
$\quad$ *and* $\forall 1 \le i \le k : S_i = \{G_1^i,\ldots,G_{l_i}^i\}$ *and* $\forall 1 \le j < l_i : G_j^i \le G_{j+1}^i$
$nf((G_1;\ldots;G_n)) = (nf(G_1');\ldots;nf(G_n'))$
$\quad$ *where* $(G_1';\ldots;G_n')$ *is a reordering of* $(G_1;\ldots;G_n)$
$\quad$ *such that* $\forall i : 1 \le i < n : G_i \le G_{i+1}$.
$nf(A) = A$ *for* $A \in Atom$

As can be seen from the definition above, the transformation to normal form imposes an ordering on the goals within a conjunction and on those within a disjunction. Since these orderings are unique, the transformation is well-defined.

Example 7. The `app1` and `conc1` predicates of Example 1 in normal form would look like

```
app1(X,Y,Z):- ( X=[], Z=Y ;
                X=[Xe|Xs], E=1, app1(Xs,Y,Zs), XN=(Xe + E), Z=[XN|Zs]).

conc1(A,B,C):- ( B=[], A=C ;
                 B=[Be|Bs], E=1, conc1(As,Bs,C), NB=(Be + E), A=[NB|As]).
```

The advantage of combining the notion of proper arrangement with the lexicographic ordering defined as above is that if there exists an isomorphism between two conjunctions (modulo a permutation of the body atoms), than there exists an isomorphism between between the corresponding goals in the respective normal forms. In other words, it seems that detecting duplication between predicates in normal form does not require to consider permutations of the conjunctions, thereby removing a layer of complexity.

3 Ongoing and further work

3.1 Normalising types and type definitions

Mercury is a strongly typed programming language, which offers the opportunity to search for equivalences between types based on isomorphisms between the type definitions, much in the same way as the analysis of [9] searches for equivalent predicates based on isomorphisms between the predicate definitions. Let us consider the following example:

```
:- type tag --> tagged ; untagged.
:- type data --> simple(int,tag) ; complex(int,float,tag).

:- type bool --> true ; false.
:- type val --> two(bool,float,int) ; one(bool,int).
```

Intuitively, it is clear that the types `tag` and `bool` serve the same purpose: being able to represent and differentiate between two values. Taking this equivalence between `tag` and `bool` into account, it becomes clear that also `data` and `val` are equivalent in the same sense: both are meant to represent either a "tagged" integer value, or a tagged combination of a float and an integer. It is clear that such equivalences between types could be detected by an analysis that performs a pairwise comparison of type definitions and, for each pair of definitions, a pairwise comparison between the different elemens of each definition – as such effectively computing whether one defintion can be considered a renaming of the other. As it is the case when comparing code fragments, such an analysis would profit from the fact that type definitions were somehow normalized such that less comparisons need to be performed.

Formally, types are constructed, like terms, from (type) variables and (type) symbols. A type containing variables is said to be *polymorphic*, otherwise it is *monomorphic*. We denote by T the set of types constructed from type variables V_t and type symbols Σ_t. The sets of type variables, type symbols, variables and function symbols are assumed to be disjoint. For each type symbol, a unique type rule associates that symbol with a finite set of function symbols, the set of which we denote by Σ.

Definition 6. *A* type rule *for a type symbol $h/k \in \Sigma_t$ is of the form*

$$h(\overline{V}) \longrightarrow f_1(\overline{\tau}_1)\ ;\ \ldots\ ;\ f_n(\overline{\tau}_n)$$

where $\overline{V}$ is an k-tuple of distinct type variables from V_t, $f_1, \ldots, f_n$ are distinct function symbols from Σ associated with the type symbol h, $\overline{\tau}_i$ $(1 \leq i \leq n)$ are tuples from T of the appropriate arity. Moreover, a type variable occuring on the right-hand side of a type rule must also occur on the left-hand side. Given the type rule above, we say that the type symbol h/k is defined by n clauses and we refer to $f_i(\overline{\tau}_i)$ as the i'th clause in the type rule. For a type symbol h, we denote by $c(h)$ the number of clauses in its type rule.

Example 8. The type rules given above define four monomorphic types: `tag`, `data`, `bool` and `val`. The type rule for `bool`, for example, associates the function symbols `true` and `false` with the type symbol `bool`.

In order to normalise type definitions, we need to define an order relation that allows to order the sequence of clauses comprising a type rule which, in turn, requires an order relation on the types themselfs. Let us start by defining an order relation on the set of types:

Definition 7. *Let $<_s$ denote the normal lexicographic ordering on strings; let h/n and h'/m be two type symbols from Σ_t and let $\tau_1, \ldots, \tau_n$ and $\tau'_1, \ldots, \tau'_m$ types from T. Then we define the order relation $<_t$ on the set of types, T, as follows:*

$$\begin{array}{ll}
X <_t h(\tau_1, \ldots, \tau_n) & \text{for } X \in V_T \\
X <_t Y & \text{if } X <_s Y \\
h(\tau_1, \ldots, \tau_n) <_t h'(\tau'_1, \ldots, \tau'_m) & \text{if either} \begin{cases} c(h) < c(h') \\ c(h) = c(h') \text{ and } h <_s h' \end{cases} \\
h(\tau_1, \ldots, \tau_n) <_t h(\tau'_1, \ldots, \tau'_n) & \text{if } \exists i : \tau_i <_t \tau'_i \text{ and } \forall j < i : \tau_j = \tau'_j
\end{array}$$

Note that in this ordering, a type variable is considered smaller than any type constructed from a type symbol and type variables are ordered themselfs lexicographically. Two types t_1 and t_2 having different outermost type symbols are ordered $t_1 < t_2$ if the outermost type symbol of t_1 is defined by less clauses than the outermost type symbol of t_2 or, in case they are defined by the same number of clauses, the former is lexicographically smaller than the latter. Two types having the same outermost type symbol are ordered by the lexicographic ordering on their arguments.

Example 9. Let `bool/0` and `val/0` be defined as above, we have `bool` $<_t$ `val`. Let `list/1` be a type symbol defined by the following rule:

```
list(T) --> [] ; [T|list(T)].
```

We have that `list(bool)` $<_t$ `list(val)`. With respect to the (predefined) types `int` and `float`, it seems reasonable to assume that we have `int` $<_t$ `float` and $t <$ `int` for any defined type t.

Note that $<_t$ allows to reorder the argument types in each clause of a type rule. To reorder the clauses themselfs, we need yet another order, which we basically define as the lexicographic order on tuples of types, without taking the function symbols into account.

Definition 8. *Let f and g be function symbols in Σ and let $\tau_1, \ldots, \tau_n$ and $\tau'_1, \ldots, \tau'_m$ types from T. Then we define the order relation $<_c$ as follows:*

$$f(\tau_1, \ldots, \tau_n) <_c g(\tau'_1, \ldots, \tau'_m) \text{ if } m > n$$
$$f(\tau_1, \ldots, \tau_n) <_c g(\tau'_1, \ldots, \tau'_n) \text{ if } \exists i : \tau_i <_t \tau'_i \text{ and } \forall j < i : \tau_j = \tau'_j$$

The combination of $<_t$ and $<_c$ allows to normalise a type rule, using $<_t$ to order the argument types in each clause, and $<_c$ to order the clauses themselfs.

Example 10. Reconsider the type symbols `tag/0`, `data/0`, `bool/0` and `val/0`. Normalising their type rules would give us

```
:- type tag --> tagged ; untagged.
:- type data --> simple(tag,int) ; complex(tag,int,float).

:- type bool --> false ; true.
:- type val --> one(bool,int) ; two(bool,int,float).
```

As the above example suggests, a pairwise compairison of type rules need no longer consider a permutation of the clauses in the definitions, nor a permutation of of the argument types in each clause. This should allow for a substantial speedup when searching for type equivalence. We are currently investigating if and how these results cfan be formally stated.

Also note that the order $<_t$ between types could be used to normalise the sequence of arguments in a predicate definition. While this might be beneficial when surching for equivalence between predicate definitions, the influence of such argument reordering on closure construction must be investigated.

3.2 Other issues

Ongoing research include reformulating the analysis of [9] in the context of normalised Mercury programs. It must also be investigated if and how the proposed transformation to normal form can be reversed, such that the results of the analysis (indications of what code fragments are equivalent) can be displayed on the *original* source code rather than the normalised code.

Acknowledgments

We thank anonymous referees for their constructive comments and feedback.

References

1. X. Chen, B. Francia, M. Li, B. McKinnon, and A. Seker. Shared information and program plagiarism detection. *IEEE Transactions on Information Theory*, 50(7):1545–1551, 2004.
2. Susan Horwitz. Identifying the semantic and textual differences between two versions of a program. *ACM SIGPLAN Notices*, 25(6):234–245, 1990.
3. K. A. Kontogiannis, R. Demori, E. Merlo, M. Galler, and M. Bernstein. Pattern matching for clone and concept detection. *Reverse engineering*, pages 77–108, 1996.

4. A. Mycroft and R.A. O'Keefe. A polymorphic type system for Prolog. *Artificial Intelligence*, 23:295–307, 1984.
5. Univ. of Melbourne. Mercury language reference manual. 2006.
6. D. Overton, Z. Somogyi, and P. Stuckey. Constraint-based mode analysis of Mercury. In *Proceedings of the 4th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 109–120, New York, NY, USA, 2002. ACM Press.
7. S. Schleimer, D.S. Wilkerson, and A. Aiken. Winnowing: local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of Data*, San Diego, CA, 2003.
8. Z. Somogyi, H. Henderson, and T. Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29(1), 1996.
9. W. Vanhoof. Searching semantically equivalent code fragments in logic programs. In S. Etalle, editor, *Proceedings of LOPSTR 2004*, volume 3573 of *Lecture Notes in Computer Science*, pages 1–18. Springer-Verlag, 2005.
10. J. Winstead and D. Evans. Towards differential program analysis. In *Proceedings of the 2003 Workshop on Dynamic Analysis*, 2003.
11. Wise. YAP3: Improved detection of similarities in computer program and other texts. *SIGCSEB: SIGCSE Bulletin (ACM Special Interest Group on Computer Science Education)*, 28, 1996.
12. Wuu Yang. Identifying syntactic differences between two programs. *Software Practice and Experience*, 21(7):739–755, 1991.
13. Wuu Yang, Susan Horwitz, and Thomas Reps. Detecting program components with equivalent behaviors. Technical Report CS-TR-1989-840, University of Wisconsin, Madison, 1989.

Aggregates for CHR through Program Transformation

Peter Van Weert*, Jon Sneyers**, and Bart Demoen

Department of Computer Science, K.U.Leuven, Belgium
FirstName.LastName@cs.kuleuven.be

Abstract. We propose an extension of Constraint Handling Rules (CHR) with aggregates such as `sum`, `count`, `findall`, and `min`. This new feature significantly improves the conciseness and expressiveness of the language. In this paper, we describe an implementation based on source-to-source transformations to CHR (extended with some low-level compiler directives). We allow user-defined aggregates and nested aggregate expressions over arbitrary guarded conjunctions of constraints. Both an on-demand and an incremental aggregate computation strategy are supported.

1 Introduction

Constraint Handling Rules (CHR) [1, 4] is a powerful, elegant committed-choice CLP language, based on multi-headed, guarded multiset rewrite rules. Originally designed for the implementation of constraint solvers, CHR has matured towards a general purpose language, used in a wide range of application domains, including natural language processing, multi-agent systems, and type system design.

In [8, 9] we proposed an extension of CHR with *aggregates*. This declarative language feature allows the aggregation of information from an unbounded number of constraints to be captured concisely in a single expression in the head of a CHR rule. Example aggregates are `sum`, `count`, `findall`, and `min`. Without language support for aggregates, these common programming idioms require cumbersome, low-level auxiliary constructs, cross-cutting the entire program. Case studies show aggregates reduce program size by up to 50%. The resulting programs are also significantly more understandable, maintainable, and robust.

This paper presents how existing CHR systems can be extended with a general, extensible aggregate framework using source-to-source transformations to lower-level CHR. Only a small number of easily implemented low-level compiler directives have to be added to the CHR system itself. The transformation takes care of introducing auxiliary and cross-cutting code, not unlike an aspect weaver in Aspect-Oriented Programming [5].

The source-to-source transformation schemes presented in this paper support user-defined, application-tailored aggregates, nested aggregate expressions, and

* Research Assistant of the Research Foundation – Flanders (FWO-Vlaanderen).
** Research funded by a Ph.D. grant of the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen).

efficient aggregate computation using either on-demand or incremental aggregate computation. The design of these non-trivial transformation schemes is discussed in detail, the different issues identified and addressed one by one.

Overview. Section 2 briefly recalls the syntax and operational semantics of CHR. More information can be found in [3, 4]. Section 3 motivates and introduces the extension of CHR with aggregates. Next, two different source-to-source schemes are presented in Section 4. The implementation approach is evaluated in Section 5. Finally, Section 6 provides conclusions and directions of future work.

2 Preliminaries: Constraint Handling Rules

2.1 Syntax of CHR

A constraint $c(x_1, \ldots, x_n)$ is an atom of predicate c/n, with all x_i values of a host language data type. Two types of constraints exist: *built-in constraints*, solved by an underlying solver, and *CHR constraints*, solved by the CHR program.

A CHR program consists of a sequence of CHR rules of the form:

$$name \ @ \ \ H_k \setminus H_r \iff G \mid B$$

The *name* is optional and unique; rules without a name get one implicitly. The *head* consists of two conjunctions of CHR constraints, H_k and H_r. Their conjuncts are called *occurrences* (*kept* and *removed occurrences* resp.). The *guard* G is a conjunction of built-in constraints. If "$G \mid$ " is omitted, it is considered to be "*true* $\mid$ ". The *body* B is a conjunction of CHR and built-in constraints.

There are three types of rules. If H_k is empty, the rule is a *simplification* rule. If H_r is empty, the rule is a *propagation* rule and the symbol " $\Longrightarrow$ " is used instead of " $\iff$ ". If both parts are non-empty, the rule is a *simpagation* rule. At least one of H_r and H_k must be non-empty.

Logically, a simplification rule corresponds to an equivalence: $G \rightarrow (H_r \leftrightarrow B)$, while a propagation rule corresponds to an implication: $G \rightarrow (H_k \rightarrow B)$.

2.2 Operaional Semantics of CHR

Informally, the operational semantics of a CHR rule is as follows: if for each occurrence in the head a matching constraint is found in the *constraint store*, and the guard is satisfied, then the rule *fires*: the constraints that matched the removed occurrences (H_r) are deleted from the store and the body is executed.

Formally, the execution of a CHR program follows the *theoretical* or *high-level* operational semantics, denoted as ω_t. For brevity, we do not present the formal transition rules of ω_t here; we refer to [3, 4] instead. A version of ω_t extended with aggregates is presented in Section 3.3.

The *theoretical* operational semantics is highly nondeterministic. Only programs that do not depend on the order of rule application have guaranteed

behavior under ω_t. Such programs are called *confluent* (cf. [4]). However, writing confluent programs is often overly difficult. Many programs are non-confluent under ω_t as CHR programmers exploit the execution strategy implemented by most CHR systems to obtain the desired behavior. The *refined operational semantics*, denoted with ω_r, instantiates ω_t to capture the behavior of most current systems. A complete exposition, including a formal description, is found in [3].

A central concept in the refined semantics is the *active constraint*. Each time a constraint becomes active, all CHR rules are tried in a *top-down textual order*, until all applicable rules that match the active constraint have been executed, or the active constraint is removed. If a rule fires, the constraints in its body are processed one at a time, in a *left-to-right textual order*. If a CHR constraint is processed, it is added to the constraint store and immediately becomes the new active constraint. Processing a built-in constraint entails solving it, and reactivating all CHR constraints whose arguments are affected, one at a time. The order in which CHR constraints are reactivated is undetermined. The activation and reactivation of a CHR constraint is treated as a *procedure call*: only when its execution is finished, the execution returns to the previous active constraint.

3 Extending CHR with Aggregates

As CHR is already Turing complete [7], aggregates do not add to the computational power of CHR. Section 3.1 shows they are nevertheless invaluable when it comes to expressiveness, maintainability and conciseness. The extension of CHR with aggregates is introduced in Section 3.2, and given a formal operational semantics in Section 3.3. A more thorough introduction to the proposed extension, more examples and case studies can be found in [8, 9].

3.1 Motivation and Running Example

As the head of each CHR rule only considers a fixed number of constraints, any form of aggregation over unbounded parts of the constraint store necessarily requires explicit encoding, using auxiliary constraints and rules. The following example clearly shows the inadequacy of such ad hoc approaches. It is also used as a running example throughout the paper.

Example 1. Suppose the constraints `account(AccountId,ClientId,Balance)` and `client(ClientId)` constitute a simplified representation of the accounts and clients of a bank. At some point, the bank decides to add the business rule:

> "A client whose accumulated sum of account balances is $25,000 or more is a *platinum client*"

As a client can have any number of accounts, this seemingly simple rule cannot be expressed straightforwardly in CHR. CHR practitioners therefore commonly introduce a constraint such as `accumulated_balance/2`. This allows the logic of platinum clients to be captured concisely in a single rule as follows:

```
client(C), accumulated_balance(C,Sum) ==> Sum ≥ 25000 | platinum(C).
```

This approach, however, also necessitates the explicit maintenance of the accumulated balance. This inherently cross-cutting concern requires invasive modifications to all rules that alter the balance of an account. The bank e.g. has to add at least the following underlined code:

```
deposit(A,X), account(A,C,B), accumulated_balance(C,Acc) <=>
        account(A,C,B+X), accumulated_balance(C,Acc+X).
...
withdraw(A,X), account(A,C,B), accumulated_balance(C,Acc) <=>
  B > X, account(A,C,B-X), accumulated_balance(C,Acc-X).
```

Many variations to the above maintenance scheme can be concocted, but they all require similar modifications scattered throughout the entire program. Similar auxiliary code has to be written for *every* aggregate; a very tedious and repetitive task. Clearly, this approach displays poor compliance with common software quality criteria: it is highly error-prone, and it impairs the readability and maintainability of the program, as the logic of many rules becomes tangled with obfuscating auxiliary code. In other words, many practical advantages of declarative programming – understandability, maintainability, robustness, and shortened development time – are severely handicapped.

3.2 An Extensible Framework for Aggregates in CHR

This section introduces an extension of CHR with aggregates, designed to overcome the expressivity problems outlined in the previous section. It allows rule heads to contain *aggregates*. These expressions accumulate information over possibly unbounded parts of the constraint store. Aggregates can be written in both the kept and the removed part of the head; there is no semantical difference.

This section provides a short summary on the proposed, extensible aggregate framework. More information can be found in [8, 9].

Predefined aggregates. Our framework provides a wide range of predefined aggregates, including all aggregates commonly found in related paradigms such as database query languages [10] (i.e. `min`, `max`, `sum`, `count` and `avg`) and production rule systems (i.e. `not`, `exists` and `forall`). A complete list of predefined aggregates, together with a number of example uses, can be found in [8, 9].

Example 2. Using the `sum` aggregate (in italics), the platinum client business rule of Example 1 is again declaratively expressed in a single rule:

```
client(C), sum(B,account(_,C,B),Sum) ==> Sum ≥ 25000 | platinum(C).
```

However, no further changes to the program are required, as the aggregate's semantics already guarantees the correct behavior implicitly: it accumulates the sum of the balances `B` of all matching `account/3` constraints, and ensures that the rule fires as soon as this sum, `Sum`, reaches 25,000.

Contrasting the above example with the approach outlined in Example 1 in the previous section clearly shows that aggregates render CHR programs more declarative, readable and maintainable. Relieved from the cumbersome and repetitive task of implementing aggregates, the programmer can focus exclusively on the application domain. So, productivity is improved considerably as well.

User-defined aggregates. Often information has to be aggregated in application-specific ways. Therefore, we designed a general high-level mechanism that enables CHR end-users to create *user-defined aggregates*:

```
aggregate(Start, Inc, Dec, Final, Template, Goal, Result)
```

The `aggregate/7` construct is expressive enough to effectively specify any aggregate. In fact, all predefined aggregates are also implemented by it.

Example 3. The predefined `sum(T,G,R)` aggregate for instance is specified as `aggregate(=(0),plus,minus,=,T,G,R)`, where '`=(0)`' indicates unification with zero, and `plus/3` and `minus/3` are two straightforward Prolog predicates computing the sum, respectively the difference of the first two arguments.

The first four arguments of `aggregate/7` specify the host language procedures or CHR constraints that determine how the aggregate is computed. First, an intermediate working value is *initialized* using `Start`. Then, for each matching found for `Goal`, a corresponding instance of `Template` is passed to `Inc` to *increment* the current working value. After all increments required are made, the working value is *finalized* using `Final`, to obtain the aggregate's result `Result`. The function of `Dec` is explained in Section 4.3.

These seven arguments thus completely determine an aggregate's semantics, as also reflected in the formal operational semantics presented in the next section.

Complex aggregate goals. Example 2 showed an aggregate over a simple `Goal`, i.e., consisting of a single CHR constraint. The *aggregate goal* `Goal` however can be an arbitrary conjunction of CHR constraints and guards: for example `count((platinum(C),account(_,C,_)), N)` counts the number of accounts owned by platinum clients. We further allow *nested aggregates*, that is, aggregate expressions inside the goal of another aggregate: for instance, `max(S, (client(C), sum(B,account(_,C,B),S)), M)` returns the largest total balance `M` of any individual client.

3.3 Formal Operational Semantics

We extend the theoretical operational semantics ω_t [3, 4] to deal with the general `aggregate/7` expressions introduced in the previous section (recall that this also covers all predefined aggregates). The extended semantics is denoted ω_a. We extend ω_t because of brevity, and because it allows more implementation freedom then extending a more deterministic instance such as e.g. ω_r (cf. Section 2.2).

The ω_a semantics is formulated as a state transition system. Transition rules define the relation between an execution state and its subsequent execution state.

Definition 1 (Identified constraints). *To differentiate amongst otherwise identical copies of constraints, CHR constraints are assigned unique identifiers. An* identified *CHR constraint with* constraint identifier i *is denoted* $c\#i$*. We further introduce the functions* $chr(c\#i) = c$ *and* $id(c\#i) = i$*, and extend them to sequences and sets of identified CHR constraints in the obvious manner.*

Definition 2 (Execution state). *An* execution state σ *is a tuple* $\langle \mathbb{G}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n$*. The* goal $\mathbb{G}$ *is a multiset of constraints. The CHR constraint* store $\mathbb{S}$ *is a set of identified CHR constraints (while* $\mathbb{S}$ *is a set,* $chr(\mathbb{S})$ *is a multiset). The* built-in constraint store $\mathbb{B}$ *is the conjunction of all built-in constraints passed to the underlying solver. The* propagation history $\mathbb{T}$*, necessary to prevent trivial non-termination, is a set of tuples, each recording the name of a rule and a sequence of identities of the CHR constraints that fired that rule. Finally, the counter* $n \in \mathbb{N}$ *represents the next unique constraint identifier.*

The semantics of the built-in constraints is determined by a constraint theory $\mathcal{D}_\mathcal{B}$. Let $vars(A)$ be the variables occurring freely in A, then $\bar{\exists}_A F$ denotes $\exists x_1, \ldots, \exists x_n F$, with $\{x_1, \ldots, x_n\} = vars(F) \backslash vars(A)$.

Because aggregates can be nested, we use two mutually recursive definitions:

Definition 3 (Matching substitutions). *Let* $\mathsf{matchings}(A \wedge H \wedge G, S_h, \mathbb{S}, \mathbb{B})$

$$= \left\{ \theta \;\middle|\; H = \theta(S_h) \wedge \mathcal{D}_\mathcal{B} \models \mathbb{B} \rightarrow \bar{\exists}_{\mathbb{B}}(\theta \wedge G \wedge \mathsf{agg\_cond}(A, S_h \cup \mathbb{S}, \mathbb{B})) \right\}$$

where H *and* S_h *are conjunctions of CHR constraints,* G *and* $\mathbb{B}$ *conjunctions of built-in constraints,* A *is a conjunction of aggregates, and* $\mathbb{S}$ *is a set of identified CHR constraints (a CHR store).*

Definition 4 (Aggregate Condition). *For an aggregate* A *of the form* $\texttt{aggregate}(s, i, d, f, T, G, R)$*, a CHR store* $\mathbb{S}$ *and a built-in store* $\mathbb{B}$*:*

$$\mathsf{agg\_cond}(A, \mathbb{S}, \mathbb{B}) = \quad s(V_0) \wedge \bigwedge_{k=1}^{n} i(V_{k-1}, \theta_k(T), V_k) \;\wedge\; f(V_n, R)$$

where $V_0, \ldots, V_n$ *are new variables and* $\{\theta_1, \ldots, \theta_n\} = \bigcup_{H \subseteq \mathbb{S}} \mathsf{matchings}'(G, H, \mathbb{S}, \mathbb{B})$*. The condition is extended to conjunctions of aggregates in the obvious manner.*

In its generic syntactic form (cf. Section 2.1), the head of a rule is prepended with a conjunction of aggregates A (recall that an aggregate's location in the head has no semantic meaning):

Definition 5 (Transition rules). *Given a CHR program* $\mathcal{P}$*, execution proceeds by exhaustively applying the following transition rules, starting from an* initial state *of the form* $\langle \mathbb{G}, \emptyset, true, \emptyset \rangle_1$*:*

1. Solve. $\langle \{c\} \uplus \mathbb{G}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \rightarrowtail_\mathcal{P} \langle \mathbb{G}, \mathbb{S}, c \wedge \mathbb{B}, \mathbb{T} \rangle_n$
where c *is a built-in constraint and* $\mathcal{D}_\mathcal{B} \models \bar{\exists}_\emptyset \mathbb{B}$*.*

2. Introduce. $\langle \{c\} \uplus \mathbb{G}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \rightarrowtail_\mathcal{P} \langle \mathbb{G}, \{c\#n\} \cup \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_{n+1}$
where c *is a CHR constraint and* $\mathcal{D}_\mathcal{B} \models \bar{\exists}_\emptyset \mathbb{B}$*.*

*3. **Apply.*** $\langle \mathbb{G}, H_1 \cup H_2 \cup \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \rightarrowtail_{\mathcal{P}} \langle B \uplus \mathbb{G}, H_1 \cup \mathbb{S}, \theta \wedge \mathbb{B}, \mathbb{T} \cup \{h\} \rangle_n$
where $\mathcal{D}_{\mathcal{B}} \models \bar{\exists}_{\emptyset} \mathbb{B}$ *and* $\mathcal{P}$ *contains a rule* $r \ @ \ A, H_1' \backslash H_2' \iff G \mid B$ *and* $\theta \in \mathsf{matchings}((A, H_1', H_2', G), H_1 \cup H_2, \mathbb{S}, \mathbb{B})$ *and* $h = (r, id(H_1, H_2)) \notin \mathbb{T}$

The propagation history does not record an aggregate in any way, so a rule is never fired more then once with the same combination of constraints, even if the aggregate's value changes. We call this *fire-once semantics*. More information regarding this choice can be found in [9].

4 Implementation through Program Transformation

The transformation schemes presented here improve earlier schemes described in [9]. Two different aggregate computation strategies are supported: *on-demand* (Section 4.2), and *incremental* (Section 4.3). The source-to-source transformations are implemented in the K.U.Leuven CHR system [6] in SWI-Prolog [12], but the approach is equally applicable to other systems implementing the refined operational semantics. The implementation is based on high-level *meta CHR rules*. Their basic syntax and semantics is outlined first in Section 4.1.

4.1 Meta CHR Rules

Meta CHR rules allow concise specification of CHR source-to-source transformations. They somewhat resemble ordinary CHR rules, both syntactically and semantically. Only, instead of rewriting constraint multisets, they rewrite the CHR rules of another CHR program, called the *object program*.

A meta rule is applicable if its head can be matched with occurrences in a single *object rule*. When a meta rule fires, the occurrences that matched its removed meta occurrences, are removed from the object rule. In a meta rule's body, the '`+`' prefix operator adds kept occurrences to the object rule, '`-`' adds removed occurrences, and '`?`' adds extra conjuncts to the object rule's guard. Writing a CHR rule in the body of a meta rule adds this rule to the object program. The `remaining_head/1` operation returns those occurrences of the object rule not matched by the meta rule, and `guard/1` returns the object rule's guard.

4.2 On-demand Aggregate Computation

This section gradually introduces and explains a transformation scheme for *on-demand aggregate computation*. The scheme, depicted in Figure 1, outputs code in which aggregates are computed from scratch each time they are required.

Lines 1 to 4. The simplification rule removes each occurrence of `aggregate`/7, and replaces it with a guard (line 4). This guard calls an auxiliary CHR constraint[1], $\mathtt{agg}_i$`/2`, that computes the aggregate's result `A`. The `shared_vars/3`

[1] Even though not actually allowed by the CHR language (cf. Section 2), several CHR implementations do support CHR constraints in guards this way, a feature often exploited by expert users. To properly support such guards though, a number of changes were required to the K.U.Leuven CHR compiler ([9] provides an overview).

```
1  aggregate(Start,Inc,_,Final,T,G,A) <=>
2    new_unique_identifier(i),
3    remaining_head(Head), shared_vars(Head, (T-G), V),
4    ?agg_i(V,A),
5    +G#on_active, +G#on_removal,
6    ( agg_i(V,A) <=> Start(I), result_i(I), match_i(V), get_i(R), Final(R,A) ),
7    ( match_i(V), G ==> incr_i(T) ),
8    ( incr_i(T), result_i(R_1) <=> Inc(R_1, T, R_2), result_i(R_2) ),
9    ( result_i(R), match_i(_), get_i(Q) <=> Q = R ).
```

Fig. 1. The core of a transformation scheme using on-demand aggregate computation. For compactness, pseudo code is used. The function of each line is explained below.

predicate returns the variables shared by its first two arguments. It is used to compute `V`, the list of all variables required to compute the aggregate (line 3–4). The implementation of the aggregate computation (lines 6–9) is discussed below. The identifier i (line 2) ensures all auxiliary functors, such as $\texttt{agg}_i\texttt{/2}$, are unique.

Line 5 (`on_active` and `on_removal` heads). Under the refined operational semantics, by default, the guard added on line 4 is called each time a matching is found for the remaining occurrences. In general, this does not suffice: the aggregate also has to be (re)computed when its outcome changes. To indicate such extra conditions under which a rule, and thus its guard, have to be (re)considered, we introduced two special types of heads: `on_active` heads and `on_removal` heads. An *`on_active` head* indicates an additional trigger to fire the rule: when constraints matching the `on_active` head are *activated* (i.e. newly added or reactivated, cf. Section 2.2), the rule is tried. Similarly, an *`on_removal` head* indicates that the rule additionally has to be tried when constraints matching the `on_removal` head are *removed* from the constraint store. Neither of these types of heads is considered when an occurrence in the regular head is active. Both new types of heads are implemented with a straightforward source-to-source transformation. More information can be found in [9].

Line 5 adds the aggregate's goal `G` to the original object rule, both as an `on_active` and as a `on_removal` head. This ensures the guard computing the aggregate is called, not only when the remainder of the original head is matched, but also when constraints matching `G` are added, reactivated, or removed.

Example 4. The rule from Example 2 (Section 3.2) becomes:

```
account(_,C,_)#on_active, account(_,C,_)#on_removal,
     client(C) ==> agg_0(C,Sum), Sum ≥ 25000 | platinum(C).
```

A client's accumulated balance is thus also (re)computed when the accumulated balance changes, i.e. each time an `account/3` constraint is added or removed.

Issue 1: *Updates.* Recall the following rule from Example 1 (Section 3.1):

```
deposit(A,X), account(A,C,B) <=> account(A,C,B+X).
```

The above rule is an instance of a common CHR programming pattern, called an *update*: a constraint is removed and immediately replaced with a similar, updated version. In the context of aggregates however, the removal of the former may cause aggregates to be recomputed prior to the insertion of the updated version[2]. This behavior is not always desired. For instance, in the intermediate state right after the above rule removes '`account(A,C,B)`', the accumulated balance in Example 4 would clearly be incorrect.

As a solution, we introduce pragma `passive_removal`. If a constraint annotated with `passive_removal` is removed, no `on_removal` heads are activated:

```
deposit(A,X), account(A,C,B) # passive_removal <=> account(A,C,B+X).
```

Consequently, the aggregate is only recomputed when the new, updated account is added. This allows the CHR programmer to easily specify the desired behavior.

Lines 6–9. The rules performing the actual aggregate computation are added to the object program by lines 6 to 9. Line 6 implements the $\texttt{agg}_i/2$ operation. First, the intermediate aggregate result is initialized using the aggregate's `Start` operation. This intermediate result is stored as a $\texttt{result}_i/1$ constraint. Then the $\texttt{match}_i/1$ constraint is called, causing the intermediate result to be incremented for each matching aggregate goal `G` (lines 7–8). To perform the matching with the aggregate goal `G` (line 7), the variables `V` it shares with the remaining head of the original object rule are needed (line 3).

Example 5. For the `sum/3` aggregate in Example 2 the following code is generated (recall from Example 3 that `sum(T,G,A)` ≡ `aggregate(=(0),plus,minus,=,T,G,A)`):

```
agg0(C,Sum) <=> 0=I, result0(I), match0(C), get0(R), R=A.
match0(C), account(_,C,B) ==> incr0(B).
incr0(B), result0(R1) <=> plus(R1, B, R2), result0(R2).
result0(R), match0(_), get0(Q) <=> Q = R.
```

If the sum aggregate (the accumulated balance) has to be computed, the result is initialized to zero, stored as a constraint, and then incremented with the balance `B` of each matching `account/3` constraint. To perform this match, the variable `C` (the client's identifier) is indeed required.

The intermediate result $\texttt{result}_i/1$ is incremented through the auxiliary constraint $\texttt{incr}_i/1$ (line 8). This way, the propagation history of the rule on line 7 ensures that each matching goal `G` contributes only once. The argument passed to $\texttt{incr}_i/1$ (line 7), and subsequently to `Inc` (line 8), is the aggregate's template `T`. The refined operational semantics (cf. Section 2.2) ensures that the call to $\texttt{match}_i/1$ only returns to the rule body on line 6 after all matchings and increments are performed. A last auxiliary constraint, $\texttt{get}_i/1$, is then used to retrieve and remove the computed result (line 8). Finally, this result is finalized using `Final` to obtain the aggregate result `A` (line 6).

[2] Similar issues were outlined in [11] in the context of negation as absence.

4.3 Incremental Aggregate Computation

The performance of on-demand aggregate computation, described in the previous section, is not always adequate. Aggregates ranging over large portions of the constraint store may be recomputed from scratch many times. In such cases, it is obviously more efficient to maintain the aggregate value incrementally.

```
1  aggregate(Start,Inc,Dec,Final,T,G,A) <=>
2   new_unique_identifier(i),
3   guard(Guard), remaining_head(Head), shared_vars(Head, (T-G), V),
4   +match_i(V,I), +result_i(I,R), ?Final(R,A),
5   ( Head ==> Guard | init_i(V) ),
6   ( match_i(V,_) \ init_i(V) <=> true ),
7   ( init_i(V) <=> Start(R), match_i(V,I), result_i(I,R) ),
8   ( match_i(V,I), G ==> incr_i(I,T) ),
9   ( incr_i(I,T), result_i(I,R_1) <=> Inc(R_1, T, R_2), result_i(I,R_2) ),
10  ( match_i(V,I)#passive, G#on_removal ==> decr_i(I,T) pragma no_history),
11  ( decr_i(I,T), result_i(I,R_1) <=> Dec(R_1, T, R_2), result_i(I,R_2) ).
```

Fig. 2. Transformation scheme for maintained aggregates (a basic, first attempt).

The meta rule in Figure 2 illustrates a basic transformation scheme for incrementally maintained aggregates. The scheme is not yet fully correct with respect to the ω_a operational semantics (cf. Section 3.3) though. Subsequent subsections will refine it to deal with certain semantical issues, and more complex aggregates such as nested and non-ground aggregates.

Basic scheme. Similar to the transformation scheme of Section 4.2, aggregate results are stored in `result`$_i$`/2` constraints, and `match`$_i$`/2` constraints are used to find matches with the aggregate's goal `G` (lines 8 and 10). The need for the extra argument, an aggregate identifier, is explained below.

Line 4 The aggregate is no longer replaced by a guard that computes the aggregate result, but instead with a `match`$_i$`/2` and a `result`$_i$`/2` occurrence in the object rule's head. Both new occurrences are kept because the computed aggregate result may be needed more than once. Line 4 also adds a guard to finalizes the aggregate result.

Incremental maintenance (lines 8–11) The `result`$_i$`/2` and `match`$_i$`/2` constraints remain in the store, and the rules added by lines 8–11 ensure these results remain consistent. Maintained results are incremented each time a new matching is found for `G` (lines 8–9), and decremented each time such a matching is removed (lines 10–11). For the latter, the `Dec` argument of `aggregate/7` is used. This argument indicates the inverse operation of `Inc`.

Line 10 The different pragmas and annotations in the rule on line 10 warrant extra clarification. The rule must not fire when a `match`$_i$`/2` is active, only when constraints matching `G` are removed. Therefore, the `match`$_i$`/2` occurrence is made

passive (`pass` is short for `passive`, a common CHR pragma). Reacting to constraint removals is done, as in Section 4.2, using an `on_removal` head. Finally, pragma `no_history` is added, indicating no propagation history has to be kept for this rule. Otherwise, the rule would only fire once per $\texttt{match}_i\texttt{/2}$ constraint, as the `on_removal` head is not included in propagation history tuples.

Aggregate identifiers More than one result may have to be maintained at the same time. To ensure the right result is updated after a match is found (lines 8 and 10), we let corresponding $\texttt{match}_i\texttt{/2}$ and $\texttt{result}_i\texttt{/2}$ constraints share a unique identifier, and pass this to the $\texttt{incr}_i\texttt{/2}$ or $\texttt{decr}_i\texttt{/2}$ constraint. Other than that, the pattern used to increment and decrement the maintained results is the same as the pattern used in Section 4.2.

Initialization (lines 5–7) Eagerly maintaining all possible aggregate results would be overly expensive. Aggregate maintenance is instead only started once a matching is found for the remainder of the head, as realized by the rule added on line 5. The head and guard of this rule are copied from the original object rule (without copying the aggregate head itself), and its body calls an $\texttt{init}_i\texttt{/2}$ auxiliary constraint. This constraint is removed by the rule on line 6 if the same aggregate result is already being maintained; in the other case, the rule on line 7 initializes a new result, stores it as a $\texttt{result}_i\texttt{/2}$, and adds a $\texttt{matching}_i\texttt{/2}$ constraint.

Issue 1: *Multiple Removals.* The basic scheme does not fully implement the ω_a semantics defined in Section 3.3. This subsection addresses a first issue:

Example 6. Consider the following artificial example:

```
a, count(c, Cs) <=> Cs \== 2 | writeln(Cs).
b \ c, c <=> true.
```

where the "`count(c, Cs)`" aggregate counts the number of `c/0` constraints. Now consider the query "c, c, a, b". First two `c` constraints are added, then `a` is called. The latter causes the `count/2` aggregate to be computed. As the result is equal to two, the first rule does not fire. After adding `b`, the second rule fires and removes both `c` constraints. Suppose the count is maintained incrementally. The removal of the first `c` constraint causes the maintained result to be decremented. The count becomes equal to one, causing the first rule to fire with `Cs` equal to *one*, even though there are *no* `c` *constraints left.* This is clearly not correct. The reason is that, whilst both `c` constraints are removed simultaneously, the updates to the maintained aggregate are performed, and visible, one by one.

Our solution is based on splitting the activation of `on_removal` heads into two phases: `on_removal1` and `on_removal2`. When a rule fires, the removed constraints are first matched against `on_removal1` heads. Only after this is done for *all* removed constraints, the same is repeated for the `on_removal2` heads.

Lines 10–11 of Figure 2 are replaced with those in Figure 3. The rules added on lines $10^\star$–$11^\star$ ensure that first, in the `on_removal1` phase, all affected aggregates are made consistent. The updated results are not yet used immediately as in Example 6. Instead, the $\texttt{result}_i\texttt{/2}$ constraint is added *passively* to the

```
...
10⋆ ( match_i(V,I)#passive, G#on_removal1 ==> decr_i(I,T) pragma no_history),
11⋆ ( decr_i(I,T), result_i(I,R_1) <=> Dec(R_1,T,R_2), result_i(I,R_2)#passive)
12⋆ ( match_i(V,I)#passive, G#on_removal2,
13⋆             result_i(I,_)#Id ==> chr_reactivate_i(Id) pragma no_history).
```

Fig. 3. Code to replace lines 10–11 of the transformation scheme of Figure 2 to correctly deal with multiple constraint removals. Several new lower-level CHR constructs are used. Their semantics is explained in the accompanying text.

constraint store, that is, without searching for matching occurrences. Hence the '`#passive`' annotation in the body of the rule on line 11⋆. The results only become active once *all* results are guaranteed consistent again, that is, in the `on_removal2` phase (line 12⋆). Activating a constraint is done using the low-level `chr_reactivate/1` primitive (line 13⋆).

Issue 2: ***Updates.*** The update pattern causes similar issues in the context of incrementally computed aggregates as described before in Section 4.2, Issue 1. The solution is analogous as well, only with a slightly refined semantics of pragma `passive_removal`: if a constraint annotated with `passive_removal` is removed, no `on_removal2` heads are activated, only `on_removal1` heads (cf. previous issue). As such, the maintained result is passively decremented, but the aggregate only becomes active when the new, updated account is added.

Issue 3: ***Nested Aggregates.*** A second semantical problem with the basic transformation scheme occurs when applying it to nested aggregates. The maintained value of a nested aggregate is incremented and decremented using the update pattern. The outer aggregate consequently observes the intermediate state in which the `result/2` constraint holding the old maintained value of the nested aggregate is removed and the new, updated version is not yet added. The solution consists of slightly adjusting lines 9 and 11 in Figure 2, and line 11⋆ in Figure 3, to use pragma `passive_removal` for updates to `result/2` constraints.

Issue 4: ***Propagation Histories.*** The transformation scheme adds two extra heads per aggregate. However, according to the ω_a semantics these are not allowed to be part of the propagation history. Pragma `history/2`, introduced in [9], can be used to explicitly specify which occurrence identifiers have to be included in the history tuples. Thus the issue is solved by adding the following code after line 3 of Figure 2 (`hist`$_i$ is a unique identifier):

```
..., identifiers(Head, Ids), pragma( history(hist_i, Ids) ), ...
```

Issue 5: ***Non-ground Aggregates.*** Two final issues occur when aggregating over goals containing non-ground variables:

– A single built-in constraint (e.g. unification) may cause *multiple goals* to match. The problem is analogous to Issue 1 on *multiple removals*. It has to be ensured that *first* all aggregates are updated, i.e. incremented in this case, *before* the aggregate results are activated.

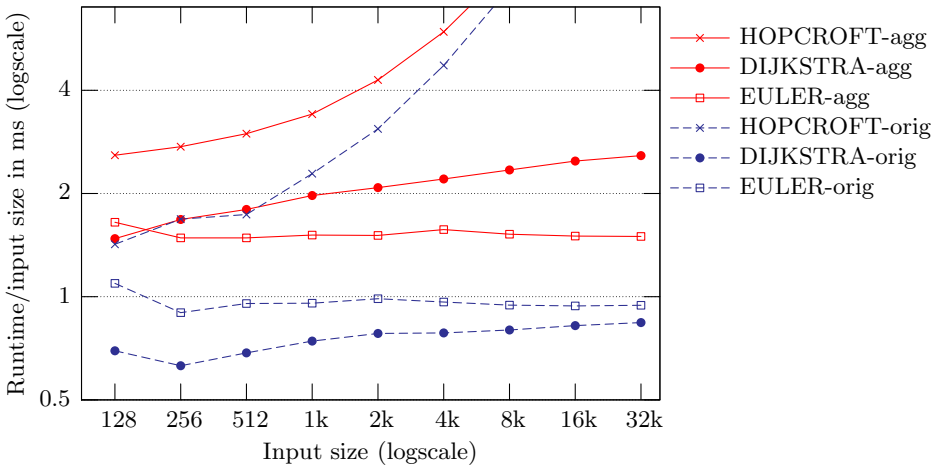


Fig. 4. Runtimes for three programs, with and without aggregates.

– A unification can cause two or more `match/2` constraints to coincide. To preserve correctness, we would have to add the following rule to Figure 2:

```
..., ( match_i(V,_) \ match_i(V,I), result(I,_) <=> true ), ...
```

Unfortunately, the refined operational semantics (which is used to execute the result of the transformation), does not determine the order in which constraints are reactivated (cf. Section 2.2). This implies their is no clear-cut way to ensure all aggregates are made consistent, or duplicate maintained results are removed, *before* other CHR constraints are reactivated and use the incorrect aggregate values. This lack of control is a general problem of current CHR systems, that warrants further research outside the scope of this paper (see also [2] and Section 5). Fortunately, most aggregates range over ground data. For aggregates ranging over non-ground data, only the on-demand transformation is correct.

5 Discussion and Evaluation

Performance Evaluation. In [8, 9] we revised a number of existing CHR programs to use aggregates. Because our transformation schemes have to deal with all possible use patterns of aggregates, and the original programs are manually specialized, we expect the programs using aggregates to be slower than the original programs. Our prototype implementation however shows the runtime complexity can be maintained, with an acceptable constant overhead. Figure 4 plots benchmark results for the different versions of the DIJKSTRA, EULER, and HOPCROFT programs (cf. [8, 9]). The DIJKSTRA-agg program is about three times slower than the manually specialized DIJKSTRA-orig. For EULER and HOPCROFT, the version with aggregates is only about 1.5 times slower.

The DIJKSTRA-agg program uses an incrementally maintained `min` aggregate. The implementation of this aggregate relies on an efficient priority queue implementation. This illustrates another advantage of language support for aggregates: the data structures required for efficient aggregate computation only have to be implemented once; end users no longer have to worry about this.

For the above figures, a transformation scheme presented in [9] is used for the incrementally maintained aggregates. This scheme is an extended version of the scheme of Section 4.3, in which aggregates are still replaced by guards. The incremental scheme of Section 4.3 considerably improves the latter scheme: it permits efficient indexing on aggregate results, and failing guards no longer backtrack over result maintenance. For the above benchmarks though, we expect no significant difference in performance.

Discussion. Section 4 indicated several issues that occur when transforming to CHR code. A common thread is the lack of control offered by the refined operational semantics, a problem also perceived outside the context of aggregates (cf. [2]). Whilst the low-level constructs we introduced in this paper are acceptable for generated code or expert use, more high-level, declarative control structures are required for the CHR programmer. A first step are the *user-definable rule priorities* introduced by [2].

Related Work. Constructs related to aggregates are found in many languages. For SQL [10], which unlike CHR [7] is not Turing-complete, aggregates do add computational power. The original SQL standard only supports five aggregates: `min`, `max`, `count`, `sum`, and `avg`. Many recent database systems also include the possibility to extend the database query language with user-defined aggregates.

Recently, several production rule systems introduced a general `accumulate` construct, similar to our `aggregate`/7. As far as we know, current versions lack support for nested aggregates, complex goals, and incremental maintenance.

In logic programming, the best-known practical implementation of aggregates are the *all solutions* predicates `findall/3`, `bagof/3` and `setof/3`. Other aggregates can be implemented in terms of these all solutions predicates.

In [11] we introduced CHR$^{\neg}$, an extension of CHR with negation as absence. CHR with aggregates is a far more expressive generalization of CHR$^{\neg}$ as negation as absence can easily be expressed using the `count/2` aggregate.

6 Conclusion and Future Work

In this paper we presented an implementation approach for aggregates, a new declarative language feature for CHR that considerably increases its expressiveness. The approach is based on source-to-source transformation to regular CHR (extended with some low-level constructs). As a side-effect of our work, we created a practical, high-level source-to-source framework based on meta CHR rules. We outlined the design of non-trivial transformation schemes for on-demand and

incremental aggregate computation, and clearly showed the effectiveness of CHR-to-CHR transformations. The source-to-source implementation approach allows for flexible and rapid implementations, easily portable to existing CHR systems. The current generation of optimizing CHR compilers ensure the desired runtime complexity is achieved, with an acceptable constant overhead. We clearly identified the issues that occur when transforming to CHR, and showed how they can be addressed using newly introduced low-level constructs. Several of these constructs have already proven useful outside the context of aggregates (e.g. [2]).

In future work, various ways can be investigated to improve the efficiency of our aggregates implementation. In particular, both specializations on the source level and dedicated support in the CHR compiler can be considered. Even though source-to-source transformation remains effective for aggregates in their full generality, specific cases can e.g. be distinguised where incremental maintenance of aggregates can be embedded directly in the constraint store insertion and removal operations. Also, static and dynamic analyses can be developed to automatically select the aggregate computation strategy (on-demand, incremental, or maybe hybrid strategies).

References

1. The CHR Home Page. *http://www.cs.kuleuven.be/∼dtai/projects/CHR/*.
2. Leslie De Koninck, Tom Schrijvers, and Bart Demoen. User-definable rule priorities for CHR. In *9th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, pages 25–36, Wrocław, Poland, July 2007.
3. Gregory J. Duck, Peter J. Stuckey, María García de la Banda, and Christian Holzbaur. The refined operational semantics of Constraint Handling Rules. In *20th Intl. Conf. Logic Programming*, LNCS 3132, Saint-Malo, France, 2004.
4. Thom Frühwirth. Theory and practice of Constraint Handling Rules. *Journal of Logic Programming*, 37(1–3):95–138, October 1998.
5. Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Meda, Christina Lopes, Jean-Marc Loingtier, and John Irwing. Aspect oriented programming. In *European Conference on Object-Oriented Programming (ECOOP)*. LNCS 1241, 1997.
6. Tom Schrijvers and Bart Demoen. The K.U.Leuven CHR system: Implementation and application. In *Selected Contributions, First Workshop on Constraint Handling Rules*, May 2004. Home page at *http://www.cs.kuleuven.be/∼toms/CHR/*.
7. Jon Sneyers, Tom Schrijvers, and Bart Demoen. The computational power and complexity of Constraint Handling Rules. In *Second Workshop on Constraint Handling Rules*, pages 3–17, Sitges, Spain, October 2005.
8. Jon Sneyers, Peter Van Weert, Tom Schrijvers, and Bart Demoen. Aggregates in CHR. In *Fourth Workshop on Constraint Handling Rules*, 2007. To appear.
9. Jon Sneyers, Peter Van Weert, Tom Schrijvers, and Bart Demoen. Aggregates in CHR. Technical Report CW481, Dept. Computer Science, K.U.Leuven, 2007.
10. ISO/IEC 9075:2003: Information technology – Database languages – SQL.
11. Peter Van Weert, Jon Sneyers, Tom Schrijvers, and Bart Demoen. Extending CHR with negation as absence. In *Third Workshop on Constraint Handling Rules*, pages 125–139, Venice, Italy, 2006.
12. Jan Wielemaker. An overview of the SWI-Prolog programming environment. In *13th Intl. Workshop on Logic Programming Environments*, pages 1–16, Heverlee, Belgium, 2003. Home page at *http://www.swi-prolog.org*.

Generation of Rule-based Constraint Solvers: Combined Approach

Slim Abdennadher and Ingi Sobhi

Computer Science Department, German University in Cairo
[slim.abdennadher, ingi.sobhi]@guc.edu.eg
http://www.cs.guc.edu.eg

Abstract. *Inductive Constraint Solving* is a subfield of inductive machine learning concerned with the automatic generation of rule-based constraint solvers. In this paper, we propose an approach to generate constraint solvers given the definition of the constraints that combines the advantages of generation by construction with generation by testing. In our proposed approach, semantically valid rules are constructed symbolically, then the constructed rules are used to prune the search tree of a generate and test method. The combined approach leads in general to more expressive and efficient constraint solvers. The generated rules are implemented in the language Constraint Handling Rules.

1 Introduction

In rule-based constraint solving, the execution of constraints consists of a repeated application of rules. In general, we distinguish between two types of rules:

- Simplification rules that rewrite constraints to simpler constraints while preserving logical equivalence (e.g. $min(A, A, C) \Leftrightarrow C{=}A$).
- Propagation rules that add new constraints, which are logically redundant but may cause further simplification (e.g. $min(A, B, C) \Rightarrow C{\leq}A \wedge C{\leq}B$).

Writing rule-based constraint solvers is a hard task as the programmer has to determine the propagation algorithms. Several methods have been proposed in the field of inductive constraint solving to automate the generation of constraint solvers for constraints defined extensionally over finite domains by means of a truth table [5, 9, 2] or intentionally over infinite domains by means of a constraint logic program (CLP) [3, 4]. In general, the algorithms follow a generate and test approach. Rule candidates are enumerated and subjected to a validity test against the definition of the constraint.
In this paper, we present a joined approach that combines the generate and test method with a symbolic construction method. Each method has its advantages and drawbacks. The construction method is an orthogonal approach to the general direction of the work done in the field. While it is able to generate recursive rules that cannot be generated by the generate and test method, it is likely to

cover a narrower spectrum of rules. The generate and test method on the other hand generates a more exhaustive set of rules, however this does come at a cost. Our aim is to combine the advantages of the two approaches, while minimizing the drawbacks.
In our combined approach, we first construct semantically valid rules symbolically. Then, we use the constructed rules to prune the search tree of the generate and test method. This will generally lead to more powerful and expressive constraint solvers at a reduced cost.
In the following, we will illustrate the combined approach by an example.

