Technical Communications of the 27th International Conference on Logic Programming

ICLP'11, July 6–10, 2011, Lexington, Kentucky, USA

Edited by

John P. Gallagher
Michael Gelfond

Editors

John P. Gallagher	Michael Gelfond
Roskilde University	Department of Computer Science
CBIT, Building 43.2	Texas Tech University, College of Engineering
Universitetsvej 1	Box 43104
4000 Roskilde, Denmark	Lubbock, Tx 79409, US
jpg@ruc.dk	michael.gelfond@ttu.edu

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

ISBN 978-3-939897-31-6

Published online and open access by
Schloss Dagstuhl – Leibniz-Center for Informatics GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. Online available at http://www.dagstuhl.de/dagpub/978-3-939897-31-6.

Publication date
July, 2011

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

License

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

- Attribution: The work must be attributed to its authors.
- Noncommercial: The work may not be used for commercial purposes.
- No derivation: It is not allowed to alter or transform this work.

The copyright is retained by the corresponding authors.

Digital Object Identifier: 10.4230/LIPIcs.ICLP.2011.i

LIPIcs – Leibniz International Proceedings in Informatics

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

Editorial Board

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

ISSN 1868-8969

www.dagstuhl.de/lipics

Contents

Preface
John P. Gallagher and Michael Gelfond i

Technical Papers

Multi-Criteria Optimization in Answer Set Programming
Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub 1

Yet Another Characterization of Strong Equivalence
Alexander Bochman and Vladimir Lifschitz 11

Evolution of Ontologies using ASP
Max Ostrowski, Giorgos Flouris, Torsten Schaub, and Grigoris Antoniou 16

Modelling Grammar Constraints with Answer Set Programming
Christian Drescher and Toby Walsh 28

Hybrid ASP
Alex Brik and Jeffrey B. Remmel 40

Representing the Language of the Causal Calculator in Answer Set Programming
Michael Casolary and Joohyung Lee 51

Static Type Checking for the Q Functional Language in Prolog
Zsolt Zombori, János Csorba, and Péter Szeredi 62

Canonical Regular Types
Ethan K. Jackson, Nikolaj Bjørner, and Wolfram Schulte 73

Compiling Prolog to Idiomatic Java
Michael Eichberg 84

Synthesis of Logic Programs from Object-Oriented Formal Specifications
Ángel Herranz and Julio Mariño 95

An Inductive Approach for Modal Transition System Refinement
Dalal Alrajeh, Jeff Kramer, Alessandra Russo, and Sebastian Uchitel 106

Constraints in Non-Boolean Contexts
Leslie De Koninck, Sebastian Brand, and Peter J. Stuckey 117

Minimizing the overheads of dependent AND-parallelism
Peter Wang and Zoltan Somogyi 128

Smart test data generators via logic programming
Lukas Bulwahn 139

Declarative Output by Ordering Text Pieces
Stefan Brass 151

Technical Communications of the 27th International Conference on Logic Programming (ICLP'11).
Editors: John P. Gallagher and Michael Gelfond

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

Transaction Logic with Defaults and Argumentation Theories
Paul Fodor and Michael Kifer 162

Multi-agent Confidential Abductive Reasoning
Jiefei Ma, Alessandra Russo, Krysia Broda, and Emil Lupu 175

BAAC: A Prolog System for Action Description and Agents Coordination
Agostino Dovier, Andrea Formisano, and Enrico Pontelli 187

Declarative Processing of Semistructured Web Data
Michael Hanus 198

CDAOStore: A Phylogenetic Repository Using Logic Programming and Web Services
Brandon Chisham, Enrico Pontelli, Tran Cao Son, and Ben Wright 209

Bayesian Annotation Networks for Complex Sequence Analysis
Henning Christiansen, Christian Theil Have, Ole Torp Lassen and Matthieu Petit 220

Improving the Outcome of a Probabilistic Logic Music System Generator by Using Perlin Noise
Colin J. Nicholson, Danny De Schreye, and Jon Sneyers 231

Abduction in Annotated Probabilistic Temporal Logic
Cristian Molinaro, Amy Sliva, and V. S. Subrahmanian 240

Doctoral Consortium

Automatic Parallelism in Mercury
Paul Bone 251

Consistency Techniques for Hybrid Simulations.
Marco Bottalico 255

Extensions of Answer Set Programming
Alex Brik 261

A Semiring-based framework for fair resources allocation
Paola Campli 268

Promoting Modular Nonmonotonic Logic Programs
Thomas Krennwallner 274

Correct Reasoning about Logic Programs
Jael Kriener 280

Accepting the natural order of rules in a logic program with preferences
Alexander Šimko 284

Implementation of Axiomatic Language
Walter W. Wilson 290

Two Phase Description Logic Reasoning for Efficient Information Retrieval
Zsolt Zombori 296

Preface

Following the initiative in 2010 taken by the Association for Logic Programming and Cambridge University Press, the full papers accepted for the International Conference on Logic Programming again appear as a special issue of Theory and Practice of Logic Programming (TPLP) and shorter papers appear in Leibniz International Proceedings in Informatics (LIPIcs) series, published on line through the Dagstuhl Research Online Publication Server (DROPS). Both sets of papers were presented by their authors at the 27th ICLP. Together, the journal special issue and the volume of short technical communications constitute the *proceedings* of ICLP.

Papers describing original, previously unpublished research and not simultaneously submitted for publication elsewhere were solicited in all areas of logic programming including but not restricted to: *Theory* (Semantic Foundations, Formalisms, Non-monotonic Reasoning, Knowledge Representation), *Implementation* (Compilation, Memory Management, Virtual Machines, Parallelism), *Environments* (Program Analysis, Transformation, Validation, Verification, Debugging, Profiling, Testing), *Language Issues* (Concurrency, Objects, Coordination, Mobility, Higher Order, Types, Modes, Assertions, Programming Techniques), *Related Paradigms* (Abductive Logic Programming, Inductive Logic Programming, Constraint Logic Programming, Answer-Set Programming), and *Applications* (Databases, Data Integration and Federation, Software Engineering, Natural Language Processing, Web and Semantic Web, Agents, Artificial Intelligence, Bioinformatics).

There were four broad categories for submissions: (1) technical papers for describing technically sound, innovative ideas that can advance the state of the art of logic programming; (2) application papers, where the emphasis is on their impact on the application domain; (3) system and tool papers, where the emphasis is on the novelty, practicality, usability and general availability of the systems and tools described; and (4) technical communications, aimed at describing recent developments, new projects, and other materials that are not ready for main publication as standard papers. The length limit for full papers was set at 15 pages plus bibliography for full papers (approximately in line with the length of TPLP technical notes) and for technical communications at 10 pages total.

In response to the call for papers we received 67 submissions. Of those, 64 were full papers submitted to the TPLP special issue track (21 of them applications or systems papers). The program chairs acting as guest editors organized the refereeing process with the help of the program committee and numerous external reviewers.1 Each paper was reviewed by at least three anonymous referees who provided full written evaluations. After the first round of refereeing 43 full papers remained. Of these, 23 went through a full second round of refereeing with written referee reports. Finally, all 43 papers went through a final, copy-editing round. In the end the special issue contains 19 technical papers, 3 application papers, and 1 systems and tools paper. During the first phase of reviewing the papers submitted to the technical communications track were also reviewed by at least three anonymous referees providing full written evaluations. Also, a number of full paper submissions were moved during the reviewing process to the technical communications track. Finally, 23 papers were accepted as technical communications and are published in this volume. The list of the 23 accepted full papers appearing in the TPLP special issue follows:

Technical Communications of the 27th International Conference on Logic Programming (ICLP'11).
Editors: John P. Gallagher and Michael Gelfond

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

Regular Papers

Complex Optimization in Answer Set Programming
Martin Gebser, Roland Kaminski and Torsten Schaub

On the Correctness of Pull-Tabbing
Sergio Antoy

(Co-)Inductive Semantics for Constraint Handling Rules
Rémy Haemmerlé

The Magic of Logical Inference in Probabilistic Programming
Bernd Gutmann, Ingo Thon, Angelika Kimmig, Maurice Bruynooghe and Luc De Raedt

RedAlert: Determinacy Inference for Prolog
Jael Kriener and Andy King

On Combining Linear-Based Strategies for Tabled Evaluation of Logic Programs
Ricardo Rocha and Miguel Areias

Estimating the overlap between dependent computations for automatic parallelization
Paul Bone, Zoltan Somogyi and Peter Schachte

Transition Systems for Model Generators — A Unifying Approach
Yuliya Lierler and Miroslaw Truszczynski

Efficient Instance Retrieval of Subgoals for Subsumptive Tabled Evaluation of Logic Programs
Flavio Cruz and Ricardo Rocha

Non-termination Analysis of Logic Programs with integer arithmetics
Dean Voets and Daniel De Schreye

A Structured Alternative to Prolog with Simple Compositional Semantics
António Porto

The PITA System: Tabling and Answer Subsumption for Reasoning under Uncertainty
Fabrizio Riguzzi and Terrance Swift

Normative Design using Inductive Learning
Domenico Corapi, Alessandra Russo, Marina De Vos, Julian Padget and Ken Satoh

Abstract Diagnosis of Timed Concurrent Constraint programs
Marco Comini, Laura Titolo and Alicia Villanueva

Parallel Backtracking with Answer Memoing for Independent And-Parallelism
Pablo Chico De Guzmán, Amadeo Casas, Manuel Carro and Manuel Hermenegildo

SAT-Based Termination Analysis Using Monotonicity Constraints over the Integers
Amir M. Ben-Amram, Michael Codish, Carsten Fuhs, Jürgen Giesl and Igor Gonopolskiy

Observational equivalences for Linear Logic CC languages
Rémy Haemmerlé

Splitting and Updating Hybrid Knowledge Bases
Martin Slota, Joao Leite and Terrance Swift

Actual causation in CP-logic
Joost Vennekens

Application Papers and Systems and Tools Papers

Automatic Network Reconstruction using ASP
Max Ostrowski, Torsten Schaub, Markus Durzinsky, Wolfgang Marwan and Annegret Wagler

Optimal Placement of Valves in a Water Distribution Network with CLP(FD)
Massimiliano Cattafi, Marco Gavanelli, Maddalena Nonato, Stefano Alvisi and Marco Franchini

Constraint-Based Deadlock Checking of High-Level Specifications
Michael Leuschel and Stefan Hallerstede

ALPprolog — A New Logic Programming Method for Dynamic Domains
Conrad Drescher and Michael Thielscher

In conclusion, we would like to thank the members of the Program Committee and the external referees for their enthusiasm, hard work, and promptness, despite the higher load of the two rounds of refereeing plus the copy editing phase. The PC members were: Slim Abdennadher, Marcello Balduccini, Chitta Baral, Maurice Bruynooghe, Manuel Carro, James Cheney, Henning Christiansen, Alessandro Dal Palù, Marc Denecker, Agostino Dovier, Esra Erdem, François Fages, John Gallagher, Martin Gebser, Michael Gelfond, Samir Genaim, Katsumi Inoue, Andy King, Evelina Lamma, Joohyung Lee, Nicola Leone, Michael Leuschel, Yuliya Lierler, Vladimir Lifschitz, Marco Maratea, Victor Marek, Davide Martinenghi, Alessandra Mileo, Emilia Oikarinen, Mauricio Osorio, Maurizio Proietti, German Puebla, Konstantinos Sagonas, Vítor Santos Costa, Tom Schrijvers, Alexander Serebrenik, Guillermo Simari, Zoltan Somogyi, Tran Cao Son, Hans Tompits, Francesca Toni, Mirek Truszczynski, Germán Vidal, Kewen Wang, Jan Wielemaker, Stefan Woltran and Jia-Huai You. We would also like to thank Georg Gottlob, Adam Lally, and Günter Kniesel for their invited talks, Michael A. Covington and Francesca Toni for their tutorials, the ICLP organizers Mirek Truszczynski and Victor Marek (General Chairs), Joohyung Lee (Workshops), Yuliya Lierler (Publicity), Alessandro Dal Palù and Stefan Woltran (Doctoral Consortium) and Tom Schrijvers (Prolog Programming Contest), and Lexmark, University of Kentucky, and Association for Logic Programming for supporting the conference.

Finally, we would like to express our thanks and great appreciation to Ilkka Niemelä, editor in chief of Theory and Practice of Logic Programming, David Tranah from Cambridge University Press, Marc Herbstritt from LIPIcs, Leibniz Center for Informatics, and all the members of the ALP Executive Committee for their continued support for this initiative, which provides a new model of computer science publishing that is already being adopted by other computing research communities.

John Gallagher and Michael Gelfond
Program Committee Chairs and Guest Editors

Multi-Criteria Optimization in Answer Set Programming

Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub

Institut für Informatik, Universität Potsdam

Abstract

We elaborate upon new strategies and heuristics for solving multi-criteria optimization problems via Answer Set Programming (ASP). In particular, we conceive a new solving algorithm, based on conflict-driven learning, allowing for non-uniform descents during optimization. We apply these techniques to solve realistic Linux package configuration problems. To this end, we describe the Linux package configuration tool *aspcud* and compare its performance with systems pursuing alternative approaches.

1998 ACM Subject Classification D.1.6 Logic Programming, I.2.3 Deduction and Theorem Proving, I.2.4 Knowledge Representation Formalisms and Methods

Keywords and phrases Answer Set Programming, Multi-Criteria Optimization, Linux Package Configuration

Digital Object Identifier 10.4230/LIPIcs.ICLP.2011.1

1 Introduction

Solving multi-criteria optimization problems is of great interest in various application domains because it allows for identifying the best solutions among all feasible ones. The quality of a solution is often associated with costs or rewards subject to minimization and/or maximization, respectively.

As detailed in the extended version of this paper (cf. [8]), we are interested in solving Linux package configuration problems by appeal to the multi-criteria optimization capacities of Answer Set Programming (ASP; [3]). To this end, we develop novel general-purpose strategies and heuristics in the context of modern (conflict-driven learning) ASP solving [10]. In particular, we conceive a new optimization algorithm allowing for non-uniform descents during optimization. In multi-criteria optimization, this enables us to optimize criteria in the order of significance, rather than pursuing a rigid lexicographical descent. We illustrate the impact of our contributions by appeal to our Linux package configuration tool *aspcud* and its performance in comparison with alternative approaches.

Pioneering work in this area was done by Tommi Syrjänen in [15, 16], using ASP for representing and solving configuration problems for the Debian GNU/Linux system. In fact, ASP allows for defining such problems through sequences of cost functions represented by (multi)sets of literals with associated weights. For instance, in the approach taken by *smodels* [13], cost functions are expressed through a sequence of `#minimize` (and `#maximize`) statements. Optimal models are then computed via a branch-and-bound extension to *smodels*' enumeration algorithm. Similarly, *dlv* [11] offers so-called weak constraints, serving the same purpose.

© Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub;
licensed under Creative Commons License NC-ND
Technical Communications of the 27th International Conference on Logic Programming (ICLP'11).
Editors: John P. Gallagher, Michael Gelfond; pp. 1–10

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

2 Background

The semantics of a (ground extended) logic program Π is given by particular models, called answer sets; see [13] for details. In addition to rules, Π can contain `#minimize` statements of the form

$$\texttt{\#minimize}[\ell_1 = w_1@L_1, \dots, \ell_n = w_n@L_n].$$

Besides literals ℓ_i and integer weights w_i for $1 \leq i \leq n$, a `#minimize` statement includes integers L_i providing priority levels [9]. The `#minimize` statements in Π distinguish optimal answer sets of Π in the following way. For any set X of atoms and integer L, let Σ_L^X denote the sum of weights w_i such that $\ell_i = w_i@L$ occurs in some `#minimize` statement in Π and ℓ_i holds wrt X. We also call Σ_L^X the utility of X at priority level L. An answer set X of Π is dominated if there is an answer set Y of Π such that $\Sigma_L^Y < \Sigma_L^X$ and $\Sigma_{L'}^Y = \Sigma_{L'}^X$ for all $L' > L$, and optimal otherwise. Note that greater priority levels are more significant than smaller ones, which allows for representing sequences of several optimization criteria. Finally, letting $\overline{\ell_i}$ denote the complement of a literal ℓ_i, the following can be used as a synonym for a `#minimize` statement: `#maximize`$[\overline{\ell_1} = w_1@L_1, \dots, \overline{\ell_n} = w_n@L_n]$.

3 Multi-Criteria Optimization Algorithm

As detailed in [13], `#maximize` statements can be turned into `#minimize` statements, literals with negative weights be transformed such that weights become positive, and multiple priority levels be collapsed into a single one by scaling the weights of literals, where all such transformations keep the optimal answer sets intact. However, while the elimination of `#maximize` statements and negative weights can be done locally, collapsing priority levels may lead to very large weights and also disguises an original multi-criteria optimization problem. Hence, we assume here that optimization criteria are represented in terms of a `#minimize` statement over literals associated with non-negative weights and, notably, priority levels; i.e., priorities are not eliminated. The restriction to non-negative weights has the advantages that the sum of weights is monotonically increasing the more literals are assigned to true and that 0 is a (trivial) lower bound of the optimum at each priority level.

As mentioned in the introduction, multi-criteria optimization can in principle be accomplished by extending a standard enumeration algorithm, like the one of *smodels* [13], in the following way: for every solution, memorize its vector of utilities, backtrack, and check (during propagation) that assignments generated in the sequel induce a lexicographically smaller vector of utilities (otherwise backtrack). This simple approach requires only the most recent utility vector to be stored, and optimality of the last solution is proven once the residual problem turns out to be unsatisfiable. But the simplicity comes along with the drawback that the number of intermediate solutions, encountered before an optimal one, is completely up to "luck" of the underlying enumeration algorithm. In fact, if no additional measures are taken, such multi-criteria optimization is logically identical to optimization of a single priority level along with scaled weights of literals.

The observation that plenty intermediate solutions improving only at low-priority utilities can gravely obstruct the convergence towards a global optimum gave the main impetus to our new approach to multi-criteria optimization in ASP. As noted in [2] for Maximum Satisfiability (MaxSAT) and Pseudo-Boolean Optimization (PBO), a better idea is to optimize priority levels stepwise in the order of significance, rather than to optimize all priority levels at once. Thereby, we adhere to the strategy of successively improving upper bounds given

by intermediate solutions. On the one hand, focusing on one priority level after the other settles the issue of intermediate solutions improving only at low-priority levels. On the other hand, it leads to the situation that, before optimization proceeds to the next priority level, optimality at the current level must be verified by proving unsatisfiability wrt an infeasible upper bound. Beyond the fact that accomplishing such unsatisfiability proofs can be a bottleneck (cf. [1]), they imply that too strong bounds need to be taken back before optimization can proceed at the next level. In particular, with solvers like *clasp* [10], exploiting conflict-driven learning, also the learned constraints that rely on an infeasible upper bound must be retracted. To this end, we make use of assumptions assigned at a solver's root level [6], i.e., unbacktrackable literals allowing for the selective (de)activation of constraints. In fact, a speculative upper bound is imposed via an assumption such that a corresponding constraint is not satisfied by making the assumption. If the upper bound turns out to be infeasible, the respective constraint and all learned information relying on it can then easily be discarded by irrevocably assigning the complement of the former assumption. Likewise, if the upper bound is feasible, the former assumption can be fixed, so that constraints involving it may be simplified and apply unconditionally in the sequel. In the following, we detail how (dedicated) multi-criteria optimization can be accomplished in modern (conflict-driven learning) Boolean constraint solvers, thereby exploiting assumptions to circumvent the need of a relaunch after an unsatisfiability proof.

Our algorithm augmenting conflict-driven learning (cf. [5, 12]) with multi-criteria optimization is shown in Algorithm 1. The sequence $\langle \mathbf{L}_1, \dots, \mathbf{L}_{low} \rangle$ determined in the first line contains the priority levels of the input `#minimize` statement in decreasing order of significance. The counters *assm*, *prio*, and *step*, initialized to 1 in the second line, are used to generate new assumptions on demand, to identify the current priority level to be optimized, and to determine the amount by which the upper bound ought to be decreased when a solution is found. The latter is always 1, thus yielding a linear decrease, if the input *leap* flag is *false*, while an exponential scheme (described below) is applied otherwise. Furthermore, the lower bound *lb*, set to 0 in the third line, stores the greatest value such that unsatisfiability has been proven for smaller bounds at the current priority level. In fact, the optimization of a priority level is finished once the utility of a solution matches the lower bound. In the loop in Line 4–45, the optimization-specific information, kept in counters and the lower bound, is used to guide conflict-driven search. As usual, the loop starts in Line 5 with a deterministic Propagate step, assigning literals implied by the current assignment. Afterwards, one of the following is the case: a conflict (Line 6–23), a solution (Line 24–44), or a heuristic decision (Line 45). While the latter simply leads to reentering the loop, the first two cases deserve more attention. We describe next the reaction to a solution and then the one to a conflict.

Upon encountering a solution, we start by checking whether its objective value at the current priority level provides us with a new (non-speculative) upper bound. This is clearly the case if the current solution is the first one, as tested via $assm = 1$ in Line 25, and setting *recd* to *true* informs our algorithm that the upper bound needs to be recorded before proceeding to the next priority level. On the other hand, if a speculative upper bound $ub_{prio}{-}step$ has already been imposed, the current solution witnesses that this bound is feasible. Hence, a respective optimization constraint is made unconditional by fixing the former assumption $\overline{\alpha_{assm}}$ in Line 27. In view of this, adding another constraint before proceeding to the next priority level is required only if the current solution's objective value is smaller than $ub_{prio}{-}step$, as tested in Line 28. The sequence $\langle ub_1, \dots, ub_{low} \rangle$ of upper bounds given by the current solution is memorized in Line 30 and printed along with an

Algorithm 1: CDNL-OPT

Input: A logic program Π, a statement $\#\texttt{minimize}[\ell_1 = w_1@L_1, \dots, \ell_n = w_n@L_n]$, and a flag *leap* $\in \{true, false\}$.

1. $\langle \mathbf{L}_1, \dots, \mathbf{L}_{low} \rangle \leftarrow \langle \max(\{L_1, \dots, L_n\} \setminus \{\mathbf{L}_1, \dots, \mathbf{L}_{m-1}\}) \rangle_{1 \leq m \leq |\{L_1,\dots,L_n\}|}$
2. $assm \leftarrow prio \leftarrow step \leftarrow 1$ — *// assumption, priority, and step counter*
3. $lb \leftarrow 0$ — *// lower bound*
4. **loop**
5.   Propagate — *// deterministically assign implied literals*
6.   **if** *conflict* **then**
7.   **if** at root level **then** — *// unsatisfiability modulo optimization constraint*
8.   **if** $assm = 1$ **then exit**
9.   Assign α_{assm} — *// deactivate old optimization constraint*
10. $lb \leftarrow (ub_{prio} - step) + 1$
11.   **while** $prio \leq low$ **and** $ub_{prio} = lb$ **do**
12.   **if** $recd = true$ **then** Add $\#\texttt{sum}[\ell_i = w_i \mid 1 \leq i \leq n, L_i = \mathbf{L}_{prio}]lb$
13. $lb \leftarrow 0$
14. $recd \leftarrow true$
15. $prio \leftarrow prio + 1$
16.   **if** $prio > low$ **then exit**
17. $step \leftarrow 1$
18. $assm \leftarrow assm + 1$
19.   Add $\big(\alpha_{assm} \vee \#\texttt{sum}[\ell_i = w_i \mid 1 \leq i \leq n, L_i = \mathbf{L}_{prio}]ub_{prio} - step\big)$
20.   Assume $\overline{\alpha_{assm}}$ — *// activate new optimization constraint*
21.   **else**
22.   Analyze — *// analyze conflict and add (violated) conflict constraint*
23.   Backjump — *// unassign literals until conflict constraint is unviolated*
24.   **else if** *solution* **then**
25.   **if** $assm = 1$ **then** $recd \leftarrow true$ — *// upper bound of witness yet unrecorded*
26.   **else**
27.   Assign $\overline{\alpha_{assm}}$ — *// fix old optimization constraint*
28.   **if** $(\Sigma_{1 \leq i \leq n, L_i = \mathbf{L}_{prio}, \ell_i \text{assigned to true}} w_i) < ub_{prio} - step$ **then** $recd \leftarrow true$
29.   **else** $recd \leftarrow false$
30. $\langle ub_1, \dots, ub_{low} \rangle \leftarrow \langle \Sigma_{1 \leq i \leq n, L_i = \mathbf{L}_m, \ell_i \text{assigned to true}} w_i \rangle_{1 \leq m \leq low}$
31.   **print** answer set along with $\langle ub_1, \dots, ub_{low} \rangle$
32. $prio' \leftarrow prio$
33.   **while** $prio \leq low$ **and** $ub_{prio} = lb$ **do**
34.   **if** $recd = true$ **then** Add $\#\texttt{sum}[\ell_i = w_i \mid 1 \leq i \leq n, L_i = \mathbf{L}_{prio}]lb$
35. $lb \leftarrow 0$
36. $recd \leftarrow true$
37. $prio \leftarrow prio + 1$
38.   **if** $prio > low$ **then exit**
39.   **if** $prio = prio'$ **and** $leap = true$ **then** $step \leftarrow \min\{2 * step, \lceil (ub_{prio} - lb)/2 \rceil\}$
40.   **else** $step \leftarrow 1$
41. $assm \leftarrow assm + 1$
42.   Add $\big(\alpha_{assm} \vee \#\texttt{sum}[\ell_i = w_i \mid 1 \leq i \leq n, L_i = \mathbf{L}_{prio}]ub_{prio} - step\big)$
43.   Assume $\overline{\alpha_{assm}}$ — *// activate new optimization constraint*
44.   Backjump — *// unassign literals until optimization constraint is unviolated*
45.   **else** Decide — *// non-deterministically assign some literal*

answer set of the input program Π in Line 31. Then, the loop in Line 33–37 proceeds to the next priority level to optimize, depending on whether the condition $ub_{prio} = lb$ holds in Line 33. If so, it means that the upper bound witnessed by the solution at hand matches the lower bound at a priority level, so that no further improvement is possible. Furthermore, if the current upper bound still needs to be recorded, a corresponding `#sum` constraint, as available in ASP input languages [14, 7], is added to the constraint database of the solver in Line 34; this makes sure that future solutions cannot exceed the lower bound lb at a forsaken priority level. Also note that lb is set to the minimum 0 in Line 35, so that proceeding by more than one priority level is possible only if some upper bound given by the solution at hand is trivially optimal. After finishing the loop in Line 33–37, multi-criteria optimization has been accomplished if the test $prio > low$ succeeds in Line 38, meaning that the utilities $\langle ub_1, \dots, ub_{low} \rangle$ cannot be improved. Otherwise, an amount by which the current upper bound ought to be decreased is determined in Line 39–40. If the priority level has not been changed and the *leap* flag is *true*, we take the minimum of the double former *step* size and half of the gap between the upper and lower bound as the amount by which to decrease the upper bound. This exponential scheme aims at balancing two objectives: try to skip non-optimal intermediate solutions while decreasing the upper bound, but do not provoke many unnecessary (and potentially hard) proofs of unsatisfiability. Given the next *step* size, an optimization constraint, being the disjunction of a fresh literal α_{assm} and a `#sum` constraint enforcing the new (speculative) upper bound, is added to the constraint database of the solver in Line 42, and $\overline{\alpha_{assm}}$ is assumed in Line 43, so that any further solution must fall below the speculative upper bound $ub_{prio} - step$. Finally, backjumping in Line 44 retracts literals (but not $\overline{\alpha_{assm}}$ assumed at the root level) in order to re-enable the search for solutions satisfying the new optimization constraint.

In case of a conflict, we distinguish whether it is encountered at the root level or beyond it. The latter means that the conflict is related to decisions made previously (in Line 45), so that regular conflict analysis and backjumping (cf. [5, 12]) can in Line 22–23 be applied to identify a reason in terms of a conflict constraint and to resume search at a point where the conflict constraint yields an implication. On the other hand, a conflict at the root level indicates unsatisfiability. Provided that $assm = 1$ does not hold in Line 8, i.e., if Π has some answer set, there is no solution meeting the upper bound $ub_{prio} - step$. This bound is imposed by the most recently added optimization constraint, which is in Line 9 retracted by assigning α_{assm}, thus withdrawing the former assumption and unconditionally satisfying the optimization constraint (as well as all conflict constraints relying on it). Furthermore, the unsatisfiability relative to the upper bound provides us with the lower bound $(ub_{prio} - step) + 1$, assigned to lb in Line 10. As in the case of a solution, the loop in Line 11–15 proceeds to the next priority level to optimize, where a gap between the lower and upper bound leaves room for improvements. If such a level *prio* exists, i.e., $prio > low$ does not hold in Line 16, the *step* size is reduced to 1 in Line 17, and the next optimization constraint along with a fresh assumption are put into effect in Line 18–20. By reducing the *step* size to the smallest value that would still improve ub_{prio}, we reset the exponential scheme applied if the input *leap* flag is *true*. This directs search to first check whether improvements are possible at all before reattempting to decrease the upper bound more aggressively.

Multi-criteria optimization via Algorithm 1 is implemented in *clasp* from version 2.0.0 on. We do not detail the implementation here, but mention matters of interest. To begin with, note that *clasp* stores a statement `#minimize`$[\ell_1 = w_1@L_1, \dots, \ell_n = w_n@L_n]$ in a single optimization constraint, using as data-structure a two-dimensional array of size $|\{L_1, \dots, L_n\}| * |\{\ell_1, \dots, \ell_n\}|$ with $w_1, \dots, w_n$ as its (non-zero) entries. Furthermore, the

vector $\langle ub_m \rangle_{1 \leq m \leq |\{L_1,\dots,L_n\}|}$ of upper bounds is initialized to $\langle \infty_m \rangle_{1 \leq m \leq |\{L_1,\dots,L_n\}|}$ and then updated whenever a solution is found. For one, this permits to accomplish the simple approach to multi-criteria optimization, described at the beginning of this section, via lexicographic comparisons without scaling weights in view of priority levels. For another, dedicated multi-criteria optimization wrt a current priority level *prio* merely requires to (temporarily) ignore upper bounds at less significant priority levels, thus providing easy means to strengthen the readily available optimization constraint by subtracting the value of *step* from ub_{prio} (cf. Line 19 and 42 of Algorithm 1). To further facilitate such steps, *clasp* includes a single assumption $\overline{\alpha}$ in its optimization constraint and, for the most significant priority level $L = \max\{L_1, \dots, L_n\}$, sets the weight $w@L$ of $\overline{\alpha}$ to $(\sum_{1 \leq i \leq n, L_i = L} w_i) + 1$. This makes sure that α belongs to every conflict constraint relying on the optimization constraint, so that these conflict constraints can be fixed (by discharging α) or withdrawn, respectively, immediately upon encountering either a solution or a conflict. To this end, *clasp* invokes the method `strengthenTagged()` when a solution is found and `removeTagged()` when a root-level conflict occurs, while keeping the assumption $\overline{\alpha}$ in place at the root level; applying either method turns $\overline{\alpha}$ into a fresh assumption without presuming any particular solver state, as otherwise required when performing constraint database simplifications.

The command-line parameters `--opt-hierarch` and `--opt-heuristic` allow for configuring (multi-criteria) optimization in *clasp*. If the value `0` is provided for the former, simple lexicographic optimization (without assumptions) is applied, while `1` and `2` switch to Algorithm 1 with the *leap* flag set to *false* and *true*, respectively. Furthermore, `--opt-heuristic` determines how `#minimize` statements are taken into account in *clasp*'s decision heuristics (Line 45 of Algorithm 1). While `0` falls back to the default heuristic, a static sign heuristic, preferably falsifying literals that occur in a `#minimize` statement, is applied for `1`. Value `2` switches to a dynamic heuristic that, after a solution has been found, falsifies its literals in a `#minimize` statement until a conflict is encountered. Finally, `3` combines `1` and `2`, thus falsifying literals subject to minimization if a respective variable is selected, while also picking such variables after a solution has been found (until hitting a conflict). The additional parameter `--restart-on-model` is a prerequisite for the values `2` and `3` to be effective; without it, they drop down to `0` and `1`, respectively.

4 Experiments

We developed the tool *aspcud*[1] applying our approach to multi-criteria optimization in ASP to Linux package configuration. At the start, *aspcud* translates a package configuration problem in Common Upgradability Description Format (CUDF; [17]) into ASP facts, described in the extended version of this paper [8]. The translation involves mapping CUDF package formulas to sets of packages (clauses) and tracing virtual packages that cannot directly be installed back to packages that implement them. Such flattening makes the problem encoding (cf. [8]) in ASP more convenient. Beyond syntactic simplifications, the translation by *aspcud* also exploits optimization criteria and package interdependencies to reduce the resulting ASP instance.

As ASP tools, *aspcud* (version 1.3.0) exploits *gringo* (version 3.0.3) for grounding and *clasp* (version 2.0.0-RC2) for solving. To illustrate the impact of the strategies and heuristics supported by *clasp*, our experiments consider several variants of it. Three settings are

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

obtained by configuring `--opt-hierarch` with the values described above, indicated by a subscript:

- $clasp_0$: optimizing whole utility vectors (as described at the beginning of Section 3 and implemented also in *smodels* as well as *clasp* versions below 2.0.0),
- $clasp_1$: applying Algorithm 1 with the *leap* flag set to *false*, and
- $clasp_2$: applying Algorithm 1 with the *leap* flag set to *true*.

We further combine each $clasp_i$ ($i \in \{0, 1, 2\}$) with optimization-oriented heuristics, activated by setting `--opt-heuristic` to the value indicated by a superscript:

- $clasp_i^0$: applying no optimization-specific decision heuristic,
- $clasp_i^1$: applying the static sign heuristic to falsify literals of a `#minimize` statement,
- $clasp_i^2$: falsifying literals of a `#minimize` statement that were contained in a recent solution, and
- $clasp_i^3$: combining the sign heuristic of $clasp_i^1$ with the dynamic approach of $clasp_i^2$.

We thus obtain twelve variants of *clasp*, each invoked with the (additional) command-line parameters `--sat-prepro`, `--heuristic=vsids`, `--restarts=128`, `--local-restarts`, and `--solution-recording`, which turned out to be helpful on large underconstrained optimization problems confronted in Linux package configuration. As mentioned above, $clasp_i^2$ and $clasp_i^3$ further require `--restart-on-model` to be effective, and we indicate the use of this parameter by writing $clasp_i^j$-r, where "-r" is mandatory for $j \in \{2, 3\}$ but optional for $j \in \{0, 1\}$. The reasonable combinations of the variable options amount to 18 variants of *clasp* to perform the optimization within *aspcud*.

For comparison, we also consider the package configuration tools *cudf2msu*[2] (version 1.0), *cudf2pbo*[3] (version 1.0), and *p2cudf*[4] (version 1.11). The PBO-based approaches of *cudf2pbo* and *p2cudf* are closely related to multi-criteria optimization in ASP via Algorithm 1, while the MaxSAT approach of *cudf2msu* utilizes unsatisfiable cores to iteratively refine lower bounds. The tools included for comparison belong to the leaders in a recent trial-run[5], called MISC-live, of the competition organized by mancoosi.

Table 1 reports experimental results on package configuration problems used in the recent MISC-live run,[6] divided by the tracks *paranoid*, *trendy*, and *user1–3*, each applying a different combination of optimization criteria. Note that the number of lexicographically ordered utilities is two in the *paranoid* track, three in the *user1* track, and four in the *trendy* and *user2–3* tracks. We ran the five criteria combinations on 117 instances considered in the *paranoid* and *trendy* tracks of the MISC-live run (all instances except for the ones in the "debian-dudf" category, which were not available for download). For each track, the column headed by S provides the sums of solvers' scores according to the MISC-live ranking: a solver that returns a solution earns $b + 1$ points, where b is the number of solvers that returned strictly better solutions; a solver that returns no solution earns $2 * s$ points, where s is the total number of participating solvers ($s = 21$ in our case); finally, a solver that crashes or returns a wrong solution (i.e., an invalid installation profile) is awarded $3 * s$ points (for s as before). Note that a smaller score is better than a greater one, and solvers are ranked by their scores in ascending order. The columns headed by T/O report total runtimes per solver

[2] `http://sat.inesc-id.pt/~mikolas/cudf2msu.html`
[3] `http://sat.inesc-id.pt/~mikolas/cudf2pbo.html`
[4] `http://wiki.eclipse.org/Equinox/p2/CUDFResolver`
[5] `http://www.mancoosi.org/misc-live/20101126`
[6] The results of (a preliminary version of) *aspcud* in this trial-run were scrambled due to scripting problems, which led to complete failure rather than a sub-optimal solution if an optimum could not be proven in time.

	paranoid		*trendy*		*user1*		*user2*		*user3*	
Solver	*S*	*T/O*	*S*	*T/O*	*S*	*T/O*	*S*	*T/O*	*S*	*T/O*
$clasp_0^0$-r	431	2,287/6	1730	23,829/ 80	935	14,349/35	525	5,097/12	1031	14,184/37
$clasp_0^0$	416	2,294/6	2375	29,781/105	1727	21,897/73	1224	14,697/45	671	11,178/21
$clasp_0^1$-r	**410**	2,210/6	1560	22,660/ 73	898	13,466/30	502	4,654/ 9	980	13,682/35
$clasp_0^1$	**410**	2,326/6	2079	26,471/ 92	1723	21,525/72	922	10,767/31	658	10,675/23
$clasp_0^2$-r	427	2,135/6	712	16,867/ 51	527	5,891/11	426	2,981/ 5	587	7,628/20
$clasp_0^3$-r	429	**2,134**/6	740	17,079/ 52	507	5,863/12	425	3,044/ 6	576	7,769/21
$clasp_1^0$-r	425	2,428/6	579	16,713/ 50	550	5,819/14	434	3,000/ 6	710	8,958/25
$clasp_1^0$	417	2,418/6	549	16,544/ 50	**475**	5,318/12	**411**	2,538/ 5	502	6,279/16
$clasp_1^1$-r	429	2,405/6	622	17,304/ 50	518	5,908/13	438	2,976/ 6	676	8,938/23
$clasp_1^1$	427	2,372/6	613	16,946/ 49	490	5,478/12	416	2,562/ 5	496	6,144/16
$clasp_1^2$-r	427	2,352/6	571	16,646/ 50	518	5,358/13	418	2,582/ 5	471	6,356/16
$clasp_1^3$-r	429	2,346/6	**547**	**16,386**/ 50	499	**5,306**/12	413	2,498/ 5	497	6,255/16
$clasp_2^0$-r	425	2,392/6	806	16,598/ 50	523	5,583/13	421	2,677/ 6	479	5,548/12
$clasp_2^0$	417	2,364/7	748	17,132/ 50	487	5,823/14	422	2,583/ 5	482	5,592/15
$clasp_2^1$-r	416	2,378/6	752	17,269/ 52	492	5,663/12	414	**2,409**/ 5	451	**5,349**/11
$clasp_2^1$	425	2,365/6	864	17,128/ 51	517	6,151/15	412	2,681/ 5	463	5,972/14
$clasp_2^2$-r	445	2,402/6	706	16,551/ 50	528	5,788/13	419	2,700/ 5	**436**	5,519/13
$clasp_2^3$-r	434	2,345/6	748	16,982/ 51	518	5,850/14	415	2,559/ 5	457	5,360/13
cudf2msu	610	3,051/8	**669**	**5,318**/ 8	1270	8,709/18	548	**3,238**/ 7	**504**	4,750/ 9
cudf2pbo	465	**2,727**/7	1082	21,302/ 68	520	6,168/13	**462**	3,575/ 7	537	**3,487**/ 8
p2cudf	**463**	2,920/8	696	19,105/ 60	**516**	**3,947**/ 7	573	6,927/16	577	8,063/21

Table 1 Results on package configuration problems used in a recent MISC-live run.

in seconds followed by the number of instances on which the solver was aborted, either before finding the optimum or while still attempting to prove it (or unsatisfiability, respectively). These statistics are used for tie-breaking wrt scores in MISC-live ranking, and they also yield valuable information regarding solvers' capabilities to prove optima: after 280 seconds of running, closeness of runtime exhaustion (300 seconds) is signaled to a solver, so that the remaining time can be used to output the best solution found so far. Accordingly, we count solutions returned after more than 280 seconds as aborts, which are not reflected in scores (columns S) if output solutions happen to be optimal without the proof being completed. We ran our experiments under MISC-live conditions on an Intel Quad-Core Xeon E5520 machine, possessing 2.27GHz processors and 48GB main memory, under Linux. The best scores and runtimes obtained among the variants of *clasp* as well as the best ones among its competitors are highlighted in bold face in Table 1.

Recall that two optimization criteria are applied in the *paranoid* track, three in the *user1* track, and four in the remaining tracks. One may expect solvers optimizing criteria in the order of significance (all but the variants of $clasp_0$) to have greater advantages the longer the sequence of criteria is. In fact, we observe that $clasp_0$, optimizing criteria in parallel, is competitive in the *paranoid* track; in particular, the static sign heuristic applied by the variants of $clasp_0^1$ helps them to achieve the smallest score. However, the gap to other solvers is not large, neither in terms of scores nor runtimes. Unlike this, the disadvantages of $clasp_0^0$ and $clasp_0^1$ variants are remarkable in the other four tracks; they are compensated to some extent by the optimization-oriented dynamic variable selection applied by $clasp_0^2$-r and $clasp_0^3$-r. Comparing the variants of $clasp_1$ and $clasp_2$, applying Algorithm 1, we note that

they are less sensitive to heuristic aspects. Nonetheless, their relative performance varies over tracks, thus not suggesting any universal strategy to multi-criteria optimization. For instance, the variants of $clasp_1$, decreasing upper bounds linearly, are more successful than $clasp_2$ variants in the *trendy* track, where the large total runtimes and numbers of aborts indicate that many instances were hard to complete (proving optima failed in many cases). On the other hand, the exponential decrease scheme of $clasp_2$ enables some of its variants to achieve the smallest score and runtime in the *user3* track. Finally, comparing the variants of *clasp* with its three competitors, we observe that the ASP-based approach to Linux package configuration is highly competitive. In particular, its consistent performance is confirmed by scores, while each of the other tools achieved an impressive runtime (mainly by succeeding to prove optima) in some track: *cudf2msu* in *trendy*, *cudf2pbo* in *user3*, and *p2cudf* in *user1*. Unfortunately, *cudf2msu* produced non-optimal solutions and crashes in two tracks, *trendy* and *user1*, so that its ranking in these two tracks is not very conclusive.

5 Discussion

We presented an approach to dedicated multi-criteria optimization in ASP. In particular, we detailed the use of assumptions in modern (conflict-driven learning) Boolean constraint solvers, so that speculative upper bounds can be imposed temporarily and withdrawn after unsatisfiability proofs without relaunching the solver. In fact, our approach is readily applicable in related areas like PBO and MaxSAT. Albeit Linux package configuration tools based on these formalisms may already exploit similar techniques, we are unaware of precise specifications of them. In the future, regular comparisons in competitions by mancoosi could provide a fruitful platform for improving and sharing methods of optimization.

The interested reader is referred to the extended version of this paper [8] for a detailed description of solving Linux package configuration problems by appeal to the multi-criteria optimization capacities introduced in the previous sections.

Acknowledgments

This work was partly funded by DFG grant SCHA 550/8-2. We are grateful to Daniel Le Berre for useful discussions on the subject of this work and to the mancoosi project team for organizing MISC(-live).

References

1 J. Argelich, D. Le Berre, I. Lynce, J. Marques-Silva, and P. Rapicault. Solving Linux upgradeability problems using Boolean optimization. In I. Lynce and R. Treinen, editors, *Proceedings of the First International Workshop on Logics for Component Configuration (LoCoCo'10)*, pages 11–22. 2010.

2 J. Argelich, I. Lynce, and J. Marques-Silva. On solving Boolean multilevel optimization problems. In C. Boutilier, editor, *Proceedings of the Twenty-first International Joint Conference on Artificial Intelligence (IJCAI'09)*, pages 393–398. AAAI Press/The MIT Press, 2009.

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

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

5 A. Darwiche and K. Pipatsrisawat. Complete algorithms. In Biere et al. [4], pages 99–130.

6 N. Eén and N. Sörensson. An extensible SAT-solver. In E. Giunchiglia and A. Tacchella, editors, *Proceedings of the Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT'03)*, pages 502–518. Springer-Verlag, 2004.
7 M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and S. Thiele. A user's guide to `gringo`, `clasp`, `clingo`, and `iclingo`. Available at `http://potassco.sourceforge.net`.
8 M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub. Multi-criteria optimization in ASP and its application to Linux package configuration. Available at `http://www.cs.uni-potsdam.de/wv/pdfformat/gekakasc11b.pdf`.
9 M. Gebser, R. Kaminski, A. König, and T. Schaub. Advances in *gringo* series 3. In J. Delgrande and W. Faber, editors, *Proceedings of the Eleventh International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'11)*, pages 345–351. Springer-Verlag, 2011.
10 M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. Conflict-driven answer set solving. In M. Veloso, editor, *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI'07)*, pages 386–392. AAAI Press/The MIT Press, 2007.
11 N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. The DLV system for knowledge representation and reasoning. *ACM Transactions on Computational Logic*, 7(3):499–562, 2006.
12 J. Marques-Silva, I. Lynce, and S. Malik. Conflict-driven clause learning SAT solvers. In Biere et al. [4], pages 131–153.
13 P. Simons, I. Niemelä, and T. Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1-2):181–234, 2002.
14 T. Syrjänen. Lparse 1.0 user's manual. Available at `http://www.tcs.hut.fi/Software/smodels/lparse.ps.gz`.
15 T. Syrjänen. A rule-based formal model for software configuration. Technical Report A55, Helsinki University of Technology, 1999.
16 T. Syrjänen. Including diagnostic information in configuration models. In J. Lloyd, V. Dahl, U. Furbach, M. Kerber, K. Lau, C. Palamidessi, L. Pereira, Y. Sagiv, and P. Stuckey, editors, *Proceedings of the First International Conference on Computational Logic (CL'00)*, pages 837–851. Springer-Verlag, 2000.
17 R. Treinen and S. Zacchiroli. Common upgradability description format (CUDF) 2.0. Technical Report 003, mancoosi — managing software complexity, 2009.

Yet Another Characterization of Strong Equivalence*

Alexander Bochman[1] and Vladimir Lifschitz[2]

1 **Holon Institute of Technology, Israel**
2 **University of Texas at Austin, USA**

Abstract

Strong equivalence of disjunctive logic programs is characterized here by a calculus that operates with syntactically simple formulas.

1998 ACM Subject Classification D.1.6 Logic Programming, I.2.3 Deduction and Theorem Proving

Keywords and phrases Strong equivalence, logic program

Digital Object Identifier 10.4230/LIPIcs.ICLP.2011.11

1 Introduction

Logic programs Π_1 and Π_2 are said to be strongly equivalent to each other if, for every logic program Π, the program $\Pi_1 \cup \Pi$ has the same stable models as $\Pi_2 \cup \Pi$ [4]. The study of strong equivalence is important because we learn from it how one can simplify a part of a logic program without looking at the rest of it. Characterizations of strong equivalence of logic programs that allow us to establish it more easily than by using the definition directly are given in [4], [6], and [7].

According to the main theorem of the first of these papers, grounded programs are strongly equivalent to each other iff the equivalence between them can be proved in the logic of here-and-there *HT*—the extension of intuitionistic propositional logic obtained by adding to it the axiom schema

$$F \vee (F \to G) \vee \neg G. \tag{1}$$

This statement assumes that grounded rules are viewed as alternative notation for propositional formulas. Specifically, a disjunctive rule

$$A_1; \ldots; A_k; \textit{not } A_{k+1}; \ldots; \textit{not } A_l \leftarrow A_{l+1}, \ldots, A_m, \textit{not } A_{m+1}, \ldots, \textit{not } A_n \tag{2}$$

$(n \geq m \geq l \geq k \geq 0)$, where each A_i is an atom, is identified with the propositional formula

$$A_{l+1} \wedge \cdots \wedge A_m \wedge \neg A_{m+1} \wedge \cdots \wedge \neg A_n \to A_1 \vee \cdots \vee A_k \vee \neg A_{k+1} \vee \cdots \vee \neg A_l. \tag{3}$$

In the special case when each rule of the program has the form (2) without negation in head $(l = k)$, strong equivalence can be characterized by a calculus that operates with such rules directly, without rewriting them as propositional formulas [8]. This fact shows that

* This work was partially supported by the National Science Foundation under Grant IIS-0712113.

© Alexander Bochman and Vladimir Lifschitz;
licensed under Creative Commons License NC-ND
Technical Communications of the 27th International Conference on Logic Programming (ICLP'11).
Editors: John P. Gallagher, Michael Gelfond; pp. 11–15

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

under some conditions strong equivalence can be described in terms of derivations that do not involve syntactically complex expressions, such as (1).

In this note we show that results of [1, Chapter 5] give us, implicitly, a calculus similar to the one proposed in [8], slightly more general (negation in the head is allowed) and slightly simpler (an inference rule with many premises is replaced by a rule with one premise). In this modification of the calculus from [8], derivable objects are "flat implications"—arbitrary formulas of the form (3). Negation in the head is important because it is needed to encode the choice construct, frequently used in answer set programming [3]. For instance, the choice rule

$$\{p\} \leftarrow q, \mathit{not}\ r$$

can be thought of as shorthand for

$$p; \mathit{not}\ p \leftarrow q, \mathit{not}\ r.$$

2 Calculus of Flat Implications

A *flat implication* is a propositional formula of the form $C \to D$, where C is a conjunction of literals (possibly the empty conjunction $\top$), and D is a disjunction of literals (possibly the empty disjunction $\bot$).

In the description of the calculus of flat implications *CFI* below, A is an atom; L is a literal; C, C_1, C_2 are conjunctions of literals; D, D_1, D_2 are disjunctions of literals; N is a disjunction of negative literals.

The calculus consists of two axiom schemas

$$A \to A, \tag{4}$$

$$A \wedge \neg A \to \bot \tag{5}$$

and three inference rules: *cut*

$$\frac{C_1 \to D_1 \vee L \qquad L \wedge C_2 \to D_2}{C_1 \wedge C_2 \to D_1 \vee D_2},$$

regularity

$$\frac{A \wedge C \to N}{C \to N \vee \neg A},$$

and the *structural rule*

$$\frac{C \to D}{C_1 \to D_1}$$

where each member of C is a member of C_1
and each member of D is a member of D_1.

Theorem. *A flat implication I is derivable from a set Π of flat implications in CFI iff I is derivable from Π in HT.*

In Section 4 we will show that this theorem is essentially a restatement of [1, Theorem 5.36].

Corollary. *For any sets Π_1, Π_2 of flat implications, the following conditions are equivalent:*

- Π_1 *is strongly equivalent to* Π_2*,*

- *in the calculus of flat implications, each element of* Π_1 *can be derived from* Π_2*, and each element of* Π_2 *can be derived from* Π_1*.*

The main feature of *CFI* that distinguishes it from the calculus proposed in [8] is the regularity rule, which takes advantage of the availability of negation in the heads of rules.

3 Examples

Example 1. We would like to verify that in the presence of the choice rule $\{p\}$, the rule $p \leftarrow q$ can be replaced by the constraint $\leftarrow q, not\ p$. In other words, we want to show that the program

$$\begin{array}{l} \{p\} \\ p \leftarrow q \end{array}$$

is strongly equivalent to

$$\begin{array}{l} \{p\} \\ \leftarrow q, not\ p. \end{array}$$

According to the corollary above, it is sufficient to derive in the calculus of flat implications

(a) $q \wedge \neg p \to \bot$ from $q \to p$;
(b) $q \to p$ from $\top \to p \vee \neg p$ and $q \wedge \neg p \to \bot$.

Part (a):

1. $q \to p$ (assumption).
2. $p \wedge \neg p \to \bot$ (axiom).
3. $q \wedge \neg p \to \bot$ (by cut from 1 and 2).

Part (b):

1. $\top \to p \vee \neg p$ (assumption).
2. $q \wedge \neg p \to \bot$ (assumption).
3. $\neg p \wedge q \to \bot$ (by the structural rule from 2).
4. $q \to p$ (by cut from 1 and 3).

Example 2. We would like to verify that the disjunctive program

$$\begin{array}{l} p; q \\ \leftarrow p, q \end{array}$$

is strongly equivalent to the nondisjunctive program

$$\begin{array}{rl} p & \leftarrow not\ q \\ q & \leftarrow not\ p \\ & \leftarrow p, q. \end{array}$$

It is sufficient to derive in the calculus of flat implications

(a) $\top \to p \vee q$ from the formulas

$$\neg q \to p, \quad \neg p \to q, \quad p \wedge q \to \bot;$$

(b) $\neg q \to p$ and $\neg p \to q$ from the formulas

$$\top \to p \vee q, \quad p \wedge q \to \bot.$$

Part (a):

1. $p \wedge q \rightarrow \bot$ (assumption).
2. $q \rightarrow \neg p$ (by regularity from 1).
3. $\top \rightarrow \neg p \vee \neg q$ (by regularity from 2).
4. $\neg q \rightarrow p$ (assumption).
5. $\top \rightarrow \neg p \vee p$ (by cut from 3 and 4).
6. $\top \rightarrow p \vee \neg p$ (by the structural rule from 5).
7. $\neg p \rightarrow q$ (assumption).
8. $\top \rightarrow p \vee q$ (by cut from 6 and 7).

Part (b):

1. $\top \rightarrow p \vee q$ (assumption).
2. $q \wedge \neg q \rightarrow \bot$ (axiom).
3. $\neg q \rightarrow p$ (by cut from 1 and 2).

The derivation of $\neg p \rightarrow q$ is similar.

4 Proof of the Theorem

According to [1], a *bisequent* is an expression of the form

$$a : b \Vdash c : d \tag{6}$$

where a, b, c, d are finite sets of atoms. Bisequents can be thought of as flat implications in disguise if we agree to identify (6) with the formula

$$\bigwedge_{A \in a} A \wedge \bigwedge_{A \in b} \neg A \;\rightarrow\; \bigvee_{A \in c} A \vee \bigvee_{A \in d} \neg A.$$

From [1, Proposition 5.84] we see that, given this convention, stable models of a set Π of flat implications are identical to the extensions of Π in the sense of [1, Definitions 5.7 and 5.8].

The characterization of strong-extension equivalence of bisequent theories given by [1, Theorem 5.36] provides a characterization of strong equivalence of sets of flat implications in terms of a calculus that is almost identical to *CFI*. The axioms of that calculus are our axioms (4) and (5) (called in the book positive reflexivity and consistency, see [1, Definitions 3.1 and 3.8]) plus the axiom schema

$$\neg A \rightarrow \neg A \tag{7}$$

(negative reflexivity, see [1, Definition 3.1]). Its inference rules are the inference rules of *CFI* (monotonicity, positive cut, negative cut, and C-regularity, see [1, Definition 5.7 and Section 3.2.5]). It remains to observe that (7) can be derived from (5) by one application of the regularity rule.

The "only if" part of the theorem can be proved also by noting that all postulates of *CFI* can be justified in *HT*. In fact, the axioms, the cut rule, and the structural rule are even intuitionistically acceptable. As to the regularity rule, its conclusion can be intuitionistically derived from its premise and the weak excluded middle axiom $\neg A \vee \neg\neg A$; the latter is provable in *HT* (in the axiom schema (1), take A as F and $\neg A$ as G). This line of reasoning shows, incidentally, that on the level of flat implications the logic of here-and-there does not differ from the logic of the weak excluded middle *WEM*—a fact known from [2].

5 Conclusion

There is a certain degree of freedom when we decide which monotonic logic can be viewed as the basis of the stable model semantics of disjunctive logic programs. From the results of [4] and [2] we see that each of the systems *HT* and *WEM* can play this role; the theorem presented in this note shows that *CFI* would do as well.

In [5], the theorem from [4] is extended to logic programs with variables and to a first-order version of *HT*. It would be interesting to extend the property of *CFI* proved above in a similar way.

6 Acknoweldgements

We are grateful to Fangzhen Lin and to the anonymous referees for useful comments.

References

1 Alexander Bochman. *Explanatory nonmonotonic reasoning*. World Scientific, 2005.
2 Dick De Jongh and Lex Hendriks. Characterization of strongly equivalent logic programs in intermediate logics. *Theory and Practice of Logic Programming*, 3:259–270, 2003.
3 Paolo Ferraris and Vladimir Lifschitz. Weight constraints as nested expressions. *Theory and Practice of Logic Programming*, 5:45–74, 2005.
4 Vladimir Lifschitz, David Pearce, and Agustin Valverde. Strongly equivalent logic programs. *ACM Transactions on Computational Logic*, 2:526–541, 2001.
5 Vladimir Lifschitz, David Pearce, and Agustin Valverde. A characterization of strong equivalence for logic programs with variables. In *Procedings of International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, pages 188–200, 2007.
6 Fangzhen Lin. Reducing strong equivalence of logic programs to entailment in classical propositional logic. In *Proceedings of International Conference on Principles of Knowledge Representation and Reasoning (KR)*, pages 170–176, 2002.
7 Hudson Turner. Strong equivalence made easy: nested expressions and weight constraints. *Theory and Practice of Logic Programming*, 3(4,5):609–622, 2003.
8 Ka-Shu Wong. Sound and complete inference rules for SE-consequence. *Journal of Artificial Intelligence Research*, 31:205–216, 2008.