Example 1. Given the following CLP program defining $min(A, B, C)$ that holds if C is the minimum of A and B.

$$min(A, B, C) \leftarrow A \leq B \ \wedge \ C = A.$$
$$min(A, B, C) \leftarrow A > B \ \wedge \ C = B.$$

The combined approach will construct rules symbolically.

Symbolic Construction The basic idea of the symbolic construction method stems from the observation that in general, the execution of one clause in a CLP program excludes the execution of all other clauses. Thus, to construct a simplification rule that replaces the head of a clause by the body of the clause while preserving the semantics of the CLP program, the construction algorithm adds to the head of the rule the negation of the bodies of all the other clauses. The negation of the bodies of clauses may result in a disjunction of constraints, thus for each clause a set of rules might be generated. Note that constraints that are added to the head of the rule are also added to its body to ensure that constraints removed unnecessarily are added again.
The construction algorithm generates the following simplification rules:

$$min(A, B, C) \ \wedge \ A \leq B \ \Leftrightarrow \ A \leq B \ \wedge \ C = A.$$
$$min(A, B, C) \ \wedge \ C \neq B \ \Leftrightarrow \ A \leq B \ \wedge \ C = A \ \wedge \ C \neq B.$$
$$min(A, B, C) \ \wedge \ A > B \ \Leftrightarrow \ A > B \ \wedge \ C = B.$$
$$min(A, B, C) \ \wedge \ C \neq A \ \Leftrightarrow \ A > B \ \wedge \ C = B \ \wedge \ C \neq A.$$

To generate a simplification rule that replaces the head of the first clause with the body of the clause, the construction algorithm negates the bodies of all other clauses (i.e. second clause) to add to the head of the rule. The negation of the body of the second clause gives $A \leq B \ \vee \ C \neq B$, a disjunction of constraints. This results in two separate rules (first and second rule), one for each disjunct. Similarly, the last two rules are generated from the second clause.
Then the combined approach will eliminate the above constructed rules from the search tree of the generate and test algorithm. The generate and test algorithm used is the one proposed in [4].

Generate and Test All possible candidate constraints for the left hand side C of the rule and right hand side D of the rule are generated and tested based on the observation that a rule of the form $C \Rightarrow D$ is valid if the execution of the goal $C \wedge \neg(D)$ finitely fails with respect to the definition.
The generate and test algorithm will add the following rules to the constraint solver:

$$min(A,B,C) \;\Rightarrow\; C{\leq}A \;\wedge\; C{\leq}B.$$
$$min(A,B,C) \;\wedge\; B{\leq}A \;\Leftrightarrow\; B{\leq}A \;\wedge\; C{=}B.$$

The propagation rule (first rule) is generated by calling the CLP system to execute the goal $min(A,B,C) \;\wedge C{>}A \;\wedge C{>}B$ that fails.

The generated solvers are implemented in the language Constraint Handling Rules (CHR) [8].
The paper is organized as follows. The generate and test algorithm of [4] is summarized in Section 2. In Section 3, we present the construction algorithm. Then in Section 4, we present the combined approach. Finally, we conclude in Section 5 with a summary and future work.

2 Generate and Test Method

In this section, we summarize the generate and test algorithm that we use for the combined approach and which is given in [4]. The algorithm requires as input a CLP program defining the user-defined constraint for which the solver is needed.

Definition 1. *A* CLP program *is a finite set of CLP clauses. A* CLP clause *is a rule of the form* $H \leftarrow B_1 \wedge \ldots \wedge B_n \wedge C_1 \wedge \ldots \wedge C_m$ *where* $H, B_1, \ldots, B_n$ *are atoms and* $C_1, \ldots, C_m$ *are built-in constraints. H is called the* head *of the clause and* $B_1 \wedge \ldots \wedge B_n \wedge C_1 \wedge \ldots \wedge C_m$ *is called the* body *of the clause. A user-defined constraint is* defined *in a CLP program if it occurs in the head of the clause.*

The algorithm also requires the following sets which specify the syntactic form of the generated rules of the solver:

- A set of built-in and user-defined constraints denoted by $Base_{lhs}$. These constraints are the common part that must appear in the left hand side (lhs) of all rules.
- A set of built-in and user-defined constraints denoted by $Cand_{lhs}$. These are the candidate constraints to be used in conjunction with the $Base_{lhs}$ to form the lhs of a rule.
- A set containing built-in constraints denoted by $Cand_{rhs}$. These are the candidate constraints that may appear in the right hand side (rhs) of a rule. This set can be expanded to contain user-defined constraints.

Example 2. To generate a constraint solver for *min* constraint of Example 1, the algorithm takes as input the CLP program defining the constraint, as well as, the following sets:

$$\begin{aligned} Base_{lhs} &= \{min(A,B,C)\} \\ Cand_{lhs} &= \{A{=}B, A{=}C, B{=}C, A{\neq}B, A{\neq}C, B{\neq}C, \\ &\quad\ A{\leq}B, A{\leq}C, B{\leq}A, B{\leq}C, C{\leq}A, C{\leq}B\} \\ Cand_{rhs} &= Cand_{lhs} \end{aligned}$$

Example 3. To generate a constraint solver for $append(A,B,C)$ that holds if list C is the concatenation of lists A and B, the algorithm takes as input the following CLP program:

$$append(A,B,C) \leftarrow A{=}[] \ \wedge\ C{=}B.$$
$$append(A,B,C) \leftarrow A{=}[D|E] \ \wedge\ C{=}[F|G] \ \wedge\ D{=}F \ \wedge\ append(E,B,G).$$

As well as, the following sets:

$$\begin{aligned} Base_{lhs} &= \{append(A,B,C)\} \\ Cand_{lhs} &= \{A{=}[], B{=}[], C{=}[], A{=}B, A{=}C, B{=}C, A{\neq}B, A{\neq}C, B{\neq}C, \\ &\quad\ A{\neq}[], B{\neq}[], C{\neq}[]\} \\ Cand_{rhs} &= Cand_{lhs} \end{aligned}$$

Given the specified input parameters, candidate propagation rules are generated of the form $C \Rightarrow D$, where C the lhs of the rule is a subset of $Base_{lhs} \cup Cand_{lhs}$, and D the rhs of the rule is a subset of $Cand_{rhs}$. The candidate rules are then subjected to a validity test as follows:

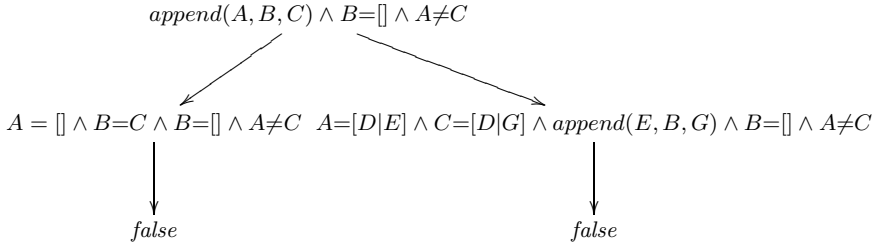
- For *primitive* propagation rules (i.e. rules with rhs consisting of only built-in constraints), the validity test is based on the observation that a rule of the form $C \Rightarrow D$ is valid if the execution of the goal $C \wedge \neg(D)$ finitely fails with respect to the given CLP program and the predefined solver for the built-in constraints.
- For *general* propagation rules (i.e. rules with rhs consisting of both built-in and user-defined constraints) to avoid the problems relating to the negation of user-defined constraints, a different validity test is proposed where the negation is performed on the set of answers to a goal (set of constraints) rather than on the constraints themselves.

For the execution of the goals, a bounded depth tabled resolution [6, 7] for CLP is used to avoid non-termination. The intuitive basic principle of tabled resolution is the following: each new subgoal S is compared to the previous intermediate subgoals (not necessarily in the same branch of the resolution tree). If there is a previous subgoal I which is equivalent to S or more general than S, then no more unfolding is performed on S and answers for S are selected among the answers of I. This process is repeated for all subsequent computed answers that correspond to the subgoal I.

Example 4. Consider the following primitive propagation rule which is generated by the algorithm for the *append* constraint:

$$append(A, B, C) \ \wedge \ B{=}[] \ \Rightarrow \ A{=}C.$$

The validity test for the rule is determined from the execution of the goal $append(A, B, C) \ \wedge \ B{=}[] \ \wedge \ A{\neq}C$. Using a classical CLP resolution scheme, the goal will lead to an infinite derivation tree, whereas in the case of a tabled resolution, the execution of the goal will fail as shown by the derivation tree below:



The initial goal $G_1 = (append(A, B, C) \ \wedge \ B{=}[] \ \wedge \ A{\neq}C)$ is more general than the subgoal $G_2 = (A{=}[D|E] \ \wedge \ C{=}[D|G] \ \wedge \ append(E, B, G) \ \wedge \ B{=}[] \ \wedge \ A{\neq}C)$, in the sense that $(append(X, Y, Z) \ \wedge \ U{=}[W|X] \ \wedge \ V{=}[W|Z] \ \wedge \ Y{=}[] \ \wedge \ U{\neq}V)$ entails $(append(X, Y, Z) \ \wedge \ Y{=}[] \ \wedge \ X{\neq}Z)$. So no unfolding is made on G_2, and the process waits for answers of G_1 to compute answers of G_2. Since G_1 has no further possibility of having answers, then G_2 fails and thus G_1 also fails.

Since a propagation rule does not remove constraints but adds new ones, the constraint store may contain superfluous information. To improve the time and space behavior of constraint solving, propagation rules should be transformed into equivalent simplification rules. For some of the automatically generated propagation rules a transformation to simplification rules is possible. For a valid propagation rule of the form $C \Rightarrow D$, if a proper subset E of C can be found such that $D \cup E \Rightarrow C$ is valid too then the propagation rule can be transformed to a simplification rule of the form $C \Leftrightarrow D \cup E$.

Example 5. For the *min* constraint of Example 1, the generate and test algorithm generates the following valid rules:

$$min(A, B, C) \ \Rightarrow \ C{\leq}A \ \wedge \ C{\leq}B. \tag{1}$$
$$min(A, B, C) \ \wedge \ C{\neq}A \ \Leftrightarrow \ C{=}B \ \wedge \ C{\neq}A. \tag{2}$$
$$min(A, B, C) \ \wedge \ C{\neq}B \ \Leftrightarrow \ C{=}A \ \wedge \ C{\neq}B. \tag{3}$$
$$min(A, B, C) \ \wedge \ A{\leq}B \ \Leftrightarrow \ C{=}A \ \wedge \ A{\leq}B. \tag{4}$$
$$min(A, B, C) \ \wedge \ B{\leq}A \ \Leftrightarrow \ C{=}B \ \wedge \ B{\leq}A. \tag{5}$$

The set of generated rules is complete, i.e. it propagates all built-in constraints (equalities and inequalities) that logically follow from the *min* constraint definition and some given equalities or inequalities.

Example 6. For the *append* constraint of Example 3, the generate and test algorithm generates among others the following valid rules:

$$append(A,B,C) \ \wedge \ A{=}[] \ \Leftrightarrow \ A{=}[] \ \wedge \ B{=}C.$$
$$append(A,B,C) \ \wedge \ B{=}[] \ \Leftrightarrow \ A{=}C \ \wedge \ B{=}[].$$
$$append(A,B,C) \ \wedge \ C{=}[] \ \Leftrightarrow \ A{=}[] \ \wedge \ B{=}[] \ \wedge \ C{=}[].$$
$$append(A,B,C) \ \wedge \ A{=}C \ \Leftrightarrow \ B{=}[] \ \wedge \ A{=}C.$$
$$append(A,B,C) \ \wedge \ A{\neq}[] \ \Rightarrow \ C{\neq}[].$$
$$append(A,B,C) \ \wedge \ B{\neq}[] \ \Rightarrow \ A{\neq}C \ \wedge \ C{\neq}[].$$

The rules handle only special cases, where equality or inequality constraints are checked between the arguments of the constraint and the empty list. The solver is incomplete due to the absence of recursive rules that are able to handle more general cases.

3 Symbolic Construction Method

In this section, we present an algorithm that constructs simplification rules symbolically for a constraint H defined by a CLP program, as follows:

$$H \leftarrow C_1, H \leftarrow C_2, \ldots, H \leftarrow C_n.$$

where C_i is a conjunction of constraints, n is the total number of clauses and the clauses are non-overlapping (i.e. in a computation at most one clause can be chosen for a goal). Note that any overlapping CLP program can be transformed into an equivalent non-overlapping one.
The algorithm is presented in Figure 1. The basic idea of the algorithm stems from the observation that in general, the execution of one clause in a CLP program excludes the execution of all other clauses. Thus, to construct a valid simplification rule that simplifies the constraint H to C_i (the body of the ith clause), the negation of the bodies of all other clauses is added to the head of the rule to ensure that the rule will only be *applicable* if the bodies of all the other clauses are not. This is needed to preserve the semantics of the CLP program defining the constraint.
The algorithm works as follows. For each clause $H \leftarrow C_i$ in the CLP program, it constructs the simplification rule(s) by:

- Setting the head of the rule to H.
- Setting the body of the rule to C_i.
- Adding to the head of the rule G_i^j; a disjunct from G_i, the expression resulting from negating the bodies of all the CLP clauses excluding C_i.
- Adding to the body of the rule G_i^j. This is done to ensure that constraints removed unnecessarily from the constraint store are added again.

The constructed simplification rules are of the form:

$$H \wedge G_i^j \Leftrightarrow C_i \wedge G_i^j \qquad 1{\leq}j{\leq}m_i, \ 1{\leq}i{\leq}n$$

where G_i^j is a conjunction of built-in constraints from G_i and m_i is the number of disjuncts G_i^j in G_i.

begin
H: the head of the clauses.
B: the set of clause bodies.
R: the set of resultant simplification rules initialized to [].

while B is not empty **do**
Remove from B its first element denoted C_i.
$Other_B$: the set of all clause bodies except C_i.
G_i: the set resulting from negating $Other_B$.
while G_i is not empty **do**
Remove from G_i its first element denoted G_i^j.
Add rule $(H \wedge G_i^j \Leftrightarrow C_i \wedge G_i^j)$ to R.
end while
end while
end

Fig. 1. The Symbolic Construction Algorithm

Determination of G_i Given a clause $H \leftarrow C_i$, the expression G_i is formally determined as follows:

- Negate the bodies of all clauses of the CLP program except the body C_i:

$$\neg (C_1 \vee \ldots \vee C_{i-1} \vee C_{i+1} \vee \ldots \vee C_n)$$

- Distribute the negation:

$$\neg C_1 \wedge \ldots \wedge \neg C_{i-1} \wedge \neg C_{i+1} \wedge \ldots \wedge \neg C_n$$

  Since C_i is a conjunction of constraints, this expands to:

$$\neg \left(c_1^1 \wedge \ldots \wedge c_1^{k_1} \right) \ \wedge \ \ldots \ \wedge \ \neg \left(c_{(i-1)}^1 \wedge \ldots \wedge c_{(i-1)}^{k_{(i-1)}} \right) \ \wedge$$

$$\neg \left(c_{(i+1)}^1 \wedge \ldots \wedge c_{(i+1)}^{k_{(i+1)}} \right) \ \wedge \ \ldots \ \wedge \ \neg \left(c_n^1 \wedge \ldots \wedge c_n^{k_n} \right)$$

  where k_i denotes the number of constraints in a body C_i.
- Push the negation into the conjunctions. This transforms the conjunctions of constraints to disjunctions of negated constraints:

$$\left(\neg c_1^1 \vee \ldots \vee \neg c_1^{k_1} \right) \ \wedge \ \ldots \ \wedge \ \left(\neg c_{(i-1)}^1 \vee \ldots \vee \neg c_{(i-1)}^{k_{(i-1)}} \right) \ \wedge$$

$$\left(\neg c_{(i+1)}^1 \vee \ldots \vee \neg c_{(i+1)}^{k_{(i+1)}} \right) \ \wedge \ \ldots \ \wedge \ \left(\neg c_n^1 \vee \ldots \vee \neg c_n^{k_n} \right)$$

- Replace each negated constraint $\neg c_q^d$ by a corresponding simplified positive constraint. The algorithm distinguishes between two cases:

- If c_q^d is a built-in constraint, the algorithm replaces $\neg c_q^d$ by its corresponding positive constraint after simplification. The set of built-in constraints is assumed to be closed under negation. For obtained constraints that consist of local variables (i.e. variables that do not occur in H), the algorithm adds the built-in constraints (in their positive form) from the body C_q that define the local variables.
- Otherwise, c_q^d is a user-defined constraint and since the negation of user-defined constraints is still not well-defined, the algorithm discards $\neg c_q^d$ (i.e. no rules will be constructed for this case).

This results in a formula of the form:

$$\left(P_1^1 \vee \ldots \vee P_1^{l_1}\right) \wedge \ldots \wedge \left(P_{(i-1)}^1 \vee \ldots \vee P_{(i-1)}^{l_{(i-1)}}\right) \wedge$$

$$\left(P_{(i+1)}^1 \vee \ldots \vee P_{(i+1)}^{l_{(i+1)}}\right) \wedge \ldots \wedge \left(P_n^1 \vee \ldots \vee P_n^{l_n}\right)$$

where P_i^e is a built-in constraint or a conjunction of built-in constraints and l_i denotes the number of built-in constraints in a disjunct C_i.

– Distribute the conjunction over the disjunction:

$$\left(P_1^1 \wedge \ldots \wedge P_{(i-1)}^1 \wedge P_{(i+1)}^1 \wedge \ldots \wedge P_n^1\right) \vee \ldots \vee$$

$$\left(P_1^{l_1} \wedge \ldots \wedge P_{(i-1)}^{l_{(i-1)}} \wedge P_{(i+1)}^{l_{(i+1)}} \wedge \ldots \wedge P_n^{l_n}\right)$$

This results in G_i, which is a formula in disjunctive normal form $G_i^1 \vee \ldots \vee G_i^{m_i}$, where G_i^j is a conjunction of built-in constraints.

Example 7. Given the CLP program for the *append* of Example 3:

$$append(A, B, C) \leftarrow A{=}[] \wedge C{=}B.$$
$$append(A, B, C) \leftarrow A{=}[D|E] \wedge C{=}[F|G] \wedge D{=}F \wedge append(E, B, G).$$

The symbolic construction algorithm will construct rules for the first clause by setting the head of the rules to $append(A, B, C)$ and the body of the rules to the body of the clause, $A{=}[] \wedge C{=}B$. It then determines G_1, the expression resulting from negating the bodies of all other clauses as follows:

– Negate the body of the second clause:

$$\neg(A{=}[D|E] \wedge C{=}[F|G] \wedge D{=}F \wedge append(E, B, G))$$

– Distribute the negation:

$$\neg(A{=}[D|E]) \vee \neg(C{=}[F|G]) \vee \neg(D{=}F) \vee \neg(append(E, B, G))$$

– Given that the equality constraint is a built-in constraint defined by a constraint theory and for which a solver is available, the algorithm performs the following operations:

- It replaces $\neg(A{=}[D|E])$ and $\neg(C{=}[F|G])$ by $A{\neq}[D|E]$ and $C{\neq}[F|G]$ which will be simplified by the built-in solver to $A{=}[]$ and $C{=}[]$, respectively.
- It replaces $\neg(D{=}F)$ by $D{\neq}F$. Since D and F are local variables, the built-in constraints $A{=}[D|E]$ and $C{=}[F|G]$ that define the local variables to be the first elements of the lists A and C are added.

– Negated user-defined constraint $\neg(append(E, B, G))$ is discarded.

This results in

$$A{=}[] \ \vee \ C{=}[] \ \vee \ (D{\neq}F \ \wedge \ A{=}[D|E] \ \wedge \ C{=}[F|G])$$

and the following three simplification rules are constructed:

$$append(A, B, C) \ \wedge \ A{=}[] \Leftrightarrow A{=}[] \ \wedge \ C{=}B.$$

$$append(A, B, C) \ \wedge \ C{=}[] \Leftrightarrow A{=}[] \ \wedge \ C{=}B \ \wedge \ C{=}[].$$

$$append(A, B, C) \ \wedge \ D{\neq}F \ \wedge \ A{=}[D|E] \ \wedge \ C{=}[F|G] \Leftrightarrow A{=}[] \ \wedge \ C{=}B \ \wedge$$
$$D{\neq}F \ \wedge \ A{=}[D|E] \ \wedge \ C{=}[F|G].$$

Similarly, the following simplification rules are constructed for the second clause:

$$append(A, B, C) \ \wedge \ A{\neq}[] \Leftrightarrow A{=}[D|E] \ \wedge \ C{=}[F|G] \ \wedge \ D{=}F \ \wedge$$
$$append(E, B, G) \ \wedge \ A{\neq}[].$$

$$append(A, B, C) \ \wedge \ C{\neq}B \Leftrightarrow A{=}[D|E] \ \wedge \ C{=}[F|G] \ \wedge \ D{=}F \ \wedge$$
$$append(E, B, G) \ \wedge \ C{\neq}B.$$

The rules are recursive. The power of the symbolic construction algorithm is in the generation of such recursive rules given a recursive constraint definition.

Simplification In general, the simplification rules constructed are not in the simplest form. To simplify the constructed rules, the head and body of the rules are executed against the solvers for the built-in constraints.

Example 8. Consider the following constructed rule for *append*:

$$append(A, B, C) \ \wedge \ D{\neq}F \ \wedge \ A{=}[D|E] \ \wedge \ C{=}[F|G] \Leftrightarrow \underline{A{=}[]} \ \wedge \ C{=}B \ \wedge$$
$$D{\neq}F \ \wedge \ \underline{A{=}[D|E]} \ \wedge \ C{=}[F|G].$$

Since the existence of the constraints $A{=}[]$ and $A = [D|E]$ leads to a contradiction, the rule will be simplified to :

$$append(A, B, C) \ \wedge \ D{\neq}F \ \wedge \ A{=}[D|E] \ \wedge \ C{=}[F|G] \Leftrightarrow false.$$

Redundancy In general, the generated rules may contain redundant rules. To remove redundant rules, the same algorithm is used as the one summarized in the redundancy pruning in Section 4, which basically states that a rule is redundant and should be removed if its operation is covered by the remaining rules of the solver.

Example 9. Consider the following two rules of the constructed solver for *append*:

$$append(A, B, C) \ \wedge \ D{\neq}F \ \wedge \ A{=}[D|E] \ \wedge \ C{=}[F|G] \Leftrightarrow false.$$

$$append(A, B, C) \ \wedge \ A{\neq}[] \Leftrightarrow A{=}[D|E] \ \wedge \ C{=}[F|G] \ \wedge \ D{=}F \ \wedge$$
$$append(E, B, G).$$

The first rule is redundant and can be removed since removing it and executing the goal $append(A, B, C) \ \wedge \ D{\neq}F \ \wedge \ A{=}[D|E] \ \wedge \ C{=}[F|G]$ on the remaining rules, the second rule will be fired and leads to a contradiction.

Example 10. For the *append* of Example 3, the simplification rules reduce to the following set:

$$append(A, B, C) \ \wedge \ A{=}[] \Leftrightarrow C{=}B \ \wedge \ A{=}[].$$

$$append(A, B, C) \ \wedge \ C{=}[] \Leftrightarrow C{=}B \ \wedge \ A{=}[] \ \wedge \ C{=}[].$$

$$append(A, B, C) \ \wedge \ A{\neq}[] \Leftrightarrow A{=}[D|E] \ \wedge \ C{=}[F|G] \ \wedge \ D{=}F \ \wedge$$
$$append(E, B, G).$$

$$append(A, B, C) \ \wedge \ C{\neq}B \Leftrightarrow A{=}[D|E] \ \wedge \ C{=}[F|G] \ \wedge \ D{=}F \ \wedge$$
$$append(E, B, G) \ \wedge \ C{\neq}B.$$

The rules cover some of the cases, where list A is empty (first and second rules), as well as, when it consists of at least one element (third and fourth rules). In the latter case, the simplification rule is called recursively on each of the elements of list A. However, it should be noted that the solver is not propagation complete, i.e. it does not produce all built-in constraints that logically follows from the constraint definition such as that the list B is empty if it is known that the lists A and C are equal.

Recursive Rules The power of the symbolic construction approach is its ability to generate recursive rules which cannot be generated by other approaches.

Example 11. Consider the following CLP program that defines the constraint $replace(A, B, C, D)$ that holds if list D is the result of replacing all occurrences of A in list C by B.

$$replace(A, B, C, D) \ \leftarrow \ C{=}[] \ \wedge \ D{=}[].$$

$$replace(A,B,C,D) \leftarrow C{=}[E|F] \wedge D{=}[G|H] \wedge E{=}A \wedge G{=}B \wedge replace(A,B,F,H).$$

$$replace(A,B,C,D) \leftarrow C{=}[E|F] \wedge D{=}[G|H] \wedge E{\neq}A \wedge G{=}E \wedge replace(A,B,F,H).$$

The symbolic construction algorithm will generate the following simplification rules:

$$replace(A,B,C,D) \wedge C{=}[] \Leftrightarrow C{=}[] \wedge D{=}[].$$

$$replace(A,B,C,D) \wedge D{=}[] \Leftrightarrow C{=}[] \wedge D{=}[].$$

$$replace(A,B,C,D) \wedge C{=}[E|F] \wedge E{=}A \Leftrightarrow C{=}[E|F] \wedge D{=}[G|H] \wedge E{=}A \wedge G{=}B \wedge replace(A,B,F,H).$$

$$replace(A,B,C,D) \wedge C{=}[E|F] \wedge E{\neq}A \Leftrightarrow C{=}[E|F] \wedge D{=}[G|H] \wedge G{=}E \wedge E{\neq}A \wedge replace(A,B,F,H).$$

$$replace(A,B,C,D) \wedge D{=}[G|H] \wedge G{\neq}B \Leftrightarrow C{=}[E|F] \wedge D{=}[G|H] \wedge G{=}E \wedge E{\neq}A \wedge G{\neq}B \wedge replace(A,B,F,H).$$

$$replace(A,B,C,D) \wedge C{=}[E|F] \wedge D{=}[G|H] \wedge G{\neq}E \Leftrightarrow C{=}[E|F] \wedge D{=}[G|H] \wedge E{=}A \wedge G{=}B \wedge G{\neq}E \wedge replace(A,B,F,H).$$

The symbolic construction algorithm constructs the rules by direct derivation from the definition. The first two rules apply when the lists are empty. The last four rules apply when information is known about either of the leading list elements E or G or when the relationship between them is sufficiently known. The rules do not cover all possibilities, however they represent a good basis for a constraint solver for $replace(A,B,C,D)$.

4 Combined Approach

Both the symbolic construction method and the generate and test method have advantages and disadvantages. The symbolic construction method is able to generate recursive rules where all other approaches based on generation and testing failed. However, the generate and test, in general, generates a more exhaustive set of rules.

In this section, we will present a combination of the symbolic construction method and the generate and test method that will lead to more powerful and expressive constraint solvers at a reduced cost of generation.

We will first construct semantically valid rules using the symbolic construction method then we will use the generated rules to prune the search tree of the generate and test method using the closure pruning technique. However, even with this pruning technique, the combined approach generates redundant rules that should be removed. This will be done using the second pruning technique.

1. *Closure Pruning*: If a rule of the form $C \Leftrightarrow D$ is generated using the symbolic construction algorithm then in the generate and test method there is no need to consider rules where the lhs constraint is C. Thus, during the enumeration of all possible rule lhs, unnecessary lhs candidates are removed from this list. For efficiency reasons, the concrete implementation is not based on a list but on a tree containing lhs candidates on its nodes.
2. *Redundancy Pruning*: To suppress the generation of redundant rules, we use the algorithm proposed in [1]. The idea of the algorithm is based on operational equivalence of programs. The operational equivalence test for redundancy removal is to check if the computation step due to the candidate rule that is tested for redundancy can be performed by the remainder of the program. This is done by executing the prefix of the candidate rule in both programs and comparing the results. If the results are identical, then the rule is obviously redundant and can be removed.
   A redundant rule is defined formally as follows:

   Definition 2. *A rule R is redundant in a program P if and only if for all states S: If $S \mapsto^*_P S_1$ then $S \mapsto^*_{P\setminus\{R\}} S_2$, where S_1 and S_2 are final states and are identical upto renaming of variables and logical equivalence of built-in constraints. $\mapsto^*_P$ denotes the reflexive and transitive closure of $\mapsto_P$.*

   The redundancy pruning technique is non-deterministic since the resulting solver may vary depending on the order in which rules are tried and removed.

Example 12. For the *min* constraint of Example 1, the symbolic construction method generates the following rules:

$$min(A,B,C) \ \wedge \ A{\leq}B \ \Leftrightarrow \ A{\leq}B \ \wedge \ C{=}A. \tag{6}$$

$$min(A,B,C) \ \wedge \ C{\neq}B \ \Leftrightarrow \ A{\leq}B \ \wedge \ C{=}A \ \wedge \ C{\neq}B. \tag{7}$$

$$min(A,B,C) \ \wedge \ A{>}B \ \Leftrightarrow \ A{>}B \ \wedge \ C{=}B. \tag{8}$$

$$min(A,B,C) \ \wedge \ C{\neq}A \ \Leftrightarrow \ A{>}B \ \wedge \ C{=}B \ \wedge \ C{\neq}A. \tag{9}$$

The generate and test algorithm will first generate the propagation rule (Rule 1). Using the closure pruning technique, Rules 2, 3, and 4 are not checked. Rule 5 will be generated since there is no rule that checks for $B \leq A$. Combining both sets of rules, Rule 8 will be eliminated using the redundancy pruning technique since it is covered by Rule 5. The combined approach generates the same rules as the ones generated using the generate and test method however less candidate rules are checked.

In general, the set of rules generated using the combined approach is more expressive and powerful than the ones generated either using the generate and test method or using the symbolic construction method as illustrated in the following example.

Example 13. For the *append* constraint, the combined approach generates the following rules using the symbolic construction method:

$$append(A,B,C) \ \wedge \ A{=}[] \Leftrightarrow C{=}B \ \wedge \ A{=}[].$$

$$append(A,B,C) \ \wedge \ C{=}[] \Leftrightarrow C{=}B \ \wedge \ A{=}[] \ \wedge \ C{=}[].$$

$$append(A,B,C) \ \wedge \ A{\neq}[] \Leftrightarrow A{=}[D|E] \ \wedge \ C{=}[F|G] \ \wedge \ D{=}F \ \wedge \ append(E,B,G).$$

$$append(A,B,C) \ \wedge \ C{\neq}B \Leftrightarrow A{=}[D|E] \ \wedge \ C{=}[F|G] \ \wedge \ D{=}F \ \wedge \ append(E,B,G) \ \wedge \ C{\neq}B.$$

Then the following rules (among others) will be added from the generate and test method:

$$append(A,B,C) \ \wedge \ B{=}[] \ \Leftrightarrow \ A{=}C \ \wedge \ B{=}[]. \tag{10}$$

$$append(A,B,C) \ \wedge \ A{=}C \ \Leftrightarrow \ B{=}[] \ \wedge \ A{=}C. \tag{11}$$

$$append(A,B,C) \ \wedge \ B{\neq}[] \ \Rightarrow \ A{\neq}C. \tag{12}$$

Adding these rules improves the efficiency of the solver. For example, with Rule 10 the recursion over the list `A` is replaced by a simple unification $A = C$ if list B is empty.

Implementation in CHR The head of the generated rules may contain constraints that are built-in constraints for the CHR system. To have a running CHR solver, these constraints should be removed from the head. This is done in two steps:

- Equality constraints appearing in the head of a rule are propagated all over the constraints in the head and body of the rule. Then the resulting constraints are simplified. This can be performed as follows. In turn, each equality constraint appearing in the head is removed and transformed in a substitution that is applied to the head and body.
- For other built-in constraints, the transformation leads to guarded CHR rules [8].

Example 14. The following simplification rule for min:

$$min(A,B,C) \ \wedge \ B{\leq}A \Leftrightarrow B{\leq}A \ \wedge \ C{=}B.$$

will be transformed to the following guarded CHR simplification rule:

$$min(A,B,C) \Leftrightarrow B{\leq}A \mid C{=}B.$$

Equivalent Definitions – Same Solvers The generate and test method is based on enumerating rule candidates and checking their validity against the intentional definition. Thus, having two equivalent definitions the generate and test will generate always the same set of rules.
However, using the symbolic construction method, the set of generated rules for a constraint may differ for different but equivalent definitions of the constraint.

The following example will show that the more compact the set of clauses is, the more expressive the constructed solver is. This is intuitively clear since the construction method generates rules for a clause by negating the bodies of all other clauses which are added to the head and the body of the rule. In general, negating more than a clause will lead to adding more than one constraint to the head of the rule making it more restrictive.

Example 15. The constraint *min* of Example 1 can be defined by an equivalent CLP program consisting of three clauses instead of two as follows:

$$min(A,B,C) \leftarrow A{<}B \wedge C{=}A.$$
$$min(A,B,C) \leftarrow A{>}B \wedge C{=}B.$$
$$min(A,B,C) \leftarrow A{=}B \wedge C{=}A.$$

The symbolic construction algorithm generates the following set of simplification rules:

$$min(A,B,C) \wedge A{<}B \Leftrightarrow C{=}A \wedge A{<}B. \quad (13)$$
$$min(A,B,C) \wedge A{>}B \Leftrightarrow C{=}B \wedge A{>}B. \quad (14)$$
$$min(A,B,C) \wedge A{=}B \Leftrightarrow C{=}A \wedge C{=}B \wedge A{=}B. \quad (15)$$
$$min(A,B,C) \wedge C{\neq}A \Leftrightarrow C{=}B \wedge A{>}B \wedge C{\neq}A. \quad (16)$$
$$min(A,B,C) \wedge C{\neq}B \wedge A{\neq}B \Leftrightarrow C{=}A \wedge A{<}B \wedge C{\neq}B. \quad (17)$$
$$min(A,B,C) \wedge C{\neq}B \wedge A{\geq}B \Leftrightarrow false. \quad (18)$$

Although the number of generated rules has increased compared to the set of rules presented in Example 1, these rules are less expressive since:

- Rule 6 subsumes the two rules 13 and 15. Whereas Rule 6 will be applied for the goal $min(A,B,C) \wedge A{\leq}B$, no rule is applicable using the rules above.
- Rule 7 of the first solver is more general than its counterparts, Rule 17 and Rule 18.

Using the combined approach, all rules of the generate and test method will be added except Rule 16 which will not be checked or generated. Using the redundancy pruning technique, all rules of the construction method will be removed except Rule 16. The resulting solver of the combined approach is identical to the solver generated for the *min* constraint defined using two clauses. However, it should be noted that the solver obtained by construction using only two clauses pruned the search tree better.

5 Conclusion

In this paper, we have extended the work done in the field of *Inductive Constraint Solving* by providing a method that combines the advantages of the generate and test approach with a new method that is based on symbolic rewriting of CLP programs. The basic idea of the construction method stems from the observation

that in general, the execution of one clause in a CLP program excludes the execution of all other clauses. As such, our algorithm constructs the rules by taking the body of each clause given in CLP program as the body of a rule. The head of the rule is then the head of the clause and the negation of the body of all clauses that are not in the rule body.
In the combined approach, we first generate rules using the symbolic construction method then we use them to prune the search tree of the generate and test method. In general, the combined approach leads to more expressive and efficient constraint solvers at a reduced cost. Some rules, like recursive rules that cannot be generated using the generate and test method are generated using the symbolic construction method.
One interesting direction for future work is to investigate the completeness of the solvers generated. It is clear that in general this property cannot be guaranteed, but in some cases it should be possible to check it, or at least to characterize the kind of consistency the solver can ensure.

References

1. S. Abdennadher and T. Frühwirth. Integration and Optimization of Rule-based Constraint Solvers. In *International Symposium on Logic-based Program Synthesis and Transformation, LOPSTR03*, LNCS. Springer, 2004.
2. S. Abdennadher and C. Rigotti. Automatic Generation of Propagation Rules for Finite Domains. In *6th International Conference on Principles and Practice of Constraint Programming, CP00*, LNCS 1894. Springer-Verlag, 2000.
3. S. Abdennadher and C. Rigotti. Towards Inductive Constraint Solving. In *7th International Conference on Principles and Practice of Constraint Programming, CP01*, LNCS 2239, pages 31–45. Springer-Verlag, 2001.
4. S. Abdennadher and C. Rigotti. Automatic Generation of CHR Constraint Solvers. *Journal of Theory and Practice of Logic Programming (TPLP)*, 5(2), 2005.
5. K. Apt and E. Monfroy. Automatic Generation of Constraint Propagation Algorithms for Small Finite Domains. In *5th International Conference on Principles and Practice of Constraint Programming, CP99*, LNCS 1713. Springer-Verlag, 1999.
6. P. Codognet. A Tabulation Method for Constraint Logic Programs. In *8th Symposuim and Exibition on Industrial Applications of Prolog*, 1995.
7. B. Cui and D. S. Warren. A System for Tabled Constraint Logic Programming. In *1st International Conference on Computational Logic*, LNCS 1861. Springer-Verlag, 2000.
8. T. Frühwirth. Theory and Practice of Constraint Handling Rules, Special Issue on Constraint Logic Programming. *Journal of Logic Programming*, 37(1-3):95–138, 1998.
9. C. Ringeissen and E. Monfroy. Generating Propagation Rules for Finite Domains: A Mixed Approach. In *New Trends in Constraints*, LNAI 1865, pages 150–172, 2000.

A Scalable Inclusion Constraint Solver Using Unification

(*Extended Abstract*)

Ye Zhang and Flemming Nielson

Informatics and Mathematical Modelling, Technical University of Denmark
— {yez,nielson}@imm.dtu.dk

Abstract. We present a framework for data flow analyses that enables users to achieve best balance between efficiency and precision by using unification of equality constraints over analysis variables. A systematic evaluation of performance of reaching definitions analysis is conducted and is compared with that of a state-of-the-art solver, the Succinct Solver. The result shows our solver significantly outperforms the Succinct Solver on performance. Using unification decreases the asymptotic complexity even down to almost linear from more than quadratic time in some benchmarks.

1 Introduction

Program analyses are often expressed as a collection of constraints and then implemented by an existing solver. The strategy separates analysis specification from implementation and thus enables program analysis designers to share the insights and efforts in solver technology. But it remains a challenge to develop an high-performance analysis . For instance, to speed up its computation the Succinct Solver [18] adopts former insights of state-of-the-art solvers [5, 8, 7], including the use of recursion, continuations, prefix tree and memorization. On the other hand, the solver consumes a large amount of memory to maintain its complex data structures. This becomes a problem for large programs and can significantly decrease efficiency as observed in [22].

We aim at developing a scalable solver by introducing unification of the equality constraint over analysis variables. This is based on two insights. First, the analysis result of unification is always sound with respect to that of set-inclusion. Second, unification can be solved in almost linear time and reduce memory consumption as explained later. Actually our experimental results show that our solver significantly outperforms the Succinct Solver: It is at least 30 times faster and in some cases even 200 times faster than the Succinct Solver while it consumes much less memory. A substantially lower asymptotic complexity is also observed in some benchmarks by using unification. Although equality constraints cannot model the direction of data flow and hence may lead to a loss in precision, our experiments show that a high-level of precision could still be expected for many programs.

2 Inclusion Constraint Language

We consider constraints over (finite) sets of tuples of constants. A constraint clause **P** is defined by the following grammar:

$$\varphi ::= c \subseteq \alpha \mid \beta \subseteq^{\mathsf{s}} \alpha \mid \beta \subseteq^{\mathsf{e}} \alpha \mid \alpha \setminus c \subseteq \beta \mid \alpha \setminus [D] \subseteq \beta \mid \alpha \cap \beta \subseteq \gamma \mid \varphi_1 \wedge \varphi_2$$
$$D ::= ? \mid ?, D \mid m \mid m, D$$

where the constant $c \in \widehat{\mathbf{Const}}$ is a set of tuples consisting of a list of abstract elements $m \in \mathbf{E}$, and where α, β and $\gamma \in \mathbf{AVar}$ are analysis variables. The two new operators $\subseteq^{\mathsf{s}}$ and $\subseteq^{\mathsf{e}}$ are considered as subset-inclusion and equality respectively. The superscript functions as a pointer marking where a set-inclusion can be changed to equality and vise versa. For set-minus constraint, besides removing normal constants we introduce a new syntax category D to represent a set of tuples: The value of some positions of all these tuples are fixed and the rest can be any elements (represented by '?'). This syntax category not only generates a succinct coding but also speeds up constraint solving as further illustrated in Section 4. It is possible to express the constraints $\alpha \cup \beta \subseteq^{\mathsf{s}} \gamma$ and $\alpha \subseteq^{\mathsf{s}} \beta \cap \gamma$ in terms of more primitive operations, e.g. $\alpha \cup \beta \subseteq^{\mathsf{s}} \gamma$ is equivalent to $\alpha \subseteq^{\mathsf{s}} \gamma \wedge \beta \subseteq^{\mathsf{s}} \gamma$; thus they are not included among the primitive operations. We dispense with the union on the right hand side $\alpha \subseteq^{\mathsf{s}} \beta \cup \gamma$ since it would destroy the Moore family property discussed below.

Standard Interpretation. Given an interpretation $\hat{\psi} \in \widehat{\mathbf{Env}}$, which maps analysis variables to constants, for a clause φ the satisfaction relation $\hat{\psi} \models \varphi$ is specified as in Table 1. We here extend the meaning of set-minus to cover the new operation $S \setminus [D]$ by

$$S \setminus [m_1, \cdots, m_{i_1-1}, ?, m_{i_1+1}, \cdots, m_{i_k-1}, ?, m_{i_k+1}, \cdots, m_n] =$$
$$S \setminus \{(m_1, \cdots, m_{i_1-1}, \ell_1, m_{i_1+1}, \cdots, m_{i_k-1}, \ell_k, m_{i_k+1}, \cdots, m_n) \mid \ell_1, \cdots, \ell_k \in \mathbf{E}\}$$

We are in general interested in a least solution to a clause. Consider, for instance, a constraint

$$\{(a, b)\} \subseteq \alpha \wedge \alpha \subseteq^{\mathsf{s}} \beta \wedge \{(c, d)\} \subseteq \beta \qquad (\textbf{Ex.1})$$

It is easy to verify that $\hat{\psi}$ given by $\hat{\psi}(\alpha) = \{(a, b)\}$ and $\hat{\psi}(\beta) = \{(a, b), (c, d)\}$ is a solution. Actually it is the least solution. Also any estimate $\hat{\psi}'$ such that $\hat{\psi} \sqsubseteq \hat{\psi}'$ satisfies **Ex.1** (where we use the standard partial order $\sqsubseteq$ on the mappings of $\widehat{\mathbf{Env}}$, formally, for $\hat{\psi}, \hat{\psi}' \in \widehat{\mathbf{Env}} : \hat{\psi} \sqsubseteq \hat{\psi}'$ iff $\forall x \in \mathbf{AVar} : \hat{\psi}(\alpha) \subseteq \hat{\psi}'(\alpha)$). Luckily we can show that a unique least solution always exists for constraint clauses.

Theorem 1. *For each clause φ, the set $\{\hat{\psi} \mid \hat{\psi} \models \varphi\}$ is a Moore family.*

Intuitively using equality instead of set-inclusion is safe since equality is more strict than set-inclusion. However equality constraints do not always lead to a

1. $\hat{\psi} \models c \subseteq \alpha$	iff $c \subseteq \hat{\psi}(\alpha)$
2. $\hat{\psi} \models \beta \subseteq^{\mathsf{s}} \alpha$	iff $\hat{\psi}(\beta) \subseteq \hat{\psi}(\alpha)$
3. $\hat{\psi} \models \beta \subseteq^{\mathsf{e}} \alpha$	iff $\hat{\psi}(\beta) = \hat{\psi}(\alpha)$
4.1 $\hat{\psi} \models \alpha \setminus c \subseteq \beta$	iff $\hat{\psi}(\alpha) \setminus c \subseteq \hat{\psi}(\beta)$
4.2 $\hat{\psi} \models \alpha \setminus [m_1, \cdots, m_{i_1-1}, ?, m_{i_1+1}, \cdots, m_{i_k-1}, ?, m_{i_k+1}, \cdots, m_n] \subseteq \beta$	iff $\hat{\psi}(\alpha) \setminus [m_1, \cdots, m_{i_1-1}, ?, m_{i_1+1}, \cdots, m_{i_k-1}, ?, m_{i_k+1}, \cdots, m_n] \subseteq \hat{\psi}(\beta)$
5. $\hat{\psi} \models \alpha \cap \beta \subseteq \gamma$	iff $\hat{\psi}(\alpha) \cap \hat{\psi}(\beta) \subseteq \hat{\psi}(\gamma)$
6. $\hat{\psi} \models \varphi_1 \wedge \varphi_2$	iff $\hat{\psi} \models \varphi_1$ and $\hat{\psi} \models \varphi_2$

Table 1. Standard Semantics

loss in precision. To be concrete, consider the constraints

$$\{(a,b),(c,d)\} \subseteq \alpha \wedge \{(c,d)\} \subseteq \beta \wedge \alpha \subseteq^{\mathsf{s}} \beta \wedge \beta \setminus \{(a,b)\} \subseteq \gamma \qquad (\textbf{Ex.2})$$

Here a least model β has no more data than α apparently whence switching s to e in the constraint would preserve precision. As we shall show in Section 3, general constraints can be solved in cubic time while unification on equality is nearly linear. In the framework we presented, a general strategy of tuning clauses is to try set-inclusion first since normally we would always prefer a precise solution if performance is acceptable. If the efficiency of the computation is unsatisfactory, we can tentatively adjust the superscript symbols and repeat the procedure until we reach a good balance between performance and precision.

Interpretation Using Type Variables. If several analysis variables are equivalent to each other, we would like to deal with them as one and let them associate with the same data field. As a result the constraint solving is more efficient and the space-consumption is decreased. This motivates us to introduce a new category, type variables $i \in \mathbf{TV}$, as a medium between analysis variables and constants, i.e. an interpretation consists of two components: Type environment $\hat{\psi}_1 \in \mathbf{Env_T}$, which maps analysis variables to type variables, and type-binding environment $\hat{\psi}_2 \in \widehat{\mathbf{Env_{TB}}}$, which maps type variables to constants. The satisfaction relation $(\hat{\psi}_1, \hat{\psi}_2) \models_{\mathcal{T}} \varphi$ is then defined in Table 2.

The only interesting rule is the third one that enforces α and β must be unified onto the same type variables, i.e. $\hat{\psi}_1(\beta) = \hat{\psi}_1(\alpha)$, instead of demanding $\hat{\psi}_2(\hat{\psi}_1(\beta)) = \hat{\psi}_2(\hat{\psi}_1(\alpha))$. To further illustrate the difference, consider the estimates of the lifted version of the example **Ex.1**,

(a)	$\hat{\psi}_1(\alpha) = 1$	$\hat{\psi}_2(1) = \{(a,b),(c,d)\}$	(b)	$\hat{\psi}_1(\alpha) = 1$	$\hat{\psi}_2(1) = \{(a,b),(c,d)\}$
	$\hat{\psi}_1(\beta) = 1$			$\hat{\psi}_1(\beta) = 2$	$\hat{\psi}_2(2) = \{(a,b),(c,d)\}$
	$\hat{\psi}_1(\gamma) = 2$	$\hat{\psi}_2(2) = \{(a,b)\}$		$\hat{\psi}_1(\gamma) = 3$	$\hat{\psi}_2(3) = \{(a,b)\}$

Both of the two estimates are acceptable for the first semantics whereas only (a) is valid now since $\hat{\psi}_1(\alpha) \neq \hat{\psi}_1(\beta)$. Notice also that the unification coalesces the

1. $(\hat{\psi}_1, \hat{\psi}_2) \models_{\mathcal{T}} c \subseteq \alpha$	iff $c \subseteq \hat{\psi}_2(\hat{\psi}_1(\alpha))$
2. $(\hat{\psi}_1, \hat{\psi}_2) \models_{\mathcal{T}} \beta \subseteq^{\mathsf{s}} \alpha$	iff $\hat{\psi}_2(\hat{\psi}_1(\beta)) \subseteq \hat{\psi}_2(\hat{\psi}_1(\alpha))$
3. $(\hat{\psi}_1, \hat{\psi}_2) \models_{\mathcal{T}} \beta \subseteq^{\mathsf{e}} \alpha$	iff $\hat{\psi}_1(\beta) = \hat{\psi}_1(\alpha)$
4.1 $(\hat{\psi}_1, \hat{\psi}_2) \models_{\mathcal{T}} \alpha \setminus c \subseteq \beta$	iff $\hat{\psi}_2(\hat{\psi}_1(\alpha)) \setminus c \subseteq \hat{\psi}_2(\hat{\psi}_1(\beta))$
4.2 $(\hat{\psi}_1, \hat{\psi}_2) \models_{\mathcal{T}} \alpha \setminus [m_1, \cdots, m_{i_1-1}, ?, m_{i_1+1}, \cdots, m_{i_k-1}, ?, m_{i_k+1}, \cdots, m_n] \subseteq \beta$ iff $\hat{\psi}_2(\hat{\psi}_1(\alpha)) \setminus [m_1, \cdots, m_{i_1-1}, ?, m_{i_1+1}, \cdots, m_{i_k-1}, ?, m_{i_k+1}, \cdots, m_n] \subseteq \hat{\psi}_2(\hat{\psi}_1(\beta))$	
5. $(\hat{\psi}_1, \hat{\psi}_2) \models_{\mathcal{T}} \alpha \cap \beta \subseteq \gamma$	iff $\hat{\psi}_2(\hat{\psi}_1(\alpha)) \cap \hat{\psi}_2(\hat{\psi}_1(\beta)) \subseteq \hat{\psi}_2(\hat{\psi}_1(\gamma))$
6. $(\hat{\psi}_1, \hat{\psi}_2) \models_{\mathcal{T}} \varphi_1 \wedge \varphi_2$	iff $(\hat{\psi}_1, \hat{\psi}_2) \models_{\mathcal{T}} \varphi_1$ and $(\hat{\psi}_1, \hat{\psi}_2) \models_{\mathcal{T}} \varphi_2$

Table 2. Semantics Using Type Variables

analysis variables into one type variable whence avoids storing redundant information in the environment $\widehat{\mathbf{Env_{TB}}}$. However since the choice of type variables is nondeterministic, the set of solutions is not a partial-order but a pre-order, i.e. reflexive, transitive but not antisymmetric. Formally

Definition 1. *For* $(\hat{\psi}_1, \hat{\psi}_2), (\hat{\psi}'_1, \hat{\psi}'_2) \in \mathbf{Env_T} \times \widehat{\mathbf{Env_{TB}}}$*, define*

$$(\hat{\psi}_1, \hat{\psi}_2) \preceq (\hat{\psi}'_1, \hat{\psi}'_2) \Longleftrightarrow \exists \pi : \mathbf{TV} \to \mathbf{TV} : \hat{\psi}'_1 = \pi \circ \hat{\psi}_1 \quad \wedge \quad \hat{\psi}_2 \sqsubseteq \hat{\psi}'_2 \circ \pi$$

where π *is a total function and* $\hat{\psi}_2 \sqsubseteq \hat{\psi}'_2 \circ \pi \Leftrightarrow \forall\, i \in \mathbf{TV} : \hat{\psi}_2(\alpha) \subseteq \hat{\psi}'_2(\pi(i))$.

Although a unique least solution is not guaranteed, we can generalize the concepts of complete lattice and Moore family for a pre-ordered set. The overall idea here is to prove the existence of the least solutions for the set $\{(\hat{\psi}_1, \hat{\psi}_2) | \, (\hat{\psi}_1, \hat{\psi}_2) \models_{\mathcal{T}} \varphi\}$ and these least solutions are actually equivalent to each other (defining equivalence relation by $\preceq \wedge \succeq$). Therefore it is suffice to find just one of them. Furthermore it can be proved that the function composition $\hat{\psi}_2) \circ \hat{\psi}_1)$ of a least solution gives the least solution for the set $\{\hat{\psi} | \, \hat{\psi} \models \varphi\}$ as well. Due to the space constraints, the detail of formal development is omitted but it should not be a big surprise that there exists least solutions for each clause and they are equivalent.

3 Constraint-solving Algorithm

We shall give a high-level introduction to our algorithm and concentrate on explaining how the algorithm works. The algorithm is structured with two phases: Unification and iteration. Initially each analysis variable is assigned a unique type variables and is associated with an empty set. At first unification phase deals with all the equality constraints. It computes the set of equivalence class of analysis variables according to the equality constraints. As illustrated in **Ex.3**, for each equivalence class A we chose the corresponding type variable of any one of its analysis variables, say t, and associate all the equivalent analysis variables with t, i.e. $\forall \alpha \in A : \hat{\psi}_1(\alpha) = t$.

For the iteration phase, we refer to the approach of [17] for constructing a graph formulation of constraints. The algorithm iteratively propagates any enlargement of constant fields along the graph edges until a (least) fix-point reaches. A constraint graph consists of nodes $i \in \mathbf{TV}$ and directed edges decorated with constructs that give rise to them respectively. We describe specifically how the graph is built up for the convenience of the discussion of algorithm complexityV: The constructs $\alpha \subseteq^{\mathsf{s}} \beta$, $\alpha \setminus c \subseteq \beta$ and $\alpha \setminus [D] \subseteq \beta$ give rise to an edge from $\hat{\psi}_1(\alpha)$ to $\hat{\psi}_1(\beta)$; similarly the construct $\alpha \cap \beta \subseteq \gamma$ contributes two edges at the same time, i.e. from $\hat{\psi}_1(\alpha)$ to $\hat{\psi}_1(\gamma)$, and $\hat{\psi}_1(\beta)$ to $\hat{\psi}_1(\gamma)$. Note that an equality relation never generates any edge. Furthermore equality could simplify graphs. For example, assume that $\alpha \subseteq^{\mathsf{e}} \gamma$, then the edge from α to γ is not needed for any of the constraints $\alpha \subseteq^{\mathsf{s}} \gamma, \alpha \setminus c \subseteq \gamma$ or $\alpha \setminus D \subseteq \gamma$ and $\alpha \cap \beta \subseteq \gamma$.

Algorithm Complexity. Observe that for a clause of size n there are $O(n)$ nodes and $O(n)$ constructs at most. In the unification phase, fast union-find data structures [21] is used to compute the equivalence classes of analysis variables and it takes time $O(m \cdot \alpha(m, n))$ where m is the number of unifying operations that is bound to $O(n)$ and n the number of analysis variables. At last $O(n)$ operations are needed to re-associate equivalence analysis variables with a designated type variable. This completes the complexity analysis of unification phase.

To make a sensible analysis for the iteration phase, fist we need to make it clear how many steps are required for the operations upon constants (set of tuples), e.g. set intersection and union, etc. In our implementation, each tuple is encoded as a bit and the number of tuples is $O(n)$; thus the set operations are over the bit-vector of the length n and take linear time. Next observe the two facts: (1) there are $O(n)$ edges generated from a clause of size n by the way of graph construction and (2) each edge can be traversed at most $O(n)$ times as there are $O(n)$ nodes. Therefore let n_i be the edge number bound to the node i and the time of iteration is $O(\Sigma_{i \in \mathbf{TV}_\star}(n \cdot n_i \cdot n)) = O(n^3)$ where the fist n is the number of traverse on each edges and the second is time of set operations.

4 Experiment with a Data Flow Analysis

In this section we study the effects of applying unification through an intraprocedural Data Flow Analysis for a simple C-like language. We demonstrate how much improvement on performance can be achieved from using unification. A series of experiments are conducted on our solver and the Succinct Solver. The fact that both of the two solvers are implemented in NJ SML makes the comparison more reliable. A program statement is defined by

$$S ::= [x := a]^\ell \mid [\mathsf{skip}]^\ell \mid S_1; S_2 \mid \text{if } [b]^\ell \text{ then } S_1 \text{ else } S_2 \mid \mathsf{while}\ [b]^\ell\ \mathsf{do}\ S$$

Here $x \in \mathbf{Var}$ is a program variable, a and b are arithmetic and boolean expressions respectively. Each elementary block is assigned a *unique* label $\ell \in \mathbf{Lab}$. We refer to [17] for the operational semantics of the language. We give analysis specification for reaching definitions analysis (in Table 3) and the approach

[ass]	$(RD_\circ, RD_\bullet) \models [x := e]^l$	iff $\{(x,l)\} \subseteq RD_\bullet(l) \wedge$ $RD_\circ(l) \setminus [x,?] \subseteq RD_\bullet(l)$	
[skip]	$(RD_\circ, RD_\bullet) \models [skip]^l$	iff $RD_\circ(l) \subseteq^{\mathsf{s}} RD_\bullet(l)$	(i)
[exp]	$(RD_\circ, RD_\bullet) \models [e]^l$	iff $RD_\circ(l) \subseteq^{\mathsf{s}} RD_\bullet(l)$	(ii)
[comp]	$(RD_\circ, RD_\bullet) \models S_1; S_2$	iff $(RD_\circ, RD_\bullet) \models S_1 \wedge$ $(RD_\circ, RD_\bullet) \models S_2 \wedge$ $\wedge_{\forall l \in \mathsf{final}(S_1)} RD_\bullet(l) \subseteq^{\mathsf{s}} RD_\circ(\mathsf{init}(S_2))$	(iii)
[if]	$(RD_\circ, RD_\bullet) \models \text{if } [b]^l \text{ then } S_1 \text{ else } S_2$	iff $(RD_\circ, RD_\bullet) \models S_1 \wedge$ $(RD_\circ, RD_\bullet) \models S_2 \wedge$ $(RD_\circ, RD_\bullet) \models b \wedge$ $RD_\bullet(l) \subseteq^{\mathsf{s}} RD_\circ(\mathsf{init}(S_1)) \wedge$ $RD_\bullet(l) \subseteq^{\mathsf{s}} RD_\circ(\mathsf{init}(S_2))$	(iv) (v)
[wh]	$(RD_\circ, RD_\bullet) \models \text{while } [b]^l \text{ do } S$	iff $(RD_\circ, RD_\bullet) \models S \wedge$ $(RD_\circ, RD_\bullet) \models b \wedge$ $RD_\bullet(l) \subseteq^{\mathsf{s}} RD_\circ(\mathsf{init}(S)) \wedge$ $\wedge_{\forall l' \in \mathsf{final}(S)} RD_\bullet(l') \subseteq^{\mathsf{s}} RD_\circ(l)$	(vi) (vii)

Table 3. Reaching Definitions Analysis: Set Inclusion.

applies naturally on other analyses, e.g. available expressions analysis and live variables analysis, etc. The judgement $(RD_\circ, RD_\bullet) \models S$ is true if and only if the analysis is satisfiable for S. The caches $RD_\circ, RD_\bullet : Lab \times \mathcal{P}(\mathbf{Var} \times \mathbf{Lab})$ record the estimates of the entry and exit of elementary statements.

The two auxiliary functions $\mathsf{initial}$ and final return the initial label and the set of final labels of a statement respectively; for instance for the while loop, while $[b]^l$ do S, the initial label is l and the set of final labels is $\{l\}$. Note that we use the extended version of set-minus in [ass]. As a result the size of the constraint generated for each assignment is constant; otherwise it could be linear for just one assignment.

4.1 Experiments on Scalable Programs

We designed a series of benchmarks to testing the scalability of the two solvers. The two families of the benchmarks are selected for detailed discussion since their asymptotic complexities are considered typical for presenting all the interesting results.

$\mathrm{Wh}_{(1,n)}$: while $x_0 < 2$ do ($x_1 := x_2$; $\vdots$ $x_{n-1} := x_n$; $x_n := 1$)

$\mathrm{If}_{(n,1)}$: if $x_1 < 0$ then skip else $\vdots$ if $x_n < 0$ then skip else $x_0 := 1$

Here the first number of the subscript denotes the nesting depth of conditions, and the second the number of all assignments (usually at the deepest level).

Fig. 1. Experimental results: $\mathrm{Wh}_{(1,n)}$ and $\mathrm{If}_{(n,1)}$.

From the analysis specification, the constraints generated for $\mathrm{Wh}_{(1,n)}$ and $\mathrm{If}_{(n,1)}$ are both of size $O(n)$. Indeed, given the method used for constructing the graph in the algorithm, it can be shown that both of the graphs have $O(n)$ edges using an amortization technique.

As presented in Section 3, adopting unification will simplify the graph and further reduce the number of iterations. We show, however, that the results of this simplification differ for the two benchmarks very much: For the first one, it decreases the number of edges by a constant factor; in contrast, for the second there is only a constant number of edges left. This is because the constraints generated for n assignments of $\mathrm{Wh}_{(1,n)}$ remain $O(n)$ by the rule [ass]. But applying unification to $\mathrm{If}_{(n,1)}$ means we only keep set-inclusion in the constraints for one assignment(s) and thus the resulting graph has only a constant number of edges.

All benchmarks have been run on a PC with 2.0 GHz CPU and 1.5 GB RAM under Windows XP. Each experiment was repeated 5 times and the average was used. The experimental results of the two families are presented in Fig. 1.

The first diagram shows that the execution time is improved 25% by using unification and the computation using set-inclusion is at least 70 times and sometimes even 200 times faster than the Succinct Solver. Our solver also scale much larger program than the Succinct Solver. Moreover the Succinct Solver can not analyze the program of $n \geq 1300$. Notice that both of the solvers suffer a sharp performance-decline for large values of n: $n \geq 750$ in the case of the Succinct Solver, and $n \geq 9000$ and $n \geq 11000$ in the case of our solver. We hypothesize that this is because the large memory consumption requires much extra effort in memory management. For our solver especially, the computation time is so small when n is less than 250 that the initialization time becomes a large constant factor impacting the asymptotic complexity. To get the asymptotic growth rate of the solvers, we select the data before performance deterioration happens and after the constant factor is no longer dominating. By a least square fit technique on the model $t = c_1 \cdot m^c + c_0$, we estimate that the time complexity of the Suc-

cinct Solver, and our solver without and with unification are $O(n^{2.21}), O(n^{2.02})$ and $O(n^{2.01})$ respectively.

A significant improvement, as expected, is observed in the program family $\text{If}_{(1,n)}$ (the second graph of Fig. 1. As shown our solver remains 30 times faster than the Succinct Solver when using unification. Since no performance-deterioration is observed, the estimated complexities are printed out directly. Furthermore unification results in almost linear time complexity while set-inclusion takes more than quadratic time and the Succinct Solver takes time $O(n^{1.3})$.

We turn to the precision now. Since we are doing over-approximation and all results obtained from using unification are sound, the imprecise caused by unification can be directly observed by enlargement of the size of the solutions as summarized in the table below, where $n = 4000$.

Program	Sol. Size ($\subseteq^s$)	Sol. Size ($\subseteq^e$)	
$\text{Wh}_{(1,n)}$	30487128	30487128	√
$\text{Wh}_{(n,1)}$	32014891	32014891	√
$\text{Wh}_{(n,n)}$	62551110	62551110	√
$\text{If}_{(1,n)}$	30423193	30423193	√

Program	Sol. Size ($\subseteq^s$)	Sol. Size ($\subseteq^e$)	
$\text{If}_{(n,1)}$	64014890	64014890	√
$\text{If}_{(n,n)}$	94423189	94423189	√
$\text{WhIf}_{(n,1,1)}$	32030895	32030895	√
$\text{IfWh}_{(n,1,1)}$	63998893	64014889	

As the table shows, for most cases precision is preserved (denoted by √). The only exception is $\text{IfWh}_{(n,1,1)}$ because it contains a branch of an if-statement starting with a while-loop. The imprecision, however, can easily be removed by turning back to using set-inclusion for the branch containing the while statement, i.e. only one equality needs to be changed back to set-inclusion in our case.

To summarize, our worked example shows that using unification can speed up calculation by at least 30% and in some cases decrease the time complexity to almost linear. At the same time, we can avoid loss of precision. Compared with the Succinct Solver, our solver is much more efficient: It is at least 30 times faster than the Succinct Solver. The experiments on other benchmarks show the performance improvement is proportional to the percentage of equality constraints adopted in a constraint program.

5 Related Works

In this paper we have looked at analysis problems over a finite universe and have computed complete solutions. The Succinct Solver, which uses the alternation-free fragment of *Least Fixpoint Logic* (ALFP) as its specification logic, works on the same universe as ours. Because of the expressiveness of ALFP, the solver has been used for the implementation of many analyses [19, 9, 2]. The result of [3] shows that reordering constraints can improve the performance considerably. In order to generate efficiently solvable constraints, however, one needs to understand how clauses are solved in the Succinct Solver. We here, however, attempt to optimize the performance of our solver from the user point of view by simply adjusting the use of set-inclusion and equality. The users, therefore, do not have to know any technical details inside a solver but are still able to tune a system to fit their specific needs. While all of our constraints can be expressed in

ALFP, i.e. they are a restricted form of Horn clauses, we gain much efficiency in having equality constraint explicitly. To construct the same analysis, specifically data flow analyses, we observed our solver is a large constant factor better than the Succinct Solver. That is probably because we use much more simpler data structures and that reduces the expense of operating them in turn.

Unification has been used to yield efficient implementation and concise results in the analyses, such as type inference system [16, 13, 20] and control flow analyses [11]. The work of [6] further presented a parameterized framework that allows expression of constraint-based analyses [1, 14] in varying levels of efficiency and precision with mixed-terms. While our approach is close in spirit to this framework, we confine ourselves to the flat universe which corresponds to allowing only term constructors of arity 0. Even with this restricted domain it is still powerful enough to model most data flow analysis problems.

Heintze and Jaffar [10] have investigated definite set constraints and showed all satisfiable constraints in the class have a least model. Charatonik and Podelski [4] further showed solving definite set constraints has DEXPTIME complexity. Although the set minus operation, which contains negative set expression, i.e. $\alpha \backslash c \equiv \alpha \cap \neg c$, makes our constraints be out of the scope of definite set constraints, we showed that the Moore family result still holds for the constraints of interest. Melski and Rep [15] proved a subclass of definite set constraints can be solved in cubic time in studying a simple data-flow reachability problem. While their constraints use only projection and terms, we selected to include operations set minus and intersection on a flat universe and showed it also have the same complexity.

In essence our solver computes a dynamic transitive closure except for equality constraints. Heintze and McAllester have showed that this problem is in the class two way nondeterministic push down automata (2NPDA) and is 2NPDA-hard [12]. Thus it is considered inherently cubic as no sub-cubic algorithm for any 2NPDA problem is known so far. This is also confirmed by the result of Section 3. As demonstrated in the same section, however, unification can remove edges and nodes from a graph and thus speed up the calculation. Then the question is really what is the tradeoff from using unification. Our results showed that many analysis variables can be considered equivalent even for intraprocedural data flow analysis of a simple C-like language. The level of efficiency and precision achieved are quite encouraging. We are optimistic about finding more places of using unification for the analysis of real imperative languages, e.g. C.

6 Conclusion

We have presented a framework which allows users to take advantage of unification in order to implement efficient and precise analyses. ¿From the experimental results of our worked example, we conclude that (1) our constraint solver has a much more efficient implementation for data flow analysis than the Succinct Solver; (2) using unification can lower the asymptotic complexity even to almost linear while the loss in precision is still acceptable.

References

1. A. Aiken. Introduction to set constraint-based program analysis. *Sci. Comput. Program.*, 35(2):79–111, 1999.
2. C. Bodei, M. Buchholtz, P. Degano, F. Nielson, and H. R. Nielson. Static validation of security protocols. *Journal of Computer Security*, 13(3):347–390, 2005.
3. M. Buchholtz, H. R. Nielson, and F. Nielson. Experiments with succinct solvers. Technical report, Informatics and Mathematical Modelling, Richard Petersens Plads, Building 321, DK-2800 Kgs. Lyngby, Denmark, feb 2002.
4. W. Charatonik and A. Podelski. Set constraints with intersection. *Inf. Comput.*, 179(2):213–229, 2002.
5. B. L. Charlier and P. V. Hentenryck. A universal top-down fixpoint algorithm. Technical Report CS-92-25, 1992.
6. M. Fähndrich and A. Aiken. Program analysis using mixed term and set constraints. In *SAS*, pages 114–126, 1997.
7. C. Fecht and H. Seidl. Propagating differences: An efficient new fixpoint algorithm for distributive constraint systems. In *ESOP*, pages 90–104, 1998.
8. C. Fecht and H. Seidl. A faster solver for general systems of equations. *Sci. Comput. Program.*, 35(2):137–161, 1999.
9. H. Gao. Using the succinet solver to implement flow logic specifications of classical data flow analysis. Master's thesis, Informatics and Mathematical Modelling, Technical University of Denmark, DTU, 2004.
10. N. Heintze and J. Jaffar. A decision procedure for a class of set constraints (extended abstract). In *LICS*, pages 42–51. IEEE Computer Society, 1990.
11. N. Heintze and D. A. McAllester. Linear-time subtransitive control flow analysis. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 261–272, 1997.
12. N. Heintze and D. A. McAllester. On the cubic bottleneck in subtyping and flow analysis. In *LICS*, pages 342–351, 1997.
13. F. Henglein. Global tagging optimization by type inference. In *LISP and Functional Programming*, pages 205–215, 1992.
14. J. Kodumal and A. Aiken. Banshee: A scalable constraint-based analysis toolkit. In *SAS*, pages 218–234, 2005.
15. D. Melski and T. W. Reps. Interconveritibility of set constraints and context-free language reachability. In *PEPM*, pages 74–89, 1997.
16. R. Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17(3):348–375, 1978.
17. F. Nielson, H. R. Nielson, and C. L. Hankin. *Principles of Program Analysis*. Springer, 1999.
18. F. Nielson, H. Seidl, and H. R. Nielson. A succinct solver for ALFP. *Nord. J. Comput.*, 9(4):335–372, 2002.
19. H. R. Nielson, F. Nielson, and M. Buchholtz. Security for mobility. In *FOSAD*, pages 207–265, 2002.
20. B. Steensgaard. Points-to analysis in almost linear time. In *POPL*, pages 32–41, 1996.
21. R. E. Tarjan. *Data Structures and Network Algorithms*, volume CMBS44 of *Reginal Conference Series in Applied Mathematics*. SIAM, 1983.
22. Y. Zhang. Static analysis for protocol validation in hierarchical networks. Master's thesis, Technical University of Denmark, 2005.

Annotation Algorithms for Unrestricted Independent And-Parallelism in Logic Programs

Amadeo Casas,[1] Manuel Carro,[2] and Manuel V. Hermenegildo[1,2]

{amadeo, herme}@cs.unm.edu
{mcarro, herme}@fi.upm.es

[1] Depts. of Comp. Science and Electr. and Comp. Eng., Univ. of New Mexico, USA.
[2] School of Computer Science, Universidad Politécnica de Madrid, Spain.

Abstract. We present two new algorithms which perform automatic parallelization via source-to-source transformations. The objective is to exploit goal-level, *unrestricted* independent and-parallelism. The proposed algorithms use as targets new parallel execution primitives which are simpler and more flexible than the well-known `&/2` parallel operator. This makes it possible to generate better parallel expressions by exposing more potential parallelism among the literals of a clause than is possible with `&/2`. The difference between the two algorithms stems from whether the order of the solutions obtained is preserved or not. We also report on a preliminary evaluation of an implementation of our approach. We compare the performance obtained to that of previous annotation algorithms and show that relevant improvements can be obtained.

Keywords: Logic Programming, Automatic Parallelization, And-Parallelism, Program Transformation.

1 Introduction

Parallelism capabilities are becoming ubiquitous thanks to the widespread use of multi-core processors. Indeed, most laptops on the market contain two cores (capable of running up to four threads simultaneously) and single-chip, 8-core servers are now in widespread use. Furthermore, the trend is that the number of on-chip cores will double with each processor generation. In this context, being able to exploit such parallel execution capabilities in programs as easily as possible becomes more and more a necessity. However, it is well-known [17] that parallelizing programs is a hard challenge. This has renewed interest in language-related designs and tools which can simplify the task of producing parallel programs.

The comparatively higher level of abstraction of declarative languages and, among them, logic programming languages, allows writing programs which are closer to the specification of the solution. Besides, there is often more freedom in the implementation of different operational semantics which respect the declarative semantics. In particular, the notion of control in declarative languages frequently allows for more flexibility to arrange the evaluation order of some

operations, including executing them in parallel if deemed convenient, without affecting the semantics of the original program. Additionally, the cleaner declarative semantics makes it possible to automatically detect more accurately any lack of dependencies among operations and hence to exploit opportunities for parallelism more easily than in imperative languages. At the same time, in most other respects in the case of logic programs the presence of dynamic data structures with "declarative pointers" (logical variables), irregular computations, or complex control makes the parallelization of logic programs a particularly interesting case that allows tackling the more complex parallelization-related challenges in a formally simple and well-understood context [11].

Because of this potential, automatic parallelization has received significant attention in logic programming [10], where two main forms of parallelism have been studied. *Or-parallelism* is exploited when the alternatives created by nondeterministic goals are explored simultaneously. Some relevant or-parallelism systems are Aurora [20] and MUSE [1]. *And-parallelism* aims at executing simultaneously (conjunctive) goals in clauses or in the resolvent. Examples of systems that have exploited and-parallelism are DDAS [25] and &-Prolog [12]. Additionally, some systems such as ACE [9], AKL [16], and Andorra [24] exploit certain combinations of both and- and or-parallelism. While or-parallelism can only obtain speedups when there is search involved, and-parallelism can be used in more algorithmic schemes, with divide-and-conquer and map-style algorithms being classic representatives. In this paper, we concentrate on and-parallelism.

A correct parallelization has been defined as one that preserves during and-parallel execution some key properties, typically correctness and no-slowdown [14]. The preservation of these properties is ensured by executing in parallel goals which meet some notion of *independence*, meaning that the goals to be executed in parallel do not interfere with each other in some particular sense. This can include for example absence of competition for binding variables plus other considerations such as, e.g., absence of side effects. For simplicity, in the rest of the paper we will assume that we are only dealing with side-effect free program sections. Note however that this does not affect the generality of our presentation, as we deal with dependencies in a generic way.

One of the best understood sufficient conditions for ensuring that goals meet the efficiency and correctness criteria for parallelization is *strict independence* [14], which entails the absence of shared variables at runtime between any two goals being parallelized. It should be noted that some proposals exploit and-parallelism between goals which do not meet this condition, but on which other restrictions are imposed which also ensure no-slowdown and correctness. Examples of such restrictions are determinism and non-failure [14] (determinism is exploited for example in [24]) and absence of conflicts due to the binding of shared variables (as in *non-strict* independent and-parallelism [14]). Another interesting issue is at what level of granularity the notion of independence is applied: at the goal level, at the binding level, etc. Our work in this paper will focus on *goal-level* (strict and non-strict) independent and-parallelism.

One particularly successful approach to automatically parallelizing a logic program uses three different stages [15, 2, 10]. The first one detects data (and control) dependencies between pairs of literals in the original program. A dependency graph (see Figure 1 as an example) is built to capture this information. Nodes in the graph correspond to literals in the body of the clause and edges represent dependencies between them. Edges are labeled with the associated dependency conditions (which may be trivially *true* or *false* —we will not represent those edges labeled with *true*). The second stage performs (global) analysis [3] to gather information regarding, e.g., variable aliasing, groundness, side effects, etc. in order to remove edges from the dependency graph or to simplify the conditions labeling these edges, if they cannot be evaluated statically to completion. Labeled edges will result in run-time checks if conditional parallel expressions are allowed. Alternatively, unresolved dependencies can be assumed to always hold, and parallel execution will be allowed only between literals which have been statically determined to be independent. This approach saves run-time checks at the expense of losing some parallelism. Finally, the third stage transforms the original program into a parallel version by *annotating* it with parallel execution operators using the information gathered by the analyzers [22]. This annotation should respect the dependencies found in the original program while, at the same time, exploiting as much parallelism as possible.

This annotation process is the focus of this paper. We will present and evaluate new annotation algorithms which target and-parallelism primitives which can express richer dependency graphs than those which can be encoded with the *nested fork-join* approaches which have been previously proposed (e.g., [22]). Our hope is that since the transformed programs will contain in some cases more parallelism, we will be able to obtain better speedups for such cases.

2 Background and Motivation

We will introduce, with the help of an example, the well-known `&/2` operator for parallelism and its limitations, and we will show how better annotations for parallelism are possible when other, simpler primitives, are used.

2.1 *Fork-Join*-Style Parallelization

We will use as running example the following clause:

```
p(X,Y,Z) :- a(X,Z), b(X), c(Y), d(Y,Z).
```

and will assume that the dependencies detected between the literals in the predicate are defined by the graph $G = (V, E)$, shown in Figure 1. The vertices V correspond to the literals of the clause and there exists an edge between two literals L_i and L_j in E if $ind(L_i, L_j) \neq true$ (i.e., the literals

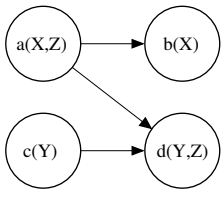


Fig. 1. Dependency graph for `p/3`.

```
p(X, Y, Z):-
        (a(X, Z), b(X)) & c(Y),
        d(Y, Z).
```

(a) *fj1*: Order-preserving

```
p(X, Y, Z):-
        a(X, Z) & c(Y),
        b(X) & d(Y, Z).
```

(b) *fj2*: Non-order-preserving

Fig. 2. *Fork-Join* annotations for `p/3` (Section 2).

L_i and L_j are dependent and thus the literal L_i has to be completed before the literal L_j), where *ind* is the notion of independence. As mentioned before, this information is obtained in our case from global data-flow analysis [3].

We will assume in the rest of the paper that all the dependencies are unconditional —i.e., conditional dependencies are assumed to be always false. This brings simplicity and avoids potentially costly run-time checks in the parallelized code at the expense of having fewer opportunities for parallelism. However, it has been experimentally found to be a good compromise [22, 3].

Conjunctive parallel execution has traditionally been denoted using the `&/2` operator instead of the sequential comma (','). The former binds more tightly than the latter. Thus, the expression "`a, b & c, d`" means that literals `b` and `c` can be safely executed in parallel after the execution of literal `a` finishes. When both `b` and `c` have successfully finished, execution continues with `d`.

While this single operator is enough to parallelize many programs, the class of dependencies it can express directly (i.e., dependency graphs with a nested fork-join structure) is a subset of that which can possibly appear in a program [22]. This makes parallelism opportunities to be inevitably lost in cases with a complex enough structure (e.g., that in Figure 1). Likewise, inter-procedural parallelism (i.e., parallel conjunctions which span literals in different predicates) cannot be exploited without program transformation.

In general, several annotations are possible for a given clause. As an example, Figure 2 shows two annotations for our running example.[3] Some goals appear switched w.r.t. their order in the sequential clause. This respects the dependencies in Figure 1, which reflects a valid notion of parallelism (i.e., if solution order is not important). If additional ordering requirements are needed (due to, e.g., side effects or impurity), these should appear as additional edges in the graph.

Note that none of the annotations in Figure 2 fully exploits all parallelism available in Figure 1: Figure 2(a) misses the parallelism between `b(X)` and `d(Y, Z)`, and Figure 2(b) misses the parallelism between `b(X)` and `c(Y)`.

One relevant question is which of these two parallelizations is better. Arguably, a meaningful measure of their quality is how long each of them takes to execute. We will term those times T_{fj1} and T_{fj2} for Figures 2(a) and 2(b), respectively. This length depends on the execution times of the goals involved (i.e., T_a, T_b, T_c, T_d), which we assume to be non-zero. T_{fj1} and T_{fj2} are:

$$T_{fj1} = \max(T_a + T_b,\ T_c) + T_d \tag{1}$$

[3] The parallelization `p :- a(X, Z), b(X) & c(Y), d(Y, Z)` has been left out of Figure 2. It would not add anything to the discussion as it would not change the comparison we make in Section 2.2.

$$T_{fj2} = \max(T_a,\ T_c) + \max(T_b,\ T_d) \tag{2}$$

Comparing the quality of the annotations in Figure 2(a) and Figure 2(b) boils down to finding out whether it is possible to show that $T_{fj1} < T_{fj2}$ or the other way around. It turns out that they are non-comparable. In fact:

- $T_{fj1} < T_{fj2}$ holds if, for example, $T_a + T_b < T_c$, $T_d < T_b$, and then $T_{fj2} = T_b + T_c$, $T_{fj1} = T_d + T_c$, and
- $T_{fj2} < T_{fj1}$ holds if, for example, $T_c \leq T_a$, $T_d \leq T_b$, and then $T_{fj1} = T_a + T_b + T_d$, $T_{fj2} = T_a + T_b$.

Several annotation algorithms have been proposed so far [22, 4] which use the `&/2` operator as the basic construction to express parallelism between goals. These annotators produce clauses that are parallelized differently, such as those in Figure 2. It is in principle possible to statically decide (or, at least, approximate) whether some annotation is better than some other, for example by using the number of goals annotated for parallelism in a clause or, more interestingly, by using information regarding the expected runtime of goals (see, e.g., [21, 19] and its references). However, finding an optimal solution is a computationally expensive combinatorial problem [22] and, in practice, annotators use heuristics which may be more or less appropriate in concrete cases.

2.2 Parallelization with Finer Goal-Level Operators

It has been observed [4, 5] that more basic constructions can be used to represent and-parallelism by using two operators, `&>/2` and `<&/1`, defined as follows:

Definition 1. `G &> H` *schedules goal* `G` *for parallel execution and continues executing the code after* `G &>H`. `H` *is a* handler *which contains (or* points to*) the state of goal* `G`.

Definition 2. `H <&` *waits for the goal associated with* `H` *to finish. After that point any bindings made by* `G` *are available to the executing thread.*

With the previous definitions, the `&/2` operator can be written as `A & B :- A &> H, call(B), H <&.` This indicates that any parallelization performed using `&/2` can be made using `&>/2` and `<&/1` without loss of parallelism. We will term these operators *dep-operators* henceforth.

Two motivations justify the use of these operators instead of `&/2`. Firstly, their implementation is (in our experience) actually easier to devise and maintain than the monolithic `&/2` [8], and, secondly, the dep-operators allow more freedom to the annotator (and to the programmer, if parallel code is written by hand) to

```
p(X, Y, Z) :-
        c(Y) &> Hc,
        a(X, Z),
        b(X) &> Hb,
        Hc <&,
        d(Y, Z),
        Hb <&.
```

Fig. 3. dep-operator-annotated clause

express data dependencies and, therefore, to extract more potential parallelism. We will now illustrate this last point (the former is out of our current scope).

Figure 3 shows an annotation of our running example using dep-operators. Note that this code allows executing in parallel `a/2` with `c/1`, `b/2` with `c/1`, and `b/1` with `d/2`. The execution time of `p/3`, based on that of the individual goals, is:[4]

$$T_{dep} = \max(T_a + T_b,\ T_d + \max(T_a,\ T_c)) \tag{3}$$

If we compare expression (3) with expressions (1) and (2), it turns out that:

- It is possible that $T_{dep} < T_{fj1}$, $T_{dep} < T_{fj2}$, $T_{dep} = T_{fj1}$, and $T_{dep} = T_{fj2}$ (possibly with different lengths for every goal in each case [7]).
- It is **not** possible that $T_{dep} > T_{fj1}$ or that $T_{dep} > T_{fj2}$.

This means that the annotation in Figure 3 cannot be worse than those of Figure 2, and can perform better in some cases. It is, therefore, a better option than any of the others.

In addition to these basic operators, other specialized versions can be defined and implemented in order to increase performance by adapting better to some particular cases. In particular, it appears interesting to introduce variants for the very relevant and frequent case of deterministic goals. For this purpose we propose two new operators: `&!>/2` and `<&!/1`. These specialized versions do not perform backtracking and do not prepare the execution data structures to cope with that possibility, which has previously been shown to result in a significant efficiency increase in the underlying machinery [23].

3 The UOUDG and UUDG Algorithms

In this section we will present two concrete algorithms which generate code annotated for unrestricted independent and-parallelism (as in Figure 3) starting from sequential code. The proposed algorithms process one clause at a time and work on a directed acyclic dependency graph (V, E) where nodes are associated with body goals in the clause. We require that literals which are lexically identical give rise to different nodes, by, e.g., attaching a unique identifier to them. This is necessary in order not to lose information when building sets of nodes.

We assume a preprocessing stage which collapses sequences of mutually dependent goals with a single incoming (resp., outgoing) dependency. For example, in `p:- a(X), b(X), c(X), d(Y), e(Y), f(X, Y)` the sets {`a/1, b/1, c/1`} and {`d/1,e/1`} are sequences in the clause, but they have a single outgoing dependency on `f/2`. The preprocessing stage groups these sequences and assigns them to a single node in the dependency graph. Every one of these sequences can, for efficiency reasons, be folded into a unique predicate in order to avoid meta-interpretation of sequential conjunctions.

The idea behind these algorithms is to publish goals for parallel execution as soon as possible and to delay issuing joins as much as possible —but always

[4] See [7] for a deduction.

Algorithm: `UOUDG`(G, Pub)

Input : **(1)** A directed acyclic graph $G = (V, E)$.
(2) A set of already forked goals.

Output: A clause parallelized in *unrestricted and* fashion in which the order of the solutions in the original clause is preserved.

begin
  if $V = \emptyset$ **then return** (`true`)
  else
    $Indep \leftarrow \{v \mid v \in V,\ \texttt{incoming}(v, E) = \emptyset\}$;
    $Dep \leftarrow \{(v, I_v) \mid v \in V,\ I_v = \texttt{incoming}(v, E),\ I_v \neq \emptyset,\ I_v \subseteq Indep\}$;
    if $Dep = \emptyset$ **then**
      $(pvt, Join) \leftarrow (u, V)$ s.t. $\forall(w \in (V \setminus \{u\}))\ .\ w \prec u$;
    else
      $(pvt, Join) \leftarrow$
        (u, S) s.t. $(u, S) \in Dep \wedge \forall((w, D) \in (Dep \setminus \{(u, S)\}))\ .\ u \prec w$;
    end
    $Seq \leftarrow \{v \mid v \in\ (Indep \setminus Pub),\ v \rightarrow pvt \in E,\ v = pred(pvt)\}$;
    $Fork \leftarrow \{v \mid v \in\ (Indep \setminus Pub),\ v \prec pvt\} \setminus Seq$;
    $Join \leftarrow Join \setminus Seq$;
    $Pub \leftarrow Pub \cup Fork \cup Seq$;
    $G \leftarrow G - (Join \cup Seq)$;
    return (`gen_body`($Fork$, Seq, $Join$, $\emptyset$), `UOUDG`(G, Pub));
  end
end

Algorithm 1: UOUDG Annotation Algorithm.

respecting the dependencies in the graph (as in Figure 1). Intuitively, this should maximize the number of goals available for parallel execution. In the following, both algorithms use an auxiliary definition to denote the set of nodes which are connected to some node v: $\texttt{incoming}(\texttt{v}, \texttt{E}) = \{u \mid (u \rightarrow v) \in E\}$.

Note that, as mentioned in Section 2.1, we will consider in this paper only unconditional parallelism. However, the algorithms that we describe can be adapted to deal with conditional parallelism without too much effort.

3.1 Order-Preserving Annotation: the UOUDG Algorithm

Algorithm 1 parallelizes a clause while preserving the order of the solutions by respecting the relative order of literals in the original clause. In order to keep track of that order, we assume that there is a relation $\prec$ on the literals L_i of the body of every clause H :- $L_1, L_2, \ldots, L_{k-1}, L_k$ such that $L_i \prec L_j$ iff $i < j$. Additionally, we assume that there is a partial function *pred* defined as $pred(L_{i+1}) = L_i$, i.e., the literal at the left of some other literal in a clause. We assume $\prec$ and *pred* are suitably extended to the nodes of the dependency graph.[5]

[5] Note, also, that the graph edges must respect the $\prec$ relation: $(u \rightarrow v) \in E \Rightarrow u \prec v$. The graph would have been incorrectly generated otherwise.

At every recursion step, new nodes (i.e., literals) in the graph are selected to be published, joined, and executed sequentially. Subsequent iterations proceed with a simplified graph in which the literals which have been joined and executed sequentially, together with their outgoing edges, have been removed. The set of goals which have already been published is kept in a separate argument to schedule goals for parallel execution only once.

Two sets are key in each iteration: *Indep*, which contains the *sources* (i.e., all vertices without incoming edges in the current graph, which can therefore be published), and *Dep*, which contains tuples (v, I_v) where, for each non-source vertex v which can be reached from source vertices only, I_v is the set of source vertices ($I_v \subseteq Indep$) on which v depends. I.e., I_v is the set of vertices to be joined before v can start.

Also, *pvt* is the *pivot* vertex which will be used to decide which nodes are to be joined, taking into account that we do not want to change the order of solutions. If there are no *Dep* nodes, then all the remaining literals are already independent and we can join up to the rightmost literal in the clause. Otherwise, we select the leftmost node among those which have dependencies which can be fulfilled in one step. These dependencies are readily available in *Dep*. Note that as we select the leftmost node among those which can be joined, we are delaying as much as possible joining nodes —or, alternatively, we are performing in every step only the joins which are needed to continue one more step. This is aimed at maximizing the number of parallel goals being executed at any moment.

It is possible for a literal to be scheduled to be forked and then immediately joined. In order to detect these situations, which in practice would cause unnecessary overhead, we select (in *Seq*) the literal (only one) to which this applies, and it is not taken into account for the set of $Fork$ed literals and removed from the set of the $Join$ed literals.

The algorithm then continues outputting a parallelized expression (returned by `gen_body`, Algorithm 3) composed with the parallelization of a simplified graph, generated by a recursive call. Algorithm 3 is able to use determinism information to reorder goals. Since Algorithm 1 preserves the order of solutions, we do not use this capability at the moment. Therefore an empty set is passed as determinism data and we define the function $det(Lit, DetInfo)$ (used by Algorithm 3) to return *false* if $DetInfo = \emptyset$, thus safely assuming non-determinism.

Termination can be proved based on the following observation: G is a finite graph and it is simplified in each iteration provided $Join$ or Seq are non-empty. But $Join$ is always non-empty because it is either V (which is non-empty) when $Dep = \emptyset$ or else it is the second component of a tuple in *Dep* when $Dep \neq \emptyset$, and this component is by definition non-empty. Note that we are not using acyclicity to prove termination. However, all input graphs will be acyclic by definition.

3.2 Non Order-Preserving Annotation: the UUDG Algorithm

Algorithm 2 follows the same idea underlying Algorithm 1: publish early and join late. However, it has more freedom to publish goals, since the order of solutions

Algorithm: `UUDG`(G, Pub, I_D)

Input : **(1)** A directed acyclic graph $G = (V, E)$. **(2)** A set of goals already forked. **(3)** Determinacy information.

Output: An unrestricted parallelized clause in which the order of the solutions in the original clause need not be preserved.

begin
  if $V = \emptyset$ **then return** (`true`);
  else
    $Indep \leftarrow \{v \mid v \in V,\ \texttt{incoming}(v, E) = \emptyset\}$;
    $Dep \leftarrow \{I_v \mid v \in V,\ I_v = \texttt{incoming}(v, E),\ I_v \neq \emptyset,\ I_v \subseteq Indep\}$;
    if $Dep = \emptyset$ **then**
      $Join \leftarrow V$;
    else
      $SS \leftarrow \{I_v \mid I_v \in Dep,\ |I_v| = \texttt{min\_card}(Dep)\}$;
      $Join = \{s\}$ s.t. $s \in SS$; `/* s any element from SS */`
    end
    if $(Join \cap (Indep \setminus Pub)) = \emptyset$ **then**
      $Seq = \emptyset$;
    else
      $Seq = \{v\}$ s.t. $v \in (Join \cap (Indep \setminus Pub))$; `/* v any element */`
    end
    $Fork \leftarrow Indep \setminus (Pub \cup Seq)$;
    $Join \leftarrow Join \setminus Seq$;
    $Pub \leftarrow Pub \cup Fork \cup Seq$;
    $G \leftarrow G - (Join \cup Seq)$;
    return (`gen_body_det`($Fork$, Seq, $Join$, I_D), `UUDG`(G, Pub, I_D));
  end
end

Algorithm 2: UUDG Annotation Algorithm.

does not need to be preserved. This is implemented by selecting, among the sets of goals which can be joined at every moment, the one with the lowest cardinality —i.e., we join as few goals as possible, thus postponing the rest of the joins as much as possible, in order to exploit more parallelism. This is taken care of by `min_card`(`S`) = $\min(\{|s| \mid s \in S\})$, which returns the size of the smallest set in S.

Note that a random selection from a set is done at two points. Data regarding, e.g., the relative run time of goals would allow us to take a more informed decision and therefore precompute a perhaps better scheduling. Since we are not using this information here, we just pick any available goal to join / execute sequentially.

Algorithm 2 again uses Algorithm 3 to output a parallelized clause. In this case Algorithm 3 makes use of determinism information as follows:

– Since we already have the possibility of switching goals around, we try to minimize relaunching goals which are likely to be executed in parallel by forking deterministic goals first.

Algorithm: `gen_body`($Fork$, Seq, $Join$, I_D)

Input : **(1)** A set of vertices to be forked. **(2)** A set of vertices to be sequentialized. **(3)** A set of vertices to be joined. **(4)** Determinacy information.

Output: A parallelized sequence of literals Exp.

begin

$Exp \leftarrow$ (`true`);
$ForkDet = \{g \mid g \in Fork, det(g, I_D)\}$;
$ForkNonDet = \{g \mid g \in Fork, \neg det(g, I_D)\}$;
$JoinDet = \{g \mid g \in Join, det(g, I_D)\}$;
$JoinNonDet = \{g \mid g \in Join, \neg det(g, I_D)\}$;
forall $v_i \in ForkDet$ **do** $Exp \leftarrow (Exp,\ v_i$ `&!>` $H_{v_i})$;
forall $v_i \in ForkNonDet$ **do** $Exp \leftarrow (Exp,\ v_i$ `&>` $H_{v_i})$;
if $Seq = \{v\}$ **then** $Exp \leftarrow (Exp,\ v)$;
forall $v_i \in JoinDet$ **do** $Exp \leftarrow (Exp,\ H_{v_i}$ `<&!`$)$;
forall $v_i \in JoinNonDet$ **do** $Exp \leftarrow (Exp,\ H_{v_i}$ `<&`$)$;
return Exp;

end

Algorithm 3: Determinism-aware generation of a parallel body.

G=(V,E)	I	D	J	S	F	J\S	P	Parallel Code
$(\{a,b,c,d\},\{(a,b),(a,d),(c,d)\})$							$\emptyset$	`p(X,Y,Z) :-`
$(\{a,b,c,d\},\{(a,b),(a,d),(c,d)\})$	$\{a,c\}$	$\{b,d\}$	$\{a\}$	$\{a\}$	$\{c\}$	$\emptyset$	$\{a,c\}$	`c(Y) &> Hc, a(X,Z),`
$(\{b,c,d\},\{(c,d)\})$	$\{b,c\}$	$\{d\}$	$\{c\}$	$\emptyset$	$\{b\}$	$\{c\}$	$\{a,b,c\}$	`b(X) &> Hb, Hc <&,`
$(\{b,d\},\emptyset)$	$\{b,d\}$	$\emptyset$	$\{b,d\}$	$\{d\}$	$\emptyset$	$\{b\}$	$\{a,b,c,d\}$	`d(Y,Z), Hb <&.`
$(\emptyset,\emptyset)$								

Table 1. Iterations of the UUDG algorithm when parallelizing `p/3`.

– Additionally, when a goal is known to have exactly one solution, we can use specialized versions of the dep-operators [8] which do not need to perform bookkeeping for backtracking (always complex in parallel implementations), and are thus more efficient.

This program information can often be automatically inferred by the abstract interpretation-based determinism analyzer in CiaoPP [18], and is provided as input to the proposed annotators. Alternatively, this information can be stated by the programmer via assertions [13].

Example 1 (UUDG Annotation). In order to illustrate how the UUDG algorithm works, Table 1 shows the results obtained at each of the iterations of the parallelization process for the `p/3` predicate introduced in Section 2.1 and whose dependency graph is shown in Figure 1. Columns are labeled with the first character of each of the variables they represent. Note that in the first algorithm step, both a and c are candidates for parallel execution (they are in $Indep$). However, as a has to be joined too (it is necessary to continue executing either b or d) it is selected to be sequentially executed.

AIAKL	An abstract interpreter for the AKL language.
FFT	An implementation of the Fast Fourier transform.
FibFun	A version of **Fib** written in functional notation.
Hamming	A program to compute the first N Hamming numbers.
Hanoi	A program to compute movements to solve the well-known puzzle.
Takeuchi	Computes the Takeuchi function.
WMS2	A scheduler assigning a number of workers to a series of jobs.

Table 2. Benchmark programs

4 Performance Evaluation

Our annotation algorithms have been integrated in the Ciao/CiaoPP system [13]. Information gathered by the analyzers on variable sharing, groundness, and freeness is used to determine goal independence, using the libraries available in CiaoPP. Determinism is used in the annotators as described previously.

As execution platform we have used a high level implementation of the proposed parallelism primitives [8], which we have developed as an extension of the Ciao system. This implementation is an evolution and simplification of [12] which is based on raising the level of certain components to the level of the source language and keeping only some selected operations (related to thread handling, locking, etc.) at a lower level. This approach does not eliminate altogether modifications to the abstract machine, but it greatly simplifies them. It should be noted however that the dep-operators do not assume any particular architecture: while our current implementation and all the performance results were obtained on a multicore machine, the techniques presented can be also applied in distributed memory machines —and in fact, the first prototype implementation of the dep-operators [5, 4] was actually made on a distributed environment.

We have evaluated the impact of the different annotations on the execution time by running a series of benchmarks (briefly described in Table 2) in parallel. Table 3 shows the speedups obtained *with respect to the sequential execution*, i.e., they are *actual* speedups,[6] when using from 1 to 8 threads. The machine we used is a Sun UltraSparc T2000 (a *Niagara*) with 8 4-thread cores.[7] The *fork-join* annotators we chose to compare with are MEL [22] (which preserves goal order and tries to maximize the length of the parallel expressions) and UDG [4] (which can reorder goals). MEL can add runtime checks to decide dynamically whether to execute or not in parallel. In order to make the annotation unconditional (as the rest of the annotators we are dealing with), we simply removed the conditional parallelism in the places where it was not being exploited. This is why it appears in Table 3 under the name *UMEL*.

All the benchmarks executed were parallelized automatically by CiaoPP, starting from their sequential code. Since UOUDG and UUDG can improve the results of fork-join annotators only when the code to parallelize has at least a cer-

[6] This is the reason why some speedups start below 1 for, e.g., one thread.

[7] We did not use more than 8 cores since in that case, and due to access to shared units, speedups are sublinear even for completely independent tasks.

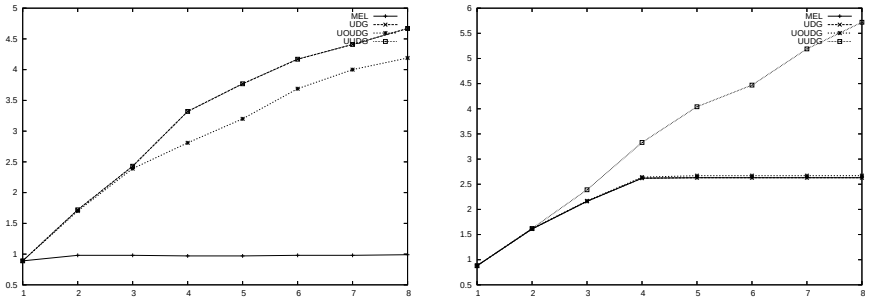
Benchmark	Annotator	Number of threads							
		1	2	3	4	5	6	7	8
AIAKL	UMEL	0.97	0.97	0.98	0.98	0.98	0.98	0.98	0.98
	UOUDG	0.97	1.55	1.48	1.49	1.49	1.49	1.49	1.49
	UDG	0.97	1.77	1.66	1.67	1.67	1.67	1.67	1.67
	UUDG	0.97	1.77	1.66	1.67	1.67	1.67	1.67	1.67
FFT	UMEL	0.98	1.76	2.14	2.71	2.82	2.99	3.08	3.37
	UOUDG	0.98	1.76	2.14	2.71	2.82	2.99	3.08	3.37
	UDG	0.98	1.76	2.14	2.71	2.82	2.99	3.08	3.37
	UUDG	0.98	1.82	2.31	3.01	3.12	3.26	3.39	3.63
FibFun	UMEL	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	UOUDG	0.99	1.95	2.89	3.84	4.78	5.71	6.63	7.57
	UDG	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	UUDG	0.99	1.95	2.89	3.84	4.78	5.71	6.63	7.57
Hamming	UMEL	0.93	1.13	1.52	1.52	1.52	1.52	1.52	1.52
	UOUDG	0.93	1.15	1.64	1.64	1.64	1.64	1.64	1.64
	UDG	0.93	1.13	1.52	1.52	1.52	1.52	1.52	1.52
	UUDG	0.93	1.15	1.64	1.64	1.64	1.64	1.64	1.64
Hanoi	UMEL	0.89	0.98	0.98	0.97	0.97	0.98	0.98	0.99
	UOUDG	0.89	1.70	2.39	2.81	3.20	3.69	4.00	4.19
	UDG	0.89	1.72	2.43	3.32	3.77	4.17	4.41	4.67
	UUDG	0.89	1.72	2.43	3.32	3.77	4.17	4.41	4.67
Takeuchi	UMEL	0.88	1.61	2.16	2.62	2.63	2.63	2.63	2.63
	UOUDG	0.88	1.62	2.17	2.64	2.67	2.67	2.67	2.67
	UDG	0.88	1.61	2.16	2.62	2.63	2.63	2.63	2.63
	UUDG	0.88	1.62	2.39	3.33	4.04	4.47	5.19	5.72
WMS2	UMEL	0.85	0.81	0.81	0.81	0.81	0.81	0.81	0.81
	UOUDG	0.99	1.09	1.09	1.09	1.09	1.09	1.09	1.09
	UDG	0.99	1.01	1.01	1.01	1.01	1.01	1.01	1.01
	UUDG	0.99	1.10	1.10	1.10	1.10	1.10	1.10	1.10

Table 3. Speedups for several benchmarks and annotators.

tain level of complexity, not all benchmarks with (independent) parallelism can benefit from using the dep-operators. Additionally, comparing speedups obtained with programs parallelized using order-preserving and non-order-preserving annotators is not completely meaningful.

Note that in this paper we are not focusing on the speedups themselves. Although of utmost practical interest, raw speed is very connected with the implementation of the underlying parallel abstract machine, and improvements on it can be expected to uniformly affect all parallelized programs. Rather, our main focus of attention is in the *comparison* among the speedups obtained using different annotators.

A first examination of the experimental results in Table 3 allows inferring that in no case is UUDG worse than any other annotator, and in no case is UOUDG worse than (U)MEL. They should therefore be the *annotators of choice* if available. Besides, there are cases where UOUDG is better than UDG, and the



(a) Hanoi (b) Takeuchi

Fig. 4. Speedups with different annotations for Hanoi and Takeuchi.

other way around, which is in accordance with the non-comparable nature of these two algorithms.

Among the cases in which a better speedup is obtained by some of the U(O)UDG annotators, improvements range between "no improvement" (because no benefit is obtained for some particular cases and combinations of annotators) to an increase of 752% in speedup, with several other stages in between. Also, it is worth pointing out that the speedup does not stabilize in any benchmark (at least in a sizable amount) as the number of threads increases; moreover, in some cases the difference in speedup between the restricted and the unrestricted versions grows substantially with the number of threads. This can (clearly) be seen in, e.g., Figure 4(b).

Finally, we would like to comment specially on three benchmarks. **FibFun** is the result of parallelizing a definition of the Fibonacci numbers written using the functional notation capabilities of Ciao [6]. Because of the order in which code is generated in the (automatic) translation into Prolog, the result is only parallelizable by UOUDG and UUDG, hence the speedup obtained in this case. The case of **Hanoi** is also interesting, as it is the first example in [22]: in the arena of order-preserving parallelizers, UOUDG can extract more parallelism than MEL for this benchmark. Lastly, the **Takeuchi** benchmark has a relatively small loop which only allows parallelizing with a simple `&/2`. However, by unrolling one iteration the resulting body has dependencies which are complex enough to take advantage of the increased flexibility of the dep-operator annotators.

5 Conclusions

We have proposed two annotation algorithms which perform a source-to-source transformation of a logic program into an unrestricted independent and-parallel version of itself. Both algorithms rely on the use of more basic high-level primitives than the fork-join operator, and differ on whether the order of the solutions in the original program must be preserved or not. We have implemented the proposed algorithms in the CiaoPP system, which infers automatically groundness,

sharing, and determinacy information, used to simplify the initial dependency graph. The results of the experiments performed show that, although the parallelization provided by the new annotation algorithms is the same in quite a few of the traditional parallel benchmarks, it is never worse and in some cases it is significantly better. This supports the observations made based on the expected performance of the annotations. We have also noticed that the benefits are larger for programs with high numbers of goals in their clauses, since their more complex graphs make the ability to exploit non-restricted parallelism more relevant.