Evolution of Ontologies using ASP

Max Ostrowski[1], Giorgos Flouris[2], Torsten Schaub[3], and Grigoris Antoniou[4]

1,3 Universität Potsdam
2 FORTH-ICS
4 University of Crete

Abstract

RDF/S ontologies are often used in e-science to express domain knowledge regarding the respective field of investigation (e.g., cultural informatics, bioinformatics etc). Such ontologies need to change often to reflect the latest scientific understanding on the domain at hand, and are usually associated with constraints expressed using various declarative formalisms to express domain-specific requirements, such as cardinality or acyclicity constraints. Addressing the evolution of ontologies in the presence of ontological constraints imposes extra difficulties, because it forces us to respect the associated constraints during evolution. While these issues were addressed in previous work, this is the first work to examine how ASP techniques can be applied to model and implement the evolution process. ASP was chosen for its advantages in terms of a principled, rather than ad hoc implementation, its modularity and flexibility, and for being a state-of-the-art technique to tackle hard combinatorial problems. In particular, our approach consists in providing a general translation of the problem into ASP, thereby reducing it to an instance of an ASP program that can be solved by an ASP solver. Our experiments are promising, even for large ontologies, and also show that the scalability of the approach depends on the morphology of the input.

1998 ACM Subject Classification I.2.4 Semantic Networks

Keywords and phrases Ontology evolution, Evolution in the presence of constraints, incremental ASP application

Digital Object Identifier 10.4230/LIPIcs.ICLP.2011.16

1 Introduction

Semantic Web [3], aims to extend the current web so as to allow information to be both understandable by humans and processable by machines. Ontologies describe our understanding of the physical world in a machine-processable format and form the backbone of the Semantic Web. They are usually represented using the RDF/S [13, 4] language; in a nutshell, RDF/S permits the representation of different types of resources like individuals, classes of individuals and properties between them, as well as basic taxonomic facts (such as subsumption and instantiation relationships).

Several recent works [16, 14, 12, 5, 19] have acknowledged the need for introducing constraints in ontologies. Given that RDF/S does not impose any constraints on data, any application-specific constraints (e.g., functional properties) or semantics (e.g., acyclicity in subsumptions) can only be captured using constraints on top of RDF/S data. In this paper, we consider DED constraints [8], which form a subset of first-order logic and have been shown to allow capturing many useful types of constraints; we will consider populated ontologies represented using RDF/S, and use the term *RDF/S knowledge base* (KB) to denote possibly interlinked and populated RDF/S ontologies with associated (DED) constraints.

© Max Ostrowski, Giorgos Flouris, Torsten Schaub and Grigoris Antoniou;
licensed under Creative Commons License NC-ND
Technical Communications of the 27th International Conference on Logic Programming (ICLP'11).
Editors: John P. Gallagher, Michael Gelfond; pp. 16–27

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

An important task towards the realization of the Semantic Web is the introduction of techniques that allow the efficient and intuitive evolution of KBs in the presence of constraints. Note that a *valid* evolution result should satisfy the constraints; this is often called the *Principle of Validity* [1]. In addition, the *Principle of Success* [1] should be satisfied, which states that the change requirements take priority over existing information, i.e., the change must be applied in its entirety. The final important requirement is the *Principle of Minimal Change* [1], which states that, during a change, the modifications applied upon the original KB must be minimal. In other words, given many different evolution results that satisfy the principles of success and validity, one should return the one that is "closer" to the original KB, where "closeness" is an application-specific notion. The above non-trivial problem was studied in [11], resulting in a general-purpose changing algorithm that satisfies the above requirements. Unfortunately, the problem was proven to be exponential in nature, so the presented general-purpose algorithmic solution to the problem (which involved a recursive process) was inefficient.

ASP is a flexible and declarative approach to solve NP-hard problems. The solution that was presented in [11] regarding the problem of ontology evolution in the presence of constraints can easily be translated into a logic program with first-order variables; this is the standard formalism that is used by ASP, which is then grounded into a variable free representation by a so called *grounder* that is then solved by a highly efficient Boolean *solver*. As it is closely related to the SAT paradigm, knowledge about different techniques for solving SAT problems are incorporated into the ASP algorithms. Using first-order logic programs is a smart way to represent the evolution problem while remaining highly flexible, especially with respect to the set of constraints related to the ontology.

The objective of the present work is to recast the problem of ontology evolution with constraints in terms of ASP rules, and use an efficient grounder and ASP solver to provide a modular and flexible solution. In our work, we use *gringo* for the grounding and *clasp* for the solving process as they are both state-of-the-art tools to tackle ASP problems [9]. Our work is based on the approach presented in [11], and uses similar ideas and notions. The main contribution of this work is the demonstration that ASP can be used to solve the inherently difficult problem of ontology evolution with constraints in a decent amount of time, even for large real-world ontologies. ASP was chosen for its advantages in terms of a principled, rather than ad hoc implementation, its modularity and flexibility, and for being a state-of-the-art technique to tackle hard combinatorial problems.

In the next section we present the problem of ontology evolution in the presence of constraints, and the solution proposed in [11]. In Section 3, we present ASP. Section 4 is the main section, where our formulation of the problem in terms of an ASP program is presented and explained. This approach is refined and optimized in Section 5. We present our experiments in Section 6 and conclude in Section 7.

2 Problem Statement

2.1 RDF/S

The RDF/S [13, 4] language uses triples of the form (subject, predicate, object) to express knowledge. RDF/S permits the specification of various entities (called *resources*), which may be classes (i.e., collections of resources), properties (i.e., binary relations between resources), and individuals (i.e., atomic entities). We use the symbol $type(u)$ to denote the type of a resource u (class, property, individual). RDF/S supports various predefined relations between resources, like the domain and range of properties, subsumption relationships between classes

Table 1 Representation of RDF/S Triples Using Predicates

RDF/S triple	Intuitive meaning	Predicate
c *rdf:type rdfs:Class*	c is a class	$cs(c)$
x *rdf:type rdfs:Resource*	x is an individual	$ci(x)$
c_1 *rdfs:subClassOf* c_2	IsA between classes	$c_IsA(c_1, c_2)$
x *rdf:type* c	class instantiation	$c_Inst(x, c)$

and between properties, and instantiation relationships between individuals and classes, or between pairs of individuals and properties. RDF/S associates such relations with semantics, e.g., subsumption is transitive.

RDF/S KBs are commonly represented as labeled graphs, whose nodes are resources and edges are relations (see Fig. 1). In Fig. 1, a, b, and c are classes and x is an individual. Solid arrows represent subsumption relationships between classes (e.g., b is a subclass of c), and dashed arrows represent instantiation relationships (e.g., x is an instance of b). The bold arrow represents the change we want to make, namely to make a a subclass of b.

2.2 Ontology Evolution Principles

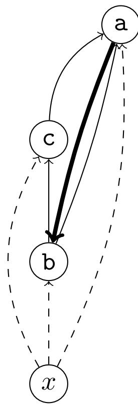

Figure 1 A knowledge base with change (appearing as bold arrow)

In the presence of constraints in the ontology, one should make sure that the evolution result is valid, i.e., it does not violate any constraints. This is called the Principle of Validity [1]. Manually enforcing this principle is an error-prone and tedious process. The objective of this work is to assist knowledge engineers in applying their changes in an automated manner, while making sure that no invalidities are introduced in the KB during the evolution.

In addition to the Validity Principle, two other principles are usually considered. The first is the *Principle of Success* [1], stating that the required changes take priority over existing information, i.e., the change must be applied in its entirety. The second is the *Principle of Minimal Change* [1], which requires that the modifications applied upon the original KB to accommodate the change must be minimal. Thus, if there are several different results that satisfy the principles of success and validity, one should return the one that is "closer" to the original KB, i.e., causes the least important modifications. Note that the importance of modifications (i.e., "closeness") is an application-specific notion; in this work, we model "closeness" using a relation; details on this relation will be given later.

2.3 Formal Setting

To address the problem of ontology evolution, we use the general approach presented in [11]. An RDF/S KB $\mathcal{K}$ is modeled as a set of ground facts of the form $p(\vec{x})$ where p is a predicate and $\vec{x}$ is a vector of constants. Constants represent resources in RDF/S parlance, and each predicate represents one type of RDF/S relationship (e.g., domain, range, subsumption etc). For example, the triple $(a, \mathit{rdfs{:}subClassOf}, b)$, which denotes that a is a subclass of b, is represented by the ground fact $c_IsA(a, b)$. For the rest of the paper, predicates and constants will start with a lower case letter, whereas variables will start with an upper case letter. Table 1 shows some of the predicates we use and their intuitive meaning (see [11] for a complete list).

We assume closed world, i.e., $\mathcal{K} \nvdash p(\vec{x})$ whenever $p(\vec{x}) \notin \mathcal{K}$. A *change* $\mathcal{C}$ is a request to

Table 2 Ontological Constraints

ID, Constraint	Intuitive Meaning
$R5$: $\forall U, V$ $c_IsA(U, V) \rightarrow cs(U) \wedge cs(V)$	Class subsumption
$R12$: $\forall U, V, W$ $c_IsA(U, V) \wedge c_IsA(V, W) \rightarrow c_IsA(U, W)$	Class IsA transitivity
$R13$: $\forall U, V$ $c_IsA(U, V) \wedge c_IsA(V, U) \rightarrow \bot$	Class IsA irreflexivity

Table 3 Facts from example in Fig. 1

$\mathcal{K}_0$		$ci(x)$	
	$cs(a)$	$cs(b)$	$cs(c)$
	$c_IsA(b, c)$	$c_IsA(b, a)$	$c_IsA(c, a)$
	$c_Inst(x, b)$	$c_Inst(x, c)$	$c_Inst(x, a)$
$\mathcal{C}$		$c_IsA(a, b)$	

add/remove fact(s) to/from the KB, and it is modeled as a set of positive/negative ground facts.

Ontological constraints are modeled using DED rules [8], which allow for formulating various useful constraints, such as primary and foreign key constraints (used, e.g., in [12]), acyclicity and transitivity constraints for properties (as in [16]), and cardinality constraints (used in [14]). Here, we use the following simplified form of DEDs, which still includes the above constraint types:

$$\forall \vec{U} \bigvee_{i=1,\ldots,head} \exists \vec{V_i} q_i(\vec{U}, \vec{V_i}) \leftarrow e(\vec{U}) \wedge p_1(\vec{U}) \wedge \cdots \wedge p_{body}(\vec{U}),$$

where $e(\vec{U})$ is a conjunction of (in)equality atoms. We denote by p the facts $p_1(\vec{U}), \ldots, p_{body}(\vec{U})$ and by q the facts $q_1(\vec{U}, \vec{V_1}), \ldots, q_{head}(\vec{U}, \vec{V}_{head})$. Table 2 shows some of the constraints used in this work; for a full list, refer to [11]. We say that a KB $\mathcal{K}$ *satisfies* a constraint r (or a set of constraints $\mathcal{R}$), iff $\mathcal{K} \vdash r$ ($\mathcal{K} \vdash \mathcal{R}$). Given a set of constraints $\mathcal{R}$, $\mathcal{K}$ is *valid* iff $\mathcal{K} \vdash \mathcal{R}$.

Now consider the KB $\mathcal{K}_0$ and the change of Fig. 1, which can be formally expressed using the ground facts of Table 3. To satisfy the principle of success, we should add $c_IsA(a, b)$ to $\mathcal{K}_0$, getting $\mathcal{K}_1 = \mathcal{K}_0 \cup \{c_IsA(a, b)\}$. The result ($\mathcal{K}_1$) is called the *raw application* of $\mathcal{C}$ upon $\mathcal{K}_0$, and denoted by $\mathcal{K}_1 = \mathcal{K}_0 + \mathcal{C}$. $\mathcal{C}$ is called a *valid change* w.r.t. $\mathcal{K}_0$ iff $\mathcal{K}_0 + \mathcal{C}$ is valid. In our example, this is not the case, because $\mathcal{K}_1$ violates $R13$; thus, it does not constitute an acceptable evolution result. The form of the violated rule implies that the only possible solution to this problem is to remove $c_IsA(b, a)$ from $\mathcal{K}_1$ (removing $c_IsA(a, b)$ is not an option, because its addition is dictated by the change – cf. the Principle of Success). This is an extra modification, that is not part of the original change, but is, in a sense, enforced by it; such extra modifications are called *side-effects*.

We note that the result, $\mathcal{K}_2 = \mathcal{K}_0 \cup \{c_IsA(a, b)\} \setminus \{c_IsA(b, a)\}$ is no good either, because $R12$ is violated, so, we need to repeat the above process recursively for $\mathcal{K}_2$. Note that $R12$ can be resolved in more than one ways, each of which should be evaluated independently; this fact leads to a recursive tree of resolutions (and side-effects). Eventually, after possibly several recursive steps, we will reach one or more valid KBs (leaves in the resolution tree);

these are possible results for the evolution, as they satisfy the principles of success and validity. In our example, these are: $\mathcal{K}_{4.1} = \mathcal{K}_0 \cup \{c_IsA(a,b), c_IsA(c,b)\} \setminus \{c_IsA(b,a), c_IsA(b,c)\}$ and $\mathcal{K}_{4.2} = \mathcal{K}_0 \cup \{c_IsA(a,b), c_IsA(a,c)\} \setminus \{c_IsA(b,a), c_IsA(c,a)\}$.

It remains to determine the "preferable" KB, i.e., the one that is "closest" to $\mathcal{K}_0$. To do so, we first determine the "distance" between KBs using difference sets, called *deltas*, which contain the positive/negative ground facts that need to be added/removed from one KB to get to the other (denoted by $\Delta(\mathcal{K}, \mathcal{K}')$). In our example, $\Delta(\mathcal{K}_0, \mathcal{K}_{4.1}) = \{c_IsA(a,b), c_IsA(c,b), \neg c_IsA(b,a), \neg c_IsA(b,c)\}$, $\Delta(\mathcal{K}_0, \mathcal{K}_{4.2}) = \{c_IsA(a,b), c_IsA(a,c), \neg c_IsA(b,a), \neg c_IsA(c,a)\}$. Then, we can determine "closeness" using an ordering that ranks $\Delta(\mathcal{K}_0, \mathcal{K}_{4.1}), \Delta(\mathcal{K}_0, \mathcal{K}_{4.2})$; both deltas have the same size (and this occurs often), so ranking cannot be based on cardinality, but should also consider more subtle differences, like the severity of changes.

Here, we consider the ordering defined in [11], which is denoted by $<_{\mathcal{K}_0}$, where $\mathcal{K}_0$ the original KB. To define $<_{\mathcal{K}_0}$, we first order the available predicates in terms of severity ($<_{pred}$); for example, the addition of a class (predicate *cs*) is more important than the addition of a subsumption (predicate *c_IsA*), i.e., $c_IsA <_{pred} cs$. Then, Δ_1 is preferable than Δ_2 (denoted by $\Delta_1 <_{\mathcal{K}_0} \Delta_2$) iff the most important predicate (per $<_{pred}$) appears less times in Δ_1. In case of a tie, the next most important predicate is considered, and so on. If the deltas contain an equal number of ground facts per predicate, the ordering considers the constants involved: a constant is considered more important if it occupies a higher position in its corresponding subsumption hierarchy in the original KB. In this respect, $\Delta(\mathcal{K}_0, \mathcal{K}_{4.1})$ causes less important changes upon $\mathcal{K}_0$ than $\Delta(\mathcal{K}_0, \mathcal{K}_{4.2})$, because the former affects b, c ($c_IsA(c,b), \neg c_IsA(b,c)$) whereas the latter affects c, a ($c_IsA(a,c), \neg c_IsA(c,a)$); this means that $\mathcal{K}_{4.1}$ is a preferred result (over $\mathcal{K}_{4.2}$), as $\Delta(\mathcal{K}_0, \mathcal{K}_{4.1}) <_{\mathcal{K}_0} \Delta(\mathcal{K}_0, \mathcal{K}_{4.2})$. The ordering between ground facts that allows this kind of comparison is denoted by $<_G$. For a more formal and detailed presentation of the ordering, we refer the reader to [11].

We denote the evolution operation by $\bullet$. In our example, we get $\mathcal{K}_0 \bullet \mathcal{C} = \mathcal{K}_{4.1}$. Note that $\mathcal{K}_0 \bullet \mathcal{C}$ results from applying the change, $\mathcal{C}$, and its most preferable side-effects upon $\mathcal{K}_0$.

3 Answer Set Programming (ASP)

In what follows, we rely on the input language of the ASP grounder *gringo* [9] (extending the language of *lparse* [18]) and introduce only informally the basics of ASP. A comprehensive, formal introduction to ASP can be found in [2].

We consider extended logic programs as introduced in [17]. A *rule* r is of the following form:

$$h \leftarrow b_1, \ldots, b_m, \sim b_{m+1}, \ldots, \sim b_n.$$

By $head(r) = h$ and $body(r) = \{b_1, \ldots, b_m, \sim b_{m+1}, \ldots, \sim b_n\}$, we denote the *head* and the *body* of r, respectively, where "$\sim$" stands for default negation. The head H is an atom a belonging to some alphabet $\mathcal{A}$, the falsum $\bot$, or a cardinality constraint $L\,\{\ell_1, \ldots, \ell_k\}\,U$. In the latter, $\ell_i = a_i$ or $\ell_i = \sim a_i$ is a *literal* for $a_i \in \mathcal{A}$ and $1 \le i \le k$; L and U are integers providing a lower and an upper bound. Such a constraint is true if the number of its satisfied literals is between L and M. Either or both of L and U can be omitted, in which case they are identified with the (trivial) bounds 0 and ∞, respectively. A rule r such that $head(r) = \bot$ is an *integrity constraint*; one with a cardinality constraint as head is called a *choice rule*. Each body component B_i is either an atom or a cardinality constraint for $1 \le i \le n$. If $body(r) = \emptyset$, r is called a *fact*, and we skip "$\leftarrow$" when writing facts below. In addition to

rules, a logic program can contain `#minimize` statements of the form

$$\#\texttt{minimize}[\ell_1 = w_1@L_1, \ldots, \ell_k = w_k@L_k].$$

Besides literals ℓ_j and integer weights w_j for $1 \leq j \leq k$, a `#minimize` statement includes integers L_j providing priority levels. A `#minimize` statement distinguishes optimal answer sets of a program as the ones yielding the smallest weighted sum for the true literals among $\ell_1, \ldots, \ell_k$ sharing the same (highest) level of priority L, while for $L' > L$ the sum equals that of other answer sets. For a formal introduction, we refer the interested reader to [17], where the definition of answer sets for logic programs containing extended constructs (cardinality constraints and minimize statements) under "choice semantics" is defined.

Likewise, first-order representations, commonly used to encode problems in ASP, are only informally introduced. In fact, *gringo* requires programs to be *safe*, that is, each variable must occur in a positive body literal. Formally, we only rely on the function *ground* to denote the set of all ground instances, $\mathit{ground}(\Pi)$, of a program Π containing first-order variables. Further language constructs of interest, include conditional literals, like "$a\!:\!b$", the range and pooling operator ".." and ";" as well as standard arithmetic operations. The ":" connective expands to the list of all instances of its left-hand side such that corresponding instances of literals on the right-hand side hold [18, 9]. While ".." allows for specifying integer intervals, ";" allows for pooling alternative terms to be used as arguments within an atom. For instance, $p(1..3)$ as well as $p(1;2;3)$ stand for the three facts $p(1)$, $p(2)$, and $p(3)$. Given this, $q(X)\!:\!p(X)$ results in $q(1), q(2), q(3)$. See [9] for detailed descriptions of the input language of the grounder *gringo*.

4 Evolution using ASP

4.1 Potential Side-Effects

In order to determine the result of updating a KB, we need to determine the side-effects that would resolve any possible validity problems caused by the change. The general idea is simple: since the original KB is valid, a change causes a violation if it adds/removes a fact that renders some constraint invalid. Let us denote by ∇ the set of potential side effects of a change $\mathcal{C}$. Given a set of facts $\mathcal{C}$, we will write $\mathcal{C}^+/\mathcal{C}^-$ to denote the positive/negative facts of $\mathcal{C}$ respectively. First of all, we note that ∇ will contain all facts in $\mathcal{C}$, except those already implied by $\mathcal{K}$, i.e., if $p(\vec{x}) \in \mathcal{C}^+$ and $p(\vec{x}) \notin \mathcal{K}$, then $p(\vec{x}) \in \nabla^+$, and if $\neg p(\vec{x}) \in \mathcal{C}^-$ and $p(\vec{x}) \in \mathcal{K}$ then $\neg p(\vec{x}) \in \nabla^-$ (Condition I). The facts in the set $\nabla^+ \cup \mathcal{K}$ are called *available*. This initial set of effects may cause a constraint violation. Note that a constraint r is violated during a change iff the right-hand-side (rhs) of r becomes true and the left-hand-side (lhs) is made false. Thus, if a potential addition ∇^+ makes the rhs of r true, and lhs is false, then we have to add some fact from the lhs of the implication to the potential positive side-effects (to make lhs true) (Condition II), *or* remove some fact from rhs (to make it false) (Condition III). If a removal in ∇^- makes the lhs of r false, and all other facts in rhs are available (so rhs is true), we have to remove some fact from rhs (to make it false) (Condition IV). To do that, we first define a select function $s_i(X) = X \setminus \{X_i\}$ on a set X of *atoms*, to remove exactly one element of a set. So we can then refer to the element X_i and the rest of the set $s_i(X)$ separately. Abusing notation, we write $\mathit{pred}(p, \vec{U})$ for $\mathit{pred}(p_1, \vec{U}), \ldots, \mathit{pred}(p_n, \vec{U})$, for any predicate name *pred* where p is the set of atoms $p_1(\vec{U}), \ldots, p_{body}(\vec{U})$.

Formally, a set ∇ is a *set of potential side-effects* for a KB $\mathcal{K}$ and a change $\mathcal{C}$, if the following *conditions* are all true:

Table 4 Instance from example in Fig. 1

$$
\begin{array}{ccc}
 & kb(ci,(x)). & \\
kb(cs,(a)). & kb(cs,(b)). & kb(cs,(c)). \\
kb(c_IsA,(b,c)). & kb(c_IsA,(b,a)). & kb(c_IsA,(c,a)). \\
kb(c_Inst,(x,b)). & kb(c_Inst,(x,c)). & kb(c_Inst,(x,a)). \\
 & changeAdd(c_IsA,(a,b)). &
\end{array}
$$

I $x \in \nabla$ if $x \in \mathcal{C}^+$ and $x \notin \mathcal{K}$ *or* $x \in \mathcal{C}^-$ and $\neg x \in \mathcal{K}$,

II $\forall \vec{V}_h\ q_h(\vec{U},\vec{V}_h) \in \nabla^+$ if $s_l(p(\vec{U})) \subseteq \nabla^+ \cup \mathcal{K}$ and $p_l(\vec{U}) \in \nabla^+$ and $q_h(\vec{U},\vec{V}_h) \notin \mathcal{K}$

III $\neg p_j(\vec{U}) \in \nabla^-$ if $s_j(s_l(p(\vec{U}))) \subseteq \nabla^+ \cup \mathcal{K}$ and $p_l(\vec{U}) \in \nabla^+$ and $p_j(\vec{U}) \in \mathcal{K}$ and for all $\vec{V}_h$ either $\neg q_h(\vec{U},\vec{V}_h) \in \nabla^-$ or $q_h(\vec{U},\vec{V}_h) \notin \mathcal{K}$

IV $\neg p_l(\vec{U}) \in \nabla^-$ if $s_l(p(\vec{U})) \subseteq \nabla^+ \cup \mathcal{K}$ and $p_l(\vec{U}) \in \mathcal{K}$ and $\forall \vec{V}_h\ \neg q_h(\vec{U},\vec{V}_h) \in \nabla^-$,

for each constraint r defined in Section 2.3 and for all variable substitutions for $\vec{U}$ wrt $E(\vec{U})$ and for all $1 \leq l, j \leq body$, $l \neq j$, $1 \leq h \leq head$.

Our goal is to find a $\subset$-minimal set of potential side-effects ∇. We do this using the grounder *gringo*, which ground-instantiates a logic program with variables. We create a logic program where the single solution is the subset minimal set of potential side-effects ∇.

To build a logic program, we first have to define the inputs to the problem, called *instance*. An instance $\mathcal{I}(\mathcal{K},\mathcal{C})$ of a KB $\mathcal{K}$ and a change $\mathcal{C}$ is defined as a set of facts

$$
\begin{aligned}
\mathcal{I}(\mathcal{K},\mathcal{C}) = &\{kb(p,\vec{x}) \mid p(\vec{x}) \in \mathcal{K}\} \\
&\cup \{changeAdd(p,\vec{x}) \mid p(\vec{x}) \in \mathcal{C}^+\} \\
&\cup \{changeDel(p,\vec{x}) \mid p(\vec{x}) \in \mathcal{C}^-\}.
\end{aligned}
$$

In the above instance, predicate *kb* contains the facts in the KB, whereas predicates *changeAdd*, *changeDel* contain the facts that the change dictates to add/delete respectively. Note that this representation forms a twist from the standard representation, since a ground fact $p(\vec{x}) \in \mathcal{K}$ is represented as $kb(p,\vec{x})$ (same for the change). The representation of the KB $\mathcal{K}$ in Fig. 1 and its demanded change $\mathcal{C}$ can be found in Table 4.

Furthermore we have to collect all resources available in the KB (1) or newly introduced by the change (2). So the predicate *dom* associates a resource to its type,

$$dom(type(X_i),X_i) \leftarrow kb(T,\vec{X}). \tag{1}$$

$$dom(type(X_i),X_i) \leftarrow changeAdd(T,\vec{X}). \tag{2}$$

for all $X_i \in \vec{X}$. The following two rules ((3) and (4)) correspond to Condition I above, stating that the effects of $\mathcal{C}$ should be in ∇ (unless already in $\mathcal{K}$). The predicates *pAdd* and *pDelete* are used to represent potential side effects (additions and deletions respectively), i.e., facts in the sets ∇^+, ∇^-.

$$pDelete(T,\vec{X}) \leftarrow changeDel(T,\vec{X}), kb(T,\vec{X}). \tag{3}$$

$$pAdd(T,\vec{X}) \leftarrow changeAdd(T,\vec{X}), {\sim}kb(T,\vec{X}). \tag{4}$$

To find those facts that are added due to subsequent violations, we define, for the set $\nabla^+ \cup \mathcal{K}$, the predicate *avail* in (5) and (6). For negative potential side-effects ∇^- we use a predicate

Table 5 Potential side-effects of example in Fig. 1

$c_IsA(a,b)$	$c_IsA(c,c)$	$c_IsA(c,b)$
$c_IsA(b,b)$	$c_IsA(a,a)$	$c_IsA(a,c)$
$\neg c_IsA(b,a)$	$\neg c_IsA(b,c)$	$\neg c_IsA(c,a)$

$nAvail$ (7).

$$avail(T,\vec{X}) \leftarrow kb(T,\vec{X}). \tag{5}$$

$$avail(T,\vec{X}) \leftarrow pAdd(T,\vec{X}). \tag{6}$$

$$nAvail(T,\vec{X}) \leftarrow pDelete(T,\vec{X}). \tag{7}$$

At a next step, we need to include the ontological constraints $\mathcal{R}$ into our ASP program, by creating the corresponding ASP rules. Unlike standard ontological constraints which determine whether there is an invalidity, the ASP rules are used to determine how to handle an invalidity. So now consider a constraint $r \in \mathcal{R}$ as defined in Section 2.3. For r, we define a set of rules ((8)) that produce the set of potential side-effects according to Condition II.

$$pAdd(q_h,(\vec{U},\vec{V}_h)) \leftarrow e(\vec{U}), avail(s_l(p),\vec{U}), pAdd(p_l,\vec{U}), \sim kb(q_h,(\vec{U},\vec{V}_h)), dom(type(\vec{V}_h),\vec{V}_h). \tag{8}$$

for all $1 \leq l \leq body$ and $1 \leq h \leq head$. Similarly, to capture Condition III, we need two sets of rules ((9) and (10)), since we do not want to do this only for negative side-effects $nAvail$ on the lhs of the rule, but also for facts that are not in the KB $\mathcal{K}$,

$$pDelete(p_j,\vec{u}) \leftarrow e(\vec{U}), avail(s_j(s_l(p)),\vec{U}), pAdd(p_l,\vec{U}), kb(p_j,\vec{U}), nAvail(q_h,(\vec{U},\vec{V}_h)) : dom(type(\vec{V}_h),\vec{V}_h). \tag{9}$$

$$pDelete(p_j,\vec{U}) \leftarrow e(\vec{U}), avail(s_j(s_l(p)),\vec{U}), pAdd(p_l,\vec{U}), kb(p_j,\vec{U}), \sim kb(q_h,(\vec{U},\vec{V}_h)) : dom(type(\vec{V}_h),\vec{V}_h). \tag{10}$$

for all $1 \leq l, j \leq body$, $l \neq j$, $1 \leq h \leq head$. The last Condition IV can be expressed by the following rule set (11)

$$pDelete(p_l,\vec{U}) \leftarrow e(\vec{U}), avail(s_l(p),\vec{U}), kb(p_l,\vec{U}), pDelete(q_h,\vec{U},\vec{V}_h) : dom(type(\vec{V}_h),\vec{V}_h). \tag{11}$$

for all $1 \leq l \leq body$ and $1 \leq h \leq head$.

▶ Proposition 1. Given a KB $\mathcal{K}$ and a change $\mathcal{C}$, and let A be the unique answer set of the stratified logic program $ground(\mathcal{I}(\mathcal{K},\mathcal{C}) \cup \{(1)\ldots(11)\})$, then $\nabla = \{p(\vec{x}) \mid pAdd(p,\vec{x}) \in A\} \cup \{\neg p(\vec{x}) \mid pDelete(p,\vec{x}) \in A\}$ is a subset minimal set of potential side-effects of the KB $\mathcal{K}$ and the change $\mathcal{C}$.

For our example in Fig. 1, this results in the set of potential side-effects in Table 5. Note that the potential side-effects contain all possible side-effects, including side-effects that will eventually not appear in any valid change.

4.2 Solving the Problem

Note that the set of potential side-effects computed above contains all options for evolving the KB. However, some of the potential changes in $pAdd$, $pDelete$ are unnecessary; in our

running example, the preferred solution was $\{c_IsA(a,b), \neg c_IsA(b,a), \neg c_IsA(b,c)\}$ (see Section 2), whereas Table 5 contains many more facts.

To compute the actual side-effects (which is a subset of the side-effects in *pAdd*, *pDelete*), we use a generate and test approach. In particular, we use the predicate $add(p,\vec{x})$ and $delete(p',\vec{x}')$ to denote the set of side-effects $p(\vec{x}) \in \Delta(\mathcal{K},\mathcal{K}')$ (respectively $\neg p'(\vec{x}') \in \Delta(\mathcal{K},\mathcal{K}')$) and use choice rules to guess side-effects from *pAdd*, *pDelete* to *add*, *delete* respectively (see (12), (13) below).

$$\{add(T,\vec{X}) : pAdd(T,\vec{X})\}. \tag{12}$$

$$\{delete(T,\vec{X}) : pDelete(T,\vec{X})\}. \tag{13}$$

Our changed KB is expressed using predicate kb' and is created in (14) and (15) consisting of every entry from the original KB that was not deleted and every entry that was added.

$$kb'(T,\vec{X}) \leftarrow kb(T,\vec{X}), \sim delete(T,\vec{X}). \tag{14}$$

$$kb'(T,\vec{X}) \leftarrow add(T,\vec{X}). \tag{15}$$

Moreover, we have to ensure that required positive (negative) changes $\mathcal{C}$ are (not) in the new KB respectively (*Principle of Success*) ((16) and (17)).

$$\leftarrow changeAdd(T,\vec{X}), \sim kb'(T,\vec{X}). \tag{16}$$

$$\leftarrow changeDel(T,\vec{X}), kb'(T,\vec{X}). \tag{17}$$

To ensure the *Principle of Validity* we construct all constraints from the DEDs $\mathcal{R}$, using the following transformation for each $r \in \mathcal{R}$:

$$\leftarrow kb'(p,\vec{U}), \sim 1\{kb'(q_i,(\vec{U},\vec{V_i}) : dom(type(\vec{V_i}),\vec{V_i}))\}, e(\vec{U}). \tag{18}$$

for all $1 \leq i \leq head$. Rule (18) ensures that if the rhs of a constraint is true wrt to the new KB and the lhs if false, then the selected set of side-effects is no valid solution.

▶ Proposition 2. Given a KB $\mathcal{K}$, a change $\mathcal{C}$ and a set of potential side-effects ∇, we define a set of facts $\nabla' = \{pAdd(p,\vec{x}) \mid p(\vec{x}) \in \nabla\} \cup \{pDelete(p,\vec{x}) \mid \neg p(\vec{x}) \in \nabla\}$. Let A be the answer set of the logic program $ground(\mathcal{I}(\mathcal{K},\mathcal{C}) \cup \nabla' \cup \{(12)\ldots(18)\})$, then $\Delta(\mathcal{K},\mathcal{K}') = \{p(\vec{x}) \mid add(p,\vec{x}) \in A\} \cup \{\neg p(\vec{x}) \mid delete(p,\vec{x}) \in A\}$ is a valid change of KB $\mathcal{K}$.

4.3 Finding the Optimal Solution

The solutions contained in *add*, *delete* are all valid solutions, per the above proposition, but only one of them is optimal, per the Principle of Minimal Change. So, the solutions must be checked wrt to the ordering $<_{\mathcal{K}}$. We generate minimize statements for the criteria $<_{pred}$ and $<_G$ (see Section 3). Several minimize constraints can be combined and the order of the minimize statements is respected. As *gringo* allows hierarchical optimization statements, we can easily express the whole ordering $<_{\mathcal{K}}$ in a set of optimize statements $\mathcal{O}$.

▶ Proposition 3. Given a KB $\mathcal{K}$, a change $\mathcal{C}$ and a set of potential side-effects ∇, we define a set of facts $\nabla' = \{pAdd(p,\vec{x}) \mid p(\vec{x}) \in \nabla\} \cup \{pDelete(p,\vec{x}) \mid \neg p(\vec{x}) \in \nabla\}$. Let A be the answer set of the logic program $ground(\mathcal{I}(\mathcal{K},\mathcal{C}) \cup \nabla' \cup \{(12)\ldots(18)\})$, which is minimal wrt the optimize statements $\mathcal{O}$ then $\Delta(\mathcal{K},\mathcal{K}') = \{p(\vec{x}) \mid add(p,\vec{x}) \in A\} \cup \{\neg p(\vec{x}) \mid delete(p,\vec{x}) \in A\}$ is the unique valid minimal change of KB $\mathcal{K}$.

5 Refinements

In this section, we refine the above direct translation, in order to increase the efficiency of our logic program. Our first optimization attempts to reduce the size of the potential side-effects ∇, whereas the second takes advantage of deterministic consequences of certain side-effects to speed-up the process.

5.1 Incrementally Computing Side-Effects

As the set of potential side-effects directly corresponds to the search space for the problem (see (12), (13) in Section 4), we could improve performance if a partial set of potential side-effects that contains the minimal solution was found, instead of the full set. According to the ordering of the solutions $<_{pred}$, a set of side-effects that does not contain any fact with a level greater than k is "better" than a solution that does. Thus, we split the computation of the possible side-effects into different parts, one for each level of $<_{pred}$ optimization. We start the computation of possible side-effects with $k = 1$, only adding facts of level 1 to repair our KB. If with this subset of possible side-effects no solution to the problem can be found, we increase k by one and continue the computation, reusing the already computed part of the potential side-effects. For grounding, this means we only want to have the possibility to find potential side-effects $p(\vec{x})$ of a level less than or equal to k. The corresponding ASP rules can be found in the extended version of this paper [15] and are denoted by I.

We define the operator $\mathcal{T}(\mathcal{K}, \mathcal{C}, k)$, as $\mathcal{T}(\mathcal{K}, \mathcal{C}, 0) = \mathit{ground}(\mathcal{I}(\mathcal{K}, \mathcal{C}))$ and $\mathcal{T}(\mathcal{K}, \mathcal{C}, k)$ where $k > 0$ is the set of facts of the unique answer set of the logic program $\mathit{ground}(\mathcal{T}(\mathcal{K}, \mathcal{C}, k - 1) \cup \{I\})$. $\mathcal{T}(\mathcal{K}, \mathcal{C}, n)$ produces a subset of the potential side-effects only using repairs up to level n. Given our example in Fig. 1, $\mathcal{T}(\mathcal{K}, \mathcal{C}, 7)$ gives us the first two rows of Table 5.

5.2 Exploiting Deterministic Side-Effects

A second way to improve performance is to consider deterministic side-effects of the original changes. As an example of a deterministic side-effect, suppose that the original change includes the deletion of a class a (corresponding to the side-effect $\neg cs(a)$). Then, per rule R5 (cf. Table 2), all class subsumptions that involve a must be deleted as well (corresponding to the side-effect $\neg c_IsA$). Therefore, the latter side-effect(s) are a necessary (deterministic) consequence of the former, so they can be added to the set of side-effects right from the beginning (at level 1). For the detailed logic program we refer to [15]. In this way we extend our change by deterministic consequences, to possibly reduce the number of incremental steps. For our example in Fig. 1 this results in the additionally required $\mathit{changeDel}(c_IsA, (b, a))$.

6 Experiments

We experimented with two real-world ontologies of different size and structure, namely GO [7] and CIDOC [6] (~458.000/~1.500 facts). GO's emphasis is on classes, whereas CIDOC contains many properties. To generate the changes, we took each ontology $\mathcal{K}$, randomly selected 6 facts $I \subseteq \mathcal{K}$, and deleted I from $\mathcal{K}$, resulting in a valid KB $\mathcal{K}'$. We then created our "pool of changes", $I_{\mathcal{C}}$, which contains 6 randomly selected facts from $\mathcal{K}'$ (deletions) and the 6 facts from I (additions). The change $\mathcal{C}$ was a random selection of n facts from $I_{\mathcal{C}}$ ($1 \leq n \leq 6$). Our experiment measured the time required to apply $\mathcal{C}$ upon $\mathcal{K}'$. The above process was repeated 100 times for each n ($1 \leq n \leq 6$). The benchmark was run on a machine with 4×4 CPUs, 3.4Ghz each and was restricted to 4 GB of RAM. Our implementation uses

Table 6 (a) GO benchmark

n	times	level	timeouts
1	123.3	2.37	0
2	243.7	4.72	0
3	454.6	8.50	0
4	619.0	11.94	0
5	711.1	13.44	2
6	756.1	14.27	6

(b) CIDOC benchmark

n	times	level	timeouts
1	2.2	10.70	0
2	3.3	16.28	20
3	3.4	16.15	30
4	7.0	16.96	51
5	3.5	16.19	66
6	7.3	18.00	76

*gringo*3.0.4 and *clasp*2.0.0RC1. A timeout of 3600 seconds was imposed on each run. Table 6 contains the results of our experiments in GO and CIDOC respectively. Each row in the table contains the experiments for one value of n (size of $\mathcal{C}$) and shows the average CPU time (in seconds) of all runs that did not reach the timeout (column "times"), the average level of incremental grounding where the solution was found ("level") and the number of timeouts ("timeouts").

The results of our experiments are encouraging. GO, despite its large size and the intractable nature of the evolution problem, can evolve in a decent amount of time, and has very few timeouts. On the other hand, CIDOC has lots of timeouts, but very fast execution when no timeout occurs. This indicates that the deviation of execution times, even for KBs/changes of the same size, is very large for CIDOC, i.e., the performance is largely affected by the morphology of the input. This behaviour is much less apparent in GO, and is caused by the existence of many properties in CIDOC. Any violated property-related constraint greatly increases the number of potential side-effects. Thus, for updates causing many property-related violations, the execution time increases, often causing timeouts. Given that GO contains no properties, the execution times are more smooth. Another observation is that there is a strong correlation between the level, the average time reported and the size of the change.

7 Summary and Outlook

We studied the problem of ontology evolution in the presence of ontological constraints. Based on the setting and solution proposed in [11], we recast the problem and reduced it to an ASP program that can be solved by an optimized ASP reasoner. Given that the problem is inherently exponential in nature [11], the reported times (Table 6) for the evolution of two real-world ontologies (GO/CIDOC) are decent. To the best of our knowledge, there is no comparable approach, because the approach presented in [11] did not report any experiments, and other similar approaches either do not consider the full set of options (therefore returning a suboptimal evolution result), or require user feedback. An interesting side-product of our approach is that we can *repair ontologies* by simply applying the empty change upon them; we plan to explore this idea as a future work. We will also consider additional optimizations using incremental ASP solvers such as *iclingo* [10].

References

1 C. E. Alchourron, P. Gardenfors, and D. Makinson. On the logic of theory change: Partial meet contraction and revision functions. *Journal of Symbolic Logic*, 50:510–530, 1985.

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

3 Tim Berners-Lee, James A. Hendler, and Ora Lassila. The semantic web. *Scientific American*, 284(5):34–43, 2001.

4 D. Brickley and R.V. Guha. Rdf vocabulary description language 1.0: Rdf schema. `www.w3.org/TR/2004/REC-rdf-schema-20040210`, 2004.

5 A. Cali, G. Gottlob, and T. Lukasiewicz. Datalog±: A unified approach to ontologies and integrity constraints. In *Proceedings of the International Conference on Database Theory (ICDT-09)*, 2009.

6 CIDOC. *The CIDOC Conceptual Reference Model*, 2010. `cidoc.ics.forth.gr/official_release_cidoc.html`.

7 The Gene Ontology Consortium. Gene ontology: tool for the unification of biology. In *Nature genetics*, volume 25, pages 25–29, 2000.

8 Alin Deutsch. Fol modeling of integrity constraints (dependencies). *Encyclopedia of Database Systems*, pages 1155–1161, 2009.

9 M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and S. Thiele. A user's guide to `gringo`, `clasp`, `clingo`, and `iclingo`. Available at `http://potassco.sourceforge.net`.

10 M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and S. Thiele. Engineering an incremental ASP solver. In M. Garcia de la Banda and E. Pontelli, editors, *Proceedings of the Twenty-fourth International Conference on Logic Programming (ICLP'08)*, volume 5366 of *Lecture Notes in Computer Science*, pages 190–205. Springer Verlag, 2008.

11 George Konstantinidis, Giorgos Flouris, Grigoris Antoniou, and Vassilis Christophides. A formal approach for rdf/s ontology evolution. In *Proceedings of the 18th European Conference on Artificial Intelligence (ECAI-08)*, pages 405–409, 2008.

12 G. Lausen, M. Meier, and M. Schmidt. Sparqling constraints for rdf. In *Proceedings of 11th International Conference on Extending Database Technology (EDBT-08)*, pages 499–509, 2008.

13 B. McBride, F. Manola, and E. Miller. Rdf primer. `www.w3.org/TR/rdf-primer`, 2004.

14 B. Motik, I. Horrocks, and U. Sattler. Bridging the gap between owl and relational databases. In *Proceedings of 17th International World Wide Web Conference (WWW-07)*, pages 807–816, 2007.

15 M. Ostrowski, G. Flouris, T. Schaub, and G. Antoniou. Evolution of ontologies using ASP. Technical Report TR-415, FORTH-ICS, 2011.

16 G. Serfiotis, I. Koffina, V. Christophides, and V. Tannen. Containment and minimization of rdf/s query patterns. In *Proceedings of the 4th International Semantic Web Conference (ISWC-05)*, 2005.

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

18 T. Syrjänen. Lparse 1.0 user's manual. http://www.tcs.hut.fi/Software/smodels/lparse.ps.gz.

19 J. Tao, E. Sirin, J. Bao, and D.L. McGuinness. Extending owl with integrity constraints. In *Proceedings of the 23rd International Workshop on Description Logics (DL-10)*. CEUR-WS 573, 2010.

Modelling Grammar Constraints with Answer Set Programming

Christian Drescher and Toby Walsh

NICTA and University of New South Wales
Locked Bag 6016, Sydney NSW 1466, Australia
E-mail: {Christian.Drescher,Toby.Walsh}@nicta.com.au

Abstract

Representing and solving constraint satisfaction problems is one of the challenges of artificial intelligence. In this paper, we present answer set programming (ASP) models for an important and very general class of constraints, including all constraints specified via grammars or automata that recognise some formal language. We argue that our techniques are effective and efficient, e.g., unit-propagation of an ASP solver can achieve domain consistency on the original constraint. Experiments demonstrate computational impact.

1998 ACM Subject Classification F.4.1 Mathematical Logic

Keywords and phrases answer set programming, grammar-, regular-, precedence constraint

Digital Object Identifier 10.4230/LIPIcs.ICLP.2011.28

1 Introduction

Answer set programming (ASP; [3]) provides a compact, declarative, and highly competitive approach to modelling and solving constraint satisfaction problems (CSP) [21, 9, 11]. CSP are combinatorial problems defined as a set of variables whose value must satisfy a number of limitations (the constraints), and stem from a variety of areas. One very promising method for scheduling, rostering and sequencing problems is to specify constraints via grammars or automata that recognise some formal language [23, 27, 25, 16]. For instance, we might want to ensure that *anyone working three consecutive shifts then has two or more days off*, or that *an employee changes activities only after a fifteen minutes break or one hour lunch*. Grammar-based constraint propagators were proposed in [27, 25], and modelled with Boolean satisfiability (SAT; [4]) in [26, 1], while an automata-based constraint propagator was presented in [23], and modelled with SAT in [2]. Whilst SAT models can be directly converted into ASP [21], we here show that GRAMMAR and related constraints can be modelled with ASP in a more straightforward and easily maintainable way without paying any penalty, e.g., in form of efficiency, for using ASP. First, we show that the GRAMMAR constraint can be modelled with ASP based on the production rules in the grammar. Second, we present alternative ASP models for a special case of the GRAMMAR constraint, that is, the REGULAR constraint, based on deterministic finite automata. Third, we give theoretical results on the pruning achieved by unit-propagation of an ASP solver, and runtime complexity. Forth, we provide an ASP model for the PRECEDENCE constraint [18] which is a special case of the REGULAR constraint. It is useful for breaking value symmetry in CSP. We argue that our encoding improves runtime complexity over REGULAR. Finally, we demonstrate the applicability of our approach on *shift-scheduling* and *graph colouring* instances. We show that symmetry breaking using our ASP models yield significant improvements in runtime, and outperforms recent generic approaches to symmetry breaking for ASP.

© Christian Drescher and Toby Walsh;
licensed under Creative Commons License NC-ND
Technical Communications of the 27th International Conference on Logic Programming (ICLP'11).
Editors: John P. Gallagher, Michael Gelfond; pp. 28–39

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

2 Background

Answer Set Programming

A *(normal) logic program* over a set of primitive propositions $\mathcal{A}$, $\bot \in \mathcal{A}$, is a finite set of *rules* r of the form

$$a_0 \leftarrow a_1, .., a_m, not\ a_{m+1}, .., not\ a_n$$

where $a_i \in \mathcal{A}$ are *atoms* for $0 \leq i \leq n$, and $\{a_1, .., a_m, not\ a_{m+1}, .., not\ a_n\}$ is the *body* of r. We also consider *choice rules* of the form

$$\{h_1, .., h_k\} \leftarrow a_1, .., a_m, not\ a_{m+1}, .., not\ a_n$$

that allow for the nondeterministic choice over atoms in $\{h_1, .., h_k\}$. Their semantics is given through program transformations [28]. The answer sets of a logic program Π can be characterised as Boolean assignments A over the atoms and bodies in Π, $dom(A)$, that satisfy nogoods imposed by Π [13]. Formally, an *assignment* A is a set $\{\sigma_1, .., \sigma_n\}$ of *(signed) literals* σ_i expressing that $a \in dom(A)$ is assigned *true* or *false*. The proposition $\bot$ denotes an atom that is false in every assignment. In our context, a *nogood* [8] is a set $\delta = \{\sigma_1, .., \sigma_m\}$ of literals, expressing a condition *violated* by any assignment A if $\delta \subseteq A$. Nogoods allow for a transparent technology transfer from SAT since every nogood can be syntactically represented by a clause, for instance, inferences in ASP can be viewed as unit-propagation on nogoods [13]. For a nogood δ, a literal $\sigma \in \delta$, and an assignment A, we say that δ is *unit* and the complement of σ is *unit-resulting* if $\delta \setminus A = \{\sigma\}$. *Unit-propagation* is the process of repeatedly extending A by unit-resulting literals until no unit nogood remains or is violated. A total assignment A is a *solution* to a set of nogoods Δ if $\delta \not\subseteq A$ for all $\delta \in \Delta$. Given a logic program Π, one can specify a set of nogoods such that its solutions capture the models of the Clark's completion of Π [6, 13]. We denote this set $\Delta(\Pi)$. If Π is *tight* [12], the solutions to $\Delta(\Pi)$ are precisely the answer sets of Π. (All logic programs presented in this paper are tight.) Otherwise, loop formulas, expressed in the set of nogoods $\Lambda(\Pi)$, have to be added to establish full correspondence to the answer sets of Π [19].

Constraint Satisfaction

We want to use ASP to model and solve CSP. Formally, a CSP is a triple (V, D, C) where V is a finite set of *variables*, D is a set of finite *domains* such that each variable $\mathsf{v} \in V$ has an associated domain $dom(\mathsf{v}) \in D$, and C is a set of constraints. A *constraint* c is a pair (R_S, S) where R_S is an n-ary *relation* on the variables in $S \in V^n$, called the *scope* of c. A *(variable) assignment* $\mathsf{v} \mapsto t$ means that variable $\mathsf{v} \in V$ is mapped to the value $t \in dom(\mathsf{v})$. A *(compound) assignment* $(\mathsf{v}_1, .., \mathsf{v}_m) \mapsto (t_1, .., t_m)$ denotes the assignments $\mathsf{v}_i \mapsto t_i$, $1 \leq i \leq m$. A constraint $c = (R_S, S)$ is *satisfied* iff $S \mapsto (t_1, .., t_m)$ and $(t_1, .., t_m) \in R_S$. A *solution* of a CSP is an assignment for all variables in V satisfying all constraints in C. Constraint solvers typically use backtracking search to explore assignments in a search tree. In a search tree, each node represents an assignment to some variables, child nodes are obtained by selecting an unassigned variable and having a child node for each possible value for this variable, and the root node is empty. Every time a variable is assigned a value, a propagation stage is executed, pruning the set of values for the other variables, i.e., enforcing a certain type of local consistency such as domain consistency. An n-ary constraint $c = (R_S, S)$ is *domain consistent* iff whenever a variable $\mathsf{v}_i \in S$ is assigned a value $t_i \in dom(\mathsf{v}_i)$, there exist compatible values in the domains of all the other variables $t_j \in dom(\mathsf{v}_j)$ for all $1 \leq j \leq n,\ j \neq i$ such that $(t_1, .., t_n) \in R_S$, forming a *support* for $\mathsf{v}_i \mapsto t_i$.

Grammar Constraints

We will consider constraints requiring that values taken by the sequence of variable in its scope belong to a formal language generated by a context-free grammar or accepted by an automaton. A *context-free grammar* (CFG; [5]) is formally defined as a quadruple $\mathcal{G} = (N, \Sigma, P, S)$, where N is a finite set of *nonterminal* symbols, Σ is a finite set of *terminal* symbols (disjoint from N), $P \subseteq N \times (N \cup \Sigma)^*$ is a set of *production* rules, and the distinguished *start nonterminal* $S \in N$. We often omit to specify the complete quadruple and only provide the set of productions using the following conventions: capital letters denote nonterminals in N, lowercase letters denote terminals in Σ, and υ and ω denote a sequence of letters called *string*. We also assume that S is the start nonterminal. Moreover, productions $(A, \omega) \in P$ can be written as $A ::= \omega$, and productions $(A, \omega_1), .., (A, \omega_m) \in P$ can be written as $A ::= \omega_1 \mid .. \mid \omega_m$. We write $\upsilon_1 A \upsilon_2 \Rightarrow_{\mathcal{G}} \upsilon_1 \omega \upsilon_2$ iff $(A, \omega) \in R$, $\omega_1 \Rightarrow^*_{\mathcal{G}} \omega_m$ iff there exists a sequence of strings $\omega_2, .., \omega_{m-1}$ such that $\omega_i \Rightarrow_{\mathcal{G}} \omega_{i+1}$ for all $1 \leq i < m$. For $\omega_1 \Rightarrow^*_{\mathcal{G}} \omega_m$ we say that ω_1 *generates* ω_m and ω_1 is *derived from* ω_m. The *language produced by* $\mathcal{G}$ is the set of strings $L_{\mathcal{G}} = \{\omega \in \Sigma^* \mid S \Rightarrow^*_{\mathcal{G}} \omega\}$. A CFG is in *Chomsky normal form* iff all productions are of the form $A ::= a$ or $A ::= BC$. Every CFG $\mathcal{G}$ such that the empty string ε is not generated by $\mathcal{G}$ can be transformed into a grammar $\mathcal{H}$ such that $L_{\mathcal{G}} = L_{\mathcal{H}}$ and $\mathcal{H}$ is in Chomsky normal form. Transformations are described in most textbooks on automata theory, such as [14], with at most a linear increase in the size of the grammar. Given a CFG $\mathcal{G}$, the GRAMMAR constraint GRAMMAR($\mathcal{G}, [\mathsf{v}_1, .., \mathsf{v}_n]$) is satisfied by just those assignments to the sequence of variables $(\mathsf{v}_1, .., \mathsf{v}_n)$ which belong to the language produced by $\mathcal{G}$ [27, 25].

The REGULAR constraint [23] REGULAR($\mathcal{G}, [\mathsf{v}_1, .., \mathsf{v}_n]$) is a special case of the GRAMMAR constraint, i.e., it accepts just those assignments to the sequence of variables $(\mathsf{v}_1, .., \mathsf{v}_n)$ which belong to the regular language, that is, produced by a regular grammar $\mathcal{G}$. As we are recognising only strings are of a fixed length, a transformation of grammar constraints into regular constraints which may increase the space required to represent the constraint is presented in [17]. Regular languages are strictly contained within context-free languages, and can be specified with productions of the form $A ::= t$ or $A ::= tB$. Alternatively, regular languages can be specified by means of an automaton. A *deterministic finite automaton* (DFA) M is a quintuple $(Q, \Sigma, \delta, q_0, F)$, where Q is a finite, non-empty set of *states*, Σ is a finite, non-empty input alphabet, δ is a transition function $Q \times \Sigma \rightarrow Q$, q_0 is the initial state, and F is a set of accepting states. A DFA takes a sequence of input symbols ω as input, each symbol $t \in \Sigma$ causes M to perform a transition from its current state q to a new state $\delta(q, t)$, where M starts off in the state q_0. The input ω is *recognised* by M iff ω causes M to transition from q_0 in one of the accepting states. The *language recognised by* M is the set of inputs $L_M = \{\omega \in \Sigma^* \mid M \text{ recognises } \omega\}$. Given a DFA M, the REGULAR constraint REGULAR($M, [\mathsf{v}_1, .., \mathsf{v}_n]$) is satisfied on just those assignments to the sequence of variables $(\mathsf{v}_1, .., \mathsf{v}_n)$ which belong to the language recognised by M.

The PRECEDENCE constraint [18] is a special case of the REGULAR constraint. It is used for breaking symmetries of interchangeable values in a CSP. A pair of values are *interchangeable* if we can swap them in any solution. Pairwise precedence of s over t in a sequence of variables $(\mathsf{v}_1, .., \mathsf{v}_n)$, denoted as PRECEDENCE($[s, t], [\mathsf{v}_1, .., \mathsf{v}_n]$), holds iff whenever there exists j such that $\mathsf{v}_j \mapsto t$, then there must exist $i < j$ such that $\mathsf{v}_i \mapsto s$. Many CSPs, however, involve multiple interchangeable values, not just two. For instance, when we assign colours to vertices in the *graph colouring* problem, all values are interchangeable. Then, the PRECEDENCE constraint PRECEDENCE($[t_1, .., t_m], [\mathsf{v}_1, .., \mathsf{v}_n]$) holds iff $\min(\{i \mid \mathsf{v}_i \mapsto t_k\} \cup \{n+1\}) < \min(\{i \mid \mathsf{v}_i \mapsto t_\ell\} \cup \{n+2\})$ for all $1 \leq k < \ell < m$. A similar idea for breaking symmetries in the *planning* domain is presented in [15].

Modelling Multi-valued Variables

We want to model GRAMMAR, REGULAR, and PRECEDENCE constraints with ASP. Their encoding requires access to information on the variables $\mathsf{v}_1, .., \mathsf{v}_n$ in the scope of the constraint, i.e., information on possible assignments $\mathsf{v}_i \mapsto t$ with $t \in dom(\mathsf{v})$. In this paper, we will follow previous work on modelling CSP with ASP [11], and maintain a consistent set of assignments through translation into ASP together with our constraint encodings. Each possible assignment $\mathsf{v}_i \mapsto t$ is represented by an atom $[\![\mathsf{v}_i \mapsto t]\!]$. It is false iff t has been pruned from the domain of v_i. Conversely, it is true iff $\mathsf{v}_i \mapsto t$. The following rules, collected in $encode([\mathsf{v}_1, .., \mathsf{v}_n])$, ensure that one and only one value $t_{i_j} \in \Sigma$ can be assigned, for $1 \leq i \leq n$ and $dom(\mathsf{v}_i) = \{t_{i_1}, .., t_{i_m}\}$.

$$\{[\![\mathsf{v}_i \mapsto t_{i_1}]\!], .., [\![\mathsf{v}_i \mapsto t_{i_m}]\!]\} \leftarrow$$
$$\bot \leftarrow [\![\mathsf{v}_i \mapsto t_{i_j}]\!], [\![\mathsf{v}_i \mapsto t_{i_k}]\!], j \neq k$$
$$\bot \leftarrow not\ [\![\mathsf{v}_i \mapsto t_{i_1}]\!], .., not\ [\![\mathsf{v}_i \mapsto t_{i_m}]\!]$$

Although there are $\mathcal{O}(nm^2)$ nogoods resulting from this ASP model, unit-propagation is performed on these nogoods in $\mathcal{O}(nm)$. A more compact $\mathcal{O}(nm)$ representation that propagates in $\mathcal{O}(nm)$ time is presented in [11, 28].

3 Modelling the Grammar Constraint

We show here how to model GRAMMAR$(\mathcal{G}, [\mathsf{v}_1, .., \mathsf{v}_n])$ with ASP in a very straightforward way, based on the well known CYK parser [29] which requires the CFG to be in Chomsky normal form. Our encoding of the CYK parser constructs a parsing table T where each $A \in T[i, j]$ is a nonterminal symbol that is derived from a substring ω of j symbols starting at the ith symbol such that the assignment $(\mathsf{v}_i, \mathsf{v}_{i+1}, .., \mathsf{v}_{i+j-1}) \mapsto \omega$ is possible. Our $\mathcal{O}(|P|n^3)$ sized ASP model, denoted as $encode(\mathcal{G})$, is as follows:

1. We introduce new atoms $A(i, j)$ which are true iff $A \in A[i, j]$. A consistent assignment to $A(i, j)$ is enforced by the rules that follow below.
2. Each production of the form $A ::= t$ is encoded by
   $$A(i, 1) \leftarrow [\![\mathsf{v}_i \mapsto t]\!]$$
   which states that A can be derived from the ith symbol if $\mathsf{v}_i \mapsto t$, i.e., the ith symbol is t.
3. Each production of the form $A ::= BC$ is encoded by
   $$A(i, j) \leftarrow B(i, k), C(i+k, j-k).$$
   Intuitively, above rule states that A is derived from the string starting at the ith symbol of length j if B is derived from the substring starting at the ith symbol of length k, and C is derived from the substring starting at the $i+k$th symbol of length $j-k$. In other words, k splits the string generated by A into the substrings B and C.
4. Finally, the condition that the start nonterminal S has to be derived from the input string is captured by
   $$\bot \leftarrow not\ S(1, n)$$
   which expresses that every answer set of the resulting ASP model contains $S(1, n)$.

▶ **Theorem 1.** *The assignments satisfying* GRAMMAR$(\mathcal{G}, [\mathsf{v}_1, .., \mathsf{v}_n])$ *correspond one-to-one to the answer sets of* $encode(\mathcal{G}) \cup encode([\mathsf{v}_1, .., \mathsf{v}_n])$.

The theorem follows from the proof of correctness of the CYK parser. Unfortunately, our straightforward ASP model is not very efficient from a theoretical point of view, i.e., it does not prune all possible values.

▶ **Example 2.** Consider the following CFG $\mathcal{G}$ given through the productions $S ::= SA \mid AS \mid b$ and $A ::= a$. Suppose the input sequence of length 2, $(\mathsf{v}_1, \mathsf{v}_2)$ with $dom(\mathsf{v}_1) = dom(\mathsf{v}_2) = \{a, b, c\}$. Our model $encode(\mathcal{G})$ comprise the following rules

$$A(1,1) \leftarrow [\![\mathsf{v}_1 \mapsto a]\!] \qquad S(1,1) \leftarrow [\![\mathsf{v}_1 \mapsto b]\!] \qquad S(1,2) \leftarrow S(1,1), A(2,1)$$
$$A(2,1) \leftarrow [\![\mathsf{v}_2 \mapsto a]\!] \qquad S(2,1) \leftarrow [\![\mathsf{v}_2 \mapsto b]\!] \qquad S(1,2) \leftarrow A(1,1), S(2,1)$$
$$\bot \leftarrow not\ S(1,2)$$

Although the value c does not appear in a satisfying assignment for GRAMMAR$(\mathcal{G}, [\mathsf{v}_1, \mathsf{v}_2])$, unit-propagation on $\Delta(encode(\mathcal{G}) \cup encode([\mathsf{v}_1, \mathsf{v}_2]))$ does not prune c from the domains.

A SAT model of the GRAMMAR constraint, based on and-or-graphs, such that unit-propagation achieves domain consistency on the original constraint was presented in [26]. To achieve a similar result, we revise our ASP model that we now denote $encode^{DC}(\mathcal{G})$. The idea is that $A(i,j)$ is set to true iff the nonterminal A participates in a successful parsing of the input string starting from the start nonterminal S.

1. We introduce atoms $A(i,j)$ which are true iff $S \Rightarrow^*_{\mathcal{G}} \omega_1 A \omega_2 \Rightarrow^*_{\mathcal{G}} \omega$ for some ω_1, ω_2, where ω is the input string and A is derived by the substring starting from i of length j. Similarly, we introduce atoms $\omega_{BC}(i,j)$ which indicate whether $S \Rightarrow^*_{\mathcal{G}} \omega_1 BC \omega_2 \Rightarrow^*_{\mathcal{G}} \omega$ and the two nonterminals BC is derived by the substring starting from i of length j. A consistent assignment to $A(i,j)$, $\omega_{BC}(i,j)$ respectively, is enforced by the rules that follow below.
2. Each production of the form $A ::= t$ is now encoded by
   $$\{A(i,1)\} \leftarrow [\![\mathsf{v}_i \mapsto t]\!]\ .$$
   To ensure that unit-propagation prunes all possible values, we capture the condition that for each assignment $\mathsf{v}_i \mapsto t$ there exist a nonterminal A that generates t at the ith symbol and participates in a successful parsing starting from S. Let $A_1, .., A_m$ be all nonterminals such that $A_\ell ::= t$ for $1 \leq \ell \leq m$. We encode *support* for $\mathsf{v}_i \mapsto t$ by
   $$\bot \leftarrow [\![\mathsf{v}_i \mapsto t]\!], not\ A_1(i,1), .., not\ A_m(i,1)\ .$$
3. Each production of the form $A ::= BC$ is encoded by
   $$\{A(i,j)\} \leftarrow \omega_{BC}(i,j) \qquad \{\omega_{BC}(i,j)\} \leftarrow B(i,k), C(i{+}k, j{-}k)$$
   stating that if a production for A applies (e.g., the string ω_{BC}), then A may or may not be in a parsing starting from S. To ensure that unit-propagation prunes all possible values, we have to encode the condition that whenever $\omega_{BC}(i,j)$ is true then there must be a nonterminal A that generates BC, i.e., $S \Rightarrow^*_{\mathcal{G}} \omega_1 A \omega_2 \Rightarrow_{\mathcal{G}} \omega_1 BC \omega_2 \Rightarrow^*_{\mathcal{G}} \omega$ for some ω_1, ω_2, where ω is the input string and $A ::= BC$. Let $A_1, .., A_m$ be all such nonterminal A. Then we encode this condition by
   $$\bot \leftarrow \omega_{BC}(i,j), not\ A_1(i,j), .., not\ A_m(i,j)\ .$$
   Similarly, we encode support for each nonterminal $A' \in N \setminus \{S\}$, i.e., A' must be in the right-hand-side of a production, say the string ω_{BC} for $A' = B$ or $A' = C$, such that $S \Rightarrow^*_{\mathcal{G}} \omega_1 \omega_{BC} \omega_2 \Rightarrow_{\mathcal{G}} \omega_1 BC \omega_2 \Rightarrow^*_{\mathcal{G}} \omega$ for some ω_1, ω_2, where ω is the input string. Let $\omega_{BC,1}, .., \omega_{BC,m}$ be all such strings ω_{BC}. Then we encode this condition by
   $$\bot \leftarrow A'(i,j), not\ \omega_{BC,1}(i_1, j_1), .., not\ \omega_{BC,m}(i_m, j_m)\ .$$
   Observe that for all $1 \leq \ell \leq m$ we have either $i_\ell = i$ or $j_\ell = j$. Hence, there are only $\mathcal{O}(|P|n^3)$ rules from this item, i.e., $encode^{DC}(\mathcal{G})$ does not increase the space complexity over $encode(\mathcal{G})$.
4. The condition that the starting nonterminal S has to generate the input string remains unchanged.
   $$\bot \leftarrow not\ S(1,n)$$

Altogether, the rules in $encode^{DC}(\mathcal{G})$ enforce that whenever $A(i,j)$ is in an answer set then the nonterminal A is used to generate the substring at the ith symbol of length j in a successful parsing starting from S. This observation also applies to $\omega_{BC}(i,j)$.

▶ **Example 3.** Consider again the CFG $\mathcal{G}$ from Example 2, again applied to $(\mathsf{v}_1, \mathsf{v}_2)$ with $dom(\mathsf{v}_1) = dom(\mathsf{v}_2) = \{a, b, c\}$. Our ASP model $encode^{DC}(\mathcal{G})$ comprises the following rules

$$\begin{array}{lll}
\{A(1,1)\} \leftarrow [\![\mathsf{v}_1 \mapsto a]\!] & \{S(1,1)\} \leftarrow [\![\mathsf{v}_1 \mapsto b]\!] & \{\omega_{SA}(1,2)\} \leftarrow S(1,1), A(2,1) \\
\{A(2,1)\} \leftarrow [\![\mathsf{v}_2 \mapsto a]\!] & \{S(2,1)\} \leftarrow [\![\mathsf{v}_2 \mapsto b]\!] & \{\omega_{AS}(1,2)\} \leftarrow A(1,1), S(2,1) \\
\{S(1,2)\} \leftarrow \omega_{SA}(1,2) & \{S(1,2)\} \leftarrow \omega_{AS}(1,2) & \bot \leftarrow not\ S(1,2) \\
\bot \leftarrow \omega_{SA}(1,2), not\ S(1,2) & \bot \leftarrow A(1,1), not\ \omega_{AS}(1,2) & \bot \leftarrow S(1,1), not\ \omega_{SA}(1,2) \\
\bot \leftarrow \omega_{AS}(1,2), not\ S(1,2) & \bot \leftarrow A(2,1), not\ \omega_{SA}(1,2) & \bot \leftarrow S(2,1), not\ \omega_{AS}(1,2) \\
\bot \leftarrow [\![\mathsf{v}_1 \mapsto a]\!], not\ A(1,1) & \bot \leftarrow [\![\mathsf{v}_1 \mapsto b]\!], not\ S(1,1) & \bot \leftarrow [\![\mathsf{v}_1 \mapsto c]\!] \\
\bot \leftarrow [\![\mathsf{v}_2 \mapsto a]\!], not\ A(2,1) & \bot \leftarrow [\![\mathsf{v}_2 \mapsto b]\!], not\ S(2,1) & \bot \leftarrow [\![\mathsf{v}_2 \mapsto c]\!]
\end{array}$$

Unit-propagation on $\Delta(encode^{DC}(\mathcal{G}) \cup encode([\mathsf{v}_1, \mathsf{v}_2]))$ prunes the value c from the domains, i.e., sets $[\![\mathsf{v}_1 \mapsto c]\!]$ and $[\![\mathsf{v}_2 \mapsto c]\!]$ to false.

Unit-propagation of an ASP solver provides an efficient propagator for free, i.e., unit-propagation on our revised encoding is enough to achieve domain consistency.

▶ **Theorem 4.** *The assignments satisfying* GRAMMAR$(\mathcal{G}, [\mathsf{v}_1, .., \mathsf{v}_n])$ *correspond one-to-one to the answer sets of* $encode^{DC}(\mathcal{G}) \cup encode([\mathsf{v}_1, .., \mathsf{v}_n])$*. Unit-propagation on* $\Delta(encode^{DC}(\mathcal{G}) \cup encode([\mathsf{v}_1, .., \mathsf{v}_n]))$ *enforces domain consistency on* GRAMMAR$(\mathcal{G}, [\mathsf{v}_1, .., \mathsf{v}_n])$ *in* $\mathcal{O}(|P|n^3)$ *down any branch of the search tree.*

The proof follows the one in [26], where an and-or-graph is created that is very similar to the structure (i.e., the body-atom graph [20]) of $encode^{DC}(\mathcal{G})$, and subsequently, a SAT formula is constructed in a fashion that resembles the Clark's completion of our ASP model. Note that the propagators for the GRAMMAR constraint presented in [27, 25] have a similar overall complexity.

An extension which is sometimes useful in practice but goes slightly outside CFGs considers restrictions on some of the productions [26], e.g., in *shift scheduling* we want that *an employee works on an activity for a minimum of one hour*. Then, for a production of the form $A ::= BC$ and conditions represented by $\mathrm{c}_A(i,j), \mathrm{c}_B(i,j), \mathrm{c}_C(i,j)$ we restrict its application by encoding $A ::= BC$ in $encode(\mathcal{G})$ by

$$A(i,j) \leftarrow B(i,k), C(i+k, j-k), \mathrm{c}_A(i,j), \mathrm{c}_B(i,k), \mathrm{c}_C(i+k, j-k)\ .$$

This rule encodes that the nonterminal A generates a string starting at the ith symbol of length j if (1) the condition $\mathrm{c}_A(i,j)$ is satisfied, (2) B generates a string starting at the ith symbol of length k such that the condition $\mathrm{c}_B(i,k)$ is satisfied, and (3) C generates a string starting at the $i+k$th symbol of length $j-k$ such that the condition $\mathrm{c}_C(i+k, j-k)$ is satisfied. Similarly, productions of the form $A ::= t$ can be constrained. The changes to $encode^{DC}(\mathcal{G})$ are symmetric.

4 Modelling the Regular Constraint

In some cases, we only need a regular language, e.g., generated by a regular grammar, to specify problem constraints. One important point about regular grammars is that each nonterminal $A \in T[i,j]$ is derived by a substring at the ith symbol to the nth symbol, i.e., $j = n$. Using this insight we encode a production of the form $A ::= tB$ in $encode(\mathcal{G})$ as

below, for $1 \leq i \leq n$.

$A(i, n-i+1) \leftarrow [\![\mathsf{v}_i \mapsto t]\!], B(i+1, n-i)$

The changes to $encode^{DC}(\mathcal{G})$ are symmetric. For regular languages, propagation is faster.

▶ **Theorem 5.** *If $\mathcal{G}$ is regular then unit-propagation on $\Delta(encode^{DC}(\mathcal{G}) \cup encode([\mathsf{v}_1, .., \mathsf{v}_n]))$ enforces domain consistency on* GRAMMAR$(\mathcal{G}, [\mathsf{v}_1, .., \mathsf{v}_n])$ *in $\mathcal{O}(|P|n)$ down any branch of the search tree.*

Recall that each regular language $\mathcal{L}$ can also be specified by means of a DFA $M = (Q, \Sigma, \delta, q_0, F)$ that accepts assignments to a sequence of variables iff it is a member of $\mathcal{L}$. One important advantage of using an automata-based representation is that it permits to compress the ASP model using standard techniques for automaton minimisation. An automata-based propagator for the REGULAR constraint was first proposed in [23], and a SAT model such that unit-propagation enforces domain consistency was presented in [2]. We show here how to model REGULAR$(M, [\mathsf{v}_1, .., \mathsf{v}_n])$ with ASP in a straightforward and easily maintainable way. Given M, we propose an ASP model, denoted *encode*(M), which represents all possible states the DFA can be in after processing i symbols. An accepted input string must generate a sequence of transformations starting at q_0 and ending in some finite state. Our encoding is as follows:

1. We introduce atoms $q_k(i)$ for each step i of M's processing, $0 \leq i \leq n$, and each state $q_k \in Q$ to indicate whether M is in state q_k after having processed the first i symbols.
2. Each transition $\delta(q_j, t) = q_k$ is encoded as follows, for $1 \leq i \leq n$.

   $q_k(i) \leftarrow q_j(i-1), [\![\mathsf{v}_i \mapsto t]\!]$

   Intuitively, whenever M is in state q_j after having processed the first $i-1$ symbols and M reads t as the ith symbol then M transitions to the state q_k in step i.
3. The condition that M must start processing in starting state q_0 is captured by

   $q_0(0) \leftarrow$

   which sets $q_0(0)$ unconditionally to true.
4. To represent that M must not finish processing in a rejecting state $q_{rej} \in Q \setminus F$, we post

   $\bot \leftarrow q_{rej}(n)$.

   Intuitively, it expresses the condition that no answer set contains $q_{rej}(n)$ for $q_{rej} \in Q \setminus F$.

To ensure that unit-propagation prunes all possible values, support has to be encoded. We extend *encode*(M) by one item, resulting in $encode^{DC}(M)$.

6. There is support for $\mathsf{v}_i \mapsto t_i$ whenever there exists a transition $\delta(q_j, t) = q_k$ from the state q_j to the state q_k at step i while reading t. To encode support for $\mathsf{v}_i \mapsto t$, we define auxiliary atoms $d(q_j, q_k, i)$ for each transition $\delta(q_j, t) = q_k$, for $1 \leq i \leq n$, by

   $d(q_j, q_k, i) \leftarrow q_j(i-1), q_k(i)$.

   Now, for each assignment $\mathsf{v}_i \mapsto t$, we encode the existence of such a support by

   $\bot \leftarrow [\![\mathsf{v}_i \mapsto t]\!], not\ d(q_{j_1}, q_{k_1}, i), not\ d(q_{j_2}, q_{k_2}, i), .., d(q_{j_m}, q_{k_m}, i)$.

   This rule is satisfied if either $\mathsf{v}_i \mapsto t$ has a support, or $[\![\mathsf{v}_i \mapsto t]\!]$ is false.

▶ **Theorem 6.** *The answer sets of encode$(M) \cup$ encode$([\mathsf{v}_1, .., \mathsf{v}_n])$ correspond one-to-one to assignments that satisfy* REGULAR$(M, [\mathsf{v}_1, .., \mathsf{v}_n])$*. Unit-propagation on $\Delta(encode^{DC}(M) \cup encode([\mathsf{v}_1, .., \mathsf{v}_n]))$ enforces domain consistency on* REGULAR$(M, [\mathsf{v}_1, .., \mathsf{v}_n])$ *in $\mathcal{O}(|\delta|n)$ down any branch of the search tree.*

The proof follows the one in [2], i.e., we can exploit the fact that Clark's completion of our ASP model results in the SAT formula presented in [2]. Note that the propagator for the REGULAR constraint proposed in [23] has a similar overall complexity.

5 Modelling the Precedence Constraint

For breaking value symmetry in a CSP, we only need a special case of the REGULAR constraint. Recall, PRECEDENCE$([t_1, .., t_m], [\mathsf{v}_1, .., \mathsf{v}_n])$ holds iff whenever $\mathsf{v}_j \mapsto t_\ell$ then there exists $\mathsf{v} \mapsto t_k$ such that $i < j$, for all $1 \leq k < \ell \leq m$. We can model the PRECEDENCE constraint with ASP. Our encoding, denoted $encode(pre[t_1, .., t_m])$, is as follows:

1. We introduce new atoms $taken(t_\ell, j)$ for each $t_\ell \in [t_1, .., t_m]$, and each position j, $0 \leq j \leq n$, to indicate whether $\mathsf{v}_i \mapsto t_\ell$ for some $i < j$.
2. For each value $t_\ell \in [t_1, .., t_m]$, we encode that it has been taken if $\mathsf{v}_i \mapsto t_\ell$, $1 \leq i \leq n$.
   $taken(t_\ell, i{+}1) \leftarrow [\![\mathsf{v}_i \mapsto t_j]\!]$
   To propagate the information to the other indices greater than i, our ASP model contains
   $taken(t_\ell, i{+}1) \leftarrow taken(t_\ell, i)$.
3. Finally, we post the condition that v_j cannot be assigned a value t_ℓ such that $t_\ell \in [t_1, .., t_m]$ if some value t_k has not been taken by a variable v_i such that $i < j$, $1 \leq k < \ell \leq m$.
   $\bot \leftarrow [\![\mathsf{v}_j \mapsto t_\ell]\!], not\ taken(t_k, i)$

Compared to our encodings for REGULAR constraints, our ASP model for the PRECEDENCE constraint is more economical with respect to auxiliary variables and rule size while unit-propagation can still enforce domain consistency.

▶ **Theorem 7.** *The assignments satisfying* PRECEDENCE$([t_1, .., t_m], [\mathsf{v}_1, .., \mathsf{v}_n])$ *correspond one-to-one to the answer sets of* $encode(pre[t_1, .., t_m]) \cup encode([\mathsf{v}_1, .., \mathsf{v}_n])$. *Down any branch of the search tree, unit-propagation on* $\Delta(encode(pre[t_1, .., t_m]) \cup encode([\mathsf{v}_1, .., \mathsf{v}_n]))$ *enforces domain consistency on* PRECEDENCE$([t_1, .., t_m], [\mathsf{v}_1, .., \mathsf{v}_n])$ *in* $\mathcal{O}(m^2 n)$.

6 Experiments on Shift Scheduling

We tested the practical utility of our ASP models of the GRAMMAR constraint on a set of shift-scheduling benchmarks [7]. The problem is to schedule employees in a company to activities subject to the following rules. An employee either works on activity a_i, has a break b, has lunch l, or rests r. If the company business is open, an employee works on an activity for a minimum of one hour and can change activities after a fifteen minutes break or one hour lunch. Break and lunch both are scheduled between periods of work. A part-time employee works at least three hours and at most six hours plus a fifteen minutes break, while a full-time employee works at least six hours and at most eight hours plus an hour and a half for the lunch and the breaks. Our goal is to minimise the number of hours worked. The schedule of an employee is modelled with a sequence of 96 variables, each represents a time slot of 15 minutes, that must be generated by the following CFG $\mathcal{G}$.

$S ::= RFR, \mathsf{c}_F(i,j) \equiv 30 \leq j \leq 38 \quad F ::= PLP \quad L ::= lL \mid l, \mathsf{c}_L(i,j) \equiv j = 4$
$S ::= RPR, \mathsf{c}_F(i,j) \equiv 13 \leq j \leq 24 \quad P ::= WbW \quad W ::= A_i, \mathsf{c}_W(i,j) \equiv j \geq 4$
$A_i ::= a_i A_i \mid a_i, \mathsf{c}_{A_i}(i,j) \equiv open(i) \quad R ::= rR \mid r$

Related work in [26] converts the *shift scheduling* problem into a SAT model, denoted SAT. Experiments also consider our two ASP models of the GRAMMAR constraint: $encode(\mathcal{G})$ and $encode^{DC}(\mathcal{G})$. We denote these models as ASP and ASP-DC, respectively. A bottom-up ASP grounder such as GRINGO (3.0.3; [24]) can be employed to instantiate our models. Then, the grounder simulates a CYK parser, i.e., it constructs all possible parsings for all possible subsequences of the input string. However, as the CYK parser, it also instantiates productions that cannot participate in a successful parsing, i.e., productions which are uninteresting for us. We have implemented a grounder for the special purpose of this benchmark based on

Table 1 Results on shift scheduling; $|A|$: number of activities; #: problem number; m: number of employees; number of worked hours (boldfaced if best solution found amongst the different methods)

$\|A\|$	#	m	ASP	ASP-DC	SAT	$\|A\|$	#	m	ASP	ASP-DC	SAT
1	2	4	**26.00**	26.25	**26.00**	2	1	5	**25.00**	**25.00**	**25.00**
1	3	6	**37.25**	37.50	37.50	2	2	10	**58.00**	58.75	59.25
1	4	6	**38.00**	**38.00**	**38.00**	2	3	6	**39.50**	40.25	**39.50**
1	5	5	**24.00**	**24.00**	**24.00**	2	4	11	**68.25**	68.50	69.00
1	6	6	**33.00**	**33.00**	**33.00**	2	5	4	**24.50**	24.75	25.50
1	7	8	**49.00**	**49.00**	**49.00**	2	6	5	**28.25**	29.25	28.50
1	8	3	**20.50**	**20.50**	**20.50**	2	8	5	**32.00**	32.75	32.25
1	10	9	**54.00**	54.25	54.25	2	9	3	**19.00**	**19.00**	**19.00**
						2	10	8	57.25	57.75	**57.00**

the algorithm in [26]. This will allow for unit-propagation in the ASP model to achieve a strong type of local consistency close to domain consistency. Experiments were run with the system CLASP (1.3.5; [24]) on a 2.00 GHz PC under Linux, where each run was limited to 3600 sec time and 1 GB RAM. Table 1 presents the results for 17 satisfiable instances of the benchmark involving one or two activities. The solver returned a feasible solution for all instances regardless of the model after a few seconds, and subsequently optimised the solution, where no model performs significantly better that the other. However, we can draw a few conclusions. First, the ASP model $encode(\mathcal{G})$ is strong enough in our setting, i.e., enforcing domain consistency does not increase runtime. And, second, we do not pay any penalty for using our straightforward, easily maintainable ASP models vs the SAT model.

7 Experiments on Graph Colouring

A *colouring* of a graph (V, E) is a a mapping c from V to $\{1, .., k\}$ such that $c(v) \neq c(w)$ for every edge $(v, w) \in E$ with a given number k of colours. The *graph colouring* problem is to determine the existence of a colouring. Our experiments consider different options for breaking value symmetry between the colours: The options REGULAR and REGULAR-DC use our ASP model for a DFA-based encoding of the PRECEDENCE constraint. The option ALL uses our ASP model for the PRECEDENCE constraint to break all value symmetry. We denote PAIRWISE our ASP model of the method [18] which posts PRECEDENCE constraints between pairwise interchangeable values. The option NONE breaks no symmetry while GENERIC employs the preprocessor SBASS (1.0; [24]) for symmetry breaking in terms of detected symmetry generators [10]. We experimented on random *graph colouring* instances, but restricted ourselves to 3-, 4- and 5-colourings, when we noticed that the relative performance of symmetry breaking increased with each additional colour (exponentially with the number of colours). For each of the k-colouring experiments we generated 600 instances around the phase transition density with 400, 150, 75 vertices, respectively. All tests were run with the system CLASP (1.3.2) on a 2.00 GHz PC under Linux, where each run was limited to 600 s time and 1 GB RAM. The results on 3-, and 5- colourings shown in Figure 1 indicate that all symmetry breaking techniques considered in our study are effective, i.e., improve runtime. The data on any colouring clearly suggest that the GENERIC and REGULAR methods are inferior to enforcing value precedence through the PRECEDENCE constraint using PAIRWISE and ALL, or REGULAR-DC, where REGULAR-DC outperforms REGULAR due to the strong type of local consistency achieved in REGULAR-DC. For the 3-colouring case, ALL gives a significantly better improvement compared to PAIRWISE. For the 5-colouring case the

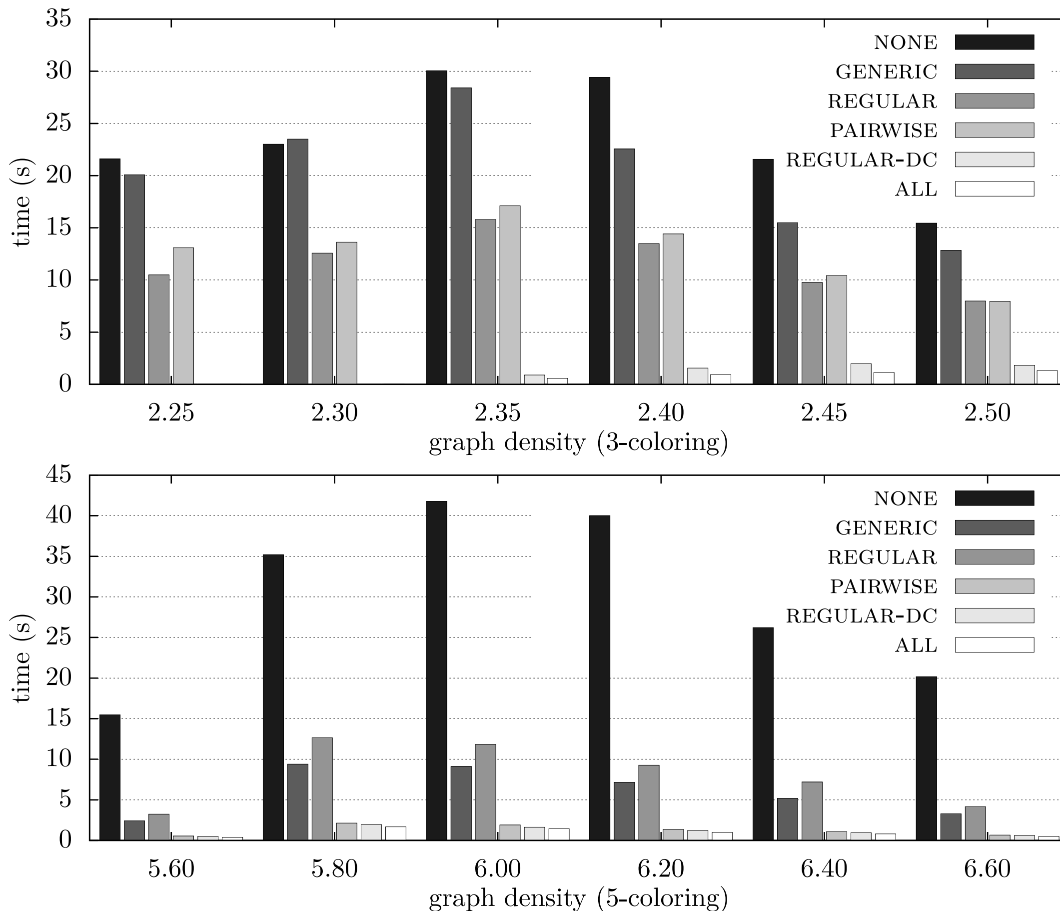

Figure 1 Histogram of the average time required by different symmetry breaking approaches.

same conclusion can be drawn, albeit less convincing. (5-colouring instances have fewer vertices, i.e., variables. This can improve propagation between PRECEDENCE constraints in PAIRWISE.) The overall conclusion from our graph colouring experiments is that breaking all value symmetry enforcing *precedence* is most effective.

8 Conclusions

Our work addresses modelling and solving CSP with ASP. We have presented ASP models of GRAMMAR and related constraints specified via grammars or automata. Finally, we have given an ASP model for the PRECEDENCE constraint. All our encodings are straightforward and easily maintainable without paying any penalty vs related SAT models. Unit-propagation of an ASP solver can prune all values, i.e., unit-propagation can achieve domain consistency on the original constraint. Future work concerns *lazy* modelling techniques [22] and ASP encodings of further constraints useful for modelling and solving CSP.

Acknowledgements We would like to thank Claude-Guy Quimper and Nina Narodytska for providing us with the benchmark data. NICTA is funded by the Department of Broadband, Communications and the Digital Economy, and the Australian Research Council.

References

1 R. Axelsson, K. Heljanko, and M. Lange. Analyzing context-free grammars using an incremental sat solver. In *Proceedings of ICALP'08*, pages 410–422, 2008.
2 F. Bacchus. GAC via unit propagation. In *Procedings of CP'07*, pages 133–147. Springer, 2007.
3 C. Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2003.
4 A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors. *Handbook of Satisfiability*. IOS Press, 2009.
5 N. Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, 2:113–124, 1956.
6 K. Clark. Negation as failure. In *Logic and Data Bases*, pages 293–322. Plenum Press, 1978.
7 M.-C. Côté, B. Gendron, C.-G. Quimper, and L.-M. Rousseau. Formal languages for integer programming modeling of shift scheduling problems. *Constraints*, 16:54–76, 2011.
8 R. Dechter. *Constraint Processing*. Morgan Kaufmann, 2003.
9 A. Dovier, A. Formisano, and E. Pontelli. A comparison of CLP(FD) and ASP solutions to NP-complete problems. In *Proceedings of ICLP'05*, pages 67–82. Springer, 2005.
10 C. Drescher, O. Tifrea, and T. Walsh. Symmetry-breaking answer set solving. In *Proceedings of ICLP'10, ASPOCP'10 Workshop*, 2010.
11 C. Drescher and T. Walsh. A translational approach to constraint answer set solving. *Theory and Practice of Logic Programming*, 10(4-6):465–480, 2010.
12 E. Erdem and V. Lifschitz. Tight logic programs. *Theory and Practice of Logic Programming*, 3(4-5):499–518, 2003.
13 M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. Conflict-driven answer set solving. In *Proceedings of IJCAI'07*, pages 386–392. AAAI Press/MIT Press, 2007.
14 J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.
15 X. Jia, J.-H. You, and L.-Y. Yuan. Adding domain dependent knowledge into answer set programs for planning. In *Proceedings of ICLP'04*, pages 400–415. Springer, 2004.
16 G. Katsirelos, S. Maneth, N. Narodytska, and T. Walsh. Restricted global grammar constraints. In *Proceedings of CP'09*, pages 501–508. Springer, 2009.
17 G. Katsirelos, N. Narodytska, and T. Walsh. Reformulating global grammar constraints. In *Proceedings of CPAIOR'09*, pages 132–147. Springer, 2009.
18 Y. C. Law and J. H.-M. Lee. Global constraints for integer and set value precedence. In *Proceedings of CP'04*, pages 362–376. Springer, 2004.
19 J. Lee. A model-theoretic counterpart of loop formulas. In *Proceedings of IJCAI'05*, pages 503–508. Professional Book Center, 2005.
20 T. Linke. Suitable graphs for answer set programming. In *Proceedings of ASP'03*, pages 15–28, 2003.
21 I. Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25(3-4):241–273, 1999.
22 O. Ohrimenko and P. J. Stuckey. Modelling for lazy clause generation. In *CATS*, pages 27–37. Australian Computer Society, 2008.
23 G. Pesant. A regular language membership constraint for finite sequences of variables. In *Proceedings of CP'04*, pages 482–495. Springer, 2004.
24 `http://potassco.sourceforge.net/`.
25 C.-G. Quimper and T. Walsh. Global grammar constraints. In *Proceedings of CP'06*, pages 751–755. Springer, 2006.

26 C.-G. Quimper and T. Walsh. Decompositions of grammar constraints. In *Proceedings of AAAI'08*, pages 1567–1570. AAAI Press, 2008.

27 M. Sellmann. The theory of grammar constraints. In *Proceedings of CP'06*, pages 530–544. Springer, 2006.

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

29 D. H. Younger. Recognition and parsing of context-free languages in time n^3. *Information and Control*, 10(2):372–375, 1967.

Hybrid ASP

Alex Brik[1] and Jeffrey B. Remmel[2]

1 **Department of Mathematics, UC San Diego, U.S.A**
2 **Department of Mathematics, UC San Diego, U.S.A**

Abstract

This paper introduces an extension of Answer Set Programming (ASP) called Hybrid ASP which will allow the user to reason about dynamical systems that exhibit both discrete and continuous aspects. The unique feature of Hybrid ASP is that it allows the use of ASP type rules as controls for when to apply algorithms to advance the system to the next position. That is, if the prerequisites of a rule are satisfied and the constraints of the rule are not violated, then the algorithm associated with the rule is invoked.

1998 ACM Subject Classification I.2.3 Deduction and Theorem Proving

Keywords and phrases answer set programming, hybrid systems, modeling and simulation

Digital Object Identifier 10.4230/LIPIcs.ICLP.2011.40

1 Introduction

The purpose of this paper is to introduce an extension of Answer Set Programming (ASP) which we call *Hybrid ASP*. The goal of Hybrid ASP is to allow the user to reason about dynamical systems that exhibit both discrete and continuous aspects. The unique feature of Hybrid ASP is that Hybrid ASP rules can be thought of as general input-output devices. In particular, Hybrid ASP programs allow the user to include ASP type rules that act as controls for when to apply a given algorithm to advance the system to the next position.

Modern computational models and simulations such as the model of dog's heart described in [6], the model of tsunami propagation described in [3], and the model of internal tides within Monterey Bay and the surrounding area described in [5] rely on existing PDE solvers and ODE solvers to determine the values of parameters. Such simulations proceed by invoking appropriate algorithms to advance a system to the next state, which is often distanced by a short time interval into the future from the current state. In this way, a simulation of continuously changing parameters is achieved, although the simulation itself is a discrete system. The parameter passing mechanisms and the logic for making decisions regarding what algorithms to invoke and when are part of the ad-hoc control algorithm. Thus the laws of a system are implicit in the ad-hoc control software.

On the other hand, action languages [2] which are also used to model dynamical systems allow the users to describe the laws of a system explicitly. Initially action languages did not allow simulation of the continuously changing parameters. This severely limited applicability of such languages. Recently, Chintabathina introduced an action language H [1] where he proposed an elegant approach to modeling continuously changing parameters. That is, a program in H describes a state transition diagram of a system where each state models a time interval where the parameter dynamics is a known function of time. However, the implementation of H discussed in [1] cannot use PDE solvers nor ODE solvers. This means that parameters governed by physical processes such as the distribution of heat or air flow, that cannot be described explicitly as functions of time and realistic simulations of

© Alex Brik, Jeffrey R. Remmel;
licensed under Creative Commons License NC-ND
Technical Communications of the 27th International Conference on Logic Programming (ICLP'11).
Editors: John P. Gallagher, Michael Gelfond; pp. 40–50

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

which often require sophisticated numerical methods, cannot be modeled using the current implementations of H.

Hybrid ASP is an extension of ASP that allows users to combine the strength of the ad-hoc approaches, i.e. the use of numerical methods to faithfully simulate physical processes, and the expressive power of ASP that provides the ability to elegantly model laws of a system. Hybrid ASP provides mechanisms to express the laws of the modeled system via hybrid ASP rules which can control execution of algorithms relevant for simulation.

In action languages like H, the goal is to compile an H program into a variant ASP program that can be processed with current variant ASP solvers. Our long term goal is to develop extensions of ASP solvers that can process Hybrid ASP programs. This would allow us to develop Hybrid ASP type extensions of action languages like H that could be compiled to Hybrid ASP programs which, in turn, would be processed by Hybrid ASP solvers.

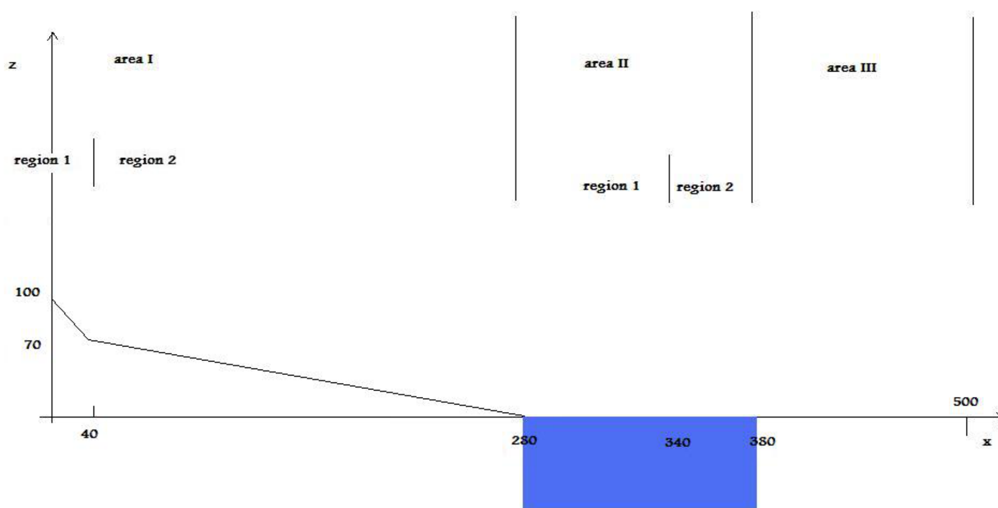

Figure 1 A cross section of the regions to be traversed by Secret Agent 00111.

In this paper, we shall present the basic definitions of Hybrid ASP programs and define an analogue of stable models for such programs. To help motivate our definitions, we shall consider the following toy example. Imagine that Secret Agent 00111 (the agent, for short) needs to move through a domain consisting of 3 areas: Area I, Area II, and Area III. The domain's vertical cross section is shown on the diagram **??**. Area I is a mountain, Area II is a lake, and Area III is a desert. Secret Agent 00111 needs to descend down the mountain in his car until he reaches the lake at which point the car can be reconfigured so that it can be used as a boat that can navigate across the lake. We shall assume that the lake has a water current moving with a constant speed of $5m/s$ which makes an angle $\frac{4\pi}{3}$ clockwise from the positive direction of the x-axis. If Secret Agent 00111 is pursued by evil agents on his trip down the mountain, he will accelerate the car at $4m/s^2$ in addition to the acceleration due to gravity. If he is not pursued by evil agents, then he will simply coast down the hill. Furthermore, if the agent is pursued by the evil agents then he will attempt to travel through the lake as fast as possible, always steering at a 90 degrees angle to the opposite shore. If the agent is not pursued by the evil agents then he would like to exit the lake at a point with a y-coordinate being close to the y-coordinate of the point of his entrance into the lake. To accomplish this, Secret Agent 00111 will be able to steer the boat in directions which

make various angles to the x-axis. Finally upon entering the desert Secret Agent 00111 can again begin to accelerate at $4m/s^2$.

The outline of this paper is as follows. In section 2 we shall introduce Hybrid ASP programs. In section 3 we shall show how H-ASP programs can be used to model a dynamical system for our secret agent problem. In section 4 we shall provide conclusions and directions for further research.

2 Hybrid ASP

The main feature of H-ASP is to view rules as general input-output devices. Informally, this means that given a set of parameter values and a set of facts associated with it, a rule may or may not produce one or more new parameter values and an associated fact. This ensures that H-ASP is suitable for defining control mechanisms for discrete time simulations that require the use of PDE solvers or ODE solvers or both.

A H-ASP program P will have an underlying parameter space S. For instance, in our secret agent example, imagine that we allow Secret Agent 00111 to make decisions every Δ seconds. Then one can think of describing the Secret Agent 00111's position and situation at time $k\Delta$ by a sequence of parameters $\mathbf{x}(k\Delta) = (x_0(k\Delta), x_1(k\Delta), x_2(k\Delta), \ldots, x_m(k\Delta))$ that specify both continuous parameters such as time, position, velocity, and acceleration as well as discrete parameters such as is the car configured as a car or as a boat. In more complicated simulations, the programmer may not know the value Δ ahead of time as the exact value of Δ may be determined by the needs of the parameter passing algorithms. Nevertheless, we shall always assume that any parameter passing algorithm advances time in some discrete time steps which may vary depending on the values of the input parameters. Thus in a H-ASP program, one can always think of the parameter x_0 as specifying time and the range of x_0 is either $\{k\Delta : k = 0, \ldots, n\}$ for some fixed n and $\Delta > 0$ or $\{k\Delta : k \in \mathbb{N}\}$ where $\mathbb{N}$ is the set of natural numbers $\{0, 1, 2, \ldots, \}$. In particular, for a finite H-ASP program, there will be no loss of generality in assuming that the range of x_0 is $\{k\Delta : k = 0, \ldots, n\}$ for some fixed n and $\Delta > 0$. In such a situation, we shall always write an element of S in the form $\mathbf{x} = (k\Delta, x_1(k\Delta), \ldots, x_m(k\Delta))$ for some k. However, in our general definitions, we shall just assume that elements of the parameter space S are of the form $\mathbf{p} = (t, x_1, \ldots, x_m)$ where t is time and we shall let $t(\mathbf{p})$ denote t and $x_i(\mathbf{p})$ denote x_i for $i = 1, \ldots, m$. We refer to the elements of S as *generalized positions*. A H-ASP program will also have an underlying set of atoms At. Then the underlying universe of the program will be $At \times S$.

If $M \subseteq At \times S$, then we let $\widehat{M} = \{\mathbf{x} \in S : (\exists a \in At)((a, \mathbf{p}) \in M)\}$. We will say that M satisfies $(a, \mathbf{p}) \in At \times S$, written $M \models (a, \mathbf{p})$, if $(a, \mathbf{p}) \in M$. For any element $(t, x_1, \ldots, x_m) \in S$, we let $W_M(t, x_1, \ldots, x_m) = \{a \in At : (a, (t, x_1, \ldots, x_m)) \in M\}$ and we shall refer to $W_M(t, x_1, \ldots, x_m)$ as the *world of M at the generalized position* $(t, x_1, \ldots, x_m)$. We let $T(S) = \{t : \exists \mathbf{p} \in S(t = t(\mathbf{p})\}$. We say that M is a **single trajectory model** if for each $t \in T(S)$, there is exactly one generalized position of the form $(t, x_1, \ldots, x_m)$ in $\widehat{M}$. If M is a single trajectory model, then we let $(t, x_1(t), \ldots, x_m(t))$ be the unique element of the form $(t, x_1, \ldots, x_m)$ in $\widehat{M}$ and we can write M as a disjoint union

$$M = \bigsqcup_{t \in T(S)} W_M(t, x_1(t), \ldots, x_m(t)) \times \{(t, x_1(t), \ldots, x_m(t))\}.$$

We will say that M is **multiple trajectory model** if for each $t \in T(S)$, there is at least one generalized position of the form $(t, x_1, \ldots, x_m)$ in $\widehat{M}$ and for some $t \in T(S)$, there are

at least two generalized positions of the form $(t, x_1, \ldots, x_m)$ in $\widehat{M}$. Single trajectory stable models are desirable when the objective of a computation is to obtain a trajectory satisfying certain constraints. For example, in the planning problem, the objective is to find a sequence of actions that achieves a predefined goal given an axiomatized initial situation [8]. Thus solving planning problem provides a motivation for considering single trajectory models. Multiple trajectory models are natural to consider when the objective of a computation is a set of conclusions that depend on all the possible trajectories. For instance such a model would be useful in reasoning about a best strategy for an agent acting within a dynamical system where the consequences of actions are non-deterministic. Thus multiple trajectory models are natural to consider in the context of dynamic programming problems (see [10] for an introduction to dynamic programming).

If we drop the requirement that there is a generalized position $(t, x_1, \ldots, x_m) \in \widehat{M}$ for each $t \in T(S)$ in the definition of single trajectory or multiple trajectory models, we get what we call *partial single trajectory* and *partial multiple trajectory* models. For example, if $T(S) = \{k\Delta : k = 0, \ldots, n\}$, then we may want to allow our parameter passing algorithms the flexibility of advancing time by multiple steps of Δ so that it may be the case that our set of rules will derive no information about what happens a certain time $k\Delta$ in which case $W_M(k\Delta, x_1(k\Delta), \ldots, x_m(k\Delta))$ will be empty for all generalized positions of the form $(k\Delta, x_1(k\Delta), \ldots, x_m(k\Delta)$. Thus it is quite natural to consider partial single trajectory and partial multiple trajectory models.

Given $M \subseteq At \times S$ and $B = a_1, \ldots, a_n, \neg b_1, ..., \neg b_m$, and $\mathbf{p} \in S$, we say that M satisfies B at the generalized position $\mathbf{p}$, written $M \models (B_i, \mathbf{p})$, if $(a_i, \mathbf{p}) \in M$ for $i = 1, \ldots, n$, and $(b_j, \mathbf{p}) \notin M$ for $j = 1, \ldots, m$. For B as above define $B^- = \neg b_1, ..., \neg b_m$. Note that if B_i is empty. then we assume that $M \models (B_i, \mathbf{p})$ automatically holds.

There are two types of rules in H-ASP programs.

Advancing Rules are of the form

$$\frac{B_1; B_2; \ldots; B_r : A, O}{a} \tag{1}$$

where A is an algorithm, each B_i is of the form $a_1^{(i)},\ldots,a_{n_i}^{(i)},\neg b_1^{(i)},...,\neg b_{m_i}^{(i)}$ where $a_1^{(i)},....a_{n_i}^{(i)}$, $b_1^{(i)},...,b_{m_i}^{(i)}$, and a are atoms, and $O \subseteq S^r$ is such that if $(\mathbf{p}_1, \ldots, \mathbf{p}_r) \in O$, then $t(\mathbf{p}_1) < \ldots < t(\mathbf{p}_r)$, $A(\mathbf{p}_1, \ldots, \mathbf{p}_r) \subseteq S$, and for all $\mathbf{q} \in A(\mathbf{p}_1, \ldots, \mathbf{p}_r)$, $t(\mathbf{q}) > t(\mathbf{p}_r)$. Here and in the next rule, we allow n_i or m_i to be equal to 0 for any given i. Moreover, if $n_i = m_i = 0$, then B_i is empty and we automatically assume that B_i is satisfied by any $M \subseteq At \times S$. We shall refer to O as the *constraint set* of the rule and the algorithm A as the *advancing algorithm* of the rule. The idea is that if $(\mathbf{p}_1, \ldots, \mathbf{p}_r) \in O$ and for each i, B_i is satisfied at the general position $\mathbf{p}_i$, then the algorithm A can be applied to $(\mathbf{p}_1, \ldots, \mathbf{p}_r)$ to produce a set of generalized positions O' such that if $\mathbf{q} \in O'$, then $t(\mathbf{q}) > t(\mathbf{p}_r)$ and $(a, \mathbf{q})$ holds. Advancing rules formalize reasoning steps in which new sets of parameter values are derived.

For instance consider the movement of an object through the 3 dimensional space $\mathbb{R}^3$ with a constant velocity $v = (v_1, v_2, v_3)$ where the parameter space S is defined so that $T(S) = \{k\Delta : k \in \mathbb{N}\}$. Then we can use the following rule to model object's movement: $\frac{:A,S}{T}$, where $A((t, x_1, x_2, x_3)) = \{(t + \Delta,\ x_1 + v_1 \cdot \Delta,\ x_2 + v_2 \cdot \Delta,\ x_3 + v_3 \cdot \Delta)\}$. Here T is a place holder atom for “true”. That is, since an advancing rule requires a conclusion, "T" is used to fulfill the requirement and has no significance otherwise.

Stationary rules are of the form

$$\frac{B_1; B_2; \dots; B_r : H, O}{a} \tag{2}$$

where each B_i is of the form $a_1^{(i)}$,...,$a_{n_i}^{(i)}$, $\neg b_1^{(i)}$,...,$\neg b_{m_i}^{(i)}$ where $a_1^{(i)}$,....$a_{n_i}^{(i)}$, $b_1^{(i)}$,...,$b_{m_i}^{(i)}$ and a are atoms, $O \subseteq S^r$ is such that if $(\mathbf{p}_1, \dots, \mathbf{p}_r) \in O$, then $t(\mathbf{p}_1) < \cdots < t(\mathbf{p}_r)$, and H is a Boolean algorithm such that for all $(\mathbf{p}_1, \dots, \mathbf{p}_r) \in O$, $H(\mathbf{p}_1, \dots, \mathbf{p}_r)$ is defined. We shall refer to O as the *constraint set* of the rule and the algorithm H as the *Boolean algorithm* of the rule.

The idea is that if $(\mathbf{p}_1, \dots, \mathbf{p}_r) \in O$ and for each i, B_i is satisfied at the generalized position $\mathbf{p}_i$, and $H((\mathbf{p}_1, \dots, \mathbf{p}_r))$ is true, then $(a, \mathbf{p}_r)$ holds. Stationary rules formalize reasoning steps where new facts are derived about a particular set of parameters that has already been derived. As an example consider the following. Suppose that we are considering a program where $T(S) = \{k\Delta : k = 0, \dots, n\}$, where we want to derive a single trajectory stable model, and where once an atom A holds at $(k\Delta, x_1(k\Delta), \dots, x_m(k\Delta)$, then A must hold at all generalized positions of the form $((k+2j)\Delta, x_1((k+2j)\Delta), \dots, x_m((k+2j)\Delta)$ where $j \geq 1$. To enforce this condition we can use the following stationary rule: $\frac{A; :H, O}{A}$ where for all $O = \{(\mathbf{p}_1, \mathbf{p}_2) : \ t(\mathbf{p}_1) < t(\mathbf{p}_2)\}$ and for all $(\mathbf{p}_1, \mathbf{p}_2) \in O$, $H((\mathbf{p}_1, \mathbf{p}_2))$ holds iff and only if $\frac{t(\mathbf{p}_2) - t(\mathbf{p}_1)}{\Delta}$ is a positive even number.

A H-ASP program is a collection of rules of the form (1) and (2). Note that H-ASP programs have the following features.

1. H-ASP advancing rules allows to pass parameters over variable length time steps. Moreover, advancing algorithms are quite general so that the output of an advancing algorithm can depend on only some of the parameters in any given generalized position.
2. Reasoning about future events can proceed with only partial information available about past events and present events.
3. Rules in H-ASP can refer to a finite number of sets of parameter values in the past. This means that to make a conclusion about a set of parameter values for a time t, a rule can use information about the sets of parameter values for times $t_1 < t_2 < \dots < t_k < t$.
4. H-ASP allows users to perform reasoning when the computations made by the algorithms are imprecise. That is, the advancing algorithm in an advancing rule is set valued. This allows us to consider advancing algorithms that use random bits or advancing algorithms that give approximate solutions.
5. H-ASP provides an indirect mechanism by which algorithms can specify values for only some of the parameters. This is accomplished when some of the parameters of generalized positions in an output set of an advancing algorithm are limited to few values, whereas other parameters are allowed to take all the possible values. This will be illustrated by the following example. Suppose that the parameter space S consists of triples of values (t, x, y), where x ranges over set X, and y ranges over set Y. Our program P consists of two rules, $\dfrac{:A, S}{set(x)}$, $\dfrac{:B, S}{set(y)}$, where $set(x)$ and $set(y)$ are atoms. Suppose further that an algorithm A can only specify the value for x which is 2 at time t'. Then the output of algorithm should be $\{(t', 2, y) : \ y \in Y\}$. If algorithm B specifies the value of y which is 1 at time t' then the output of B should be $\{(t', x, 1) : \ x \in X\}$. Intuitively we would want to select only $\mathbf{p} = (t', 2, 1)$ as a valid generalized position, since only in $\mathbf{p}$ among the generalized positions produced by A and by B both x and y are correctly specified. However besides $\mathbf{p}$ the algorithms will produce many other generalized positions. To restrict the valid generalized positions to $\mathbf{p}$, we could add the following two rules, $\dfrac{\neg set(x) : \text{TRUE}, S}{set(x)}$, $\dfrac{\neg set(y) : \text{TRUE}, S}{set(y)}$, where TRUE is a Boolean algorithm

that returns 1 on any input. These rules will restrict any stable model to contain only $\mathbf{p}$ as a generalized position with time equal to t'.

Next we define an analogue of stable models for H-ASP programs. To do this, we first must define H-ASP Horn programs and a one-step provability operator for H-ASP Horn programs.

A *H-ASP Horn program* is a H-ASP program that does not contain any negated atoms in At. A *consistent H-ASP Horn program* P is a H-ASP Horn program such that if whenever two pairs of an advancing algorithm and a constraint set, (A, O) and (A', O'), appear in P and $O, O' \subseteq S^r$, then $A \restriction_{O \cap O'} = A' \restriction_{O \cap O'}$. Intuitively, consistent H-ASP Horn programs will be used to derive single trajectory stable models while H-ASP Horn programs without the consistency constraint will be used to derive multiple trajectory stable models.

Let P be a H-ASP Horn program and $I \in S$ be an initial condition. Then the one-step provability operator $T_{P,I}$ is defined so that given $M \subseteq At \times S$, $T_{P,I}(M)$ consists of M together with the set of all $(a, J) \in At \times S$ such that

1. there exists a stationary rule $C = \frac{B_1;B_2;\ldots;B_r:H,O}{a}$ and $(\mathbf{p}_1, \ldots, \mathbf{p}_r) \in O \cap \left(\widehat{M} \cup \{I\}\right)^r$ such that $(a, J) = (a, \mathbf{p}_r)$ and $M \models (B_i, \mathbf{p}_i)$ for $i = 1, \ldots, r$ and $H(\mathbf{p}_1, \ldots, \mathbf{p}_r) = 1$.
2. there exists an advancing rule $C = \frac{B_1;B_2;\ldots;B_r:A,O}{a}$ and $(\mathbf{p}_1, \ldots, \mathbf{p}_r) \in O \cap \left(\widehat{M} \cup \{I\}\right)^r$ such that $J \in A(\mathbf{p}_1, \ldots, \mathbf{p}_r)$ and $M \models (B_i, \mathbf{p}_i)$ for $i = 1, \ldots, r$.

There is a nice subclass of H-ASP programs which are particularly well behaved that we call *Basic H-ASP* (BH-ASP) programs. In a BH-ASP program, we assume that the underlying parameter space S has the property that $T(S)$ is of the form $\{k\Delta : k = 0, \ldots, n\}$ or $\{k\Delta : k \in \mathbb{N}\}$ for some fixed $\Delta > 0$. Moreover, we do not allow the rules to refer to multiple times in the past and we assume all advancing algorithms define functions that just give us information about the next time step. That is, a BH-ASP program consists of collections of the following two types of rules.