Acknowledgments: This work was funded in part by Ministry of Education and Science (MEC) project TIN2005-09207-C03 *MERIT-COMVERS*, by Ministry of Industry (MIN) PROFIT project FIT-350400-2006-44 *GGCC*, by Madrid Regional Government (CM) project S-0505/TIC/0407 *PROMESAS*, and by IST program of the European Commission FP6 FET project IST-15905 *MOBIUS*. Manuel Hermenegildo and Amadeo Casas were also funded in part by the Prince of Asturias Chair in Information Science and Technology at UNM.

References

1. K. A. M. Ali and R. Karlsson. The Muse Or-Parallel Prolog Model and its Performance. In *1990 North American Conference on Logic Programming*, pages 757–776. MIT Press, October 1990.
2. F. Bueno, M. García de la Banda, and M. Hermenegildo. Effectiveness of Abstract Interpretation in Automatic Parallelization: A Case Study in Logic Programming. *ACM Transactions on Programming Languages and Systems*, 21(2):189–238, March 1999.
3. F. Bueno, M. García de la Banda, and M. Hermenegildo. Effectiveness of Abstract Interpretation in Automatic Parallelization: A Case Study in Logic Programming. *ACM TOPLAS*, 21(2):189–238, March 1999.
4. D. Cabeza. *An Extensible, Global Analysis Friendly Logic Programming System.* PhD thesis, Universidad Politécnica de Madrid (UPM), Facultad Informatica UPM, 28660-Boadilla del Monte, Madrid-Spain, August 2004.
5. D. Cabeza and M. Hermenegildo. Implementing Distributed Concurrent Constraint Execution in the CIAO System. In *Proc. of the AGP'96 Joint conference on Declarative Programming*, pages 67–78, San Sebastian, Spain, July 1996. U. of the Basque Country. Available from `http://www.cliplab.org/`.
6. A. Casas, D. Cabeza, and M. Hermenegildo. A Syntactic Approach to Combining Functional Notation, Lazy Evaluation and Higher-Order in LP Systems. In *FLOPS'06*, Fuji Susono (Japan), April 2006.
7. A. Casas, M. Carro, and M. Hermenegildo. Annotation Algorithms for Unrestricted Independent And-Parallelism in Logic Programs. Technical Report CLIP4/2007.0, Technical University of Madrid (UPM), School of Computer Science, UPM, June 2007.
8. A. Casas, M. Carro, and M. Hermenegildo. Towards High-Level Execution Primitives for And-Parallelism: Preliminary Results. In *Colloquium on Implementation of Constraint and LOgic Programming Systems (ICLP associated workshop)*. ACM Press, September 2007.

9. G. Gupta, M. Hermenegildo, E. Pontelli, and V. Santos-Costa. ACE: And/Or-parallel Copying-based Execution of Logic Programs. In *International Conference on Logic Programming*, pages 93–110. MIT Press, June 1994.
10. G. Gupta, E. Pontelli, K. Ali, M. Carlsson, and M. Hermenegildo. Parallel Execution of Prolog Programs: a Survey. *ACM TOPLAS*, 23(4):472–602, July 2001.
11. M. Hermenegildo. Parallelizing Irregular and Pointer-Based Computations Automatically: Perspectives from Logic and Constraint Programming. *Parallel Computing*, 26(13–14):1685–1708, December 2000.
12. M. Hermenegildo and K. Greene. The &-Prolog System: Exploiting Independent And-Parallelism. *New Generation Computing*, 9(3,4):233–257, 1991.
13. M. Hermenegildo, G. Puebla, F. Bueno, and P. López García. Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). *Science of Computer Programming*, 58(1–2):115–140, October 2005.
14. M. Hermenegildo and F. Rossi. Strict and Non-Strict Independent And-Parallelism in Logic Programs: Correctness, Efficiency, and Compile-Time Conditions. *Journal of Logic Programming*, 22(1):1–45, 1995.
15. M. Hermenegildo and R. Warren. Designing a High-Performance Parallel Logic Programming System. *Computer Architecture News, Special Issue on Parallel Symbolic Programming*, 15(1):43–53, March 1987.
16. Sverker Janson. *AKL. A Multiparadigm Programming Language*. PhD thesis, Uppsala University, 1994.
17. A.H. Karp and R.C. Babb. A Comparison of 12 Parallel Fortran Dialects. *IEEE Software*, September 1988.
18. P. López-García, F. Bueno, and M. Hermenegildo. Determinacy Analysis for Logic Programs Using Mode and Type Information. In *Proceedings of the 14th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'04)*, number 3573 in LNCS, pages 19–35. Springer-Verlag, August 2005.
19. P. López-García, M. Hermenegildo, and S. K. Debray. A Methodology for Granularity Based Control of Parallelism in Logic Programs. *Journal of Symbolic Computation, Special Issue on Parallel Symbolic Computation*, 21(4–6):715–734, 1996.
20. E. Lusk et al. The Aurora Or-Parallel Prolog System. *New Generation Computing*, 7(2,3), 1990.
21. E. Mera, P. López-García, G. Puebla, M. Carro, and M. Hermenegildo. Combining Static Analysis and Profiling for Estimating Execution Times. In *Ninth International Symposium on Practical Aspects of Declarative Languages*, number 4354 in LNCS, pages 140–154. Springer-Verlag, January 2007.
22. K. Muthukumar, F. Bueno, M. García de la Banda, and M. Hermenegildo. Automatic Compile-time Parallelization of Logic Programs for Restricted, Goal-level, Independent And-parallelism. *Journal of Logic Programming*, 38(2):165–218, February 1999.
23. E. Pontelli, G. Gupta, D. Tang, M. Carro, and M. Hermenegildo. Improving the Efficiency of Nondeterministic And–parallel Systems. *The Computer Languages Journal*, 22(2/3):115–142, July 1996.
24. V. Santos-Costa, D.H.D. Warren, and R. Yang. Andorra-I: A Parallel Prolog System that Transparently Exploits both And- and Or-parallelism. In *Proceedings of the 3rd. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 83–93. ACM, April 1991. SIGPLAN Notices vol 26(7), July 1991.
25. K. Shen. Overview of DASWAM: Exploitation of Dependent And-parallelism. *Journal of Logic Programming*, 29(1–3):245–293, November 1996.

A Flexible, (C)LP-based Approach to the Analysis of Object-Oriented Programs[3]

Mario Méndez-Lojo[1], Jorge Navas[1], and Manuel V. Hermenegildo[1,2]

[1] University of New Mexico (USA)
[2] Technical University of Madrid (Spain)

Abstract. Static analyses of object-oriented programs usually rely on intermediate representations that respect the original semantics while having a more uniform and basic syntax. Most of the work involving object-oriented languages and abstract interpretation usually omits the description of that language or just refers to the Control Flow Graph (CFG) it represents. However, this lack of formalization on one hand results in an absence of assurances regarding the correctness of the transformation and on the other it typically strongly couples the analysis to the source language. In this work we present a framework for analysis of object-oriented languages in which in a first phase we transform the input program into a representation based on Horn clauses. This facilitates on one hand proving the correctness of the transformation attending to a simple condition and on the other allows applying existing analyzers for (constraint) logic programming to automatically derive a safe approximation of the semantics of the original program. The approach is flexible in the sense that the first phase decouples the analyzer from most language-dependent features, and correct because the set of Horn clauses returned by the transformation phase safely approximates the standard semantics of the input program. The resulting analysis is also reasonably scalable due to the use of mature, modular (C)LP-based analyzers. This allows us to report good results for medium-sized programs.

1 Introduction

Analysis of object-oriented languages using abstract interpretation [9] is currently the subject of significant research (see, e.g., [21] and its references). The abstract interpretation approach brings an interesting and useful combination of characteristics: it is automatic and practical, producing useful results for a good number of applications, while at the same time being rigorous and semantics-based. The gap between programs and semantics is greater in the case of object-oriented languages than in, for example, declarative languages. For this reason,

[3] This work was supported in part by the Prince of Asturias Chair in Information Science and Technology at UNM, the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-15905 *MOBIUS* project, the Spanish Ministry of Education under the TIN-2005-09207 *MERIT* project, and the Madrid Regional Government under the S-0505/TIC/0407 *PROMESAS* program.

static analyses of object-oriented programs usually rely on intermediate languages that respect the original semantics while having a more uniform and basic syntax (e.g., block-based representations) and a more declarative semantics (e.g., static single assignment transformations). Some significant concrete examples which have been proposed of such intermediate representations for object-oriented programs are Jimple [32] for Java or BoogiePL [10] for .NET.

In this paper we propose the use of a Horn clause-based representation as an intermediate language. Our objective is twofold. On one hand we would like to take advantage of existing analyzers for (constraint) logic programs. On the other, we want to be able to offer assurances that the output of the process of transformation into the intermediate representation safely approximates the standard semantics of the input program. Performing the analysis using logic programming tools offers a number of advantages, such as the relative maturity and sophistication of the solutions available, like abstract interpreters (which offer parametric, efficient, and modular fixpoint algorithms) and verifiers (see, e.g., [15, 12] and their references). A second strength of our transformational approach is that the framework can be easily adapted to the analysis of other languages without having to redefine the fixpoint algorithm [24]. In fact, using the intermediate representation that we propose, from the analyzer point of view an object-oriented program is indistinguishable from, e.g., a Prolog one (although of course different abstract domains and definitions of pseudo-builtins are used). This brings in the additional advantage of being able to analyze multiple languages within the same framework.

We start by describing our methodology (Section 2) and our approach to ensuring correctness using some fundamental parts of the transformation of Java programs into our representation as examples (Section 3). Section 4 shows how analysis of specific aspects of Java can be optimized using metainformation. We then illustrate the application of our approach to other languages, such as C# (Section 5). We also report on an implementation of the ideas presented in this paper using the abstract interpretation-based CiaoPP framework [15]. It can be configured for many different analyses by simply plugging the corresponding abstract domain. The examples try to detect null pointer dereferences (nullity analysis) and eliminate dynamic dispatch (class analysis) in Java programs. The experiments in Section 6 show that the technique scales well in non trivial scenarios, and results in smaller analysis times than similar previous work. Related abstract interpretation-based frameworks, and how they differ from ours, are discussed in Section 7, and Section 8 presents our conclusions.

2 Methodology: the transformational approach

Our framework is composed of a front-end preprocessor and a back-end analyzer, as shown in Figure 1. The preprocessor transforms an input in Java source format into a set of Horn clauses that represent a safe approximation of its standard semantics (Sect 3). Sometimes the source code is not available, so we also accept Java bytecode as a valid input format. In this case the (de)compilation from

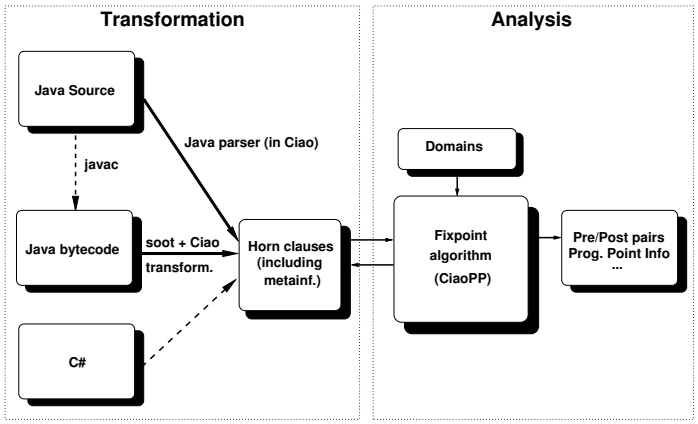


Fig. 1. Transformation and analysis pipeline.

bytecode to Horn clauses is based on a postprocessing of the Jimple representation returned by the Soot [32] tool. It is beyond the scope of this paper to provide a detailed description of this particular transformation; the reader is referred to [24] (which presents our transformation and a specific fixpoint algorithm for analysis) for details. In both cases the same subset of the language is covered by the framework. Our ultimate objective is to support the full Java language but the current implementation has some limitations: it does not support dynamic loading of classes, threads, or runtime exceptions. Also, analysis of the JDK libraries is done under worst-case assumptions.

The resulting Horn-clause intermediate representation is then analyzed using the CiaoPP framework [15] and benefits from its advanced features: efficient computation of fixpoints using memoization, context-sensitivity, modularity, etc. The programmer needs only to implement (in Ciao [6], or in plain Prolog) the particular abstract domain of interest, which includes also defining the abstract meaning of a set of "built-in" predicates that represent the language-dependent semantics of the basic operations of the source language. On the other hand, our approach does liberate the designer of an analysis from the burden of coding a fast, reliable, and efficient abstract interpretation platform. Analysis results are given in the standard form (p, σ), where p uniquely identifies a program point and σ is an abstract state which safely approximates all the possible states at that program point during runtime. Metainformation computed during the transformation process allows relating those line numbers with the ones of the original bytecode or source program, making it possible to reflect back the results on the original program text (as JML-like assertions [18]), pinpoint errors in the original program, or implement compiler optimizations.

Other languages can be incorporated into the framework (i.e., analyzed) by providing a correct transformation for them. For example, support for other object-oriented languages like C#, that share many syntactic and semantics features with Java, is easily achievable as illustrated in Section 5. In addition,

programs written in Ciao, which CiaoPP deals with natively, are obviously also accepted by the system as input.

3 Overview of the semantic basis and correctness of the transformation phase

Our Horn clause representation of a Java program is basically an unfolded, three-address version of the source where the operational semantics of some instructions is made explicit. The transformed code is denoted by the c subindex: for example, the result of transforming a virtual invocation $v.\mathtt{m}(v_1,\ldots,v_n)$ is $v_c.\mathtt{m}_c(v_{1_c},\ldots,v_{n_c}) = v.\mathtt{m}_c(v_1,\ldots,v_n)$, since variable expressions are not transformed ($v_c = v$).

Correctness of the transformation requires that the original program *prog* be emulated by $prog_c$ thus $\mathcal{C}[\![prog]\!] = \mathcal{C}[\![prog_c]\!]$, where the semantics operator $\mathcal{C}[\![]\!] : com \mapsto (\mathcal{D} \mapsto \mathcal{D})$ takes as input a command *com* and a concrete state, and returns the output state. The operator has been defined in [14] and (from a denotational point of view) in [2, 29]. Correctness of preprocessing and analysis requires that if the Horn clause program is safely approximated (using a given abstract domain) by the analysis, so is the original: $\mathcal{C}^*[\![prog]\!] = \mathcal{C}^*[\![prog_c]\!]$. The $\mathcal{C}^*[\![]\!] : com \mapsto (\mathcal{D}^* \mapsto \mathcal{D}^*)$ operator is the abstract counterpart of $\mathcal{C}[\![]\!]$.

We will take a slightly different approach by interpreting Java semantics as a particular case of SLD [17] resolution, in which the *computation* rule in use is left-to-right (commands are executed in the order they appear in the program) and the *search* rule used to determine the target method in an invocation in principle does not really matter, since execution of the Java program is deterministic and therefore for any literal there is exactly one clause that unifies with it at run time. Therefore, if $\mathcal{S}[\![]\!] : com \mapsto (\mathcal{D} \mapsto \mathcal{P}(\mathcal{D}))$ is the SLD semantics operator, the condition $\mathcal{S}[\![prog]\!] = \{\mathcal{C}[\![prog]\!]\}$ ensures $\mathcal{S}^*[\![prog]\!] = \mathcal{C}^*[\![prog]\!]$. Again, $\mathcal{S}^*[\![]\!] : com \mapsto (\mathcal{D}^* \mapsto \mathcal{D}^*)$ is the (collecting) abstract version of $\mathcal{S}[\![]\!]$.

This formalization is useful since it helps understanding the Java source as a set of Horn clauses (methods) composed by zero or more goals, the commands. It is also helpful because our transformation introduces new clauses such that now more than one clause might unify with a given literal. This is equivalent to saying that the execution of the transformed program on some input state might result in multiple output states, of which one is the unique state that the original program would return: $\mathcal{S}[\![prog]\!] \subseteq \mathcal{S}[\![prog_c]\!]$. An interesting property of that transformed program is that its abstract semantics $\mathcal{S}^*[\![prog_c]\!]$ still correctly approximates that of the original, i.e., $\mathcal{S}^*[\![prog]\!] \leq \mathcal{S}^*[\![prog_c]\!]$. Therefore, all we have to prove in order to show that the results of the analysis are correct is that $\mathcal{S}[\![prog]\!] \subseteq \mathcal{S}[\![prog_c]\!]$ (or $\mathcal{C}[\![prog]\!] \in \mathcal{S}[\![prog_c]\!]$) holds. Space limitations prevent us from discussing the whole transformation algorithm and providing proofs. Instead, we describe and provide a proof sketch for the case of the virtual invocation expression, which is one of the most complex operations supported.

```
staticCallSemantics(k$m(v, v_1, ..., v_n), σ)
    s = signature(call)
    body = getBody(k$m, s)
    return(bodySemantics(body, σ))

virtualCallSemantics(k?m(v, v_1, ..., v_n), σ)
    s = signature(call)
    c = lookup(runtime_class(v), s)
    return staticCallSemantics(c$m(v, v_1, ..., v_n), σ)

lookup(k, s)
    a = k
    do
      if declares(a, s)
        return(a)
      a = ancestor(a)
    while (true)
```

```
compileStaticCall(k$m(v, v_1, ..., v_n), prog_c)
    return k$m(v, v_1, ..., v_n)

compileVirtualCall(k?m(v, v_1, ..., v_n), prog_c)
    s = signature(call)
    C = resolve(k, s)
    forall c ∈ C add to prog_c the clause
        k$dyn*m(v, v_1, ..., v_n) : −
            c$m(v, v_1, ..., v_n)
    return k$dyn*m(v, v_1, ..., v_n)

resolve(k, s)
    result = ∅
    Sub = subclasses(k) ∪ {k}
    forall sub ∈ Sub
        sk = lookup(sub, s)
        result = result ∪ sk
    return result
```

Fig. 2. Standard semantics (left) and transformation (right) of method calls.

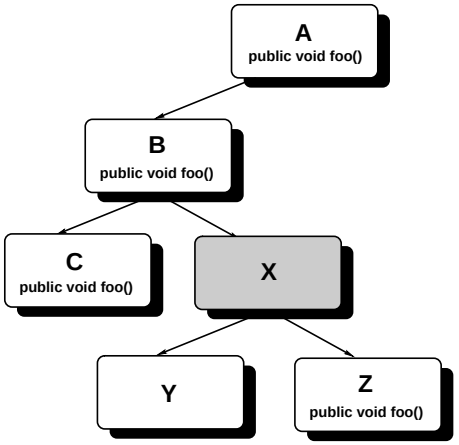
3.1 Correctness of a virtual invocation

The description of the standard semantics in this section is a slightly simplified version of the more formal specification described in [29]. We distinguish between two different kinds of invocations: virtual and static. Assume that calls of the first type have been rewritten as $\texttt{k?m}(v, v_1, \ldots, v_n)$ and the static ones as $\texttt{k\$m}(v, v_1, \ldots, v_n)$, where k is the declared type of v. Note that we rewrote the call syntax so the invoked object v is now the first actual parameter. The main difference between the two is that while in virtual invocations we need to figure out the particular class of v through a *lookup* in the class hierarchy, that operation is unnecessary in static calls since there is only one possible receiver.

In the left column of Figure 2 we present the pseudocode for the semantics of a static call (here denoted by `staticCallSemantics`) and a virtual call (here denoted by `virtualCallSemantics`). The particular signature of the invocation has to be calculated in order to distinguish which implementation to choose, since in Java (as in the Horn clauses) there can be many methods with the same name and arity, but here they will differ in the type of at least one of the formal parameters. Also, we will assume that there exists a function `runtime_class` that returns the runtime type of the object passed as parameter.

We refer to the tuple $(v, v_1, \ldots, v_n)$ as *pars*. The standard semantics of the call in the original program is $\mathcal{C}[\![k\texttt{?m}(pars)]\!]\sigma = \mathcal{C}[\![c\texttt{\$m}(pars)]\!]\sigma$, where c is the value returned by $\texttt{lookup}(\texttt{runtime\_class}(v), s)$. The SLD semantics of the transformed version is $\mathcal{S}[\![k\texttt{?m}_c(pars)]\!]\sigma$, which the transformation ensures to be $\mathcal{S}[\![k\texttt{\$dyn*m}(pars)]\!]\sigma = \bigcup_i \mathcal{S}[\![c_i\texttt{\$m}(pars)]\!]\sigma$, where $c_i \in \texttt{resolve}(k, s)$. The correctness condition is now reduced to proving that c is equal to some c_i. This is equivalent to showing that $\texttt{lookup}(\text{runtime\_class}(v), s) \in \texttt{resolve}(k, s)$, which can be further rewritten as $\texttt{lookup}(\texttt{runtime\_class}(v), s) \in \{\texttt{lookup}(sub, s) \mid sub \in \texttt{subclasses}(k) \cup \{k\}\}$. But the runtime type of v can only be k or a subclass of it in a type safe language as Java, and therefore the condition always holds.

Example 1. Assume a hierarchy of classes like in Figure 3. The root class `A` declares a method `foo` which is further redefined (overwritten) in subclasses `B`,




```
someMethod(){
  X v;
  ...
  v.foo();
}
```

(a) Fragment of the original program

```
SomeClass$someMethod():-
  ...
  X$dyn*foo(v).

X$dyn*foo(w):-
   B$foo(w)
X$dyn*foo(w):-
   Z$foo(w).
```

(b) Fragment of the transformed program

Fig. 3. Transformation of a virtual invocation.

`C`, and `Z`. If the original program in Figure 3a) contains a virtual invocation to `foo` in an instance declared as being of class `X`, our compiler automatically transforms it into a call to a new method with two new clauses (methods) that represent all the possible receiver implementations for the call. Because `X` is a direct subclass of `B`, it can never inherit the original `A` implementation but only the `B` one, represented by the first clause of `x$dyn$foo`. Alternatively, any object of type `Y` and `Z` is also of type `X` and therefore we include a call to the `Z` version of `foo` in the second clause. The `C` implementation is discarded because of type incompatibility.

The process described has many interesting properties. First, it is based on assuming SLD resolution semantics for the transformed Horn clause program. This allows reusing existing analyzers without having to redefine the abstract unification operator in order to deal with language-dependent features, as in the case of virtual invocation. We implemented our Java analyses on top of the CiaoPP Prolog analyzer [15] without modifying its code, even when specific abstract domains and "builtin" definitions for Java language constructs had to be provided. A second strength is that correctness of the transformation depends only on showing that $\mathcal{C}[\![comm]\!] \in \mathcal{S}[\![comm_c]\!]$ holds for every command (and expression) in the source language. Although not trivial, the proof can be slightly modified for similar languages to Java, so neither the compiler nor the abstract domains need to be completely rewritten. In the case of Ciao, the proof is trivial since $prog_c = prog$.

4 Other (meta-)information added by the transformation

The addition of meta-information during the transformation, although not strictly required, can help both efficiency and full independence from the source language. In some cases the fixpoint algorithm can be optimized if some characteristics related to the original source are known. In other cases the abstract

```
package examples;

public class Vector {

    Element first;

    public void add(int value){
        Element e = new Element();
        e.value = value;
        Vector v = new Vector();
        v.first = e;
        append(v);
    }
    public void append(Vector v){
        Element e = first;

        if (e == null)
            first = v.first;
        else{
             while (e.next != null)
                e = e.next;

             e.next = v.first;
          }
    }
}
```

```
class SubVector extends Vector{

    public void append(Vector v){
        //...
    }
}
```

class	ancestor
Vector	Object
SubVector	Vector
Element	Object

method	entry
Vector$init	y
Vector$add	y
Vector$dyn*append	y
Vector$append	y
Vector$append#1#2	n
Vector$append#3#4	n
SubVector$init	y
SubVector$append	y
Element$init	y

Fig. 4. Vector example: source code and corresponding metainformation.

domain can use certain information about the program not directly encoded in the Horn clauses. Both demands are solved via the addition of *metainformation* to the transformation. We illustrate this point with the example in Figure 4, which shows an alternative version of the JDK `Vector` class. The descendant `SubVector` contains an alternative version of the `append` method. The corresponding Horn clauses (represented as a Control Flow Graph) are shown in Figure 5. We omitted the constructor (`init`) clauses for simplicity.

Space reasons prevent us from listing a complete description of the metainformation; only hierarchy and method type tables are shown in Figure 4 (such tables are represented as sets of facts). In the case of the parent-child relations, the purpose is to provide the abstract domain code with access to the class tree, the more obvious application being class analysis [3]. The second table contains a classification for each method, which can be y (entry) or n (internal). It is used to optimize the performance of the fixpoint engine, avoiding *projection* and *extension* operations [5] (e.g., for blocks that share variable scope with the calling context, such as conditionals).

An *entry* method corresponds in the original program to the first clause [14] of the Java method of the same name and shares its signature, except for an extra parameter that represents the value returned. The other clauses present in the Java method are compiled into (components of) *internal* methods which share the same set of variables: all the formal parameters and local variables they reference. Examples of constructions converted into internal clauses are `if`, `while`, or `for` loops. In the example, we can see how the `if (e==null)...else` conditional in the `Vector` implementation of `append` is converted into two different clauses, one for each branch, which actually share the same name `Vector$append#1#2` (Figure 5). In this case, the internal method is composed of two clauses which

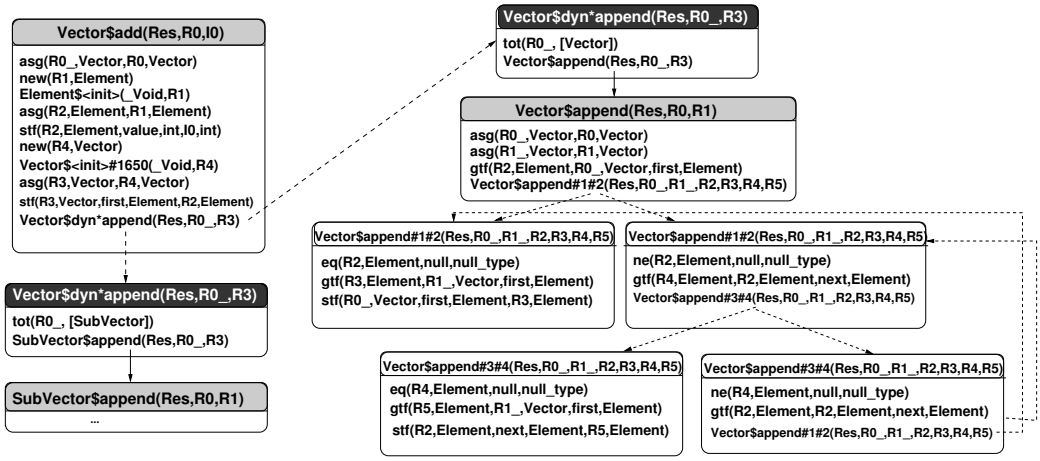


Fig. 5. Call Graph for the example in Figure 4.

are indistinguishable from the caller's point of view, thus causing invocations to the method to be non-deterministic (i.e., causing the execution of one clause or another). Entry clauses are marked in gray, internal ones in white; dotted arrows denote non-deterministic flows while the continuous ones symbolize deterministic calls.

Another flow transformation (*extra* clauses) tries to expose the internal structure of some complex Java features, which sometimes encode sophisticated operations. That is the case of the virtual invocations studied in Section 3. Coming back to the example in Figure 4, note that the call to `append` within `add` is polymorphic: it might execute the implementation in `Vector` or the one in `SubVector`. We make this semantics explicit by inspecting the application hierarchy and replacing the virtual invocation with a set of resolved calls, one for each possible implementation. The method acting as a "hub" is called an *extra* clause; in the example we have two, `Vector$dyn*append`, marked in black. They behave in a very similar way to the conditional discussed previously, since the program flow might go through two alternative paths (clauses), one for each implementation of `append`. Each branch contains a guard (`tot`, see the first statement in each of the `Vector$dyn*append` clauses) listing the acceptable types for the callee.

It is interesting how, in an analogous way to the clause case, we introduced *extra* statements to further simplify analysis. For example, the mentioned `tot` (type of this) builtin filters the execution of subsequent statements when the class of the instance is not listed in the set of possibilities; guard statements have a similar goal in clauses that come from conditional constructions. In Figure 5 the `eq` call at the beginning of the leftmost `Vector$append#1#2` clause refers to the condition for executing the first branch, while the `ne` call contains its negated version, for the second alternative. Also, those methods that are *entry* but not *extra* contain assignments to shadow variables that simulate the call-by-reference semantics [24].

```
public class Lang{

  public void foo(Location loc){
    String lang = loc.getDefaultLanguage();
    ...
  }
}

class Location {
    public String getDefaultLanguage(){
      return "English";
    }
}

class China extends Location{
    public String getDefaultLanguage(){
      return "Mandarin";
    }
}

class Sichuan extends China{
}
```

```
Lang$foo(Res,R0,R1):-
    asg(R0_,Lang,R0,Lang),
    asg(R1_,Location,R1,Location),
    Location$dyn*getDefaultLanguage(R4,R1_),
    ret.

Location$getDefaultLanguage(Res,R0):-
    asg(R0_,Location,R0,Location),
    asg(Res,java.lang.String,"English",java.lang.String),
    ret.

China$getDefaultLanguage(Res,R0):-
    asg(R0_,China,R0,China),
    asg(Res,java.lang.String,"Mandarin",java.lang.String),
    ret.

Location$dyn*getDefaultLanguage(Res,R1_):-
    tot(R1_, [China,Sichuan]),
    China$getDefaultLanguage(Res,R1_).
Location$dyn*getDefaultLanguage(Res,R1_):-
    tot(R1_, [Location]),
    Location$getDefaultLanguage(Res,R1_).
```

Fig. 6. Transformation for dynamic dispatch in Java.

5 Explicit semantics in other OO languages

Our framework can be adapted to other languages apart from Java (and Ciao), especially for those like C# that share similar syntax and statement semantics to Java. The examples in Figures 6 and 7 illustrate this point. In Figure 6, the value returned by the `getDefaultLanguage` invocation in the `foo` method returns `English` if `loc` has runtime type `Location` and `Mandarin` if the runtime type is `China` or `Sichuan`, since this last class inherits the implementation of `getDefaultLanguage` from `China` according to standard Java semantics [14]. The C# language is quite similar in most aspects, but polymorphic invocations have been further refined (and complicated). In Figure 7 only class `China` overshadows the default definition for the `getDefaultLanguage` method given in the superclass; `HongKong` inherits the `Location` implementation. Therefore, an invocation like `(new Hong Kong()).getDefaultLanguage()` returns `English`.

When analyzing a virtual invocation like the one in the first line of `foo`, we could have implemented internal mechanisms in the analyzer for differentiating the two possible interpretations that the call might have in each language. That implies an undesirable, double implementation of either the fixpoint algorithm or the abstract domains, since the analyzer would then be language-dependent. To bypass this problem, we introduce additional pseudo-builtins that contain language-dependent features. We can see in Figures 6 and 7 how the Horn clause representation is almost identical in both cases, except for the bodies of the two `Location$dyn*getDefaultLanguage` clauses. In the case of Java, we indicate that the first clause is executed if the runtime type of `this` (`tot`) is either `China` or `Sichuan`, while the second requires that variable to be of runtime type `Location`. The situation is reversed in the C# example, in which instances of `Location` and `HongKong` share the implementation

```
namespace Lang{

public class Lang{
  public void foo(Location loc){
    string lang = loc.getDefaultLanguage();
    ...
  }
}
class Location {
  public string getDefaultLanguage(){
    return "English";
  }
}
class China:Location{
  private string getDefaultLanguage(){
    return "Mandarin";
  }
}
class HongKong:China{}
}
```

```
Lang$foo(Res,R0,R1):-
    asg(R0_,Lang,R0,Lang),
    asg(R1_,Location,R1,Location),
    Location$dyn*getDefaultLanguage(R4,R1_),
    ret.

Location$getDefaultLanguage(Res,R0):-
    asg(R0_,Location,R0,Location),
    asg(Res,string,"English",string),
    ret.

China$getDefaultLanguage(Res,R0):-
    asg(R0_,China,R0,China),
    asg(Res,string,"Mandarin",string),
    ret.

Location$dyn*getDefaultLanguage(Res,R1_):-
    tot(R1_, [China]),
    China$getDefaultLanguage(Res,R1_).
Location$dyn*getDefaultLanguage(Res,R1_):-
    tot(R1_, [Location,HongKong]),
    Location$getDefaultLanguage(Res,R1_).
```

Fig. 7. Transformation for dynamic dispatch in C#.

`Location$getDefaultLocation` while invocations on objects of (exactly) class `China` are redirected to `China$getDefaultLocation`.

The abstract domain is not required to know anything about which actual language is to be analyzed but only to provide a common, correct transfer function for the `tot` builtin, which will return as output state the same input state if the instance happens to have a runtime type included in the list of accepted classes, and $\perp$ if not.

6 Experimental results

We have completed a preliminary implementation of our framework within the CiaoPP preprocessor [15]. CiaoPP offers a parametric and efficient top-down analysis engine [23, 16] with a good number of abstract domains, including the ones illustrated in this section. The efficiency of the algorithm relies on keeping dependencies between different predicates during analysis so that only the really affected parts need to be revisited after a change during the fixpoint process. In addition, recomputation is avoided using *memoization* [11, 33, 23]. Another characteristic is that it is *multivariant* (i.e., abstract calls to a given predicate that represent different input patterns are automatically analyzed separately) and follows a top-down approach, in order to allow modeling properties that depend on the data flow characteristics of the program.

We have performed two experiments with our framework using the benchmarks corresponding to the JOlden suite [31]. The first experiment is summarized in Figure 8 and shows the scalability of the transformation phase. The first three columns contain basic metrics about the application: number of classes (k), methods (m) and instructions (i). Since the latter corresponds to the bytecode representation of the source, we also list how many program points (pp) are present in the Horn clause program analyzed. This metric differs slightly

name	k	m	i	pp	ct
`jolden.health.Health`	8	30	637	933	1.1
`jolden.bh.BH`	9	70	1208	1739	3.2
`jolden.voronoi.Voronoi`	6	73	988	1340	2.2
`jolden.mst.MST`	6	36	445	665	0.1
`jolden.power.Power`	6	32	1017	1270	2.1
`jolden.treeadd.TreeAdd`	2	12	193	274	2.0
`jolden.em3d.Em3d`	4	22	447	669	0.1
`jolden.perimeter.Perimeter`	10	45	543	814	0.1
`jolden.bisort.BiSort`	2	15	323	476	0.1
`jolden.all.All`	50	317	5839	7251	11.0

Fig. 8. Statistics from the transformation phase.

from the number of instructions in the sense that extra clauses and builtins make it somewhat larger; *pp* also provides a better approximation of the size and complexity of the program analyzed because the semantics of the object-oriented program is made explicit, as seen in Section 2. The fifth column (*ct*) shows the time invested (given in seconds) in transforming the input program and producing the Horn clause version and the metainformation.

The second experiment, shown in Figure 9, illustrates the scalability, efficiency, and precision of the analysis component of our framework. We first use a simple abstract domain, Nullity, capable of approximating which variables are definitely null and which ones definitely point to a non-null location. The second abstract domain is a Class Hierarchy Analysis (CHA) [3], which uses the combination of the statically declared type of an object and the class hierarchy of the program to determine the set of possible targets of a virtual invocation. The use of a CHA shows the scalability of our framework for a domain with non-linear worst-case complexity in its operations. Additionally, it also reflects the usefulness of *metainformation* files since they are required by the CHA domain in order to access the hierarchy tree. The columns labeled pp' show the number of program points reachable by the analyses. Therefore, pp' may differ from pp because the number of analyzed program points is not always the total number of program points in the program: some commands are found to be unreachable. Since our framework is multivariant and can thus keep track of different *contexts* at each program point, at the end of analysis there may be more than one abstract state associated with each program point. Thus, the number of abstract states is typically larger than the number of reachable program points. The *ast* columns provide the total number of these abstract states inferred by analysis. The level of multivariance is the ratio ast/pp', presented in the *st* columns. In general, such a larger number for *st* tends to indicate more precise results. Running times are listed in columns *pt* (time invested in preprocessing the program which includes the extraction of metainformation for each method in the Horn clause program and the construction of the class hierarchy) and *at* (analysis time); both are also given in seconds.

Both experiments have been performed on a Pentium M 1.73Ghz with 1Gb of RAM, and averaging several runs after eliminating the best and worst values. We

		Nullity				CHA			
	pt	*pp′*	*ast*	*st*	*at*	*pp′*	*ast*	*st*	*at*
`jolden.health.Health`	2.1	921	5836	6.3	9.6	933	3542	3.8	52.1
`jolden.bh.BH`	2.2	1739	12384	7.1	50.1	1739	4757	2.7	59.4
`jolden.voronoi.Voronoi`	2.2	1277	5492	4.3	11.5	1340	5147	3.8	81.3
`jolden.mst.MST`	2.1	496	1503	3.0	1.1	665	1609	2.4	11.6
`jolden.power.Power`	2.1	1270	10560	8.3	29.9	1270	2908	2.3	32.7
`jolden.treeadd.TreeAdd`	2.0	274	880	3.2	0.6	274	729	2.6	6.1
`jolden.em3d.Em3d`	2.0	669	5565	8.3	0.9	669	3320	4.9	49.5
`jolden.perimeter.Perimeter`	2.1	814	2653	3.2	1.7	814	3731	4.5	25.0
`jolden.bisort.BiSort`	2.1	476	3353	7.0	5.8	476	1614	3.4	15.6
`jolden.all.All`	2.6	7188	48476	6.7	145.9	7251	29586	4.1	391.2

Fig. 9. Statistics for the Nullity and Class Hierarchy (CHA) domains.

chose to show separately the total times of the two phases (transformation and analysis) because we expect the transformation process to be fully run only once. Later executions can use incremental compilation for those files that changed, so that the overhead of the preprocessing phase should be almost negligible in medium to large programs. Although the same approach can be taken for the analysis [16], the current implementation is not incremental.

7 Related work

Most previous research in analysis of object-oriented programs concentrates on finding new abstract domains that better approximate a particular concrete property of the program analyzed in order to optimize compilation (e.g., [4, 28]) or statically verify certain properties about the runtime behavior of the code (e.g., [13, 19]). In contrast there has been comparatively little work on the formal specification of the *intermediate language* to which the analyzed program is transformed or in the application of existing logic programming techniques. In [25] the authors describe how to automatically derive Prolog versions of Java programs that share the same operational semantics. However, the compilation applies to a smaller subset of Java than that supported in our work and no experimental results are provided. Also, the technique is presented from a more informal perspective and no analysis is attempted over the transformed logic programs.

More closely related to ours is the work presented in [1], which draws in part on the ideas of [26]. The authors also focus on how to reuse existing logic programming tools, in order to analyze Java bytecode. The approach is based on encoding an interpreter of the Java Virtual Machine bytecode in a logic language, Ciao [6], and then partially evaluating this interpreter with respect to the concrete program to be analyzed. This results in a *residual* program which has the same semantics as the original one but is often easier to analyze than the original set of bytecode+interpreter. As in our case, the analysis and verification experiments are performed using the CiaoPP [15] tool.

While the approach of [1] is obviously very interesting, it also has the shortcoming that it is quite dependent on the quality of the results obtained by the partial evaluator. Given the state of the art in partial evaluation, this may clearly vary significantly depending on the input program. The approach presented herein is based instead on a direct translation from the Java program into a Horn clause representation, which obviates this problem, at the cost of having to write and prove correct the transformer. Also, in this translation we do not try to mimic the operational semantics of the Java program in the Horn clause version (i.e., the resulting program if run, e.g., on a Prolog system, would not necessarily produce equivalent results to those of the Java program). Instead, the aim is to *safely approximate* the semantics of the Java program in the Horn clause representation by taking advantage of the (collecting) SLD semantics assumed by the analyzer. This allows flexibility in the translation and eliminates the burden of having to simulate exactly the operational semantics of the source language since we do not want to execute the program but only to obtain safe results by analyzing it. The flexibility and directness of this approach also allows supporting a much larger subset of the language than in [1], including exceptions, inheritance, interfaces, etc. Also, presumably because of the directness of the approach with it we have been able to analyze significantly larger programs, and in less time.

In most of the (non CLP-based) abstract interpretation frameworks for analysis of Java (e.g., [4, 7]) the authors prefer to focus on particular properties and therefore their solutions (abstract domains and analysis algorithms) are tied to them, even when if they may be explicitly labeled as multipurpose [20]. In [27] the authors use a framework that is closely related to Gaia [8] (itself closely related to [23]). However, the intermediate representation is not described and the semantics of the interprocedural operations is again tied to the Java language. Also, the benchmarks used are smaller than those that we report on. The more recent Julia framework [30] is intended to be generic from the point of view of domains but once more also targets Java as unique source language. This framework is capable of analyzing large programs in a top-down way, as in our approach, the main other difference being that we support multivariance, inherited from the CiaoPP analyzer. Finally, in [22] another interesting generic static analyzer for the modular analysis and verification of Java classes is presented. The algorithm presented is also top down but is again tailored specifically to Java source.

8 Conclusions and future work

We have presented a transformation-based framework for analysis of object-oriented programs, which is generic in terms of the source language and abstract domain in use. The framework consists of a two-step process: a transformation of the program into a set of Horn clauses that represents a correct approximation of its standard semantics, and a mature and sophisticated fixpoint algorithm. We claim that our approach is flexible in the sense that the first phase decou-

ples the fixpoint algorithm from any language-dependent feature. Furthermore, our experimental evaluations support the scalability of our framework showing results for medium-sized programs as well as its efficiency analyzing them in a reasonable amount of time, and precision showing high rates of multivariance.

We have performed some promising experiments on an ample subset of Java, as shown in this paper, but our aim is to support the full Java language. Also, we are currently incorporating more sophisticated abstract domains (e.g., points-to/sharing analysis). Moreover, we expect to increase the scalability of our approach, analyzing larger programs than shown in this paper. To this end, we are studying the inclusion of modular and incremental features in our fixpoint algorithm.

References

1. E. Albert, M. Gómez-Zamalloa, L. Hubert, and G. Puebla. Verification of Java Bytecode using Analysis and Transformation of Logic Programs. In *Proc. PADL*, number 4354 in LNCS. Springer-Verlag, 2007.
2. Jim Alves-Foss, editor. *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes in Computer Science*. Springer, 1999.
3. David F. Bacon and Peter F. Sweeney. Fast static analysis of c++ virtual function calls. *Proc. of OOPSLA'96, SIGPLAN Notices*, 31(10):324–341, October 1996.
4. Bruno Blanchet. Escape Analysis for Object Oriented Languages. Application to Java(TM). In *Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'99)*, pages 20–34. ACM, November 1999.
5. M. Bruynooghe. A Practical Framework for the Abstract Interpretation of Logic Programs. *Journal of Logic Programming*, 10:91–124, 1991.
6. F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López-García, and G. Puebla (Eds.). The Ciao System. Reference Manual (v1.10). Technical report, School of Computer Science (UPM), 2004. Available at `http://www.ciaohome.org`.
7. Bor-Yuh Evan Chang and K. Rustan M. Leino. Abstract interpretation with alien expressions and heap structures. In *VMCAI'05*, number 3385 in LNCS, pages 147–163. Srpinger, 2005.
8. B. Le Charlier and P. Van Hentenryck. Experimental Evaluation of a Generic Abstract Interpretation Algorithm for Prolog. *ACM Transactions on Programming Languages and Systems*, 16(1):35–101, 1994.
9. 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.
10. Rob DeLine and K. Rustan M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, Microsoft Research, 2005.
11. S. W. Dietrich. Extension Tables: Memo Relations in Logic Programming. In *Fourth IEEE Symposium on Logic Programming*, pages 264–272, September 1987.
12. Christian Fecht. Gena - a tool for generating prolog analyzers from specifications. In *SAS '95: Proceedings of the Second International Symposium on Static Analysis*, pages 418–419, London, UK, 1995. Springer-Verlag.
13. S. Genaim and F. Spoto. Information Flow Analysis for Java Bytecode. In *Proc. of VMCAI*, LNCS. Springer-Verlag, 2005.

14. James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java(TM) Language Specification, The (3rd Edition)*. Addison-Wesley Professional, 2005.
15. M. Hermenegildo, G. Puebla, F. Bueno, and P. López-García. Program Development Using Abstract Interpretation (and The Ciao System Preprocessor). In *Proc. of SAS'03*, pages 127–152. Springer LNCS 2694, 2003.
16. M. Hermenegildo, G. Puebla, K. Marriott, and P. Stuckey. Incremental Analysis of Constraint Logic Programs. *ACM TOPLAS*, 22(2):187–223, March 2000.
17. R. Kowalski and D. Kuehner. Linear resolution with selection function. *Artificial Intelligence*, 2:227–260, 1971.
18. Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of jml: a behavioral interface specification language for java. *SIGSOFT Softw. Eng. Notes*, 31(3):1–38, 2006.
19. Xavier Leroy. Java bytecode verification: An overview. In *CAV'01*, number 2102 in LNCS, pages 265–285. Springer, 2001.
20. Tal Lev-Ami and Shmuel Sagiv. TVLA: A system for implementing static analyses. In *SAS*, number 1824 in LNCS, pages 280–301. Springer, 2000.
21. F. Logozzo and A. Cortesi. Abstract interpretation and object-oriented languages: quo vadis? In *Proc. of the 1st. Int'l. Workshop on Abstract Interpretation of Object-oriented Languages (AIOOL'05)*, ENTCS. Elsevier Science, January 2005.
22. Francesco Logozzo. Cibai: An abstract interpreation-based static analyzer for modular analysis and verification of java classes. In *VMCAI'07*, number 4349 in LNCS. Springer, Jan 2007.
23. K. Muthukumar and M. Hermenegildo. Compile-time Derivation of Variable Dependency Using Abstract Interpretation. *JLP*, 13(2/3):315–347, 1992.
24. J. Navas, M. Méndez-Lojo, and M. Hermenegildo. An Efficient, Context and Path Sensitive Analysis Framework for Java Programs. In *9th Workshop on Formal Techniques for Java-like Programs FTfJP 2007*, July 2007.
25. J. Peralta and J.Cruz-Carlon. From static single-assignment form to definite programs and back. Extended abstract in International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR), July 2006.
26. J.C. Peralta, J. Gallagher, and H. Sağlam. Analysis of imperative programs through analysis of constraint logic programs. In G. Levi, editor, *Static Analysis. 5th International Symposium, SAS'98, Pisa*, volume 1503 of *LNCS*, pages 246–261, 1998.
27. Isabelle Pollet. *Towards a generic framework for the abstract interpretation of Java*. PhD thesis, Catholic University of Louvain, 2004. Dept. of Computer Science.
28. Erik Ruf. Effective synchronization removal for java. *PLDI'00, SIGPLAN Notices*, 35(5):208–218, 2000.
29. S. Secci and F. Spoto. Pair-sharing analysis of object-oriented programs. In *SAS*, pages 320–335, 2005.
30. F. Spoto. JULIA: A Generic Static Analyser for the Java Bytecode. In *Proc. of the 7th Workshop on Formal Techniques for Java-like Programs, FTfJP'2005*, Glasgow, Scotland, July 2005.
31. JOlden Suite. `http://www-ali.cs.umass.edu/DaCapo/benchmarks.html`.
32. Raja Vallee-Rai, Laurie Hendren, Vijay Sundaresan, Patrick Lam, Etienne Gagnon, and Phong Co. Soot - a Java optimization framework. In *Proceedings of CASCON 1999*, pages 125–135, 1999.
33. R. Warren, M. Hermenegildo, and S. K. Debray. On the Practicality of Global Flow Analysis of Logic Programs. In *Fifth International Conference and Symposium on Logic Programming*, pages 684–699. MIT Press, August 1988.

Preserving Sharing in the Partial Evaluation of Lazy Functional Programs*

Sebastian Fischer[1], Josep Silva[2], Salvador Tamarit[2], and Germán Vidal[2]

[1] University of Kiel, Olshausenstr. 40, D-24098 Kiel, Germany.
sebf@informatik.uni-kiel.de
[2] Technical University of Valencia, Camino de Vera S/N, E-46022 Valencia, Spain.
{jsilva,stamarit,gvidal}@dsic.upv.es

Abstract. The goal of partial evaluation is the specialization of programs w.r.t. part of their input data. Although this technique is already well-known in the context of functional languages, current approaches are either overly restrictive or destroy sharing through the specialization process, which is unacceptable from a performance point of view. In this work, we present a new partial evaluation scheme for first-order lazy functional programs that preserves sharing through the specialization process and still allows the unfolding of arbitrary functions.

1 Introduction

Partial evaluation [7] is an automatic technique for the specialization of programs. This technique has already been developed for a variety of programming languages, like C [4], Curry [12], Prolog [9], Scheme [13], etc.

In this work, we focus on a problem associated to the partial evaluation of *lazy* functional languages. In these languages (e.g., Haskell [10]), it is essential to *share* program variables in order to avoid losing efficiency due to the repeated evaluation of the same expression. Consider, e.g., the following program excerpt:[3]

$$\begin{array}{ll}
\texttt{sumList}([\,]) & = \texttt{Z} \\
\texttt{sumList}(\texttt{x} : \texttt{xs}) & = \texttt{add}(\texttt{x}, \texttt{sumList}(\texttt{xs})) \\
\texttt{incList}(\texttt{n}, [\,]) & = [\,] \\
\texttt{incList}(\texttt{n}, \texttt{x} : \texttt{xs}) & = \texttt{add}(\texttt{n}, \texttt{x}) : \texttt{incList}(\texttt{n}, \texttt{xs}) \\
\texttt{add}(\texttt{Z}, \texttt{m}) & = \texttt{m} \\
\texttt{add}(\texttt{S}(\texttt{n}), \texttt{m}) & = \texttt{S}(\texttt{add}(\texttt{n}, \texttt{m}))
\end{array}$$

where function `sumList` sums the elements of a list, `incList` increments the elements of a list by a given number, and `add` performs the addition of two natural numbers. Now, consider a partial evaluation of the following function call: $\texttt{sumList}(\texttt{incList}(\boxed{e}, \texttt{Z} : \texttt{Z} : [\,]))$, where $\boxed{e}$ is any arbitrary expression:

$$\texttt{sumList}(\texttt{incList}(\boxed{e}, \texttt{Z} : \texttt{Z} : [\,])) \;\Rightarrow\; \texttt{sumList}(\texttt{add}(\boxed{e}, \texttt{Z}) : \texttt{incList}(\boxed{e}, \texttt{Z} : [\,]))$$

* This work has been partially supported by the EU (FEDER) and the Spanish MEC under grants TIN2005-09207-C03-02 and *Acción Integrada* HA2006-0008.

[3] We use [ ] and ":" as constructors of lists and `Z` and `S` to built natural numbers.

Note that, although the expression $\boxed{e}$ appears twice, it will only be evaluated once in current lazy programming languages since the two occurrences of variable `n` in the second rule of function `incList` are shared. If we build a residual rule—a *resultant*—associated to the above partial evaluation, we would get the rule

$$\texttt{new\_function}(\ldots) \;=\; \texttt{sumList}(\texttt{add}(\boxed{e}, \texttt{Z}) : \texttt{incList}(\boxed{e}, (\texttt{Z} : [\,])))$$

Now, however, if we evaluate `new_function` using the above rule, the expression $\boxed{e}$ will be evaluated twice since both occurrences are not shared anymore, which is unacceptable from a performance point of view.

Current partial evaluation schemes for lazy functional (logic) languages have mostly ignored this point.[4] Usually, partial evaluators include a restriction so that the unfolding of functions whose right-hand side is not *linear* (i.e., whose right-hand side contains multiple occurrences of the same variable) is forbidden.

In this work, we present an alternative to such trivial, overly restrictive treatment of sharing during partial evaluation. In particular, we would like to produce a residual rule of the following form:

$$\begin{array}{lcl} \texttt{new\_function}(\ldots) & = & \texttt{let}\ \ \texttt{w} = \boxed{e} \\ & & \texttt{in}\ \ \ \texttt{sumList}(\texttt{add}(\texttt{w}, \texttt{Z}) : \texttt{incList}(\texttt{w}, \texttt{Z} : [\,])) \end{array}$$

In this way, the two occurrences of the fresh variable `w` would be shared and $\boxed{e}$ would not be evaluated twice. In principle, one could define a post-unfolding phase where, given a partial evaluation $e_0 \Rightarrow e_1 \Rightarrow \ldots \Rightarrow e_n$, every occurrence of a common subexpression e in e_n would be replaced by a fresh variable w and a new let of the form $let\ \ w = e\ \ in\ \ \ldots$ would be added. However, if the semantics used for partial evaluation does not model sharing, the identification of common subexpressions would be rather difficult because their degree of evaluation need not be the same.

In contrast, here we present a novel method which is based on a lazy semantics [1] that models variable sharing by means of an updatable heap, which is appropriately extended in order to perform symbolic computations. Then, we also define how residual rules should be extracted from these symbolic computations. For simplicity, we will not introduce the details of a complete partial evaluation scheme (but it would be similar to that of [2] by replacing the underlying partial evaluation semantics and the construction of residual rules from partial computations, i.e., control issues would remain basically unaltered).