Basic Stationary rules are of the form

$$\frac{a_1, ..., a_s, \neg b_1, ..., \neg b_t : O}{a} \tag{3}$$

where $a, a_1, ..., a_s, b_1, ..., b_t \in At$, O is a set of generalized positions in the parameter space S. The idea is that if for a generalized position $\mathbf{p} \in O$, $(a_i, \mathbf{p})$ holds for $i = 1, ..., s$ and $(b_j, \mathbf{p})$ does not hold for $j = 1, ..., t$, then $(a, \mathbf{p})$ holds. Thus stationary rules are typical general logic programming rules relative to a fixed world $W_M(\mathbf{p})$.

Basic Advancing rules are of the form

$$\frac{a_1, ..., a_s, \neg b_1, ..., \neg b_t : A, O}{a} \tag{4}$$

where $a, a_1, ..., a_s, b_1, ..., b_t \in At$, O is a set of generalized positions in the parameter space S, and A is an algorithm such that for any generalized position $\mathbf{p} \in O$, $A(\mathbf{p})$ is defined and is an element of S. Here as in H-ASP advancing rules, A can be any sort of algorithm that might require solving a differential or integral equation, solving a set of linear equations or linear programming equations, running a program or automaton, etc. However, we assume that if $\mathbf{p} = (k\Delta, x_1, \ldots, x_m) \in O$, then $A(\mathbf{p})$ is of the form $((k+1)\Delta, y_1, \ldots, y_m)$ for some $y_1, \ldots, y_m$.

In that case, we can prove the following result for consistent BH-ASP programs by induction.

▶ **Theorem 1.** *Let $I = (0, x_1(0), \ldots, x_m(0))$ be an initial condition.*

1. *Let P be a consistent BH-ASP Horn program over the parameter space S and set of atoms At. Then the least model of P is a partial single trajectory model of the form $\bigsqcup_{k \in V} W_M(k\Delta, x_1(\Delta), \ldots, x_m(k\Delta)) \times \{(k\Delta, x_1(\Delta), \ldots, x_m(k\Delta))\}$ where either $V = \mathbb{N}$ or V is some initial segment of $\mathbb{N}$.*
2. *Let P be a BH-ASP Horn program over parameter space S and set of atoms At. Then if $(a, (k\Delta, x_1, \ldots, x_m))$ is in the least model of P, then for all $1 \leq j \leq k$, there must be an element of the form $(a_j, (j\Delta, x_1^j, \ldots, x_m^j))$ in the least model of P.*

Part 2 of the theorem has a simple interpretation - if the least model contains an element for time $k\Delta$ then it also contains an element for every (discrete) time preceding $k\Delta$.

We define the stable model semantics for general H-ASP programs as follows. Suppose that we are given a H-ASP program P over a set of atoms At and a parameter space S, a set $M \subseteq At \times S$, and an initial condition $I \in S$ such that $t(I) = 0$. Then we form the Gelfond-Lifschitz reduct of P over M and I, $P^{M,I}$ as follows.

1. Eliminate from P all advancing rules $C = \frac{B_1;\ldots,B_r:A,O}{a}$ such that for all $(\mathbf{p}_1, \ldots, \mathbf{p}_r) \in O \cap \left(\widehat{M} \cup \{I\}\right)^r$, there is an i such that $M \not\models (B_i^-, \mathbf{p}_i)$ or $A(\mathbf{p}_1, \ldots, \mathbf{p}_r) \cap \widehat{M} = \emptyset$.
2. If the advancing rule $C = \frac{B_1;\ldots,B_r:A,O}{a}$ is not eliminated by (1), then replace it by $\frac{B_1^+;\ldots,B_r^+:A^+,O^+}{a}$ where for each i, B_i^+ is the result of removing all the negated atoms from B_i, O^+ is equal to the set of all $(\mathbf{p}_1, \ldots, \mathbf{p}_r)$ in $O \cap \left(\widehat{M} \cup \{I\}\right)^r$ such that $M \models (B_i^-, \mathbf{p}_i)$ for $i = 1, \ldots, r$ and $A(\mathbf{p}_1, \ldots, \mathbf{p}_r) \cap \widehat{M} \neq \emptyset$, and $A^+(\mathbf{p}_1, \ldots, \mathbf{p}_r)$ is defined to be $A(\mathbf{p}_1, \ldots, \mathbf{p}_r) \cap \widehat{M}$.
3. Eliminate from P all stationary rules $C = \frac{B_1;\ldots,B_r:H,O}{a}$ such that for all $(\mathbf{p}_1, \ldots, \mathbf{p}_r) \in O \cap \left(\widehat{M} \cup \{I\}\right)^r$, either there is an i such that $M \not\models (B_i^-, \mathbf{p}_i)$ or $H(\mathbf{p}_1, \ldots, \mathbf{p}_r) = 0$.
4. If the stationary rule $C = \frac{B_1;\ldots,B_r:H,O}{a}$ is not eliminated by (3), then replace it by $\frac{B_1^+;\ldots,B_r^+:H|_{O^+},O^+}{a}$ where for each i, B_i^+ is the result of removing all the negated atoms from B_i, O^+ is equal to the set of all $(\mathbf{p}_1, \ldots, \mathbf{p}_r)$ in $O \cap \left(\widehat{M} \cup \{I\}\right)^r$ such that $M \models (B_i^-, \mathbf{p}_i)$ for $i = 1, \ldots, r$ and $H(\mathbf{p}_1, \ldots, \mathbf{p}_r) = 1$.

We then say that M is a *general stable model of P with initial condition I* if $T_{P^{M,I}}(\emptyset) \uparrow \omega = M$.

3 The Example

We will now present the use of these definitions in the Secret Agent 00111 example. In order to keep things simple, our presentation will be restricted to the agent's movement in the Area I of the domain. We will also use the generalized positions (t, d, y, v, s) where d denotes the displacement, i.e. the distance traveled from the top of the mountain, v is the agent's velocity in the direction of displacement, y is the agent's y-coordinate, and $s = 1$ if a shot is heard at time t and $s = 0$ if a shot is not heard at time t. It should be noted that the parameter y will not be used below, however parameter y would be important for modeling the agent's movement in the lake. Since in the lake the agent can choose the steering direction, the y-coordinate will vary according to the steering.

A central consideration in determining the proper constraint sets associated with H-ASP rules is the problem of modeling the interaction between regions. In the case of our example, this means modeling accurately the agent's movements as the agent crosses the boundary between regions. In our example, the issue arises as the agent moves from region 1 of Area I into region 2 of Area I.

Recall that the agent will simply coast down the mountain unless pursued by the evil agents. If the agent is pursued, he will apply additional acceleration of $4m/s^2$ in the direction of the movement. The agent is considered to be pursued during a time interval of Δ seconds if a shot is heard in the beginning of the interval, or if two shots were heard before the start of the interval and the shots were separated by Δ seconds.

For the rest of the presentation we will set $\Delta = 1$. To keep the numbers simple, we shall round the acceleration due to gravity to 10m/s. We have carefully picked the slopes so that we have (3,4,5) right triangle in region 1 and a (7,24,25) right triangle in region 2 so that the acceleration due to gravity in region 1 in the direction of displacement will be $\frac{3}{5}10 = 6m/s^2$ and the acceleration due to gravity in the direction of displacement will be $\frac{7}{25}10 = \frac{14}{5}m/s^2$ in region 2 of Area I. We shall only model the agents behavior at times $t = 0, 1, 2, 3, 4, 5$ and we shall assume that the agent cannot exit Area I within 5 seconds so that we will not concern ourselves with the boundary between Area I and Area II.

The set of atoms is $At = \{\text{PURSUED},\ A,\ T\}$ where PURSUED will indicate that the agent is being pursued, A will indicate that two shots separated by 1 second were heard in the past, and T is a place holder atom.

Given a generalized position $\mathbf{p} = (t, d, y, v, s)$ and an acceleration a let $B(\mathbf{p},\ a) = b$ be such that if the agent is at distance d as measured from the top of the mountain in region 1 on the path along the mountain's slope and has initial velocity v and he accelerates continuously with the acceleration a in the direction of his movement, then at time $t + b$ he will reach the boundary of region 1 of Area I which is a generalized position with $d = 50m$. That is, we want $50 = \frac{a}{2}b^2 + vb + x$ so that $B((t, x, y, v, s),\ a) = \frac{-v+\sqrt{v^2-2a(x-50)}}{a}$. Thus $O_1 = \{(t, x, y, v, s) : 0 \leq d < 50 \text{ and } B((t, x, y, v, s), 6) \geq 1\}$ is the set of generalized positions in region 1 where the agent coasting down hill will not reach the boundary between region 1 and region 2 in < 1 second and $O_2 = \{(t, x, y, v, s) : 0 \leq d < 50$ and $B((t, x, y, v, s), 6) < 1\}$ is the set of generalized positions in region 1 where the agent coasting down hill will reach the boundary between region 1 and region 2 in < 1 second. Similarly, $O_3 = \{(t, x, y, v, s) : 0 \leq d < 50 \text{ and } B((t, x, y, v, s), 10) \geq 1\}$ is the set of generalized positions in region 1 where the agent accelerating at $4m/s^2$ in addition to gravity will not reach the boundary between region 1 and region 2 in < 1 second and $O_4 = \{(t, x, y, v, s) : 0 \leq d < 50 \text{ and } B((t, x, y, v, s), 10) < 1\}$ is the set of generalized positions in region 1 where the agent accelerating at $4m/s^2$ in addition to gravity will reach the boundary between region 1 and region 2 in < 1 second. Finally let O_5 be the set of all the generalized positions in region 2.

We will have two types of advancing algorithms. That is, if we are using a constant acceleration a in a region, then by the usual formulas for displacement $d(t) = \frac{a}{2}t^2 + v(0)t + d(0)$ and velocity $v(t) = at + v(0)$ at time t, one can see that our advancing algorithm should be

$$A_a(t, d, y, v, s) = \{(t + 1, \frac{a}{2} + v + d, y, a + v, s) : s \in \{0, 1\}\}.$$

If at $\mathbf{p} = (t, d, y, v, s)$, we know that we will cross from region 1 to region 2 so that we reach the boundary $d = 50$ of region 1 with a velocity of $aB(\mathbf{p}, a) + v$ and then in region 2, our acceleration is a', one can see that our advancing algorithm should be

$$C_{a,a'}(t, d, y, v, s) = \{(t + 1, d', y, v', s) : s \in \{0, 1\}\}.$$

where $d' = \frac{a'}{2}(1 - B(\mathbf{p}, a))^2 + (aB(\mathbf{p}, a) + v)(1 - B(\mathbf{p}, a)) + 50$ and $v' = a'(1 - B(\mathbf{p}, a)) + aB(\mathbf{p}, a) + v$.

We then have the following rules for when our agent is not being pursued by evil agents.

$$\frac{\neg\,\text{PURSUED} : A_6, O_1}{T} \quad \frac{\neg\,\text{PURSUED} : C_{6,\frac{14}{5}}, O_2}{T} \quad \frac{\neg\,\text{PURSUED} : A_{\frac{14}{5}}, O_5}{T}$$

We have the following rules for when our agent is begin pursued by evil agents.

$$\frac{\text{PURSUED} : A_{10}, O_3}{T} \quad \frac{\text{PURSUED} : C_{10,\frac{34}{5}}, O_4}{T} \quad \frac{\text{PURSUED} : A_{\frac{34}{5}}, O_5}{T}.$$

The atom A is to be derived at time t if a shot was heard at time $t - \Delta t$ and at time t. Also atom A is to be derived at time t if A was already derived at time $t - \Delta t$. This is formalized by the following two stationary rules:

$$\frac{: H, G}{A} \quad \frac{A;\ :\text{TRUE}_2,\ G}{A}$$

where $G = \{(\mathbf{p}_1, \mathbf{p}_2) \in S^2 : 1 + t(\mathbf{p}_1) = t(\mathbf{p}_2)\}$, $\text{TRUE}_2\,(\mathbf{p}_1, \mathbf{p}_2) = 1$, and and $H\,(\mathbf{p}_1, \mathbf{p}_2) = \chi(s\,(\mathbf{p}_2) - s\,(\mathbf{p}_1) = 1)$. Here for any statement Q, we let $\chi(Q) = 1$ if Q is true and $\chi(Q) = 0$ if Q is false.

Finally the atom PURSUED is to be derived at time t if a shot is heard at time t or if atom A is derived at time t. Hence, we have the following two stationary rules

$$\frac{: U, S}{\text{PURSUED}} \quad \frac{A : \text{TRUE}_1, S}{\text{PURSUED}}$$

where $U\,(\mathbf{p}) = \chi(s\,(\mathbf{p}) = 1)$ and for all $\mathbf{p} \in S$, $\text{TRUE}_1\,(\mathbf{p}) = 1$.

Let P be the program consisting of the rules described above. We will now consider a particular single trajectory stable model M of P. According to this stable model there is a shot heard at time $t = 0$ then at time $t = 2$ and then at time $t = 3$. We will only consider the elements of the stable model up to and including the time $t = 5$.

Suppose that at time $t = 0$ the agent has displacement $5m$ and initial velocity $2m/s$. That is $I = (0, 5, 0, 2, 1)$. Then the following table describes a stable model M of P.

t	$d\ (m)$	$y\ (m)$	$v\ \ (m/s)$	s	$W_M\ (t, d, y, v, s)$
0	5	0	2	1	T, PURSUED
1	$5 + 2 + 5 = 12$	0	$10 + 2 = 12$	0	T
2	$3 + 12 + 12 = 27$	0	$6 + 12 = 18$	1	T, PURSUED
3	$5 + 18 + 27 = 50$	0	$10 + 18 = 28$	1	T, A, PURSUED
4	$\frac{17}{5} + 28 + 50 = \frac{407}{5}$	0	$\frac{34}{5} + 28 = \frac{174}{5}$	0	T, A, PURSUED
5	$\frac{17}{5} + \frac{174}{5} + \frac{407}{6} = \frac{598}{5}$	0	$\frac{34}{5} + \frac{174}{5} = \frac{208}{5}$	0	T, A, PURSUED

M induces the following Gelfond-Lifschitz reduct

$$\frac{:A_6^+,\ \{(1,12,0,12,0)\}}{T} \qquad \frac{\text{PURSUED}:A_{10}^+,\{(0,5,0,2,1),(2,27,0,18,1)\}}{T}$$

$$\frac{\text{PURSUED}:A_{\frac{34}{5}}^+,\left\{(3,50,0,28,1),\left(4,\frac{407}{5},0,\frac{174}{5},0\right),\left(5,\frac{598}{5},0,\frac{208}{5},0\right)\right\}}{T}$$

$$\frac{:H|_{G^+}, G^+}{A} \qquad \frac{A;\ :\text{TRUE}_2\,|_Y, Y}{A} \qquad \frac{:U|_{S^+}, S^+}{\text{PURSUED}} \qquad \frac{A:\text{TRUE}_1\,|_{\widehat{M}},\ \widehat{M}}{\text{PURSUED}}$$

where $G^+ = \{((2, 27, 0, 18, 1), (3, 50, 0, 28, 1))\}$, $S^+ = \{(0, 5, 0, 2, 1), (2, 27, 0, 18, 1), (3, 50, 0, 28, 1)\}$, $Y = \{((0, 5, 0, 2, 1), (1, 12, 0, 12, 0)), ((1, 12, 0, 12, 0), (2, 27, 0, 18, 1)), ((2, 27, 0, 18, 1), (3, 50, 0, 28, 1)),$

$((3, 50, 0, 28, 1), (4, \frac{407}{5}, 0, \frac{174}{5}, 0)), ((4, \frac{407}{5}, 0, \frac{174}{5}, 0), (5, \frac{598}{5}, 0, \frac{208}{5}, 0))\}$, and
$\widehat{M} = \{(0, 5, 0, 2, 1), (1, 12, 0, 12, 0), (2, 27, 0, 18, 1), (3, 50, 0, 28, 1), (4, \frac{407}{5}, 0, \frac{174}{5}, 0), (5, \frac{598}{5}, 0, \frac{208}{5}, 0)\}$.

The fact that M is stable model can be easily verified by computing the least model of the Gelfond-Lifschitz reduct of P with respect to M and I.

Remark: We should note that the restriction on O in both extended stationary rules and extended advancing rules that $t(\mathbf{p}_1) < \cdots < t(\mathbf{p}_r)$ can in fact be dropped by slightly modifying our definitions of the one step provability operator and stable models. This could in principle allow one to use rules that refer to both past and future times or to multiple generalized positions that occur at the same time. We put in the restriction $t(\mathbf{p}_1) < \cdots < t(\mathbf{p}_r)$ mainly for ease of presentation and the fact that we did not have space to develop applications of these more extended rules.

4 Conclusion

In this paper, we introduced Hybrid ASP (H-ASP) - an ASP type system to reason about dynamical systems which exhibit both continuous and discrete aspects. The key feature of the system is that it includes advancing rules that incorporate an algorithm to allow for parameter passing from one time step to the next and stationary rules which incorporate an auxiliary algorithm to allow non-logical checking that govern the applicability of the rule. We then defined an analogue of the stable model semantics for H-ASP by defining an appropriate analogues of the one-step provability operator for Horn programs and the Gelfond-Lifschitz reduct of normal logic programs.

We envision that an actual implementation of a H-ASP solver will require that the system makes calls to other modules. That is, the algorithms that are part of advancing rules or extended stationary rules are allowed to be any sort of algorithms which require solving a differential or integral equation, solving a set of linear equations or linear programming equations, running a program or automaton, etc. Thus a H-ASP-solver should naturally allow calls to specialized software outside the system to run such algorithms. We have implemented preliminary version of our system where the algorithm required solving PDEs associated with the Heat equation (for a discussion of the Heat equation see [4] and [9]). This type of application goes well beyond the simple toy model that we used to illustrate our ideas in this paper and will be the subject of future papers.

We view the extension of the answer set programming paradigm, H-ASP, that we introduced in this paper as a first step for further work that will lead to both theoretical tools used for the modeling and analysis of dynamic systems and for computer applications that simulate dynamical systems. There is considerable work to be done in developing a theory of such programs which is similar to the theory that has been developed for ASP programs. For example, a careful analysis of the complexity of the stable models of H-ASP programs as a function of the complexity of the advancing and Boolean algorithms in the program needs to be done. One can ask under what circumstances are there analogues of the forward chaining algorithm of [7] for H-ASP programs. One can consider more extended sets of rules that allow for partial parameter passing or allow different rules to instantiate disjoint sets of parameters for the next time step. These issues will be pursued in later papers. Nevertheless, we believe that the point of view of thinking of rules as general input-output devices has the potential for many new applications of ASP techniques.

References

1. Sandeep Chintabathina. 2010. Towards Answer Set Programming Based Architectures for Intelligent Agents. *PhD thesis, Texas Tech University.*
2. M. Gelfond and V. Lifschitz. 1998. Action languages. *Electronic Transactions on AI*, 3(16).
3. David L. George, Randall J. LeVeque. 2006. Finite Volume Methods and Adaptive Refinement for Global Tsunami Propagation and Local Inundation. *Science of Tsunami Hazards,* Vol. 24, No. 5, pp. 319-328.
4. Gustafsson, Kreiss, Oliger. 1995. Time Dependent Problems and Difference Methods. *John Wiley & Sons, Inc.*
5. S. M. Jachec, O. B. Fringer, M. G. Gerritsen, R. L. Street. 2006. Numerical Simulation of Internal Tides and the Resulting Energetics within Monterey Bay and the Surrounding Area. *Geophysical Research Letters,* Vol. 33, L12605, doi: 10.1029/2006GL026314.
6. Kerckhoffs, Neal, Gu, Bassingthwaighte, Omens, McCulloch. January 2007. Coupling of a 3D Finite Element Model of Cardiac Ventricular Mechanics to Lumped Systems Models of the Systemic and Pulmonic Circulation. *Annals of Biomedical Engineering*, Vol. 35, No. 1, pp. 1-18.
7. W. Marek, A. Nerode, and J.B. Remmel. 1999 Logic programs, well orderings, and forward chaining. Annals of Pure and Applied Logic Vol. 96 , 231-276.
8. Reiter. Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems.2001. *The MIT Press.*
9. John C. Strikwerda. 2004. Finite Difference Schemes and Partial Differential Equations. Second Edition. *SIAM.*
10. Richard S. Sutton, Andrew G. Barto. Reinforcement Learning. 1998. *The MIT Press.*

Representing the Language of the Causal Calculator in Answer Set Programming

Michael Casolary and Joohyung Lee

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

Abstract

Action language $\mathcal{C}+$, a formalism based on nonmonotonic causal logic, was designed for describing properties of actions. The definite fragment of $\mathcal{C}+$ was implemented in system the Causal Calculator (CCalc), based on a reduction of nonmonotonic causal logic to propositional logic. On the other hand, in this paper, we represent the language of CCalc in answer set programming (ASP), by translating nonmonotonic causal logic into formulas under the stable model semantics. We design a standard library which describes the constructs of the input language of CCalc in terms of ASP, allowing a simple modular method to represent CCalc input programs in the language of ASP. Using the combination of system f2lp and answer set solvers, our prototype implementation of this approach, which we call Cplus2ASP, achieves functionality close to CCalc while taking advantage of answer set solvers to yield efficient computation that is orders of magnitude faster than CCalc on several benchmark examples.

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

Keywords and phrases answer set programming, nonmonotonic causal logic, action languages

Digital Object Identifier 10.4230/LIPIcs.ICLP.2011.51

1 Introduction

Action languages are formal models of parts of natural language that are used for describing properties of actions. Among them, language $\mathcal{C}+$ [9] and its predecessor $\mathcal{C}$ [10] are based on nonmonotonic causal logic. The definite fragment of nonmonotonic causal logic can be turned into propositional logic by the literal completion method, which resulted in an efficient way to compute $\mathcal{C}+$ using satisfiability (SAT) solvers. The Causal Calculator (CCalc) is an implementation of this idea. Version 1 of CCalc was created by McCain [16], accepting $\mathcal{C}$ as its input language; Version 2 is an enhancement described in [11], which accepts $\mathcal{C}+$ as its input language. Language $\mathcal{C}+$ is significantly more enhanced than $\mathcal{C}$ in several ways, such as being able to represent multi-valued formulas, defined fluents, additive fluents, rigid constants and defeasible causal laws. Although CCalc was not aimed at large scale applications, it has been applied to several challenging commonsense reasoning problems, including problems of nontrivial size [1], to provide a group of robots with high-level reasoning [4], to give executable specifications of norm-governed computational societies [3], and to automate the analysis of business processes under authorization constraints [2].

An alternative way to compute $\mathcal{C}+$ is to turn it into answer set programs and to use existing answer set solvers. This can be achieved by first turning multi-valued causal logic into Boolean-valued causal logic as described in [11] and then turning the latter into answer set programs as described in [16, 5, 15]. In fact, a system called Coala (*Co*mpiler for *a*ction *la*nguages) was implemented based on this idea [8]. The system turns a fragment of language

© Michael Casolary and Joohyung Lee;
licensed under Creative Commons License NC-ND
Technical Communications of the 27th International Conference on Logic Programming (ICLP'11).
Editors: John P. Gallagher, Michael Gelfond; pp. 51–61

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

$\mathcal{C}+$ into the input language of GRINGO[1], but that fragment lacks several important features of $\mathcal{C}+$ mentioned above, which are available in CCALC.

In this paper, we provide a way to encode CCALC input language in answer set programming, and present a prototype implementation called CPLUS2ASP based on this idea. Our approach differs from that of COALA in a few ways. First, CPLUS2ASP can handle multi-valued constants, which COALA does not allow. Second, we turn the language of CCALC into formulas under the stable model semantics [7], and use system F2LP[2] ("*f*ormulas *to* *l*ogic *p*rograms") [14] to turn them into the input language of ASP solvers. This allows users to write the same complex formulas as in the input language of CCALC. Third, we design a standard library that defines the constructs of the CCALC input language in terms of the language of GRINGO. Using the standard library and reusing the existing software F2LP and CLINGO[3] allowed a simple design of CPLUS2ASP that achieves functionality close to CCALC. Also thanks to the efficiency of answer set solvers, our experiments show that CPLUS2ASP is orders of magnitude faster than CCALC in several benchmark examples.

The paper is organized as follows. Section 2 provides preliminaries, and Section 3 shows how to encode the language of CCALC in ASP, and presents the prototype implementation CPLUS2ASP. We compare the efficiency of CPLUS2ASP against that of CCALC in Section 4.

2 Preliminaries

2.1 Nonmonotonic Causal Theories and $\mathcal{C}+$

Due to lack of space, the reviews in this section are rather dense. We refer the reader to [9] for the details. In $\mathcal{C}+$, formulas are multi-valued. A *(multi-valued propositional) signature* is a set σ of symbols called *constants*, along with a nonempty finite set $Dom(c)$ of symbols called the *domain* of c. An *atom* of a signature σ is an expression of the form $c = v$ ("the value of c is v") where $c \in \sigma$ and $v \in Dom(c)$. A *(multi-valued) formula* of σ is a propositional combination of atoms. A *causal rule* is an expression of the form

$$F \Leftarrow G$$

where F and G are multi-valued propositional formulas. A *causal theory* is a set of causal rules.

Language $\mathcal{C}+$ is a high level notation for causal theories that was designed for describing transition systems—directed graphs whose vertices represent states and edges are labeled by actions that affect the states. In $\mathcal{C}+$, constants are partitioned into *fluent* constants and *action* constants. Fluent constants are further partitioned into *simple* and *statically determined* fluents. A *fluent formula* is a formula where all constants occurring in it are fluent constants. An *action formula* is a formula that contains at least one action constant and no fluent constants. A *static law* is an expression of the form

$$\textbf{caused } F \textbf{ if } G \tag{1}$$

where F and G are fluent formulas. An *action dynamic law* is an expression of the form (1) in which F is an action formula and G is a formula. A *fluent dynamic law* is an expression

[1] `http://potassco.sourceforge.net`
[2] `http://reasoning.eas.asu.edu/f2lp`
[3] CLINGO is a system that combines GRINGO and CLASP in a monolithic way, available from the same link as the one in Footnote 1.

of the form

$$\textbf{caused } F \textbf{ if } G \textbf{ after } H \tag{2}$$

where F and G are fluent formulas and H is a formula, provided that F does not contain statically determined constants. A *causal law* is a static law, or an action dynamic law, or a fluent dynamic law. An *action description* is a set of causal laws.

The semantics of $\mathcal{C}+$ in [9] is described via a translation into causal logic. For any action description D and any nonnegative integer m, the causal theory D_m is defined as follows. The signature of D_m consists of the pairs $i\!:\!c$ such that

- $i \in \{0,\dots,m\}$ and c is a fluent constant of D, or
- $i \in \{0,\dots,m-1\}$ and c is an action constant of D.

The domain of $i\!:\!c$ is the same as the domain of c. By $i\!:\!F$ we denote the result of inserting $i\!:$ in front of every occurrence of every constant in a formula F, and similarly for a set of formulas. The rules of D_m are

$$i\!:\!F \Leftarrow i\!:\!G \tag{3}$$

for every static law (1) in D and every $i \in \{0,\dots,m\}$, and for every action dynamic law (1) in D and every $i \in \{0,\dots,m-1\}$;

$$i\!+\!1\!:\!F \Leftarrow (i\!+\!1\!:\!G) \wedge (i\!:\!H) \tag{4}$$

for every fluent dynamic law (2) in D and every $i \in \{0,\dots,m-1\}$;

$$0\!:\!c\!=\!v \Leftarrow 0\!:\!c\!=\!v \tag{5}$$

for every simple fluent constant c and every $v \in \mathit{Dom}(c)$.

The causal models of D_m correspond to the paths of length m in the transition system described by D.

2.2 Language of the Causal Calculator

The language of CCalc provides a convenient way of expressing $\mathcal{C}+$ descriptions. It allows us to declare `sorts`, `objects`, `variables` and `constants`, as well as to describe causal laws. A causal law may contain variables, which are understood in terms of grounding. Such causal laws are *schemas* for ground instances, as in answer set programming.

The left column of Figure 1 is a simple $\mathcal{C}+$ description in the language of CCalc. The symbol `>>` in the sort declaration between the names of two sorts expresses that the second is a subsort of the first, so that every object that belongs to the second sort belongs also to the first. Lines 1–2 declare that `s_num` is a subsort of `num`. Lines 11–13 introduce objects of the two sorts. The integers from 0 to `n` $-$ 1 belong to sort `s_num`; the integers from 0 to `n` belong to sort `num`. Line 19 declares that `has` is an inertial fluent whose domain is `num`. CCalc understands this line the same as the declaration

```
has    :: simpleFluent(num)
```

followed by the fluent dynamic law

```
caused has=X if has=X after has=X
```

where `X` ranges over all objects of sort `num`. Similarly, Line 22 declares that `buy` is an exogenous action with Boolean values. Lines 32–35 represent a simple query for finding a path of length 3 from the initial state where `has=2` to the goal state where `has=4`.

2.3 Stable Model Semantics of First-Order Formulas and System F2LP

We refer the reader to [7, 14] for the details. The stable model semantics from [7] is defined for first-order formulas, which allow for arbitrary nesting of connectives and quantifiers as in first-order logic. Strong negation ($\sim$) occurs only in front of an atom. For instance,

$$\neg \sim p(x) \rightarrow p(x) \qquad (6)$$

expresses that x belongs to p by default.

System F2LP can be used to turn any "almost universal sentence" into an answer set program so that answer set solvers can be used to compute the Herbrand stable models of the almost universal sentence. As far as this paper is concerned, it is sufficient to know that any sentence where every quantifier is in the scope of negation is almost universal. (6) can be encoded in the language of F2LP as

```
p(X) <- not -p(X).
```

(In the language of F2LP, the default negation ($\neg$) is expressed as `not`; the strong negation ($\sim$) is encoded as `-`, following the convention in the input language of ASP solvers. In addition, `?` and `!` denote the existential and universal quantifiers, respectively; `|` denotes disjunction and `&` denotes conjunction.) The input language of F2LP also allows aggregates and choice rules as in the language of GRINGO.

```
1  :- sorts
2    num >> s_num.
3
4
5
6
7
8
9
10
11 :- objects
12   0..n-1   :: s_num;
13   n        :: num.
14
15 :- variables
16   K        :: s_num.
17
18 :- constants
19   has :: inertialFluent(num);
20
21
22   buy :: exogenousAction(boolean).
23
24
25 buy causes has=K+1 if has=K.
26
27
28
29 nonexecutable buy if has=n.
30
31
32 :- query
33 maxstep :: 3;
34 0: has=2;
35 maxstep: has=4.
```

```
1
2  sort(num).
3  #domain num(V_num).
4  sort_object(num,V_num).
5
6  sort(s_num).
7  #domain s_num(V_s_num).
8  sort_object(num,V_s_num).
9
10 num(V_s_num).
11
12 s_num(0..n-1).
13 num(n).
14
15
16 #domain s_num(K).
17
18
19 inertialFluent(has).
20 constant_sort(has,num).
21
22 exogenousAction(buy).
23 constant_sort(buy,boolean).
24
25 h(eql(has,K+1),V_astep+1) <-
26    h(eql(buy,true),V_astep) &
27    h(eql(has,K),V_astep).
28
29 false <-
30    h(eql(buy,true),V_astep) &
31    h(eql(has,n),V_astep).
32
33 false <- query_label(0) &
34          not (h(eql(has,2),0) &
35               h(eql(has,4),maxstep)).
```

Figure 1 Simple Transition System in the Language of CCALC and in the Language of F2LP

3 Representing the Language of the Causal Calculator in ASP

3.1 Translating $\mathcal{C}+$ into Answer Set Programs

We consider a finite definite $\mathcal{C}+$ description D of signature σ, where the heads of the rules are either an atom or $\bot$. Without losing generality, we assume that, for any constant c in σ, $Dom(c)$ has at least two elements. Description D can be turned into a logic program following these steps: (i) turn D into the corresponding multi-valued causal theory D_m (as explained in Section 2.1); (ii) turn D_m into a Boolean-valued causal theory D'_m; (iii) turn D'_m into formulas under the stable model semantics; (iv) turn the result further into a logic program (using F2LP as explained in Section 2.3). In this section we explain Steps (ii) and (iii).

Definite Elimination of Multi-Valued Constants Consider the causal theory D_m with signature σ_m consisting of rules of the form (3), (4) and (5). Consider all constants $i:c$ $(0 \le i \le m)$ in σ_m, where c is a fluent constant of D. By σ^c_m we denote the signature obtained from σ_m by replacing every constant $i:c$ with Boolean constants $i:eql(c,v)$ for all $v \in Dom(c)$.

The causal theory D^c_m with signature σ^c_m is obtained from D_m by replacing each occurrence of an atom $i:c=v$ in D_m with $i:eql(c,v)=\mathbf{t}$, and adding the causal rules

$$i:eql(c,v')=\mathbf{f} \Leftarrow i:eql(c,v)=\mathbf{t} \qquad (0 \le i \le m) \tag{7}$$

for all $v, v' \in Dom(c)$ such that $v \ne v'$.

The following proposition is a simplification of Proposition 9 from [11].[4]

▶ Proposition 1. There is a 1-1 correspondence between the models of D_m and the models of D^c_m.

The elimination of multi-valued action constants is similar.

Turning Boolean-valued Action Descriptions into SM In [6], McCain's translation is modified and extended as follows. Take any set T of causal rules of the forms

$$A \Leftarrow G, \tag{8}$$
$$\neg A \Leftarrow G, \tag{9}$$
$$\bot \Leftarrow G, \tag{10}$$

where A is an atom and G is an arbitrary propositional formula. For each rule (8), take the formula $\neg\neg G \rightarrow A$; for each rule (9), the formula $\neg\neg G \rightarrow {\sim}A$; for each rule (10), the formula $\neg G$. Also add the following completeness constraints for all atoms A:

$$\neg A \wedge \neg{\sim}A \rightarrow \bot\ . \tag{11}$$

Note that, for T, which is definite, the modified McCain translation yields a first-order theory that is tight [7].

Consider D'_m which is obtained from D_m by eliminating all multi-valued constants in favor of Boolean constants. $h(i:F)$ is a formula obtained from $i:F$ by replacing every

[4] Proposition 9 from [11] involves adding two kinds of rules. Vladimir Lifschitz pointed out that one kind of rules can be dropped if the given theory is definite.

occurrence of $i : eql(c, v) = \mathbf{t}$ in it with $h(eql(c, v), i)$ and every occurrence of $i : eql(c, v) = \mathbf{f}$ with $\sim h(eql(c, v), i)$. According to the modified McCain translation, the causal rules (3) that represent a static law (1) are represented by formulas under the stable model semantics as

$$h(i : F) \leftarrow \neg\neg h(i : G) \tag{12}$$

($i \in \{0, \dots, m\}$). The translation of causal rules for an action dynamic law is similar except that i ranges over $\{0, \dots, m-1\}$. In the special case when $h(i : F)$ and $h(i : G)$ are the same literal, (12) can be represented using choice rules in ASP:

$$\{h(i{:}F)\}. \tag{13}$$

This is because (12) is strongly equivalent to $h(i{:}F) \vee \neg h(i{:}F)$, which can be abbreviated as (13) [12]. In fact, we observe in many cases (13) can be used in place of (12).

Similarly, the modified McCain translation turns the causal rules (4) that correspond to a fluent dynamic law (2) into

$$h(i{+}1{:}F) \ \leftarrow \ \neg\neg\Big(h(i{+}1{:}G) \ \wedge \ h(i{:}H)\Big).$$

In fact, we can also turn (4) into

$$h(i{+}1{:}F) \ \leftarrow \ \neg\neg h(i{+}1{:}G) \ \wedge \ h(i{:}H) \tag{14}$$

because the change does not affect the stable models of the resulting theory, which is tight [7]. Similarly, certain occurrences of $\neg\neg$ in (12) and (14) can be further dropped if removing them does not cause the resulting theory to become non-tight.

Again in the special case when $h(i{+}1{:}F)$ and $h(i{+}1{:}G)$ are the same literal, (14) can be represented using choice rules as follows:

$$\{h(i{+}1{:}F)\} \ \leftarrow \ h(i{:}H).$$

3.2 Representing Domain Descriptions in the Language of F2LP

Figure 1 shows a side-by-side comparison of an example CCALC input program and its representation in the language of F2LP. As shown, the translation is modular. For each sort name S that is declared in the CCALC input program, the translation introduces a fact `sort`(S) and a variable V_S that ranges over all objects of sort S by the line #`domain` $S(V_S)$, and relates the sort name and the objects by the fact `sort_object`(S, V_S). (This "meta predicate," together with another meta predicate `constant_sort` that is shown later, is used in the standard library to associate constants with their domains.) As an example, Lines 2–8 in the right column of Figure 1 is a representation of sort declarations in the language of GRINGO. In addition, the declaration that S_1 is a supersort of S_2 is represented by $S_1(V_{S_2})$, as illustrated in Line 10.

The ASP representation of the object and variable declarations are straightforward. The declaration that O is an object of sort S is encoded as a fact $S(O)$. In order to declare a user-defined variable V of sort S, we write #`domain` $S(V)$. See Lines 12–13, and Line 16 for example.

A constant declaration in the language of CCALC of the form

$$C :: \mathit{CompositeSort}(V)$$

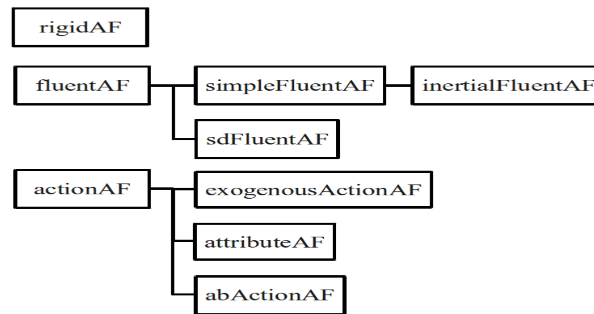

Figure 2 The Hierarchy of Atomic Formulas

is turned into a fact $CompositeSort(C)$, followed by another fact `constant_sort`(C, V), which is used in the standard library. See Lines 19–23 for example.

Encoding causal laws in the language of F2LP follows the method in Section 3.1. Like the input language of CCALC, the variables in the F2LP rules are understood as schemas for ground terms. Lines 25–31 are example encodings of causal laws in the language of F2LP. Since every variable is sorted, these F2LP rules are safe according to the definition of safety in [13], and its translation into an ASP program also results in a safe logic program.

Note that Figure 1 does not contain the other causal laws (the inertial assumption for `has` and the exogeneity assumption for `buy`). Since such causal laws and rules are frequently used, they are described in the standard library, which we explain in the next section.

3.3 Standard Library

The standard library[5] declares built-in sorts and objects, such as sort `boolean` and its objects `true` and `false`; sorts `step` and `astep` and the integer objects that belong to the sorts ($0, \dots,$ `maxstep` for `step` and $0, \dots,$ `maxstep` $- 1$ for `astep`). More importantly, it contains postulates specific to each kind of fluents and actions.

3.3.1 Postulates for Specific Fluents and Actions

First, we assume the presence of certain meta-variables that are used in the postulates. For instance, `V_inertialFluentAF` is a meta-variable that ranges over all ground terms of the form `eql`(c, v) where c is an `inertialFluent` and v is an object in the domain of c as introduced in the domain description. For the domain description in Figure 1, `V_inertialFluentAF` ranges over `eql(has, 0)`, `eql(has, 1)`, $\dots$, `eql(has, n)`. Similarly, we have other meta-variables `V_fluentAF`, `V_simpleFluentAF`, `V_sdFluentAF`, `V_rigidAF`, `V_actionAF`, `V_exogenousActionAF`, `V_attributeAF` that range over ground terms of the form `eql`(c, v) where c and v range over corresponding constants and values. We show later how to prepare a program so that meta-variables range over the atoms as intended.

- The inertial assumption for `inertialFluents` is represented by

```
{h(V_inertialFluentAF,V_astep+1)} <- h(V_inertialFluentAF,V_astep).
```

[5] See `http://reasoning.eas.asu.edu/cplus2asp` for the complete file.

- The exogeneity assumption (5) for simple fluents at time 0 is represented by

```
{h(V_simpleFluentAF,0)}.
```

- The exogeneity assumption for `exogenousAction` is stated as

```
{h(V_exogenousActionAF,V_astep)}.
```

- The completeness assumption (11) for fluents is represented as follows.

```
false <- not h(V_fluentAF,V_step) & not -h(V_fluentAF,V_step).
```

  or equivalently as

```
false <- {h(V_fluentAF,V_step), -h(V_fluentAF,V_step)}0.
```

- The definite elimination rules for multi-valued fluent constants corresponding to (7) can be represented as

```
-h(eql(V_fluent,Object1),V_step) <-
   h(eql(V_fluent,Object),V_step) & constant_object(V_fluent,Object)
   & constant_object(V_fluent,Object1) & Object != Object1.
```

  Here `V_fluent` is a meta-variable that ranges over all fluent constants. The predicate `constant_object` is defined in terms of `sort_object` and `constant_sort`:

```
constant_object(V_constant,Object) <-
   constant_sort(V_constant,V_sort) & sort_object(V_sort,Object).
```

  (Recall that `sort_object` is introduced in translating `sort` declarations from the domain description and `constant_sort` is introduced in translating `constant` declarations from the domain description.)
  The definite elimination rules and the completeness assumptions for action constants are similar to those for fluent constants.

3.4 Meta-Sorts and Meta-Variables

In order to make grounding replace all meta-variables with the corresponding ground terms as intended in the previous section, we first introduce meta-level sorts for representing the constant hierarchy in $\mathcal{C}+$. This is done in the same way as the object-level (user-defined) sorts are introduced. For instance, the following are sort declarations for `simpleFluent` and `inertialFluent`, and the declaration of their subsort relation.

```
sort(simpleFluent).      #domain simpleFluent(V_simpleFluent).
sort_object(simpleFluent,V_simpleFluent).

sort(inertialFluent).    #domain inertialFluent(V_inertialFluent).
sort_object(inertialFluent,V_inertialFluent).

simpleFluent(V_inertialFluent).
```

Recall that in Figure 1, the constant declarations included the fact `inertialFluent(has)`; the variable `V_simpleFluent` ranges over all simple fluent constants, including the inertial fluent `has`.

Similarly, we introduce meta-level sorts for different kinds of atomic formulas depending on the different kinds of constants. For instance, the following is a part of the declaration for `simpleFluentAF` and `inertialFluentAF`.

```
sort(simpleFluentAF).     #domain simpleFluentAF(V_simpleFluentAF).
sort_object(simpleFluentAF,V_simpleFluentAF).

sort(inertialFluentAF).   #domain inertialFluentAF(V_inertialFluentAF).
sort_object(inertialFluentAF,V_inertialFluentAF).

simpleFluentAF(V_inertialFluentAF).
```

These declarations are used to define *domain predicates* $Constant$`AF`s which contain terms of the form `eql`(c, v) where c is a constant of meta-level sort $Constant$ and v is a value in the domain of c. For instance, the following represents that `inertialFluentAF` is a domain predicate that contains all ground terms of the form `eql`(c, v) where c is an `inertialFluent`, and v is a value in the domain of c, using the meta-predicate `constant_object`.

```
inertialFluentAF(eql(V_inertialFluent,Object)) <-
   constant_object(V_inertialFluent,Object).
```

(Recall the definition of `constant_object` in the previous section.) The grounding process replaces the meta-variable `V_inertialFluentAF` by every ground term of the form `eql`(c, v) where c is a constant of meta-level sort `inertialFluent` and v is an element in the domain of c, as specified by the `constant_object` relation. So once the user declares that c is an `inertialFluent` (or one of its subsorts) in the domain description, the postulates for the inertial assumption for c are generated by ASP grounders.

4 Implementation and Experiments

We implemented the techniques described in Section 3 in a prototype implementation, which we call CPLUS2ASP. The system achieves functionality close to CCALC by using the standard library and the combination of the existing software F2LP and CLINGO. As documented in Figure 3, the system turns the CCALC domain description into the language of F2LP, calls F2LP to turn it into the language of CLINGO, calls CLINGO to find answer sets, and displays them in CCALC-style solutions (`as2transition` is a post-processor that takes answer sets and transforms them into a format of CCALC output.) CPLUS2ASP is designed to be compatible with the input language of CCALC. It supports most of the basic features of CCALC, but does not yet handle features like user-defined macros, `where` clauses, and shifting. The system was written in C++, utilizing the tools `flex` and `bison` to aid in the creation of a CCALC language grammar and syntax parser.

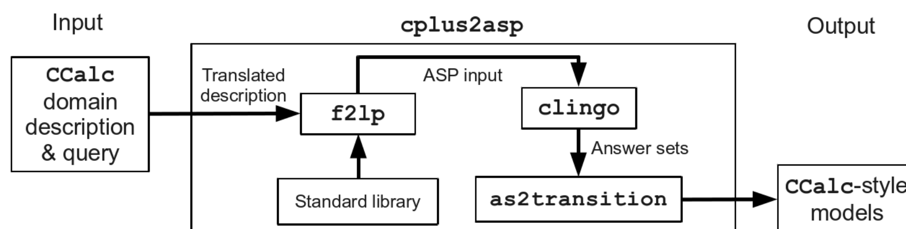

Figure 3 The Architecture of CPLUS2ASP

Problem	CCalc with zchaff			Cplus2ASP with Clingo		
	Total	Preparation[a]	Solving	Total	Preparation[b]	Solving[c]
Traffic World: Scenario 1 (`maxstep`=5)	1.55s	1.52s (1.06s + 0.29s + 0.17s)	0.03s	0.22s	0.20s (0.11s + 0.09s)	0.02s (0.02s + 0.00s)
Traffic World: Scenario 2 (`maxstep`=3)	22.74s	22.38s (17.85s + 3.45s + 1.08s)	0.36s	1.42s	1.14s (0.11s + 1.03s)	0.28s (0.28s + 0.00s)
Traffic World: Scenario 3 (`maxstep`=5)	6.29s	6.05s (4.48s + 0.99s + 0.58s)	0.24s	0.52s	0.40s (0.08s + 0.32s)	0.12s (0.12s + 0.00s)
Traffic World: Scenario 3-1 (`maxstep`=11)	608.76s	558.46s (415.06s + 85.45s + 57.95s)	50.30s	59.84s	34.42s (0.08s + 33.34s)	25.42s (17.95s + 7.47s)

[a] (grounding time + completion time + shifting & writing input clause time)
[b] (Cplus2ASP processing time + Clingo grounding time)
[c] (Clingo pre-processing time + solving time)

Figure 4 Experiments with CCalc and Cplus2ASP

We ran both CCalc and Cplus2ASP on a series of benchmark problems designed to utilize a variety of CCalc syntactic elements and tested the speed of each program with respect to grounding and solving. These tests included all examples from [9] along with specific versions of the larger domains described in [1]. Figure 4 shows the performance comparison of CCalc and Cplus2ASP on the Traffic World domain from [1] using the same scenarios described there, plus one more that is a scaled-up version of Scenario 3. All tests were run in a native install of Ubuntu on a machine with a 3.2 GHz Pentium 4 processor and 2 GB of RAM. Overall Cplus2ASP consistently performs significantly faster than CCalc, producing identical solutions to those of CCalc. The preparation times that are spent for Cplus2ASP in producing the input to Clingo are negligible, as they do not involve grounding.

5 Conclusion

Based on the theoretical result that turns nonmonotonic causal logic into the stable model semantics, we presented a method that represents the language of the Causal Calculator in answer set programming, and implemented it in a prototype called Cplus2ASP. In comparison with Coala, Cplus2ASP allows the full expressivity of action language $\mathcal{C}+$ using input language syntax that is almost identical to the language of CCalc. Our ongoing work involves making Cplus2ASP fully compatible with CCalc by implementing the remaining features of CCalc missing in Cplus2ASP.

Acknowledgements: We are grateful to Michael Bartholomew, Vladimir Lifschitz, Yunsong Meng, Ravi Palla, and anonymous referees for their useful comments on this paper. The authors were partially supported by the National Science Foundation under Grant IIS-0916116 and by the IARPA SCIL program.

References

1. Varol Akman, Selim Erdoğan, Joohyung Lee, Vladimir Lifschitz, and Hudson Turner. Representing the Zoo World and the Traffic World in the language of the Causal Calculator. *Artificial Intelligence*, 153(1–2):105–140, 2004.
2. Alessandro Armando, Enrico Giunchiglia, and Serena Elisa Ponta. Formal specification and automatic analysis of business processes under authorization constraints: an action-based approach. In *Proceedings of the 6th International Conference on Trust, Privacy and Security in Digital Business (TrustBus'09)*, 2009.
3. Alexander Artikis, Marek Sergot, and Jeremy Pitt. Specifying norm-governed computational societies. *ACM Transactions on Computational Logic*, 9(1), 2009.
4. Ozan Caldiran, Kadir Haspalamutgil, Abdullah Ok, Can Palaz, Esra Erdem, and Volkan Patoglu. Bridging the gap between high-level reasoning and low-level control. In *Proceedings of International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, 2009.
5. Paolo Ferraris. A logic program characterization of causal theories. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*, pages 366–371, 2007.
6. Paolo Ferraris, Joohyung Lee, Yuliya Lierler, Paolo Lifschitz, and Fangkai Yang. Representing first-order causal theories by logic programs. *TPLP*, 2010. To appear.
7. Paolo Ferraris, Joohyung Lee, and Vladimir Lifschitz. Stable models and circumscription. *Artificial Intelligence*, 175:236–263, 2011.
8. Martin Gebser, Torsten Grote, and Torsten Schaub. Coala: A compiler from action languages to asp. In *Proceedings of European Conference on Logics in Artificial Intelligence (JELIA)*, pages 360–364, 2010.
9. Enrico Giunchiglia, Joohyung Lee, Vladimir Lifschitz, Norman McCain, and Hudson Turner. Nonmonotonic causal theories. *Artificial Intelligence*, 153(1–2):49–104, 2004.
10. Enrico Giunchiglia and Vladimir Lifschitz. An action language based on causal explanation: Preliminary report. In *Proceedings of National Conference on Artificial Intelligence (AAAI)*, pages 623–630. AAAI Press, 1998.
11. Joohyung Lee. *Automated Reasoning about Actions*[6]. PhD thesis, University of Texas at Austin, 2005.
12. Joohyung Lee, Vladimir Lifschitz, and Ravi Palla. A reductive semantics for counting and choice in answer set programming. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, pages 472–479, 2008.
13. Joohyung Lee, Vladimir Lifschitz, and Ravi Palla. Safe formulas in the general theory of stable models (preliminary report). In *Proceedings of International Conference on Logic Programming (ICLP)*, pages 672–676, 2008.
14. Joohyung Lee and Ravi Palla. System F2LP – computing answer sets of first-order formulas. In *Procedings of International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, pages 515–521, 2009.
15. Vladimir Lifschitz and Fangkai Yang. Translating first-order causal theories into answer set programming. In *Proceedings of the European Conference on Logics in Artificial Intelligence (JELIA)*, 2010.
16. Norman McCain. *Causality in Commonsense Reasoning about Actions*[7]. PhD thesis, University of Texas at Austin, 1997.

[6] `http://peace.eas.asu.edu/joolee/papers/dissertation.pdf`
[7] `ftp://ftp.cs.utexas.edu/pub/techreports/tr97-25.ps.gz`

Static Type Checking for the Q Functional Language in Prolog

Zsolt Zombori[1], János Csorba[1], and Péter Szeredi[1]

1 **Department of Computer Science and Information Theory**
Budapest University of Technology and Economics
Budapest, Magyar tudósok körútja 2. H-1117, Hungary
`{zombori,csorba,szeredi}@cs.bme.hu`

Abstract

We describe an application of Prolog: a type checking tool for the Q functional language. Q is a terse vector processing language, a descendant of APL, which is getting more and more popular, especially in financial applications. Q is a dynamically typed language, much like Prolog. Extending Q with static typing improves both the readability of programs and programmer productivity, as type errors are discovered by the tool at compile time, rather than through debugging the program execution.

We designed a type description syntax for Q and implemented a parser for both the Q language and its type extension. We then implemented a type checking algorithm using constraints. As most built-in function names of Q are overloaded, i.e. their meaning depends on the argument types, a quite complex system of constraints had to be implemented.

Prolog proved to be an ideal implementation language for the task at hand. We used Definite Clause Grammars for parsing and Constraint Handling Rules for the type checking algorithm. In the paper we describe the main problems solved and the experiences gained in the development of the type checking tool.

1998 ACM Subject Classification I.2.3 Deduction and Theorem Proving

Keywords and phrases logic programming, types, static type checking, constraints, CHR, DCG

Digital Object Identifier 10.4230/LIPIcs.ICLP.2011.62

1 Introduction

The paper presents a type analysis tool for the Q vector processing language, which has been implemented in Prolog. The tool has been developed in a collaborative project between Budapest University of Technology and Economics and Morgan Stanley Business and Technology Centre, Budapest. We emphasize two merits of our system: 1) it enforces certain type declarations on Q programmers, making their code better documented and easier to maintain and 2) we check the type correctness of Q programs and detect type errors that can be inferred from the code before execution.

In Section 2 we give some background information on the Q language and typing. Next, in Section 3 an overview of the type analysis tool is presented. The subsequent three sections discuss the three main tasks that had to be solved in the development of the application: extending Q with a type description language (Section 4); implementing the parser (Section 5); and developing the type checker (Section 6). In Section 7 we provide an evaluation of the tool developed and give an outline of future work, while in Section 8 we provide an overview of approaches related to our work. Finally, Section 9 concludes the paper. A more detailed version of this paper is available online as a technical communication at [16].

© Zsolt Zombori, János Csorba and Péter Szeredi;
licensed under Creative Commons License NC-ND

Technical Communications of the 27th International Conference on Logic Programming (ICLP'11).
Editors: John P. Gallagher, Michael Gelfond; pp. 62–72

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

2 Background

In this section we first present the Q programming language. Next, we introduce some notions related to type handling that will be important for understanding the type analyser.

2.1 The Q Programming Language

Q is a highly efficient vector processing functional language, which is well suited to performing complex calculations quickly on large volumes of data. Consequently, numerous investment banks (Morgan Stanley, Goldman Sachs, Deutsche Bank, Zurich Financial Group, etc.) use this language for storing and analysing financial time series [4]. The Q language first appeared in 2003 and is now (December 2010) so popular, that it is ranked among the top 50 programming languages by the TIOBE Programming Community [15].

Types Q is a strongly typed, dynamically checked language. This means that while each expression has a well-defined type, it is not declared explicitly, but stored along its value during execution. The most important types are as follows:

- **Atomic types** in Q correspond to those in SQL with some additional date and time related types that facilitate time series calculations. Q has the following 16 atomic types: `boolean`, `byte`, `short`, `int`, `long`, `real`, `float`, `char`, `symbol`, `date`, `datetime`, `minute`, `second`, `time`, `timespan`, `timestamp`.
- **Lists** are built from Q expressions of arbitrary types.
- **Dictionaries** are a generalisation of lists and provide the foundation for tables. A dictionary is a mapping that is given by exhaustively enumerating all domain-range pairs.
- **Tables** are lists of special dictionaries called **records**, that correspond to SQL records.

Main Language Constructs Q being a functional language, functions form the basis of the language. A function is composed of an optional parameter list and a body comprising a sequence of expressions to be evaluated. Function application is the process of evaluating the sequence of expressions obtained after substituting actual arguments for formal parameters. As an example, consider the expression

```
f: {[x] $[x>0;sqrt x;0]}
```

which defines a function of a single argument x, returning $\sqrt{x}$, if $x > 0$, and 0 otherwise. Note that the formal parameter specification `[x]` can be omitted from the above function, as Q assumes `x`, `y` and `z` to be implicit formal parameters.

Although it is a functional language, Q also has imperative features, such as multiple assignment variables, loops, etc. Q is often used for manipulating data stored in tables. Therefore, the language contains a sublanguage called Q-SQL, which extends the functionality of SQL, while preserving a very similar syntax.

Principles of evaluation In Q, expressions are always parsed from right to left. For example, the evaluation of the expression `a:2*3+4` begins with adding `4` to `3`, then the result is multiplied by `2` and finally, the obtained value is assigned to variable `a`. In Q it is quite common to have a series of expressions f_1 f_2, evaluated as the function f_1 applied to arguments f_2. For example we can define the function $\sqrt[4]{x}$ using the above function for $\sqrt{x}$ as follows: `g : f f`.

There is no operator precedence, one needs to use parentheses to change the built-in right-to-left evaluation order.

Type restrictions in Q The program code environment can impose various kinds of restrictions on types of expressions. In certain contexts, only one type is allowed. For example, in the do-loop `do[n;x*:2]`, the first argument specifies how many times `x` has to be multiplied by `2` and it is required to be an integer. In other cases we expect a polymorphic type. If, for example, function `f` takes arbitrary functions for argument, then its argument has to be of type `A -> B` (a function taking an argument of type `A` and returning a value of type `B`), where `A` and `B` are arbitrary types. In the most general case, there is a restriction involving the types of several expressions. For instance, in the expression `x = y + z`, the type of `x` depends on those of `y` and `z`. A type analyser for Q has to use a framework that allows for formulating all type restrictions that can appear in the program.

2.2 Static vs. Dynamic Typing

The Q language is dynamically typed. This means that the type of an expression is not explicitly associated with it, but the type is stored along the value. However, since the value is only known during execution, there is few possibility of detecting a type error during compilation. In statically typed languages, each expression has a declared type associated with it, which is known before execution. Handling type information requires extra effort, but it allows for compile time detection of some errors, namely type errors.

3 The Q Type Analyser – Architecture

In this section we give a bird's eye view of the architecture of the system. The type analysis can be divided into three parts:

- Pass 1: lexical and syntactic analysis
  The Q program is parsed into an abstract syntax tree structure.
- Pass 2: post processing
  Some further transformations make the abstract syntax tree easier to work with.
- Pass 3: type checking proper
  The types of all expressions are processed, type errors are detected.

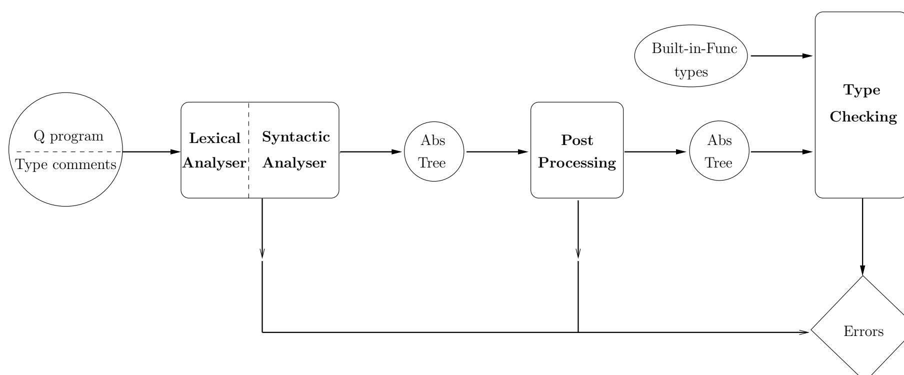

Figure 1 Architecture of the type analyser

The algorithm is illustrated in Figure 1. The analyser receives the Q program along with the user provided type declarations. The lexical analyser breaks the text into tokens. The tokenizer recognises constants and hence their types are revealed at this early stage. Afterwards, the syntactic analyser parses the tokens into an abstract syntax tree representation of the Q program. Parsing is followed by a post processing phase that encompasses various small transformation tasks.

In the post processing phase some context sensitive transformations are carried out, such as filling in the omitted formal parameter parts in function definitions; and finding, for each variable occurrence, the declaration the given occurrence refers to.

Finally, in pass 3, the type analysis component traverses the abstract syntax tree and imposes constraints on the types of the subexpressions of the program. This phase builds on the user provided type declarations and the types of built-in functions. The latter are listed in a separate text file with a syntax that is an extension of the type language described in Section 4. The difference is that the current version of the type analyser does not support polymorphic types for user defined functions, while this is unavoidable for characterising built-in-functions.[1] The predefined constraint handling rules trigger automatic constraint reasoning, by the end of which the types of all subexpressions are inferred.

Each phase of the type analyser detects and stores errors. At the end of the analysis, the user is presented with a list of errors, indicating the location and the kind of error. In case of type errors, the analyser also gives some justification, in the form of conflicting constraints.

The type checking tool has been implemented in SICStus Prolog 4.1 [13]. The subsequent sections deal with three main parts of the development process: extending Q with a type language, implementing parsing and implementing type checking.

4 Extending Q with a Type Language

In order to allow the users to annotate their programs with type declarations, we had to devise a type language that could be comfortably integrated into a Q program. Type annotations appear as Q comments and hence do not interfere with the Q compiler. A type declaration can appear anywhere in the program and it will be attached to the smallest expression that it follows immediately. For example, in the code `x + y //$: int` variable `y` is declared to be an integer.

Due to lack of space, we can only give a brief illustration the type language. The type language contains the atomic types of Q, such as `int, float, real, symbol` and constructors for building lists (`list(int)`), dictionaries (`dict(int,list(int))`), tables (`table(‘name: symbol; ‘age: int)`), records (`record(‘name: symbol; ‘age: int)`) and functions (`int -> symbol`). There are a couple of generic types, such as `numeric` and `any`. Furthermore, we introduced some other types, such as the tuple type, which allow us to describe fixed length generic lists (`tuple(int,real,int)`). Such types require extra care, because the same expression can have different descriptions. For example `(3;2.2;4)` could have type `list(numeric)` or `tuple(int,float,int)`, and this has to be kept in mind constantly during type analysis. Type constructors can be embedded into each other, building complex types of arbitrary depth.

Type declarations can be of two kinds, having slightly different semantics: *imperative* (believe me that the type of expression E is T) or *interrogative* (I think the type of E is T,

[1] We are currently working to extend the type language to polymorphic types. See more about this in Section 7.

but please do check). To understand the difference, suppose the value of `x` is loaded from a file. This means that both the value and the type is determined at runtime and the type checker will treat the type of `x` as `any`. If the user gives an imperative type declaration that `x` is a list of integers, then the type analyser will believe this and treat `x` as a list of integers. If, however, the type declaration is interrogative, then the type analyser will issue a warning, because there is no guarantee that `x` will indeed be a list of integers (it can be anything). Interrogative declarations are used to check that a piece of code works the way the programmer intended. Imperative declarations provide extra information for the type analyser. A Q program is guaranteed to be free of type errors in case the analyser issues no errors and all the imperative declarations are indeed true.

Different comment tags have to be used for introducing the two kinds of declarations. We give an example for each:

```
f //$: int -> boolean        interrogative
g //!: int -> int            imperative
```

5 Parsing

In this section we give an outline of the data structures and algorithms used by the parser component of the type checker. The input of this phase is the original Q program extended with type declarations embedded into Q comments. Its output is the abstract syntax tree and a (possibly empty) list of lexical and syntactic errors. The parser consists of three parts: a lexical analyser, a syntax analyser and a post processor. A particular challenge of parsing Q was that there is no publicly available syntax for the language. We had to use various incomplete tutorials and experiment a lot. Not only is the language poorly documented, we found that it supports lots of exceptions and extreme cases. Some exceptions are demanded by programmers while others only lead to wrong programming habits, hence we were in constant negotiation with Q programmers about what the final syntax should look like.

5.1 Lexical and Syntactic Analysis

As the first processing step, the type checker converts its input Q program to a list of lexical elements, or tokens, for short. The syntax analyser takes a list of tokens as its input and builds an abstract syntax tree representation of the program. In the syntax analyser we exploit the backtracking search of the Prolog language and use the Definite Clause Grammar (DCG) [11] extension of Prolog to perform the parsing.

It is beyond the scope of this paper to present the syntax of the Q language. We only mention that the most frequent expressions are function definitions, function applications, infix function applications, list expressions, table expressions, assignments, identifiers and constant atomic values. The abstract syntax tree form of an expression consists of a node whose label identifies an expression constructor, and whose children are the abstract syntax forms of its subexpressions. For example, the abstract form of a function application is a node labelled with `app`, whose left subtree is the expression providing the function, and the right subtree is the argument (or the list of arguments). Atomic constants are represented with a node labelled `cons`, whose left subtree is the constant itself (as a Prolog string), while the right subtree identifies the type of the constant. As an example, in Figure 2 we show the abstract format of the expression `f (4+2)`.

In the rest of the subsection, we list some of the challenges that we had to overcome for building the syntax analyser.

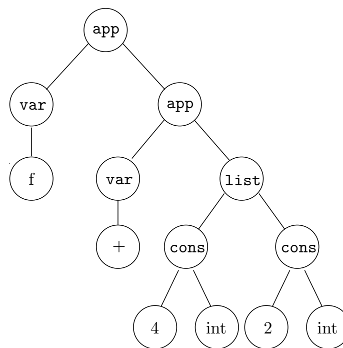

Figure 2 The abstract syntax tree format of the expression `f (4+2)`

Left-Recursion and Evaluation Order Our first version of the grammar was simple to understand, but it was not free from (indirect) left recursion. A formal grammar that contains left recursion cannot be parsed using DCGs. The right-to-left evaluation order of Q also caused difficulties. Both problems were solved using the well known algorithm from [8] to convert the grammar into an equivalent right recursive grammar. The algorithm is based on dividing the problematic nonterminals to a start and a tail part. In this grammar, implementing right-to-left evaluation became simple.

Avoiding Exponential blowup Improper grammar can drive the parsing to exponential run time. Consider the following simplified grammar fragment:

```
expression :=
      ...
      |  assignment
      |  application ;
application :=
      identifier , expression ;
assignment :=
      application , :, expression ;
```

An expression starting with an application can be parsed directly as an application or as an assignment whose left-hand side starts with an application. Hence, the expression $e_1\ \ e_2\ \ldots e_k$ can be parsed in 2^k different ways. The problem occurs whenever there are two disjunctive non-terminals that can be parsed to start with the same expression. We solved this problem by transforming the grammar to eliminate unnecessary choice points.

Parsing Q-SQL The query sublanguage of Q, called Q-SQL, has a syntax that differs from the rest of the code. To make matters worse, some language elements are defined both inside and outside of Q-SQL, with different meaning. For example, ',' is the join operator in Q, but it serves to separate conditional arguments in Q-SQL. The `where` function is another example. The parser had to be prepared for this double parsing, which was further complicated by the fact that one can insert a normal Q expression between parentheses into a Q-SQL expression. This requires knowledge about the current context during parsing. In our solution the DCG clauses were extended with an environment argument, which carries information about the parenthesis depth and the actual position of the parser whenever parsing inside a Q-SQL expression.

5.2 Post Processing

In the post processing phase we perform some transformations on the abstract syntax tree provided by the syntax analyser. The result is another abstract syntax tree conforming to the same abstract syntax. This phase has two main tasks:

- Implicit formal parameters are made explicit, e.g. the function definition `f:{x+y}` is extended to `f:{[x;y] x+y}`.
- Local variables are made globally unique:
  Variable name `x` refers to different variables in different function definitions. Since we would like to associate a type with each variable, we attach a context specific part to the variable name, making all variables globally unique.

6 The Type Analysis Component

In this section we give an outline of the data structures and algorithms used by the type analysis component. The input of this phase is the abstract syntax tree, constructed by the parser. Its output is a (possibly empty) list of type errors.

6.1 Type Analysis Proper

Figure 3 gives a brief summary of the type analysis component. Our aim is to determine whether we can assign a type to each expression of the program in a coherent manner. Some types are known from the start, since the type analyser requires that the Q program to be checked includes type definitions for all user defined functions and variables. Furthermore, we know the types of the built-in functions. The analyser infers the types of further expressions and checks for the consistency of all types.

1. To each node of the abstract syntax tree, we assign a type variable.
2. We traverse the tree and formulate type constraints.
3. Constraint reasoning is used to automatically
   - propagate constraints,
   - deduce unknown types
   - detect and store clashes, i.e., type errors.
4. By the end of the traversal, each node that corresponds to a type correct expression is assigned a type. The types satisfy all constraints.

Figure 3 Overview of the type analysis component

Each expression in the concrete syntax corresponds to a subtree in the abstract syntax. Hence, we maintain a variable (in mathematical sense) for each node of the tree, that stands for the type of the subtree rooted at the node. The task of the type checker is to substitute the variables with proper types.

We use type expressions to describe the type of an expression in the Q language. The type checker uses type expressions similar to those described in Section 4, extended with type variables.

We traverse the tree and formulate context specific constraints on the type of the current node and those of its children. For instance, in the example in Figure 2, when we reach the

`app` node, we know it is a function application, so the left child has to be of type `a -> b`, the right child of type `a` and the whole subtree of type `b`. In some cases the constraint determines the type of some node, but in many others it only narrows down the range of possible values. In case of clash between the restrictions, there is a type error in the program.

The type checker also detects hazardous code that contains potential type error. This is the case when the expected type of some expression is a subtype of the inferred one. An example for this is when a function is declared to expect an integer argument and all we know about the argument is that it is numeric. We cannot determine the runtime behaviour of such a code, since the type error depends on what sort of numeric argument will be provided. Instead of an error, we give a warning in such cases that the user can decide to suppress.

Constraints are handled using the Prolog CHR [12] library. For each constraint, the program contains a set of constraint handling rules. Once the arguments are sufficiently instantiated (what this means differs from constraint to constraint), an adequate rule wakes up. The rule might instantiate some type variable, it might invoke further constraints or else it infers a type error. In the latter case we mark the location of the error, along with the clashing constraint.

In case all variables and user defined functions are provided with a type declaration, we start the analysis with the knowledge of the types of all leaves of the abstract syntax tree. This is because a leaf is either an atomic expression, a variable or a function. Once the leaf types are known, propagation of types from the leaves upwards is immediate, because we can infer the type of an expression from those of its subexpressions. Constraints wake up immediately when their arguments are instantiated, as a result of which the type variables of the inner nodes become instantiated.

6.2 Constraints

The constraints that can be used for type inference come from two sources. First, we know the types of atomic expressions and built-in functions. For example, `2.2` is immediately known to be a float. Similarly, we know that the function `count` is of type `any -> int`. Such knowledge allows us to set – or at least constrain – the types of certain leaves of the abstract syntax tree. The other source of constraints is the language syntax. This can be used to propagate constraints, because the language syntax imposes restrictions on the types of neighbouring nodes.

Besides these type constraints, there can be type information provided by the user at any level of the abstract syntax tree.

Constraint Handling Rules To handle type constraints, we use constraint logic programming. More precisely we use the Prolog CHR (Constraint Handling Rules) library [12], which provides a general framework for defining constraints and describing how they interact with each other. The advantage of CHR is that the constraint variables can take values from arbitrary Prolog structures, so we can comfortably represent all values that a type expression can have.

6.3 Issues about Type Declarations

As we have discussed in Section 3, the user is required to provide every variable and user-defined function with a type description. In this subsection we give reasons for this requirement.

As we have seen before, the immediate benefit is that the types of all leaves of the abstract syntax tree are known at the beginning of the analysis. Without type declarations, some constraints might remain suspended and lots of types unknown. In this case we have to use some sort of labeling to assign a type to each expression.

Furthermore, if the arguments of constraints are ground, we do not have to worry about the interaction of constraints. Consider, for example the following two constraints:

```
int_or_float(X) <=> (X == int ; X == float) | true.
int_or_long(X)  <=> (X == int ; X == long)  | true.
```

If these two constraints apply to type `T`, then they will not do anything as long as `T` is a variable, even though there is only one solution, namely `T=int`. In order for the type analyser to infer this, we have to add a new rule that describes the interaction of the two constraints, such as

```
int_or_float(X), int_or_long(X) <=> X = int.
```

More complex constraints can interact in many different ways and the number of constraint handling rules necessary for capturing all interactions can be exponential in the number of constraints. Given that we work with more than 50 different constraints, it is not realistic to exhaustively write up all rules. If, on the other hand, the arguments are sufficiently instantiated that the constraints can wake up individually (not knowing about the others), then we only need to provide a couple of rules for each constraint. In the above example, if `X` is instantiated, then either `X=int` and both constraints exit successfully or else at least one constraint indicates an error.

7 Evaluation and Future Work

The static type checking tool has been developed in Prolog in about 6 months by the three authors of this paper. Having undergone some initial testing, it is now being evaluated on real-life Q programs at Morgan Stanley Business and Technology Centre. Even in this early stage of testing, the type checking tool pointed out several type errors in real-life Q programs.

Implementation The DCG rule formalism of Prolog was extremely useful in implementing the parser. Because no precise definition for the Q language is publicly available, the syntax accepted by the tool often changed during the development. Hence it was crucial that the parser is easy to modify. For this reason we believe that DCG was a particularly good choice.

Similarly, we were satisfied with the choice of CHR for implementing the type checking. The development of the constraint rules describing the types of the built-in functions was fairly straightforward. Even without rules describing the interaction of constraints, we experienced no performance problems in type checking (although this may change if we move on to type inference).

From Type Checking towards Type Inference In Subsection 6.3 we gave justification for requiring type information about variables and user defined functions. However, it is often rather uncomfortable for the programmer to write so many declarations, so it is worth trying to lift this restriction at least partially. First, (non-function) variables that are initialised do not require a type declaration since the analyser can infer the type from the code. For functions, in many cases it is enough to declare the types of the input parameters, since from them the type of the output can be inferred.

A related question is that of polymorphic types. The type definition language can be extended in a straightforward way to allow polymorphism. In principle, it seems to be feasible to use a variant of the Hindley-Milner algorithm [7] for type inference involving polymorphic types. However, it is yet unclear whether this can be done with acceptable performance, given the large number of overloaded function symbols.

In the immediate future, we plan to further examine and implement all possibilities of omitting type declarations and allowing for polymorphic types.

8 Related Work

Use of Prolog for Parsing Prolog has been used for writing parsers and compilers from its very conception. [2] defines Metamorphosis Grammars as a Prolog extension to support the task of parsing, while [11] describe Definite Clause Grammars as a simplified form of Metamorphosis Grammars. This extension is supported by practically all Prolog implementations today.

[1] give a comprehensive overview of parsing and compiling techniques in the context of Prolog language. [10] reports on the Prolog implementation of a compiler for a programming language called Edison.

Types and constraints Several dynamically typed languages have been extended with a type system allowing for static type checking or type inference. [9] describe a polymorphic type system for Prolog. [6] present a type system for Erlang, which is similar to Q in that they are both dynamically typed functional languages. Several of the shortcomings of this system were addressed in [5]. The tool presented in this work differs from ours in its motivation. It requires no alteration of the code (no type annotations) and infers function types from their usage. Instead of well-typing, it provides success typing: it aims to discover provable type errors. We, on the other hand, guarantee type safety by providing warnings in cases of potential errors. Besides, type annotations are an important means to enhance program readability and although we are working to reduce the number of mandatory annotations, it is very unlikely that we would ever want to eliminate all of them. [3] report on using constraints in type checking and inference for Prolog. They transform the input logic program with type annotations into another logic program over types, whose execution performs the type checking. They give an elegant solution to the problem of handling infinite variable domains by not explicitly representing the domain on unconstrained variables. We believe that their work can be useful for us as we move from type checking towards type analysis. [14] describe a generic type inference system for a generalisation of the Hindley-Milner approach using constraints, and also report on an implementation using Constraint Handling Rules.

9 Conclusions

We presented a type checking tool for the Q language as a Prolog application. We developed a type description language for the type system of Q, which helps in making Q programs easier to read and maintain. We implemented a parser, and a constraint-based type checker. Using constraints enabled us to capture the highly polymorphic nature of built-in functions due to overloading. The type checker provides type safeness: a program that is deemed type correct cannot produce type errors during execution. The tool is now being deployed in an industrial environment, with positive initial feedback.

Acknowledgements

We acknowledge the support of Morgan Stanley Business and Technology Centre, Budapest in the development of the Q type checker system. We are especially grateful to Balázs G. Horváth and Ferenc Bodon for their encouragement and help.

References

1 Jacques Cohen and Timothy J. Hickey. Parsing and compiling using Prolog. *ACM Transactions on Programming Languages and Systems*, 9(2):125–163, 1987.

2 Alain Colmerauer. Metamorphosis grammars. In Leonard Bolc, editor, *Natural Language Communication with Computers*, volume 63 of *Lecture Notes in Computer Science*, pages 133–189. Springer, 1978.

3 Bart Demoen, M. García de la Banda, and P. Stuckey. Type constraint solving for parametric and ad-hoc polymorphism. In *Proceedings of Australian Workshop on Constraints*, pages 1–12, 1998.

4 Kx-Systems. Representative customers . `http://kx.com/Customers/end-user-customers.php`.

5 Tobias Lindahl and Konstantinos F. Sagonas. Practical type inference based on success typings. In Annalisa Bossi and Michael J. Maher, editors, *PPDP*, pages 167–178. ACM, 2006.

6 Simon Marlow and Philip Wadler. A practical subtyping system for Erlang. *SIGPLAN Not.*, 32:136–149, August 1997.

7 Robin Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

8 Robert C. Moore. Removing left recursion from context-free grammars. In *Proceedings of the first conference on North American chapter of the Association for Computational Linguistics*, pages 249–255, May 2000.

9 Alan Mycroft and Richard A. O'Keefe. A polymorphic type system for Prolog. *Artificial Intelligence*, 23(3):295–307, 1984.

10 Jukka Paakki. Prolog in practical compiler writing. *The Computer Journal*, 34(1):64–72, 1991.

11 Fernando C. N. Pereira and David H. D. Warren. Definite clause grammars for language analysis – a survey of the formalism and a comparison with augmented transition networks. *Artificial Intelligence*, 13(3):31–278, 1980.

12 Tom Schrijvers and Bart Demoen. The k.u.leuven chr system: implementation and application. In *First Workshop on Constraint Handling Rules: Selected Contributions*, pages 1–5, 2004.

13 SICS. *SICStus Prolog Manual version 4.1.3.* Swedish Institute of Computer Science, September 2010.
`http://www.sics.se/sicstus/docs/latest4/html/sicstus.html`.

14 Martin Sulzmann and Peter J. Stuckey. HM(X) type inference is CLP(X) solving. *Journal of Functional Programming*, 18:251–283, March 2008.

15 TIOBE. TIOBE programming-community, TIOBE index, 2010. `http://www.tiobe.com`.

16 Zsolt Zombori, János Csorba, and Péter Szeredi. Static type checking for the q functional language in prolog. Technical report, Budapest, University of Technology and Economics, 2011. http://www.cs.bme.hu/ zombori/publications/2011/iclp2011/iclp2011_full.pdf.

Canonical Regular Types

Ethan K. Jackson[1], Nikolaj Bjørner[1], and Wolfram Schulte[1]

1 **Microsoft Research, Redmond, WA**
`ejackson, nbjorner, schulte@microsoft.com`

Abstract

Regular types represent sets of structured data, and have been used in logic programming (LP) for verification. However, first-class regular type systems are uncommon in LP languages. In this paper we present a new approach to regular types, based on *type canonization*, aimed at providing a practical first-class regular type system.

1998 ACM Subject Classification D.3.3 Language Constructs and Features

Keywords and phrases Regular types, Canonical forms, Type canonizer

Digital Object Identifier 10.4230/LIPIcs.ICLP.2011.73

1 Introduction

Regular types describe (infinite) sets of structured data, and have a long history in the verification of logic programs. Briefly, a *data term* is a term t built up from a universe of constants using *data constructors*:

$$t \doteq employee(name(\text{``}John\text{''}, \text{``}Smith\text{''}), id(100))$$

The functions *name*, *employee*, and *id* are constructors for building data; the remaining symbols are constants. A *type term* is a term τ denoting a set of data terms:

$$\tau \doteq employee(name(\text{``}John\text{''}, \mathsf{String}), id(\mathsf{Natural}))$$

The special constants String and Natural denote the sets of all strings and natural numbers. In this example, τ denotes the set of all employee data terms where the employee's first name is "*John*" and the employee's ID is a natural number. We write $[\![\tau]\!]$ for the set of data terms denoted by τ:

$$[\![\tau]\!] \doteq \{employee(name(\text{``}John\text{''}, x), id(y)) \mid x \in [\![\mathsf{String}]\!] \wedge y \in \mathbb{Z}_+ \cup \{0\}\}$$

In this paper we develop a language of type terms to represent, manipulate and canonize regular types. We use the phrase *type term* and *regular type* interchangeably.

Regular types have been primarily used to verify properties of untyped logic programs using the following technique [7]: For each n-ary program relation r define an n-ary data constructor f_r. Compute a type term τ_r such that if $r(x_1, \ldots, x_n)$ holds in the program, then the data term $f_r(x_1, \ldots, x_n)$ is a member of $[\![\tau_r]\!]$. Now, τ_r can be used to check properties of r using type-theoretic operations of *type equality* ($\approx$) and *subtype testing* ($<:$). For example, if $[\![\tau_r]\!]$ is empty then r never holds in the program, which is likely a mistake.

However, unlike other typing paradigms, regular types have not been embraced as a first-class type system in logic programming. Their application remains narrowly scoped to verification of untyped logic programs. There are several reasons why regular types are difficult to use at the language level:

© Ethan K. Jackson, Nikolaj Bjørner, and Wolfram Schulte;
licensed under Creative Commons License NC-ND
Technical Communications of the 27th International Conference on Logic Programming (ICLP'11).
Editors: John P. Gallagher, Michael Gelfond; pp. 73–83

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

No Language-Level Representation Regular types are equivalent to powerful recognizers on data terms, called *non-deterministic tree automata* (NDTAs). Most verification tools use NDTAs to represent types, but this low-level representation is unsuitable for programmer manipulation. Meanwhile, the shallow embedding of regular types via *type programs* [7] does not provide a first-class type system.

No Unique Representation Regardless of representation, regular types are non-unique. Two types may have the same meaning, but look drastically different. Programmers cannot be expected to know if this is the case, because type equality is EXPTIME-complete. This limits the effectiveness of type inference to report useful information.

No Infinite Base Types Existing approaches do not support an infinite universe of constants. This restriction is fundamental to all approaches based on finite tree automata, where it provides key closure properties. Thus, languages with infinite base types, such as the mathematical set of integers, are not supported.

In this paper we present a new approach to regular types aimed at typed logic programs over a first-class regular type system. This approach has been implemented in our LP language FORMULA [1][11]. Our contributions are:

First-class Types and Declarations We define a language for regular types using type terms. Programmers explicitly declare the types of data constructor arguments using type terms. We develop a special class of type declarations, which we call *uniform*, where type equality and subtype testing are in coNP.

Unique Representations Under the uniformity restriction, we develop a *type canonizer* that converts any two semantically equivalent type terms into syntactically identical type terms. In this way, programmers observe the results of type inference as uniquely represented type terms, regardless of the steps taken by type inference. We experimentally show that inferred type terms are small.

New Algorithms/Infinite Base Types We present alternative algorithms based on algebraic manipulation of type terms instead of tree automata. This formalization eliminates the finite signature restriction and supports infinite base types directly.

This paper is organized as follows. Section 2 contains related work. Section 3 formalizes regular types over infinite base types. Section 4 introduces uniform declarations. Section 5 describes the canonization algorithm. We conclude in Section 6.

2 Related Work

Regular types have been primarily been used for verification of untyped logic programs [8, 13]. *Ciao-Prolog* uses a shallow embedding into logic programming [9]. *NU-Prolog* was reported to have first-class regular types [5], while *Mercury* employs a Hindley-Milner style type system [12]. However, other programming paradigms provide support for first-class regular types [10, 3]. Though none of them employ canonical forms to present the results of type inference. Instead, regular types are commonly stored as optimized NDTAs [1]. Further optimizations can be obtained by restricting the class of type declarations [4], which is conceptually similar to our approach. Extensions to regular types include *feature algebras* [2] and may add arrow types [6].

---

[1] See `http://research.microsoft.com/formula`

3 Regular Types and their Semantics

In this section we develop a theory of regular types through a language of type terms. We start from an algebraic signature Σ_Δ, called the *data signature*, identifying the name/arity of data constructors and the universe U of constants. A data signature contains a finite number of non-nullary *function symbols* and may contain an infinite universe of constants U. Here is an example of a data signature: $(\mathbb{Z} \cup \mathbb{S}, employee(,), name(,), age())$, where $\mathbb{S}$ is the set of all strings. Typically, the language predefines many constants in the universe and the programmer defines the non-nullary function symbols.

A *data term* is either a constant, or an application of a Σ_Δ-function to data terms. Importantly, data terms are *untyped*, meaning any arity-respecting sequence of applications is a legal data term, even if it appears to be nonsensical: $name(id(100), employee(40, \text{"}Foo\text{"}))$ Semantically, data terms are interpreted in a very simple way: Two data terms denote the same object if and only if they are identical sequences of function applications and constants. This interpretation yields the *Herbrand Universe* of Σ_Δ, written $\mathcal{H}(\Sigma_\Delta)$, which is the set of all objects labeled by data terms under this rule of equality. From henceforth, the phrases *data constructor* and Σ_Δ*-function* are equivalent.

Regular types allow the programmer to identify the meaningful data terms by describing subsets of data terms. For example, the regular type $name(\mathsf{String}, \mathsf{String})$ identifies all the data terms with string arguments; the unwanted term $name(id(100), id(100))$ does not belong to this set. In our setting a regular type is type term that can be formed using: (1) data constructors, (2) constants, (3) *base types* such as Integer or String, (4) *type variables* such as α_{cons} or α_{list}, and (5) the operations $\cap$ and $\cup$.

3.1 Type Signatures and Terms

Formally, a type term is a term formed over a *type signature* Σ_τ:

▶ **Definition 1** (Type Signature). A *type signature* Σ_τ is extends a data signature Σ_Δ:

$$\Sigma_\tau \doteq (\Sigma_\Delta, V, B, \bot, \cap, \cup). \tag{1}$$

1. V is a (possibly infinite) set of nullary functions called the set of type variables. The symbols in V are disjoint from other symbols.
2. B is a finite set of constants called the set of base types. The symbols in B are disjoint from other symbols, except for the distinguished base type $\bot \in B$, called void.
3. The type intersection ($\cap$) operation and type union ($\cup$) operation are binary operations. These operations are not in Σ_Δ, V, or B.

Conventions A type term τ is a term over Σ_τ, and $\mathcal{H}(\Sigma_\tau)$ is the Herbrand Universe of type terms. Let $\sigma, \sigma_1, \sigma_2, \ldots$ range over the nullary symbols of Σ_Δ and $f, g, \ldots$ range over the non-nullary symbols of Σ_Δ. Also, $\alpha, \alpha_1, \ldots$ range over type variables and $\beta, \beta_1, \ldots$ range over base types. We write $f(\overline{\tau})$ as shorthand for $f(\tau_1, \ldots, \tau_n)$. For the remainder of this paper we assume all data constructors are binary functions. However, all definitions and theorems generalize for functions of arbitrary arity.

3.2 Type Environments and Denotations

The meaning of a type term τ is a set of data terms, which is fixed except for the meaning of type variables. For example, $[\![id(\alpha)]\!]$ depends on the denotation of the type variable α. A *type environment* η provides the missing information in the form of a function from type variables to sets of data terms.

▶ **Definition 2** (Type Environments). A *type environment* $\eta : V \to 2^{\mathcal{H}(\Sigma_\Delta)}$ is a function from type variables to sets of data terms. Set-theoretical operations on $2^{\mathcal{H}(\Sigma_\Delta)}$ can be lifted to type environments by defining:

$$\begin{array}{llll} \eta \sqcap \eta' & \doteq \lambda\alpha \,.\, \eta(\alpha) \cap \eta'(\alpha) & \bot_{env} & \doteq \lambda\alpha \,.\, \emptyset \\ \eta \sqcup \eta' & \doteq \lambda\alpha \,.\, \eta(\alpha) \cup \eta'(\alpha) & \top_{env} & \doteq \lambda\alpha \,.\, \mathcal{H}(\Sigma_\Delta). \end{array}$$

▶ **Definition 3** (Type Denotation). A type denotation function $[\![\ ]\!]_\eta$ is a function from type terms and environments to sets of data terms. It gives a denotation to every type term with respect to a fixed environment.

$$[\![\ ]\!] : \mathcal{H}(\Sigma_\tau) \to \left(V \to 2^{\mathcal{H}(\Sigma_\Delta)}\right) \to 2^{\mathcal{H}(\Sigma_\Delta)}. \tag{2}$$

satisfying:

$$\begin{array}{llll|lll} [\![\bot]\!]_\eta & \doteq & \emptyset. & & [\![\beta]\!]_\eta & \doteq & \{\sigma_1, \sigma_2, \ldots\}, \quad \beta \in B - \{\bot\}. \\ [\![\alpha]\!]_\eta & \doteq & \eta(\alpha), & \alpha \in V. & [\![\tau_1 \cup \tau_2]\!]_\eta & \doteq & [\![\tau_1]\!]_\eta \cup [\![\tau_2]\!]_\eta. \\ [\![\sigma]\!]_\eta & \doteq & \{\sigma\}, & \sigma \in \Sigma_\Delta. & [\![\tau_1 \cap \tau_2]\!]_\eta & \doteq & [\![\tau_1]\!]_\eta \cap [\![\tau_2]\!]_\eta. \\ & & & & [\![f(\tau_1, \tau_2)]\!]_\eta & \doteq & \left\{ f(t_1, t_2) \;\middle|\; \begin{array}{l} t_1 \in [\![\tau_1]\!]_\eta \wedge \\ t_2 \in [\![\tau_2]\!]_\eta \end{array} \right\}, \quad f \in \Sigma_\Delta. \end{array}$$

The denotations of base types are fixed by the language. In other words, they are independent of the environment, denote unique sets, and are closed under intersection. For all $\beta, \beta', \eta, \eta'$:

$$[\![\beta]\!]_\eta = [\![\beta']\!]_{\eta'} \Leftrightarrow \beta = \beta', \quad \exists \beta'' \quad [\![\beta]\!]_\eta \cap [\![\beta']\!]_\eta = [\![\beta'']\!]_\eta.$$

3.3 Type Variables and Declarations

Type variables have a very different use in regular types compared to other type systems. They are used to define recursive data types via a system of *type equations*. Solutions to these equations are type environments where the equations hold. A *type equation* is a pair of type terms, written $\tau \approx \tau'$. A type equation holds for a type environment η if $[\![\tau]\!]_\eta = [\![\tau']\!]_\eta$. Programmers introduce type variables through *type declarations*, which are equations of the form $\alpha \approx \tau$. The smallest solution to these equations gives a unique type environment fixing the denotation of all variables.

▶ **Definition 4** (Type Declarations). A *set of type declarations* $\mathcal{D}$ is a finite set of type equations of the form $\alpha \approx \tau$, where $\alpha \in V$. For each $\alpha \in V$ there is at most one equation in $\mathcal{D}$ with α on the left hand side.

▶ **Example 5** (Declaration of Integer Lists). The following declarations characterize finite lists of integers:

$$\mathcal{D} \doteq \{\alpha_{cons} \approx cons(\mathsf{Integer}, \alpha_{list}),\ \alpha_{list} \approx nil \cup \alpha_{cons}\}, \quad \Sigma_\Delta \doteq (\mathbb{Z}, nil, cons(,)).$$

The solution to this system of equations produces the expected result, because any solution η must have $nil \in [\![\alpha_{list}]\!]_\eta$, implying $[\![cons(\mathsf{Integer}, nil)]\!]_\eta \subseteq \alpha_{cons}$, implying $[\![nil \cup cons(\mathsf{Integer}, nil)]\!]_\eta \subseteq \alpha_{list}$. By induction α_{list} and α_{cons} obtain their usual denotations. We now formalize this result.

▶ **Definition 6** (Least Environment)**.** A set of type declarations distinguishes a unique type environment $\eta(\mathcal{D})$ and denotation $[\![\ ]\!]_{\eta(\mathcal{D})}$, which is the least environment satisfying all type declarations:

$$\eta(\mathcal{D}) \doteq min\left\{ \eta \in Env(\Sigma_\tau) \;\middle|\; \forall \alpha \approx \tau \in \mathcal{D}, [\![\alpha]\!]_\eta = [\![\tau]\!]_\eta \right\}. \tag{3}$$

▶ **Lemma 7.** *The environment $\eta(\mathcal{D})$ exists and is unique. It is the least fixpoint of a monotone operator $\Gamma : Env(\Sigma_\tau) \to Env(\Sigma_\tau)$.*

▶ **Definition 8** (Models Relation)**.** A set of declarations $\mathcal{D}$ may imply additional type equations. Define:

$$\mathcal{D} \models \tau \approx \tau' \quad \doteq \quad [\![\tau]\!]_{\eta(\mathcal{D})} = [\![\tau']\!]_{\eta(\mathcal{D})}. \tag{4}$$

Conventions The $\models$ relation is read: *"$\mathcal{D}$ models the equation $\tau \approx \tau'$"*. Two sets of declarations are equivalent, written $\mathcal{D} \approx \mathcal{D}'$, if they model the same equations:

$$\mathcal{D} \approx \mathcal{D}' \doteq (\forall \tau \approx \tau' \ \mathcal{D} \models \tau \approx \tau' \Leftrightarrow \mathcal{D}' \models \tau \approx \tau'). \tag{5}$$

For the remainder of this paper we drop the subscript $\eta(\mathcal{D})$ from $[\![\ ]\!]$ when the corresponding $\mathcal{D}$ is clear from context. Similarly, we write $\tau \approx \tau'$ instead of $\mathcal{D} \models \tau \approx \tau'$. The type term τ' is a subtype of τ, written $\tau' <: \tau$, if $[\![\tau']\!] \subseteq [\![\tau]\!]$. Equivalently, $\tau' <: \tau$ if and only if $\mathcal{D}$ models the equation $\tau \cap \tau' \approx \tau'$.

▶ **Lemma 9.** *The type intersection ($\cap$) and union ($\cup$) operations inherit the properties of set-theoretical intersection and union: they are idempotent, commutative, associative, and satisfy distributivity and absorbtion properties. Furthermore, the following identities hold for every $\mathcal{D}$:*

(*product*-$\cap$)	$f(\tau_1, \tau_2) \cap f(\tau_1', \tau_2') \approx f(\tau_1 \cap \tau_1', \tau_2 \cap \tau_2')$	
(*disjoint*-f, g)	$g(\tau_1, \tau_2) \cap f(\tau_1', \tau_2') \approx \bot$	*when f and g are different.*
(*disjoint*-σ, f)	$\sigma \cap f(\tau_1, \tau_2) \approx \bot$	
(*disjoint*-β, f)	$\beta \cap f(\tau_1, \tau_2) \approx \bot$	
(*disjoint*-σ, σ')	$\sigma \cap \sigma' \approx \bot$	*when* $\sigma \neq \sigma'$
(*member*)	$\sigma \cap \beta \approx \sigma$	*when* $\sigma \in [\![\beta]\!]$
(*non-member*)	$\sigma \cap \beta \approx \bot$	*when* $\sigma \notin [\![\beta]\!]$
(*base*-$\cap$)	$\beta \cap \beta' \approx \beta''$	*when* $[\![\beta]\!] \cap [\![\beta']\!] = [\![\beta'']\!]$

4 Uniform Declarations

For the remainder of this paper we study type environments generated by a restricted class of type declarations, which we call *uniform declarations*. In order to avoid confusion, let us emphasize that uniformity is a restriction *only* on type declarations $\mathcal{D}$. Arbitrary type terms can be constructed w.r.t. to uniform declarations; as before the denotations of terms continues to be given by $[\![\ ]\!]_{\eta(\mathcal{D})}$. To illustrate this restriction, we begin with an example of declarations that are not uniform:

▶ **Example 10** (Lists of Various Lengths).

$\alpha_{L2} \approx cons(\mathsf{Integer}, cons(\mathsf{Integer}, nil \cup \alpha_{L2})).$
$\alpha_{L3} \approx cons(\mathsf{Integer}, cons(\mathsf{Integer}, cons(\mathsf{Integer}, nil \cup \alpha_{L3}))).$
$\alpha_{L} \approx \alpha_{L2} \cup \alpha_{L3}.$

The type variables α_{L2} and α_{L3} denote lists with lengths divisible by two and three; α_L denotes their union. These types are related in non-trivial ways: $\alpha_{L2} \cap \alpha_{L3} \not\approx \bot$, $\alpha_{L2} <: \alpha_L$, and $\alpha_{L3} <: \alpha_L$. In general, proving these relationships may require witnesses of exponential size. Exponentially large witnesses can be eliminated (and the complexity class reduced) by restricting the structure of type declarations:

▶ **Definition 11** (Uniform Declarations). A set of declarations $\mathcal{D}$ is uniform if:
1. For every $\alpha \approx \tau \in \mathcal{D}$, either τ is free of constructor applications (application-free), or $\tau = f(\tau_1, \tau_2)$ and every τ_i is application-free.
2. For every $f \in \Sigma_\Delta$ there is exactly on equation of the form $\alpha_f \approx f(\tau_1, \tau_2)$.

Example 5 is uniform, while Example 10 is not. Importantly, the uniformity restriction does not prevent arbitrarily precise approximations of general regular types. The types denoted by α_{L2} and α_{L3} can be arbitrarily approximated by the following type terms over the uniform declaration of integer lists:

▶ **Example 12** (Uniform Approximations of Non-uniform Types).

$\tau_{L2(1)} \doteq cons(\mathsf{Integer}, cons(\mathsf{Integer}, nil)).$
$\tau_{L3(1)} \doteq cons(\mathsf{Integer}, cons(\mathsf{Integer}, cons(\mathsf{Integer}, nil))) \ldots$
$\tau_{L2(i)} \doteq \tau_{L2(i-1)} \cup cons(\mathsf{Integer}, cons(\mathsf{Integer}, \tau_{L2(i-1)})).$
$\tau_{L3(i)} \doteq \tau_{L3(i-1)} \cup cons(\mathsf{Integer}, cons(\mathsf{Integer}, cons(\mathsf{Integer}, \tau_{L3(i-1)}))) \ldots$

Observe, $\tau_{L2(i)} <: \alpha_{L2} <: \alpha_{list}$, $\tau_{L3(i)} <: \alpha_{L3} <: \alpha_{list}$, and $\tau_{L2(i)} \cap \tau_{L3(i)} \not\approx \bot$.

This example also illustrates that type terms can have arbitrary nesting of function symbols; uniformity only constrains type declarations. Uniform declarations provide two key results that aid in type canonization. First, type canonization is simpler because all variables not of the form α_f can be eliminated from type expressions over uniform declarations. Second, type equations are easier to decide; they become coNP-complete as opposed to EXPTIME-complete. We now elaborate on these results. Note that similar observations have been made for restricted classes of XML schema [4].

4.1 Orientability and Complexity of Uniform Declarations

In this section we show that uniform declarations can be rewritten to eliminate dependencies on type variables that are not of the form $\alpha_f \approx f(\overline{\tau})$. As a result, we say a variable α is an *auxiliary variable* if its declaration is $\alpha \approx \tau$ and τ is application-free. Orienting the declarations allows many simplifying assumptions when canonizing type expressions. Two lemmas are required to prove the orientability of uniform declarations.

▶ **Lemma 13** (Substitution). *If $\{\alpha \approx \tau, \alpha' \approx \tau'\} \subseteq \mathcal{D}$, then any occurrence of α in τ' can be replaced with τ. In symbols:*

$$\mathcal{D} \approx \mathcal{D} \setminus \{\alpha_1 \approx \tau_1\} \cup \{\alpha_1 \approx \tau_1[\alpha_2/\tau_2]\}, \tag{6}$$

where $\tau'[\alpha/\tau]$ is the replacement of every occurrence of α with τ.

▶ **Lemma 14** (Eliminating Self-Dependencies). *If $\alpha \approx \tau \in \mathcal{D}$, τ is application-free, and α appears in τ, then α can be eliminated from τ.*

▶ **Theorem 15** (Uniform Declarations are Orientable). *A set of uniform declarations $\mathcal{D}$ is equivalent to a set of uniform declarations $\widehat{\mathcal{D}}$ where:*

$$\forall \alpha \approx \tau \in \widehat{\mathcal{D}},\ vars(\tau) \subseteq \{\alpha_f \mid f \in \Sigma_\Delta\}. \tag{7}$$

The problem of deciding a type equation $\tau' \approx \tau$ is to decide if $\mathcal{D} \models \tau' \approx \tau$. For instance, we earlier showed that subtype testing $\tau' <: \tau$ is equivalent to deciding a type equation $\tau' \cap \tau \approx \tau'$. Our solution is by canonization; a *type canonizer* converts two type terms to syntactically identical terms if and only if the terms have the same denotation. First, we prove coNP-completeness of deciding type equalities over uniform declarations. We show this by establishing NP-completeness of the complement problem: checking type disequalities.

▶ **Theorem 16.** *Deciding type disequalities over uniform declarations is NP-complete.*

Proof. The sketch is as follows: Satisfiabiliy of a 3-CNF formula φ can be encoded as the type disequality $\tau(\varphi) \not\approx \bot$. This establishes NP-hardness, but it remains to be shown that if $\tau \not\approx \tau'$ then there is a polynomial size witness. This witness is shown to exist due to the structure of uniform declarations. ◀

5 Canonical Forms

In this section we develop a process for *canonizing* type expressions over uniform declarations. The type canonizer reduces the problem of deciding type equations to checking syntactic equality of canonical forms. To simplify theorems, we assume non-auxiliary variables never denote the empty set: $[\![\alpha_f]\!] \neq \emptyset$. It should be possible to construct at least one well-typed term for a given data constructor.

▶ **Definition 17** (Type Canonizer). For uniform declarations $\mathcal{D}$, a *type canonizer* $can()$ is a function from type expressions to type expressions satisfying:

$$can(\bot) = \bot \quad \wedge \quad \tau \approx can(\tau) \quad \wedge \quad \tau \approx \tau' \Leftrightarrow can(\tau) = can(\tau'). \tag{8}$$

5.1 Eliminating intersection and auxiliary variables

Our canonizer takes advantage of the fact that intersection and auxiliary variables can be eliminated from type expressions. Given an input to the canonizer τ, then all auxilliary variables can be eliminated from τ by Theorem 15 using the rewrites:

$$\begin{aligned} &\alpha_{aux} \rightarrow \bot, && \text{if } \ \alpha_{aux} \approx \tau_{aux} \notin \mathcal{D}. \\ &\alpha_{aux} \rightarrow \tau_{aux}, && \text{if } \ \alpha_{aux} \approx \tau_{aux} \in \widehat{\mathcal{D}}. \end{aligned} \tag{9}$$

Similarly, the equations from Lemma 9 eliminate intersections between non-variable atoms. In the context of uniform type declarations, we can also eliminate intersections involving non-auxiliary variables by using the equations:

(product-$\cap$)	$\alpha_f \cap f(\tau_1', \tau_2') \approx f(\tau_1 \cap \tau_1', \tau_2 \cap \tau_2'),\quad \alpha_f \approx f(\tau_1, \tau_2) \in \mathcal{D}.$
(disjoint-f, g)	$\alpha_f \cap \alpha_g \approx \bot,$ when f and g are different.
(disjoint-f, g)	$\alpha_f \cap g(\tau_1, \tau_2) \approx \bot,$ when f and g are different.
(disjoint-f, σ)	$\alpha_f \cap \sigma \approx \bot$
(disjoint-f, β)	$\alpha_f \cap \beta \approx \bot$

To completely eliminate intersections it remains to apply the set-theoretical properties of union and intersection from Lemma 9: Distribute intersections over unions and apply idempotency to eliminate redundant intersections. Let us call the procedure $\mathit{simplify}(\tau)$ that applies the elimination steps for intersection. We now have:

▶ **Lemma 18.** *For uniform $\mathcal{D}$ and any τ, then $\tau \approx \mathit{simplify}(\tau)$ and $\mathit{simplify}(\tau)$ contains neither intersections nor auxiliary variables.*

The remaining two subproblems are canonizing repeated unions and recognizing unfoldings of recursive data types. For example,

$$\tau = f(0,1) \cup f(1,0) \cup f(1,1), \quad \tau' = f(1, 0 \cup 1) \cup f(0 \cup 1, 1).$$

Casual inspection reveals that $\tau \approx \tau'$, even though the two terms have quite different forms. One approach to canonization is to expand constructor applications, so τ' is rewritten τ. However, this approach guarantees a combinatorial blow-up in the size of the canonical form; it is also problematic when infinite base types appear as subterms. On the other hand, if τ' should be the canonical form, then the canonizer must compress τ into τ'; it is less obvious how this can be done. The approach we present uses τ' as the canonical form, and does not eagerly expand unions into singleton constructor applications.

5.2 Base Case: Canonizing Depth-0 Terms

We build the canonizer inductively; the induction is over the depth of terms.

▶ **Definition 19** (Depth)**.** The depth of an term τ is the length of the longest sequence of constructor applications.

$$\mathit{depth}(\tau) \doteq \begin{cases} 1 + \mathit{max}(\mathit{depth}(\tau_1), \mathit{depth}(\tau_2)) & \text{if } \tau = f(\tau_1, \tau_2) \\ \mathit{max}(\mathit{depth}(\tau_1), \mathit{depth}(\tau_2)) & \text{if } \tau = \tau_1 \cup \tau_2, \text{ or } \tau = \tau_1 \cap \tau_2 \\ 0 & \text{otherwise.} \end{cases} \tag{10}$$

A depth-0 term is another name for an application-free term.

▶ **Lemma 20.** *For uniform $\mathcal{D}$ and any term τ, then $\mathit{depth}(\mathit{simplify}(\tau)) \leq \mathit{depth}(\tau)$.*

▶ **Definition 21** (Base Rewrite System)**.** Let $\mathcal{R}_{expand}$ be the term rewrite system given by the rule: $\tau \to \beta \cup \tau$, if $[\![\beta]\!] \subseteq [\![\tau]\!]$. Let $\mathcal{R}_{contract}$ be the term rewrite system: $\tau \cup \beta \to \beta$, if $[\![\tau]\!] \subseteq [\![\beta]\!]$. The term $\mathit{base}(\tau)$ is constructed by first applying $\mathcal{R}_{expand}$ exhaustively (modulo associativity and commutativity (AC) of $\cup$) until no new types can be added to τ. Second, $\mathcal{R}_{contract}$ is applied to eliminate subsumed types.

▶ **Lemma 22.** *For uniform $\mathcal{D}$ and simplified, depth-0 terms τ and τ'. If $\tau \approx \tau'$, then $\mathit{base}(\tau) = \mathit{base}(\tau')$ modulo AC of $\cup$.*

Lemma 22 guarantees syntactic equivalence up to ordering of unions. However, we would like to handle this order precisely. Fix an arbitrary ordering $\prec$ on type terms, then a type term τ is *sorted* if τ is of the form $\tau_1 \cup (\tau_2 \cup (\tau_3 \cup (\ldots \cup \tau_n)))$, and $\tau_1 \prec \tau_2 \prec \tau_3 \prec \ldots \prec \tau_n$, and the type terms $\tau_1, \ldots, \tau_n$ are not union expressions. Let *sort* be a function that takes any type term τ and rewrites it into an equivalent but sorted expression $\mathit{sort}(\tau)$.

▶ **Corollary 23** (Depth-0 Canonizer)**.** *Define can_0 as a function from depth-0 terms to depth-0 terms such that: $\mathit{can}_0(\tau) \doteq \mathit{sort}(\mathit{base}(\mathit{simplify}(\tau)))$. Then can_0 is a type canonizer for depth-0 expressions.*

5.3 Induction: Terms of Depth $k > 0$

The induction step builds a canonizer can_{k+1} for terms with at most depth $k+1$ from a canonizer can_k for terms with at most depth k. When canonizing union expressions, we utilize the lattice-theoretic properties induced by a can_k canonizer:

▶ **Definition 24.** Let $S = \{\sigma_1, \ldots, \sigma_n\}$ be a finite set of constants and $k \geq 0$ then $\Sigma_\tau(S, k)$ is the set of all terms τ such that $depth(\tau) \leq k$ and $cnsts(\tau) \subseteq S$. Note, $cnsts(\tau)$ is the set of constants appearing as subterms of τ.

▶ **Lemma 25** (S, k-Canonical Lattice). *Let S be a finite set of constants, and can_k be a canonizer for expressions with depth at most k. Define*

$$L(S,k) \doteq \langle can_k(\Sigma_\tau(S,k)), \bot, \top, \sqcap, \sqcup \rangle, \text{ where}$$

$$\bot \doteq \bot_{\Sigma_\tau}, \qquad \top \doteq can_k((\bigcup_{\alpha \in V} \alpha) \cup (\bigcup_{\beta \in B} \beta) \cup (\bigcup_{\sigma \in S} \sigma)),$$

$$\tau \sqcap \tau' \doteq can_k(\tau \cap \tau'), \qquad \tau \sqcup \tau' \doteq can_k(\tau \cup \tau').$$

Then, $L(S,k)$ is a finite lattice such that $\tau \sqcap \tau' \approx \tau \cap \tau'$ and $\tau \sqcup \tau' \approx \tau \cup \tau'$.

The effect of constructor applications on canonized terms of depth k can be understood as an f-labeled product lattice:

▶ **Lemma 26** (f, S, k-Canonical Lattice). *Let f be a binary constructor, S a finite set of constants, and can_k a canonizer, define*

$$L(f,S,k) \doteq \langle U, \bot, \top, \sqcap, \sqcup \rangle, \text{ where}$$

$$U \doteq \bot_{\Sigma_\tau} \cup \left\{ f(\tau, \tau') \;\middle|\; \begin{array}{l} \tau \in L(S,k) \setminus \{\bot_{\Sigma_\tau}\}, \\ \tau' \in L(S,k) \setminus \{\bot_{\Sigma_\tau}\} \end{array} \right\}.$$

$$\bot \doteq \bot_{\Sigma_\tau}. \quad \top \doteq f(\top_{L(S,k)}, \top_{L(S,k)}).$$
$$f(\tau_1, \tau_2) \sqcap f(\tau'_1, \tau'_2) \doteq f(\tau_1 \sqcap_{L(S,k)} \tau'_1, \tau_2 \sqcap_{L(S,k)} \tau'_2).$$
$$f(\tau_1, \tau_2) \sqcup f(\tau'_1, \tau'_2) \doteq f(\tau_1 \sqcup_{L(S,k)} \tau'_1, \tau_2 \sqcup_{L(S,k)} \tau'_2).$$

Then, $L(f,S,k)$ is a finite lattice where $\tau \sqcap \tau' \approx \tau \cap \tau'$ and $\tau \cup \tau' <: \tau \sqcup \tau'$.

The elements of $L(f,S,k)$ already denote unique type expressions (otherwise $L(f,S,k)$ would not be a lattice), and have maximum depth $k+1$. However, the join operation may over-approximate the union of two elements: $f(0,0) \sqcup f(1,1) = f(0 \cup 1, 0 \cup 1)$. Thus, $L(f,S,k)$ cannot be immediately used to canonize unions of terms with depth $k+1$. The following lemmas overcome this limitation:

▶ **Theorem 27** (Maximal Decomposition). *Given $\tau = \tau_1 \cup \ldots \cup \tau_n$ such that $\tau_1, \ldots, \tau_n \in L(f,S,k)$, then a maximal decomposition of τ is an element $\tau' \in L(f,S,k)$ such that:*

$$[\![\tau']\!] \subseteq [\![\tau]\!] \quad \wedge \quad \forall \tau'' \in L(f,S,k)\ \tau' \sqsubset \tau'' \Rightarrow [\![\tau'']\!] \not\subseteq [\![\tau]\!]. \tag{11}$$

Let $decs(\tau)$ be a type term that is the union of all maximal decompositions. It is computed by saturating τ w.r.t. the following implications and keeping the $L(f,S,k)$-maximal subterms:

$$f(\tau_1,\tau_2), f(\tau'_1,\tau'_2) \in \tau \ \Rightarrow\ f(\tau_1 \sqcap \tau'_1, \tau_2) \in \tau. \qquad f(\tau_1,\tau_2), f(\tau'_1,\tau'_2) \in \tau \ \Rightarrow f(\tau_1, \tau_2 \sqcap \tau'_2) \in \tau.$$
$$f(\tau_1,\tau_2), f(\tau_1,\tau_3) \in \tau \ \Rightarrow f(\tau_1, \tau_2 \sqcup \tau_3) \in \tau. \qquad f(\tau_2,\tau_1), f(\tau_3,\tau_1) \in \tau \ \Rightarrow f(\tau_2 \sqcup \tau_3, \tau_1) \in \tau.$$

▶ **Lemma 28** (Correctness\Uniqueness of Decompositions). *Modulo AC of* $\cup$*:*

$$\tau \approx decs(\tau) \quad \wedge \quad \tau \approx \tau' \;\Leftrightarrow\; decs(\tau) = decs(\tau')$$

The union of all maximal decompositions has the same denotation as the original type expression, but is unique for a lattice $L(f, S, k)$. We now have almost all the ingredients to define a complete canonizer. The final task is to ensure that recursive data types are folded and unfolded consistently. Suppose τ is an expression of depth at most $k+1$ and has the form $f(\tau_1', \tau_1'') \cup \ldots \cup f(\tau_n', \tau_n'')$, where every τ_i', τ_i'' is canonical with respect to can_k. We can use a maximal decomposition to canonize this union by creating the type expression $sort(fold(decs(\tau)))$ where *fold* replaces constructor applications with non-auxiliary type variables. The *unfold*() operation expands type variables with their declarations:

$$
\begin{array}{lcll}
fold(f(\tau_1,\tau_2)) & \doteq & \alpha_f, & \alpha_f \approx f(\tau_1^f,\tau_2^f) \in \widehat{\mathcal{D}}, \tau_i = can_k(\tau_i^f), i = 1,2. \\
fold(\tau_1 \cup \tau_2) & \doteq & fold(\tau_1) \cup fold(\tau_2). & \\
fold(\tau) & \doteq & \tau, & \text{otherwise.} \\
unfold(\alpha_f) & \doteq & f(can_k(\tau_1^f), can_k(\tau_2^f)), & \alpha_f \approx f(\tau_1^f,\tau_2^f) \in \widehat{\mathcal{D}}. \\
unfold(f(\tau_1,\tau_2)) & \doteq & f(can_k(\tau_1), can_k(\tau_2)), & can_k(\tau_1) \neq \bot \wedge can_k(\tau_2) \neq \bot. \\
unfold(f(\tau_1,\tau_2)) & \doteq & \bot, & can_k(\tau_1) = \bot \vee can_k(\tau_2) = \bot. \\
unfold(\tau_1 \cup \tau_2) & \doteq & unfold(\tau_1) \cup unfold(\tau_2). &
\end{array}
$$

Then define the canonizer for such terms to be $can_{f,k+1}(\tau) \;\doteq\; sort(fold(decs(unfold(\tau))))$.