2 Preliminaries

We consider in this work a simple, first-order lazy functional language. The syntax is shown in Fig. 1, where we write $\overline{o_n}$ for the *sequence of objects* $o_1, \ldots, o_n$. A program consists of a sequence of function definitions such that the left-hand side has pairwise different variable arguments. The right-hand side is an expression

[4] We note that this is a critical issue that has been considered in the context of *inlining* (see, e.g., [11]), which could be seen like a rather simple form of partial evaluation.

$P ::= D_1 \ldots D_m$	(program)	*Domains*		
$D ::= f(x_1, \ldots, x_n) = e$	(function definition)			
$e ::= x$	(variable)	$P_1, P_2, \ldots \in \mathit{Prog}$	(Programs)	
$\mid\ c(x_1, \ldots, x_n)$	(constructor call)	$x, y, z, \ldots \in \mathit{Var}$	(Variables)	
$\mid\ f(x_1, \ldots, x_n)$	(function call)	$a, b, c, \ldots \in \mathcal{C}$	(Constructors)	
$\mid\ \mathit{let}\ \{\overline{x_k = e_k}\}\ \mathit{in}\ e$	(let binding)	$f, g, h, \ldots \in \mathcal{F}$	(Functions)	
$\mid\ \mathit{case}\ x\ \mathit{of}\ \{\overline{p_k \to e_k}\}$	(case expression)	$p_1, p_2, p_3, \ldots \in \mathit{Pat}$	(Patterns)	
$p ::= c(x_1, \ldots, x_n)$	(pattern)			

Fig. 1. Syntax for normalized flat programs

composed by variables, data constructors, function calls, let bindings (where the local variables $\overline{x_k}$ are only visible in $\overline{e_k}$ and e), and case expressions of the form $\mathit{case}\ x\ \mathit{of}\ \{c_1(\overline{x_{n_1}}) \to e_1; \ldots; c_k(\overline{x_{n_k}}) \to e_k\}$, where x is a variable, $c_1, \ldots, c_k$ are different constructors, and $e_1, \ldots, e_k$ are expressions. The *pattern variables* $\overline{x_{n_i}}$ are introduced locally and bind the corresponding variables of e_i.

Observe that, according to Fig. 1, the arguments of function and constructor calls are variables. As in [8], this is essential to express sharing without the use of graph structures. This is not a serious restriction since source programs can be *normalized* so that they follow the syntax of Fig. 1 (see, e.g., [8, 1]).

Laziness of computations will show up in the description of the behavior of function calls and case expressions. In a function call, parameters are not evaluated but directly passed to the body of the function. In a case expression, the outermost symbol of the case argument is required. Therefore, the case argument should be evaluated to *head normal form* [6] (i.e., a variable or an expression with a constructor at the outermost position).

3 Partial Evaluation of Lazy Functional Programs

The main ingredients of our new proposal which preserves sharing through the specialization process are the following: i) partial computations are performed with a lazy semantics that models sharing by means of an updatable heap (cf. Sect. 3.1); ii) this semantics is then extended in order to perform symbolic computations during partial evaluation (cf. Sect. 3.2); and iii) we introduce a method to extract residual rules from partial computations (cf. Sect. 3.3).

3.1 The Standard Semantics

First, we present a lazy evaluation semantics for our first-order functional programs that models sharing. The rules of the small-step semantics are shown in Fig. 2 (they are a simplification of the calculus in [1], which in turn originates from an adaptation of Launchbury's natural semantics [8]). It follows these naming conventions:

$$\Gamma, \Delta, \Theta \in \mathit{Heap} \ = \ \mathit{Var} \to \mathit{Exp} \qquad v \in \mathit{Value} \ ::= \ x \mid c(\overline{x_n})$$

var

$$\langle \Gamma[x \mapsto e], x, S\rangle \Rightarrow \langle \Gamma[x \mapsto e], e, x : S\rangle \qquad \text{where } e \neq x$$

val

$$\langle \Gamma, v, x : S\rangle \Rightarrow \langle \Gamma[x \mapsto v], v, S\rangle \qquad \text{where } v \text{ is a value}$$

fun

$$\langle \Gamma, f(\overline{x_n}), S\rangle \Rightarrow \langle \Gamma, \rho(e), S\rangle \qquad \text{where } f(\overline{y_n}) = e \in P \text{ and } \rho = \{\overline{y_n \mapsto x_n}\}$$

let

$$\langle \Gamma, let\ \{\overline{x_k = e_k}\}\ in\ e, S\rangle \Rightarrow \langle \Gamma[\overline{y_k \mapsto \rho(e_k)}], \rho(e), S\rangle \quad \begin{array}{l}\text{where } \rho = \{\overline{x_k \mapsto y_k}\}\\ \text{and } \overline{y_n} \text{ are fresh variables}\end{array}$$

case

$$\langle \Gamma, case\ e\ of\ \{\overline{p_k \to e_k}\}, S\rangle \Rightarrow \langle \Gamma, e, \{\overline{p_k \to e_k}\} : S\rangle$$

select

$$\langle \Gamma, c(\overline{x_n}), \{\overline{p_k \to e_k}\} : S\rangle \Rightarrow \langle \Gamma, \rho(e_i), S\rangle \quad \begin{array}{l}\text{where } p_i = c(\overline{y_n}) \text{ and } \rho = \{\overline{y_n \mapsto x_n}\},\\ \text{with } i \in \{1, \ldots, k\}\end{array}$$

Fig. 2. Small-Step Semantics for (Sharing-Based) Lazy Functional Programs

A *heap* is a 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 with $\Gamma[x] = e$, i.e., we use this notation either as a condition on a heap Γ or as a modification of Γ. A *value* is a constructor-rooted term (i.e., a term whose outermost function symbol is a constructor symbol).

A *state* of the small-step semantics is a triple $\langle \Gamma, e, S\rangle$, where Γ is the current heap, e is the expression to be evaluated (often called the *control* of the small-step semantics), and S is the stack which represents the current context. We briefly describe the transition rules:

- In rule var, the evaluation of a variable x that is bound to an expression e proceeds by evaluating e and adding to the stack the reference to variable x. If a value v is eventually computed and there is a variable x on top of the stack, rule val updates the heap with $x \mapsto v$. This rule achieves the effect of sharing since the next time the value of variable x is demanded, the value v will be returned thus avoiding the repeated evaluation of e.
- Rule fun implements a simple function unfolding. We assume that the considered program P is a global parameter of the calculus and that the variables of the rule are fresh in every application of rule fun.
- In order to reduce a let construct, rule let adds the bindings to the heap and proceed with the evaluation of the main argument of *let*. Note that the variables introduced by the let construct are renamed with fresh names in order to avoid variable name clashes.
- Rule case initiates the evaluation of a case expression by evaluating the case argument and pushing the alternatives $\{\overline{p_k \to e_k}\}$ on top of the stack. If a

value is eventually reached, then rule select is used to select the appropriate branch and continue with the evaluation of this branch.

In order to evaluate an expression e, we construct an *initial state* of the form $\langle [\,], e, [\,] \rangle$ and apply the rules of Fig. 2. We denote by $\Rightarrow^*$ the reflexive and transitive closure of $\Rightarrow$.

3.2 The Partial Evaluation Semantics

To perform partial computations in a functional context, missing data is usually denoted by free variables. Therefore, the semantics of Fig. 2 is not appropriate to perform computations at partial evaluation time. Here, we follow the approach of [3] and introduce a *residualizing* version of the standard semantics as follows.

First, logical variables are used to represent missing information. In a heap Γ, a logical variable x is represented by a circular binding $x \mapsto x$ such that $\Gamma[x] = x$. Now, they are also considered *values* in rule val.

In the new semantics, we assume that the rules of the semantics are applied until no more rule is applicable. The termination of partial computations is ensured by using function *annotations*.[5] Basically, annotated functions—we use an underscore to annotate function calls and case expressions—should not be unfolded in order to have a finite computation.[6] Underlined function calls and case expressions are also treated as values in rule val.

Because of the introduction of the new "values" (logical variables and annotated functions and cases), rule select does not suffice anymore to evaluate a case expression whose argument reduces to a value. Therefore, we introduce the following new rules, which are shown in Fig. 3:

- First, rule fun_stop applies when the argument of a case expression evaluates to an annotated function call $\underline{f}(\overline{x_n})$. The current stack, $x : \{\overline{p_k \to e_k}\} : S$, means that the original case expression had the form *case* x *of* $\{\overline{p_k \to e_k}\}$, so that x was eventually reduced to $\underline{f}(\overline{x_n})$. In this case, we annotate the original case expression, update the binding for x, and return the annotated case expression. Intuitively, once an annotated function call suspends the computation, we should reconstruct the context in the heap and terminate the computation.
- Rule case_stop proceeds in a similar way, the only difference being that the computed value is now an annotated case expression.
- Rule guess applies when the argument of a case expression reduces to a logical variable (i.e., to some missing data). Here, rather than suspending the computation, we return the annotated case expression, which can then be reduced by rules case_of_case and residualize, depending on the current stack.

[5] Note that we consider an *offline* scheme for partial evaluation for simplicity; indeed, our main contributions (the partial evaluation semantics and the extraction of residual rules) could also be used within an online scheme.

[6] We do not deal with termination issues in this paper but refer the interested reader to, e.g., [12, 5].

fun_stop

$$\langle \Gamma, \underline{f}(\overline{x_n}), x : \{\overline{p_k \to e_k}\} : S\rangle \;\Rightarrow\; \langle \Gamma[x \mapsto \underline{f}(\overline{x_n})], \underline{case}\ x\ of\ \{\overline{p_k \to e_k}\}, S\rangle$$

case_stop

$$\langle \Gamma, \underline{case}\ y\ of\ \{\overline{p'_q \to e'_q}\}, x : \{\overline{p_k \to e_k}\} : S\rangle \Rightarrow \langle \Gamma[x \mapsto \underline{case}\ y\ of\ \{\overline{p'_q \to e'_q}\}], \underline{case}\ x\ of\ \{\overline{p_k \to e_k}\}, S\rangle$$

guess

$$\langle \Gamma[x \mapsto x], x, \{\overline{p_k \to e_k}\} : S\rangle \;\Rightarrow\; \langle \Gamma[x \mapsto x], \underline{case}\ x\ of\ \{\overline{p_k \to e_k}\}, S\rangle$$

case_of_case

$$\langle \Gamma[x \mapsto x], \underline{case}\ x\ of\ \{\overline{p'_m \to e'_m}\}, \{\overline{p_k \to e_k}\} : S\rangle \Rightarrow \langle \Gamma, \underline{case}\ x\ of\ \{\overline{p'_m \to case\ e'_m\ of\ \{\overline{p_k \to e_k}\}}\}, S\rangle$$

residualize

$$\langle \Gamma[x \mapsto x], \underline{case}\ x\ of\ \{\overline{p_k \to e_k}\}, [\,]\rangle \;\Rightarrow\; case\ x\ of\ \{\overline{p'_k \to \langle \Gamma[x \mapsto p'_k, \overline{y_{nk} \mapsto y_{nk}}], e'_k, [\,]\rangle}\}$$
where $p_i = c(\overline{x_{ni}})$, $\rho_i = \{\overline{x_{ni} \mapsto y_{ni}}\}$, $\overline{y_{ni}}$ are fresh,
with $p'_i = \rho_i(p_i)$, and $e'_i = \rho_i(e_i)$, for all $i = 1, \ldots, k$

Fig. 3. Partial Evaluation Rules

– Rule case_of_case (originally introduced in the context of deforestation [14]) is used to reduce a case whose argument is another case with a logical variable as argument. This rule moves the outer case to the branches of the inner case. Intuitively, it is used to lift case expressions with a logical variable, i.e., non-deterministic choices, to the topmost position so that rule residualize applies. Essentially, rule residualize *residualizes* the case expression (i.e., it is already considered part of the residual code) and continue with the evaluation of the different branches. Note that bindings of the form $x \mapsto p'_i$, $i = 1, \ldots, k$, are applied to the different branches so that information is propagated forward in the computation. As in rule let, we rename the variables of the case patterns to avoid variable name clashes, so that p'_i and e'_i denote the renaming of p_i and e_i, respectively. Moreover, since the pattern variables of p'_i are not bound in e'_i, we add them to the heap as logical variables, i.e., as circular bindings of the form $\overline{x_{ni} \mapsto x_{ni}}$.

Note that the new rules are basically required in order to deal with missing information and annotated function calls. The preservation of sharing through the specialization process is achieved thanks to the use of a standard semantics that models sharing.

Observe that, if we apply the rules of the partial evaluation semantics as much as possible, every state $\langle \Gamma, e, S\rangle$ in the derived expression (if the computation does not fail) would have an empty stack. This is an easy consequence of the fact that every function and case expression is either reduced, annotated or residualized, so that an empty stack is finally obtained.

$$
\begin{array}{lllr}
 & \langle [\,], & \texttt{let } \{\texttt{x} = \texttt{x}, \texttt{w} = \underline{\texttt{double}}(\texttt{x})\} \texttt{ in } \texttt{double}(\texttt{w}), & [\,]\rangle \\
\Rightarrow_{\mathsf{let}} & \langle [\texttt{x} \mapsto \texttt{x}, & \texttt{double}(\texttt{w}), & [\,]\rangle \\
 & \texttt{w} \mapsto \underline{\texttt{double}}(\texttt{x})], & & \\
\Rightarrow_{\mathsf{fun}} & \langle [\texttt{x} \mapsto \texttt{x}, & \texttt{add}(\texttt{w}, \texttt{w}), & [\,]\rangle \\
 & \texttt{w} \mapsto \underline{\texttt{double}}(\texttt{x})], & & \\
\Rightarrow_{\mathsf{fun}} & \langle [\texttt{x} \mapsto \texttt{x}, & \texttt{case w of} & [\,]\rangle \\
 & \texttt{w} \mapsto \underline{\texttt{double}}(\texttt{x})], & \{\texttt{Z} \to \texttt{w};\ \texttt{S(u)} \to \texttt{let } \{\texttt{v} = \texttt{add}(\texttt{u}, \texttt{w})\} \texttt{ in } \texttt{S(v)}\} & \\
\Rightarrow_{\mathsf{case}} & \langle [\texttt{x} \mapsto \texttt{x}, & \texttt{w}, & [\{\ldots\}]\rangle \\
 & \texttt{w} \mapsto \underline{\texttt{double}}(\texttt{x})], & & \\
\Rightarrow_{\mathsf{var}} & \langle [\texttt{x} \mapsto \texttt{x}, & \underline{\texttt{double}}(\texttt{x}), & [\texttt{w}, \{\ldots\}]\rangle \\
 & \texttt{w} \mapsto \underline{\texttt{double}}(\texttt{x})], & & \\
\Rightarrow_{\mathsf{fun\_stop}} & \langle [\texttt{x} \mapsto \texttt{x}, & \underline{\texttt{case}}\ \texttt{w of} & [\,]\rangle \\
 & \texttt{w} \mapsto \underline{\texttt{double}}(\texttt{x})], & \{\texttt{Z} \to \texttt{w};\ \texttt{S(u)} \to \texttt{let } \{\texttt{v} = \texttt{add}(\texttt{u}, \texttt{w})\} \texttt{ in } \texttt{S(v)}\} &
\end{array}
$$

Fig. 4. Derivation with the partial evaluation semantics

The following simple example illustrates the way our new semantics deals with sharing in a partial computation.

Example 1. Consider the following simple program:

$$
\begin{array}{l}
\texttt{double}(\texttt{x}) = \texttt{add}(\texttt{x}, \texttt{x}) \\
\texttt{add}(\texttt{n}, \texttt{m}) \ \ = \texttt{case n of } \{\texttt{Z} \to \texttt{m};\ \texttt{S(u)} \to \texttt{let } \{\texttt{v} = \texttt{add}(\texttt{u}, \texttt{m})\} \texttt{ in } \texttt{S(v)}\}
\end{array}
$$

Given the initial state $\langle [\,],\ \texttt{let } \{\texttt{x} = \texttt{x}, \texttt{w} = \underline{\texttt{double}}(\texttt{x})\} \texttt{ in } \texttt{double}(\texttt{w}),\ [\,]\rangle$, we have the computation shown in Fig. 4. Note that, thanks to the use of the partial evaluation semantics, we can evaluate the considered expression as much as needed but we still keep track of shared expressions in the associated heap.

3.3 Extracting Residual Rules

Now, we consider how residual rules are extracted from the computations performed with the semantics of Fig. 2 and 3.

Definition 1 (resultant). *Let P be an annotated program and e be an expression. Let $\langle [\,], e, [\,]\rangle \Rightarrow^* e'$ be a computation with the rules of Fig. 2 and 3 such that e' is irreducible. The associated resultant is given by the following rule:*

$$f(\overline{x_n}) \ = \ [\![del(e')]\!]$$

where f is a fresh function symbol,[7] *$\overline{x_n}$ are the logical variables of e, function del removes the annotations (if any), and the function $[\![\ ]\!]$ is defined as follows:*

$$[\![e]\!] \ = \begin{cases} case\ x\ of\ \{\overline{p_k \to [\![e_k]\!]}\} & if\ e = case\ x\ of\ \{\overline{p_k \to e_k}\} \\ let\ \overline{\Gamma}\ in\ e' & if\ e = \langle \Gamma, e', [\,]\rangle \end{cases}$$

[7] Consequently, some calls in the right-hand side should also be renamed. We do not deal with renaming of function calls in this paper; nevertheless, standard techniques would be applicable.

Here, $\overline{\Gamma}$ represents the set of bindings stored in Γ except those for $\overline{x_n}$ (which are now the parameters of the new function) as well as those which depend on $\overline{x_n}$.

Let us illustrate the extraction of a residual rule with an example.

Example 2. Consider the computation of Example 1 shown in Fig. 4. The associated resultant is as follows:

$$\begin{array}{lll} \mathtt{f(x)} & = & [\![\langle\, [\mathtt{x} \mapsto \mathtt{x}, \quad\quad\quad\quad \mathtt{case\ w\ of} \\ & & \quad \mathtt{w} \mapsto \mathtt{double(x)}],\ \{\mathtt{Z} \to \mathtt{w};\ \mathtt{S(u)} \to \mathtt{let}\ \{\mathtt{v} = \mathtt{add(u,w)}\}\ \mathtt{in}\ \mathtt{S(v)}\},\ [\,]\,\rangle]\!] \end{array}$$

which is reduced to

$$\begin{array}{ll} \mathtt{f(x)} & = \mathtt{let}\ \{\mathtt{w} \mapsto \mathtt{double(x)}\}\ \mathtt{in} \\ & \quad \mathtt{case\ w\ of}\ \{\mathtt{Z} \to \mathtt{w};\ \mathtt{S(u)} \to \mathtt{let}\ \{\mathtt{v} = \mathtt{add(u,w)}\}\ \mathtt{in}\ \mathtt{S(v)}\} \end{array}$$

Observe that sharing is preserved despite the unfolding of a function which is not right-linear (i.e., `double`). Note also that inlining the let expression (i.e., replacing all occurrences of `w` by `double(x)`) would destroy this property since `double` would be evaluated twice, once as an argument of the case expression and another one when selecting the corresponding case branch.

3.4 Correctness

The correctness of our approach to the partial evaluation of first-order lazy functional programs relies on two results. On the one hand, one should prove that the partial evaluation semantics is somehow equivalent to the standard one. Basically, this means that, given a computation with the partial evaluation semantics, $e \Rightarrow^* e'$, it represents every possible computation with the standard semantics (i.e., the only difference is that non-deterministic branching is encoded by means of residualized case expressions). A similar proof (though for the simpler LNT semantics without sharing) can be found in [3].

Regarding the extraction of resultants from computations with the partial evaluation semantics, its correctness can easily be proved by exploiting previous results and the clear operational equivalence between a configuration of the form $\langle \Gamma, e, [\,]\rangle$ and an expression like $let\ \overline{\Gamma}\ in\ e$.

4 Discussion

Despite the extensive literature on partial evaluation, we are not aware of any approach to the specialization of lazy functional (logic) languages where sharing is preserved through the specialization process in a non-trivial way. For instance, [2, 3] presents a partial evaluation scheme for a lazy language but sharing is not preserved since the underlying semantics does not model variable sharing.

In this paper, we have presented a promising approach by first extending a standard semantics (where sharing is modeled by using an updatable heap) and, then, defining a method to properly extract the associated residual rules. Our

new approach is not overly restrictive since every function can be unfolded (even if it is not right-linear) and still preserves sharing, thus avoiding the introduction of redundant computations in the residual program.

An implementation of the new partial evaluation scheme has been undertaken by extending a previous offline partial evaluator [12].

References

1. E. Albert, M. Hanus, F. Huch, J. Oliver, and G. Vidal. Operational Semantics for Declarative Multi-Paradigm Languages. *Journal of Symbolic Computation*, 40(1):795–829, 2005.
2. E. Albert, M. Hanus, and G. Vidal. A Practical Partial Evaluation Scheme for Multi-Paradigm Declarative Languages. *Journal of Functional and Logic Programming*, 2002(1), 2002.
3. E. Albert, M. Hanus, and G. Vidal. A Residualizing Semantics for the Partial Evaluation of Functional Logic Programs. *Information Processing Letters*, 85(1):19–25, 2003.
4. L.O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, 1994.
5. G. Arroyo, J.G. Ramos, J. Silva, and G. Vidal. Improving Offline Narrowing-Driven Partial Evaluation using Size-Change Graphs. In *Logic-based Program Synthesis and Transformation (revised and selected papers from LOPSTR'06)*, pages 60–76. Springer LNCS 4407, 2007.
6. H.P. Barendregt. *The Lambda Calculus—Its Syntax and Semantics*. Elsevier, 1984.
7. N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, Englewood Cliffs, NJ, 1993.
8. J. Launchbury. A Natural Semantics for Lazy Evaluation. In *Proc. of the ACM Symp. on Principles of Programming Languages (POPL'93)*, pages 144–154. ACM Press, 1993.
9. M. Leuschel, D. Elphick, M. Varea, S. Craig, and M. Fontaine. The Ecce and Logen Partial Evaluators and Their Web Interfaces. In *Proc. of PEPM'06*, pages 88–94. IBM Press, 2006.
10. S. Peyton-Jones, editor. *Haskell 98 Language and Libraries—The Revised Report*. Cambridge University Press, 2003.
11. S.L. Peyton Jones and S. Marlow. Secrets of the Glasgow Haskell Compiler Inliner. *Journal of Functional Programming*, 12(4&5):393–433, 2002.
12. J.G. Ramos, J. Silva, and G. Vidal. Fast Narrowing-Driven Partial Evaluation for Inductively Sequential Systems. In *Proc. of the 10th ACM SIGPLAN Int'l Conf. on Functional Programming (ICFP'05)*, pages 228–239. ACM Press, 2005.
13. P. Thiemann. The Program Generator Generator PGG. Available from the URL: `http://www.informatik.uni-freiburg.de/proglang/software/pgg/`.
14. P.L. Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.

Denotation by Transformation
Towards Obtaining a Denotational Semantics by Transformation to Point-free Style

Bernd Braßel and Jan Christiansen

Institute of Computer Science
University of Kiel, 24098 Kiel, Germany
{bbr,jac}@informatik.uni-kiel.de

Abstract. It has often been observed that a *point-free style* of programming provides a more abstract view on programs. In the middle-term we aim to use the gain in abstraction to obtain a denotational semantics for *functional logic languages* in a straightforward way. Here we propose a set of basic operations based on which arbitrary functional logic programs can be transformed to point-free programs. Surprisingly, the additional features of functional logic languages do require *less* basic operations to obtain point-free programs than known approaches for functional languages. This effect is mostly due to employing so called *function patterns*.

1 Introduction

The importance of a *point-free* view on *programming* has been emphasized particularly in the applications of category theory to semantics of programming languages. We expect the application of point-free style to declarative programming to be very fruitful. Our medium-term aim is to obtain a *relation algebraic* semantics for functional logic languages by interpreting a set of basic operations within that algebra. Point-free versions of arbitrary functional logic programs can then be composed from these basic operations. On the other hand, we show in this paper that every program can be transformed to point-free style such that the relation algebraic semantics would cover the whole language. Such semantics would then enable us to use algebraic calculations to optimise operator definitions and to concisely express soundness results for sophisticated techniques like partial evaluation, as done in an early work on relation algebraic semantics [16].

In this paper we present a transformation to express arbitrary functional logic programs in a point-free style, fully taking account of laziness.

1.1 Functional Logic Languages

We consider a functional logic program as a constructor-based rewriting system, allowing extra variables on the right hand side and so called *function patterns*. This section establishes some of the involved notation, referring to [10] for functional logic programming and [2] for function patterns. For our examples we adopt the syntax of Curry [12].

For a program P, Σ_P is a signature partitioned into two sets, the set of *constructors* $\mathcal{C}_P$ and the set of defined *operations* $\mathcal{O}_P$. We denote n-ary constructor (operation) symbols by c^n (f^n, g^n) omitting the arity where it is apparent. For a set of (sorted) variables $\mathcal{X}$, the sets of (well-sorted) *terms* and *constructor terms* are denoted by $\mathcal{T}(\Sigma_P, \mathcal{X})$ and $\mathcal{T}(\mathcal{C}_P, \mathcal{X})$, respectively. The function $var(t)$ yields the set of variables occurring in term t. A term is *linear* if every variable occurs at most once. Sorts and constructors are introduced by a `data` declaration, as shown in Example (1). The "`a`" in the third declaration denotes that `[a]` is a polymorphic type. We use syntactic sugar for list terms, e.g., `[True,False]` instead of `(True : (False : []))`. Operations are defined by *rewrite rules* of the form "$f\ p_1\ \ldots\ p_n = e$" where $f^n \in \mathcal{O}_P$ and $p_1, \ldots, p_n$ are called *patterns*. The standard way to define an operation in Curry is that each pattern of the rewrite rules is a *constructor term* and each variable occurs not more than once in the whole pattern. In other words, the term $(p_1, \ldots, p_n)$ must be a *linear constructor term*. Such standard Curry programs are evaluated by *weakly needed narrowing* [11]. A *narrowing step* is a rewrite step combined with substitutions for extra variables needed to match the left-hand side of an applicable rule. E.g., a narrowing step for (2) is `app x [True]` $\rightarrow\{$`x` $\mapsto$ `[]`$\}$ `[True]`.

```
data Success = Success
data Bool = True | False          (1)
data [a] = [] | a : [a]
```

```
app []     ys = ys
app (x:xs) ys = x : (app xs ys)   (2)
```

In addition to defining rules, *type signatures* are used to declare the sorts an operation is defined for. As an example, `app :: [a] -> [a] -> [a]` declares that `app` is an operation which maps two (polymorphic) lists to a list. These lists have elements of the same type.

As in Example (2) there might be more than one possible narrowing step. Functional logic languages provide *non-deterministic search* to obtain values in this situation. Non-determinism does not only stem from narrowing but also from operator definitions with overlapping left hand sides.

```
coin :: Bool
coin = True     (3)
coin = False
```

E.g., there are two derivations `coin` $\rightarrow$ `True` and `coin` $\rightarrow$ `False` or, for short, `coin` $\rightarrow$ `True | False`. The operation `coin` is very popular because it can be used to exemplify an important feature of functional logic languages: *call-time choice*, cf. [9]. With call-time choice non-deterministic choices for arguments are done before application, at least conceptually. In combination with laziness call-time choice is affine to the concept of referential transparency, as illustrated in the next example.

```
or :: Bool -> Bool -> Bool
or x y = if x then x else y   (4)
```

For example, employing call-time choice the expression e := `or coin True` is evaluated to `True` only, because both occurrences of `x` are substituted with the same value. That is, employing call-time choice there are the derivations $e \rightarrow$ `if True then True else True` $\rightarrow$ `True` and $e \rightarrow$ `if False then False else True` $\rightarrow$ `True`. In the dual conception, run-time choice, `e` is reduced to `if True then coin else True` $\rightarrow$ `coin` $\rightarrow$ `True | False`

and `if False then coin else True` → `True`. That is, the evaluation of e yields non-deterministically `True` or `False`.

An important operation in functional logic languages is the *strict equality* `(=:=) :: a -> a -> Success`. The intended meaning is that the equation e_1`=:=`e_2 is satisfied iff e_1 and e_2 can be reduced to the same constructor term, see [8] for a detailed discussion. In Curry, satisfying a predicate like the above equation is modelled by a reduction to the special type `Success`, cf. Program (1).

Strict equality can be employed to allow a certain type of non-standard operator definitions. A non-linear left hand side of a rewrite rule $l = e$ with x occurring n times in l can be taken as syntactic sugar for a rule where x is replaced by different variables `x1` ...`x`n in l and e is extended by constraints `(x1=:=x2)`, ..., `(x1=:=x`n`)`. See [1, 4.1] for a discussion of this transformation.

Finally, *function patterns* [2] allow operator definitions with arbitrary first order patterns. The intended meaning of a function pattern is that only the pattern is evaluated to a constructor term. The argument is evaluated until a unification is possible. Unlike extra variables unified with strict equality `(=:=)` this unification may bind a pattern variable to an unevaluated term. Non-linear patterns are still treated with `(=:=)` as described above.

```
last (app xs [x]) = x        (5)
```

The operation `last` yields the last element of a given list. We apply `last` (5) to the list `[True,False]`, that is, we evaluate the term e := `last [True,False]`. We get the following reduction:

`app xs [x]` →{`xs` ↦ `y:ys`} `y : (app ys [x]))` →{`ys` ↦ `[]`} `[y,x]` (6)

`[y,x]` can be unified with `[True,False]` yielding e →{`xs`↦`[True]`,`x`↦`False`} `False`.

1.2 Point-free Style

The term *point-free* originates from topology where you have points in a space and functions that operate on these points. In functional programming spaces are types, functions are functions and points are the arguments of a function. In *point-free* style you do not explicitly access the points, that is, the arguments of a function. The idea of the *point-free* programming paradigm is to build functions by combining simpler ones. It was introduced by John Backus in his Turing Award Lecture in 1977 [3]. The counterpart of *point-free* is *point-wise*, that is, functions that explicitly access their arguments. Here, *point-free* programs are based on a couple of *point-wise* primitives.

2 Transformation to Point-free Style

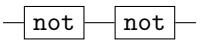
In this section we define a small set of point-wise operations which allow the definition of arbitrary functional logic operations in a point-free style.

Composition of Operations The first such "primitive" is *sequential composition*, occasionally simply referred to as "composition".

```
(*) :: (a -> b) -> (b -> c) -> a -> c
(f * g) x = g (f x)
```
(7)

The primitive `(*)` is a flipped version of `(.)`. Whereas `(f . g)` reads as "`f` after `g`", `(f * g)` is more like "`f` before `g`". This is more convenient with regard to our aim of a relation-algebraic treatment of programming semantics. Furthermore, the left-to-right reading provides a very descriptive graphical representation. The composition is visualised by connecting two operations with a line, indicating that the output of one is the input of the other. Such visualisations were also used in connecting functional programs [13] and allegory theory with hardware design [5] and to describe physical structures in general [15]. Simple definitions can be made point-free by using sequential composition, cf. Example (8).

```
involution x = not (not x)
involution = not * not
```
(8)



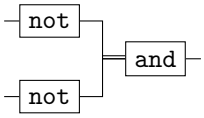
Operations with several arguments are composed by *parallel composition*.

```
(/) :: (a -> c) -> (b -> d) -> (a,b) -> (c,d)
(f / g) (x,y) = (f x,g y)
```
(9)

Example (10) illustrates the use of parallel composition. All primitive operators are right associative. Instead of using precedences we use parenthesises to increase readability.

```
nor :: Bool -> Bool -> Bool
nor x y = not x && not y

nor :: (Bool,Bool) -> Bool
nor = (not / not) * and
```
(10)



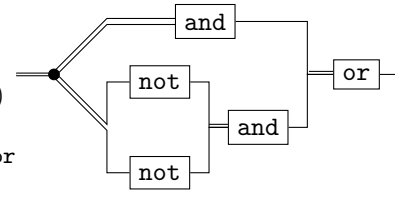
We have effectively changed the type of `nor` to a so called "uncurried" version. We use curried operations only when higher order is employed, as discussed in Paragraph "Higher Order".

Interface Adaption So far, we can express only right linear rules. Sharing arguments is the first of the primitives dealing with what we call "interface adaption". Interface adaption means that the connectives of two operations have to be copied, joined or reordered in some way. An uncurried and point-free version of the boolean operator "if and only if" (12) can be formulated using `(/)` and `fork`.

```
fork :: a -> (a,a)
fork x = (x,x)
```
(11)

```
(<=>) :: Bool -> Bool -> Bool
x <=> y = x && y || not x && not y

(<=>) :: (Bool,Bool) -> Bool
(<=>) = fork * (and/((not/not)*and)) * or
```
(12)



There are two more primitives for interface adaption. The operator `unit` to "discard a value" and the identity `id` to "pass a value on". Both are exemplified in the following sections.

```
unit :: _ -> ()
unit _ = ()
```
(13) 

```
id :: a -> a
id x = x
```
(14)

Data Structures, Inversion and Pattern Matching We do not wish to abstract from concrete domains at this point. In the semantics we would treat data structures in the standard way of sums and products. Here we define different operations to construct data. Each constructor of the original program will be assigned one operation.

```
nil :: () -> [a]
nil () = []
cons :: (a,[a]) -> [a]
cons (x,xs) = x : xs
```

```
true, false :: () -> Bool
true  () = True
false () = False
```
(15)

Note that these operations are again uncurried and that the constants `True`, `False`, and `[]` are extended with an argument. The reason for the latter extension will become apparent soon.

What we have seen so far is a more or less standard treatment of expressing functional programs in a point-free style. To concisely express pattern matching and to combine several rules we employ two additional features of functional *logic* programming, i.e., non-determinism and function patterns.

```
(?) :: a -> a -> a
x ? _ = x
_ ? y = y
```
(16)

```
coin :: () -> Bool
coin = true ? false
```
(17)

As stated in the introduction, overlapping rules in functional logic languages lead to non-deterministic search, cf. [11]. In principal, all non-determinism can be introduced by permitting only a single operation with overlapping rules `(?)` (16), cf. [1]. We use `(?)` to combine the rules of a function, cf. Example (17). Note that the introduction of the argument `()` for constant constructors extends to all definitions of constants.

```
invert :: (a -> b) -> (b -> a)
invert f = f' where f' (f x) = x
```
(18)

Function patterns can be used to invert arbitrary operations. This yields the primitive `invert` defined in (18). The semantics of function patterns are described in [2] in terms of a possibly infinite set of rewrite rules. We aim at giving a denotational semantics for function patterns for the first time.

The expressive power of function patterns can be estimated by considering that all other logic features can be obtained by using function patterns. E.g., `invert unit :: () -> a` yields a logic variable when applied to `()` and `invert fork :: (a,a) -> a` performs unification, cf. Section 1.1. Therefore we can define the following useful abbreviations for interface adaption.

```
unknown :: () -> a
unknown = invert unit
```
(19) 

```
join :: (a,a) -> a
join = invert fork
```
(20) 

There are two more functions useful abbreviations for interface adaption: `fst` and `snd`. Instead of using the standard definition we prefer to give a definition based on the primitives introduced so far:

```
fst :: (a,_) -> a
fst = (id / (unit * unknown)) * join
```
(21)

```
snd :: (_,a) -> a
snd = ((unit * unknown) / id) * join
```
(22)

Using `fst` and `snd` the standard functions `head` and `tail` can easily be expressed.

```
head :: [a] -> a                tail :: [a] -> [a]
head = invert cons * fst        tail = invert cons * snd
```
(23)

In addition to `(?)` to combine rules, the primitive `invert` can be used to express arbitrary pattern matching including function patterns. A constructor pattern is a linear constructor term, cf. Section 1.1. In order to match such a pattern we only have to invert the according constructors and then adapt the result like shown in (23). From this point of view it becomes apparent why constant constructors are extended with an argument: to make them invertible.

The expressive power gained by function patterns is paid with computational overhead [2]. It is thus desirable to replace a function pattern by an equivalent standard operation. We think that the semantics we want to base on the presented transformation will be helpful to develop according optimizing techniques.

There is one last feature concerning pattern matching in connection with laziness. If a value is discarded, e.g., by using `unit`, it is not evaluated. The semantics of pattern matching demands that matching is ensured regardless of whether the resulting variable bindings are used or not. The operations `head` and `tail` defined in (23) use one of the variables bound by the matching and therefore the pattern matching is indeed performed. In general we have to combine several of the primitives introduced so far to achieve the desired evaluation.

```
null :: [a] -> Bool
null []    = True
null (_:_) = False
```
(24)

Example (24) shows a case in which the bindings of the matching are discarded. The point-free version has to make sure that a) the empty tuple of `(invert nil)` and b) the pair resulting from `(invert cons)` are demanded, and not more. The following definition provides these properties.

```
null = invert nil * true ? invert cons * (unit/unit) * join * false
```
(25)

The astute reader might wonder why we introduce non-determinism for a perfectly deterministic operation like the pattern matching of `null`. The reason for this is twofold. 1) From a semantic point of view the non-deterministic branching does not matter. If the matching was indeed deterministic, for a given deterministic value all but one branch will finitely (even immediately) fail. 2) In a functional logic language patterns are *not* always deterministic nor treated in a sequential way (like in Haskell). Overlapping patterns induce non-determinism which is easily captured by our approach.

```
member :: [a] -> a
member (x:_)  = x
member (_:xs) = member xs
```
(26)

For example, the operation `member` defined in (26) non-deterministically relates a list with each of its elements. Without further additions this behaviour is captured by the transformation. The following definition shows a point-free version of `member`.

```
member = (invert cons * fst) ? (invert cons * snd * member)
```
(27)

Example (26) also illustrates that recursive functions simply stay recursive. There is no need for changes, e.g., a special recursion operator. Complex patterns are treated like complex expressions, i.e., they are composed with `(*)` and `(/)` before inverting the whole expression. We treat function patterns in the very same way. For example, the function `last` (5) is translated to:

```
last = invert ((id / ((id/nil) * cons))  * app) * snd * up
```
(28)

Higher Order In order to introduce higher-order operations we need to adapt the well known pair `apply` and `curry` to our setting. A first point to consider is

```
apply :: (a -> b,a) -> b
apply (f,x) = f x
```
(29)

that values of type `a` correspond to operations of type `() -> a`. Because higher-order operations should be first class objects we need to translate them in the same way. An operation of type `(a -> b)` must become an object of type `() -> (a -> b)` when used as an argument of an operation. If we assume this kind of translation we can define `apply` and `curry` straightforwardly.

```
curry :: (() -> (a,b) -> c) -> () -> (a -> b -> c)
curry f = \ () x y -> f () (x,y)
```
(30)

```
map :: (a -> b) -> [a] -> [b]
map _ []     = []
map f (x:xs) = f x : map f xs
```
(31)

The step to obtain the first curried version of a given function cannot be formulated in an equally general way because of call-time choice. This is illustrated by a standard example of a higher-order operation in Example (31). We can already translate `map` with the primitives introduced so far, adding `apply`.

```
map :: (a -> b,[a]) -> [b]
map = (invert (id / nil) * nil)
    ? (invert (id / cons) * adapt * (apply / map) * cons)
```
(32)

We assume `adapt` to map the tuple structure `(f,(x,xs))` to `((f,x),(f,xs))`. We omit its concrete definition by means of `id, unit, invert` and `fork`. We want to map the operation `not` on the list `[False,True]`.

```
not = (invert true * false) ? (invert false * true)
listFalseTrue = fork * (false / (fork * (true / nil) * cons)) * cons
mapNot = fork * (curryNot / listFalseTrue) * map
```
(33)

What should `curryNot :: () -> (Bool -> Bool)` be defined as? A first version might be `curryNot = const not`. But evaluating `mapNot ()` yields no solution. The reason is call-time choice. Because `f` is a variable the choice whether `f` is the operation "`invert true * false`" or "`invert false * true`" is made consistently for all applications of `f`. But this decision has to be made anew for each application of `f`. This can be achieved by η-expansion.

```
curryNot () x = not x
```
(34)

Using definition (34) `(mapNot ())` evaluates to `[True,False]` as intended. The example shows that a second version of each operation which will be applied higher order is needed.

We have illustrated all the point-wise primitives necessary to translate arbitrary Curry programs: `(*)` (7), `(/)` (9), `fork` (11), `unit` (13), `id` (14), `(?)` (16), `invert`(18), `apply` (29) and `curry` (30). In the final paper we will give a formal definition of the transformation and consider its soundness.

3 Related and Future Work

Cunha, Pinto and Proença [7, 6] present a framework for transformations of functional programs into point-free style. They implement a library for point-free programming in Haskell and transform Haskell programs into point-free programs which are based on this library. Conceptually, their approach first transforms a subset of Haskell to a simply-typed λ-calculus, and back to a Haskell program which represents a cartesian closed category. Because of the intermediate transformation to λ-calculus, the resulting programs bear only a remote resemblance to the original. In contrast, one of our aims is to keep the resulting programs close to the original. For example, we preserve the recursive structure of the program instead of expressing it by primitive recursion operators and we keep the data types and definitions of the original program instead of transforming them into generic sum and product types. Moreover, the framework of functional logic programming allows us to reduce the number of point-wise primitives on which the resulting point-free programs are based. For example we express the operation `snd` by using primitives while in a functional approach `snd` has to be a primitive itself. Furthermore we express all pattern matching by a single primitive, namely `invert`. Although we employ less primitives, we are able to transform a larger set of programs, proposing the first approach to transform functional logic programs to point-free style.

The book "Algebra of Programming" by Bird and de Moor [4] has been very influential for this work. They present a calculus for the algebraic manipulation of functional programs. We hope that we could give an idea that the framework of functional *logic* languages is an even more natural and promising field for this style of reasoning about programs. The elementary difference is the existence of non-determinism. Whereas in [4] every inversion and every non-deterministic definition resulting from inversion *must* be eliminated, the framework of functional logic languages allows much less restricted use of algebraic methods. The same is true a fortiori for approaches like [14] that aim on deriving a functional definition to compute the inversion of a given function definition.

Regarding the denotational semantics of functional (logic) languages, we want to relate our approach especially with two papers as future work. [16] proposes a denotational semantics for a functional language employing relation algebra. [9] provides a denotational semantics for functional logic languages based on cones. There are many interesting extensions to the framework of [9] which we want to investigate. However, our work presents a promising step towards covering function patterns for the first time.

References

1. S. Antoy. Evaluation strategies for functional logic programming. *Journal of Symbolic Computation*, 40(1):875–903, 2005.
2. S. Antoy and M. Hanus. Declarative programming with function patterns. In *Proceedings of the International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'05)*, pages 6–22. Springer LNCS 3901, 2005.
3. J. Backus. Can programming be liberated from the von neumann style?: a functional style and its algebra of programs. *Commun. ACM*, 21(8):613–641, August 1978.
4. R. Bird and O. de Moor. *Algebra of programming*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1997.
5. Carolyn Brown and Graham Hutton. Categories, Allegories, and Circuit Design. In *Proceedings of the 10th Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society Press, Los Alamitos, California, July 1994.
6. A. Cunha. *Point-free program calculation*. PhD thesis, Universidade do Minho, Departamento de Informática, 2005.
7. A. Cunha, J. Sousa Pinto, and J. Proença. A Framework for Point-free Program Transformation. In Andrew Butterfield, editor, *Revised Papers of the 17th International Workshop on Implementation and Application of Functional Languages (IFL'05)*, number 4015 in Lecture Notes in Computer Science. Springer-Verlag, 2005.
8. E. Giovannetti, G. Levi, C. Moiso, and C. Palamidessi. Kernel leaf: A logic plus functional language. *Journal of Computer and System Sciences*, 42(2):139–185, 1991.
9. J. C. González-Moreno, Maria Teresa Hortalá-González, Francisco Javier López-Fraguas, and Mario Rodríguez-Artalejo. An approach to declarative programming based on a rewriting logic. *J. Log. Program.*, 40(1):47–87, 1999.
10. M. Hanus. The integration of functions into logic programming: From theory to practice. *Journal of Logic Programming*, 19&20:583–628, 1994.
11. M. Hanus. A unified computation model for functional and logic programming. In *Proc. of the 24th ACM Symposium on Principles of Programming Languages (Paris)*, pages 80–93, 1997.
12. M. Hanus (ed.). Curry: An integrated functional logic language (vers. 0.8.2). Available at `http://www.informatik.uni-kiel.de/~curry`, 2006.
13. Geraint Jones and Mary Sheeran. Designing arithmetic circuits by refinement in Ruby. In R. S. Bird, C. C. Morgan, and J. C. P. Woodcock, editors, *Mathematics of Program Construction*, volume 669. Springer Verlag, 1993.
14. Shin-Cheng Mu. *A Calculational Approach to Program Inversion*. PhD thesis, Oxford University Computing Laboratory, 2003.
15. Hermann von Issendorff. Algebraic description of physical systems. In Roberto Moreno-Díaz, Bruno Buchberger, and José Luis Freire, editors, *EUROCAST*, volume 2178 of *Lecture Notes in Computer Science*, pages 110–124. Springer, 2001.
16. Hans Zierer. *Programmierung mit Funktionsobjekten: Konstruktive Erzeugung semantischer Bereiche und Anwendung auf die partielle Auswertung*. PhD thesis, Technische Universität München, Fakultät für Informatik, 1988.

Snapshot generation in a constructive object-oriented modeling language

Mauro Ferrari[1], Camillo Fiorentini[2], Alberto Momigliano[2] and Mario Ornaghi[2]

[1] Dipartimento di Informatica e Comunicazione, Università degli Studi dell'Insubria, Italy
mauro.ferrari@uninsubria.it
[2] Dipartimento di Scienze dell'Informazione, Università degli Studi di Milano, Italy
{fiorenti,momiglia,ornaghi}@dsi.unimi.it

Abstract. CooML is an object-oriented modeling language where specifications are theories in a constructive logic designed to handle incomplete information. In this logic we define snapshots as a formal counterpart of object populations, which are associated with specifications via the constructive interpretation of logical connectives. In this paper, we introduce the "snapshot semantics" of CooML and we describe a snapshot generation (SG) algorithm, which can be applied to validate specifications in the spirit of OCL-like constraints over UML models. Differently from the latter and from the standard BHK semantics, the logic allows us to exploit a notion of partial validation that is appropriate to encodings characterised by incomplete information. SG is akin to model generation in answer set programming. We show that the algorithm is sound and complete so that its successful termination implies consistency of the system.

1 Introduction

We are developing the constructive object-oriented modeling language CooML [19] (http://cooml.dsi.unimi.it), a specification language for OO systems. Similarly to UML/OCL [22], CooML provides a framework for the design of system specifications in the early stages of he development process. The language allows the user to distinguish between internally-defined elements and the problem domain (PD), which may involve loosely or incompletely defined components. This encourages the selection of the appropriate level of abstraction w.r.t. specifications.

CooML follows the spirit of lightweight formal methods [10]: it does not focus on full formalization, nor on whole system correctness, but emphasizes partiality in analysis and specification. In the context of OO modeling, both the *validation* of a specification and consistency checking can be achieved via the notion of *snapshot*, i.e. a population of objects in a given system state that satisfies the specification. Previous work has used snapshots for validation of UML/OCL models [8], as well as specifications in JML based on symbolic animation [4].

The novelty of CooML's approach resides in its semantics, which is related to the constructive explanation of logical connectives (a.k.a. the BHK interpretation [21]). Specifically, the truth of a CooML proposition in a given interpretation is explained by a mathematical object that we call an *information term*. For the time being, the latter can be visualized as a sort of *proof term* inhabiting a type/formula. The underlying logic

is characterized how classical and constructive information co-exists, the main "entry" point being the different way in which an *atomic* formula A is given evidence (for more details we refer the kind reader to the original formulation of the logic in [15]). If we call a *piece of information* the pair $I : P$, where P is a formula and I is its information term, then $I : P$ is a particular piece of information that may be *true* or *false* in a classical interpretation w, called a *world*. Thus, we have a notion of a *model* of a piece of information based on classical logic. In particular, we use $\mathtt{T}\{F\}$ to indicate the truth of F; in fact, $\mathtt{T}$ does not contain evidence for F, but it yields a piece of information true in all the models of F. This introduces a novel and flexible way to handle *incomplete* information, a notorious difficulty in other information systems such as relational databases.

Crucially, the constructive side of the logic allows the *identification of snapshots with information terms*, thus providing a formal counterpart to the intuitive notion of object populations. We argue that CooML's proof-theoretic snapshot generation may be advantageous w.r.t. a model-theoretic one, especially in cases where not all the information required to define a model is even present. The possibility of treating information in this less committed way means that we can select only the relevant information; this may have a cascade of benefits in terms of efficiency of the representation.

The contribution of this paper is twofold. First, we apply the semantics developed in purely logical terms in [15] to object oriented modeling languages. We model an OO system specification as a CooML theory T, the system snapshots as the pieces of information $I : T$, and the related information content as a suitable set of formulae. We show that the latter can be seen as the *minimum* information needed to give evidence to snapshots and that is related to snapshot consistency. Secondly, we describe (and implement) a snapshot generation algorithm (SGA), taking as inputs: (i) a CooML theory T, axiomatizing a set of classes in a problem domain PD; (ii) the user's generation requirements $\mathcal{G}$, which serve an analogous purpose to domain predicates in the grounding phase of ASP's [17]. As snapshots should be consistent with respect to PD and $\mathcal{G}$, we prove that consistency checking is sound and that SG is complete, i.e., if a consistent snapshot satisfying the generation requirements exists, it will be generated. This is not too faraway from adequacy results in the theory of CLP's [7].

2 CooML specifications

In this section we informally present the language via an example adapted from [3], while we defer the formal treatment to Section 2.1. The problem domain concerns a small coach company. Each coach has a specified number of seats and can be used for regular or private trips. In a regular trip, each passenger has its own ticket and seat number. In a private trip, the whole coach is rented and there may be a guide. The corresponding CooML specification is contained in the package `coachCompany` (Fig. 1). To explain our example we need to introduce CooML types system. We distinguish among *data types* (in our example, `Integer` and `Boolean`), *PD types* (`Person`), and *object types* (`Coach`, `Trip`, `Passenger`). They inherit from the top type `Value` the identity relation and the string representation. Data types are "statically" defined, i.e., their values do not depend on the current state. CooML assumes the existence of an implementation

that evaluates ground terms to values. A PD type extends `Value` with a set of problem domain functions.

```
package coachCompany;
pds{type Person;
  Integer numberOfSeats(Coach c) = (* the number of seats of c *);
  Boolean guides(Person p, Trip t) = (* p guides trip t *);
  Boolean nobooking(Passenger p, Trip t) = (* p has no booking in t *);
  Boolean vacant(Integer s, Coach c, Trip t) =
                                  (* s is a vacant seat on c in t *);
  Boolean booked(Passenger p, Integer s, Coach c, Trip t) =
                                  (* p has booked seat s on c in  t *);
  <constr name=bookingConstraints  language=prolog>
    false :- vacant(S,C,T), booked(_P,S,C,T).
    false :- booked(P1,S,C,T), booked(P2,S,C,T), not(P1==P2).
    false :- nobooking(P,T), booked(P,_Seat,_Coach,T).
  </constr>
  }
class Coach{
 coachPty: and{
     seats: exi{Integer seatsNr; seatsNr = numberOfSeats(this)}
     trips: for{Trip trip; trip is Trip(this) --> true} }
     Integer getSeats(){  return seats.seatNr  }
           }
class Trip{ env(Coach coach)
 TripPty: case{private: case{T{exi{Person p; guides(p,this)}}
                            T{not exi{Person p; guides(p,this)}}}
               regular: for{Integer seat; (seat in 1..coach.getSeats()) -->
                  case{vacant: vacant(seat,coach,this)
                       booked: exi{Passenger p; T{and{p is Passenger(this)
                                                      booked(p,seat,coach,this)}}
      }}}}}
class Passenger{ env(Trip trip)
 PsngrPty: case{c1: nobooking(this,trip)
                c2: exi{Integer seat, Coach coach;
                        T{and{trip is Trip(coach)
                              booked(this,seat,coach,trip)}}
      }}}
```

Fig. 1: The `coachCompany` package

Nothing is assumed about PD types; they may be characterized by a set of formal or informal *loose properties* that we call *PD constraints*, introduced by the tag `<constr>`.

The special subtype `Obj` of `Value` introduces object identities. Objects are created by CooML classes, which are structured in a single inheritance hierarchy rooted in `Obj`. The definition of a class *C* may depend on some *environment* parameters, namely `C(`$\underline{e}$`)` is a class with environment parameters $\underline{e}$. If $\underline{\mathbf{e}}$ is a ground instance of the environment parameters $\underline{e}$, then `C(`$\underline{\mathbf{e}}$`)` can be used to create new objects. We write "**o** `is C(`$\underline{\mathbf{e}}$`)`" to indicate that **o** has been created by `C(`$\underline{\mathbf{e}}$`)`, while "**o** `instanceof C(`$\underline{\mathbf{e}}$`)`" means that **o**

has environment $\underline{\mathbf{e}}$ and has been created by a subclass `C'` of `C`. We call those *class predicates*.