▶ **Lemma 29** (Canonizer for Unions of f-terms). *Let τ be a sequence of unions of either the type variable α_f or an f-term with depth $\leq k+1$. Also, suppose $\alpha_f \approx f(\tau_1^f, \tau_2^f) \in \widehat{\mathcal{D}}$ and $S = cnsts(\tau) \cup cnsts(\tau_1^f) \cup cnsts(\tau_2^f)$. Then, $can_{f,k+1}$ is a canonizer for terms in $\Sigma_\tau(S, k+1)$.*

The full canonizer is a obtained by canonizing base types together with constants, and then canonizing for each constructor in the signature: If $simplify(\tau)$ has the form:

$$\underbrace{\alpha_f \cup f(\tau_1,\tau_2)..}_{\tau_f} \cup \underbrace{g(\tau_3,\tau_4) \cup g(\tau_5,\tau_6)}_{\tau_g} \cup \underbrace{\sigma_1 \cup \sigma_2 \ldots \cup \beta_1 \cup \beta_2..}_{\tau_c}$$

▶ **Theorem 30** (Canonizer can_{k+1}). *Assume can_k is a canonizer for terms of depth at most k, then can_{k+1} is a canonizer for terms with depth at most $k+1$, where:*

$$can_{k+1}(\tau) \doteq sort(base(can_{f,k+1}(\tau_f) \cup can_{g,k+1}(\tau_g) \cup \tau_c)) \tag{12}$$

6 Conclusion

We developed a type canonizer for regular types expressed as type terms over uniform declarations. The canonizer solves type checking problems and returns type judgments as canonical type terms. We implemented a type system using this approach in the LP language FORMULA [11], and experimentally showed that canonization times behave linearly while canonical forms are of high quality. Please see the full technical report at `http://research.microsoft.com/~ejackson` for experimental data. Future work includes studying the interaction of regular types and constraints: It is well-known that Presburger constraints describe regular sets, so constraints, such as $x = 2y$, can be used to infer that x is even.

References

1. Alexander Aiken and Brian R. Murphy. Implementing Regular Tree Expressions. In *FPCA 1991*, pages 427–447. Springer-Verlag, 1991.
2. Hassan Aït-Kaci and Andreas Podelski. Towards a Meaning of LIFE. *J. Log. Program.*, 16(3):195–234, 1993.
3. Véronique Benzaken, Giuseppe Castagna, and Alain Frisch. CDuce: an XML-centric general-purpose language. In Colin Runciman and Olin Shivers, editors, *ICFP 2003*, pages 51–63. ACM, 2003.
4. Lei Chen and Haiming Chen. Subtyping Algorithm of Regular Tree Grammars with Disjoint Production Rules. In *ICTAC 2010*, pages 45–59, 2010.
5. Philip W. Dart and Justin Zobel. A Regular Type Language for Logic Programs. In *Types in Logic Programming*, pages 157–187. MIT Press, 1992.
6. Alain Frisch, Giuseppe Castagna, and Véronique Benzaken. Semantic subtyping: Dealing set-theoretically with function, union, intersection, and negation types. *J. ACM*, 55(4), 2008.
7. T. Fruhwirth, E. Shapiro, M.Y. Vardi, and E. Yardeni. Logic programs as types for logic programs. In *LICS 1991*, pages 300 –309, July 1991.
8. John P. Gallagher and Germán Puebla. Abstract Interpretation over Non-deterministic Finite Tree Automata for Set-Based Analysis of Logic Programs. In *PADL 2002*, pages 243–261, 2002.
9. Manuel V. Hermenegildo, Germán Puebla, Francisco Bueno, and Pedro López-García. Integrated program debugging, verification, and optimization using abstract interpretation (and the Ciao system preprocessor). *Sci. Comput. Program.*, 58(1-2):115–140, 2005.
10. Haruo Hosoya, Jerome Vouillon, and Benjamin C. Pierce. Regular expression types for XML. *ACM Trans. Program. Lang. Syst.*, 27(1):46–90, 2005.
11. Ethan K. Jackson, Eunsuk Kang, Markus Dahlweid, Dirk Seifert, and Thomas Santen. Components, platforms and possibilities: towards generic automation for MDA. In *EMSOFT 2010*, pages 39–48, 2010.
12. David Jeffery, Fergus Henderson, and Zoltan Somogyi. Type Classes in Mercury. In *ACSC 2000*, pages 128–135, 2000.
13. Claudio Vaucheret and Francisco Bueno. More Precise Yet Efficient Type Inference for Logic Programs. In *SAS 2002*, pages 102–116, 2002.

Compiling Prolog to Idiomatic Java

Michael Eichberg[1]

1 **Department of Computer Science**
Technische Universität Darmstadt
eichberg@informatik.tu-darmstadt.de

Abstract

Today, Prolog is often used to solve well-defined, domain-specific problems that are part of larger applications. In such cases, a tight integration of the Prolog program and the rest of the application, which is commonly written in a different language, is necessary. One common approach is to compile the Prolog code to (native) code in the target language. In this case, the effort necessary to build, test and deploy the final application is reduced.

However, most of the approaches that achieve reasonable performance compile Prolog to object-oriented code that relies on some kind of virtual machine (VM). These VMs are libraries implemented in the target language and implement Prolog's execution semantics. This adds a significant layer to the object-oriented program and results in code that does not look and feel native to developers of object-oriented programs. Further, if Prolog's execution semantics is implemented as a library the potential of modern runtime environments for object-oriented programs, such as the Java Virtual Machine, to effectively optimize the program is more limited. In this paper, we report on our approach to compile Prolog to high-level, idiomatic object-oriented Java code. The generated Java code closely resembles code written by Java developers and is effectively optimized by the Java Virtual Machine.

1998 ACM Subject Classification D.1.6 Logic Programming

Keywords and phrases Prolog, Compiling, Logic Programming, Object-oriented Programming

Digital Object Identifier 10.4230/LIPIcs.ICLP.2011.84

1 Introduction

One application area, in which Prolog is often used today, is static source code analysis [6, 7, 2, 11]. In this area, the tight integration of the Prolog program with the rest of the application is crucial. Developers using modern IDEs expect to be able to download corresponding source-code analysis plug-ins for their favorite IDE without requiring the installation of further tools. To solve the integration problem and to make deployment a non-issue, it is either possible to use an embedded Prolog interpreter that is written in the language of the rest of the application or to compile the Prolog code to (native) code in the target language. Unfortunately, the performance of Prolog interpreters is generally not sufficient for decent source code analyses. On the other hand, approaches that compile Prolog code to code in the target language usually offer reasonable performance. However, most approaches rely on some kind of virtual machine or framework implemented in the target language [1, 5]. This layer basically implements Prolog's execution semantics and is responsible for managing the overall control-flow and the memory. The result is, that the target code generated for the Prolog code often does not look and feel native to developers familiar with the target language.

In this paper, we report on our approach to compiling Prolog to high-level object-oriented Java code which is effectively optimized by the Java Virtual Machine. Our primary goal is to

© Michael Eichberg;
licensed under Creative Commons License NC-ND
Technical Communications of the 27th International Conference on Logic Programming (ICLP'11).
Editors: John P. Gallagher, Michael Gelfond; pp. 84–94

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

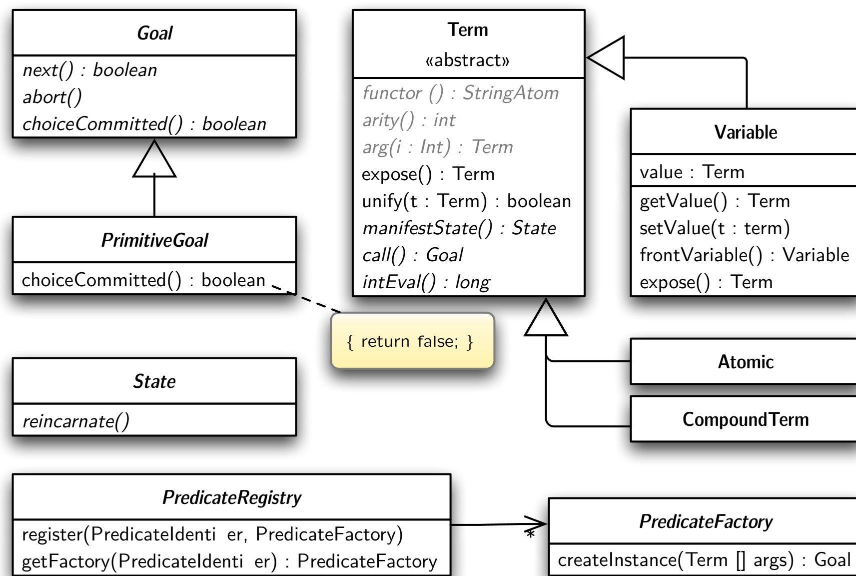

Figure 1 Core Classes and their Dependencies

generate code that is modular, easy to understand and easy to reuse/embed. I.e., we try to generate code that has the least possible impact on the overall application design, but which offers reasonable performance. To achieve this goal, we (A) rely on the functionality offered by Java Virtual Machines for memory management and (B) do not use a framework based approach which prescribes or manages the control-flow and memory used by the Prolog based part of the application. Additionally, at the interface level, the transformed code closely resembles code as written by Java developers to facilitate the integration between the Prolog based code and the (rest of the) Java application. By generating idiomatic code we also want to leverage the optimizations done by modern virtual machines. This optimizations often only fully apply to idiomatic object-oriented programs [4].

This paper is structured as follows. In the next section, we discuss our approach and its implementation — called SAE Prolog. After that, we discuss important related work in Section 3. Section 4 presents the results of our evaluation. This paper ends by drawing some final conclusions in Section 5.

2 Approach

2.1 Overview

The core idea of the chosen approach is to compile each Prolog predicate to one class that implements the complete semantics of the predicate. The goal is that classes, which implement predicates, have minimal dependencies. To this end, we defined one core interface `PrimitiveGoal` which basically defines one core method: `boolean next()` (cf. Figure 1). This interface is implemented by all classes that represent Prolog predicates and enables clients of the class (callers of the predicate) to iterate over all solutions given a certain set of arguments. I.e., the complete execution semantics of the predicate is implemented by the method `next`. The calling contract of `next` is:

```
1. Variable Rs = new Variable();
2. queens2 q = new queens2(atomic(8),Rs);
3. while (q.next()) System.out.println(Rs.toProlog());
```

Figure 2 Java code to get all solutions for the eight-queens problem.

- If `next` returns `true`, a (another) solution exists. A result is returned by binding it to the Prolog variable(s) passed to the class (predicate) when the instance was created.
- If `next` returns `false`, all parameters (Prolog variables) passed to the class are exactly as before the first call to `next`; i.e. variable bindings that were done while trying to prove the goal are undone.

Besides an implementation of the `next` method, each of the classes that implements a predicate has a constructor that defines one parameter for each argument of the corresponding Prolog predicate. The sole purpose of the constructor is to store the parameter values in corresponding fields to make them accessible later on.

For example, let's assume that we have defined a Prolog predicate `queens(N, Rs) :- ...` to solve the n-queens problem. In this case a class called `queens2` will be generated that defines a constructor (`public queens2(Term N,Term Rs){...}`) that takes two parameters: One for the problem size N and one to return a solution/to check a given solution (`Rs`). To get a solution for the eight-queens problem, it is now sufficient to create a new instance of the `queens2` class (Line 2 in Figure 2) and to pass in the problem size N (`atomic(8)`) and a free variable (`Rs` - Lines 1 and 2) to which the (next) solution will be bound, if a/another solution exist. After instantiation, we call `next` to get the first/next solution. If `next` returns `true` (Line 3), the solution is bound to the given Prolog variable; otherwise `Rs` is/remains a free variable.

The property that instances of Prolog variables have the initial state when a predicate has failed is crucial, if we have multiple dependent goals and backtracking does occur. For example, the query `?- X=a(Y),(X=a(1);X=a(2)).`has two solutions: Y == 1 and Y == 2. At runtime, when the `;/2` predicate is called/the `next` method is invoked, it first tries to prove the left goal `X=a(1)` and if it succeeds, the predicate immediately returns `true`. If we now ask for another solution, the left goal is again asked if there is another solution. Since there are no further solutions, the `;/2` predicate immediately tries to prove the right goal (`X=a(2)`). But, this requires that `Y` is (again) a free variable. If `Y` would still be bound to the value `1`, proving the right goal would fail. In general, in our approach we rely on the contract that – after a goal has failed – all terms are exactly as if we would have never tried to prove the (sub-)goal.

Given the proposed approach, it is crucial to be able to efficiently save the information which variable shares with which other variable and which variable is instantiated or free. To achieve this goal we use the State Pattern[8] to manifest the state of a (compound) term (`Term.manifestState():State` in Figure 1). Calling `manifestState` returns a `State` object that enables us to restore the term's state later on by calling `reincarnate`. The State object is basically a list of Prolog variables that do not reference other variables (at the object-oriented level). If several Prolog variables share, we chain these variables such that exactly one variable does not reference any other variable (called the `front variable` in Figure 1). For example, let's assume that the user first unifies the variable `X` with `Y` and then unifies `X` with `Z`. In this case, the Variable object (cf. Figure 1) that represents the Prolog variable `X` will reference the object that represents `Y`; i.e., the field `value` of the object representing `X` will reference the object that represents `Y` and the `value` field of `Y`

```
case X:    init new variables;
           goalStack.put( new <GOAL>(...) ); // fall through
case X+1:  if (!goalStack.peek().next()) {
             goalStack.scrapTopGoal();
             < continue with the next goal after failure >; }
           < continue with the next goal after success >;
```

Figure 3 Initializing and calling a non-deterministic goal.

will remain free. If the Prolog program then unifies `X` and `Z`, the `value` field of the object representing `Z` will reference the object referencing `Y`. If the Prolog program later on unifies `Y` and `Z` "nothing" happens. Whenever two variables are unified, we always first check if their *front variables* are different. In this case the *front variable* of `Z` and `Y` is `Y`. Hence, we know that these two variables already share and no further changes are necessary.

To support the cut operator, the interface `Goal` (cf. Figure 1) defines two further methods: `abort()` and `choiceCommitted():boolean` (cf. Figure 1). The latter method returns `true`, if the goal is the cut operator (`!/0`) or if a subgoal of an or(`;/2`) or and(`,/2`) goal returns `true`. If `true` is returned, we do commit to the current choices. If the immediate subsequent goal fails, no backtracking is done, but instead `abort` is called on all goals preceding the cut. I.e., all terms passed to the previous goals are reset to their initial states.

In our approach, we support calling of terms/predicates using `call(Goal)` by means of a factory [8]. For each predicate, we generate a second class that inherits from `PredicateFactory` (cf. Figure 1) and which defines a single method to create an instance of the predicate given the correct number of arguments. The factory object is registered with a central predicate registry (`PredicateRegistry`) that associates a predicate's identifier (`Functor/Arity`) with the corresponding factory. If the Prolog program at runtime calls a dynamic predicate or uses the `call/1` predicate, we first lookup the predicate's factory object and use it to create an instance of the actual predicate. The lookup functionality to call a term is implemented by the method `call():Goal` of class `Term` (cf. Figure 1).

2.2 Compiling Predicates to Java Code

In the previous section, we outlined the general approach and explained the public interface that all classes share that implement a predicate. In this section, we discuss how we translate a predicate that has multiple clauses to Java code.

The core idea is to represent a clause's control-flow using a switch statement where each goal that is not a conjunction or disjunction of goals is mapped to two case statements. In the following, we refer to these goals as primitive goals. From the point of view of the inter-goal control-flow the first case statement is responsible for handling the initial-call of the goal and the second case statement handles the redo case in case of backtracking. However, w.r.t. initializing and calling a goal the two case statements work closely together and "internally" there is no necessary strict separation.

In case of the call of a non-deterministic predicate the generated code is outlined in Figure 3. In this case, the first case statement first initializes new clause-local variables that are used by the goal, then the goal itself is initialized and then the goal instance is put on the clause-local goal stack. The second case statement then handles the initial call as well as all further attempts to re-satisfy the goal, in case of backtracking. All goals – except of unifications, arithmetic expressions or cuts – are compiled using this schema.

The cut operator is compiled as shown in Figure 4. The first case statement handles the

```
case X:   cutEvaluation = true;
          < continue with the next goal after success >;
case X+1: goalStack.abortAndScrapAllGoals();
          return false;
```

Figure 4 Result of the compilation of the cut operator.

```
case X:   State s1=t1.manifestState(), s2 = t2.manifestState();
          if(t1.unify(t2)) {
             goalStack.put(UndoGoal.create(s1,s2));
             < continue with the next goal after success >;
          } else {
             s1.reincarnate(); s2.reincarnate();
             < continue with the next goal after failure >; }
case X+1: goalStack.abortAndScrapAllGoals();
          < continue with the next goal after failure >;
```

Figure 5 Unification of two terms `t1` and `t2`.

initial "call" of the cut operator. The information that a cut was called is stored and we continue with the next goal. When a subsequent goal fails all previous goals on the goal stack are aborted and we let the clause as a whole fail.

If two terms are unified (cf. Figure 5), we first manifest the state of the terms, before we try to unify them. If they unify, we put a special undo-goal on the clause-local goal stack to be able to later on restore the state of both terms in case of backtracking. If the terms do not unify, we restore the initial state and continue with the next goal.

To be able to map a clause's control-flow to case statements of a switch statement, we first associate each primitive goal with a unique id in the range [0.."number of primitive goals"]. The id of the case statement that handles the initial call is then "`2 * Goal ID`" and the id of the second case statement is "`2 * Goal Id + 1`". Additionally, we analyze the control-flow of the clause to determine the order in which the primitive goals have to be called at runtime. This analysis associates each primitive goal with the goal which needs to be called next, if the current goal succeeds. Furthermore, it associates each primitive goal with the list of those goals that may need to be executed next if the goal fails. This list contains multiple entries if the clause's body contains one or more disjunctions. For example, in case of `(b;c),d`, the goal `b` or `c` may be the direct predecessor of `d` at runtime and if `d` fails at runtime, we need to continue with the goal that preceded `c`. However, the list of goals that need to be executed next if a goal fails, is not to be confused with the list of goals that may precede a certain goal at runtime. E.g., in case of `a,(b;c)` the goal that needs to be called next, if `b` fails, is `c` and not `a`, which will always directly precede `b`. In case of `c` the goal that needs to be executed next if `c` fails, is `a`. Finally, the analysis marks all goals that are not the unique predecessor of its successors as such.

Given the control-flow graph and the ids of the goals, we can then create the code to manage the control-flow of a clause. We discuss our translation scheme, based on the example: `a,(b;c),d`. The result of compiling `a,(b;c),d` is shown in Figure 6. In this case, the goal that will be called next if `a` has succeeded is `b`, if `a` has failed the corresponding list of next goals to execute is empty. In case of `b`, the next goal is `d` if `b` has succeeded and `c` otherwise. If `c` has failed, the next goal is `a` and `d` otherwise. Hence, the goal `d` has two predecessors which may have been called immediately before `d`: `b` and `c` and which may need

```
 1: private int caseToExecute = 0;
 2: private int caseToExecuteIfGoalDFails;
 3: boolean clause() {
 4:    eval_goals: do { switch (caseToExecute) {
 5:       case 0: goalStack.put(new a0());             // goal: a
 6:       case 1: if (!goalStack.peek().next()) {
 7:                  goalStack.scrapTopGoal();
 8:                  return false; }
 9:       case 2: goalStack.put(new b0());             // goal: b
10:       case 3: if (!goalStack.peek().next()) {
11:                  goalStack.scrapTopGoal();
12:                  caseToExecute = 4;
13:                  continue eval_goals; }
14:               caseToExecuteIfGoalDFails = 3;
15:               caseToExecute = 6;
16:               continue eval_goals;
17:       case 4: goalStack.put(new c0());             // goal: c
18:       case 5: if (!goalStack.peek().next()) {
19:                  goalStack.scrapTopGoal();
20:                  caseToExecute = 1;
21:                  continue eval_goals; }
22:               caseToExecuteIfGoalDFails = 5;
23:       case 6: goalStack.put(new d0());             // goal: d
24:       case 7: if (!goalStack.peek().next()) {
25:                  goalStack.scrapTopGoal();
26:                  caseToExecute = caseToExecuteIfGoalDFails;
27:                  continue eval_goals; }
28:               caseToExecute = 7;
29:               return true;
30:    }  } while (true);
31: }
```

Figure 6 The result of compiling "`a,(b;c),d`" to Java code.

to be called, if `d` fails. If `d` succeeds no further goal will be called. We need one variable to store the information which case statement needs to be executed next (`caseToExecute` - Line 1 in Figure 6). Additionally, we create one variable for each goal that has multiple predecessors (e.g., `caseToExecuteIfGoalDFails` Line 2) to store the information which goal preceded it at runtime.

If a goal succeeds, the id of the case statement that needs to be executed next (`caseToExecute`) is set to the id of the first case statement of the target goal (e.g., Line 15 in Figure 6). After that, the evaluation is continued by jumping to the respective case statement (Line 16). If a goal succeeds that has no successor, we set `caseToExecute` to the goal's second case statement. If the clause is called again in case of backtracking, the redo case is executed. After that, we return `true` (Lines 28 and 29). If the current goal is not the unique predecessor of its successor (e.g., `b` and `c` in our example), we additionally save the information which goal preceded its successor (line 14 and 22).

If a goal fails and there are no more alternatives we return `false` (line 8). If there is exactly one alternative, we set `caseToExecute` to the id of the first case statement of the target goal (Lines 13 and 22). After that, we continue the execution with the respective case statement (Lines 12 and 20). If the goal that needs to be executed next cannot be statically

```
   Prolog                     Java
 1:                           private int clauseToExecute = 0;
 2:                           public boolean next() {
 3:                              switch (clauseToExecute) {
 4: goal :- a,                   case 0: if (this.clause0()) return true;
 5:    !.                                if (cutEvaluation) return false;
 6:                                      goalToExecute = 0; clauseToExecute = 1;
 7: goal :- b.                   case 1: if (this.clause1()) return true;
 8:                                      goalToExecute = 0; clauseToExecute = 2;
 9: goal :- c.                   case 2: return this.clause2();
10:                           }  }
```

Figure 7 Code to select the current clause.

```
member(X, [X|_]).
member(X, [_|Ys]) :- member(X, Ys).
```

Figure 8 The predicate `member/2`.

determined, we set the id of `caseToExecute` to the id that was specified by the goal that preceded the current goal (Line 26).

The approach that we have chosen to model the inter-clause control-flow is based on the idea discussed, e.g., in [3]. The control-flow is encoded in the `next` method and uses a switch statement to select the current clause. In Figure 7, the generated code is outlined. After creation, the first clause is called until it fails (Line 4). When it fails and there may have been a cut, we check whether it was effective. If so, the predicate as a whole fails (Line 5), otherwise we continue by calling the next clause (Line 7).

2.2.1 Optimizing Tail-recursive Calls

To demonstrate the potential of this approach, we briefly discuss our last call optimization. Currently, we optimize tail-recursive calls where we can statically decide that there are no more choice points left, when the tail call is made. For example, in case of the predicate `member/2` shown in Figure 8, it is easy to decide that there are no more choice points left if the tail-recursive call is made – "all goals" are deterministic.

The generated code is outlined in Figure 9: First, we save the state of the arguments passed to the predicate (Lines 3 and 4). This is necessary to be able to reset the arguments' state when the predicate eventually fails (Line 15). Second, we embed the `next` method's switch statement into an endless loop (Line 7–15) to be able to jump to a previous clause. If the tail-recursive clause succeeds, we continue with the next iteration of the loop (Line 13) and again start with the first goal of the first clause (Line 12). Third, before the tail call is made, we have to update the fields that store the predicates' arguments (Line 20).

3 Related work

Our approach to compiling Prolog code to Java code uses a large number of ideas originally explored in various Prolog implementations. For example, the idea to make the overall architecture of applications — where only some part is implemented in Prolog — a core design goal was, e.g., explored by tuProlog[10].

```
 1: public member2(final Term arg0, final Term arg1) {
 2:    // ... store the arguments for future use
 3:    initialArg0state = arg0.manifestState();
 4:    initialArg1state = arg1.manifestState();
 5: }
 6: public boolean next() {
 7:    eval_clauses: do { switch(clauseToExecute) {
 8:       ...
 9:       case 1: // tail recursive clause with last call optimization
10:               if (clause1()) {
11:                  < reset clause local variables >;
12:                  goalToExecute = 0; clauseToExecute = 0;
13:                  continue eval_clauses; }
14:               abort(); return false;
15:    }  } while (true);
16: }
17: private clause1() {
18:     // forever, switch ...
19:     case 2: // tail call with last call optimization
20:             arg1 = < Ys >; // update arguments
21:             goalStack.scrapAllGoals();
22:             return true;
23: }
```

Figure 9 Code specific for for the predicate `member/2`.

The approach to compile each clause to its own method is also used by Prolog Café. However, compared to SAE Prolog, Prolog Café uses a WAM-based compilation schema with continuation passing style. The general idea of using a switch statement to branch over the different clauses of a predicate was also described in [3]. However, to the best of our knowledge, the idea to directly map the control-flow of a clause to a switch statement where each goal is represented using two case statements was not explored previously. The proposed approach enables us to treat a cut operator as a normal goal during the calculation of the control-flow graph.

The idea to generate "idiomatic code" was also explored previously. In [9] an approach is described to create idiomatic C code and [4] discuss the generation of idiomatic C# code. However, the meaning of the term idiomatic differs. We consider the code that we generate as idiomatic, because all classes that implement a predicate have a very lightweight interface. Furthermore, we rely on the Java virtual machine for memory management and general optimizations and do not implement our own abstraction layer. In the cited paper, the term idiomatic is used to describe code which uses the primitive data-types and control-flow constructs of the target language. But, to generate such idiomatic code the Prolog predicate requires type and mode annotations and has to be (semi-)deterministic. In all other cases a WAM-based translation scheme is used. Hence, the overall architecture is not idiomatic w.r.t. our understanding of the term.

4 Evaluation

We have done a twofold preliminary evaluation of our approach. First, we compared SAE Prolog to other Prolog implementations to assess its performance. Second, we evaluated it

Table 1 Performance comparison. All times are given in seconds; the programs tak, qsort, primes, queens-8 (findall), chat parser were each run several times.

Program	JLog 1.3.6	Prolog Café 1.2.5	SWI-Prolog (-O) 5.8.3, 64 bits	SAE Prolog 0.1.2
8-queens (findall)	141,85	6,30	6,15	5,09
22-queens	515,87	19,80	23,45	15,65
qsort	*failed*[a]	*failed (OOM)*[b] 4,60	4,20	4,74
primes	848,77	*failed (OOM)* 21,00	18,76	16,41
nrev	*failed (OOM)*	0,29	0,17	0,85
tak	*failed (OOM)*	21,88	16,36	14,69
chat parser	250,94	18,92	9,54	54,01

[a] The query did not return the expected result.
[b] After changing the test harness, we were able to run the unmodified program.

w.r.t. the desired property that it generates code that the Java virtual machine is able to effectively optimize. All performance measurements were done on a 2,33 GHz Core 2 Duo, with 3GB RAM and a Java HotSpot(TM) 64-Bit Server VM.

4.1 Performance Comparison

We compared our implementation with: JLog[`http://jlogic.sourceforge.net`], Prolog Café[1] and SWI-Prolog[12]. JLog is a Java-based Prolog interpreter that has a straightforward implementation and a clean object-oriented architecture. Prolog Café specifically targets performance and translates Prolog code to Java code by means of the WAM. SWI-Prolog is a mature Prolog implementation that is implemented in C and uses the ZIP VM.

The results of our performance evaluation are summarized in Table 1. When we compare the results of these Prolog implementations, we immediately see that JLog generally performs the worst, which was expected given that it is a Prolog interpreter. Prolog Café and SWI-Prolog are roughly on par. SAE Prolog is a bit faster for queens, tak and primes and is on par with Prolog Café in case of qsort. In case of chat parser and nrev SAE Prolog is considerably slower than Prolog Café and SWI-Prolog. Given the early state of the implementation of SAE Prolog these results are encouraging. Implementing well-known techniques, such as term-indexing, will lead to a significant speedup, as preliminary, manual tests have shown and will help us to close this gap.

4.2 Optimizability of the Generated Code

To assess the optimizability of the generated code, we made a detailed analysis of two of the benchmarks. First, we searched for the first solution to the twenty-queens problem. Finding the first solution usually takes more than one second, but the amount of code (3 predicates) that is involved is very small. The second benchmark that we used is the chat parser benchmark. In this case, the time to execute it once is also very small, but the number of involved predicates is much higher (> 100). We run the base benchmark several times without restarting the VM to make sure that the VM "fully optimized" the given program. Additionally, we did repeat each experiment five times. The results are shown in the Figures 10 and 11.

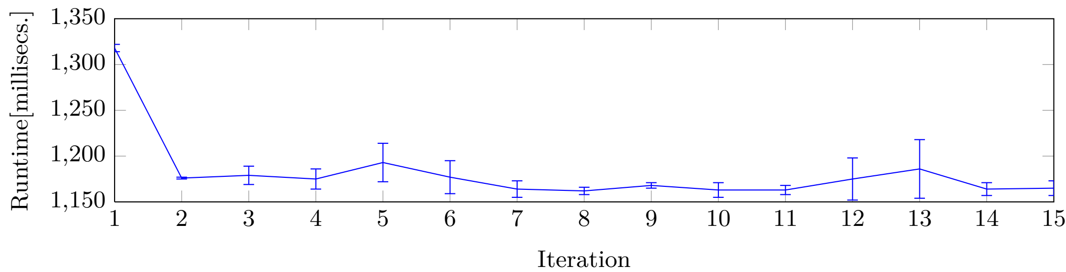

Figure 10 Performance development of the twenty-queens (first solution) benchmark.

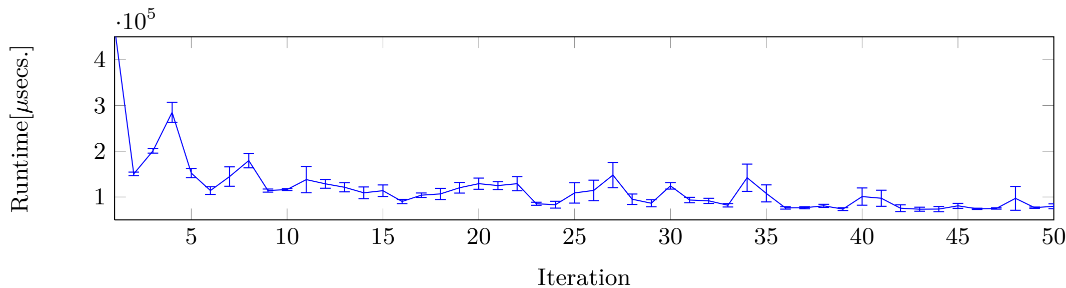

Figure 11 Performance development of the chat parser benchmark.

As shown in the respective figures, the first iteration is always dramatically slower than every succeeding iteration. The initial translation of the Java bytecode to machine code is done as part of this iteration. However, it takes the Java Virtual Machine (JVM) several more iterations (8 for twenty-queens, and 50 for chat parser) before no further performance improvements are measurable. In case of the chat parser benchmark the effect of the step-wise optimizations done by the JVM are immediately obvious. Many of the predicates are not utilized very often and therefore several runs are required before the code is "fully optimized". If we take the second iteration as the base line, we can conclude that the JVM is able to effectively optimize the generated code. The VM is able to improve the performance by a factor of 2 to 3 when compared to the second iteration.

We have repeated this experiment using Prolog Café to gain a better understanding of the optimizability of the code generated by our approach. In case of Prolog Café the VM is also able to further improve the performance. But, the effect is smaller and it takes the VM longer to reach a stable state.

5 Conclusion

In this paper, we have presented an approach to compiling Prolog code to idiomatic object-oriented (Java) code that does not use a virtual machine or framework to implement Prolog's execution semantics. Instead each Prolog predicate is compiled to one class that looks and feels natural to object-oriented developers. Our preliminary evaluation shows that Prolog applications where efficient head unification is not a major concern already perform well and are competitive. Furthermore, we have shown that a modern virtual machine, such as the Java Virtual machine, is able to very effectively optimize the generated program.

Acknowledgment The authors would like to thank Andreas Sewe for providing deep insights into the inner workings of current Java Virtual Machines.

References

1. Mutsunori Banbara, Naoyuki Tamura, and Katsumi Inoue. Prolog cafe : A prolog to java translator system. In *INAP*, pages 1–11. Springer-Verlag, 2005.
2. William C. Benton and Charles N. Fischer. Interactive, scalable, declarative program analysis: from prototype to implementation. In *Proceedings of the 9th ACM SIGPLAN international conference on Principles and practice of declarative programming*, PPDP '07, pages 13–24, New York, NY, USA, 2007. ACM.
3. Joanne L. Boyd and Gerald M. Karam. Prolog in 'c'. *SIGPLAN Not.*, 25:63–71, July 1990.
4. Jonathan J. Cook. Optimizing P#: Translating prolog to more idiomatic C#. In *Proceedings of CICLOPS 2004*, pages 59–70, 2004.
5. Jonathan J. Cook. P#: a concurrent prolog for the .net framework. *Softw. Pract. Exper.*, 34:815–845, July 2004.
6. Michael Eichberg, Sven Kloppenburg, Karl Klose, and Mira Mezini. Defining and continuous checking of structural program dependencies. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 391–400, New York, NY, USA, 2008. ACM.
7. Henry Falconer, Paul H. J. Kelly, David M. Ingram, Michael R. Mellor, Tony Field, and Olav Beckmann. A declarative framework for analysis and optimization. In *Proceedings of the 16th international conference on Compiler construction*, CC'07, pages 218–232, Berlin, Heidelberg, 2007. Springer-Verlag.
8. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.
9. Fergus Henderson and Zoltan Somogyi. Compiling mercury to high-level c code. In *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, pages 197–212, London, UK, 2002. Springer-Verlag.
10. Giulio Piancastelli, Alex Benini, Andrea Omicini, and Alessandro Ricci. The architecture and design of a malleable object-oriented prolog engine. In *Proceedings of the 2008 ACM symposium on Applied computing*, SAC '08, pages 191–197, New York, NY, USA, 2008. ACM.
11. Mohlalefi Sefika, Aamod Sane, and Roy H. Campbell. Monitoring compliance of a software system with its high-level design models. In *Proceedings of the 18th international conference on Software engineering*, ICSE '96, pages 387–396, Washington, DC, USA, 1996. IEEE Computer Society.
12. Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjörn Lager. Swi-prolog. *Theory and Practice of Logic Programming (TPLP)*, to appear.

Synthesis of Logic Programs from Object-Oriented Formal Specifications

Ángel Herranz[1] and Julio Mariño[1]

1 {aherranz,jmarino}@fi.upm.es
Babel Group, Universidad Politécnica de Madrid

Abstract

Early validation of requirements is crucial for the rigorous development of software. Without it, even the most formal of the methodologies will produce the wrong outcome. One successful approach, popularised by some of the so-called *lightweight formal methods*, consists in generating (finite, small) models of the specifications. Another possibility is to build a running prototype from those specifications. In this paper we show how to obtain executable prototypes from formal specifications written in an object oriented notation by translating them into logic programs. This has some advantages over other lightweight methodologies. For instance, we recover the possibility of dealing with recursive data types as specifications that use them often lack finite models.

1998 ACM Subject Classification F.3.1, D.3.1

Keywords and phrases Formal Methods, Logic Program Synthesis, Object-Oriented, Executable Specifications, Correct-by-Construction

Digital Object Identifier 10.4230/LIPIcs.ICLP.2011.95

1 Introduction

Lightweight formal methods [12, 9] have become relatively popular thanks to their success in early validation of requirements, a smooth learning curve and the availability of usable tools. This simplicity is obtained by replacing formal proof – which often demands human intervention – by model checking, but this also implies giving up correctness in favour of a less stringent criterion for models.

Consider, for example, the stepwise specification of queues in Alloy [13]. The specifier might start by just sketching the interface up, like in

```
module myQueue
sig Queue { root: Node }
sig Node  { next: Node }
```

that is, stating that *queues* must have a *root node* and nodes will have a *next* node to follow. The description can be "validated" by fixing a number of Queue and Node individuals and letting a tool like *Alloy Analyzer* [2] model check the specification and show graphically the different instances found. Of course, some of these instances will be inconsistent with the intuition in the specifier's mind – e.g. unreachable nodes or cyclic queues, that can be revealed with very small models. Further constraints, like

```
fact allNodesBelongToOneQueue { all n:Node | one q:Queue | n in q.root.*next }
fact nextNotCyclic {no n:Node | n in n.^next}
```

can be added to the myQueue module in order to supply some of the pieces missing in the original requirements. The first *fact* states that for every node there must be some queue

© Ángel Herranz and Julio Mariño;
licensed under Creative Commons License ND
Technical Communications of the 27th International Conference on Logic Programming (ICLP'11).
Editors: John P. Gallagher, Michael Gelfond; pp. 95–105

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

such that the node lies somewhere in the transitive-reflexive closure of the next relation starting with the root of that node. The second one says that no node can be in the transitive closure of the next relation starting in itself. Model checking the refined specification will generate less instances, thus allowing to explore bigger ones, which will hopefully lead to reveal subtler corners in the requirements.

As said before, this approach is extremely attractive: requirements are refined in a stepwise manner guided by counterexamples found by means of model checking, and the whole process is performed with the help of graphical tools. However, there are also some limitations inherent to this approach. Leaving aside the fact that total correctness of the specification is abandoned in favour of a more relaxed notion of being *not yet falsified by a counterexample*, which can make the whole enterprise unsuitable for safety critical domains, the use of model checking rather than proof based techniques also brings other negative consequences, such as limiting the choice of data types in order to keep models finite, making extremely difficult to model and reason over recursive data types like naturals, lists, trees, etc. (See [13], Ch. 4, Sec. 8.)

A natural alternative to model checking the initial requirements is to produce an executable prototype from them. Using the right language it is possible to obtain recursive code and validation can be guided by *testing*, which might also be automated by tools such as *QuickCheck* [6]. Regarding how to obtain the prototypes, there are several possibilities. One of them is to follow the *correct by construction* slogan and to produce code from the specification, either by means of a transformational approach that often requires human intervention, or by casting the original problem in some *constructive type theory* that will lead directly to an implementation in a calculus thanks to the Curry-Howard isomorphism [21, 5, 4, 20].

Another possibility is to use logic programming. In this case, executable specifications are obtained *free of charge*, as resolution or narrowing will deal with the existential variables involved in any implicit (i.e. non-constructive) specification. Readers familiar with logic programming will remember the typical examples – obtaining subtraction from addition for free, sorting algorithms from sorting test, etc. – and those familiar with logic program transformation techniques will also recognise that these can be used to turn those naive implementations into decent prototypes. However, when it comes to practical usage, none of these formalisms can compete with the lightweight methods above, due to the great distances separating them from the notations used for modelling object oriented software.

This paper studies the synthesis of logic programs from specifications written in an object oriented notation. The specification language, Clay, is being designed around two driving ideas. First, the language must be small but make room for the basic constructs in object oriented programming. Second, specifications must admit at least one translation into executable prototypes to allow the specifier to interactively validate her own specifications. We contribute, on one hand, a *static* theory of types and inheritance that somehow copes with bridging the aforementioned gap between object orientation and logic programming, and, on the other, a *dynamic* part that deals with search in the presence of equality and inheritance (Section 3). To support our contributions we review the examples of the prototypes automatically generated by our tool (Section 4).

2 Object Oriented Specifications in Clay

Clay is a stateless object oriented formal notation, a class-based language with a nominal type system. Classes are defined as algebraic types in the form of case classes: complete and

```
class BSTInt {
  state Empty { }
  state Node { data : Int , left : BSTInt, right : BSTInt }
  modifier insert (x : Int) {
    post { self : Empty ∧
           result = BSTInt.mkNode(x,BSTInt.mkEmpty,BSTInt.mkEmpty)
         ∨ self : Node ∧
           (   x < self.data : True ∧
               result = BSTInt.mkNode(self.data,self. left . insert (x), self . right )
             ∨ x = self.data ∧ result = self
             ∨ x < self.data : False ∧
                result = BSTInt.mkNode(self.data,self. left , self . right . insert (x))) }
  }
  modifier remove (x : Int) {
    post { result . contains (x) : False ∧ result . insert (x)=self}
  }
  observer contains (x : Int) : Bool {
    post { self : Empty ∧ result : False
         ∨ self : Node ∧ x = self.data ∧ result : True
         ∨ self : Node ∧ x < self.data : True ∧ result = self. left .contains (x)
         ∨ self : Node ∧ x < self.data : False ∧ result = self. right .contains (x) }
  }
}
```

Figure 1 Binary search trees in Clay

disjoint subclasses of the defining class. Classes can be extended by subclassing. Methods are specified with pre and postconditions, first order formulae involving self (the recipient), parameters and result (the resulting object). Atomic formulae are equalities ($=$) and class membership ($:$).

An *interlingua* [3] declarative semantics for Clay is provided by translation into first-order logic. Clay tools generate an axiomatisation in Prover9/Mace4 [22] syntax. Then, early detection of inconsistencies is achieved by the combination of automatic theorem proving (Prover9) and model checking (Mace4) of the first order logic theories that reflects the structure of Clay specifications. For the purposes of this paper, the move to logic program synthesis requires, on the *front-end* of the tools, to take some simplifying decisions in order to keep the resulting theory tractable and readable: no multiple inheritance, no overloading (just method refinement) and no parametric polymorphism.

Figures 1 and 2 contain examples of Clay specifications that will guide the whole paper.

2.1 Modelling Data

Let us start with a specification of binary search trees of integers (Figure 1).[1] Instances of a class are the disjoint and complete sum of the instances of its case classes (indicated with keyword **state** due to their similarity to the design pattern *State* [8]): if t is an instance of BSTInt (t : BSTInt) then it is an instance of Empty or, exclusively, of Node. The following Clay formula expresses it formally:

```
∀ t : BSTInt ((t : Empty ∨ t : Node) ∧ t : Empty ⇔ ¬ t : Node)
```

[1] Clay allows parametric polymorphism but we have not used this feature for the sake of conciseness.

```
class Cell {
  state CellCase { contents : Int }
  constructor mkCell {
    post { result .contents = 0 }
  }
  observer get : Int {
    post { result = self.contents }
  }
  modifier set (v : Int) {
    post { result .contents = v }
  }
}
```

```
class ReCell <: Cell {
  state ReCellCase { backup : Int }
  constructor mkReCell {
    post { result = Cell.mkCell ∧
           result .backup = result.contents }
  }
  modifier set (v : Int) {
    post { result .backup = self.contents }
  }
  modifier restore {
    post { result .contents = self.backup ∧
           result .backup = self.backup }
  }
}
```

Figure 2 Inheritance in Clay

The case classes Empty and Node introduce the constructor methods mkEmpty and mkNode. Both are messages that can be sent to the object BSTInt (classes are objects in Clay): BSTInt.mkEmpty creates an instance of the case class Empty and
BSTInt.mkNode(42,BSTInt.mkEmpty,BSTInt.mkEmpty) creates an instance of Node.

Composition Composition is represented by fields defined in a case class. Those fields are methods that project the encapsulated information of its case. In our example the result of the expression BSTInt.mkNode(42,l,r).data is 42.

Inheritance Classes can be extended with subclasses that inherit all the properties of the superclass. In Figure 2, class ReCell extends Cell and therefore its instances obey the property: $\forall$ c : ReCell (c : CellCase).

Inheritance induces a subtype relation (<:) with all its expected laws: reflexivity, transitivity and subsumption. The most important aspect of this relation is that subclasses cannot invalidate by overriding any property specified in a superclass, otherwise the whole specification is considered inconsistent.

This approach is essential when we are specifying in the large: the specifier needs to reason locally to a class and a subclass cannot show a behaviour that forces the specifier to take into account all the subclasses. The approach adds another advantage: specifications can be much more concise since it is not needed to state already stated properties in superclasses. The main drawback is certain loss of flexibility but, in our view, the decision pays back.

The Cell and ReCell classes in Figure 2 are brought from [1]. Instances of Cell are storage-cell objects encapsulating a natural number that can be changed (set) and read (get). The extension of Cell with a restore option yields ReCell. We can observe the conciseness of the overriding of set in ReCell since properties of Cell are inherited.

2.2 Modelling Behaviour

Methods are specified with first order formulae that relate the receiver of the message (self) and the message's parameters with the answer to the message (result). Primitive predicates include equality and class membership.

Class membership is mainly used to do pattern matching. In the specification of method insert we can see how antecedents of implications distinguish between empty and nonempty trees (self : Empty and self : Node).

Equality is particularly interesting in Clay. The predicate is implicitly indexed by the minimum subtype of the compared instances in the context in which the formula appears. The rationale behind this decision has to do with reasoning locally, a more *dynamic* equality would lead unexpected results in the specifier context. In the insert example, the minimum type of result and self in the first disjunct of the post is BSTInt. The semantics establishes that no properties of self other than those *reachable* from BSTInt are enforced in result. In the Cell/ReCell example, formula Cell.mkCellCase(42) = ReCell.mkReCell.set(13).set(42) holds, as only the state relevant in class Cell – the smallest common subtype – is considered.

Keywords **modifier** and **observer** are merely type informative (the result of a modifier is an instance of the class being specified) and has no influence in the semantics of the methods.

In the binary trees example, the specification of insert and contains are, so to say, *explicit* as they describe recursively all the possible situations. The specification of remove, however, is *implicit*: the result is specified by means of a condition that the result must meet with no additional clues.

2.3 Interacting with Clay

The prototype generated by our synthesiser supports interacting with Clay specifications by asking it to reduce a Clay object expression to a normal representation. We describe now some use cases and in Section 4 we will check the actual performance of the synthesised prototype with those use cases.

Inheritance Classes Cell and ReCell will be our first guiding example. We will interact with Clay to check that the compiler is enabling the specifier to write concise specifications with safe inheritance, and we will see the answers to expressions like ReCell.mkReCell.set(0).set(1).get and ReCell.mkReCell.set(0).set(1).restore.get.

Recursive Specifications More interesting examples are the recursive definitions of methods insert and contains of binary search trees in Figure 1. We will interact with the synthesised prototype to check that recursive definitions can be executed.

Implicit Specifications Our last guiding example will be the implicit specification of method remove in the class BSTInt in Figure 1. We will execute some examples sending the message remove to some binary search trees.

Requirements Validation The interaction with Clay should help the specifiers to gain confidence in their specifications. We will detect an error in our previous specifications.

3 Translating Clay Specifications into Logic Programs

The distance between Clay and Prolog is big enough to make the the translation far from trivial and difficult to follow. Its full formalisation can be found in [10]. This section discusses the intuitions behind the main decisions.

Given a Clay specification we will synthesise facts that represent its abstract syntax tree: classes, inheritance, case classes, fields, and pre- and post-conditions of methods. Figure 3 describes the meaning of the target predicates.

The heart of our translator is a common theory for all specifications: *the Clay theory*. The most important predicates of this axiomatisation are (`instanceof/2`, `reduce/2`, and `eq/3`), definitions that rely on the facts translated from the source specifications (Figure 3). Their meaning is:

- Predicate `instanceof(`NF`,`A`)` is a generator of instances NF of a class A. NF is a *normal form* of an instance of A. These normal forms are flexible representation of instances as incomplete data structures and will be presented in Sections 3.1 and 3.2.

`class(`C`)`	C is a Clay's class identifier.
`inherits(`A`,[`B`])`	B is the superclass of A
`cases(`C`,`Cs`)`	Cs is the list with case classes of class C
`fields(`C`,`Fs`)`	Fs is the association list with the field names and field types of case class C
`msgtype(`C`,`M`)`	M is the message identifier of a method defined or overridden in class C
`pre(`C`,`S`,`M`,`As`)`	Precondition for sending message M with arguments As to an instance S of class C
`post(`C`,`S`,`M`,`As`,`R`)`	Postcondition that establishes that R is the resulting instance of sending message M with arguments As to instance S of class C

Figure 3 Representing Clay in Prolog.

- Predicate `eq(`A`,`NF_1`,`NF_2`)`, Clay's equality, decides if the representations (NF_1 and NF_2) of two instances are indistinguishable in class A.
- Finally, predicate `reduce(`E`,`NF`)` reduces any Clay object expression E to its normal form NF. Predicates `eq` and `reduce` will be presented in Sections 3.2 and 3.3.

3.1 Representing Clay Instances in Prolog

We have mentioned that the predicate `reduce/2` reduces a Clay object expression to a normal form. Clay object expressions have a straightforward representation in Prolog:

- A class expression ci<C_1, ..., C_n> is represented by the Prolog term
$$(ci{<}C_1,\ \ldots,\ C_n{>})^{\#} = ci^{\#}\texttt{(}C_1^{\#}\texttt{,}\ \ldots\texttt{,}\ C_n^{\#}\texttt{)}.$$
- A class identifier ci is represented by a valid Prolog constant $ci^{\#}$ by quoting its lexeme.
- A class variable cv (an object variable ov) is represented by a valid Prolog variable $cv^{\#}$ ($ov^{\#}$) by prefixing its name with "_": _cv (_ov).
- A send expression $o.mi(o_1, \ldots, o_n)$ is represented by the Prolog term
$$(o.mi(o_1,\ \ldots,\ o_n))^{\#} = o^{\#}\texttt{<-}mi^{\#}\texttt{(}o_1^{\#}\texttt{,}\ \ldots\texttt{,}\ o_n^{\#}\texttt{)}.$$
- A message identifier mi is represented by a valid Prolog constant $mi^{\#}$ by using its lexeme.

To describe how the generated prototype represents the instances of our language in normal form we will use the example of restorable cells (instances of ReCell). We need to capture all the information of known superclasses (Cell) and to capture all the information about the specific case class (ReCell).

With no multiple inheritance, a sorted linear structure can represent the classes of an instance. Therefore, we can use a list where each element contains the part of the representation for a given class of the instance: `(`C`,`S`,`F`)` where C is the class, S is the particular case class, and F is an association list from field names to the representation of their instances. Let us show the representation of Cell.mkCell:

```
[('Cell','CellCase',[(contents,[('Int','Int',[42])])])]
```

The list contains one element since the object is an instance of just one class (Cell).

Under subtyping, during a deduction process where a cell with 42 is expected an instance of ReCell could appear. If we follow our rules, the representation of ReCell.mkReCell.set(42) would be:

```
[('Cell','CellCase',[(contents,[('Int','Int',[42])])]),
 ('ReCell','ReCellCase',[(backup,[('Int','Int',[0])])])]
```

The representation of the cell with 42 and the instance of ReCell are partially the same but the latter does not *fit* in the former. This is something that we would expect to happen since both instances represent the same information with respect to the properties of Cell.

We propose to make room for yet unknown information of subclasses and to use an incomplete data structure where the incomplete part represents the *room* for the information of the potential subclasses. The representation of Cell.mkCell would be

```
[('Cell','CellCase',[(contents,[('Int','Int',[42])|_])])|_]
```

and for the instance of ReCell we would have the following representation:

```
[('Cell','CellCase',[(contents,[('Int','Int',[42])|_])]),
 ('ReCell','ReCellCase',[(backup,[('Int','Int',[0])|_])])|_]
```

Apart from carrying all the information needed by methods specified in the superclasses, our normal form has the following properties:

- Information about case classes allows us to reflect the disjoint sum (case classes) of products (fields).
- The incomplete part might be instantiated with data of an instance of a subclass (like the backup of ReCell) during the deduction process. The most interesting benefit is that the instantiation can be implemented with the unification of our logic language engine. The example above shows how the instance of ReCell fits, by unification, in the cell with 42.

Predefined Integers

The predefined class `Int` encapsulates integers that get translated into Prolog integers managed via finite domain constraints. This illustrates another technique that can be applied in the translation when the target language has declarative extensions. Previous versions of the same specification used a Peano representation for naturals (predefined class `Nat`) as a way of obtaining a complete theory for numbers. The experiments in Section 4 show drastic gains over our previous implementation presented in [11].

3.2 Atomic Formulae (Instance of and Equality)

The predicate ":" (instance of) is translated into the Prolog predicate `instanceof/2`. Which generates the representation of all instances (first argument) of all classes (second argument) of a specification. Thanks to our incomplete structures every instance of a subclass is an instance of a superclass, a technique that makes the desirable property of subsumption to be a theorem in our Prolog axiomatisation.

Clay equality (=) is the other predicate used in the atomic formulae of Clay in this work. Our translation of Clay equality into Prolog consists of two steps: a reduction of the object expressions to normal form and the unification of the obtained representations.

Let us see a description of the implementation of the reduction step and postpone the formalisation of the translation of the equality literals to Section 3.3. Predicate `reduce/2` relates terms that represent abstract syntax trees of Clay expressions with their normal form. The most important clause of `reduce/2` defines the reduction of sending a message (`M`) to an object expression `O`. Functor `.-`, in infix form, represents the send operator of Clay:

```
reduce(O<--M,NF) :- M =.. [Mid|Args],
                    reduce(O,ONF), reduceall(Args,ArgsNF),
                    knownclasses(ONF,Cs),
                    checkpreposts(Cs,ONF,Mid,ArgsNF,NF,defined).
```

```
modifier insert (x : Int) {
 post {
   self : Empty ∧
   result = BSTInt.mkNode(
              x,
              BSTInt.mkEmpty,
              BSTInt.mkEmpty)


  ∨
   self : Node ∧
  (x < self.data : True ∧

    result = BSTInt.mkNode(
               self.data,
               self.left.insert(x),
               self.right)


   ∨ ...) }
}
```

```
post('BSTInt',_s,insert,[_x],_r) :-
 instanceof(_s,'Empty'),
 reduce('BSTInt'<--mkNode(
          _x,
          'BSTInt'<--mkEmpty,
          'BSTInt'<--mkEmpty),
       _NF_BSTInt_mkNode),
 eq('BSTInt',_r,NF_BSTInt_mkNode).
post('BSTInt',_s,insert,[_x],_r) :-
 instanceof(_s, 'Node'),
 reduce(_x < _s<--data,_NF__x_le),
 instanceof(_NF__x_le,'True'),
 reduce('BSTInt'<--mkNode(
          _s<--data,
          _s<--left<--insert(_x),
          _s<--right),
       _NF_BSTInt_mkNode),
 eq('BSTInt', _r, _NF_BSTInt_mkNode).
...
```

Figure 4 Translation of insert.

```
modifier remove (x : Int) {
 post { result.contains(x) : False

     ∧ result.insert(x)=self
 }
}
```

```
post('BSTInt',_s,remove,[_x],_r) :-
 reduce(_r<--contains(_x), _NF__r_contains),
 instanceof(_NF__r_contains, 'False'),
 reduce(_r<--insert(_x), _NF__r_insert),
 eq('BSTInt',_NF__r_insert,_s).
```

Figure 5 Translation of remove.

Predicate `reduceall/2` reduces a list of expressions, the second argument of `knownclasses/2` contains the known classes (`Cs`) of the recipient of the message, and `checkpreposts` checks pre- and post-conditions of every class of `Cs` in which method `Mid` is defined.

We already mentioned in Section 2 the danger of overriding the properties of methods in subclasses: the practical impossibility of reasoning in large programs. The above implementation of predicate `reduce/2` will fail if any postcondition in the inheritance hierarchy is inconsistent with the postconditions specified in superclasses.

3.3 Translation of Pre- and Post-conditions

The translation of formulae takes into account that objects involved in atomic predicates must be reduced. The translation of non-atomic formulae are directly translated into first-order logic formulae resulting in extended programs (sets of implications with an arbitrary first-order formulae in the body). Then, a Lloyd-Topor transformation [17, 18] is applied to obtain a logic program.

Figures 4 and 5 present, in parallel, the correspondence between the Clay specification of methods `insert` and `remove` of `BSTInt` and the automatically synthesised Prolog code.

4 Experimental Results

Let us get back to the problems posed in Section 2.3. Relevant parts of the code obtained for the recursive definition of insert are shown in Figure 4. We have automatically generated up to 4000 trees with up to 80 nodes with a maximum depth of 10. The insertion of elements works properly and the execution time in the worst case is less than 1000 milliseconds.

The second question was whether Clay would be able to generate an executable prototype from the implicit specification of the remove method for binary search trees. The code obtained is shown in Figure 4.