In a package: (i) data types are assumed to be externally implemented; (ii) PD types are defined in the `pds` (Problem Domain Specification) section; (iii) classes are introduced by suitable class declarations.

`pds` declaration and world states. The `pds` section specifies our general knowledge of the problem domain. It introduces PD types, functions and predicates using data and class types. In our example we introduce the PD type `Person` and functions `numberOfSeats`, `guides`, ...The informal descriptions `(*...*)` use terms of the global signature provided by the analysis phase [11]. A `<constr>` declaration introduces a set of PD constraints representing general problem domain properties that are not interpreted by CooML, but that could be interpreted by some external tool. In the example PD constraints are expressed in Prolog assisting the SG algorithm in filtering out undesired snapshots. The class predicate "`o is C(`$\underline{e}$`)`" is represented by the Prolog predicate `isOf(o, C, [`$\underline{e}$`])`, while "`o istanceOf C(`$\underline{e}$`)`" is translated into `instanceOf(o, C, [`$\underline{e}$`])`. The first constraint says that a coach seat cannot be vacant and booked at the same time, the second one excludes overbooking (a seat can be booked by at most one person), while the third says that the predicate `nobooking(P,T)` holds if person `P` has not booked a seat on the coach associated with trip `T`. In this paper, we assume that the signature Σ_T of a CooML theory T (including PD types, data types and classes) is first order and that we can represent the possible states of the "real world" by *reachable* Σ_T-interpretations, dubbed *world states*. Reachability means that each element of the interpretation domains is represented by some ground terms, in our case CooML values. In a world state, PD symbols are interpreted over the external world, data types are interpreted according to their implementation, and class predicates represent the current system objects. For instance the class predicates

```
mini is Coach(),  t1 is Trip(mini),  t2 is Trip(mini),  t3 is Trip(mini),
john is Passenger(t1)
```

represent a small company with a single mini-bus `mini`, three trips `t1`,`t2`,`t3` operated by `mini` and, so far, only one passenger `john` associated with trip `t1`.

`class` declarations and properties. A class declaration introduces the name C of the class, its (possible) environment parameters $\underline{e}$, its property $Pty_C(\texttt{this},\underline{e})$, and its methods [1]. An object $\mathbf{o}$ created by `C(`$\underline{\mathbf{e}}$`)` stores a *piece of information* structured according to $Pty_C(\mathbf{o},\underline{\mathbf{e}})$, and uses the methods implemented by `C(`$\underline{\mathbf{e}}$`)`.

For class properties, CooML uses a prefix syntax, where formulas may be labeled. Labels are used to refer to subformulae. For example, the label `seats` is used in the `getSeats` method to refer to `seatsNr`. A *class property* P is an atomic formula over Σ_T, or (recursively) a formula of the form `and`$\{P_1 \ldots P_n\}$, `case`$\{P_1 \ldots P_n\}$, `exi`$\{\underline{\tau}\ \underline{x}; P\}$, `for`$\{\underline{\tau}\ \underline{x}; G \rightarrow P\}$, `T`$\{P^{ext}\}$, where P^{ext} is a property that may also use negation `not` and implication `imp`. We stress that `not` and `imp` cannot be used outside `T`.

In CooML's semantics, a property P defines a set of possible pieces of information of the form $I : P$, where I is an *information term*, that is a structure justifying the truth of P. Each piece of information $I : P$ for P has an *information content*, a set of simple properties intuitively representing the minimum amount of information needed to justify P according to I. In fact, we call *simple property* an atomic formula of the form

[1] We use the self-reference `this` as in Java.

T{P^{ext}}. A simple property S represents a basic information unit, i.e., it has a unique information term tt where tt is a constant. This means that the only information we have is the *truth* of S, and that the associated information *content* is simply the set $\{S\}$. Exemplifying,

```
tt : t1 is Trip(mini)
```

has information content {t1 is Trip(mini)} and means that the trip t1 is assigned to the coach mini in the current world state.

The operator T may enclose a complex property P and indicates that we are interested only in its truth. Let us consider

```
tt: T{exi{Person p; guides(p,t2)}}    tt: T{not exi{Person p; guides(p,t3)}}
```

The first piece of information says that t2 is a guided trip without indicating who the guide is; the second one says that t3 has no guide.

By default[2] the truth of a simple property S in a world state w (w $\models S$) is defined as in classical logic, by ignoring T (i.e., w $\models$ T{P} iff w $\models P$}) and interpreting case as $\vee$, and as $\wedge$, not as $\neg$, imp as $\rightarrow$, exi as $\exists$ and for{$\tau\, \underline{x}$; $G(\underline{x})\rightarrow P(\underline{x})$} as $\forall \underline{x}(G(\underline{x}) \rightarrow P(\underline{x}))$.

In contrast, non-simple properties are interpreted constructively, by means of information terms. A piece of information $I : P$ may have one of the following forms:

Existential. (**x**, I) :exi{$\tau\, x$; $P(x)$}, where τ is the type of the existential variable x. The term **x** is a *witness* for x and the information content is the one of $I : P(\mathbf{x})$. For example,

```
(4,tt) : exi{Integer seatNr; seatNr = numberOfSeats(mini)}
```

has witness 4 and information content {4 = numberOfSeats(mini)}, signifying that our mini-bus has 4 passenger seats. Note that, differently from the case of simple properties, we know the value of x which makes $P(x)$ true.

Universal. ((**x**$_1$,I_1),...,(**x**$_n$,I_n)):for{$\tau\, x$; $G(x) \rightarrow P(x)$}, where $G(x)$ is an *x-generator*, i.e., a formula true for finitely many x[3]. The information content is the union of those of $I_1 : P(\mathbf{x}_1)$, ..., $I_n : P(\mathbf{x}_n)$ and of the *domain property* dom(x; $G(x)$; [**x**$_1$,...,**x**$_n$]), a special simple property interpreted as $\forall x(G(x) \leftrightarrow member(x,[\mathbf{x}_1,\ldots,\mathbf{x}_n]))$. For example, the information content of

```
((t1,tt),(t2,tt),(t3,tt)) : for{Trip trip; trip is Trip(mini) → true}
```

is {dom(trip; trip is Trip(mini); [t1,t2,t3])}, showing that the domain of the trip-generator "trip is Trip(mini)" is {t1,t2,t3}. Since the atomic formula true corresponds to no information, it can be ignored.

Conjunctive. ($I_1,\ldots,I_n$) : and{$P_1 \ldots P_n$}. The information content is the union of those of $I_j : P_j$, for all $j \in 1..n$. For instance, a piece of information for the class property coachPty(mini) and the related information content IC_1 are

```
((4,tt), ((t1,tt), (t2,tt), (t3,tt))) : and{seats(mini) trips(mini)}
IC1 = {4 = numberOfSeats(mini), dom(trip; trip is Trip(mini); [t1,t2,t3])}
```

[2] But one can change this. We do not discuss this issue here for lack of space.

[3] In this paper the precise syntax of generators is omitted.

Disjunctive. (k,I_k):case{$P_1 \ldots P_n$}. The selector $k \in 1..n$ points to the true subformula P_k and the information content is $I_k : P_k$'s. For example, if the object john with class predicate john is Passenger(t1) contains the information term (1,tt), then

(1,tt) : case{c1:nobooking(john,t1) c2: ...}

selects the first sub-property of PsngrPty, with information content {nobooking(john,t1)}, i.e. john has no booking in trip t1 in the current state.

The information content of classes. Let C($\underline{e}$) be a class with property Pty_C(this,$\underline{e}$). We associate with C the *class axiom*

clAx(C): for{Obj this, $\underline{\tau}$ $\underline{e}$; this is C($\underline{e}$) $\rightarrow$ Pty_C(this,$\underline{e}$)}

The corresponding pieces of information and information content are those for universal properties. The piece of information for class Coach and its information content IC_2 is:

((mini,CoachInfo)) : for{Obj this; this is Coach() $\rightarrow$ coachPty(this)}

IC_2 = {dom(this; this is Coach();[mini]), 4 = numberOfSeats(mini),
dom(trip; trip is Trip(mini); [t1,t2,t3])}

where CoachInfo:coachPty(mini) is defined as in the conjunctive case.

System snapshots and their information content. Let P be a package introducing a set of constraints $\mathcal{T}$ and the CooML classes $C_1, \ldots, C_n$. We associate with P a CooML theory $T_P = \langle \mathtt{thAx}, \mathcal{T} \rangle$, where thAx = and{clAx($C_1$) $\cdots$ clAx(C_n)}.

A piece of information I : thAx represents the information content of the whole system. We call it a *system snapshot*, to emphasise that the system may evolve through a sequence I_0 : thAx,...,I_n : thAx,.... A snapshot for our coachCompany system is of the form:

(I_1,I_2,I_3) : and{clAx(Coach) clAx(Passenger) clAx(Trip)}

and possible information terms I_1, I_2, I_3 are

I_1 = ((mini,CoachInfo)), I_2 = (([john,t1],(1,tt)),([ted,t2],(1,tt)))
I_3 = (([t1,mini],(2,((1,tt), (2,(john,tt)),(3,tt),(4,tt)))),
([t2,mini],(1,(1,tt))),
([t3,mini],(1,(2,tt))))

where [...] denote tuples. A relevant part of the information content for coachCompany is given in Fig. 2.

```
dom(o; o is Coach(); [mini]),  dom(o; o is Trip(mini); [t1,t2,t3]),
dom([o,t]; o is Passenger(t); [[john,t1],[ted,t2]]),
dom([o,c]; o is Trip(c); [[t1,mini],[t2,mini],[t3,mini]]),
4=numberOfSeats(mini),  nobooking(john,t1),  vacant(1,mini,t1),
booked(john,2,mini,t1), vacant(3,mini,t1),  vacant(4,mini,t1),
T{exi{Person p; guides(p,t2)}},  T{not exi{Person p; guides(p,t3)}}
```

Fig. 2: Part of the information content of coachCompany.

The above information content could be seen as an "incompletely specified" model of the `coachCompany` theory, where `numberOfSeats`, `nobooking`, `vacant`, `booked` and class predicates are completely specified, while for `guides` we have only some partial knowledge, expressed by the `T`-properties, and moreover nothing is said about `Person`. The relationship with classical models can be better explained by comparing the constructive and classical reading of CooML properties. Let $T = \langle \texttt{thAx}, \mathcal{T} \rangle$ be a CooML theory. We can switch to the classical interpretation of `thAx`simply by using the `T` operator, i.e., by considering the simple property $\texttt{T}\{\texttt{thAx}\}$. One can prove that $\texttt{T}\{\texttt{thAx}\}$ has a reachable model if and only if $\textsc{ic}(I : \texttt{thAx})$ has a reachable model, for at least one piece of information $I : \texttt{thAx}$. Furthermore, one can prove that $\textsc{ic}(I : \texttt{thAx})$ is the minimum set of simple formulas that justifies I as an explanation of `thAx`.

In this context we are mainly interested in the notion of consistency with respect to the PD constraints, assuming that the latter can be interpreted as first order sentences. In our example, we interpret a program clause $H : -B_1, \ldots, B_n$ as the universal closure of $B_1 \wedge \ldots \wedge B_n \rightarrow H$, as usual. A system snapshot $I : \texttt{thAx}$ for a theory $T = \langle \texttt{thAx}, \mathcal{T} \rangle$ is *consistent* if its information content $\textsc{ic}(I : \texttt{thAx})$ is true in a reachable classical model of $\mathcal{T}$; T is consistent if there is a consistent snapshot for it. For example, the above snapshot (I_1, I_2, I_3) is consistent with respect to the first and second constraint of the `pds` section, but not with the third, since both `nobooking(john,t1)` and `booked(john,2,mini,t1)` belong to the information content of Fig. 2.

2.1 Formal definitions

Let $T = \langle \texttt{thAx}, \mathcal{T} \rangle$ be a CooML theory and Σ_T the associated first order signature. The set of *information terms* for a property P ($\textsc{it}(P)$) is inductively defined as follows, where $\underline{\mathbf{x}}$ stands for values of $\underline{x}$:

$$
\begin{array}{lcl}
\textsc{it}(P) & = & \{\, \texttt{tt} \,\}, \text{ if } P \text{ is simple} \\
\textsc{it}(\texttt{and}\{P_1 \cdots P_n\}) & = & \{\, (I_1, \ldots, I_n) \mid I_j \in \textsc{it}(P_j) \text{ for all } j \in 1..n \,\} \\
\textsc{it}(\texttt{case}\{P_1 \cdots P_n\}) & = & \{\, (k, I) \mid 1 \leq k \leq n \text{ and } I \in \textsc{it}(P_k) \,\} \\
\textsc{it}(\texttt{exi}\{\underline{\tau}\ \underline{x}; P\}) & = & \{\, (\underline{\mathbf{x}}, I) \mid I \in \textsc{it}(P) \,\} \\
\textsc{it}(\texttt{for}\{\underline{\tau}\ \underline{x}; G(\underline{x}) \rightarrow P\}) & = & \{\, ((\underline{\mathbf{x}}_1, I_1), \ldots, (\underline{\mathbf{x}}_n, I_n)) \mid I_j \in \textsc{it}(P) \text{ for all } j \in 1..n \,\}
\end{array}
$$

A *piece of information* for a ground property P is a pair $I : P$, with $I \in \textsc{it}(P)$. A *collection* is a set of ground simple properties. The *information content* $\textsc{ic}(I : P)$ is the collection inductively defined as follows:

$$
\begin{array}{lcl}
\textsc{ic}(\texttt{tt} : P) & = & \{P\}, \text{ where } P \text{ is simple} \\
\textsc{ic}(\,(I_1, \ldots, I_n) : \texttt{and}\{P_1 \cdots P_n\}\,) & = & \bigcup_{j=1}^{n} \textsc{ic}(I_j : P_j) \\
\textsc{ic}(\,(k, I) : \texttt{case}\{P_1 \ldots P_n\}\,) & = & \textsc{ic}(I : P_k) \\
\textsc{ic}(\,(\underline{\mathbf{x}}, I) : \texttt{exi}\{\underline{\tau}\ \underline{x}; P(\underline{x})\}\,) & = & \textsc{ic}(I : P(\underline{\mathbf{x}})) \\
\textsc{ic}(\,((\underline{\mathbf{x}}_1, I_1), \ldots, (\underline{\mathbf{x}}_n, I_n)) : \texttt{for}\{\underline{\tau}\ \underline{x}; G(\underline{x}) \rightarrow P(\underline{x})\}\,) & = & \bigcup_{j=1}^{n} \textsc{ic}(I_j : P(\underline{\mathbf{x}}_j)) \\
 & & \cup\, \{\, \texttt{dom}(\underline{x}; G(\underline{x}); [\underline{\mathbf{x}}_1, \ldots, \underline{\mathbf{x}}_n]) \,\}
\end{array}
$$

The information content $\textsc{ic}(I : P)$ represents the minimum amount of information needed to get evidence for P according to I. We say that a collection $\mathcal{C}$ *gives evidence*

to $I : P$, and we write $\mathcal{C} \rhd I : P$, iff one of the following clauses holds:

$$
\begin{array}{ll}
\mathcal{C} \rhd \mathtt{tt} : P & \text{iff } P \in \mathcal{C} \\
\mathcal{C} \rhd (I_1, \ldots, I_n) : \mathtt{and}\{P_1 \cdots P_n\} & \text{iff } \mathcal{C} \rhd I_j : P_j \textit{ for all } j \in 1..n \\
\mathcal{C} \rhd (k, I) : \mathtt{case}\{P_1 \ldots P_n\} & \text{iff } \mathcal{C} \rhd I : P_k \\
\mathcal{C} \rhd (\underline{\mathbf{x}}, I) : \mathtt{exi}\{\underline{\tau}\ \underline{x}; P(\underline{x})\} & \text{iff } \mathcal{C} \rhd I : P(\underline{\mathbf{x}}) \\
\mathcal{C} \rhd ((\underline{\mathbf{x}}_1, I_1), \ldots, (\underline{\mathbf{x}}_n, I_n)) : \mathtt{for}\{\underline{\tau}\ \underline{x}; G(\underline{x}) \rightarrow P(\underline{x})\} & \text{iff } \mathtt{dom}(\underline{x}; G(\underline{x}); [\underline{\mathbf{x}}_1, \ldots, \underline{\mathbf{x}}_n]) \in \mathcal{C} \\
 & \quad \textit{and } \mathcal{C} \rhd I_j : P(\underline{\mathbf{x}}_j) \textit{ for all } j \in 1..n
\end{array}
$$

The information content $\mathrm{IC}(I : P)$ represents an information about the current world state. We define the information content of $\mathcal{C}$ as its closure under (classical) logical consequence, for $\mathcal{C}^* = \{P \mid \mathcal{C} \models P\}$. We say that $\mathcal{C}_1$ *contains less information* than $\mathcal{C}_2$ (written $\mathcal{C}_1 \sqsubseteq \mathcal{C}_2$) iff $\mathcal{C}_1^* \subseteq \mathcal{C}_2^*$. Intuitively, the definition of $\sqsubseteq$ is justified by the fact that an user will "trust" $\mathcal{C}^*$, whenever he trusts $\mathcal{C}$. We could use a different trust-relation, considering different logics. We only need the following to hold:

(1). $\mathcal{C} \subseteq \mathcal{C}^*$;
(2). $\mathcal{C}_1 \subseteq \mathcal{C}_2^*$ implies $\mathcal{C}_1 \sqsubseteq \mathcal{C}_2$.

Using the above properties, we can establishes the minimality of $\mathrm{IC}(I : P)$ with respect to $\sqsubseteq$:

Theorem 1. *Let* $I : P$ *be a piece of information:*

1. $\mathrm{IC}(I : P) \rhd I : P$
2. *For every collection* $\mathcal{C}$, $\mathcal{C} \rhd I : P$ *implies* $\mathrm{IC}(I : P) \sqsubseteq \mathcal{C}$.

Now we can apply the above discussion to the problem of checking snapshots against constraints. Let $T = \langle \mathtt{thAx}, \mathcal{T} \rangle$ be a CooML theory. We recall that a *snapshot* for T is a piece of information $I : \mathtt{thAx}$. We introduce the following notions of consistency for snapshots.

- A snapshot $I : \mathtt{thAx}$ is consistent with respect to the constraints $\mathcal{T}$ ($\mathcal{T}$*-consistent*) iff there exists a reachable model of $\mathrm{IC}(I : \mathtt{thAx}) \cup \mathcal{T}$.
- T is *snapshot-consistent* iff there is at least one snapshot $I : \mathtt{thAx}$ such that $I : \mathtt{thAx}$ is $\mathcal{T}$-consistent.

The latter definition is related to classical consistency by the following result:

Theorem 2. *Let* $T = \langle \mathtt{thAx}, \mathcal{T} \rangle$ *be a CooML theory.* T *is snapshot-consistent iff there is a reachable model of* $\mathrm{T}\{\mathtt{thAx}\} \cup \mathcal{T}$.

3 A snapshots generation algorithm and its theory

A snapshot generation algorithm (SGA) for a CooML theory $T = \langle \mathtt{thAx}, \mathcal{T} \rangle$ takes as input the user's *generation requirements* and tries to produce $\mathcal{T}$-consistent snapshots that satisfy such requirements. Roughly, *generation states* represent incomplete snapshots, e.g. in logic programming parlance, partially instantiated terms; inconsistent attempts are pruned, when recognized as such during generation.

Consistency checking plays a central role. It depends on the PD logic and it is discussed next. In Subsection 3.2 we illustrate the use of snapshot generation for validating CooML specifications. Finally, in Subsection 3.3 we briefly outline a non deterministic algorithm based on which one may develop sound and complete implementations.

3.1 Consistency check

To recognize inconsistent attempts, *SGA* uses an internal representation of the information content of the current generation state S, denoted by INFO_S.

Here we briefly discuss a simplified version of consistency check in our Prolog implementation, called SnaC. Let P_S be the internal Prolog translation of the information content INFO_S. For this simplified version, we assume that P_S is executed by a suitable *meta-interpreter*. Without giving the formal details, we notice that INFO_S consists of ground facts, clauses of the form H `:- eq(t1,t2)` or `false :-` B, where:

- We use `eq` to avoid Prolog's standard unification interfering with Skolem constants. Indeed, the latter represent unknown values originating from the translation of `T{exi{...}}`, where different constants may represent the same value. In the simplified version, the `eq` atoms are just residuated by the meta-interpreter in a list of "unsolved equations".
- The reserved atom `false` is introduced to detect inconsistency: its finite failure signals snapshot consistency, conversely, its success corresponds to inconsistency.

Clauses with head `false` are called *integrity constraints* and `false` may occur only as such. A SnaC representation P_S has the following property: if the meta-interpretation of a goal G succeeds from P_S with answer σ and a list L of unsolved equations, then $G\sigma$ is a logical consequence of $P_S \cup L$. Furthermore, consistency is preserved and the models of P_S are models of INFO_S (in the declarative reading of P_S, we interpret `eq` as equality and `false` as falsehood). As an example, let us consider the SnaC representation P_{cComp} in Fig. 3 of the information content of the `coachCompany` package (Fig. 2).

```
isOf(mini,'Coach',[]).  false :- isOf(O,'Coach',[]), not(member(O,[mini]).
isOf(john 'Passenger',[t1]). isOf(ted 'Passenger',[t2]). ...
numberOfSeats(mini,4). nobooking(john,t1). booked(john,2,mini,t1).
vacant(1,mini,t1). vacant(3,mini,t1). vacant(4,mini,t1).
guides(P,t2):- eq(P,p0).
false :- guides(P,t3).
```

Fig. 3: The SnaC representation P_{cComp}.

The facts and the constraint in the first lines come from the translation of domain properties. For example, the first row contains the translation of `dom{o; o is Coach(); [mini])}`. The other facts come from the translation of atoms. The clause `guides(P,t2):- eq(P,p0)` is the translation `T{exi{Person p; guides(p,t2)}}`, where `p0` is a fresh Skolem constant. Finally, `false :- guides(P,t3)` is the translation of `T{not exi{Person p; guides(p,t3)}}`.

Let us analyse the three possible outcomes of consistency check starting from the example in Fig. 3:

(a) `false` finitely fails for the program P_{cComp}. This entails that `false` does not belong to the minimum model $\mathcal{M}$ of $P_{cComp} \cup \{$`eq(X,X)`$\}$. The latter contains all the ground atoms in Fig. 3 as well as `guides(p0,t2)`. Since $\mathcal{M}$ is a model of P_{cComp}, it is also a model of the information content of the `coachCompany` package thanks to the properties of the translation.

(b) If we add the constraint

```
c1)   false :- nobooking(P,T), booked(P,_S,_C,T).
```

to P_{cComp}, now the goal `false` succeeds from program $P_{cComp} \cup \{\texttt{c1}\}$, residuating the empty list. This implies that the snapshot corresponding to the information content of `coachCompany` is inconsistent w.r.t. `c1`.

(c) If we add the constraint

```
c2)   false :- guides(P,T), isOf(P,'Passenger',[T]).
```

the goal `false` succeeds from program $P_{cComp} \cup \{\texttt{c2}\}$, residuating `[eq(ted,p0)]`. This implies that `false` belongs to the minimum model $\mathcal{M}$ of $P_{cComp} \cup \{\texttt{c2}, \texttt{eq(ted,p0)}\}$. The equality `eq(ted,p0)` is returned to the user as a source of inconsistency.

The above discussion is reflected in the following theorem:

Theorem 3. *Let $T = \langle \texttt{thAx}, \mathcal{T} \rangle$ be a CooML theory, I : `thAx` a snapshot and P a program containing the translation of* IC$(I : \texttt{thAx})$ *and the PD constraints $\mathcal{T}$.*

1. *If `false` finitely fails from P, then I : `thAx` is $\mathcal{T}$-consistent.*
2. *If `false` succeeds from P residuating a set of constraints $\mathcal{U}$, then I : `thAx` is inconsistent with respect to $\mathcal{T} \cup \mathcal{U}$.*

In the first case, SnaC accepts I : `thAx` as a $\mathcal{T}$-consistent snapshot. In the second, $\mathcal{U}$ being empty signals inconsistency. If $\mathcal{U}$ is not empty, it is returned as an answer. We omit the proof, since it is somewhat implicit in the above discussion.

A more general result can be obtained admitting a larger class of simple properties and PD constraints, using techniques similar to those used in CLP, as *constraint systems* [7]. Roughly, we can consider $\mathcal{T}$ as a program of a CLP system using as calculus an extension of the standard logic programming operational semantics, where the constraint system is the Herbrand universe under CET, modified to deal with Skolem constants.

3.2 Validating specifications via the SGA

One of the purposes of snapshot generation is understanding and validating a CooML specification. To this aim, the user can specify suitable *generation requirements* in order to reduce the number of generated examples to a manageable size and show only the aspects he is interested in. We explain the language of generation requirements and its semantics through our example. It may be helpful to keep in mind the analogy with how an *answer set* program is constructed by grounding.

In the implementation, the number of generated snapshots can be limited by means of the the special atom `choice(`A`)`. This plays the role of *domain* predicates in ASP. The SG algorithm will instantiate A according to its axiomatisation. For example:

```
choice(isOf(C,'Coach',[])) :- member(C,[c1,c2]).
choice(isOf(P,'Passenger',[T])) :- member(P,[anna,john,ted]).
choice(isOf(T,'Trip',[C])) :- member((T,C), [(t1,c1),(t2,c2),(t3,c1)]).
choice(numberOfSeats(c1,3)).
choice(numberOfSeats(c2,60)).
```

instructs SG to generate one coach `c1` with 3 seats and possible trips `t1`, `t3`, and another `c2` with 60 seats and trip `t2`. The declarative meaning of `choice` is given by the axiom schema $A \rightarrow \texttt{choice}(A)$, which, together with the user definition of `choice`, sets up the generation requirements. The generated snapshots will satisfy the PD constraints, as well as the generation requirements.

Once the SG algorithm loads a CooML theory and the user generation requirements, it can be queried with *generation goals* (*G-goals*). A sample *G*-goal is:

```
(g1)  [ [3,tt], Trips ] : isOf(C,'Coach',[]).
```

Since `[3,tt]:seats(C)` has information content `3 = numberOfSeats(C)`, the query looks for the information `Trips:trips(C)` for every coach `C` with 3 seats. More precisely, the *G*-goal includes both a generation goal ("generate all the coaches `C` with 3 seats that satisfy the generation requirements") and a query ("for each `C`, show the information on the trips assigned to it"). An answer to `g1` is:

`Trips = [ [t1,tt] ]` and `C = c1`

with information content

`isOf(c1,'Coach',[]),isOf(t1, 'Trip', [c1])`

The rest of the snapshot, including information terms for all classes in the package, is omitted for the sake of space. If the user asks for more solutions, all possible snapshots will be shown. In the above example, there are two more solutions, where `c1` has two assigned trip or none.

We now sketch some ways in which the SG can be used in the process of system specification and development. This will be the focus of future work.

Validating specifications The goal is to show that a CooML theory "correctly" models the problem domain. Validation is empirical by nature: it relates the theory to the modeled world. The idea is to generate models that satisfy given generation requirements and check whether they match the user expectations. To this aim, it is useful to tune the generation requirements to separately consider various aspects that can be understood within a small, "human viable" number of examples, as usual in this context [8]. For instance, we may concentrate on the validation of the booking part of the `CoachCompany` package. In particular, we can find some supporting evidence of the correctness of the specification in a match between the expected and actual number of snapshots, where parameters of the latter are chosen as small as possible, while preserving meaningfulness. Naturally, snapshots can be used as inputs to tools for automatic, specification-based testing generation, in the spirit of [18].

Partial and full model checking As traditional in software model checking, here the goal is to show that, under the assumption of the generation requirements, no snapshot satisfies an undesired property. This is obtained if the SGA finds a snapshot-inconsistency, i.e., it halts without exhibiting any snapshot. Equivalently, one can prove that every snapshot satisfies a given property by showing that its negation is snapshot-inconsistent. We call this approach *partial* model checking, because in general snapshot consistency may depend on the selection of generation requirements. We may perform full model checking if the set of generated snapshots is representative of all models of the theory w.r.t. the property under consideration.

3.3 A schematic algorithm

We now describe a general schema for the Snapshot Generation Algorithm, of which SnaC is just a first rough implementation. Let $T = \langle \mathtt{thAx}, \mathcal{T} \rangle$ be a CooML theory, where $\mathtt{thAx} = \mathtt{and}\{\mathtt{clAx}(C_1), \ldots, \mathtt{clAx}(C_n)\}$. Its information terms are represented by sets of G-goals that we call *populations*. The generation process starts from a set P_0 of G-goals to be solved, i.e. to become grounded. The SGA gradually instantiates P_0, possibly generating new G-goals. It divides the population in two separate sets: TODO, containing the G-goals not solved yet and DONE, containing the solved ones. A *generation state* has the form $S = \langle \text{DONE}, \text{TODO}, \text{CLOSED}, \text{INFO} \rangle$, where:

- CLOSED is a set of predicates $closed(C, \underline{e})$. Such a predicate is added to CLOSED when all the objects with creation class $C(\underline{e})$ have been generated. It prevents the creation of new objects of class $C(\underline{e})$ in subsequent steps.
- INFO is the representation in the PD language of the information content of DONE, i.e., for every $I : \mathtt{isOf}(o, C, [\underline{e}]) \in \text{DONE}$, $\text{IC}(I : Pty_C(o, \underline{e})) \subseteq \text{INFO}$.

The following definitions are in order:

- A state S is *in solved form* if $\text{TODO} = \emptyset$.
- $\text{Dom}(S) = \{ \mathtt{isOf}(o, C, [\underline{e}]) \mid I : \mathtt{isOf}(o, C, [\underline{e}]) \in \text{DONE} \cup \text{TODO} \}$.
- $\langle \text{DONE}_1, \text{TODO}_1, \text{CLOSED}_1, \text{INFO}_1 \rangle \preceq \langle \text{DONE}_2, \text{TODO}_2, \text{CLOSED}_2, \text{INFO}_2 \rangle$ iff
  1. $\text{DONE}_1 \subseteq \text{DONE}_2$, $\text{Dom}(S_1) \subseteq \text{Dom}(S_2)$ and $\text{INFO}_1 \subseteq \text{INFO}_2$;
  2. If $closed(C, \underline{e}) \in \text{CLOSED}_1$, then $\mathtt{isOf}(o, C, [\underline{e}]) \in \text{Dom}(S_1)$ iff $\mathtt{isOf}(o, C, [\underline{e}]) \in \text{Dom}(S_2)$.

The SGA starts from initial state $S_0 = \langle \emptyset, \text{TODO}_0, \emptyset, \emptyset \rangle$ and yields a *solution* $S = \langle \text{DONE}, \emptyset, \text{CLOSED}, \text{INFO} \rangle$ such that $S_0 \preceq S$; since $\text{TODO} = \emptyset$, for every $I : \mathtt{isOf}(o, C, [\underline{e}]) \in \text{TODO}_0$, DONE contains a ground information term $(I : \mathtt{isOf}(o, C, [\underline{e}]))\sigma$ solving it. The algorithm computes a solution of S_0 that is minimal with respect to $\preceq$ through a sequence of *expansion steps*. The latter are triples $\langle S, I : \mathtt{isOf}(o, C, [\underline{e}]), S' \rangle$ such that:

p1. $I : \mathtt{isOf}(o, C, [\underline{e}]) \in \text{TODO}$ (the selected goal);
p2. $(I : \mathtt{isOf}(o, C, [\underline{e}]))\sigma \in \text{DONE}'$ and $I : \mathtt{isOf}(o, C, [\underline{e}]) \notin \text{TODO}'$ (it has been solved);
p3. $S \prec S'$ and, for every S^* in solved form, $S \prec S^* \preceq S'$ entails $S^* = S'$ (no solution is ignored).

The high-level code for a non deterministic SGA based on expansion steps is listed in Fig.4, where TODO_0 are the G-goals to be solved under theory $\langle \mathtt{thAx}, \mathcal{T} \rangle$ and generation requirements $\mathcal{G}$. The variable UC stores the "unsolved constraints" generated by the "error tests" $\mathtt{error}(S)$ and $\mathtt{globalError}(S)$. They check consistency against "local" and "global" integrity constraints. `error` must be *monotonic*, i.e., $\mathtt{error}(S)$ and $S \preceq S'$ entails $\mathtt{error}(S')$; `globalError` applies only to states in solved form.

The SG is a general schema, whose core is the implementation of expansion steps and error predicates. The latter use the integrity constraints `false :-` B to detect inconsistency and are based on a generalization of the ideas presented in Section 3.1.

SG ($\langle \texttt{thAx}, \mathcal{T} \rangle$, $\mathcal{G}$, $ToDo_0$)

1 $Thy = \texttt{thAx}$; $PDAx = \mathcal{T} \cup \mathcal{G}$; $S = \langle \emptyset, ToDo_0, \emptyset, \emptyset \rangle$; $UC = \emptyset$;
2 **while** $ToDo \neq \emptyset$ **do**
3 **if** `error`(S) **fail**;
4 **else** % *Generation Step:*
5 Choose $I : \texttt{isOf}(o, C, [\underline{e}]) \in ToDo$ and compute $\langle S,\ I : \texttt{isOf}(o, C, [\underline{e}]),\ S' \rangle$;
6 $S = S'$;
7 **if** `globalError`(S) **fail**;
8 **else return** S, UC

Fig. 4: The SG Algorithm.

A call to an error predicate either returns "true" when an inconsistency is detected, or updates UC and returns "false". When a state S in solved form is reached, SG returns UC as an answer; if it is empty, S is consistent. The current implementation could be improved, namely in detecting more than the trivial inconsistencies; no simplification is supported.

To state the adequacy results, we introduce some additional notation (ITP) in order to associate a class C_j and population P with their information terms:

$$\begin{aligned} \text{ITP}(P, C_j) &= [\ [[o_{j_1}, \underline{e}_{j_1}], I_{j_1}]\ , \ \ldots\ , \ [[o_{j_k}, \underline{e}_{j_k}], I_{j_k}]\ ] \\ \text{ITP}(P) &= [\ \text{ITP}(P, C_1)\ , \ \ldots\ , \ \text{ITP}(P, C_n)\ ] \end{aligned}$$

where, $I_{j_1} : \texttt{isOf}(o_{j_1}, C_j, [\underline{e}_{j_1}]), \ldots, I_{j_k} : \texttt{isOf}(o_{j_k}, C_j, [\underline{e}_{j_k}])$ are the G-goals of P with class C_j ($1 \leq j \leq n$); if no G-goal with class C_j belongs to P, then $\text{ITP}(P, C_j)$ is the empty list.

Theorem 4 (Correctness). *Let $S^* = \langle \text{DONE}^*, \emptyset, \text{CLOSED}^*, \text{INFO}^* \rangle$ be a state computed by SG with theory $T = \langle thAx, \mathcal{T} \rangle$ and generation requirements $\mathcal{G}$, and let $I^* = \text{ITP}(\text{DONE}^*)$ be the information term of the population DONE^*. Then, either UC is empty and $I^* : thAx$ is $\mathcal{G} \cup \mathcal{T}$-consistent, or $I^* : thAx$ is inconsistent with respect to $\mathcal{G} \cup \mathcal{T} \cup UC$.*

The proof easily follows assuming that for every state S: (i) INFO_S satisfies $\mathcal{G}$, by the way SG performs grounding; (ii) when `error`(S) or `globalError`(S) returns "true", then INFO_S is inconsistent with respect to $\mathcal{T}$; (iii) when `globalError`(S) returns "false", then either UC is empty and $\text{INFO}_S \cup \mathcal{T}$ is consistent, or $\text{INFO}_S \cup \mathcal{T} \cup UC$ is inconsistent.

Theorem 5 (Completeness). *Let $S_0 = \langle \emptyset, \text{TODO}_0, \emptyset, \emptyset \rangle$ be an initial state of SG with theory T and generation requirements $\mathcal{G}$. If there is a state $S = \langle \text{DONE}, \emptyset, \text{CLOSED}, \text{INFO} \rangle$ such that $S_0 \preceq S$, then SGA reaches a state S^* in solved form such that $S_0 \preceq S^* \preceq S$.*

The proof of Theorem 5 follows from the above properties p1, p2, p3.

4 Related work and conclusion

We have presented the semantics of the object-oriented modeling language CooML, a language in the spirit of UML, but based on a constructive semantics, in particular the BHK explanation of logical correctives. We have introduced a proof-theoretic notion of snapshot based on populations of objects and information terms, from which snapshot generation algorithms can be designed. More technically, we have introduced generation goals and the notion of minimal solution of such goals in the setting of a CooML specification, and we have outlined a non-deterministic generation algorithm, showing how finite minimal solutions can, in principle, be generated. We use a constraint language in order to specify the general properties of the problem domain, as well as the generation requirements. In an implementation of the SGA, a consistency checking algorithm is assumed, which either establishes the (in)consistency of the current snapshot, or residuates a set of unsolved constraints.

The relevance of SG for validation and testing in OO software development is widely acknowledged. The USE tool [8] for validation of UML/OCL models has been recently extended with a SG mechanism; differently from us, this is achieved via a procedural language. Other animation tools [4] are based on JML specification. In [2] the specification of features models are translated into SAT problems; tentative solutions are then propagated with a Truth Maintenance System. If a inconsistency is discovered the TMS explains the causes in view of possible model repair. Related work includes also [16], where design space specs are seen as trees whose nodes are constrained by OCL statements and BDD's used to find solutions.

Snapshot generation is only one of CooML's aspects, once we put our software engineering glasses on and see it more generally as a *specification* rather than modeling language [9,12]. In this paper we have not considered *methods*, although the underlying logic supports a clean notion of (correct) *query* methods, namely methods that do not update the system state, but extract pieces of information from it. The existence of a method M answering P (i.e., computing $I : P$) is guaranteed when P is a constructive logical consequence of `thAx`. Moreover, M can be extracted from a constructive proof of P. The implementation of query and update methods is a crucial part of future work.

We plan to improve and extend the snapshot generation algorithm. There are two directions that we can pursue; first, we can fully embrace CLP as a PD logic, strengthening the connection that we have only scratched in Section 3.1. In the current prototype there is little emphasis on the simplification of unsolved constraints. This could be partially ameliorated by adopting CLP, in particular over finite domains. More in general, it is desirable to relate Theorem 3 with the notion of satisfaction-completeness in constraint systems [7]. Another direction comes from the relation between CooML's approach to incomplete information and answer set programming [1, 17], in particular disjunctive LP [13]. A naive extension of the SGA to this case would yield inefficient solutions, yet the literature offers several ways constraints and ASP may interact [5,14]. We may explore the possibility of combining snapshot generation with SAT provers, to which we may pass ground unsolved constraints in order to check global consistency. Finally we intend to explore the more general issue of the relationships between information terms and stable models, in particular partial stable models [20] in the context of partial logics [6].

References

1. C. Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. CUP, 2003.
2. D. S. Batory. Feature models, grammars, and propositional formulas. In J. H. Obbink and K. Pohl, editors, *SPLC*, volume 3714 of *LNCS*, pages 7–20. Springer, 2005.
3. A. Boronat, J. Oriente, A. Gómez, I. Ramos, and J. A. Carsí. An algebraic specification of generic OCL queries within the Eclipse modeling framework. In A. Rensink and J. Warmer, editors, *ECMDA-FA*, volume 4066 of *LNCS*, pages 316–330. Springer, 2006.
4. F. Bouquet, F. Dadeau, B. Legeard, and M. Utting. JML-testing-tools: A symbolic animator for JML specifications using CLP. In N. Halbwachs and L. D. Zuck, editors, *TACAS*, volume 3440 of *Lecture Notes in Computer Science*, pages 551–556. Springer, 2005.
5. F. Buccafurri, N. Leone, and P. Rullo. Strong and weak constraints in disjunctive datalog. In J. Dix, U. Furbach, and A. Nerode, editors, *LPNMR*, volume 1265 of *Lecture Notes in Computer Science*, pages 2–17. Springer, 1997.
6. M. Fitting. Partial models and logic programming. *Theor. Comput. Sci.*, 48(3):229–255, 1986.
7. T. Fruewirth and S. Abdennadher. *Essentials of Constraint Programming*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.
8. M. Gogolla, J. Bohling, and M. Richters. Validating UML and OCL models in USE by automatic snapshot generation. *Software and System Modeling*, 4(4):386–398, 2005.
9. J. V. Guttag and J. J. Horning. *Larch: languages and tools for formal specification*. Springer-Verlag New York, Inc., New York, NY, USA, 1993.
10. D. Jackson and J. Wing. Lightweight formal method. *IEEE Computer*, April 1996.
11. C. Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*. Prentice Hall, Upper Saddle River, NJ, 2004.
12. G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: a behavioral interface specification language for Java. *SIGSOFT Softw. Eng. Notes*, 31(3):1–38, 2006.
13. N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. The DLV system for knowledge representation and reasoning. *ACM Trans. Comput. Log.*, 7(3):499–562, 2006.
14. V. W. Marek, I. Niemelä, and M. Truszczynski. Logic programs with monotone cardinality atoms. In V. Lifschitz and al., editors, *LPNMR*, volume 2923 of *LNCS*, pages 154–166. Springer, 2004.
15. P. Miglioli, U. Moscato, M. Ornaghi, and G. Usberti. A constructivism based on classical truth. *Notre Dame Journal of Formal Logic*, 30(1):67–90, 1989.
16. S. Neema, J. Sztipanovits, G. Karsai, and K. Butts. Constraint-based design-space exploration and model synthesis. In R. Alur and I. Lee, editors, *EMSOFT*, volume 2855 of *Lecture Notes in Computer Science*, pages 290–305. Springer, 2003.
17. I. Niemelä and P. Simons. Smodels - an implementation of the stable model and well-founded semantics for normal LP. In *LPNMR*, pages 421–430, 1997.
18. J. Offutt and A. Abdurazik. Generating tests from UML specifications. In R. France and B. Rumpe, editors, *Proc. of UML'99*, volume 1723 of *LNCS*, pages 416–429. Springer, 1999.
19. M. Ornaghi, M. Benini, M. Ferrari, C. Fiorentini, and A. Momigliano. A constructive object oriented modeling language for information systems. *ENTCS*, 153(1):67–90, 2006.
20. T. C. Przymusinski. Well-founded and stationary models of logic programs. *Ann. Math. Artif. Intell.*, 12(3-4):141–187, 1994.
21. A. S. Troelstra. From constructivism to computer science. *TCS*, 211(1-2):233–252, 1999.
22. J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modelling with UML*. Object Technology Series. Addison-Wesley, Reading/MA, 1999.

Symbolic Generation of Optimal Control Policies for Discrete-Time Systems

Michel Sintzoff

Department of Computing Science and Engineering
University of Louvain

Extended Abstract - August 6, 2007

Abstract. We present a symbolic, logic-based technique generating optimal control policies for discrete-time systems where state-spaces are not necessarily finite. These systems are symbolically represented by finite sets of actions, which are guarded assignments with costs. The control objective is to minimize costs of exexutions reaching a given target set. Control policies are represented by tuples of action guards. Optimal control policies are generated by a symbolic iteration which is based on backwards induction and takes the control objective into account.

1 Introduction

An algorithm over states is "state-based" - or enumerative - if it handles states individually. It is "symbolic" otherwise. To design symbolic algorithms, we represent discrete-time dynamical systems by finite sets of actions. Actions are assignments with guards and costs; they represent finite or infinite sets of transitions. Guards are predicates characterizing definition domains of actions. Any tuple of guards represents some memoryless control policy. The control objective is to minimize the costs of executions reaching a given target set. Our aim is to generate the policy which realizes the latter objective. The proposed symbolic approach is simple: policy-iterates, viz. tuples of guard-iterates, are generated iteratively by backwards induction, on the basis of the optimality constraints.

The paper is organized as follows. Sections 2 and 3 introduce the framework and recall relevant results. Section 4 develops the proposed symbolic iteration. In Section 5, the latter iteration is displayed and is illustrated, and its complexity is analyzed. Related work is discussed in Section 6, and Section 7 offers a conclusion. A key proof is detailed in an Appendix.

Notations. If the context is clear, universal quantifiers and domains of variables may be omitted. The satisfiability expression $P \not\equiv \mathbf{false}$ is equivalent to $\exists x : P(x)$. The set of non-negative integers is denoted by $\mathbb{N}$, its cardinal by $\#\mathbb{N}$, and $\mathbb{N} \cup \{\#\mathbb{N}\}$ by $\mathbb{N}_\infty$. The set of (partial) functions from X to Y is denoted by $X \to Y$. The definition domain of f is denoted by $Dom(f)$, and its range $Rng(f)$ is the image of $Dom(f)$ by f. If $(P^{(m)})_{m \in M}$ is an ascending chain then $P^{(\sup M)} \doteq \sup_{m \in M}\{P^{(m)}\}$. A tuple may be denoted by $\bar{C}$ and its i-th component by C_i. The substitution of u for x in E is denoted by E^x_u.

2 Action Systems and their Graphs

Simple action-systems symbolically represent discrete transition systems on finite or infinite state-spaces. They determine labelled, weighted transition-graphs.

2.1 Action Systems and Control Policies

Action Systems. A *(symbolic action) system* S is an iteration of guarded commands [**?**], of the form $S = \mathbf{do}\ A\ \mathbf{od}$ where $A = A_0 [] \cdots [] A_{N-1}$. So N is the number of commands in S. The set I is a finite set of *(action) labels*; here $I = \mathbb{N} \cap [0, N-1]$. Thus $A = []_{i\in I} A_i$. For each $i \in I$, the *action* A_i is a guarded assignment

$$B_i(x) \to x := f_i(x) \ < w_i > . \tag{1}$$

The *state-space* X of S is a set of states. The variable x ranges over X. If $X = Z_0 \times \cdots \times Z_m$, this variable may be replaced by variables z_l over Z_l where $l = 0 \cdots m$. The *action map* is $f_i \in X \to X$. The *guard* B_i is a predicate characterizing a subset of $Dom(f_i)$. A *target (predicate)* Q accompanies S.

The *action cost* w_i is a strictly positive integer, and the maximum action cost is M_w, viz. $M_w \doteq \max_{i\in I}\{w_i\}$. In case action costs are initially given as rational numbers, they are mapped to integers by a change of scale.

Existential Predicate Transformer. It yields a predicate characterizing the set of states from which there is an execution by S to a state verifying a given predicate P. It is denoted by $pre.S$ and has a classical definition (e.g. [**?**]):

$$pre.S.P \equiv \bigvee_{n\in\mathbb{N}} (pre.A)^n.P, \quad pre.A.P \equiv \bigvee_{i\in I} pre.A_i.P, \quad pre.A_i.P \equiv B_i \wedge P^x_{f_i(x)} \tag{2}$$

where $i \in I$, and where $\alpha^{n+1}.P \equiv \alpha^n.(\alpha.P)$ and $\alpha^0.P \equiv P$ for $\alpha = pre.A$.

Symbolic Control Policies. Given an action system S with actions as in (**??**), a *(symbolic control-)policy* for S is a tuple $\bar{C}$ of guards such that the i-th guard C_i implies B_i for each $i \in I$, namely $\bigwedge_{i\in I}(C_i \Rightarrow B_i)$. A tuple $\bar{C}$ of guards can be seen as a label-to-predicate map such that $\bar{C}(i) \equiv C_i$.

If $\bar{C}$ is a policy for S then the system $S{\downarrow}\bar{C}$ - or S *controlled by* $\bar{C}$ - is the result of replacing each i-th guard in S by C_i.

Given two policies $\bar{C}$ and $\bar{C}'$ for S, $\bar{C}$ *refines* $\bar{C}'$ iff $\bigwedge_{i\in I}(C_i \Rightarrow C'_i)$ holds. We also say that $\bar{C}'$ is *weaker than* $\bar{C}$. Policy refinement - or strengthening [**?**] - is a basic form of program refinement (e.g. [**?**]).

Example. Consider the state-space $X = \mathbb{R}$, the action system

$$S = \mathbf{do}\ A_0 \colon x \geq 0 \to x := 4x \ <26> \ [] \ A_1 \colon x \geq 0 \to x := x+1 \ <13> \ \mathbf{od} \tag{3}$$

and the target predicate $Q(x) \equiv 8 \leq x \leq 10$. The actions in (**??**), without costs, amount to the recurrence inclusion $x_{k+1} \in \{4x_k \mid x_k \geq 0\} \cup \{x_k + 1 \mid x_k \geq 0\}$.

The weakest optimal policy $\bar{C}$ is the pair of guards (C_0, C_1) where $C_0 \equiv (0.5 \leq x \leq 0.625) \vee (1.5 < x \leq 2.5)$ and $C_1 \equiv (0 \leq x \leq 0.5) \vee (0.625 < x \leq 1.5) \vee (2.5 < x < 8)$; the derivation of $\bar{C}$ is tackled later (§5.1). For instance, $x = 1.40873$ verifies C_1 because there is an optimal execution with the initial state 1.40873 and the initial action A_1. The optimally controlled system $S{\downarrow}\bar{C}$ is

$$\begin{array}{ll} \textbf{do}\, A_0 : & (0.5 \leq x \leq 0.625) \vee (1.5 < x \leq 2.5) \quad \rightarrow x := 4x \quad <26> \\ [\!]\, A_1 : & (0 \leq x \leq 0.5) \vee (0.625 < x \leq 1.5) \vee (2.5 < x < 8) \quad \rightarrow x := x+1 <13> \textbf{od}. \end{array}$$

2.2 Graphs, Paths and their Costs

Graphs. Given an action system S, the *graph of* S is the tuple (X, E_S, w_S) where the set $E_S \subseteq X \times I \times X$ of labelled edges is $\bigcup_{x \in X, i \in I} \{(x, i, f_i(x)) \mid B_i(x)\}$ and the weight function $w_S \in I \rightarrow \mathbb{N}\backslash\{0\}$ is defined by $w_S(i) = w_i$ for $i \in I$.

Paths. A *path by* S is a non-empty, alternating sequence of states and compatible labels, generated by the graph of S. It may thus be a sequence such as (x_0) where $x_0 \in X$, or as $(x_0, i_1, x_1, \ldots, i_n, x_n)$ where $(x_{k-1}, i_k, x_k) \in E_S$ for $k \in \mathbb{N} \cap [1, n]$. A path by S may be infinite.

The set of paths by S is denoted by $Paths.S$. If $\bar{C}'$ is weaker than $\bar{C}$ (§2.1) then $Paths.(S{\downarrow}\bar{C}) \subseteq Paths.(S{\downarrow}\bar{C}')$. The set of paths by S which begin with $x \in X$ is denoted by $Paths.S.x$. If P is a predicate characterizing a subset of X then $Paths.S.x.P$ denotes the set of finite paths by S which begin with $x \in X$ and finish in a state verifying P. Because of (**??**), $(pre.S.P)(x) \equiv (Paths.S.x.P \neq \emptyset)$.

Consider two paths p_1 and p_2 by S such that the final state of p_1 is the initial state of p_2. Their *(chop-)concatenation* $p_1.p_2$ is the concatenation of the sequence p_1 with the sequence p_2 from which its initial state has been removed.

Path Costs. The function $cost \in Paths.S \rightarrow \mathbb{N}_\infty$ yields the total cost of any path by S. It is additive: $cost((x, i, y).p) = w_i + cost(p)$ and $cost((x)) = 0$.

Let $nb_{edges}(p)$ be the number of occurrences of edge labels in a path p by S. Since each action cost w_i verifies $1 \leq w_i \leq M_w$ (§2.1), it is easy to derive

$$\forall p \in Paths.S : \quad nb_{edges}(p) \leq cost(p) \leq nb_{edges}(p) \times M_w. \tag{4}$$

3 State-Based Generation of Optimal Control Policies

Given a system S and a target predicate Q, the *(optimality) domain* is the predicate D such that $D(x) \equiv (Paths.S.x.Q \neq \emptyset)$. We write D instead of $D_{S,Q}$ for brevity. On the basis of §2.2, we observe that

$$D \equiv pre.S.Q, \qquad \forall i \in I, x \in X : B_i(x) \wedge D(f_i(x)) \Rightarrow D(x). \tag{5}$$

We recall a few classical results (e.g. [**?**][**?**][**?**]). The *(state-to-)value function* $V \in X \rightarrow \mathbb{N}$, a.k.a. *cost-to-go function*, is standard: for all x such that $D(x)$,

$$V(x) = \min\{cost(p) \mid p \in Paths.S.x.Q\}. \tag{6}$$

Because of (**??**) and (**??**), for all $x \in X$ and $i \in I$ such that $B_i(x) \wedge D(f_i(x))$,

$$V(x) \leq w_i + V(f_i(x)). \tag{7}$$

Hence $V(x) = \min_{i \in I}\{w_i + V(f_i(x)) \mid B_i(x) \wedge D(f_i(x))\}$. The *(weakest) optimal policy* $\bar{C}$ for S and Q is the policy for S such that, for all $x \in X$,

$$Paths.(S{\downarrow}\bar{C}).x = \{p \mid p \in Paths.S.x.Q \wedge cost(p) = V(x)\}. \tag{8}$$

There is no other, weaker optimal policy because $\bar{C}$ permits all the optimal paths by S to Q (§2.2). Given (**??**), $\bar{C}$ can also be characterized as follows:

$$\forall i \in I, x \in X:\ C_i(x) \equiv B_i(x) \wedge D(f_i(x)) \wedge (\, V(x) = w_i + V(f_i(x)) \,). \tag{9}$$

So each optimal guard C_i implies $\neg Q$, given $V(x) > 0 \equiv \neg Q(x)$. Moreover

$$D \equiv Q \vee \bigvee_{i \in I} C_i, \qquad \forall x \in X : C_i(x) \Rightarrow D(x) \wedge D(f_i(x)). \tag{10}$$

*(State-Based Generation Method) Assume the state-space X is finite. The optimal control policy for a system S and a target Q can be obtained in two steps: first, compute the state-to-value function (**??**); second, unfold (**??**).*

The complexity of this method is polynomial in the number $\#X$ of states, thanks to efficient algorithms for shortest paths [**?**][**?**][**?**]. Determinism is not required by (**??**), but in practice the generated policies usually are deterministic.

4 Development of a Symbolic Generation Technique

In the greedy algorithm [**?**] for shortest paths, each iteration step computes the next optimal value, if needed, and generates one new state having the last computed optimal value; there are as many steps as states. Here we use a set-based greedy schema: each step computes the next optimal value and generates the set of all states having this value; there are as many steps as optimal values.

The optimal policy is thus iteratively generated level by level, where levels rank optimal values. The policy-iterate for level n restricts the optimal policy to the states having an optimal value less or equal to the n-th one.

4.1 Bijection between Levels and Optimal Values

The set X_L of *(optimality) levels* is $\mathbb{N} \cap [0, \#Rng(V) - 1]$ if $Rng(V)$ is finite, and it is $\mathbb{N}$ otherwise; cf. (**??**). The *level-to-value bijection* $V_L \in X_L \to Rng(V)$ enumerates the optimal values in increasing order: for $n, n+1 \in X_L$,

$$V_L(0) = 0, \qquad V_L(n+1) = \min_{x : D(x)} \{V(x) \mid V(x) > V_L(n)\}. \tag{11}$$

Obviously, the inverse bijection V_L^{-1} yields the level of any optimal value.

The *(optimality) level* of a state x verifying D is the level $V_L^{-1}(V(x))$ of its optimal value $V(x)$. So the *(state-to-)level function* $L \in X \to X_L$ is given by $L = V_L^{-1} \circ V$. Clearly, for x such that $D(x)$ and y such that $D(y)$,

$$V = V_L \circ L, \qquad V(x) \geq L(x), \qquad V(x) > V(y) \Rightarrow L(x) > L(y). \tag{12}$$

4.2 Symbolic Computation of Optimal Policies

The *policy-iterate* $\bar{C}^{(n)} = (C_0^{(n)}, \cdots, C_{N-1}^{(n)})$ is the restriction of the optimal policy $\bar{C} = (C_0, \cdots, C_{N-1})$ to the states x having a level at most n. Accordingly the *guard-iterate* $C_i^{(n)}$ characterizes a level set: for $n \in X_L$, $i \in I$ and $x \in X$,

$$C_i^{(n)}(x) \;\equiv\; C_i(x) \wedge L(x) \leq n. \tag{13}$$

The *policy-stratum* $\bar{F}^{(n)} = (F_0^{(n)}, \cdots, F_{N-1}^{(n)})$ restricts $\bar{C}$ to the states which have the level n. It is thus the fringe - or front - of the policy-iterate $\bar{C}^{(n)}$. So the *guard-stratum* $F_i^{(n)}$ is the fringe of the guard-iterate $C_i^{(n)}$:

$$F_i^{(n)}(x) \;\equiv\; C_i(x) \wedge L(x) = n. \tag{14}$$

The *domain-iterate* $D^{(n)}$ and *domain-stratum* $H^{(n)}$ are defined similarly:

$$D^{(n)}(x) \;\equiv\; D(x) \wedge L(x) \leq n, \qquad H^{(n)}(x) \;\equiv\; D(x) \wedge L(x) = n, \tag{15}$$

The *optimality radius* $\rho \in \mathbb{N}_\infty$ is the number of nonzero optimal values. So, given (**??**),

$$\rho = \sup X_L = \sup Dom(V_L). \tag{16}$$

Clearly, $C_i \equiv \bigvee_{n \in X_L} C_i^{(n)} \equiv C_i^{(\rho)}$. Hence the optimal guards and the domain-iterates are generated as follows: for $i \in I$ and $n, n+1 \in X_L$,

$$F_i^{(0)} \equiv \mathbf{false}, \quad C_i^{(0)} \equiv \mathbf{false}, \quad C_i^{(n+1)} \equiv C_i^{(n)} \vee F_i^{(n+1)}, \quad C_i \equiv C_i^{(\rho)} \tag{17}$$

$$H^{(0)} \equiv Q, \qquad D^{(0)} \equiv Q, \qquad H^{(n+1)} \equiv \bigvee_{i \in I} F_i^{(n+1)} \tag{18}$$

$$D^{(n+1)} \equiv D^{(n)} \vee H^{(n+1)}. \tag{19}$$

If ρ is finite then it is the maximum $n+1$ such that $\neg D^{(n)} \wedge pre.A.D^{(n)} \not\equiv \mathbf{false}$. The iteration condition is $\neg D^{(n)} \wedge pre.A.D^{(n)} \not\equiv \mathbf{false}$.

It remains to show how to compute the guard-strata $F_i^{(n+1)}$ symbolically.

4.3 Asynchronous Symbolic Computation of Guard-Strata

In (**??**), we unfold C_i using (**??**), apply the level-to-value bijection V_L instead of the state-to-value map V and eliminate the state-to-level map L.

To this end, we introduce the predicates opt_i over levels and the level-to-level maps g_i such that, for all $i \in I$ and $n \in X_L$,

$$opt_i(n) \equiv (V_L(n) - w_i \in Rng(V_L)), \quad opt_i(n) \Rightarrow V_L(n) - w_i = V_L(\, g_i(n)). \tag{20}$$

The guard-stratum $F_i^{(n+1)}$ is then computed by

$$F_i^{(n+1)} \equiv \begin{cases} \neg D^{(n)} \wedge pre.A_i.H^{(g_i(n+1))}, & \text{if } opt_i(n+1) \\ \mathbf{false}, & \text{otherwise.} \end{cases} \tag{21}$$

The iteration step (**??**) is asynchronous since in general we have $n > g_i(n+1)$. It results from two properties, as shown in the Appendix: for all $i \in I$, $x \in X$ and $n \in X_L$,

$$\text{(Abstraction)} \quad C_i(x) \;\Rightarrow\; opt_i(L(x)), \quad C_i(x) \;\Rightarrow\; L(f_i(x)) = g_i(L(x)) \tag{22}$$

$$\text{(Confinement)} \qquad opt_i(n) \Rightarrow (pre.A_i.H^{(g_i(n))} \Rightarrow D^{(n)}). \tag{23}$$

The functions opt_i, g_i and V_L over levels are computed as follows: for $n > 1$,

$$opt_i(n) \equiv \bigvee_{m \in \mathbb{N} \cap [0,n-1]} V_L(n) = w_i + V_L(m), \tag{24}$$

$$g_i(n) = \min_{m \in \mathbb{N} \cap [0,n-1]} \{m \mid V_L(n) = w_i + V_L(m)\} \quad \text{if } opt_i(n) \tag{25}$$

$$V_L(n) = \min_{i \in I, m \in \mathbb{N} \cap [0,n-1]} \{w_i + V_L(m) \mid \neg D^{(n-1)} \wedge pre.A_i.H^{(m)} \not\equiv \mathbf{false}\}. \tag{26}$$

5 Asynchronous Iterative Generation of Optimal Policies

The symbolic procedure resulting from the development in §4 is displayed hereafter. It is called a "semi-algorithm" because it does not always terminate.

In the generated expressions, valid simplifications may be used freely, and the applications of the auxiliary functions $pre.A_i$, opt_i, g_i and V_L are evaluated according to (**??**), (**??**), (**??**) and (**??**).

Recall that a policy-iterate $\bar{C}^{(n)}$ is the tuple $(C_0^{(n)}, \cdots, C_{N-1}^{(n)})$.

5.1 Symbolic Semi-Algorithm *OptPol*

The input data are formal expressions for an action system S and a target predicate Q (§2.1). Let $n, n+1 \in \mathbb{N}$ and $i \in I$.

$$\textbf{begin} \quad \textbf{for}\, i : (F_i^{(0)} := \textbf{false};\; C_i^{(0)} := \textbf{false}); \tag{27}$$

$$H^{(0)} := Q; \quad D^{(0)} := Q; \tag{28}$$

$$\textbf{for}\; n \;\textbf{from}\; 0 \;\textbf{while}\; \neg D^{(n)} \wedge pre.A.D^{(n)} \not\equiv \textbf{false}:$$

$$\textbf{for}\; i \;: F_i^{(n+1)} := \begin{cases} \neg D^{(n)} \wedge pre.A_i.H^{(g_i(n+1))}, & \text{if } opt_i(n+1) \\ \textbf{false}, & \text{otherwise;} \end{cases} \tag{29}$$

$$C_i^{(n+1)} := C_i^{(n)} \;\vee\; F_i^{(n+1)}\;; \tag{30}$$

$$H^{(n+1)} := \bigvee_{i \in I} F_i^{(n+1)}; \quad D^{(n+1)} := D^{(n)} \vee H^{(n+1)}; \tag{31}$$

$$\bar{C} \;:=\; \bar{C}^{(\sup Dom(V_L))} \qquad \textbf{end} \tag{32}$$

Correctness. By construction (§4), the result $\bar{C}$ is the weakest optimal policy for S and Q. The semi-algorithm *OptPol* terminates iff the number of optimal values is finite (**??**) and the auxiliary evaluations, e.g. in (**??**), terminate.

Example (continued). See (**??**). The semi-algorithm *OptPol* generates the weakest optimal policy $\bar{C} = (C_0, C_1)$. The optimality radius ρ is 6; cf. (**??**). So $\bar{C} = \bar{C}^{(6)} = (C_0^{(6)}, C_1^{(6)})$. For the first three levels, the iterates are

$$
\begin{array}{ll}
V_L(0) = 0 & \\
\quad \bar{F}^{(0)} = (F_0^{(0)}, F_1^{(0)}) = (\mathbf{false}, \mathbf{false}), & \bar{C}^{(0)} = (C_0^{(0)}, C_1^{(0)}) = (\mathbf{false},\ \mathbf{false}) \\
\quad H^{(0)} \equiv Q \equiv 8 \leq x \leq 10, & D^{(0)} \equiv H^{(0)} \equiv 8 \leq x \leq 10 \\
V_L(1) = 13 & \\
\quad \bar{F}^{(1)} = (\mathbf{false},\ (7 \leq x < 8)), & \bar{C}^{(1)} = (\mathbf{false},\ (7 \leq x < 8)) \\
\quad H^{(1)} \equiv F_0^{(1)} \vee F_1^{(1)} \equiv 7 \leq x < 8, & D^{(1)} \equiv D^{(0)} \vee H^{(1)} \equiv 7 \leq x \leq 10 \\
V_L(2) = 26 & \\
\quad \bar{F}^{(2)} = ((2 \leq x \leq 2.5),\ (6 \leq x < 7)), & \bar{C}^{(2)} = ((2 \leq x \leq 2.5),\ (6 \leq x < 8)) \\
\quad H^{(2)} \equiv (2 \leq x \leq 2.5) \vee (6 \leq x < 7), & D^{(2)} \equiv (2 \leq x \leq 2.5) \vee (6 \leq x \leq 10).
\end{array}
$$

For instance, given (**??**), $F_0^{(2)} \equiv \neg D^{(1)} \wedge pre.A_0.H^{(g_0(2))} \equiv \neg D^{(1)} \wedge pre.A_0.H^{(0)} \equiv \neg(7 \leq x \leq 10) \wedge x \geq 0 \wedge (8 \leq 4x \leq 10) \equiv (2 \leq x \leq 2.5)$. Indeed $opt_0(2)$ and $g_0(2) = 0$ follow from $V_L(2) = w_0 + V_L(g_0(2)), V_L(2) = 26, w_0 = 26$ and $V_L(0) = 0$; see (**??**) and (**??**).

The optimal policy $\bar{C}$ is nondeterministic on the state 0.5 because $C_0 \wedge C_1 \equiv (x = 0.5)$. There are two optimal paths from 0.5, namely $(0.5, 0, 2, 0, 8)$ and $(0.5, 1, 1.5, 1, 2.5, 0, 10)$; their cost 52 is the 4-th optimal value, viz. $52 = V_L(4)$. The optimality domain D is $Q \vee C_0 \vee C_1 \equiv 0 \leq x \leq 10$; see (**??**).

5.2 Complexity Analysis

Given a semi-algorithm $\mathcal{A}$, let $T(\mathcal{A})$ be its complexity, and let $T_{\text{SPRE}}(\mathcal{A})$ be the complexity of evaluating the satisfiability expressions and predicate transformations used in $\mathcal{A}$. Assume also that $f \in Poly(g)$ stands for $\exists k \in \mathbb{N} : f \in \mathcal{O}(g^k)$.

Complexity of *OptPol*. Recall $N = \#I$ (§2.1). Given §5.1,

$$T(OptPol) \in Poly(\rho + N + T_{\text{SPRE}}(OptPol)). \tag{33}$$

Thus *OptPol* is more practical than the state-based method (§3) in two cases: (*a*) X is finite but huge, and $\rho + N + T_{\text{SPRE}}(OptPol) \in Poly(\log \#X)$; (*b*) X is infinite, ρ_R is finite and the auxiliary symbolic evaluations terminate. Case (*a*) entails an exponential reduction of complexity. Case (*b*) is illustrated in §5.1.

Comparison with the Symbolic Generation of Reachability Sets. As classical (e.g. [**?**]), the reachability predicate R is generated by the semi-algorithm $ReachSet \doteq$ **begin** $R^{(0)} := Q$; (**for** n **from** 0 **while** $\neg R^{(n)} \wedge pre.A.R^{(n)} \not\equiv$ **false** : $R^{(n+1)} := R^{(n)} \vee pre.A.R^{(n)}$); $R := R^{(\rho_R)}$ **end**. The iteration in *ReachSet* clearly is synchronous. Of course $R(x) \equiv (pre.S.Q)(x) \equiv (Paths.S.x.Q \neq \emptyset)$.

The reachability radius ρ_R is the least cardinal such that, for all x verifying R, there exists $p \in Paths.S.x.Q$ such that $nb_{edges}(p) \leq \rho_R$. For important classes of

problems, ρ_R is finite (e.g. [**?**]). ¿From (**??**), it is possible to derive the following relations between the optimality radius ρ and the reachability radius ρ_R :

$$\rho_R \leq \rho \leq \rho_R \times M_w. \tag{34}$$

In particular, ρ is finite iff ρ_R is finite.

So (**??**) becomes $T(OptPol) \in Poly(\rho_R + M_w + N + T_{\text{SPRE}}(OptPol))$. On the other hand, it is clear that $T(ReachSet) \in Poly(\rho_R + N + T_{\text{SPRE}}(ReachSet))$.

As a consequence, the complexity of $OptPol$ is that of $ReachSet$ if no action cost is exponentially greater than the reachability radius and if constraint solving is not exponentially harder in $OptPol$ than in $ReachSet$. In other terms,

$$\begin{aligned} & (M_w \in Poly(\rho_R)) \wedge (T_{\text{SPRE}}(OptPol) \in Poly(T_{\text{SPRE}}(ReachSet))) \\ \Rightarrow\ & T(OptPol) \in Poly(T(ReachSet)). \end{aligned}$$

Because of this similarity, the techniques used to implement the semi-algorithm $ReachSet$ can be reused to implement the semi-algorithm $OptPol$, and the applicability range of $OptPol$ is often akin to that of $ReachSet$.

6 Related Work

The proposed semi-algorithm generalizes a synchronous iterative method [**?**] which produces termination policies. Actually, the latter policies match the optimal policies determined by action costs which are all equal. In the latter case, $g_i(n+1) = n$ for all i and n; then the iteration steps (**??**) are synchronous. If the action maps and the target predicate are left unchanged, different tuples of action costs in general give rise to different optimal policies, which correspond to different termination policies.

Symbolic methods generating optimal policies have been developed for hybrid systems. Discrete-time systems have not been tackled so far. Regarding hybrid systems, a relevant contribution [**?**] presents a symbolic procedure which yields optimal strategies for timed game-automata with costs. In the latter work, candidate strategies are first generated by a synchronous iteration, and optimal strategies are then determined using a set of cost-dependent polyhedral sets. Besides, the constraints defining the dynamics are linear, as typical for hybrid systems (e.g. [**?**]), and the resulting optimal strategies are deterministic. Here, action maps may be nonlinear, as usual in programs, and the optimal policies are the weakest ones, viz. they are maximally nondeterministic, which permits further refinement steps in design.

7 Conclusion

The approach introduced in the paper leads to a simple but unexpected conclusion: the symbolic iterative techniques which serve to ensure qualitative properties such as reachability or termination can easily be adapted in order to ensure a quantitative property such as optimality. Indeed, it suffices to replace synchronous iterations by asynchronous ones.

Appendix : Proof of (??). Let $x \in X, i \in I$ and $n+1 \in X_L$.

$$
\begin{aligned}
& F_i^{(n+1)}(x) \\
\equiv\ & (L(x) = n+1) \wedge C_i(x) && [(??)] \\
\equiv\ & (L(x) = n+1) \wedge C_i(x) \wedge opt_i(n+1) \\
& \qquad \wedge (L(f_i(x)) = g_i(n+1)) && [(??)] \\
\equiv\ & (L(x) = n+1) \wedge B_i(x) \wedge opt_i(n+1) \\
& \qquad \wedge D(f_i(x)) \wedge (L(f_i(x)) = g_i(n+1)) && [(??), (??), (??)] \\
\equiv\ & (L(x) = n+1) \wedge B_i(x) \wedge opt_i(n+1) \wedge H^{(g_i(n+1))}(f_i(x)) && [(??)] \\
\equiv\ & (L(x) = n+1) \wedge opt_i(n+1) \wedge pre.A_i.(H^{(g_i(n+1))}(x)) && [(??)] \\
\equiv\ & \neg D^{(n)}(x) \wedge opt_i(n+1) \wedge pre.A_i.(H^{(g_i(n+1))}(x)) && [(??), (??)].
\end{aligned}
$$

The proofs of (??), (??), (??) and (??) are rather easy. They are omitted because of space limitations.

Acknowledgments. We gratefully acknowledge helpful suggestions by J.-F. Raskin, F. Cassez, referees and participants to meetings of IFIP Working Groups 2.1 (Algorithmic Languages and Calculi) and 2.3 (Programming Methodology).

References

1. Back, R.-J., and J. von Wright, *Refinement Calculus*, Springer, New York, 1998.
2. Bellman, R., *Dynamic Programming*, Princeton Univ. Press, 1957.
3. Bertsekas, D., *Dynamic Programming and Optimal Control*, Athena Scientific, Belmont, Mass., 2000.
4. Bouyer, P., F. Cassez, E. Fleury and K. Larsen, Synthesis of optimal strategies using HyTech, *Electronic Notes Theor. Computer Sci.* **119** (2005) 11-31.
5. Clarke, E.M., O. Grumberg and D.A. Peled, *Model Checking*, MIT Press, Cambridge, Mass., 1999.
6. Dijkstra, E.W., A note on two problems in connexion with graphs, *Numerische Mathematik* **1** (1959) 269-271.
7. Dijkstra, E.W., Guarded commands, non-determinacy and formal derivation of programs, *Commun. ACM* **18** (1975) 453-457.
8. Floyd, R., Algorithm 97 (Shortest path), *Commun. ACM* **5** (1962) 345.
9. Henzinger, T.A., R. Majumdar and J.-F. Raskin, A classification of symbolic transition systems, *ACM Trans. Computational Logic* **6** (2005) 1-32.
10. Sontag, E.D., *Mathematical Control Theory*, Springer, New-York, 1990.
11. van Lamsweerde, A., and M. Sintzoff, Formal derivation of strongly concurrent programs, *Acta Informatica* **12** (1979) 1-31.
12. Wong-Toi, H., The synthesis of controllers for linear hybrid automata. In *Proc. 36th IEEE Conf. Decision and Control*, pp. 4607-12, IEEE Computer Press, 1997.

Synthesis of data views for communicating processes

(extended abstract)

Iman Poernomo[1]

Department of Computer Science,
King's College London, Strand, London, WC2R2LS.
iman.poernomo@kcl.ac.uk

1 Introduction

System components interact with clients by two means: they expose methods to change their state, and provide side-effect-free data views of their state. Often, a system requires that such communication adheres to a protocol or order. For instance, in a banking component, the data on an account holder's bank balance should not be accessed prior to the account holder entering a correct identification code. This paper is concerned with the specification and synthesis of such data retrieval protocols.

We describe an augmented version of Milner's Calculus of Communicating systems for defining data retrieval protocols, a novel approach to the specification of data retrieval protocols based on traditional realizability notions and a deductive system for simultaneously deriving protocols and their specification.

We will be specifying and synthesizing the behaviour of distributed programs built on a synchronous and asynchronous messaging infrastructure. In particular, we address an important and relatively unexplored issue in the formal development of complex systems: the synthesis of complex, side-effect-free data views for distributed programs. Data views are an important aspect of all software. In object-oriented terms, they are often implemented as accessor methods that enable clients to obtain information about the state that an object encapsulates. In the case of enterprise applications, data views implement domain-specific business logic and are consequently difficult to specify and implement correctly. Our work uses proofs-as-programs techniques to specify and develop provably correct complex data views in tandem with distributed programs.

Rather than work with a specific programming language, we will consider an abstract coordination language to model distributed data retrieval protocols. Our coordination language consists of Milner's Calculus of Communicating Systems (CCS) [2] without fixed points, extended with extra constructs to denote data views that can be accessed at certain points in a system execution. Terms of our language can be easily transformed into actual systems. Basic components are modelled as CCS processes. CCS messages represent side-effect producing methods of component interfaces and data views represent side-effect-free accessor methods of interfaces. The absence of recursion corresponds to the absence of feedback loops within component architectures (the usual situation in case of enterprise systems). Synchronous and asynchronous communication between components is modelled via CCS message exchange. Data views of components are represented as lambda terms.

An important aspect of our language is that it supports the modelling of system protocols – the orders in which messages should be received and sent. We extend the traditional representation of a system protocol in the *CCS* to include data views. There are points in a system's execution where data views should not be accessed. For instance, as part of a security protocol, an authorizing message might need to be received to enable access to confidential data. Our language enables us to model such protocols.

We specify program behaviour as modal Hennessy-Milner formulae and provide a constructive proof system for reasoning with these specifications. Hennessy-Milner formulae are not enough to specify associated data views and the logic alone cannot be used to synthesize required views. We will define a method for data views and their dynamic behaviour with respect to system execution. We shall be able to specify two aspects of data view behaviour. *1. Functional behaviour.* We can specify what kind of values a data view should have, with respect to an associated system description. *2. Dynamic behaviour.* As a system executes, the value of a data view will evolve. The accessor method of an object will not necessarily produce the same result at different stages in the object's lifetime, as the state of the object will change. We will show how to specify modal development in data view values: requirements of data view evolution and protocols with respect to message activity. Our method adapts notions of constructive realizability to make such specifications, enabling a synthesis methodology that adapts traditional proofs-as-programs to extract data retrieval protocols from proofs of their specification.

This paper proceeds as follows. Section 2 summarizes our augmented version of the CCS. Section 3 defines the Hennessy-Milner formulae, explaining how these formulae specify behaviour of *CCS* processes and, by extending realizability notions, data views. Section 4 presents our Hennessy-Milner logic and shows how proofs of the logic can be encoded within a logical type theory. We sketch the idea of proofs-as-distributed-programs in Section 5. We briefly review related work and provide concluding remarks in section 6.

Lambda terms for a many sorted signature. Our approach is parametrized by the choice of a *many-sorted signature* $\Sigma = \langle S, TF, P \rangle$, consisting of: 1) a set S of sorts. Sorts are generated from a set of *basic sorts*, $B(S)$ according to the following inductive definition. First, $B(S) \subseteq S$. Also, if s_1 and s_2 are in S, then so are the function sort $(s_1 \to s_2)$, the product sort $(s_1 * s_2)$ and the disjoint union $(s_1|s_2)$. We assume that $B(S)$ includes a special sort, called $Unit$. 2) sorted function symbols, TF. We assume a single inhabitant $()$ of the sort $Unit \in B(S)$. 3) sorted predicate symbols P of predicate symbols. We define the *terms* $Term(\Sigma)$ for a signature $\Sigma = \langle S, TF, P \rangle$ in the usual manner, but extended to include a lambda calculus, written in an *SML* style syntax.The terms have a semantics given by the usual lambda reduction rules. The semantics is given by the reduction relation $\rhd_\Sigma$.The relation is defined by the usual reduction rules and additional rules for dealing with projections and cases, together with a rule for evaluating function symbol application. We assume that function symbols will evaluate according to a semantics given by Σ.

2 CCS with data views

We now define our extended version of *CCS*. We define the set of *Actions* to consist of incoming messages m and outgoing messages $\bar{m}$ with m ranging over some set of messages. The grammar for our extension, *CCS* with data views (written *CCS*) is given as follows.

$$CCS := 0 \mid X \mid (p + q) \mid (p|q) \mid p/s \mid s.p \mid \mu X.p \mid (p \; view \; a)$$

where $p, q \in CCS$, X is ranges over a given set of variables $TVar$, s ranges over *Actions* and a is a closed lambda term from $Term(\Sigma)$. Process terms form a semantics of system architectures in the following standard sense. A *CCS* process denotes the state of a distributed system in terms of its ability to perform actions and the protocol in which actions are to be performed. Actions are either sending or receiving messages or internal computation. Incoming and outgoing message actions are denoted by letters, taken from the same set, with outgoing messages marked by an overbar ($^-$). The τ action designates internal computation (processing that is not observable to a client). Proceses are built using the standard recursion, nondeterministic choice, parallel composition and action sequencing constructs of [2].

Data views are functions that access the state of the distributed system and provide information on it. The domain-specific function symbols of Σ provide basic data views. More complex combinations of functional views are provided as lambda terms of $Term(\Sigma)$. A data view t is associated with a process p via the constructor $(p \; view \; t)$.

Operational semantics of *CCS*. The way in which *CCS* programs evaluate is given by a labelled transition system semantics. A process p can receive or send a message m, resulting in a new process p', denoted by a labelled transition $p \xrightarrow{m} p'$. We use the same rules as Milner gives in [2] but with the following addition that says data views can be discarded when evaluating processes

$$\frac{p \xrightarrow{m} q}{(p \; view \; f) \xrightarrow{m} q} \text{ (pure)}$$

Definition 1. *We define the relation $\stackrel{m}{\Rightarrow}$ to hold between two terms a and b when b evolves from a via the action m with possibly some number of τ transitions in between:* $a = \underbrace{a_0 \xrightarrow{\tau} a_1 \xrightarrow{\tau} \ldots \xrightarrow{\tau} a_i}_{0 \leq i} \xrightarrow{m} b_0 \underbrace{\xrightarrow{\tau} b_1 \xrightarrow{\tau} b_j}_{0 \leq j} = b$

3 Specification of system architectures

We define modal many-sorted formulae for a signature $WWF(\Sigma)$ as ordinary many-sorted formulae with universal and existential quantification, extended by modal necessity $[m]F$. We use modal many-sorted formulae to specify and reason about two related aspects of our architectures: possible behaviours and possible data views. This understanding of formulae as specifications is key to our adaptation of proofs-as-programs.

Behavioural specification. Possible behaviour is specified in the standard fashion for Hennessy-Milner formulae with many-sorted quantification.

Definition 2. *A formula F is true of the behaviour of a term t, written $t \Vdash F$, according to the following recursive definition:*

- *If F is atomic, then $h(F,t) = True$.*
- *If $F \equiv \forall x : T \bullet G$, then for every $a : T$, $t \Vdash G[a/x]$.*
- *If $F \equiv \exists x : T \bullet G$, then there is an $a : T$ such that $t \Vdash G[a/x]$.*
- *If $F \equiv G \vee H$, then $t \Vdash G$ or $t \Vdash H$.*
- *If $F \equiv G \wedge H$, then $t \Vdash G$ and $t \Vdash H$.*
- *If $F \equiv G \Rightarrow H$, then $t \Vdash G$ entails $t \Vdash H$.*
- *If $F \equiv [m]G$, then for every u where $t \stackrel{m}{\Rightarrow} u$ it is the case that $u \Vdash G$.*
- *If $F \equiv \langle m \rangle G$, then there is a u such that $t \stackrel{m}{\Rightarrow} u$ and $u \Vdash G$.*
- *$t \vdash \bot$ is never true.*

Specification of data views. A specification of a data view defines the required behaviour of a data view function at a state in a system's execution. Data views are specified as required constructive content of formulae, in a fashion analogous to how functional programs are specified as constructive content of intuitionistic formulae in the proofs-as-programs approach. For instance, the formula $\exists l : Location \bullet ConnectedDB(l)$ can also be seen as specifying a process that *evaluates with a data view* whose content is a constructive witness for l, the location of the database which the process has connected to,

A possible data view specification is a specification of a data view function's behaviour at some future stage in a system's execution. We utilize modalities to make such specifications. For instance, the formula $\langle iCard \rangle \exists d : Account \bullet ValidDetails(d)$ specifyies a realizing a data view r of a program. The formula requires that, if the program receives a message $iCard$, then *possibly* the program provides an output data view r, acting as the witness for d, such that r is a valid account record (not an error record, when $ValidDetails(r)$ is true). The program $(iCard.\overline{chkCard}.0$ *view* $acDetails)$ satisifes this specification, as there are possible executions after receiving $iCard$ such that the data view $acDetails$ is a valid account record ($ValidDetails(r)$).

We use the definition of Harrop formulae and the type extraction map $\mathsf{xsort}(.)$ defined in [5]. This maps logical formulae to Σ sorts. Then we need to extend the notion of Skolem form to our modal formulae, as follow.

Definition 3 (Skolem form and Skolem functions). *Given a closed formula A, we define the* Skolemization *of A to be the Harrop formula $Sk(A) = Sk'(A, \emptyset)$, where $Sk'(A, AV)$ is defined as follows. A unique function letter f_A (of sort $\mathsf{xsort}(A)$) called the* Skolem function*, is associated with each A. AV represents a list of variables which will be arguments of f_A.*

- *If A is Harrop, then $Sk'(A, AV) \equiv A$.*
- *item If $A \equiv B \vee C$, then*

$$\begin{aligned} Sk'(A, AV) = (&\forall x : \mathsf{xsort}(A).f_A(AV) = Inl(x) \Rightarrow Sk'(B, AV)[x/f_B]) \\ &\wedge (\forall y : \mathsf{xsort}(B).f_A(AV) = Inr(y) \Rightarrow Sk(C, AV)[y/f_C]) \end{aligned}$$

– *If* $A \equiv B \wedge C$, *then*

$$Sk'(A, AV) = Sk'(B, AV)[fst(f_A)/f_B] \wedge Sk'(C, AV)[snd(f_A)/f_C]$$

– *If* $A \equiv B \rightarrow C$, *then*
  - *if* B *is Harrop,* $Sk'(A, AV) = B \rightarrow Sk'(C, AV)[f_A/f_C]$.
  - *if* B *is not Harrop and* C *is not Harrop,*

$$Sk'(A, AV) = \forall x : s.(Sk'(B, AV)[x/f_B] \rightarrow Sk'(C, AV)[(f_A x)/f_C])$$

– *If* $A \equiv \exists y : s.P$, *then*
  - *when* P *is Harrop,* $Sk'(A, AV) = Sk'(P, AV)[f_A(AV)/y]$
  - *when* P *is not Harrop,*

$$Sk'(A, AV) = Sk'(P, AV)[fst(f_A(AV))/y][snd(f_A(AV))/f_P]$$

– *If* $A \equiv \forall x : s.P$, *then* $Sk'(A, AV) = \forall x.Sk'(P, AV)[(f_A x)/f_P]$.
– *If* $A \equiv [m]P$, *then* $Sk'(A, AV) = [m]Sk'(P, AV)$.
  If $A \equiv \langle m \rangle P$, *then* $Sk'(A, AV) = \langle m \rangle Sk'(P, AV)$.

In a typical proofs-as-programs method such as [5], a formula A specifies a functional lambda term program p if, and only if, the program is an *intuitionistic modified realizer* of A, now defined.

Definition 4 (Intuitionistic modified realizers). *Let* p *be closed element of* $Term(\Sigma)$. *Let* A *be a non-modal formula. Then* p *is an intuitionistic modified realizer of* A *when* $\vdash Sk(A)[p/f_A]$.

We extend this definition to hold between process terms and formulae, to specify possible data views of processes. Data views are functional programs. So, a data view can be specified as an intuitionistic modified realizer. The presence of modal formulae permits us to formally extend the concept of realizability to specification of *possible* data views of processes. For instance, we treat modal formulae of the form $[m]B$ to specify processes whose execution of event m will result in a data view that is a Skolem formula for B.

Data views may be contained *within* process terms. This fact requires us to extend realizability to views for subterms that are contained within parallel, choice, recursion and message input or output terms. The idea is as follows. A formula can describe the view for an entire process, if such a view exists, and it can also describe visible views of subprocesses in the process. For instance, a parallel term is of the form $a|b$ or $((a|b)\ view\ f)$. If it is the latter, then f is the data view for the process, and a formula F correctly describes this view if f is an intuitionistic realizer. If it is the former, then the term contains two data views – one for each of the subprocesses a and b. The single formula F describes this term accurately if it describes the views of both a and b as realizers.

The definition below extends these ideas recursively to all terms.

Definition 5 (Modal Realizability). *A process* p *is a modal realizer of a formula* A, *written* p mr A, *when the following conditions are satisfied.*

– *If A is Harrop, then $\vdash p \diamond A$ is provable.*
– *Assume A is of the form $[m]B$. Then for all p' such that $p \stackrel{m}{\Rightarrow} p'$ we know that p' mr B.*
– *Otherwise,*
  - *if p is of the form $(p$ view $f)$, and $f \hat{\triangleright}_{\Sigma,p} answer$, then $\vdash p \diamond Sk(A)[answer/f_A]$ holds.*
  - *If p is of the form $q|r$, then q mr A and r mr A.*
  - *If p is of the form $q + r$, then q mr A and r mr A.*
  - *If p is of the form q/m, then q mr A.*
  - *If p is of the form $\mu X.q$ then $q[\mu X.q/X]$ mr A.*

A process p is said to satisfy a process/formula pair $q \diamond A$ when 1) the formula is true of the behaviour of p, 2) the formula correctly specifies a possible data view of p as a modal realizer and 3) p and q are identical, modulo differences in data views. When this is the case, we say that p is a *process realizer* of $l \diamond A$, and we write p pr $l \diamond A$.

$$\frac{p \diamond A \in \mathcal{AX}}{\vdash p \diamond A}\ \text{(Axiom-I)}$$

$$\frac{\vdash p^{a \diamond G} \quad \vdash q^{b \diamond G}}{\vdash \mathsf{parallel}(p,q)^{a|b \diamond G}}\ \text{(parallel)} \qquad \frac{\vdash p^{a \diamond F} \quad \vdash q^{b \diamond F}}{\vdash \mathsf{union}(p,q)^{a+b \diamond F}}\ \text{(union)}$$

provided $Msg(G) \not\subseteq Msg(a) \cup Msg(b)$

$$\frac{\vdash p^{b \diamond P} \quad a \stackrel{n}{\Rightarrow} b}{\vdash \mathsf{pos}(p,n)^{a \diamond \langle n \rangle P}}\ \text{(pos)} \qquad \frac{\vdash p^{a \diamond P} \quad X \text{ is free in } a}{\vdash \mathsf{repl}(X.p)^{\mu X.a \diamond P}}\ \text{(rec)}$$

$$\frac{\vdash p^{a \diamond P} \quad m \text{ does not occur in } P}{\vdash \mathsf{hide}(p,m)^{a/m \diamond P}}\ \text{(hide)} \qquad \frac{\vdash p^{a \diamond F} \quad \vdash_{\mathsf{Int}} q^{F \Rightarrow G}}{\vdash \mathsf{cons}(p,q)^{a \diamond G}}\ \text{(cons)}$$

Fig. 1. Type theoretic presentation of the structural rules of the IHM logic. The standard rules of IHM can be recovered by ignoring the proof-term subscripts, retaining only the superscript types (the program/formula pairs).

4 Deductive system

Hennessy-Milner logics are formal systems for simultaneously reasoning about and constructing *CCS* programs. A sequent-based Hennessy-Milner logic was first described in [6]. We shall employ a simpler, constructive, natural deduction version of that logic, called Intuitionistic Hennessy-Milner logic (IHM), for reasoning about and synthesizing provably correct *CCS* programs with views.

Calculus. The logic manipulates theorems, which consist of pairs of programs and formulae of the form $p \diamond F$, where the left hand side of the diamond is a process, and the right hand side is a specification of the process's behaviour.

Our system is defined with respect to a separate logical subsystem. For purposes of adapting proofs-as-programs, we take this subsystem to be intuitionistic logic, as presented in [4].

The rules of IHM can be obtained from Fig. 1. We motivate each rule as follows. 1) The (parallel) rule tells us that, if G is a property shared by two programs a and b, then G is also true of their parallel composition $a|b$. 2) The (union) rule says that, if G is a property shared by two programs a and b, then G is also true of their nondeterministic choice composition $a + b$. 3) The rule (pos) asserts that, if process a can possibly evolve to b by performing action m (and possibly some internal actions), and A is known to hold over b, then $\langle m \rangle A$ holds for a. 4) The (repl) rule says that if P is known for a then it is known for the replication of a. 5) The (hide) rule says that if P is true of a and does not involve a statement about m, then P is still a true statement about a/m. 6) The (cons) rule permits us to use intuitionistically derived inferences to conclude new things about the same process.

Both the subsystem Int and IHM proper have rules for introducing axioms from a set $\mathcal{AX}$ into a proof. The former rule permits the use of intuitionistic axioms which, by application of (cons), are provable to hold for all processes. The latter rule permits us to add known domain specific truths about processes to the calculus.

One of the important properties of the calculus is that, given a proof of a theorem $p \diamond F$, the formula F is a correct behavioural description of the process p.

The calculus is sound: a proof of a theorem F will result in an accompanying process that satisfies F as a behavioural specification. However, it says nothing about the satisfaction of F as a data view specification. The calculus alone is not enough to produce processes with correct data views with respect to a specification. We need to employ program extraction techniques to do this. The next step in providing such an adaptation is to define the logic as a type theory, to encode proofs for eventual transformation.

Type-theoretic presentation. Our calculus forms a *logical type theory*, *LTT*, with proofs represented as terms (called proof-terms), program/formula pairs represented as types, and logical deduction given as type inference. The proof-terms use a grammar similar to that of standard proofs-as-programs approaches for denoting proofs in Int, but extended with new terms to incorporate the structural rules of the IHM. Because of the (cons) rule, proof-terms corresponding to structural rule applications can involve proof-terms corresponding to intuitionistic logic rule application. Type theoretic presentations of Int and IHM are given in [5](It is important to note that this *LTT* is a lambda calculus that is separate and distinct from the lambda calculus that is used for data views. See [4] for a deeper discussion of the general form of type theories for logics.)

5 Extraction

We now outline our process of extracting process realizers from proofs of specifications. We define an extraction map $\mathsf{extract} : LTT \rightarrow CCS$, from the terms of the logical type theory, *LTT*, to processes of *CCS*. Our map is an extension of the usual intuitionistic extraction map $\mathsf{extract}_{\mathsf{Int}}$ from Int proof-terms to modified realizers, as presented in [5].

We assume that the intuitionistic extraction map always takes axiom introduction proof-terms (of the form $\mathsf{Axiom}_{\mathsf{Int}}(A)$) to modified realizers (lambda terms that

realize A). Also, we assume that each IHM axiom introduction rule is with a proof-term of the form $\mathsf{Axiom}((l\ view\ f) \diamond A)$ such that f is a modified realizer of A and $(l\ view\ f)$ is a process realizer of A. This assumption means that axioms that specify processes and views are always transformed into programs that satisfy these specifications.

We have the following soundness result for intuitionistic extraction.

Theorem 1 (Soundness of intuitionistic extraction). *Take any intuitionistic proof, represented in the LTT as* $\vdash_{\mathsf{Int}} p^P$ *Then* $\mathsf{extract}_{\mathsf{Int}}(p)$ *will produce a modified realizer of* $P \vdash_{\mathsf{Int}} Sk(P)[\mathsf{extract}_{\mathsf{Int}}(p)/f_P]$.

Proof. The proof proceeds according to the usual proofs of extraction soundness for intuitionistic logic (see, e.g., [5]). The presence of modal formulae does not affect the proof.

We wish to derive a similar result for the extraction map over IHM proof-terms. However, the proof is not as straightforward as the intuitionistic case. We require the following definition of *modular* proof-terms.

Definition 6. *A proof-term* t *is* modular *if, and only if, it does* not *contain subterms of the form* $\mathsf{cons}(a, b^{C \Rightarrow D})$ *where* C *is non-Harrop and* a *contains subterms of the form* $\mathsf{parallel}(p, q)$ *or* $\mathsf{union}(p, q)$.

Soundness of extraction is provable for modular proofs.

Theorem 2 (Soundness of modular proof extraction).
Consider any modular proof $\vdash r^{g \diamond G}$. *It is true that* $\mathsf{extract}(r)\ \mathsf{pr}\ g \diamond G$.

In general Theorem 2 does not hold for non-modular terms. If we have a systematic way of transforming non-modular proof-terms into equivalently typed modular proof-terms, then we can use the extraction map and Theorem 2 to obtain correct processes from proofs. This transformation is done via a normalization strategy that moves all applications of the (cons) rule up a derivation, before applications of other structural rules, in the fashion of our example. It is essentially the normalization relation (β reduction) adapted to our lambda calculus of proof-terms and to moving structural rules.

The strong normalization property tells us that the normalization process over a calculus will always terminate. To show that this property holds over our calculus, we need to show that the proof-terms are strongly normalizable, posessing only finite reduction sequences that result in normal, irreducible proof-terms.

Lemma 1. *After normalization, all proof-terms are modular.*

Then, by soundness, Lemma 1 and Theorem 2, we can normalize proof-terms and then apply extract to obtain required process realizers from any proof of a specification.

6 Related work and conclusions

To the best of our knowledge, there has been only one attempt to adapt proofs-as-programs methods to distributed systems synthesis, in the recent paper [1]. Those authors also use a constructive version of the Hennessy-Milner logic, but, rather than using a transformative extraction mapping, they directly take the modal proofs as distributed programs. Proof-terms for the various modalities are understood as remote procedure calls, commands to broadcast computations to all nodes in the network, commands to use portable code and commands to invoke computational agents. An important difference between our approach and these methods is that they are not concerned with data view synthesis, while this is our primary focus. To the best of our knowledge, proofs-as-program style synthesis has never been adapted for synthesis of distributed programs with data views, nor to the case of assertion generation.

In particular, it is interesting to compare the treatment of normalization to work done in the synthesis of (non-distributed) structured code from proofs about CASL specifications [3]. Our results show a successful and practical approach to merging constructive proofs-as-programs with a Hennessy-Milner logic. We retain the advantages of both methods, using them to target their concerns separately. Hennessy-Milner logic is retained to reason about and develop the behaviour of processes. Constructive realizability is adapted to reason and develop functional views of processes. Throughout the extraction process, programs with both aspects are synthesized from proofs.

References

1. Limin Jia and David Walker. Modal proofs as distributed programs (extended abstract). In David A. Schmidt, editor, *Programming Languages and Systems, 13th European Symposium on Programming, ESOP 2004, Barcelona, Spain, March 29 - April 2, 2004*, volume 2986 of *Lecture Notes in Computer Science*, pages 219–233. Springer-Verlag, 2004.
2. Robin Milner. *Communication and Concurrency*. Prentice-Hall, 1989.
3. Iman Poernomo, John Crossley, and Martin Wirsing. Programs, proofs and parametrized specifications. In Maura Cerioli and Gianna Reggio, editors, *Recent Trends in Algebraic Development Techniques, 15th International Workshop, WADT 2001, Genova, Italy, April 1-3, 2001, Selected Papers*, volume 2267 of *Lecture Notes in Computer Science (LNCS)*, pages 280–304, Berlin, 2002. Springer-Verlag.
4. Iman Poernomo, John Crossley, and Martin Wirsing. *Adapting Proofs-as-Programs: The Curry-Howard Protocol*. Monographs in Computer Science. Springer, 2005.
5. Iman Poernomo and John N. Crossley. Protocols between programs and proofs. In Kung-Kiu Lau, editor, *Logic Based Program Synthesis and Transformation, 10th International Workshop, LOPSTR 2000 London, UK, July 24-28, 2000, Selected Papers*, volume 2042 of *Lecture Notes in Computer Science*. Springer-Verlag, 2001.
6. Alex K. Simpson. Compositionality via cut-elimination: Hennessy-milner logic for an arbitrary gsos. In *LICS 1995, Proceedings 10th Annual IEEE Symposium on Logic in Computer Science, San Diego, California, USA, 26-29 June 1995*, pages 420–430. IEEE Computer Society Press, 1995.

A Clausal View for Access Control and XPath Query Evaluation

Barbara Fila, Siva Anantharaman

LIFO - Université d'Orléans (France),
e-mail: {fila, siva}@univ-orleans.fr

Abstract. Any positive XPath query Q (in a suitable format) can be evaluated on any given XML-like document t, under unambiguous runs of a transition system S_Q that we associate with Q. The transitions of S_Q on t are expressed as clauses. Query evaluation can be subject to access control policies enforced on the documents; the policies themselves are formulated as clauses, possibly subject to some constraints.
Keywords: XML, XPath, query, access control, constraints, clauses.

1 Introduction

We briefly recall some details on the XPath language (cf. [10]). We are concerned in this work with XPath queries involving the navigational axes of XPath as well as the data at the nodes on documents (given as unranked trees). We shall assume the queries to be *positive*, to simplify, in the sense that no negation is allowed on the navigational axes (but is allowed on the data filter parts of the query). The following XPath axes – referred to as *basic* in the sequel – will be used for navigation: `self`, `child`, `parent`, `ancestor`, `descendant`, `following-sibling`, `preceding-sibling`. (The other axes of XPath can be described in terms of these basic axes, up to a notion of equivalence; cf. e.g., [4].)

An alphabet Σ is assumed given for naming the nodes (or tags) on the documents; its elements will be referred to as nodenames or tagnames. *Att* will stand for the set of all attribute names at the nodes of all possible documents; and $att, att_1, \ldots$ will stand for variables running over *Att*. All data (including PCDATA) at the nodes on the documents will be assumed given in the form $att = `val'$ where *val* stands for a value assignable to *att*.

A *filter type* is an expression generated by the non-terminal F in the following grammar – where `A` stands for a basic XPath axis among the seven mentioned above, and $\sigma \in \Sigma \cup \{*\}$ ('$*$' stands here for an 'arbitrary' element of Σ), and *op* is an operator in the set $Op = \{=, \neq, >, <, \geq, \leq\}$:

$L ::= @\,att\ op\ `val'\ \mid\ position() = i\ \mid\ true$
$S ::= \texttt{A::}\sigma\ \mid\ \texttt{A::}\sigma\texttt{[}L\texttt{]}\ \mid\ \texttt{A::}\sigma\texttt{[}L\texttt{][}S\texttt{]}$
$G ::= S\ \mid \texttt{A::}\sigma\texttt{[}L\texttt{][}G\texttt{]}\ \mid\ G\ or\ G\ \mid\ G\ and\ G$
$F ::= L\ \mid\ G$

The expressions of the form `[`F`]` where F is a filter type, are said to be *filter expressions* or just *filters*. A filter expression of the form `[`L`]`, with no

navigational axes, will be said to be *local*. We shall identify `A::`σ`[`*true*`]` with `A::`σ. Given a context node u on any XML document t, any given filter expression `[`F`]` evaluates either to 'true' or to 'false' at u on t; cf. e.g., [10].

Positive XPath *query expressions* are defined as the expressions Q generated by the grammar below – where `A`, σ are as above, and `[`F`]` is any filter expression:

Q_0 ::= `/A::`σ | `/A::`σ`[`F`]` | $Q_0\,Q_0$

Q' ::= Q_0 | Q_0`/attribute::`*att* | Q_0`/attribute::`$*$

Q ::= Q' | Q' *or* Q'

(Note: `attribute` is a non-navigational axis of XPath, and `/@`*att* is short for the syntax `/attribute::`*att*; the '$*$' in the last production stands for 'any'.) The query expressions generated by this grammar are either of the form `/`C_1`/`$\ldots$`/`C_n, – each C_i being of the type `A::`σ`[`F`]` –, or a disjunction of such queries; they will be referred to as *canonical*. All our queries in the sequel will be assumed to be canonical. The filter expressions in the C_i are referred to as their *filter components*. A query of the form `/axis::`σ`[`F`]` will be said to be *elementary*. A *location expression* (resp. *location step*) is an expression of the form `[axis::`σ`[`F`]]` (resp. `axis::`σ`[`F`]`) where `axis` is one of the seven (basic) navigational axes mentioned above, $\sigma \in \Sigma$, and `[`F`]` is a filter. If the filter is local, the location expression (resp. location step) will be said to be *atomic*. Any location step will be written in the form `axis::`σ`[`L`][`F`]`, where `[`L`]` is local, and the navigational axes are all in `[`F`]`.

The notion of selection of any given node v, on a document t, wrt the *Root* node of t, by any given query expression Q, is defined inductively; cf. e.g., [10]. (*Root* is the fictive root node assigned in XML to any document t, just above the actual root node of the tree t; we shall denote this latter as *root*.) For any node u on a document t, we denote by $Data_t(u)$ the data stored at u on t, and set $t(u)$ to be the pair $(\sigma, Data_t(u))$, where σ is the tagname of t at u. If $Q =$`/`C_1`/`$\ldots$`/`C_n is any query in canonical form, the *answer* for Q on t is the set $Eval_t(Q)$ of all $t(u)|_Q$, where u is any node selected by Q on t, and $t(u)|_Q$ is the projection of $t(u)$ on the set of attributes selected by C_n.

Remark. a) We shall also need two further navigational axes that are only implicitly defined in XPath, namely the right-sibling and left-sibling axes; we shall refer to these as `right,left`, respectively; by definition, we have (cf. [10]):

`right::`σ is equivalent to `following-sibling::`σ`[`$position() = 1$`]`,

`left::`σ is equivalent to `preceding-sibling::`σ`[`$position() = 1$`]`.

b) To each `axis` among the seven basic axes above, we associate a step-wise axis denoted as `dir-axis`, the role of which is to move from node to node (on any tree t), along the direction given by `axis`. `self` is its own step-wise axis; here is the correspondence table for the other axes:

axis	dir-axis
parent	parent
child	child
descendant	child

axis	dir-axis
ancestor	parent
following-sibling	right
preceding-sibling	left

c) To any given basic XPath axis `axis`, we associate a (Boolean valued) unary predicate `Fin-axis()`, which evaluates to 'true' at any node u on any given XML document t, iff there exists no node v such that u `dir-axis` v holds on t.

(We define `Fin-self()` to be always 'true'.) For an `axis` $\neq$ `self` it evaluates to 'true' at the *root*, or at the leafs, or at the right-most or left-most nodes on t.

Any (positive XPath) query Q will be seen as a concatenation of its steps of certain types; to each of which is associated a transition system, and the transition system S_Q for Q will be the join of all these individual systems (Section 2). The evaluation of Q on any given document t is obtained under *step-wise* runs of S_Q on t; and each step-wise run will be formulated in terms of clauses. The runs of S_Q on t are defined in such a way that we get a linear complexity bound wrt the number of atomic location steps in Q, and the number of edges on t. Such a step-wise evaluation of any query Q allows us to integrate access control policies specified on any given document t, into the evaluation mechanism: the conditions specified by the policies are expressed as first-order clauses over the attribute names and/or attribute values (Section 3).

2 The Transition System for Evaluating a Query

Let $Q = /C_1/C_2/\ldots/C_n$ be a canonical query where each $/C_i$ is of the form $/C_i = /\texttt{axis}_i::\sigma_i\texttt{[}L_i\texttt{]}\texttt{[}F_i\texttt{]}$, $1 \leq i \leq n$, with a local filter `[`L_i`]`, and a filter `[`F_i`]` containing all the navigational part. For redactional simplicity, we assume that the filter components of Q are free from conjunction and disjunction.