Running tests on this specification shows several problems. First, the logic program obtained from the specification seemed to be only partially correct. Given a (valid) binary insertion tree and one of its elements, the returned tree was, in some cases, a tree meeting the specification but failed in the rest.

Analysis of the tests revealed that the prototype was working properly exactly in those cases where the element to remove was *at the leaves* of the structure — i.e. in a node with two empty subtrees as children. This solves the mystery: the specification for remove uses predefined (structural) equality while the specifier was probably thinking in the intended set semantics for the trees as collections. The order in which elements are stored in the tree affects its actual shape. That is why only elements that make their way down to the "bottom" of the structure via method insert meet the specification of remove.

In other words, the specification was flawed and the execution allowed us to spot the bug. There are several ways to solve the problem. One of them is, of course, to use a *self normalizing* data structure – balanced tree, heap... – for which predefined equality behaves as set equality. A quicker fix – less efficient – is to *flatten* both sides of the equality:

```
modifier remove (x : Int) {
  post {result.contains(x) : False ∧ result.insert(x).flatten() = self.flatten()}
}
```

where flatten is an observer defined recursively in the obvious way:

```
observer flatten () : List <Int> {
  post { self : Empty ∧ result = []
       ∨ self : Node ∧ result = self.left.flatten.append([].cons(self.data)) .
                                  append(self.right.flatten) }
}
```

We show now the effects of the safe inheritance:

```
?- reduce('Cell'<--mkCellCase(0),R).
R = 'CellCase'{contents : 0}
?- reduce('Cell'<--mkCellCase(0)<--set(1)),R).
R = 'CellCase'{contents : 1}
?- reduce('Cell'<--mkCellCase(0)<--set(1)<--get,R).
R = 1
?- reduce('ReCell'<--mkReCell<--set(0)<--set(1)<--restore<--get,R).
R = 0
?- reduce('Cell'<--mkCell<--set(0)<--set(1)<--restore<--get,R).
no
```

The table below shows performance figures obtained in an Intel Dual Core T7200@2.00GHz, with 4096KB of cache and 2GB of RAM running GNU/Linux 2.6.32-25 SMP and SWI-Prolog v. 5.10.0. The depth limit used for the iterative deepening strategy for predicate instanceof

was 38. The full Clay code for these examples, their translation into Prolog and the Prolog implementation of the Clay theory can be found at `http://babel.ls.fi.upm.es/~angel`.

Test	**Time** (ms.)
Generation of trees (1000 trees)	202
Creation of trees (15 insertions)	929
Removing leaf from tree (1 node)	0
Removing leaf from tree (3 nodes)	391
Removing leaf from tree (7 nodes)	7211
Removing leaf from tree (15 nodes)	18300

5 Related Work and Conclusions

We have presented the compilation scheme of an object oriented formal notation into logic programs. This allows the generation of executable prototypes that help in validating requirements, e.g. by means of automated test generation. We have emphasized the generation of code from implicit method specifications, specially in presence of recursive definitions, something which is seldom supported by other lightweight methods and tools.

Early experiments with our prototype compiler show the feasibility of the approach, but also the limitations of a naive application of Prolog's standard search mechanisms. In fact, obtaining an efficient search scheme is one of the challenges for future research. Our current implementation combines techniques such as the Lloyd-Topor transforms of first-order formulae and iterative deepening search for achieving completeness in some examples.

We expect to increase efficiency with the use of constructive negation and also with techniques that allow for *lazy* instance generation, that is, *coroutining* the logic code that implements quantification via instance generation with the one that implements the implicit postconditions. More mature tools, like ProB [15, 16] already take advantage of these.

One improvement that has already been incorporated in this version is the use of constraints for arithmetic. A previous version used a Peano representation for naturals (predefined class `Nat`) as a way of obtaining a complete theory for numbers. Now, predefined class `Int` encapsulates integers that get translated into Prolog integers managed via finite domain constraints. The tests show drastic gains over our previous implementation.

One aspect hard to implement properly is nondeterminism. If the specs are assumed correct, then it suffices to choose one interpretation at random to obtain an executable prototype. This can be achieved, for instance, by limiting nondeterminism in the logic program generated by always choosing the first solution at any choice point.

But if the goal is to use the prototypes for requirement validation, then choosing the good one by chance does not help. In this case, interpretations must be generated randomly, but any of these must be internally consistent, i.e. methods intended to be deterministic must always return the same answer in each interpretation. Ensuring this in Prolog is trickier and can be achieved, for instance, using tabulation techniques.

Certain features of object oriented programming (e.g. mutable state) have been left out of this presentation. Studying the introduction of state in our code generation scheme would help in applying the ideas presented in this paper to other object oriented formal notations like VDM++, Object-Z, Troll or OASIS [7, 23, 14, 19].

References

1 Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.
2 Alloy Website. `http://alloy.mit.edu`.
3 Jeffrey Van Baalen and Richard E. Fikes. The role of reversible grammars in translating between representation languages. In *Proceedings of the 4th International Conference on Principles of Knowledge Representation and Reasoning*, pages 562–571, 1994.
4 Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Springer Verlag, 2004.
5 James L. Caldwell. Extracting general recursive program schemes in Nuprl's type theory. In *LOPSTR '01*, pages 233–244, London, UK, 2001. Springer-Verlag.
6 Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *ICFP*, pages 268–279, 2000.
7 John Fitzgerald, Peter Gorm Larsen, Paul Mukherjee, Nico Plat, and Marcel Verhoef. *Validated Designs For Object-oriented Systems*. Springer-Verlag TELOS, 2005.
8 E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns - Elements of Reusable Object Oriented Software*. Addison-Wesley, 1995.
9 Vinu George and Rayford Vaughn. Application of lightweight formal methods in requirement engineering. *CrossTalk - The Journal of Defense Software Engineering*, January 2003.
10 Ángel Herranz. *An Object-Oriented Formal Notation: Executable Specifications in Clay*. PhD thesis, Universidad Politécnica de Madrid, January 2011.
11 Ángel Herranz and Julio Mariño. Executable specifications in an object oriented formal notation. In *LOPSTR 2010*, July 2010.
12 D. Jackson and J. Wing. Lightweight formal methods. *IEEE Computer*, April 1996.
13 Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, 2006.
14 Ralf Jungclaus, Gunter Saake, Thorsten Hartmann, and Cristina Sernadas. TROLL a language for object-oriented specification of information systems. *ACM Trans. Inf. Syst.*, 14(2):175–211, 1996.
15 Michael Leuschel and Michael Butler. ProB: A model checker for B. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *FME 2003: Formal Methods*, LNCS 2805, pages 855–874. Springer-Verlag, 2003.
16 Michael Leuschel, Dominique Cansell, and Michael Butler. Validating and animating higher-order recursive functions in B. In Jean-Raymond Abrial and Uwe Glässer, editors, *Festschrift for Egon Börger*, 2007.
17 John W. Lloyd and Rodney W. Topor. Making Prolog more expressive. *J. Log. Program.*, 1(3):225–240, 1984.
18 John Wylie Lloyd. *Foundations of Logic Programming*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1993.
19 Oscar Pastor Lopez, Fiona Hayes, and Stephen Bear. *OASIS: An Object-Oriented Specification Language.*, volume 593 of *LNCS*, pages 348–363. Springer, January 1992.
20 Ulf Norell. Dependently typed programming in Agda. In *TLDI '09: Proceedings of the 4th international workshop on Types in language design and implementation*. ACM, 2009.
21 Nicolas Oury and Wouter Swierstra. The power of Pi. *SIGPLAN Not.*, 43(9):39–50, 2008.
22 Prover9 and Mace4 Website. `http://www.cs.unm.edu/%7emccune/mace4`.
23 Graeme Smith. *The Object-Z specification language*. Kluwer Academic Publishers, 2000.

An Inductive Approach for Modal Transition System Refinement

Dalal Alrajeh[1], Jeff Kramer[1], Alessandra Russo[1], and Sebastian Uchitel[1,2]

1 **Department of Computing, Imperial College London**
180 Queens Gate, London SW7 2AZ, UK
`{dalal.alrajeh04,j.kramer,a.ruso,s.uchitel}@imperial.ac.uk`

2 **Departamento de Computaciòn, FCEyN, UBA,**
Buenos Aires, Argentina
`suchitel@dc.uba.ar`

Abstract

Modal Transition Systems (MTSs) provide an appropriate framework for modelling software behaviour when only a partial specification is available. A key characteristic of an MTS is that it explicitly models events that a system is required to provide and is proscribed from exhibiting, and those for which no specification is available, called *maybe* events. Incremental elaboration of maybe events into either required or proscribed events can be seen as a process of MTS refinement, resulting from extending a given partial specification with more information about the system behaviour. This paper focuses on providing automated support for computing strong refinements of an MTS with respect to event traces that describe required and proscribed behaviours using a non-monotonic inductive logic programming technique. A real case study is used to illustrate the practical application of the approach.

1998 ACM Subject Classification F.4.1 Mathematical Logic

Keywords and phrases Modal transition system, refinement, event calculus, inductive logic programming

Digital Object Identifier 10.4230/LIPIcs.ICLP.2011.106

1 Introduction

A Modal Transition System (MTS) is a state-transition formalism for specifying and verifying system behaviour. It extends conventional models such as Labelled Transition Systems (LTSs) by introducing modalities over transitions. Hence, an MTS not only models the events that a system is required to provide and is proscribed from exhibiting, but also explicitly models the events which system being modelled cannot guarantee to admit or prohibit, called *maybe* events.

Though MTSs have been introduced for over twenty years [9], it is only recently that the software engineering community has begun to develop automated support for stepwise elaboration of system requirements through MTSs. Intuitively, an MTS can be seen as a class of possible system implementations, each generated by changing maybe transitions into either required or proscribed. Within this context, a key notion is that of *modal refinement* [8]. Modal refinement is the process of incrementally refining an MTS, as more information about the system becomes available, by modifying possible behaviours into behaviours that *must* be provided or prevented by every system implementation of the given MTS. A final refined MTS would therefore be an LTS with just required events, where all unspecified

© Dalal Alrajeh, Jeff Kramer, Alessandra Russo, and Sebastian Uchitel;
licensed under Creative Commons License NC-ND
Technical Communications of the 27th International Conference on Logic Programming (ICLP'11).
Editors: John P. Gallagher, Michael Gelfond; pp. 106–116

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

events are assumed to be proscribed. We refer to this refined model as an *implementation*. Much work has been done on theoretical aspects of modal refinement and different notions of refinements have been presented (i.e. strong, weak and branching refinement[4]). However, how to compute these different refinements remains still an open problem.

The aim of our work is to provide a formal, tool-supported platform for incremental refinement of MTSs. This paper presents a first step towards such a general framework, in which it is shown how inductive learning can be used to compute *strong* refinements of MTSs from event traces. We consider a partial system description, consisting of a specific class of safety properties expressed in temporal logic, and assume that traces and the transitions to be required or proscribed are either provided by the user or automatically computed by model checking a given MTS with respect to some property. This description is encoded into an Event Calculus (EC) logic program, whose language is close to existing logic formalisms used in MTS synthesis and verification, and which allows explicit representation of event occurrences at different (time) points, through its time structure. We deploy a non-monotonic Inductive Logic Programming (ILP) system to generate new safety properties from event traces that would either require and proscribe transitions, and show how the learned properties characterise a *class* of implementations of the original MTS with respect to the given traces. The proposed approach is much influenced by our recent work on the elaboration of software requirements through ILP [1, 2] where we have shown how non-monotonic learning can be used to compute an implementation (i.e. an LTS) that satisfy a given set of properties from scenarios. In this paper, we show how inductive learning can be used to computes classes of implementations.

The paper is organised as follows. Section 2 describes background work on MTSs and the specification language used to construct and verify MTSs. Section 3 describes the proposed methodology. Section 4 presents a case study used to evaluate the our refinement process and Sections 5 and 6 conclude the paper with a discussion of related and future work.

2 Background

A Modal Transition System is a formalism used for modelling and reasoning about the behaviour of a system. A formal definition of an MTS is given below.

▶ **Definition 1** (Modal Transition System). A Modal Transition System is a tuple $M = (S, Act, \Delta^r, \Delta^p, s_0)$ where Act is the alphabet of label events, S a set of state, $\Delta^r \subseteq S \times Act \times S$ the set of *required* transitions, and $\Delta^p \subseteq S \times Act \times S$ the set of *possible* transitions, such that $\Delta^r \subseteq \Delta^p$. Transitions that are possible but not required are called *maybe* transitions. An MTS is said to have a required transition on a, denoted $s \xrightarrow{a}_r s'$, if $(s, a, s') \in \Delta^r$. It is said to have a possible transition on a, denoted $s \xrightarrow{a}_p s'$, if $(s, a, s') \in \Delta^p$.

An implementation is an MTS $(S, Act, \Delta^r, \Delta^p, s_0)$ where $\Delta^r = \Delta^p$, also referred to as an LTS. An MTS can be represented as a directed graph in which nodes correspond to states and the edges between two nodes represent the transition relation. Maybe transitions are denoted with a question mark following the event label. System executions are represented as sequences of transitions called *traces*. A trace can be either required, possible or proscribed. A formal definition is given below. The definition of a trace presupposes that only one event can occur at a single position in a trace, i.e. events cannot occur simultaneously.

▶ **Definition 2** (Traces). Let $M = (S, Act, \Delta^r, \Delta^p, s_0)$ be an MTS. A trace $\sigma = a_1, a_2, ...$ where $a_i \in Act$ is said to be a *required trace in* M if there exists in M an infinite sequence of transitions $s_0 \xrightarrow{a_1}_r s_1, s_1 \xrightarrow{a_2}_r s_2 ...$; it is said to be a *possible trace in* M if there exists in M

an infinite sequence of transitions $s_0 \xrightarrow{a_1}_p s_1, s_1 \xrightarrow{a_2}_p s_2...$, where $(s_i, a_{i+1}, s_{i+1}) \in \Delta^p$ for each $i \geq 0$, and there is at least one transition relation in the sequence that is in $\Delta^p - \Delta^r$. Otherwise, it is said to be a *proscribed trace in M.*

In this paper, we refer to traces that should be possible in a model as *positive* traces, and to traces that should be proscribed as *negative* traces. The notion of *modal refinement* is defined between two MTSs and states when one MTS is "more defined" than another. Given two MTSs N and M, N is said to *refine M* if N preserves all the required and all the proscribed transitions of M. In this paper we assume a particular notion of refinement relation, called *strong refinement*, which assumes that all MTSs share the same *alphabet* [9]. This is defined as follows.

▶ **Definition 3** (Strong Refinement)**.** Let $\wp$ be the universe of all MTSs for a given alphabet *Act*. An MTS $N = (U, Act, \delta^r, \delta^p, u_0)$ is a refinement of an MTS $M = (S, Act, \Delta^r, \Delta^p, s_0)$, written as $M \preceq N$, if there exists some refinement relation $\mathcal{R} \subseteq S \times U$ such that, for all $s \in S$ and $u \in U$, if $(s, u) \in \mathcal{R}$, the following holds for every label a in *Act*:

- if $(s \xrightarrow{a}_r s')$, then for some $u' \in U$, $(u \xrightarrow{a}_r u' \;\wedge\; (s', u') \in \mathcal{R})$; and
- if $(u \xrightarrow{a}_p u')$, then for some $s' \in S$, $(s \xrightarrow{a}_p s' \;\wedge\; (s', u') \in \mathcal{R})$.

An MTS can be synthesised from and verified against formulae expressed in some form of temporal logic. We give here a brief description of FLTL[7] as the language used by the MTSA model checker [3]. In FLTL, a fluent f is defined by a tuple consisting of a set I_f of initiating events, a set T_f of terminating events and an initial truth value (**tt** or **ff**), such that $I_f \subseteq Act$, $T_f \subseteq Act$ and $I_f \cap T_f = \emptyset$. We write $f = \langle I_f, T_f, Init\rangle$ as a shorthand for a fluent definition, where $Init \in \{\mathbf{tt}, \mathbf{ff}\}$. Every event label $a \in Act$ defines a fluent $\dot{a} = \langle a, Act \backslash \{a\}, \mathbf{ff}\rangle$. We refer to such fluents as event fluents.

Given the set of fluents F, FLTL formulae can be constructed using the standard boolean connectives and temporal operators

$$\mathbf{tt} \mid \mathbf{ff} \mid f \mid \neg\phi \mid \phi \vee \psi \mid \phi \wedge \psi \mid \mathsf{X}\phi \mid \phi \mathsf{U} \psi \mid \mathsf{F}\phi \mid \mathsf{G}\phi$$

Given a set of traces Σ over *Act* and a set D of fluent definitions, a fluent is said to be true in a given trace $\sigma = \langle a_1, ..., a_n\rangle$ at position i with respect to D, denoted $\sigma, i \models_D f$, if and only if either of the following conditions hold:

- f is defined initially true and $\forall j \in \mathcal{N}.\ ((0 < j \leq i) \rightarrow a_j \notin T_f)$;
- $(\exists j \in \mathcal{N}.\ (j \leq i) \wedge (a_j \in I_f)) \wedge (\forall k \in \mathcal{N}.((j < k \leq i) \rightarrow a_k \notin T_f))$.

In other words, a fluent f holds if and only if it is initially true or an initiating event for f has occurred, and no terminating event has occurred since. The semantics of boolean connectives are defined in the standard way. The semantics of the temporal operators are defined as follows:

$\sigma, i \models_D \mathsf{X}\phi$, *if and only if* $\sigma, i+1 \models_D \phi$
$\sigma, i \models_D \phi\mathsf{U}\psi$, *if and only if* $\exists j \geq i.\ \sigma, j \models_D \psi$ *and* $\forall i \leq k < j.\ \sigma, k \models_D \phi$
$\sigma, i \models_D \mathsf{G}\phi$, *if and only if* $\forall j \geq i.\ \sigma, j \models_D \phi$
$\sigma, i \models_D \mathsf{F}\phi$, *if and only if* $\exists j \geq i.\ \sigma, j \models_D \phi$

Given that MTSs represent a set of possible implementations, it is often necessary to reason about properties that hold in some or all implementations or none. Thus the satisfaction of FLTL formulae over MTSs is given a 3-valued semantics. An MTS M is said to

satisfy a property ϕ with respect to D, if ϕ is satisfied in every possible trace of M with respect to D. It is said to be violated in M, if there is a required trace in M that refutes it or if all possible traces in M violate it. Otherwise, the satisfaction of ϕ is *unknown*, meaning that some implementations satisfy ϕ while others do not. Two FLTL formulae ϕ and ψ are consistent if there exists a model M that satisfies the formula $\phi \wedge \psi$. For the remainder of this paper, we use $\wp(\phi)$ to denote an MTS that satisfies ϕ with respect to D. Furthermore, we consider only safety properties of the form $(\mathsf{G}\ (\bigwedge_{0\leq i} f_i \rightarrow \mathsf{X}(\neg)\dot{a}))$ where f_i is a literal over event or non-event fluent, and a is an event label. We also assume that the MTS synthesised from given safety properties is the least refined MTS that satisfies the given properties (for further detail see [13]).

3 Approach

The aim of our work is to develop an automated approach for refining MTSs, given a set of traces that represent required and proscribed system behaviour. In this paper we focus on the notion of strong refinement given in Definition 3 and on FLTL safety properties of the form $(\mathsf{G}\ (\bigwedge_{0\leq i} f_i \rightarrow \mathsf{X}(\neg)\dot{a}))$ described above. The input to our approach is a set D of fluent definitions, a set $\Gamma = \{\gamma_i\}$ of safety properties and two disjoint sets of positive and negative traces, $\Sigma = \Sigma^+ \cup \Sigma^-$ that are possible traces in the MTS M synthesised from Γ. The input D, Γ and Σ are encoded into an Event Calculus (EC) logic program, and a non-monotonic ILP system is used to learn rules about required and proscribe transitions that can be translated back into FLTL safety properties Φ satisfying the following property: the MTS N synthesised from the (refined) property $\bigwedge \gamma_i \wedge \Phi$ is a strong refinement of the given MTS M where every trace in Σ^+ is a possible trace in N and every trace in Σ^- is a proscribed trace in N.

▶ **Definition 4** (Refinement with respect to traces). Let $M = \langle S, Act, \Delta^r, \Delta^p, s_0\rangle$ and let $\Sigma = \Sigma^+ \cup \Sigma^-$ be a set of positive and negative traces that are possible in M. An MTS $N = \langle U, Act, \delta^r, \delta^p, u_0\rangle$ is a *correct refinement of* M *with respect to* Σ if and only if N is a refinement of M ($M \preceq N$) and every trace $\sigma^+ \in \Sigma^+$ is a possible trace in N, and every trace $\sigma^- \in \Sigma^-$ is a proscribed trace in N.

▶ **Definition 5** (Refinement Task). Let $\Gamma = \{\gamma_1, ..., \gamma_n\}$ be a set of FLTL safety properties, let D a set of fluent definitions, let $M = \wp(\bigwedge \gamma_i)$ be an MTS that satisfies Γ w.r.t D and $\Sigma = \Sigma^+ \cup \Sigma^-$ a set of positive and negative traces that are possible in M. The refinement task is to find an FLTL safety property Φ such that the MTS $N = \wp(\bigwedge \gamma_i \wedge \Phi)$ is a correct refinement of M with respect to the traces in Σ. We call Φ a *consistent extension* of Γ.

3.1 An Event Calculus for MTSs

The Event Calculus is a widely-used logic programming formalism for reasoning about actions and time [12]. The definition of an EC language in this paper includes terms of four different types: *event* terms, *fluent* terms, *time* (here referred as position) terms and *trace* terms. Position terms are represented by the non-negative integers $0, 1, 2, \ldots$, events correspond to actions that can be performed, fluents correspond to time-varying Boolean properties, and traces are constants denoting different (independent) time lines.

The EC ontology includes the basic predicates *happens*, *initiates*, *terminates*, *initially* and *holdsAt*. The atom *happens*(e, p, c) indicates that event e occurs at position p in a trace c, *initiates*(e, f) (resp. *terminates*(e, f)) means that, if event e were to occur, it would cause fluent f to be true (resp. false) immediately afterwards. The predicate *holdsAt*(f, p, c)

indicates that fluent f is true at position p in a trace c, and *initially*(f) means that fluent f is initially true. The formalism also includes an auxiliary predicate *clipped*(p_1, f, p_2, c) which is defined as an event that terminates f occurs between positions p_1 and p_2 in a trace c. The interactions between these EC predicates are governed by the set of domain-independent core axioms defined below, where *not* denotes negation by failure.

```
clipped(P1,F,P2,C):-  happens(E, P, C), terminates(E, F), P1 ≤ P<P2.
   holdsAt(F,P2,C):- happens(E,P1,C), initiates(E,F),                    (1)
                     P1 < P2, not clipped(P1,F,P2,C).
   holdsAt(F,P,C) :- initially(F), not clipped(0,F,P,C).
```

To capture in EC the different types of MTS transitions we have extended the language with the predicates *required*, *proscribed*, and *maybe*. The atom *required*(e, p, c) (resp. *proscribed*(e, p, c)) means that event e's occurrence is required (resp. proscribed) at position p in a given trace c. The atom *maybe*(e, p, c) is defined in (2) below and means that the occurrence of event e at position p in trace c has not yet been specified.

```
maybe(E,P,C):- not required(E,P,C), not proscribed(E,P,C).               (2)
```

Auxiliary predicates are also introduced to refine and appropriately constraint the notion of occurrence of event: *req_happens*(e, p, c) is used to capture the fact that all required events must happen, and *may_happens*(e, p, c) defines that a maybe event may happen at some position in a trace if executed at that position in that trace. They are related to the *happen* predicate by the following axioms:

```
 req_happens(E,P,C):- required(E,P,C).                                   (3)
may_happens(E,P,C):-  executed(E,P,C), maybe(E,P,C).                     (4)
    happens(E,P,C):-  may_happens(E,P,C).                                (5)
    happens(E,P,C):- req_happens(E,P,C).                                 (6)
```

Integrity constraints over these predicates are captured by the following denial rules[1]:

```
false:- required(E,P,C), proscribed(E,P,C).                              (7)
false:- required(E,P,C), not executed(E,P,C).                            (8)
false:- happens(E1,P,C), happens(E2,P,C), not eq(E1,E2).                 (9)
```

Constraint (7) states that an event cannot be required and proscribed at the same position in a trace, whereas constraint(8) states that a required event must be executed at the position where it is required. A weaker semantics for required transitions is discussed in Section 5.

In addition to the above domain-independent axioms, our EC program is equipped with *domain-dependent axioms* that capture properties of the given partial system description. The following definition shows how FLTL partial specifications are encoded into domain-dependent EC axioms.

▶ **Definition 6** (EC Encoding). Let $\Gamma = \{\gamma_1, ..., \gamma_n\}$ be a set of safety properties, D a set of fluent definitions and Σ a set of finite traces, such that every trace in Σ is a possible trace in MTS $M = \wp(\bigwedge \gamma_i)$. The EC encoding of Γ and D, denoted $EC(\Gamma \cup D \cup \Sigma)$, is the EC program Π constructed as follows:

[1] Denial rules take the form `false :- (not)b_1, ..., (not)b_n` where *b_i* can be any atom defined in the language.

- add to Π, for each fluent definition $f = \langle \{a_i\}, \{b_i\}, Init \rangle$ in D,
  - the set of facts $initiates(a_i, f)$ and $terminates(b_i, f)$,
  - the fact $initially(f)$ if f is defined in D as initially true,
- add to Π, for each safety property of the form $\Box(\bigwedge_{0 \leq i \leq k} (\neg) f_i \rightarrow \bigcirc \neg \dot{a})$, the rule $proscribed(a, P, C)$:-(not) $holdsAt(f_1, P, C), ..., (not)$ $holdsAt(f_k, P, C)$[2],
- add to Π, for each safety property of the form $\Box(\bigwedge_{0 \leq i \leq k} (\neg) f_i \rightarrow \bigcirc \dot{a})$, the rule $required(a, P, C)$:-(not) $holdsAt(f_1, P, C), ..., (not)$ $holdsAt(f_k, P, C)$,
- add to Π, for each trace $\sigma_j = \langle a_1, ..., a_n \rangle \in \Sigma$, the set of facts $executed(a_i, i-1, c_j)$, where $0 < i \leq n$ and c_j denotes the trace σ_j.

The EC program described above is a *normal logic program* for which we use the notions of stable model semantics and entailment under credulous stable model semantics [6]. Hence we say that an EC program Π *entails* an expression π, denoted $\Pi \models \pi$, if and only if π is satisfied in at least one stable model of Π. The following theorem proves that the above EC encoding is sound.

▶ **Theorem 7** (Soundness of EC Encoding). *Let Γ be a set of safety properties, D be a set of fluent definitions and $M = (S, Act, \Delta^r, \Delta^p, s_0)$ an MTS that satisfies Γ with respect to D. Let $\sigma = \langle a_1, ..., a_n \rangle$ be a finite possible trace in M. Let Π be the EC logic program given by $EC(\Gamma \cup D \cup \sigma)$ and let I be a stable model of Π. Then, for each fluent f in D and position p in σ, where $0 \leq p \leq n$, f is true at position p in σ if and only if $holdsAt(f, p, \sigma) \in I$; for every event $a \in Act$ and position p in the trace σ, where $0 \leq p \leq n$, there is a required transition on a at position p in M if and only if $req_happens(a, p-1, \sigma) \in I$ and there is a maybe transition on a at p in M if and only if $may_happens(a, p-1, \sigma) \in I$.*

The proof is by induction of the position p in the trace σ, using the fact that Π is a locally stratified program and, as such, has a unique stable model.

3.2 Refining MTS using Inductive Logic Programming

Inductive Logic Programming (ILP) is concerned with the computation of hypotheses H that extend a prior background theory B to entail a set of examples E, i.e. $B \cup H \models E$ [10]. The hypotheses H are assumed to be part of a set of clauses HS, called the *hypothesis space*, which defines all hypotheses that would be accepted as a solution.

▶ **Definition 8** (Inductive Solution). Given a normal logic program B, a set of ground literals E, and a set of clauses HS, the task of ILP is to find a normal logic program $H \subseteq HS$, consistent with B, such that $B \cup H \models E$ under the stable model semantics. H is called an *inductive solution* for E with respect to B and HS.

In our refinement approach, the inductive learning task is to compute hypotheses H from a background B, given by the EC encoding of a partial system specifications, that is consistent with the given specification, and that entails, together with B, require and proscribe event transitions specified in a given set of traces. The (possible) traces in a given Σ are therefore translated into facts about what is required to happen and what should be proscribed from happening in the refined model. These facts constitute the set E of examples for our inductive learning task.

[2] (not) preceding a literal is a shorthand for either the positive or negative form of that literal.

▶ **Definition 9** (EC encoding of event traces into examples)**.** Let $\Gamma = \{\gamma_1, ..., \gamma_n\}$ be a set of safety properties, D a set of fluent definitions and $\Sigma^+ \cup \Sigma^-$ a set of finite positive and negative traces, such that every trace in Σ is a possible trace in MTS $M = \wp(\bigwedge \gamma_i)$. The EC translation of traces in Σ into examples E, denoted EC(Σ), is constructed as follows:

- for each trace $\sigma_j^+ = \langle a_1, ..., a_m \rangle \in \Sigma^+$ where $s_0 \xrightarrow{a_1}_p s_1, ..., s_{m-1} \xrightarrow{a_m}_p s_m$ in M
  - add to E facts $req_happens(a_i, i-1, c_j)$ for every transition $s_{i-1} \xrightarrow{a_i}_p s_i$ that should be required, where $1 \leq i \leq m$,
  - add to E facts $happens(a_i, i-1, c_j)$ for all other transitions, where $1 \leq i < m$,
- for each trace $\sigma_k^- = \langle b_1, ..., b_n \rangle \in \Sigma^-$, where $s_0 \xrightarrow{b_1}_p s_1, ..., s_{n-1} \xrightarrow{b_n}_p s_n$ in M
  - add to E the fact *not* $happens(b_n, n-1, c_k)$.

So transitions in the traces of Σ^+ that are intended to be required in the refined model are formalised as *req_happens* atoms, and transitions in the traces of Σ^- that are intended to be proscribed are encoded with *not happens* literals. All other transitions that are not defined as *required* or *proscribed*, are represented as *happens* atoms.

To learn safety properties rules, we defined the HS to be the set of clauses with *proscribed* or *required* in the head and *holdsAt* literals in the body. Because the examples are given in terms of *req_happens* and (*not*) *happens*, and given our EC background knowledge, our learning task requires an ILP algorithm capable of non-observational predicate and non-monotonic learning. For this we have used the ILP system in [11].

In brief, for every transition that is assumed to be executed, and for which the background knowldge B does not include any proscribed rule, the consequence that it may happen can be derived from the background knowledge. This is inconsistent with the examples *req_happens* or *not happens* given in E for the same event. So the learning algorithm explains the example (consistently) by abducing a set A of ground required and proscribed facts for these transitions. These then are used to form the head of the learned rules. Body literals are grounds *holdsAt* literals derived from $B \cup A$ at the same positions of the traces of the adbuced literals. These constructed ground rules are then generalised in a way that preserves consistency with the integrity constraints included in the background knowledge. The learning may produce alternative solutions. Once a solution H is chosen, this is translated back into FLTL. Rules of the form $proscribed(a, P, C)$:-(*not*) $holdsAt(f_1, P, C), ...,$ (*not*) $holdsAt(f_k, P, C)$ are translated back into safety properties of the form $\Box(\bigwedge_{0 \leq i \leq k} (\neg) f_i \rightarrow \bigcirc \neg \dot{a})$, while rules of the form $required(a, P, C)$:-(*not*) $holdsAt(f_1, P, C), ...,$ (*not*) $holdsAt(f_k, P, C)$ are translated back into FLTL properties of the form $\Box(\bigwedge_{0 \leq i \leq k} (\neg) f_i \rightarrow \bigcirc \dot{a})$. The soundness of the learning step is proved by Lemma 10 and Theorem 11.

▶ **Lemma 10.** *Let $\Gamma = \{\gamma_1, ..., \gamma_n\}$ be a set of properties expressed in FLTL, D a set of fluent definitions and $\Sigma = \Sigma^+ \cup \Sigma^-$ a set of traces, such that every trace in Σ is a possible trace in the MTS $M = \wp(\bigwedge \gamma_i)$. Let $\Pi = EC(\Gamma \cup D \cup \Sigma)$ be the EC translation of Σ, D and Γ, and let $E = EC(\Sigma)$ be the encoding of Σ into examples. Let H be an inductive solution for E with respect to Π, within the hypothesis space HS. Then FLTL(H) is consistent with the given specification Γ.*

The proof is by contradiction were Γ and FLTL(H) are assumed to be inconsistent and show that it falsifies the assumption of H be an inductive solution.

▶ **Theorem 11.** *Let $\Gamma = \{\gamma_1, ..., \gamma_n\}$ be a set of safety properties expressed in FLTL, D a set of fluent definitions and $\Sigma = \Sigma^+ \cup \Sigma^-$ a set of traces, such that every trace in Σ is a possible trace in the MTS $M = \wp(\bigwedge \gamma_i)$. Let $\Pi = EC(\Gamma \cup D \cup \Sigma)$ be the EC translation of Σ, D and Γ, and let $E = EC(\Sigma)$ be the encoding of Σ into examples. Let H be an inductive solution*

for E with respect to Π*, under the hypothesis space HS. Then FLTL(H) is a consistent extension of* Γ *with respect to the traces in* Σ*.*

4 A Case Study

In this section we illustrate an application of our approach to a real case study, the Philips Television Set Configuration reported in [14], which describes an industrial protocol for a product family of Philips television sets. We used the MTSA tool described in [3] to construct an MTS from the available (partial) system description and to verify the resulting refined MTS.

The system comprises multiple tuners, a video output device and a switch that can be configured by the television user to display several signals in different configurations. The protocol is concerned with controlling the signal path to avoid visual artefacts appearing on video outputs when a tuner is changing frequency. The alphabet is composed of {*set_Active_t1*, *set_Active_t2*, *t1_tune*, *s_restore*, *t1_dropReq*, *s_dropAck_t1*}. The fluent definitions are given by the following tuples:

Active_t1 =<*set_Active_t1, set_Active_t2, tt*>
Tuning_t1 =<*t1_tune, {s_restore, set_Active_t1, set_Active_t2}, ff*>
WaitingDropAck_t1 =<*t1_dropReq, s_dropAck_t1, ff*>
Dropped_t1=<*s_dropAck_t1, t1_restore, ff*>

Figure 1 shows an MTS generated from the initial descriptions. Note that the numbered nodes are used for reference and do not designate a particular state in Δ^p. The positive and negative traces $\Sigma^+ \cup \Sigma^-$ include:

$$\Sigma^+ = \{\langle start, t1_tune\rangle\} \quad (10)$$

$$\Sigma^- = \{\langle start, t1_tune, t1_newValue, t1_tune\rangle, \langle start, t1_tune, t1_newValue, t1_dropReq, s_dropAck_t1, t1_restore, t1_tune\rangle\} \quad (11)$$

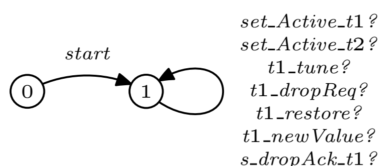

Figure 1 **An initial MTS** M **for the Philip TV set**

In the traces given in Σ the proscribed transitions are last occurrence of transition *t1_tune* in each trace in Σ^-, whereas the required transitions are the transition *t1_tune* in Σ^+. To learn safety properties that would ensure that only those MTS refinements that would meet these proscribed and required transitions are generated, we translate the given fluent definitions into EC domain-dependent axioms and the given traces in Σ into EC narratives and examples as defined in Section 3. Part of this translation is given below.

```
initiates(set_Active_t1,active_t1).     terminates(set_Active_t2,active_t1).
initially(active_t1).
initiates(t1_tune,tuning_t1).           terminates(s_restore,tuning_t1).
terminates(set_Active_t1,tuning_t1).    terminates(set_Active_t2,tuning_t1).
initiates(s_dropReqAck_t1,dropped_t1).     terminates(t1_restore,dropped_t1).
initiates(t1_dropReq,waitingDropAck_t1).terminates(s_dropAck_t1,waitingDropAck_t1).

executed(start,0,c1). executed(t1_tune,1,c1).
executed(start,0,c2). executed(t1_tune,1,c2).
executed(t1_newValue,2,c2).

    executed(t1_tune,3,c2). executed(start,0,c3).
    executed(t1_tune,1,c3). executed(t1_newValue,2,c3).
    executed(t1_dropReq,3,c3).executed(t1_tune,4,c3).

    examples:-
    happens(start,0,c1), req_happens(t1_tune,1,c1),
    not happens(t1_tune,3,c2), not happens(t1_tune,4,c3).
```

At the beginning, as no domain specific properties are included in the background knowledge, all event transitions are by default maybe transitions. Hence in the stable model of the EC program we have *maybe(e,p,c)* for every combination of event e, position p and trace c given in the language. Furthermore, the model contains *happens(e,p,c)* for those events where also *executed(e,p,c)* has been added to the narrative of the background knowledge. In other words, before the learning, the background knowledge entails that all executed events happen as maybe events. However, the examples state that last *t1_tune* executed in traces $c2$ and $c3$ should not occur and that the same event is required to happen in trace $c1$. So the given background knowledge does not entail the given examples. The ILP task then computes the following set of hypotheses:

```
 required(t1_tune,X1,X2):-holds_at(f_start,P,C).
proscribed(t1_tune,X1,X2):-holds_at(tuning,P,C).                    (12)
                           not holds_at(waitingDropAck_t1,P,C).
```

The above rules are translated back into the safety properties

$$\begin{array}{l} \mathsf{G}(start \rightarrow \mathsf{X}\ (t1_tune)) \\ \mathsf{G}((Tuning_t1 \wedge WaitingDropAck_t1) \rightarrow \mathsf{X}\ \neg(t1_tune)) \end{array} \tag{13}$$

The MTS generated from the conjunction of learned assertions in (13) is shown in Figure 2. It is easy to show that the refined model is a strong refinement of the initial model given in Figure 1 since every possible transition in the refined MTS is a possible transition in the initial MTS, and every required transition in M, in this case just the single transition *start* from the initial state, is also preserved in the refined model.

5 Discussion

In our present approach, we have focused on learning universal properties that force one required transition from states that satisfy the properties' conditions. The choice to adopt this semantics was inspired by existing work on synthesising MTSs that satisfy properties universally and existentially. Weaker semantics, whereby there can be several required

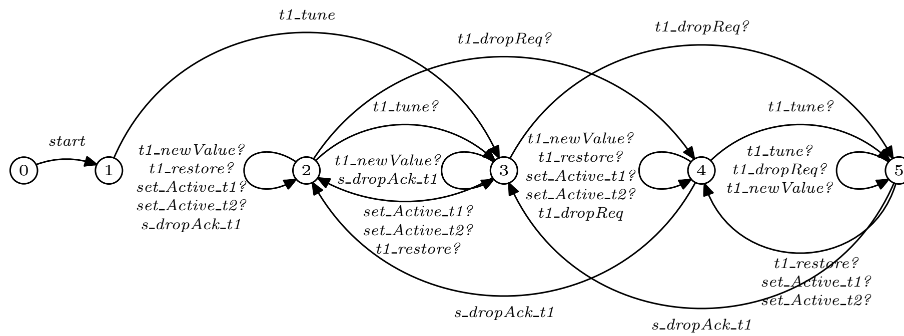

Figure 2 A refined MTS N for the Philip TV set

transitions from a single state to satisfy existential properties, can be modelled in our EC programs by adding an *executed* literal to the body of the definition of *req_happens* in (3) and dropping the constraint (8).

In defining our refinement, we have considered the MTSs that are generated from the conjunction of a given set of formulae instead of constructing an MTS for each formula and merging the single MTS models as proposed in [5]. Though only argued in [5] to be equivalent fro the case of safety properties and MTSs with the same alphabet, we have now verified through our refinement approach that this is indeed the case. We have also compared our work with techniques that synthesise MTS from scenarios directly [13] and found that such approaches do not support the use of negative scenarios which are crucial in our case to avoid over generalising the learned conditions.

In [1], we have presented an approach for detecting and resolving incompleteness in operational specifications using ILP. The this previous work the system models are Labelled Transition Systems which are less expressive than MTS as they are based on a completion assumption by which the system behaviour is strictly classified as either proscribed or required. The properties that were learned aimed at pruning undesirable traces from the initial Labelled Transition System. In this paper, the learning task is more general as it prunes traces where current possibly transitions should instead be proscribed, and also forces possible traces to be required. Both sets Δ_r and Δ_p of a given MTS are therefore consistently refined, whilst preserving the refinement relation between the MTS of the given description and that of the refined specifications.

6 Conclusion and Future Work

The overall aim of the work presented in this paper is to provide a formal and tool-supported approach for incremental software development through modal refinement by means of inductive learning. In brief, we have described the use of EC logic programs and ILP for computing strong refinements of MTSs from given event traces. We have shown how the EC language can sufficiently capture the different notions of the system models and how non-monotonic learning preserves the conditions of a strong refinement. We have argued that our computed hypothesis characterise a set of implementations of the original MTS.

As part of our future work, we intend to extend the approach to a wider class of safety properties, both as initial partial system descriptions and as learned properties, and to other forms of refinements such as weak refinement. This will enhance the applicability of our methodology to problems where the incremental refinement requires extending the given alphabet of events as new descriptions become available. We would also like to explore the

use of ILP to reason and refine non-deterministic MTSs, so fully exploiting the benefit of the EC representation.

Acknowledgments

We acknowledge financial support for this work from ERC project PBM - FIMBSE (No. 204853). Additionally, we would like to thank Rob Miller for initial discussions and Oliver Ray for providing a prototype implementation of XHAIL.

References

1. D. Alrajeh, J. Kramer, A. Russo, and S. Uchitel. Learning operational requirements from goal models. In *Proc. of 31st Intl. Conf. on Soft. Eng.*, pages 265–275, 2009.
2. D. Alrajeh, O. Ray, A. Russo, and S. Uchitel. Using abduction and induction for operational requirements elaboration. *J. of Applied Logic*, 7(3):275 – 288, 2009.
3. N. D'Ippolito, D. Fischbein, M. Chechik, and S. Uchitel. MTSA: The Modal Transition System Analyser. In *Proc. of 23rd Intl. Conf. on Auto. Soft. Eng.*, pages 475 –476, 2008.
4. D. Fischbein, V. Braberman, and S. Uchitel. A sound observational semantics for modal transition systems. In *Proc. of the 6th Intl. Colloquium on Theoretical Aspects of Comp.*, pages 215–230, 2009.
5. D. Fischbein and S. Uchitel. On correct and complete strong merging of partial behaviour models. In *Proc. of 16th Intl. Symp. on Foundations of Soft. Eng.*, pages 297–307, 2008.
6. M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In R.A. Kowalski and K. Bowen, editors, *Proc. of 5th Intl. Conf. on Logic Programming*, pages 1070–1080, 1988.
7. D. Giannakopoulou and J. Magee. Fluent model checking for event-based systems. In *Proc. 11th ACM SIGSOFT Symp. on Foundations Soft. Eng.*, pages 257 – 266, 2003.
8. K. Larsen, U. Nyman, and A. Wasowski. On modal refinement and consistency. In *Proc. of 18th Intl. Conf. on Concurrency Theory*, pages 105–119, 2007.
9. K.G. Larsen and B. Thomsen. A modal process logic. In *Proc. of 3rd Annual Symp. on Logic in Computer Science*, pages 203 –210, 1988.
10. S.H. Muggleton. Inverse Entailment and Progol. *New Generation Comp. , Special issue on Inductive Logic Programming*, 13(3-4):245–286, 1995.
11. O. Ray. Nonmonotonic abductive inductive learning. *J. of Applied Logic*, 7(3):329–340, 2009.
12. M. Shanahan. The event calculus explained. In *Artificial Intelligence Today: Recent Trends and Developments*, volume 1600 of *LNCS*, pages 409–430. 1999.
13. S. Uchitel, G. Brunet, and M. Chechik. Synthesis of partial behavior models from properties and scenarios. *Trans. on Soft. Eng.*, 35:384–406, 2009.
14. R. van Ommering, F. van der Linden, J. Kramer, and J. Magee. The koala component model for consumer electronics software. *Computer*, 33:78–85, 2000.

Constraints in Non-Boolean Contexts

Leslie De Koninck, Sebastian Brand, and Peter J. Stuckey

National ICT Australia, Victoria Research Laboratory
Dept. of Computer Science & Software Engineering, University of Melbourne
Australia
{lesliedk,sbrand,pjs}@csse.unimelb.edu.au

Abstract

In high-level constraint modelling languages, constraints can occur in non-Boolean contexts: implicitly, in the form of partial functions, or more explicitly, in the form of constraints on local variables in non-Boolean expressions. Specifications using these facilities are often more succinct. However, these specifications are typically executed on solvers that only support questions of the form of existentially quantified conjunctions of constraints.

We show how we can translate expressions with constraints appearing in non-Boolean contexts into conjunctions of ordinary constraints. The translation is clearly structured into constrained type elimination, local variable lifting and partial function elimination. We explain our approach in the context of the modelling language Zinc. An implementation of it is an integral part of our Zinc compiler.

1998 ACM Subject Classification F.4.1 Mathematical Logic: Logic and constraint programming; F.3.3 Semantics of Programming Languages; I.2.2 Automatic Programming: Program transformation

Keywords and phrases Constraint modelling languages, model transformation, partial functions

Digital Object Identifier 10.4230/LIPIcs.ICLP.2011.117

1 Introduction

In high-level constraint modelling languages such as Essence [3], OPL [13] and Zinc [8] constraints can occur in non-Boolean expressions: i.e. expressions whose value is not a Boolean, but for instance an integer. Most commonly, such constraints appear in the form of partial functions. For example "$y + 1\ \texttt{div}\ y = 2 \quad \vee \quad y \leqslant 0$" is a constraint involving an application of the partial function `div` (integer division), which is undefined if the divisor equals zero. The subexpression $1\ \texttt{div}\ y$ is an integer expression that implicitly introduces the constraint that y must be non-zero.

Several proposals for dealing with such constraints resulting from partiality are studied in [5]. In an imperative language, we may simply abort if y takes the value 0, but in a declarative language, this is unacceptable. The most popular approach for (constraint) modelling languages is the so-called *relational* semantics, which treats functions as "shorthand" for relations, since this is easy to support by solvers. In this semantics, the implicit constraint $y \neq 0$ is active at the nearest enclosing Boolean context. The above example thus means "$(y + 1\ \texttt{div}\ y = 2 \ \wedge \ y \neq 0) \quad \vee \quad y \leqslant 0$".

In Zinc, constraints in non-Boolean contexts can also arise in the form of a local variable declaration in a non-Boolean context if the type-inst[1] is constrained. Local variables are created using expressions of the form "`let` $\{T\colon\ x = A\}$ `in` E" where E is a (Boolean or

[1] A *type-inst* in Zinc is the combination of a type, such as `int`, and an instantiation pattern. See Section 2.

© Leslie De Koninck, Sebastian Brand, and Peter J. Stuckey;
licensed under Creative Commons License NC-ND
Technical Communications of the 27th International Conference on Logic Programming (ICLP'11).
Editors: John P. Gallagher, Michael Gelfond; pp. 117–127
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

non-Boolean) expression in which we introduce a new local variable called x whose type-inst is T and whose assignment (optional) is A. A formal semantics of these `let` expressions is given in Section 2.

Now consider the following constraint, imposing an ordering between two tuples

$(\texttt{u}, \texttt{v}) < (\texttt{y}, \texttt{let}\ \{\texttt{var}\ 0..4\text{: } \texttt{x} = 2 \cdot \texttt{y} + 1\}\ \texttt{in}\ \texttt{x} \cdot \texttt{x})$

where a local variable `x` is introduced to factor out a computation. It also implicitly introduces the constraint $0 \leqslant \texttt{x} \leqslant 4$ in the context of the non-Boolean expression $\texttt{x} \cdot \texttt{x}$. In the relational semantics this constraint is active at the level of the nearest enclosing Boolean expression, that is, at the level of the tuple comparison.

In the above example, we introduced a range type-inst (`var` 0..4), which is one of the forms of constrained type-ints in Zinc. The most general form is the arbitrarily constrained type-inst: (T: x `where` $C(x)$) with T a type-inst, x a name to refer to any value of T, and $C(x)$ a constraint that all values of the resulting constrained type-inst have to satisfy.

Constrained type-insts are useful to improve locality during modelling, but in particular they also frequently arise in the process of (automatic) type reduction: the mapping of complex structured types to basic types that can be handled by solvers. Consider the expression $\texttt{min}((x, 1), (y, 2))$ where x and y are integer variables, which represents the minimum of the tuples $(x, 1)$ and $(y, 2)$ (in lexicographic order). It may be mapped to

$\texttt{let}\ \{(\texttt{tuple}(\texttt{var int}, \texttt{var int})\text{: } \texttt{t}$
$\quad \texttt{where}\ (x \leqslant y \rightarrow \texttt{t} = (x, 1)) \wedge (x > y \rightarrow \texttt{t} = (y, 2)))\text{: } \texttt{m}\}\ \texttt{in m}$

so that it can be dealt with by solvers that do not support `min`/2 over tuples.

These examples illustrate how constraint models can involve (in some cases quite complex) constraints attached to non-Boolean expressions. To match the relational semantics of constraint modelling languages, and to eventually reach a constraint model that can be directly supported by underlying solvers, these constraints must be lifted to the nearest enclosing Boolean context. This is a non-trivial task since the enclosing non-Boolean operations can be deeply nested.

While Zinc has a direct way of expressing constraints in non-Boolean contexts using local variables with constrained type-insts, other constraint modelling languages such as $\mathcal{F}$ [7] and Essence [3] have to deal with such constraints as well during model transformations. In these languages, the model transformation rules need to keep track of constraints attached to non-Boolean expressions, and each rule needs to state how to deal with them appropriately.

In this paper, we propose a series of transformations to obtain the relational semantics for non-Boolean expressions that have constraints attached to them. Partial functions are a special case of this, and in that respect, this work generalises [5] by dealing with any sort of constraint in a non-Boolean context, and by decomposing the problem into partial function elimination, local variable lifting and constrained type elimination. While we focus on the Zinc language, this work is relevant to all constraint modelling languages and logic programming-based languages that allow partial function applications, and in particular also to functional-logic programming languages such as Curry [6]. Our transformations have the following significant features:

Zinc-to-Zinc: the result of the transformations is Zinc – no separate information outside a model needs to be maintained. This means these transformations can be followed by or combined with other Zinc transformations.

Data-independence: the transformations are independent of parameter values. Instance data need not be available at transformation time.

Locality: the transformations are only concerned with those parts of the model that contain non-Boolean constraints. Flattening of the model is not required.

2 Preliminaries

Zinc. A Zinc model consists of a set of items for decision variable and parameter declarations, constraints, solving objective and output. A variable or parameter declaration has the form "T: x" or "T: $x = A$", where x is the name of the variable, T is its type-inst and expression A is its optional assignment.[2] A *type-inst* is the combination of a type (a set of values) and an instantiation pattern, which determines which components of a variable are fixed when solving starts. Examples of type-insts are `int`, `var` 0..1, `tuple`(`bool`, `var float`) and `array`[`int`] `of var set of` 1..9, representing respectively an integer parameter, a binary variable, a tuple whose first and second field are a Boolean parameter and a float variable respectively, and an integer-indexed array whose elements are set variables taking elements from 1..9. The distinction between variables and parameters via the instantiation pattern in the type is crucial for data-independent transformation and compilation of Zinc models.

A constraint item holds a constraint. A solve item states whether the aim of solving is satisfaction, or optimisation in which case it also holds the objective function. An output item states how a solution to the problem should be presented. There are other types of items; see [8] for more details.

▶ **Example 1.** Here is a simple Zinc model:

```
int: c;                          % declare an integer parameter c
var set of 1..3: s;              % declare a set variable s, subset of {1,2,3}
constraint card(s) = c;          % enforce that the cardinality of s is c
solve satisfy;                   % search for any solution
output ["s = ", show(s)];        % output the resulting set

c = 2;
```

The last line defines the parameter `c`. It could be in a separate data file. ◀

In the following, we highlight some features of Zinc that are important for our discussion.

Zinc supports type-insts beyond the base ones (`int`, `float`, etc.), called *constrained* type-insts. One particularly expressive case of these are *arbitrarily constrained* type-insts. An example is (`int`: `i where i` > 0), denoting the positive integers.

Variables with a local scope can be introduced by `let` expressions. The scope, and thus the type-inst of the `let` expression itself, can be Boolean or non-Boolean. One can have multiple comma-separated variable declarations inside a single `let` expression.

Zinc allows the user to define their own functions and predicates (functions returning `var bool`). A function definition has the form "`function` T: $f(T_1\colon x_1, \ldots, T_n\colon x_n) = E$", where T is the return type-inst, f is the function name, T_i and x_i are respectively the type-insts and names of its arguments, and E is its body, which is an expression of type-inst T.

Semantics of Zinc. In this paper we are mainly concerned with constraints on non-Boolean expressions. In the introduction, we already gave an example of the intended meaning of an expression involving division. In the relational semantics, we consider Boolean expressions only, and in particular we decompose nested non-Boolean expressions into a conjunction of equality constraints. We denote the meaning of a Boolean expression E by $\mathcal{I}(E)$. Here is a (partial) definition of $\mathcal{I}$, given in priority order:

$\mathcal{I}(x = y) \;:=\; x = y$ where x and y are variables or constants;

[2] A parameter must be assigned, but the assignment can be in a separate data file.

$\mathcal{I}(x = B)$:= $x \leftrightarrow \mathcal{I}(B)$ with x a variable or constant and B a Boolean expression;

$\mathcal{I}(x = f(y_1, \ldots, y_n))$:= $\exists y'_1, \ldots, y'_n : \bigwedge_{i \in 1..n} \mathcal{I}(y'_i = y_i) \wedge x = f(y'_1, \ldots, y'_n)$ for any *total* non-Boolean function f/n where x is a variable or constant;

$\mathcal{I}(x = f(y_1, \ldots, y_n))$:= $\exists y'_1, \ldots, y'_n : \bigwedge_{i \in 1..n} \mathcal{I}(y'_i = y_i) \wedge p(y'_1, \ldots, y'_n, x)$ for any *partial* non-Boolean function f/n where x is a variable or constant and $p(z_1, \ldots, z_n, z_{n+1}) \leftrightarrow (f(z_1, \ldots, z_n) = z_{n+1})$;

$\mathcal{I}(x = \texttt{let}\ \{T : y = A\}\ \texttt{in}\ E)$:= $\exists y : \mathcal{I}_{ti}(T, y) \wedge \mathcal{I}(y = A) \wedge \mathcal{I}(x = E)$ with x a variable or constant and $\mathcal{I}_{ti}(T, y)$ the interpretation of y being of type-inst T;

$\mathcal{I}(p(x_1, \ldots, x_n))$:= $\exists x'_1, \ldots, x'_n : \bigwedge_{i \in 1..n} \mathcal{I}(x'_i = x_i) \wedge p(x'_1, \ldots, x'_n)$ for any predicate p/n, including operators such as $\wedge/2$, $\neg/1$, $=/2$ and $\leqslant/2$.

The interpretation for partial functions can also be used for total functions; however, the latter is more compact.

▶ **Example 2.** For the example from the introduction we find

$$\mathcal{I}(y + 1\ \texttt{div}\ y = 2\ \vee\ y \leqslant 0) = \\ (\exists i_1 : (\exists i_2 : \texttt{div}(1, y, i_2) \wedge i_1 = y + i_2) \wedge i_1 = 2)\ \vee\ y \leqslant 0$$

(after some simplification). For $y = 0$ it evaluates to `true`. ◀

From Zinc to solver. Our compiler for Zinc proceeds in two main stages. First, the high-level Zinc model is reduced to an equivalent model in a subset of Zinc called CoreZinc. For fixed expressions (expressions over parameters), CoreZinc is equivalent to Zinc. For expressions involving decision variables, CoreZinc is similar to MiniZinc [9]. The transformation from Zinc to CoreZinc is done independently of instance data by rewrite rules in the model transformation language Cadmium [1].

The resulting CoreZinc model is compiled into procedural code for the entire solution process, that is, to create variables and post constraints in the solvers, maintain communication between solvers, handle search, and generate output.

To handle constraints in non-Boolean context there are three interacting translations: elimination of constrained type-insts, lifting local variables, and elimination of unsafe partial function applications. They are described in the following three sections.

3 Elimination of Constrained Type-Insts

In Zinc, type-insts can be basic (e.g. `bool`, `int`, `float`) or constrained. Constrained type-insts have one of the following forms:

- ranges of the form "$l..u$" with l and u either (fixed) integer or float expressions;
- enumerated sets, e.g. $\{1, 3, 4\}$, or set parameters;
- arbitrarily constrained type-insts of the form (T: x `where` $C(x)$) with T a type-inst, x a name, and $C(x)$ a constraint;
- structured type-insts with constrained type-insts as components.

Solvers typically support range domains for their variables. The more complex type constraints, however, need to be converted into regular constraints.