Case of an atomic elementary query: We first consider the case where $/C_i$ is of the form $/C_i = /\texttt{axis}_i::\sigma_i\texttt{[}L_i\texttt{]}$, with a local filter `[`L_i`]`.

The elementary transition system (ETS) for the atomic elementary query $C_i = /\texttt{axis}_i::\sigma_i\texttt{[}L_i\texttt{]}$, is defined as the system S_i whose set of states is $States_i = \{init_i, ok_i, fail_i\}$, and whose transitions are defined by the following clauses $t1, \ldots, t6$ – where $x = \sigma_i\texttt{[}L_i\texttt{]}$, and $y = \overline{\sigma_i\texttt{[}L_i\texttt{]}}$ stands for the complement of the data represented by $\sigma_i\texttt{[}L_i\texttt{]}$:
If $\texttt{axis}_i \in \{$`self, child, parent`$\}$:

$t1$. $\langle ok_i, v\rangle \leftarrow \langle init_i, u\rangle$, if $(t(v) \models x)$, $(u\ \texttt{dir-axis}_i\ v)$;
$t2$. $\langle fail_i, v\rangle \leftarrow \langle init_i, u\rangle$, if $(t(v) \models y)$, $(u\ \texttt{dir-axis}_i\ v)$;

If $\texttt{axis}_i \notin \{$`self, child, parent`$\}$:

$t3$. $\langle ok_i, v\rangle \leftarrow \langle init_i, u\rangle$, if $(t(v) \models x)$, $(u\ \texttt{dir-axis}_i\ v)$;
$t4$. $\langle fail_i, v\rangle \leftarrow \langle init_i, u\rangle$, if $(t(v) \models y)$, $(u\ \texttt{dir-axis}_i\ v)$;
$t5$. $\langle fail_i, v\rangle \leftarrow \langle fail_i, u\rangle$, if $(t(v) \models y)$, $(u\ \texttt{dir-axis}_i\ v)$;
$t6$. $\langle ok_i, v\rangle \leftarrow \langle fail_i, u\rangle$, if $(t(v) \models y)$, $(u\ \texttt{dir-axis}_i\ v)$;

The role of the system S_i for $/C_i$, is to select the nodes on any given document t answering the part $/C_1/\ldots/C_i$ of the given query Q; this is done with the help of a run of S_i on t. The *run* of S_i on t is defined as a mapping $M_i\colon Nodes(t) \rightarrow \mathcal{P}(States_i)$, using the following transition rules, where u, v are nodes on t, and $\langle q, u\rangle$ stands for $q \in M_i(u)$:

$1a$. $\langle init_i, u\rangle \leftarrow u = Root$, if $i = 1$;
$\langle init_i, u\rangle \leftarrow \langle ok_{i-1}, u\rangle$, if $i > 1$;
$2a$. $\langle init_i, u\rangle \leftarrow \langle ok_i, u\rangle$, if $\texttt{axis}_i \notin \{$`self, child, parent`$\}$.

Case of a non-atomic elementary query: Consider now an elementary query $/C_i = /\texttt{axis}_i^0\texttt{::}\sigma_i^0\texttt{[}L_i^0\texttt{]}\texttt{[}F_i\texttt{]}$; it can be written under the form $/C_i = step_i^0\texttt{[}step_i^1\texttt{[}step_i^2\texttt{[}\ldots\texttt{[}step_i^{k(i)}\texttt{]}\ldots\texttt{]}$, for some positive integer $k(i)$, and each $step_i^p = \texttt{axis}_i^p\texttt{::}\sigma_i^p\texttt{[}L_i^p\texttt{]}$ is atomic, for $0 \leq p \leq k(i)$. The transition system S_i for such an elementary subquery $/C_i$ is defined as the 'concatenation' of *one-step transition systems* (STS, for short) S_i^p corresponding to $step_i^p$. In the sequel, for every $p \in \{0, \ldots, k(i)\}$, x will stand for $\sigma_i^p\texttt{[}L_i^p\texttt{]}$, and y for $\overline{\sigma_i^p\texttt{[}L_i^p\texttt{]}}$. Depending on the role played by the location step $step_i^p = \texttt{axis}_i^p\texttt{::}\sigma_i^p\texttt{[}L_i^p\texttt{]}$, we have three different types of STS; in each case, the transitions are similar to those of the atomic case; only the state sets will be different (cf. [4] for full details):

- S_i^0 corresponds to the step $/step_i^0\texttt{[}$; the states here are $init_i^0, fail_i^0, ok_i^0[-]$.
- For $1 \leq p < k(i)$, the system S_i^p corresponds to the step $\texttt{[}step_i^p\texttt{[}$; here the states are $init_i^p[-], fail_i^p[-], fail_i^p[\bot], ok_i^p[-]$.
- $S_i^{k(i)}$ for $\texttt{[}step_i^{k(i)}\texttt{]}$; its states are $init_i^{k(i)}[-], fail_i^{k(i)}[-], fail_i^{k(i)}[\bot], ok_i^{k(i)}[\top]$.

The system S_i^0 finds the nodes v *potentially selected* by $/C_1/\ldots/C_i$, and the systems S_i^p, for $1 \leq p \leq k(i)$, *validate* (or invalidate) the selections of S_i^0.

The system S_i, for $/C_i = /step_i^0\texttt{[}step_i^1\texttt{[}step_i^2\texttt{[}\ldots\texttt{[}step_i^{k(i)}\texttt{]}\ldots\texttt{]}$, is defined as the union of the systems S_i^p, for $0 \leq p \leq k(i)$; its set of states is defined as:

$States_i = \{init_i^0, fail_i^0\} \bigcup_{p=0}^{k(i)-1}\{ok_i^p[\gamma] \mid \gamma \in \{-, \bot, \top\}\}$
$\bigcup_{p=1}^{k(i)}\{init_i^p[\gamma], fail_i^p[\gamma] \mid \gamma \in \{-, \bot, \top\}\} \cup \{ok_i^{k(i)}[\top]\}.$

For any given document t, the *run* of S_i on t, is a mapping $M_i\colon Nodes(t) \to \mathcal{P}(States_i)$, under the transitions $1-11$ below, all formulated in terms of clauses. We first introduce some notation used in these transitions: we consider a 3-valued logic over the truth table $\{1, 0, \omega\}$ with $0 < \omega < 1$; the ω here stands for 'undefined', and we set: $\overline{1} = 0, \overline{0} = 1, \overline{\omega} = \omega$. For every $p \in \{0, \ldots, k(i)\}$ we define a (3-valued) unary predicate Ω_i^p, which will be evaluated recursively, at any node u of any given document t, as follows:

i) Case where $\texttt{Fin-axis}_i^p(u)$ is true:
$\Omega_i^p(u) = 1$ iff a state of the form $q_i^p[\top]$ is in $M_i(u)$;
$\Omega_i^p(u) = 0$ iff $M_i(u)$ has a state of the form $q_i^p[\bot]$;
$\Omega_i^p(u) = \omega$ otherwise.
ii) Case where $\texttt{Fin-axis}_i^p(u)$ is false:
$\Omega_i^p(u) = Sup_v\{\Omega_i^p(v) \mid u\ \texttt{dir-axis}_i^p\ v\}$

Transition rules for M_i (u, v are nodes on t, $p \in \{0, \ldots, k(i)\}$, $\langle s, u\rangle$ stands for $s \in M_i(u)$, $q, q' \in \{init, fail, ok\}$):

1. $\langle init_i^0, u\rangle \leftarrow u = Root$, if $i = 1$;
   $\langle init_i^0, u\rangle \leftarrow \langle ok_{i-1}, u\rangle$, if $i > 1$ (rule to go from S_{i-1} to S_i)
2. $\langle init_i^{p+1}[-], u\rangle \leftarrow \langle ok_i^p[-], u\rangle$, if $p \leq k(i) - 1$ (rule to go from S_i^p to S_i^{p+1})
3. $\langle q_i^p[\top], u\rangle \leftarrow \langle q_i^p[-], u\rangle, \Omega_i^p(u)$, for $p \geq 1$
   (back-propagate $[\top]$ along path traversed)
4. $\langle ok_i^{p-1}[\top], u\rangle \leftarrow \langle ok_i^{p-1}[-], u\rangle, \langle init_i^p[\top], u\rangle$ (signal filter 'true' to $step_i^{p-1}$)
5. $\langle ok_i, u\rangle \leftarrow \langle ok_i^0[\top], u\rangle$ (validate potential selection at u)

6. $\langle init_i^0, u\rangle \leftarrow \langle ok_i, u\rangle$, (continue with M_i from a validated node u)
   if $\texttt{axis}_i \notin \{\texttt{self, child, parent}\}$
7. $\langle fail_i^p[\bot], u\rangle \leftarrow \langle fail_i^p[-], u\rangle$, $\texttt{Fin-axis}_i^p(u)$, if $p \geq 1$
   (we are at an 'end', $step_i^p$ is false)
8. $\langle q_i^p[\bot], u\rangle \leftarrow \langle q_i^p[-], u\rangle$, $\overline{\Omega_i^p(u)}$, if $p \geq 1$
   (back-propagate $[\bot]$ along path traversed)
9. $\langle ok_i^{p-1}[\bot], u\rangle \leftarrow \langle ok_i^{p-1}[-], u\rangle$, $\langle init_i^p[\bot], u\rangle$, (signal filter 'false' to $step_i^{p-1}$)
   if $\texttt{axis}_i^{p-1} \in \{\texttt{self, child, parent}\}$
10. $\langle init_i^{p-1}[-], u\rangle \leftarrow \langle ok_i^{p-1}[-], u\rangle$, $\langle init_i^p[\bot], u\rangle$,
    if $\texttt{axis}_i^{p-1} \notin \{\texttt{self, child, parent}\}$
    (filter false at u for $step_i^p$, continue with $step_i^{p-1}$)
11. $\langle init_i^0, u\rangle \leftarrow \langle ok_i^0[\bot], u\rangle$, (continue with M_i from an invalidated node u)
    if $\texttt{axis}_i \notin \{\texttt{self, child, parent}\}$

Rules $1, 2$ are the 'start' transitions; validating a potentially selected node is done by rules $3-6$; rules $7-11$ take care of invalidating potential selections (see [4] for the details on the semantics of runs).

The transition system S_Q for the full query $Q = /C_1/C_2/\ldots/C_n$, is then defined as the system whose set of states is $States_Q = \bigcup_{i=1}^n States_i$. A run M_Q of S_Q on any given document t, is defined as a mapping $M_Q \colon Nodes(t) \to \mathcal{P}(States_Q)$, with $M_Q = \bigcup_{i=1}^n M_i$, with the following two additional transitions:

12. $\langle init_1, u\rangle \leftarrow \langle ok_n, u\rangle$, if $\texttt{axis}_1 \notin \{\texttt{self, child, parent}\}$
13. $\langle init_1^0, u\rangle \leftarrow \langle ok_n, u\rangle$, if $\texttt{axis}_1^0 \notin \{\texttt{self, child, parent}\}$

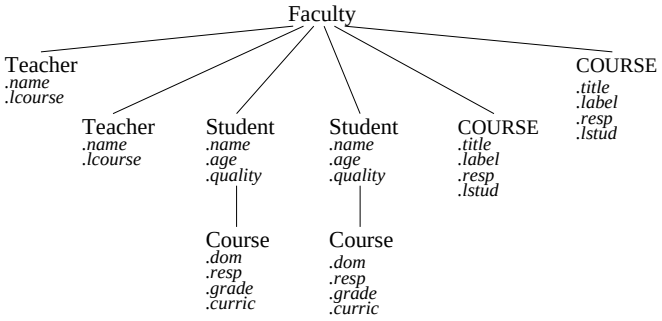
The answer for Q on t is the set of all nodes u of t, such that $ok_n \in M_Q(u)$. We have then the following result (cf. [4], for the proof):

Proposition 1. *The complexity of the step-wise evaluation of any XPath query Q on any XML document t by the run of S_Q, is linear on the number of atomic location steps in Q and the number of edges on t.*

3 Access Control

Various methods have been proposed for controlling the access to data (cf. e.g., [6]). The approach of [8] allots access keys to nodes, or more generally to subsets of attributes at the nodes; the keys could depend on a notion of category to which the consultant belongs. In [3] is suggested a somewhat different approach: to every category of consultants, associate a set of first-order clauses with constraints. [9] proposes to attach some special labels to the nodes of the document, such as $-r, -R$, to signify 'local' (resp. 'recursive') access denial. Our proposal in this paper is to model access control policies as first-order clauses. (It is shown in [4], that such a view can cover all the approaches mentioned above.) The clauses modeling access control will be subject, in general, to some constraints referred to as *scope constraints*, as illustrated in in the following example:

Example 1. Suppose given an (XML-format) database "Faculty", with as children nodes "Teacher", "Student", and "COURSE".



Here is an access control policy on this document, formulated in plain text, for two categories of users consulting such a document: *Adm* (resp. *Acad*) for administrative (resp. academic) staff.

Adm: a user of this category may not have access both to the name of a student and to the grade obtained in any of the course the student follows; but access to either one of these two data is allowed.

Acad: a user of this category is allowed access to the name of a student, and to the grade obtained by the student in any course, *iff* the user is the person giving the course.

We consider any category as a unary predicate, evaluating to 'true' only on the identifiers assigned to each user of the category. The policy for *Adm* can then be modeled as the following pure negative clause:

(1): $\leftarrow Adm(*),$ Student.name, Course.grade,
[Student.name `child` Course.grade]

where '$*$' stands for any identifier for a user of the category *Adm*. The intended meaning is that, for a user of category *Adm*, the policy is violated if the knowledge (s)he gains by consulting the database (possibly repeatedly), contains the name of a student, as well as the grade obtained by the student in a course; the scope constraint (in square brackets) says that the attributes 'name' and 'grade' are at two `child`-related nodes with tagnames 'Student' and 'Course', respectively.

The current state of data, stored at a node v, disclosed directly or indirectly to the user with identifier *id*, having launched a query Q (and possibly also some other queries prior to Q), will be represented by a set $Hist_{id}(v, Q)$ of pure positive clauses, containing at least the positive clause $Cat(id) \leftarrow$, where Cat stands for the category of the user. Suppose we are at a context node u on t, under the run of the transition system S_Q, and a node v is to be reached under the next transition of S_Q; this transition will correspond to a well-determined location step Q' of Q; a set $Deduce_{id}(v, Q')$ of positive clauses will then be constructed to model the knowledge that the user can acquire – directly or indirectly – on the data stored at v on t, by firing this transition. By definition, then, the run of S_Q on t will select the node v, if and only if:

$$Deduce_{id}(v, Q') \cup Hist_{id}(v, Q) \cup \{(1)\} \not\models \bot$$

where (1) is the negative clause above, modeling the access control policy for the category *Adm*. The positive clauses of $Deduce_{id}(v, Q')$ will all be added to the

history record of the user, for controlling the data (s)he accesses under future queries. The construction of the sets of clauses – such as $Hist_{id}()$ – needed for such a clause-based, step-wise evaluation of queries, is done below.

Definitions: We formalize here the clause-based view for access control.

Definition (1): By *User*, we mean a (finite) set of individual users; and by *Category* we shall mean a finite set of users, with a given specific name or label (as in the Example above). Both are seen as unary predicates: *User()*, *Cat()*. Consider a given document t, on which a given access control policy has been enforced; it will be assumed that every access condition of the policy is expressed as a pure negative clause, possibly subject to a scope constraint (as in the Example above); the set of all such negative (constrained) clauses will be denoted as $\mathcal{P}$. Every literal appearing in any of these clauses is therefore either a unary predicate (of the form *User()* or *Cat()*), or a propositional symbol of the form $u.a$ where u is a nodename, and a is an attribute name. The *scope constraint* of a clause in $\mathcal{P}$ is, by definition, a constraint of the form [*constr*] where *constr* is a (finite) conjunction of expressions of the form $u_i.a_i$ `axis`$_i$ $v_i.b_i$ (where `axis`$_i$ is a basic axis or a corresponding `dir-axis`), and/or of expressions of the form $u_j.a_j$ *op val*, where *val* is a data value, and $op \in Op$.

For instance, a policy clause of the form:

$\leftarrow$ *User*(10), u.a, v.b, w.c, [u.a `child` v.b, u.a `right` w.c, v.b $\neq$ 10]

stipulates that a user with identity 10 is not allowed access to the data stored by attribute a at node u, by attribute b at node v, and by attribute c at node w, all three together; by the "all three together", it is meant that such a combined access is disallowed to *User*(10), *under one or more queries* on the document.

Definition (2): At any node u of t, if a set $\mathcal{D}_u(t)$ of defining functional relations is specified on the data stored at u on t, then we also assume that $\mathcal{D}_u(t)$ is formulated as a set of Horn clauses of the form $u.a_{i_r} \leftarrow u.a_{i_1}, u.a_{i_2}, \ldots, u.a_{i_k}$, where the a_j are attribute names at u.

The document t being given, we shall drop references to t as indices (or otherwise), in the definitions below.

Definition (3): For any query $Q = /C_1/C_2/\ldots/C_n$ that is being evaluated by a run of its associated transition system S_Q on the given document t, suppose a transition of the run, from a context node u to a node v on t, is a transition of a one-step transition system S_k^p for some p, k; then, by the *current query-step* of Q at u, we mean the corresponding location step $step_k^p$ in Q (notation of the previous section).

Definition (4): For any given query Q launched by a given user with identity *id*, and a run of the associated system S_Q, we define three sets of positive clauses – denoted as $Hist_{id}(v, Q)$, $Scope_{id}(v, Q)$ and $Access_{id}(v, Q)$ – at every node v traversed by the run; these three sets are constructed, inductively in the following manner, wrt the the various queries successively launched by the user. Let $Q^{(0)}, Q^{(1)}, \ldots, Q^{(i)}, \ldots$ denote the sequence of the successive queries launched by the given user; the identity *id* of the user once fixed, we shall omit it (for readability), in the constructions below (in these constructions, by '*root*', we mean the actual root node of the tree t):

Step 0. Set $i = 0$.
Step 1. Case v is the root node of t: Define $Scope(root, Q^{(i)}) = \emptyset$;
if $i = 0$, then $History(root, Q^{(i)})$ is a singleton: $\{User(id) \leftarrow\}$ or $\{Cat(id) \leftarrow\}$;
if $i > 0$, $History(root, Q^{(i)})$ is defined as $Hist(root, Q^{(i-1)})$;
if $i = 0$, then $Access(root, Q^{(i)})$ is set to be empty;
if $i > 0$, $Access(root, Q^{(i)})$ is defined as $Hist(root, Q^{(i-1)})$.
Step 2. Case v is a non-root node:
Let u be the context node for the current transition of $S_{Q^{(i)}}$ to the node v. For all nodes u' traversed prior to u (including u), by the run of $S_{Q^{(i)}}$ evaluating $Q^{(i)}$ (i.e., the i-th query launched by the given user), assume having constructed the sets $Access(u', Q^{(i)})$, $Scope(u', Q^{(i)})$, and $Hist(u', Q^{(i)})$. The sets $Scope(v, Q^{(i)})$, $Access(v, Q^{(i)})$, and $Hist(v, Q^{(i)})$ are then constructed as follows:

1. Let `axis` be the axis of the current query-step at u, and suppose there is a node u' on t already traversed by the run such that u' `axis` v holds on t; if a (resp. att) is an attribute name at u' (resp. at v), such that $u'.a$ `axis` $v.att$ appears in the scope constraint of some policy clause, then create a positive 'scope clause' $[u'.a$ `axis` $v.att] \leftarrow$; and define $Scope(v, Q^{(i)})$ as: $Scope(v, Q^{(i-1)}) \cup \{[u'.a$ `axis` $v.att] \leftarrow\}$
2. For every $v.att$ appearing in the body of a policy clause in $\mathcal{P}$, such that $v.att$ is consistent with σ`[`L`]` (where the current query-step is of the form `axis::`σ`[`L`]`), create a positive 'data-access clause' $v.att \leftarrow$, and set
$Access(v, Q^{(i)}) = Access(v, Q^{(i-1)}) \cup \{v.att \leftarrow\}$.
3. Set $Hist(v, Q^{(i)}) = Hist(v, Q^{(i-1)}) \cup Scope(v, Q^{(i)}) \cup Access(v, Q^{(i)})$.

Step 3. Set $i = i + 1$, and GOTO **Step 1**.

Definition (5): In the presence of an access control policy $\mathcal{P}$ on t, a transition of the system S_Q, from a context node u to some node v on t, assigns a selecting state to the node v if and only if: $Hist(v, Q^{(i)}) \cup \mathcal{D}_v(t) \cup \mathcal{P} \not\models \perp$.

Effectively Using the Method: i) Clausal resolution is essential for the step-wise query evaluation technique proposed above. In addition to the usual first-order rules for resolving between positive and negative literals, we shall also need some additional rules, such as those in the following non-exhaustive list:

1. a literal $u.att$ can resolve with a literal (of the opposite sign) of the form $u.*$; the same also for literals (of opposite sign) of the form $User(id)$ and $User(*)$, etc.
2. a scope literal $[u.a$ `child` $v.b]$ can resolve with a scope literal of the form $[v.b$ `parent` $u.a]$, etc.
3. a negative scope literal $[u.a$ `axis` $v.b]$ can resolve with a positive scope literal of the form $[u.a$ `axis` $v.b$`,` $u.a \neq$ 'val'$]$

ii) Keeping the history records at the nodes is necessary if the access control policy is not to be violated; for details, see [4].

iii) The step-wise evaluation method described above, is *deductively complete* in the sense of the following proposition (its proof follows from definitions):

Proposition 2. *Let $\mathcal{P}$ be a set of negative clauses, implementing an access control policy for a given category of users, on a document t. Then a piece of data, stored on t by an attribute* att *at some node with tagname u on t, is accessible to a user of this category (under* some *XPath query)* iff *$u.att \leftarrow$ is element of a set $\mathcal{S}$ of positive data-access and scope clauses, which is consistent with $\mathcal{P}$.*

4 Related Works, Conclusion

We have modeled access control policies, as well as the knowledge that a person can acquire by querying the documents, as a set of first-order clauses. The idea of constructing, for any given query Q, a node selecting transition system S_Q has some similarity with the automaton associated with Q in [7]; but our system S_Q is not constructed in one single piece, and it also serves a very different purpose. [6] is a survey presenting the generalities on logical frameworks suited for access control, but the concern is not on query evaluation techniques. In [3], the focus is on a notion of non-interference of a given policy wrt a given query, and access control via clauses seems suggested only at the end. It is not difficult to extend the result of Proposition 1 in the presence of an access control policy, to get a complexity bound for query evaluation that is linear on the number of atomic location steps in Q, the number of edges of t, *and* the size of the data stored on t (which always bounds the current size of the history record). Finally, it seems possible to extend our approach to cover negation on the navigational part, by using suitable 'built-in' unary predicates, as in [5].

References

1. M. Abadi, *Logic in Access Control* In Proc. LICS'03, IEEE, pp. 228–233,
2. M. Abadi, B. Warinschi, *Security Analysis of Cryptographically Controlled Access to XML Documents* In Proc. of PODS'05, pp. 108–117, ACM, June 2005,
3. V. Benzaken, M. Burelle, G. Castagna, *Information Flow Security for XML Transformations*, In Proc. of ASIAN'03, pp. 33–53, LNCS 4246, Springer-Verlag, 2003,
4. B. Fila, S. Anantharaman, *A Clausal View for Access Control and XPath Query Evaluation*, Research Report available at: `www.univ-orleans.fr/lifo/prodsci/rapports/RR/RR2007/RR-2007-12.ps.gz`
5. M. Frick, M. Grohe, C. Koch, *Query Evaluation on Compressed Trees*, In Proc. of LICS'03, IEEE, pp. 188–197.
6. I. Fundulaki, M. Marx, *Specifying Access Control Policies for XML Documents with XPath* In Proc. of SACMAT'04, pp. 61–69, ACM, 2004.
7. T. Green, G. Miklau, M. Onizuka, D. Suciu, *Processing XML Streams with Deterministic Automata*, In ACM Trans. on Database Systems, 29(4):752–788, 2004.
8. G. Miklau, D. Suciu, *Controlling access to published data using cryptography*, In Proc. of VLDB'03, pp. 898–909, 2003.
9. M. Murata, A. Tozawa, M. Kudo, *XML Access Control Using Static Analysis*, In Proc. of the 10th ACM Conf. on Computer and Communications Security (CCS'03), pp.73–84, ACM, 2003.
10. World Wide Web Consortium, *XML Path Language (XPath Recommendation)*, Available at: `http://www.w3c.org/TR/xpath/`

Action Refinement in Process Algebra and Security Issues*

Annalisa Bossi[1], Carla Piazza[2], and Sabina Rossi[1]

[1] Dipartimento di Informatica, Università Ca' Foscari di Venezia, Italy
{bossi,srossi}@dsi.unive.it
[2] Dipartimento di Matematica e Infomatica, Università di Udine, Italy
carla.piazza@dimi.uniud.it

Abstract. In the design process of distributed systems we may have to replace abstract specifications of components by more concrete specifications, thus providing more detailed design information. In the context of process algebra this well-known approach is often referred to as *action refinement*. In this paper we study the relationships between action refinement and security properties within the Security Process Algebra.

1 Introduction

In the development of a complex system it is common practice first to describe it succinctly as a simple abstract specification and then to stepwise refine it towards a more concrete implementation [14].

In the context of process algebra, the refinement methodology amounts to defining a mechanism for replacing abstract actions with more concrete processes. We adopt the terminology *action refinement* to refer to this stepwise development of systems specified as terms of a process algebra (see, e.g., [9]). In this context, action refinement is also referred as *vertical refinement* as opposed to *horizontal refinement* indicating any transformation of a system making it more nearly executable, for instance more deterministic, without adding new actions or expanding sub-computations. The latter is usually expressed in terms of pre-orders such as trace inclusion or various kinds of simulation. We studied the relationships between this second form of refinement and information flow security in [2]. However these results cannot be used to deal with vertical refinement since the two forms of refinement provide orthogonal mechanisms for program development.

We model action refinement as a ternary function *Ref* taking as arguments an action r to be refined, a system description E on a given level of abstraction and an interpretation of the action r on this level by a more concrete process F on a lower abstraction level. The refined process, denoted by $Ref(r, E, F)$, is

* This work has been partially supported by the MIUR projects "Fondamenti Logici dei Sistemi Distribuiti e Codice Mobile" (grant 2005015785) and "Vincoli per la programmazione con insiemi, l'analisi di sistemi con automi, il ragionamento su intervalli e la bioinformatica" (grant 2005015491).

intended to be obtained from E by expanding each occurrence of r in E through F. This operation can be easily realized in a language including a sequential composition operator, but in this paper we consider the *Security Process Algebra* (SPA) language introduced in [5], which is a variant of CCS, and does not provide a sequential composition operator. To overcame this problem, we define action refinement of SPA processes by exploiting the notion of *well-terminating* process and the *Before* operator introduced by Milner in [12].

The main motivation behind our approach is that of studying the relationships between action refinement and security of SPA processes. Indeed, in system development, it is important to consider security related issues from the very beginning. A security-aware stepwise development requires that the security properties of interest are either preserved or gained during the development steps, until a concrete (i.e., implementable) specification is obtained. In this paper we consider *information flow security* properties (see, e.g., [8, 5, 10]), i.e., properties that allow one to express constraints on how information should flow among different groups of entities. These properties are usually formalized by considering two groups of entities labelled with two security levels: *high* (H) and *low* (L). The only constraint is that no information should flow from H to L. For example, to guarantee confidentiality in a system, it is sufficient to label every confidential (i.e., secret) information with H and then partition each system user as H and L, depending on whether such a user is authorized to access confidential information. The constraint of no information flow from H to L guarantees that no access to confidential information is possible by L-labelled users.

In [3] we studied persistent information flow security properties for the SPA language. These properties are obtained as instances of a *generalized unwinding condition* which requires that each high level action is "simulated" in such a way that it is impossible for the low level user to infer which high level actions have been performed. This general framework allows us to uniformly deal with some decidable subclasses of the well-known *NDC* and *BNDC* properties for SPA processes defined in [5]. The fact that we do not modify our language to introduce action refinement allows us to reason on the relationships between action refinement and the security properties of SPA processes. In particular, we study the conditions under which our notions of security are preserved under action refinement.

2 The SPA Language

The *Security Process Algebra* (SPA) language [5] is a variation of Milner's CCS [12] where the set of visible actions is partitioned into two security levels, high and low, in order to specify multilevel systems. SPA syntax is based on the same elements as CCS, i.e.: a set $\mathcal{L} = I \cup O$ of *visible* actions where $I = \{a, b, \ldots\}$ is a set of *input* actions and $O = \{\bar{a}, \bar{b}, \ldots\}$ is a set of *output* actions; a special action τ which models internal computations, not visible outside the system; a function $\bar{\cdot} : \mathcal{L} \rightarrow \mathcal{L}$, such that $\bar{\bar{a}} = a$, for all $a \in \mathcal{L}$. $Act = \mathcal{L} \cup \{\tau\}$ is the set of

$$\frac{}{a.E \xrightarrow{a} E} \qquad \frac{E_1 \xrightarrow{a} E_1'}{E_1 + E_2 \xrightarrow{a} E_1'} \qquad \frac{E_2 \xrightarrow{a} E_2'}{E_1 + E_2 \xrightarrow{a} E_2'}$$

$$\frac{E_1 \xrightarrow{a} E_1'}{E_1|E_2 \xrightarrow{a} E_1'|E_2} \qquad \frac{E_2 \xrightarrow{a} E_2'}{E_1|E_2 \xrightarrow{a} E_1|E_2'} \qquad \frac{E_1 \xrightarrow{l} E_1' \quad E_2 \xrightarrow{\bar{l}} E_2'}{E_1|E_2 \xrightarrow{\tau} E_1'|E_2'}$$

$$\frac{E \xrightarrow{a} E'}{E \setminus v \xrightarrow{a} E' \setminus v} \text{ if } a, \bar{a} \notin v \qquad \frac{E \xrightarrow{a} E'}{E[f] \xrightarrow{f(a)} E'[f]} \qquad \frac{T[recZ.T[Z]] \xrightarrow{a} E'}{recZ.T[Z] \xrightarrow{a} E'}$$

Fig. 1. The operational semantics of SPA terms.

all *actions*. The set of visible actions is partitioned into two sets, H and L, of high security actions and low security actions such that $\overline{H} = H$ and $\overline{L} = L$.

The syntax of SPA *terms* is as follows[3]:

$$T ::= \mathbf{0} \mid Z \mid a.T \mid T + T \mid T|T \mid T \setminus v \mid T[f] \mid recZ.T$$

where Z is a variable, $a \in Act$, $v \subseteq \mathcal{L}$, $f : Act \to Act$ is such that $f(\bar{l}) = \overline{f(l)}$ for $l \in \mathcal{L}$, $f(\tau) = \tau$, $f(H) \subseteq H \cup \{\tau\}$, and $f(L) \subseteq L \cup \{\tau\}$. We apply the standard notions of *free* and *bound* (occurrences of) variables in a SPA term. More precisely, all the occurrences of the variable Z in $recZ.T$ are *bound*; while an occurrence of Z is *free* in a term T if it is not bound. A SPA *process* is a SPA term without free variables.

We introduce also a notion of *bound* and *free* actions. We say that an action a is *bound* in a term T if it belongs to a restriction, i.e., $\setminus v$ occurs in T and $a \in v$, or is used in a relabelling operator, i.e., f occurs in T and $f(a) \neq a$ or $f(b) = a$ for some action $b \neq a$. We identify SPA terms up to α-conversion, thus we can assume that a bound action can occur only in a restriction or a relabelling operator or in their scopes. Hence, as a general assumption, we assume that the set of actions occurring in a term T can be split into two disjoint sets: the set $bound(T)$ of actions which are bound in T and the set $free(T)$ of actions which are not bound in T.

The operational semantics of SPA processes is given in terms of *Labelled Transition Systems* (LTS) as depicted in Figure 1 where $a \in Act$ and $l \in \mathcal{L}$.

The concept of *behavioural equivalence* is used to establish equalities among processes and it is based on the idea that two processes have the same semantics if and only if their behaviour cannot be distinguished by an external observer. We recall here the definition of *strong bisimulation* [12], which equates two processes when they are able to mutually simulate their behaviour step by step.

[3] Actually in [5] recursion is introduced through constant definitions instead of the *rec* operator.

Definition 1. (Strong Bisimulation) *A symmetric binary relation $\mathcal{R} \subseteq \mathcal{E} \times \mathcal{E}$ over processes is a* strong bisimulation *if $(E, F) \in \mathcal{R}$ implies, for all $a \in Act$, if $E \xrightarrow{a} E'$, then there exists F' such that $F \xrightarrow{a} F'$ and $(E', F') \in \mathcal{R}$.*

Two processes E and F are strongly bisimilar*, denoted by $E \sim F$, if there exists a strong bisimulation $\mathcal{R}$ containing the pair (E, F).*

The free occurrences of variables in a SPA term can be seen as holes in which other SPA terms can be inserted. The result of this substitution is still a SPA term. For instance, in the term $h.\mathbf{0}|(l.X + \tau.\mathbf{0})$ we can replace the variable X with the process $\bar{h}.\mathbf{0}$ obtaining the process $h.\mathbf{0}|(l.\bar{h}.\mathbf{0} + \tau.\mathbf{0})$; or we can replace X by the term $a.Y$ obtaining the term $h.\mathbf{0}|(l.a.Y + \tau.\mathbf{0})$. A SPA term with free variables is called *context*[4]. Given the context $C[Y_1, \ldots, Y_n]$ with free variables $Y_1, \ldots, Y_n$, we denote by $C[T_1, \ldots, T_n]$ the term obtained from $C[Y_1, \ldots, Y_n]$ by simultaneously replacing all the free occurrences of $Y_1, \ldots, Y_n$ with the terms $T_1, \ldots, T_n$, respectively.

Finally, observe that our calculus does not provide a sequential composition operator. However, following Milner [12], we can define it by introducing the convention that processes indicate their termination by a distinguished label $\overline{\texttt{done}}$.

Definition 2. (Strongly Well-terminating process) *Let F be a SPA process. F is* strongly well-terminating *if for all $F' \in Reach(F)$: (1) $F' \xrightarrow{\texttt{done}}$ is impossible; (2) if $F' \xrightarrow{\alpha} \sim \mathbf{0}$ then $F' \sim \overline{\texttt{done}}.\mathbf{0}$; (3) if $F' \xrightarrow{\overline{\texttt{done}}}$ then $F' \sim \overline{\texttt{done}}.\mathbf{0}$.*

When F is strongly well-terminating, the sequential composition of processes F and E can be defined through the operator *Before* introduced by Milner in [12].

Definition 3. (Before operator) *Let E and F be SPA processes such that F is strongly well-terminating.*

$$Before[F, E] \stackrel{\text{def}}{=} (F[\bar{f}/\overline{\texttt{done}}]|f.E) \setminus \{f\}$$

where $\bar{f}/\overline{\texttt{done}}$ denotes the relabelling function replacing $\overline{\texttt{done}}$ with a new name $\bar{f}$.

3 Action Refinement

It is standard practice in software development to obtain the final program by first defining an abstract, possibly not executable, specification and then refining it until one arrives to a concrete specification that can directly be implemented. Abstract operations are replaced by more detailed programs which can possibly be further refined. In the context of process algebra, this stepwise development amounts to interpreting actions on a higher level of abstraction by more complex processes on a lower level. This is obtained by introducing a mechanism to transform actions into processes. There are several ways to do this. Here we follow a syntactic approach defining the refinement as a syntactic process transformation.

First, we need to specify which are the processes F that can be used to refine a process E and which are the actions r refinable in E.

[4] Notice that a SPA term denotes either a process or a context.

Definition 4. (Replaceable process and Refinable actions) *Let E and F be SPA terms and $r \in \mathcal{L}$.*

- *The process F is* replaceable *in E if:* (1) $bound(E) \cap free(F) = bound(F) \cap free(E) = \emptyset$*;* (2) *$F$ is not the process $\mathbf{0}$;* (3) *F is strongly well-terminating.*
- *The action r is said to be* refinable in E with F *if:* (1) *F is replaceable in E;* (2) $r \notin bound(E)$*;* (3) *r does not occur in F.*

The notion of *action refinement* for SPA processes is defined by structural induction on the term to be refined by using in the main basic case the operator *Before*.

Definition 5. (Action Refinement) *Let E, F be SPA terms such that r is an action refinable in E with F. The* refinement of r in E with F *is the term $Ref(r, E, F)$ inductively defined on the structure of E as follows:*

$$
\begin{array}{ll}
(1) & Ref(r, \mathbf{0}, F) \stackrel{\text{def}}{=} \mathbf{0} \\
(2) & Ref(r, Z, F) \stackrel{\text{def}}{=} Z \\
(3) & Ref(r, r.E_1, F) \stackrel{\text{def}}{=} Before[F, Ref(r, E_1, F)] \\
(4) & Ref(r, a.E_1, F) \stackrel{\text{def}}{=} a.Ref(r, E_1, F), \ \text{if } a \neq r \\
(5) & Ref(r, E_1[f], F) \stackrel{\text{def}}{=} Ref(r, E_1, F)[f] \\
(6) & Ref(r, E_1 \setminus v, F) \stackrel{\text{def}}{=} Ref(r, E_1, F) \setminus v \\
(7) & Ref(r, E_1 + E_2, F) \stackrel{\text{def}}{=} Ref(r, E_1, F) + Ref(r, E_2, F) \\
(8) & Ref(r, E_1 | E_2, F) \stackrel{\text{def}}{=} Ref(r, E_1, F) | Ref(r, E_2, F) \\
(9) & Ref(r, recZ.E_1, F) \stackrel{\text{def}}{=} recZ.Ref(r, E_1, F)
\end{array}
$$

Point (3) of the above definition deals with the basic case in which we replace an occurrence of r with the refining process F. In all the other cases the refinement process enters inside the components of E.

Example 1. Let $E \stackrel{\text{def}}{=} r.a.\mathbf{0}$ and F be a strongly well-terminating process such that r is refinable in E with F. Then $Ref(r, E, F) \stackrel{\text{def}}{=} Before[F, Ref(r, a.\mathbf{0}, F)] \stackrel{\text{def}}{=} Before[F, a.\mathbf{0}]$ represents the process which first behaves as F and then, when the execution of F is terminated, proceeds as $a.\mathbf{0}$.

From now on when we write $Ref(r, E, F)$ we tacitly assume that r is refinable in E with F.

Our refinement operation satisfies the following compositional properties.

Lemma 1. *Let $E_1, \ldots, E_n$ and F be SPA terms and $r \in \mathcal{L}$ be refinable with F in $E_1, \ldots, E_n$. Let $C[Z_1, \ldots, Z_n]$ be a SPA context with no occurrences of r. Then $Ref(r, C[E_1, \ldots, E_n], F) \stackrel{\text{def}}{=} C[Ref(r, E_1, F), \ldots, Ref(r, E_n, F)]$.*

Lemma 2. *Let E, F_1, F_2 be SPA terms and r_1 and r_2 be two actions refinable in E with F_1 and F_2, respectively. If r_1 does not occur in F_2 and r_2 does not occur in F_1 then*

- $Ref(r_2, Ref(r_1, E, F_1), F_2) \stackrel{\text{def}}{=} Ref(r_1, Ref(r_2, E, F_2), F_1)$,
- $Ref(r_2, Ref(r_1, E, F_1), F_2) \sim Ref(r_1, Ref(r_2, E, F_2), Ref(r_2, F_1, F_2))$.

4 Preserving Security Properties under Refinement

In this section we first present some information flow security properties for SPA processes. Then we investigate conditions under which our notions of security are preserved under action refinement.

Security Properties. Information flow security in a multilevel system aims at guaranteeing that no high level (confidential) information is revealed to users running at low security levels [7, 11], even in the presence of any possible malicious process (attacker).

In [5] Focardi and Gorrieri introduce the properties *Non-Deducibility on Compositions* (*NDC*) and *Bisimulation-based Non-Deducibility on Compositions* (*BNDC*) in order to capture every possible information flow from a *classified* (*high*) level of confidentiality to an *untrusted* (*low*) one. The definitions of *NDC* and *BNDC* are based on the basic idea of Non-Interference [8]: "No information flow is possible from high to low if what is done at the high level *cannot interfere* in any way with the low level". More precisely, a system E is secure if what a low level user sees of the system is not modified by composing any high process Π to E. The concept of *low observation* is expressed in terms of an *equivalence relation on low level actions* between processes. The idea is that two systems cannot be distinguished by a low level observer if and only if they are equated by an equivalence relation considering low level actions only. The two properties *NDC* and *BNDC* differ only on the low level observation equivalence they consider. *NDC* is based on *trace equivalence on low actions*, denoted by $\approx^l_T$, while *BNDC* considers the notion of *weak bisimilarity on low actions*, denoted by $\approx^l_B$. The definitions of weak bisimilarity on low actions and trace equivalence on low actions are the same as the definitions of weak bisimilarity and trace equivalence except that low and silent actions only (belonging to the set $L \cup \{\tau\}$), instead of all actions (belonging to the set Act), are considered.

Properties *BNDC* and *NDC* are thus formally defined as follows:

$$E \in BNDC \text{ if for all high level process } \Pi,\ \ E \approx^l_B (E|\Pi)$$
$$E \in NDC \text{ if for all high level process } \Pi,\ \ E \approx^l_T (E|\Pi)$$

Since weak bisimilarity on low actions is stronger than trace equivalence on low actions, it holds that *BNDC* implies *NDC*.

Properties *NDC* and *BNDC* are difficult to use in practice: *NDC* is not decidable in polynomial time, while the decidability of *BNDC* is still an open problem. In [6], Focardi and Rossi introduce the property *Persistent BNDC* (*P_BNDC*) which is a natural persistent extension of *BNDC* (i.e., a system E is *P_BNDC* if every state E' reachable from E is *BNDC*) and it is a sufficient condition for *BNDC*. They show the decidability of *P_BNDC* by exploiting a bisimulation based characterization. Other persistent security properties have been later introduced, e.g., the properties *Persistent NDC* (*P_NDC*) in [4] and *Compositional P_BNDC* (*CP_BNDC*) in [1].

All the *persistent* properties mentioned above can be defined as instances of a *generalized unwinding condition* [1] which requires that each high level action

is "simulated" in such a way that it is impossible for the low level user to infer which high level actions have been performed [13]. The generalized unwinding condition is parametric with respect to two binary relations on processes: an equivalence relation on low actions, $\backsim^l$, which represents the low level view, and a transition relation, $\dashrightarrow$, which characterizes a local connectivity.

Definition 6. (Generalized Unwinding) *Let $\sim^l$ be an equivalence relation on low actions and $\dashrightarrow$ be a binary relation on processes. The* unwinding class $\mathcal{W}(\backsim^l, \dashrightarrow)$ *is defined as*

$$\mathcal{W}(\backsim^l, \dashrightarrow) \stackrel{\text{def}}{=} \{E \in \mathcal{E} \mid \forall\ F, G \in Reach(E) \\ \textit{if } F \stackrel{h}{\rightarrow} G \textit{ then } \exists G' \textit{ such that } F \dashrightarrow G' \textit{ and } G \backsim^l G'\}.$$

It holds that P_NDC coincides with $\mathcal{W}(\approx^l_T, \stackrel{\hat{\tau}}{\Longrightarrow})$ [4], P_BNDC coincides with $\mathcal{W}(\approx^l_B, \stackrel{\hat{\tau}}{\Longrightarrow})$ and CP_BNDC coincides with $\mathcal{W}(\approx^l_B, \stackrel{\tau}{\Longrightarrow})$ [1], where $\stackrel{\hat{\tau}}{\Longrightarrow}$ denotes a sequence of zero or more τ transitions while $\stackrel{\tau}{\Longrightarrow}$ denotes a sequence of at least one τ transition. We have $P\_NDC \subseteq NDC$ and $P\_BNDC, CP\_BNDC \subseteq BNDC$.

Example 2. Let us consider a distributed data base (adapted from [9]) which can take two values and which can be both queried and updated. In particular, the high level user can query it through the high level actions qry_1 and qry_2, while the low level user can only update it through the low level actions upd_1 and upd_2. Hence $qry_1, qry_2 \in H$ and $upd_1, upd_2 \in L$. We can model the data base with the SPA process E defined as

$$E \stackrel{\text{def}}{=} recZ.(qry_1.Z + upd_1.Z + \tau.Z + \\ upd_2.recW.(qry_2.W + upd_2.W + \tau.W + upd_1.Z)).$$

The process E is P_BNDC. Indeed, whenever a high level user queries the data base with a high level action moving the system to a state X then a τ action moving the system to the same state X may be performed, thus masking the high level interactions with the system to low level users. □

Classes of Secure Processes closed under action refinement. We now investigate conditions under which our notions of security are preserved under action refinement. In particular, we are interested in the definition of classes of processes satisfying an instance of $\mathcal{W}(\sim^l, \dashrightarrow)$ and closed under action refinement. We first introduce the concept of $(\mathcal{P}, r)$-refinable context, where $\mathcal{P}$ is a process property and r is an action. Intuitively, a class of contexts is $(\mathcal{P}, r)$-refinable if it contains all the processes satisfying $\mathcal{P}$ and is closed under refinement of the action r.

Definition 7 ($(\mathcal{P}, r)$-refinable contexts). *Let $\mathcal{P}$ be a class of processes and r be an action. A class $\mathcal{C}$ of contexts is said to be a class of* $(\mathcal{P}, r)$-refinable contexts *if:*

- *if $E \in \mathcal{C}$ and E is a process, then $E \in \mathcal{P}$;*

– *if* $E, F \in \mathcal{C}$ *and* r *is refinable in* E *with* F *then* $Ref(r, E, F) \in \mathcal{C}$.

Given a sequence $s = s_1, s_2, \ldots, s_n$ of actions and a set v of actions, we denote by $s.E$ the process $s_1.s_2.\ldots.s_n.E$ and by $s \cap v$ the set of actions occurring both in s and in v. Moreover, given a relabelling f we denote by $f[s]$ the set $\{s_i \mid f(s_i) \neq s_i\}$. Let $\dashrightarrow$ be a binary relation on processes we say that s *entails* $\dashrightarrow$ if $E \xrightarrow{s} E'$ implies $E \dashrightarrow E'$.

Definition 8. **($\mathcal{C}_s(\dashrightarrow)$)** *Let* $s \in (L \cup \{\tau\})^*$ *be a sequence of low and silent actions and* $\dashrightarrow$ *be a binary relation on processes such that* s entails $\dashrightarrow$.
$\mathcal{C}_s(\dashrightarrow)$ *is the class of contexts containing: the process* $\mathbf{0}$*;* Z*, where* Z *is a variable;* $l.C_1$*,* $h.C_1 + s.C_1$*,* $C_1 \setminus v$*,* $C_1[f]$*,* $C_1 + C_2$*,* $C_1|C_2$*, and* $recZ.C_1$*, with* $l \in L \cup \{\tau\}$*,* $h \in H$*,* $s \cap v = \emptyset$*,* $f[s] = \emptyset$*, and* $C_1, C_2 \in \mathcal{C}_s(\dashrightarrow)$.

We are now ready to state the main results of this work. The next theorem says that for any property $\mathcal{P}$ derived from our generalized unwinding (see Definition 6), all the contexts of Definition 8 are $(\mathcal{P}, r)$-refinable. The subsequent corollary provides a sufficient condition to preserve the security property $\mathcal{P}$ under refinement.

Theorem 1. *Let* $\mathcal{W}(\sim^l, \dashrightarrow)$ *be an unwinding condition. Let* r *be an action which does not occur in* s*. It holds that the class* $\mathcal{C}_s(\dashrightarrow)$ *is* $(\mathcal{W}(\sim^l, \dashrightarrow), r)$*-refinable.*

Corollary 1. *Let* E, F *be terms and* r *be a refinable action in* E *with* F*. If* $E, F \in \mathcal{C}_s(\dashrightarrow)$ *and* r *does not occur in* s*, then* $Ref(r, E, F) \in \mathcal{W}(\sim^l, \dashrightarrow)$ *and* $Ref(r, E, F) \in \mathcal{C}_s(\dashrightarrow)$.

Example 3. Consider again the abstract specification of the distributed data base represented through the SPA process E of Example 2. The process E belongs to the class $\mathcal{C}_\tau(\stackrel{\tau}{\Longrightarrow})$ of Definition 8. In fact, $C_1 \stackrel{\text{def}}{=} qry_2.W + upd_2.W + \tau.W + upd_1.Z \in \mathcal{C}_\tau(\stackrel{\tau}{\Longrightarrow})$, so $C_2 \stackrel{\text{def}}{=} recW.C_1 \in \mathcal{C}_\tau(\stackrel{\tau}{\Longrightarrow})$. Hence, $C_3 \stackrel{\text{def}}{=} qry_1.Z + upd_1.Z + \tau.Z + upd_2.C_2 \in \mathcal{C}_\tau(\stackrel{\tau}{\Longrightarrow})$. Thus $E \stackrel{\text{def}}{=} recZ.C_3 \in \mathcal{C}_\tau(\stackrel{\tau}{\Longrightarrow})$.

We can refine the update actions by requiring that each update is requested and confirmed, i.e., we refine upd_1 using $F_1 \stackrel{\text{def}}{=} req_1.cnf_1.\overline{\texttt{done}}.\mathbf{0}$ and upd_2 using $F_2 \stackrel{\text{def}}{=} req_2.cnf_2.\overline{\texttt{done}}.\mathbf{0}$, where $req_1, cnf_1, req_2, cnf_2$ are low security level actions. By automatically applying Definition 5 we obtain:

$$Ref(upd_2, Ref(upd_1, E, F_1), F_2) \stackrel{\text{def}}{=} \\ req_2.cnf_2.\tau.recW.(qry_2.W + req_2.cnf_2.\tau.W + \tau.W + req_1.cnf_1.\tau.Z)).$$

Since F_1 and F_2 are in $C_\tau(\stackrel{\tau}{\Longrightarrow})$, by applying Corollary 1 we have that the process $Ref(upd_2, Ref(upd_1, E, F_1), F_2)$ is in $\mathcal{W}(\approx^l_B, \stackrel{\tau}{\Longrightarrow})$, i.e., it is P_BNDC.

References

1. A. Bossi, R. Focardi, D. Macedonio, C. Piazza, and S. Rossi. Unwinding in Information Flow Security. *Electronic Notes in Theoretical Computer Science*, 99:127–154, 2004.
2. A. Bossi, R. Focardi, C. Piazza, and S. Rossi. Refinement Operators and Information Flow Security. In *Proc. of the 1st IEEE Int. Conference on Software Engineering and Formal Methods (SEFM'03)*, pages 44–53. IEEE Computer Society Press, 2003.
3. A. Bossi, R. Focardi, C. Piazza, and S. Rossi. Verifying Persistent Security Properties. *Computer Languages, Systems and Structures*, 30(3-4):231–258, 2004.
4. A. Bossi, C. Piazza, and S. Rossi. Modelling Downgrading in Information Flow Security. In *Proc. of the 17th IEEE Computer Security Foundations Workshop (CSFW'04)*, pages 187–201. IEEE Computer Society Press, 2004.
5. R. Focardi and R. Gorrieri. Classification of Security Properties (Part I: Information Flow). In R. Focardi and R. Gorrieri, editors, *Proc. of Foundations of Security Analysis and Design (FOSAD'01)*, volume 2171 of *LNCS*, pages 331–396. Springer-Verlag, 2001.
6. R. Focardi and S. Rossi. Information Flow Security in Dynamic Contexts. In *Proc. of the IEEE Computer Security Foundations Workshop (CSFW'02)*, pages 307–319. IEEE Computer Society Press, 2002.
7. S. N. Foley. A Universal Theory of Information Flow. In *Proc. of the IEEE Symposium on Security and Privacy (SSP'87)*, pages 116–122. IEEE Computer Society Press, 1987.
8. J. A. Goguen and J. Meseguer. Security Policies and Security Models. In *Proc. of the IEEE Symposium on Security and Privacy (SSP'82)*, pages 11–20. IEEE Computer Society Press, 1982.
9. R. Gorrieri and A. Rensink. Action Refinement. Technical Report UBLCS-99-09, University of Bologna (Italy), 1999.
10. H. Mantel. Possibilistic Definitions of Security - An Assembly Kit -. In *Proc. of the IEEE Computer Security Foundations Workshop (CSFW'00)*, pages 185–199. IEEE Computer Society Press, 2000.
11. J. McLean. Security Models and Information Flow. In *Proc. of the IEEE Symposium on Security and Privacy (SSP'90)*, pages 180–187. IEEE Computer Society Press, 1990.
12. R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.
13. A. W. Roscoe and M. H. Goldsmith. What is intransitive noninterference? In *Proc. of the IEEE Computer Security Foundations Workshop (CSFW'99)*, pages 228–238. IEEE Computer Society Press, 1999.
14. N. Wirth. Program Development by Stepwise Refinement. *Communications of the ACM*, 14(4):221–227, 1971.