A constraint in the type-inst of a global variable can immediately be extracted and posted as a regular constraint at the model root level. The interesting case is that of a local variable with constrained type-inst. In that case, the type constraints need to be lifted to

the nearest enclosing Boolean context. This is done by first lifting the variable declaration to this nearest Boolean context and then extracting the parts of the type constraint that can cause failure. The lifting to a Boolean context is described in Section 4. We show here how to extract the type constraint and add it to the Boolean context of the declaration.

The type-inst in the declaration of a local variable can cause failure in three ways:

(a) the type domain of the variable is empty;

(b) its assignment is not within its type-inst;

(c) its assignment fails.

We deal with case (a) by redeclaring the local variable with a guaranteed non-empty type-inst and adding a constraint stating that the original type-inst must be non-empty. For range type-insts, we do so by transforming "$\texttt{let}\ \{\texttt{var}\ l..u\texttt{:}\ x\}\ \texttt{in}\ B$" into "$\texttt{let}\ \{\texttt{var}\ l..\texttt{max}(l,u)\texttt{:}\ x\}\ \texttt{in}\ B\ \wedge\ l \leqslant u$". Here and in what follows, B is Boolean because we assume the `let` expression has already been lifted to the nearest Boolean context.

Local declarations with non-range set type-insts are dealt with as follows:

$\texttt{let}\ \{\texttt{var}\ s\texttt{:}\ x\}\ \texttt{in}\ B$

which states that x can take any value in s, is transformed into

$\texttt{let}\ \{\texttt{var}\ (\texttt{if}\ \texttt{card}(s) > 0\ \texttt{then}\ s\ \texttt{else}\ \{d\}\ \texttt{endif})\texttt{:}\ x\}\ \texttt{in}\ B \wedge \texttt{card}(s) > 0$

where d is an arbitrary value of the appropriate type, e.g. 0 for the integers.

For tuples and record types, we ensure each field has a non-empty type-inst. Set types are always non-empty, as the empty set is always a possible value. Array types are non-empty if their element type is non-empty. The index type can be empty, as in such case the empty array is a possible value.

For case (b), an assignment that might not be within the type-inst, we relax the type-inst to its base type-inst, which always includes the assigned value. The original type-inst is made explicit as an arbitrarily constrained type-inst and then extracted.

For range type-insts, "$\texttt{let}\ \{\texttt{var}\ l..u\texttt{:}\ x = A\}\ \texttt{in}\ B$" is transformed into "$\texttt{let}\ \{\texttt{var int:}\ x = A\}\ \texttt{in}\ B\ \wedge\ l \leqslant A\ \wedge\ A \leqslant u$".

For set type-insts in general, "$\texttt{let}\ \{\texttt{var}\ s\texttt{:}\ x = A\}\ \texttt{in}\ B$" is transformed into "$\texttt{let}\ \{\texttt{var}\ T\texttt{:}\ x = A\}\ \texttt{in}\ B\ \wedge\ A\ \texttt{in}\ s$" where T is the (base) element type of s.

The final case (c), a (non-Boolean) assignment that fails, can only occur because of partial functions. We treat it in Section 5.

4 Lifting Local Variables

Constraints that appear in non-Boolean contexts, either implicitly for partial functions or explicitly, must be lifted to the nearest enclosing Boolean context. Similarly, in order to allow solving by a constraint solver that can only handle existentially quantified conjunctions of constraints, all local variables must be lifted to the top level.

Local variable lifting is the process of lifting local variable declarations through enclosing expressions, modifying the declarations and occurrences of the declared variables as needed. It is the key step in dealing with these kinds of requirements.

In this section we show how we can lift local variable definitions through the different Zinc constructs.

4.1 Base Case

The base case for lifting is simple. For non-Boolean expression E_i, introducing local variable x and appearing as an argument to function f: $f(E_1, \ldots,$ `let` $\{T\colon x\}$ `in` $E_i, \ldots, E_n)$ transforms into `let` $\{T\colon x\}$ `in` $f(E_1, \ldots, E_i, \ldots, E_n)$. We assume x does not occur in $E_1, \ldots, E_{i-1}, E_{i+1}, \ldots, E_n$, renaming it to a unique new name if necessary.

4.2 Boolean Contexts

For a local variable declaration in a Boolean context, we apply the transformations of Section 3 to extract potentially failing type constraints. We can then safely lift the local variable above the Boolean context in which it appears using the base case.

One complication is that Zinc does not allow unassigned local variables in negative Boolean contexts, such as the argument of `not` or the first argument of implication `->`. This is because existentially quantified variables become universally quantified by lifting through negation, and our target solvers do not implement universal quantification.

▶ **Example 3.** The expression "`not` (`let` $\{$`var float`: $x\}$ `in` B)" means that there is no float value x for which B holds. In other words, for all float values x, expression B is false. ◀

Original Zinc models with unassigned local variables in negative contexts will be rejected, hence all originally occurring `let` constructs can be lifted out. The failure extraction transformations of Section 3 never remove an assignment.

However, sometimes an unassigned local variable may in fact be constrained to a single value by a type constraint. For example, "`let` $\{(T\colon x$ `where` $x = A)\colon y\}$ `in` $E(y)$" is of course equivalent to "`let` $\{T\colon x = A\}$ `in` $E(x)$". Another example, that cannot be written equivalently as an assignment, was given in the introduction where we translated the minimum of two tuples into an appropriately constrained tuple variable. We allow such implicitly assigned variables to be lifted over negative contexts.

4.3 Structured Types

Constraints can occur within a component of a structured type, e.g. within a field of a tuple or within an element of an array. Following the relational semantics, this constraint holds for the whole structure, and so we have to be careful with evaluating structure access as shown in the following example.

▶ **Example 4.** Consider the constraint "(5, `let` $\{$`var` 1..10: `i` $= 0\}$ `in` `i`).1 = 5". Evaluating the tuple access would result in "5 = 5", which is trivially true. Alternatively, we could lift the declaration, giving us "`let` $\{$`var` 1..10: `i` $= 0\}$ `in` ((5, `i`).1 = 5)" which further evaluates to `false`. The correct answer is found by looking at the semantics (Section 2) and treating tuple construction and access as non-Boolean functions. We obtain $t.1 = 5 \wedge t = (5, t_2) \wedge (\exists i : i = 0 \wedge i \in 1..10 \wedge t_2 = i)$. The failure holds for the entire structure. ◀

One important consequence of this is that some seemingly reasonable simplifications are incorrect. For example, the tuple access $(a_1, \ldots, a_n).i$ cannot always be replaced by a_i.

4.4 Comprehensions

An array comprehension $[H(i) \mid i$ `in` G `where` $W(i)]$ generates an array of instances of H for each value in the array G satisfying condition W. Local variables can appear in comprehensions in three places: in the comprehension head H, in a generator expression G, and in the condition W. We now look at each of these in detail.

Comprehension head. The way a local variable declaration is lifted outside of a comprehension head depends on which of the following properties the declaration has:

(1) the type-inst is independent of the generator;
(2) the assignment (if present) is independent of the generator;
(3) the assignment (if present) is free of local unassigned variables.

Before describing the general approach, we give two special cases for which we can do better.

Special case 1: an assignment is present and all properties (1), (2) and (3) hold. We can lift the variable declaration outside as is, as the different instantiations of the declaration, one for each value of the generator, have the same value and type-inst. More formally, [`let` $\{T$: `x` $= A\}$ `in` $E(x)$ | g `in` G] is rewritten to `let` $\{T$: `x` $= A\}$ `in` [$E(x)$ | g `in` G].

▶ **Example 5.** To see that property (3) is indeed needed, consider

[`let` {`var int: x` = `y` + `let` {`var` 0..8: `z`} `in z`} `in a`[`x`] > 0 | `i in` 1..9]

which evaluates to an array of 9 Boolean variables. This would become

`let` {`var int: x` = `y` + `let` {`var` 0..8: `z`} `in z`} `in` [`a`[`x`] > 0 | `i in` 1..9].

However, this is not equivalent to the original version: it would force all 9 instantiations of `z` (and `x` by transition) to take the same value. ◀

Special case 2: property (1) holds, and if an assignment is present, it does not have property (2) or (3). In this case the instantiations of the local variable within the comprehension are different in general. We need one copy of it for each generator value. Therefore, we create an array of variables outside the comprehension and replace each occurrence of the original local variable by a lookup into this array. More concretely,
[`let` $\{T$: $x = A(g)\}$ `in` $E(x)$ | g `in` G] is rewritten to
`let` {array[G] of T: y = [$A(g)$ | g `in` G]} `in` [$E(y[g])$ | g `in` G]. The case without an assignment to x is similar: we simply omit the assignment to y.

Here, we use the generator as an index into the array. This is not always possible, i.e. when the generator values are either not fixed, or potentially contain duplicates. If necessary, we first transform the comprehension so that its generator ranges are sets. For example, [$E(x)$ | g `in` G] is rewritten to [$E(G[g'])$ | g' `in index_set`(G)].

General case: property (1) does not necessarily hold. Again, we create an array for separate declarations outside of the comprehension, but this time we need to generalise the type-inst of its element type to make it independent of the generator. This is done in two steps. We will illustrate them using the comprehension

[`let` $\{T(g)$: $x = A(g)\}$ `in` $E(x)$ | g `in` G].

First, we make those type constraints that depend on the generator explicit by using an arbitrarily constrained type-inst:

[`let` $\{(T'$: y `where` $C_T(y,g))$: $x = A(g)\}$ `in` $E(x)$ | g `in` G]

where T' is a supertype of T that does not depend on g. In the second step, we create a constrained type-inst for the resulting array in which each element has the appropriate generator-dependent type constraint:

`let` {(`array`[G] of T': w `where forall`(g `in` G)($C_T(w[g], g)$)): z =
  [$A(g)$ | g `in` G]} `in` [$E(z[g])$ | g `in` G]

▶ **Example 6.**

[`let` {`tuple`(`var` 1..`j`, `var i`..`j`): `x` = `f`(`i`)} `in g`(`x`) | `i in` 1..5]

is a more concrete example. Only the second component of the type-inst of `x` depends on a generator. It is first transformed to

```
[ let ({tuple(var 1..j, var int): y where i ⩽ y.2 ∧ y.2 ⩽ j): x = f(i)}
  in g(x) | i in 1..5 ].
```

Then the constrained type-inst is lifted out of the comprehension to give

```
let {(array[1..5] of tuple(var 1..j, var int): y where
      forall(i in 1..5)(i ⩽ y[i].2 ∧ y[i].2 ⩽ j)): x = [ f(i) | i in 1..5 ]}
in [ g(x[i]) | i in 1..5 ].
```

◀

Generator expression. A declaration in the generator expression can directly be lifted out of the comprehension since it is independent of the generator.

`where` condition. The `where` condition forms a Boolean context, and so the rules of Section 3 apply. A non-failing declaration in a `where` condition can be lifted outside the comprehension similar to declarations in the comprehension head.

Multiple generators. Zinc allows multiple generators for the same comprehension. The extension to handle these is straightforward: essentially the declaration is lifted to a multi-dimensional array declaration. The most complicated case is declarations in generator expressions. In general

$[\ E \mid g_1 \texttt{ in } G_1,\ \ldots,\ g_i \texttt{ in let } \{T\colon x = A\} \texttt{ in } G_i(x),\ \ldots,\ g_n \texttt{ in } G_n \texttt{ where } C\ ]$

can be rewritten into

$\texttt{let } \{\texttt{array}[T_1, \ldots, T_{i-1}] \texttt{ of } T\colon x =$
$\qquad [\ (g_1, \ldots, g_{i-1})\colon A \mid g_1 \texttt{ in } G_1, \ldots, g_{i-1} \texttt{ in } G_{i-1}\ ]\}$
$\texttt{in } [\ E \mid g_1 \texttt{ in } G_1, \ldots, g_i \texttt{ in } G_i(x[g_1, \ldots, g_{i-1}]), \ldots, g_n \texttt{ in } G_n \texttt{ where } C\ ]$

where $T_1 \ldots T_{i-1}$ are the element type-insts of the arrays $G_1, \ldots, G_{i-1}$.

5 Elimination of Unsafe Partial Function Applications

Zinc includes various built-in partial functions, such as division and modulo (not defined if the divisor equals zero), array lookup (not defined for index values outside of the index set of the array), or the minimum of a set (not defined if the set is empty). Furthermore, there are a number of partial real functions, such as trigonometric ones.

The treatment of a partial function application depends on whether it operates on a fixed value. If so, the application is left as is, and it is simply evaluated when required. Not translating fixed applications avoids the associated increase in expression size.

For non-fixed values, the partial function acts as a constraint that restricts the values to be within the domain of the function. These applications are transformed to make the constraints explicit. In the process, the partial function application is made *safe*: its argument is guaranteed to be within the function's domain.

Solvers generally do not support reified versions of constraints such as the `element` constraint, which models an array lookup with a variable as the index, or integer division. Moreover, such reified constraints are absent in low-level solver input languages, such as FlatZinc [9] or XCSP [10]. If we simply lift such constraints to the top level conjunction, we end up with the strict semantics rather than the relational one [5]. We must therefore eliminate partiality from models.

The basic idea behind the transformation is that we encode the potential failure of a partial function application by a local variable declaration with a constrained type-inst.

How to move it to the nearest Boolean context is discussed earlier in the paper. Furthermore, we ensure that the partial function application in constraint form never fails because it is applied using input values outside of its domain.

Integer division. As an example of a partial arithmetic function, we show how integer division is made safe. A propagator for division as a constraint excludes zero from the domain of the divisor. Let $x \, \texttt{div} \, y$ be a *potentially unsafe division.* That is, it is not a priori clear that y can never assume the value 0. A first attempt to transform this division is

$\texttt{let} \ \{(\texttt{var int}\colon\ w' \ \texttt{where} \ w' = y)\colon\ w\} \ \texttt{in} \ x \ \texttt{div} \ w.$

Constrained-type elimination will extract the type constraint $w' = y$ and add it to the nearest enclosing Boolean context. An issue here is that w is an unassigned local variable. If the original expression $x \, \texttt{div} \, y$ appears in a negative Boolean context, then we now have an unassigned local variable in a negative context, a situation we disallow. Therefore, we find the above formulation undesirable. We prefer

$\texttt{let} \ \{(\texttt{var int}\colon\ w' \ \texttt{where} \ w' = y)\colon\ w = y + \texttt{bool2int}(y = 0)\} \ \texttt{in} \ x \ \texttt{div} \ w.$

Again, constrained-type elimination will extract the type constraint and add it to the appropriate context. Moreover, w has an assignment which is guaranteed to be inside the domain of the partial function.

Looking at the interpretation of both expressions (see Section 2), we have that $\mathcal{I}(z = x \, \texttt{div} \, y)$, which reduces to $\texttt{div}(x, y, z)$, is equivalent to

$$\mathcal{I}(z = \texttt{let} \ \{(\texttt{var int} : w' \ \texttt{where} \ w' = y) : w = y + \texttt{bool2int}(y = 0)\} \ \texttt{in} \ x \, \texttt{div} \, w).$$

Assuming a *total* division operation, this reduces to

$$\exists w : w = y \wedge w = y + \texttt{bool2int}(y = 0) \wedge z = x \, \texttt{div} \, w$$

which for $y = 0$ simplifies to $\texttt{false}$ and for $y \neq 0$ to $z = x \, \texttt{div} \, y$. Moreover, we can safely remove zero from the domain of w without affecting the possibility of y being zero.

Alternative encodings are possible, for example

$\texttt{let} \ \{(\texttt{var int}\colon\ w' \ \texttt{where} \ w' = y)\colon\ w = [y,\, 1][\texttt{bool2int}(y = 0) + 1]\} \ \texttt{in} \ x \ \texttt{div} \ w.$

Which encoding is more suitable depends on which solver is being used.

General approach. In general, let $f/1$ be a unary partial function, $c/1$ a constraint that succeeds when its input is in the domain of $f/1$ and fails otherwise, T be the type-inst of the domain of $f/1$ and d a value within the domain of $f/1$. We can transform a call $f(x)$ into a safe partial function application as follows:

$\texttt{let} \ \{(\texttt{var} \ T\colon\ w' \ \texttt{where} \ w' = x)\colon\ w = [d, x][\texttt{bool2int}(c(x)) + 1]\} \ \texttt{in} \ f(w).$

We can consider n-ary functions as unary functions operating on n-ary tuples.

6 Conclusions

Local variables and constrained types are crucial for both high-level modelling and effective model transformation (e.g. type reduction or solver-specific constraint transformations). To solve Zinc models, we must have a way to transform away these constructs to obtain existentially quantified conjunctions of constraints. The transformations that do this in a data-independent way are challenging and form a core part of Zinc. Any declarative language that wishes to treat partial functions correctly and in a data-independent manner must address these issues.

We have presented three sets of transformation rules to deal with respectively constrained type-insts, local variables and partial functions in Zinc. They transform valid Zinc models to semantically equivalent ones that are free of arbitrarily constrained type-insts, local variable declarations and potentially failing partial function applications. Our transformations are data-independent and can be run concurrently. Previous work on Zinc [8] required flattening the model and was not data independent.

In practice, we run partial function elimination first. Local variable lifting and constrained type elimination take place concurrently. They are run multiple times during the model transformation process, because other transformations introduce local variables and constrained type-insts but may assume a model without them.

Related Work

Zinc belongs to the family of constraint modelling languages that also includes $\mathcal{F}$ [7] and its successor ESRA [2], s-COMMA [12], ESSENCE [3] and OPL [13].

$\mathcal{F}$ provides function variables. Both partial and total functions can be represented. It supports a function membership operation $\langle i, j \rangle \in F$ which is equivalent to $i \in dom(F) \wedge F(i) = j$. However, function application is only allowed for values within the domain of the function. $\mathcal{F}$ models are translated into a lower-level language. For some expressions, this translation requires the introduction of new variables and constraints. Since local variables and a way to encapsulate constraints in non-Boolean contexts are not part of the language, these new variables and constraints need to be lifted. It is not described how this is done.

The s-COMMA language is an object-oriented constraint modelling language. It appears to support division and variable index array lookups, but it is unclear how it deals with undefinedness in these operations.

ESSENCE is a specification language for CSPs and shares many features with Zinc. ESSENCE models are transformed into the lower-level language ESSENCE′ using transformation rules written in CONJURE [4]. In CONJURE, the result of rewriting an expression is a new expression that may be tagged with a set of constraints. Since ESSENCE lacks local variables and an encapsulation mechanism for constraints in a non-Boolean context, rules need to state explicitly what to do with constraints that result from refining subexpressions.

Frisch and Stuckey [5] study undefinedness in the constraint (logic) programming languages ECL^{i}PS^e, SWI-Prolog, SICStus Prolog, OPL and MiniZinc. They discuss three different formal semantics and show how to transform models to ensure they behave as desired. The transformations consist of first identifying the nearest Boolean context of every unsafe function application and creating a local variable alias for it, which is immediately lifted to the top level. Next, each of these Boolean contexts is made safe by replacing unsafe function applications by safe versions and adding the necessary constraints. The approach of [5] is not always applicable to Zinc, as it can be the case that the unsafe partial function application uses variables that are not defined at the level of the nearest Boolean context.

In the functional-logic programming language Curry [6], a program is a set of functions. A function in Curry can be nondeterministic and in particular it can be partial. Whenever a partial function is applied to a value outside of its domain, the function application fails, which is different from returning false. As a result, we have that for instance `not x` evaluates to `True` if `x` evaluates to `False` and vice versa, but fails if `x` fails.

Mercury [11] allows functions to have a solution or to fail. Unlike Curry, it allows reasoning about success and failure using conjunction, disjunction, negation, etc. Failure is automatically lifted to the nearest Boolean context. However, Mercury is only concerned with evaluating fixed expressions.

References

1 Gregory J. Duck, Leslie De Koninck, and Peter J. Stuckey. Cadmium: An implementation of ACD Term Rewriting. In María García de la Banda and Enrico Pontelli, editors, *24th International Conference on Logic Programming*, volume 5366 of *Lecture Notes in Computer Science*, pages 531–545. Springer, 2008.

2 Pierre Flener, Justin Pearson, and Magnus Ågren. Introducing ESRA, a relational language for modelling combinatorial problems. In Sandro Etalle, editor, *14th International Symposium on Logic Based Program Synthesis and Transformation*, volume 3018 of *Lecture Notes in Computer Science*, pages 214–232. Springer, 2004.

3 Alan M. Frisch, Warwick Harvey, Christopher Jefferson, Bernadette Martínez Hernández, and Ian Miguel. Essence: A constraint language for specifying combinatorial problems. *Constraints*, 13(3):268–306, 2008.

4 Alan M. Frisch, Christopher Jefferson, Bernadette Martínez Hernández, and Ian Miguel. The rules of constraint modelling. In Leslie Pack Kaelbling and Alessandro Saffiotti, editors, *19th International Joint Conference on Artificial Intelligence*, pages 109–116. Professional Book Center, 2005.

5 Alan M. Frisch and Peter J. Stuckey. The proper treatment of undefinedness in constraint languages. In Ian Gent, editor, *15th International Conference on Principles and Practice of Constraint Programming*, volume 5732 of *Lecture Notes in Computer Science*, pages 367–382. Springer, 2009.

6 Michael Hanus. Curry: a multi-paradigm declarative language. In François Bry, Burkhard Freitag, and Dietmar Seipel, editors, *12th Workshop on Logic Programming*, 1997.

7 Brahim Hnich. *Function variables for constraint programming*. PhD thesis, Uppsala University, 2003.

8 Kim Marriott et al. The design of the Zinc modelling language. *Constraints*, 13(3):229–267, 2008.

9 Nicholas Nethercote et al. MiniZinc: Towards a standard CP modelling language. In Christian Bessiere, editor, *13th International Conference on Principles and Practice of Constraint Programming*, volume 4741 of *Lecture Notes in Computer Science*, pages 529–543. Springer, 2007.

10 Olivier Roussel and Christophe Lecoutre. XML representation of constraint networks: Format XCSP 2.1. *Computing Research Repository (CoRR)*, abs/0902.2362, 2009.

11 Zoltan Somogyi, Fergus Henderson, and Thomas Conway. The execution algorithm of Mercury: an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29(1-3):17–64, 1996.

12 Ricardo Soto. *Languages and Model Tranformation in Constraint Programming*. PhD thesis, CNRS, LINA, Université de Nantes, France, 2009.

13 Pascal Van Hentenryck, Irvin Lustig, Laurent Michel, and Jean-François Puget. *The OPL Optimization Programming Language*. MIT Press, 1999.

Minimizing the overheads of dependent AND-parallelism

Peter Wang[1] and Zoltan Somogyi[2]

1 **Mission Critical Australia**
novalazy@gmail.com
2 **Department of Computer Science and Software Engineering**
University of Melbourne, Australia and
National ICT Australia (NICTA)
zs@unimelb.edu.au

Abstract

Parallel implementations of programming languages need to control synchronization overheads. Synchronization is essential for ensuring the correctness of parallel code, yet it adds overheads that aren't present in sequential programs. This is an important problem for parallel logic programming systems, because almost every action in such programs requires accessing variables, and the traditional approach of adding synchronization code to all such accesses is so prohibitively expensive that a parallel version of the program may run more slowly on four processors than a sequential version would run on one processor. We present a program transformation for implementing dependent AND-parallelism in logic programming languages that uses mode information to add synchronization code only to the variable accesses that actually need it.

1998 ACM Subject Classification D.3.3 Concurrent Programming Structures, D.3.4 Compilers

Keywords and phrases synchronization, program transformation

Digital Object Identifier 10.4230/LIPIcs.ICLP.2011.128

1 Introduction

The usual motivation for parallelism is higher performance. Parallelizing imperative programs is notoriously errorprone, whereas in some declarative languages, introducing parallelism into a program can be as easy as adding a directive asking for it. Unfortunately, parallel implementations of declarative languages have a big enemy: overheads.

Parallel systems have overheads for the mechanisms used to manage the parallelism itself, as well as the overheads present in the sequential system they are based on. To get speedups over the best sequential systems, overheads must be kept to a minimum. Unfortunately, most sequential implementations of logic programming languages have quite high overheads. The best way to minimize overheads is to design a language that removes the need for them as far as possible, which is best done by moving all possible decisions to compile time. The only logic programming language designed with this objective is Mercury. Its implementation has consequently long been the fastest among sequential implementations of logic programming languages [8], though recently systems like YAP [3] have been catching up. This speed makes Mercury a good starting point for a parallel logic programming implementation.

Most parts of most Mercury programs are deterministic, so Mercury needs AND-parallelism, not OR-parallelism, and since most goals in predicate bodies depend on the goals preceding them, it needs *dependent* AND-parallelism. If you want to use this parallelism to achieve higher speeds than would be possible on one processor in *any* language (which may be the "killer application" of declarative languages), then you need to be very aggressive in

© Peter Wang and Zoltan Somogyi;
licensed under Creative Commons License ND
Technical Communications of the 27th International Conference on Logic Programming (ICLP'11).
Editors: John P. Gallagher, Michael Gelfond; pp. 128–138

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

keeping down the overheads of the synchronization required to implement the dependencies between parallel goals. Our paper presents a way to do this.

The structure of the paper is as follows. Section 2 briefly reviews the Mercury language. Sections 3 and 4 present the main contributions of the paper, our synchronization and specialization transformations respectively. Section 5 gives some performance results, while section 6 concludes with comparisons to related work.

2 Background

The part of Mercury relevant to this paper can summarized by this abstract syntax:

pred P	: $p(x_1, \ldots, x_n) \leftarrow G$	predicates
goal G	: $x = y \mid x = f(y_1, \ldots, y_n)$	unifications
	$\mid p(x_1, \ldots, x_n) \mid x_0(x_1, \ldots, x_n)$	first and higher order calls
	$\mid (G_1, \ldots, G_n) \mid (G_1 \ \& \ \ldots \ \& \ G_n)$	seq and par conjunctions
	$\mid (G_1; \ldots; G_n) \mid$ switch x $(\ldots;\ f_i : G_i; \ldots)$	disjunctions and switches
	$\mid (\mathit{if}\ G_c\ \mathit{then}\ G_t\ \mathit{else}\ G_e) \mid \mathit{not}\ G$	if-then-elses and negations
	$\mid \mathit{some}\ [x_1, \ldots, x_n]\ G$	quantifications

The atomic constructs of Mercury are unifications, plain first-order calls, and higher-order calls. The composite constructs include sequential disjunctions, if-then-elses, negations and existential quantifications. These should all be self-explanatory. A parallel conjunction is a conjunction in which all the conjuncts execute in parallel. They may be independent or dependent; if dependent, the dependencies must be respected. (The mechanism for enforcing the dependencies is the subject of this paper.) A switch is a disjunction in which each disjunct unifies the same variable, whose value is known, with a different function symbol.

Mercury has a strong mode system. The mode system classifies each argument of each predicate as either input or output; there are exceptions, but they are not relevant to this paper. If input, the caller must pass a ground term for that argument. If output, the caller must pass a distinct free variable, which the predicate will instantiate to a ground term. A predicate may have more than one mode; we call each one a *procedure*. The Mercury compiler generates different code for different procedures. The compiler is responsible for reordering conjuncts in both sequential and parallel conjunctions as necessary to ensure that for each variable shared between conjuncts, the goal that generates the value of the variable (the *producer*) comes before all the goals that use this value (the *consumers*). This means that for each variable in each procedure, the compiler knows exactly where in that procedure that variable gets grounded.

The mode system both establishes and requires some invariants. The conjunction invariant says that on any execution path consisting of conjoined goals, each variable that is consumed by any one of the goals is produced by exactly one goal. The branched goal invariant says that in disjunctions, switches and if-then-elses, each branch of execution must produce the exact same set of variables that are visible from outside the branched goal, with one exception: a branch of execution that cannot succeed may produce a subset of this set.

Each procedure and goal has a determinism, which puts limits on the number of its possible solutions. Goals with determinism *det* succeed exactly once, *semidet* goals at most once, *multi* goals at least once, while *nondet* goals may succeed any number of times. Each procedure's mode declaration declares the procedure's determinism.

Before we started this work, Mercury already supported independent AND-parallelism [2]. This meant that programmers who wanted to use parallelism could indicate *which*

```
:- pred p(int, int, int).        % after transformation
:- mode p(in, in, out) is det.   p(A, B, C) :-
                                   new_future(FutureD),
% before transformation            (
p(A, B, C) :-                        q(A, D),     % produces D
  (                                  signal_future(FutureD, D),
    q(A, D),      % produces D       r(D, E)      % produces E
    r(D, E)       % produces E     ) & (
  ) & (                              s(B, F),     % produces F
    s(B, F),      % produces F       wait_future(FutureD, D'),
    t(D, F, G)    % produces G       t(D', F, G) % produces G
  ),                               ),
  C = D + E + G. % produces C      C = D + E + G.  % produces C
```

Figure 1 Example of synchronizing a parallel conjunction

conjunctions should be executed in parallel by writing the parallel connective '&' instead of the sequential connective ',' between the conjuncts. The compiler then checked that the code satisfied the restrictions imposed on parallel conjunctions. The first restriction required all conjuncts in parallel conjunctions to be det, and the second required them to be all independent (no conjunct could produce any variable consumed by any other conjunct). The reason for the first check is that while there are ways to run nondeterministic conjuncts in parallel, the overheads of the algorithms required to assure correctness in such cases make it almost impossible to achieve speedups, even on several cores, over the best possible sequential version of the program at hand. We therefore keep the first restriction, even though our work obviously removes the second.

3 Synchronization for dependent AND-parallelism

Our main objective is to minimize the cost of synchronization by incurring it only when it is truly needed. In Mercury, this is relatively easy to do, due to the compiler's complete knowledge of the data flow in every predicate. (This was one of the design criteria for the language, even though the original implementation was sequential.) Given two or more conjuncts in a conjunction, the compiler knows not only which variables they have in common (i.e. which variables they *share*), but for each shared variable, it also knows which conjunct produces the variable and which conjuncts consume it.

The main parts of our implementation are the synchronization transformation (described in this section) and the specialization transformation (in section 4) . They both operate on a representation of the source code, which is based on the Mercury abstract syntax (section 2).

The synchronization transformation processes each parallel conjunction in the program independently. For each parallel conjunction, it computes the set of variables that need synchronization because they are produced by one conjunct and consumed by later conjuncts. For each such variable, it creates a new variable that we call a *future*. It then modifies all conjuncts to use these futures to synchronize all accesses to the shared variables.

We will introduce the synchronization transformation using the example in figure 1. The body of `p` contains a parallel conjunction, whose first conjunct is the sequential conjunction `q(A, D), r(D, E)` and whose second is the sequential conjunction `s(B, F), t(D, F, G)`. The comments indicate the modes of the calls.

```
add_sync_par_conj(Conjuncts, Purity) returns Goal:
let SharedVars be the set of variables that are
  (a) produced by one conjunct and
  (b) consumed by one or more later conjuncts
if SharedVars = ∅:
  Goal := par_conj(Conjuncts)
else:
  FutureMap := ∅
  AllocGoals := []
  for each Var in SharedVars:
    create a new var FutureVar
    FutureMap := FutureMap ∪ {Var → FutureVar}
    AllocGoals := [new_future(FutureVar) | AllocGoals]
  ParConjuncts := []
  for each Conjunct in Conjuncts:
    ParConjuncts := ParConjuncts ++
      [add_sync_par_conjunct(Conjunct, SharedVars, FutureMap, yes)]
  Goal := seq_conj(AllocGoals ++ [par_conj(ParConjuncts)])
```

Figure 2 Synchronizing a parallel conjunction

The basic idea of our work is that

- the producer of a shared variable will signal the availability of a value for the shared variable *immediately after* that value has been computed, and
- all consumers of a shared variable will wait for the availability of a value for the shared variable *immediately before* they use that value for the first time.

No consumer ever accesses the shared variable directly. Instead, each consumer waits on the future, and creates its own copy (`D'` in this case).

The data structures we use for this synchronization are the futures. (We got the name from MultiLisp [4]; our futures resemble theirs.) A future consists of these four fields:

- a boolean that says whether the future has been signalled yet;
- the value of the variable, if it has been signalled;
- a list of the contexts suspended waiting for this future, if it has not;
- a mutex to allow only one engine to access the future at a time.

The implementations of the three operations on futures are quite straightforward.

- `new_future` allocates a new future on the heap, and initializes its fields to the obvious values: not yet signalled, no waiting contexts.
- `wait_future` locks the mutex. If the future has been signalled, it picks up the its value and unlocks the mutex. Otherwise, the thread suspends itself on the future, unlocking the mutex. It will pick up the future's value when it resumes.
- `signal_future` locks the mutex, sets the value of the shared variable, records that the value is now available, makes all the threads suspended on the future runnable, and unlocks the mutex.

The synchronization transformation traverses the bodies of all procedures looking for parallel conjunctions. When it finds one, it invokes the algorithm in figure 2. In our algorithmsall variable names start with an upper case letter, names starting with lower case letters represent

```
add_sync_par_conjunct(Goal, SharedVars, FutureMap, Rename) returns NewGoal:
ConsumedVars := SharedVars ∩ Goal's consumed vars
ProducedVars := SharedVars ∩ Goal's produced vars
if ConsumedVars and ProducedVars are both ∅:
  NewGoal := Goal
else:
  NewGoal := Goal
  for each ConsumedVar in ConsumedVars:
    NewGoal := insert_wait_into_goal(NewGoal, ConsumedVar, FutureMap)
  for each ProducedVar in ProducedVars:
    NewGoal := insert_signal_into_goal(NewGoal, ProducedVar, FutureMap)
  if Rename:
    map each var in ConsumedVars to a fresh clone var
    apply this substitution to NewGoal
```

Figure 3 Synchronizing one parallel conjunct

the names of functions or data constructors, square parentheses represent lists as in Prolog syntax, and ++ means list concatenation. The `seq_conj` and `par_conj` data constructors create conjunctions of the named kinds out of a list of conjuncts.

If none of the conjuncts consumes a variable produced by one of the other conjuncts, then the conjunction represents independent AND-parallelism, and requires no synchronization (beyond the barrier at the end, which we will ignore from now on). Otherwise, the conjunction is dependent and needs synchronization using futures. We create one future for each shared variable; `FutureMap` maps each shared variable to the variable that holds its future. The goals that create and initialize the futures (`AllocGoals`) will all run before execution enters the parallel conjunction (`ParConjuncts`) in the transformed code.

Figure 3 shows the algorithm that transforms each parallel conjunct to ensure that the conjunct waits for each shared variable it consumes before its first use, and that it signals the availability of each shared variable it produces just after it is bound. First, the algorithm finds out which shared variables it consumes and which it produces. If there are no variables in

```
insert_wait_into_goal(Goal, ConsumedVar, FutureMap)
  returns <NewGoal, WaitAllPaths>:
if Goal does not consume ConsumedVar:
  NewGoal := Goal
  WaitAllPaths := false
else:
  switch on the type of Goal:
    the switch cases are shown in the following figures
    each case sets NewGoal and WaitAllPaths
  if not WaitAllPaths:
    create a clone of ConsumedVar, call it CloneVar
    apply the substitution {ConsumedVar → CloneVar} to NewGoal
```

Figure 4 Inserting waits into goals, top level

```
case unify, plain call, higher order call:
  look up ConsumedVar in FutureMap; call the result FutureVar
  WaitGoal := wait_future(FutureVar, ConsumedVar)
  NewGoal := seq_conj([WaitGoal, Goal])
  WaitAllPaths := true
case sequential conjunction:
  let Goal be seq_conj([Conjunct_1, ..., Conjunct_n])
  NewConjuncts := []
  WaitAllPaths := false
  for i = 1 to n:
    if Conjunct_i consumes ConsumedVar:
      <NewConjunct, ConjunctWaitAllPaths> :=
        insert_wait_into_goal(Conjunct_i, ConsumedVar, FutureMap)
      if ConjunctWaitAllPaths:
        NewConjuncts := NewConjuncts ++ [NewConjunct] ++
          [Conjunct_i+1, ...  Conjunct_n]
        WaitAllPaths := true
        break out of the loop
      else:
        NewConjuncts := NewConjuncts ++ [NewConjunct]
    else:
      NewConjuncts := NewConjuncts ++ [Conjunct_i]
  NewGoal := seq_conj(NewConjuncts)
```

Figure 5 Inserting waits into goals, part 1

either category, then this conjunct is independent of the others and needs no synchronization. Otherwise, we loop over the consumed and produced variables, each iteration modifying the conjunct by adding the code required to synchronize accesses to that variable.

The reason why `add_sync_par_conj` tells `add_sync_par_conjunct` to rename the variables consumed by each conjunct is to ensure that no variable is produced more than once on any one execution path (the mode system's conjunction invariant). The calls to `wait_future` bind their second argument. Since each shared variable is bound once in its producing conjunct, binding it again in one or more of its consuming conjuncts would violate this invariant. This way, each consuming conjunct gets its own copy of the shared variable, and the code *after* the parallel conjunction gets the original version from the producing conjunct.

Figures 4, 5, 6 and 7 each show part of the algorithm for inserting the call to `wait_future` on `ConsumedVar` into a goal.

The `insert_wait_into_goal` function returns `NewGoal`, an updated version of `Goal` in which every occurrence of `ConsumedVar` is preceded by code that picks up the value of `ConsumedVar` by waiting on its future. It also says whether all execution paths that lead to the success of `NewGoal` have such a wait operation on them. If they all do, then the code following the original `Goal` will be able to access `ConsumedVar` without waiting, and thus does not need to be transformed. If some or all do not, then the code after the original `Goal` will need to wait for `ConsumedVar`, and thus does need to be transformed. However, we do not want both `Goal` and the code following it to wait for and thus bind `ConsumedVar`, as this would violate the conjunction invariant. That is why, if `WaitAllPaths = false`, we replace all occurrences of `ConsumedVar` in `NewGoal` with a fresh variable. This renaming

```
case parallel conjunction:
  let Goal be par_conj([Conjunct_1, ..., Conjunct_n])
  NewConjuncts := []
  WaitAllPaths := false
  for i = 1 to n:
    if Conjunct_i is the first conjunct that consumes ConsumedVar:
      <NewConjunct, WaitAllPaths> :=
        insert_wait_into_goal(Conjunct_i, ConsumedVar, FutureMap)
      NewConjuncts := NewConjuncts ++ [NewConjunct]
    else if Conjunct_i consumes ConsumedVar:
      <NewConjunct, _> :=
        insert_wait_into_goal(Conjunct_i, ConsumedVar, FutureMap)
      create a clone of ConsumedVar, call it CloneVar
      apply the substitution {ConsumedVar → CloneVar} to NewConjunct
      NewConjuncts := NewConjuncts ++ [NewConjunct]
    else:
      NewConjuncts := NewConjuncts ++ [Conjunct_i]
  NewGoal := par_conj(NewConjuncts)
```

Figure 6 Inserting waits into goals, part 2

leaves the meaning of `NewGoal` unchanged. Therefore when `insert_wait_into_goal` returns `WaitAllPaths = false`, `NewGoal` will not bind `ConsumedVar`; it won't refer to it at all.

If the goal given to `insert_wait_into_goal` does not consume `ConsumedVar`, then the given goal does not need modification. If it does consume `ConsumedVar`, then the kind of modification it needs depends on what kind of goal it is, which is why the main body of the `insert_wait_into_goal` is a switch on goal type. The base case handles atomic goals (goals that do not contain other goals): unifications, calls, and higher order calls. For these goals, we insert a call to `wait_future` before the goal to wait for and pick up the value of the consumed variable. For unifications, this is the best we can do. For calls, we can wait for `ConsumedVar` closer to the time when it is actually needed, but we will worry about that in the next section.

`insert_wait_into_goal` processes the conjuncts of plain sequential conjunctions left to right. When it finds a conjunct that consumes `ConsumedVar`, it calls itself recursively to insert the call to `wait_future` into that conjunct. If all successful execution paths inside this conjunct wait for `ConsumedVar`, then it is guaranteed to be available by the time execution gets to the following conjuncts, so we can leave them unchanged. If only some execution paths inside this conjunct wait for `ConsumedVar`, then `NewConjunct` will wait on a renamed version of `ConsumedVar`, so `ConsumedVar` itself won't be bound when execution gets to the next conjunct. If none of the conjuncts that consume `ConsumedVar` wait for it on all paths, then `ConsumedVar` itself won't be bound by the conjunction itself, which is why we return `WaitAllPaths = false`.

Parallel conjunctions differ from sequential conjunctions in that even if one conjunct waits for `ConsumedVar` on all paths, the later conjuncts cannot assume that it will be available when they need it. Each conjunct that consumes `ConsumedVar` thus needs to wait for it independently. However, as above, having several conjuncts all ask their call to `wait_future` to bind the same variable (`ConsumedVar`) would violate the invariant that no variable have more than one producer in a conjunction. `insert_wait_into_goal` therefore renames the

```
case switch:
  if the switch is on ConsumedVar:
    look up ConsumedVar in FutureMap; call the result FutureVar
    WaitGoal := wait_future(FutureVar, ConsumedVar)
    NewGoal := seq_conj([WaitGoal, Goal])
    WaitAllPaths := true
  else:
    NewCases := []
    WaitAllPaths := true
    for each Case in Goal:
      if Case's goal consumes ConsumedVar:
        <NewCaseGoal, CaseWaitAllPaths> :=
          insert_wait_into_goal(Case's goal, ConsumedVar, FutureMap)
        NewCase := replace Case's goal with NewCaseGoal
        NewCases := NewCases ++ [NewCase]
        WaitAllPaths := WaitAllPaths and CaseWaitAllPaths
      else:
        NewCases := NewCases ++ [Case]
        WaitAllPaths := false
    NewGoal := replace Goal's cases with NewCases
```

Figure 7 Inserting waits into goals, part 3

occurrences of `ConsumedVar` in all but one of the conjuncts; the last one may also be renamed by `add_sync_to_par_conjunct` to avoid a similar collision with the binding made by a producer conjunct.

Given a switch on `ConsumedVar`, we obviously need to wait for the value of `ConsumedVar` before the switch. For switches on other variables, we insert the wait into the switch arms that consume `ConsumedVar`.

To see how `insert_wait_into_goal` treats other kinds of goals, and for the code of `insert_signal_into_goal`, see the long version of this paper on the Mercury web site.

4 The specialization transformation

The algorithms in the previous section try to push each `wait_future` operation as late as they can and each `signal_future` operation as early as they can, in order to maximize parallelism by maximizing the time during which the producer and the consumers of the shared variable can all run in parallel. However, the amount of parallelism they can create is limited by procedure boundaries. Overcoming this limitation is the task of the specialization algorithm. The first step in this algorithm is the "making requests" step, which looks for places in the code where creating specialized versions of procedures can increase parallelism:

- We look for plain calls to procedures preceded by one or more calls to `wait_future`, followed by one or more calls to `signal_future`, or both.
- For each of these `wait_future` operations, we check whether the waited-for variable is an input argument of the call and the callee does non-trivial work before it needs the variable. `PushedInputs` is the set of variables for which the answer is "yes".
- For each of these `signal_future` operations, we check whether the signalled variable is an output argument of the call and the callee does non-trivial work after it produces the

variable. `PushedOutputs` is the set of variables for which the answer is "yes".

- If `PushedInputs` ∪ `PushedOutputs` is not empty, and we have access to the code of the callee, we replace the call, all the `wait_futures` on `PushedInputs` and all the `signal_futures` on `PushedOutputs` with a call to a specialized version of the called procedure, one in which all the replaced waits and signals will be done in the callee. We also request that this specialized version be created. This request identifies the variables in `PushedInputs` and `PushedOutputs` by their position in the call's argument list.

The specialized version of the callee will have each formal parameter corresponding to a variable in `PushedInputs` or `PushedOutputs` replaced by a newly-created variable that will hold its future.

The second step executes each request by creating the specialized version of the request's callee. To generate the body of the new procedure, it invokes `add_sync_par_conjunct` from figure 3 with the callee's original code as `Goal`. As `SharedVars`, it passes the formal parameters corresponding to `PushedInputs` ∪ `PushedOutputs`, and `FutureMap` will map each variable in `SharedVars` to the future that replaced it in the list of formal parameters. `add_sync_par_conjunct` will push the synchronization code for each variable in `SharedVars` as deeply into the code of the callee as possible. Since the goal returned by `add_sync_par_conjunct` will be the body of the specialized procedure and thus will not be conjoined to anything, it is ok for this goal to bind the consumed variables, which is why we pass "no" as the value of `Rename`.

We invoke the specialization algorithm after the algorithms of section 3 have inserted synchronization operations into all the procedures in the module being compiled that contain dependent parallel conjunctions. We invoke the "making requests" step on all of these procedures, looking for opportunities for specialization, and modifying the code at each such opportunity as if the specialized version of the callee already existed. In doing so, we collect a queue of specialization requests, none of which have yet been acted upon. The rest of the specialization algorithm is a fixpoint operation. While there are specialization requests that have not yet been acted upon, the algorithm selects one such request, executes it (creating the requested specialized procedure), and then invokes the "making requests" step of the algorithm again on the newly created procedure to look for more specialization requests. Some of these may have been already acted upon, some of them may already be in the queue, and some may be new; we add the new ones to the queue. The algorithm must terminate because the number of procedures and of possible specialization requests are both finite.

5 Performance evaluation

We have room to report on only one benchmark, but it should suffice to demonstrate the low overheads of our system. This benchmark is a raytracer: its top level loops over the rows of the picture, with each iteration computing the RGB values of the pixels in a chunk of consecutive rows. We have two versions of this loop. The independent version (i1) computes the pixels for these rows, (i2) recurses to compute the pixels for the remaining rows, then (i3) add both sets of pixels to the list of pixels so far. The parallelized conjunction is `(i1 & i2), i3`, and there is no data flow from i1 to i2. The dependent version (d1) computes the pixels for these rows, (d2) adds these pixels to the list of pixels so far, and then (d3) recurses to compute the pixels for the remaining rows, based on the updated list of the pixels so far. The parallelized conjunction is `(d1, d2) & d3`, and there *is* data flow from d1 to d2 and from d2 to d3. We tested both forms of the loop with varying chunk sizes, which divided the rows into 1, 8, 16 and 32 chunks. The table shows speedups compared to the best sequential

version on our test machine, which was a Dell Optiplex 755 PC with a 2.4 GHz Intel Core 2 Quad Q6600 CPU. Each test was run ten times.

	1i	8i	16i	32i	1d	8d	16d	32d
sequential	1.00	1.00	1.00	1.00	0.99	0.99	0.99	0.99
parallel, 1 CPU	0.91	1.08	1.08	1.10	0.90	1.02	1.03	1.04
parallel, 2 CPUs	N/A	1.46	1.73	1.74	N/A	2.05	2.14	2.13
parallel, 3 CPUs	N/A	2.16	2.19	2.21	N/A	2.36	2.73	2.72
parallel, 4 CPUs	N/A	2.41	2.49	2.55	N/A	2.83	3.28	3.32

The parallel version of the system needs to use a real machine register to point to thread-specific data, such as each engine's abstract machine registers. On x86s, this leaves only one real register for the Mercury abstract machine, which is why switching to the parallel version of the system can yield a 10% slowdown on one CPU. However, inevitable differences between program versions (such as in the placement of code and data) happen to allow similar speedups as well.

The independent versions on the left do not need any of the algorithms in this paper. The dependent versions on the right do, and thus incur extra synchronization costs. However, these costs must be very small, since they are *massively* outweighed by the better cache utilization resulting from appending each chunk's pixels to the pixels so far immediately after computing those pixels. (In all versions, the append is done in constant time using a cord, not a list, and all versions do the same appends and compute the same cord.)

Different rows take different times to render. Dividing the input into more chunks evens out such fluctuations. The data shows this to be worthwhile (up to a point) even though it also increases synchronization overhead. Overall, the speedup of 3.32 over the sequential version and $3.32/1.04 = 3.19$ over the 1-CPU parallel version shows that we have succeeded in keeping synchronization costs to a very low level.

6 Conclusion

We don't have space for a detailed comparison with all existing parallel logic programming systems [9], so we list only the main differences between our work and previous systems.

- Unlike Concurrent Prolog, Parlog, GHC and their descendants, our system incurs synchronization overhead on accesses only to a very small proportion of variables; the fraction can be as low as tens out of billions.
- Our system cannot deadlock: every shared variable has a producer, which (unless it throws an exception) cannot exit without binding the variable.
- Our system does not require programmers to divide clauses into guard and body, much less into ask guard, tell guard and body.
- Unlike e.g. Janus [6], our system allows a variable to have more than one consumer.
- Unlike most other systems that allow multiple consumers, ours requires synchronization for consumers only up to the first one that waits for the variable on all paths.
- Unlike the DASWAM and similar parallel Prolog systems [7], our system does not allow the producer of a variable to be decided at runtime, so it does not have to keep track of which goal is the leftmost goal referring to a variable.
- Unlike CIAO's `<&` operator [1], our futures are significantly simpler to implement, and support finer grained parallelism.
- Unlike Moded FGHC [10], our system supports separate compilation.

One important further advantage of our system is that it completely eliminates the need for a system for controlling the order of parallel code's interactions with the outside world, such as the one in [5]. This is because Mercury uses dummy variables called I/O states to represent states of the outside world, and models operations that interact with the outside world as predicates that consume and destroy the current I/O state and produce a new one. If two or more goals in a conjunction update I/O states, the algorithm in section 3 will ensure that the parallel version of the conjunction executes the same actions in the same order as the sequential version.

The system we have described is part of recent versions of Mercury, which are available for free download from the Mercury project's web page.

We would like to thank Tom Conway for implementing independent AND-parallelism.

References

1 Daniel Cabeza and Manuel V. Hermenegildo. Implementing distributed concurrent constraint execution in the CIAO system. In *Proceedings of the AGP'96 Joint Conference on Declarative Programming*, pages 67–78, 1996.
2 Thomas C. Conway. *Towards parallel Mercury.* PhD thesis, University of Melbourne, 2002.
3 Anderson Faustino da Silva and Vítor Santos Costa. The design of the YAP compiler: an optimizing compiler for logic programming languages. *J. UCS*, 12(7):764–787, 2006.
4 Robert H Halstead. Implementation of Multilisp: Lisp on a multiprocessor. In *Proceedings of the 1984 ACM Symposium on List and Functional Programming*, pages 9–17, Austin, Texas, 1984.
5 K. Muthukumar and M. Hermenegildo. Complete and efficient methods for supporting side effects in independent/restricted and-parallelism. In *Proceedings of ICLP 6*, pages 80–101, Lisbon, Portugal, June 1989.
6 Vijay Saraswat. Janus: a step towards distributed constraint programming. In *Proceedings of the Second North American Conference on Logic Programming*, pages 431–446, Austin, Texas, October 1990.
7 Kish Shen. Overview of DASWAM: exploitation of dependent AND-parallelism. *Journal of Logic Programming*, 29(1-3):245–293, 1996.
8 Zoltan Somogyi, Fergus Henderson, and Thomas Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming*, 26(1-3):17–64, October-December 1996.
9 Evan Tick. The deevolution of concurrent logic programming languages. *Journal of Logic Programming*, 23(2):89–123, May 1995.
10 Kazunori Ueda and Masao Morita. Moded flat GHC and its message-oriented implementation technique. *New Generation Computing*, 13(1):3–43, 1994.

Smart test data generators via logic programming*

Lukas Bulwahn

Fakultät für Informatik
Technische Universität München
bulwahn@in.tum.de

Abstract

We present a novel counterexample generator for the interactive theorem prover Isabelle based on a compiler that synthesizes test data generators for functional programming languages (e.g. Standard ML, OCaml) from specifications in Isabelle. In contrast to naive type-based test data generators, the smart generators take the preconditions into account and only generate tests that fulfill the preconditions. The smart generators are constructed by a compiler that reformulates the preconditions as logic programs and analyzes them by an enriched mode inference. From this inference, the compiler can construct the desired generators in the functional programming language. These test data generators are applied to find errors in specifications, as we show in a case study of a hotel key card system.

1998 ACM Subject Classification D.2.5 Testing and Debugging

Keywords and phrases specification-based testing, functional programming

Digital Object Identifier 10.4230/LIPIcs.ICLP.2011.139

1 Introduction

Writing programs and specifications is an error-prone business and testing is common practice to find bugs and to validate software. Being aware that testing cannot prove the absence of bugs, formal methods are applied for safety- and security-critical systems. To ensure the correctness of programs, critical properties are guaranteed by a formal proof. Proof assistants are used to develop a proof with trustworthy sound logical inferences. Once one has completed the formal proof, the absence of bugs is certified. But in the process of proving, bugs could still be revealed and tracking down such bugs by failed proof attempts is a tedious task for the user. When reaching for the "holy grail", the formal proof, testing is still fruitful on the way to save time detecting bugs in programs and specifications. Modern interactive theorem provers therefore do not only provide means to prove properties, but also to *disprove* properties by counterexample generators.

Without specifications, it is common practice to write manual test suites to check properties. However, having a formal specification at hand, we can automatically generate test data and check if the program fulfills its specification. Such an automatic specification-based testing technique for functional Haskell programs was introduced by the tool QuickCheck [8]. The interactive theorem prover Isabelle [27] provides a counterexample generator [2] based on random testing, similar to QuickCheck. It works fine on specifications that have weak preconditions and properties stated in a form to be directly executable in the functional language. If the properties to be tested only hold under very specific preconditions, test data with a random distribution seldom fulfill the preconditions, and most execution time for testing is spent generating useless test values and rejecting them.

* This work was partially supported by DFG doctorate program 1480 (PUMA).

© Lukas Bulwahn;
licensed under Creative Commons License ND
Technical Communications of the 27th International Conference on Logic Programming (ICLP'11).
Editors: John P. Gallagher, Michael Gelfond; pp. 139–150

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

Our new approach aims to only generate test data that fulfill the preconditions. The test data generator for a given precondition is produced by a compiler[1] that analyzes precondition and synthesizes a purely functional program that serves as generator. For this purpose, the compiler reformulates the preconditions as logic programs. In this process, we adopt various techniques from logic programming. Formulas in predicate logic with quantifiers and recursive functions are translated to logic programs. The compiler then analyzes the logic program by an enriched mode inference. From this inference, the compiler can construct the desired generators. This way, a much smaller number of test cases suffices to find errors in specifications.

We introduce the Isabelle system and illustrate how Isabelle's users interactively explore proofs (Section 2). We present a concrete case study for the counterexample generator (Section 3). In the main part, we then describe key ideas of this counterexample generator, the preprocessing, enriched mode inference and compilation (Sections 4 to 7). In the end, we discuss related work (Section 8) and conclude.

2 Interactive Theorem Prover Isabelle

The Isabelle system is a generic framework for interactive theorem proving. Isabelle's logic is a higher-order logic with Hindley-Milner polymorphism. It provides the usual boolean operations, functional programming constructs, e.g., *let*, *if* and *case* expressions, and mechanisms to define recursive datatypes, recursive functions, and inductive predicates by Horn clauses. The code generation facility creates programs in functional programming languages, e.g., ML, Haskell or OCaml, from their specifications in the theorem prover.

Users of interactive theorem prover intend to construct a machine-checked proof in cooperation with the theorem prover. After stating a proposition, the system invokes automated proof methods and counterexample generators. If the user's proposition is proved automatically, its validity is certified by the system. If a counterexample is found, the user must refine the proposition. If neither happens, the user continues to explore the proof by stating further propositions. In a horizontal exploration, the user believes its truth, but does not prove it formally. Instead, he sketches further lemmas and proof steps based on the skipped proposition.

If the counterexample generators were not be in place, the horizontal exploration would have a major pitfall: With no counterexample generator, the statement is unchecked, but the user continues to develop the proof upon this wrong assertion. Realizing the flaw at a later stage requires much effort restructuring the proof. Counterexample generators are vital to spare the user this frustration and time-consuming work of unsuccessful proof attempts.

3 Case Study: Hotel Key Card System

Most hotels employ some kind of digital key card system. We describe a hotel key card system where every room is secured by a digital lock. Every guest of the hotel receives a card at the reception. The locks at the rooms can read the cards from the guest, and open the door if it is the card of the owner. The key card system is *decentralized*, i.e., the locks cannot communicate with each other or the reception. Nevertheless, only guests that check

[1] Throughout the presentation, we use the term *compilation* with a very specific meaning: to designate our translation of Horn specifications in Isabelle into programs written in a functional programming language.

in for a room should be allowed to enter, and previous guests should not have further access to the room once they checked out.

Safety is achieved by the following system: Upon check-in, a new guest gets a card at the reception which carries two keys, the old key of the previous guest of the room and his own new key. The locks only store their current key, i.e., the key of the latest guest the lock has been aware of. When a new guest enters the room, the lock checks if the old card key matches its current key, and if so discards the old key and stores the new key as its new current key. Once the lock has been recoded, it allows access only to the card with the current key until the next guest enters the room. This recoding ensures that the previous guest cannot enter the room after the new guest has been in his room.

Nipkow [17] gives a formalization of this hotel key card system in Isabelle which itself was inspired by a model from Jackson [13]. The safety property of the hotel key card system is: Once the owner of the room, i.e., the guest who was the last to check in, entered his room, no previous guest can enter the room (even if they have kept or copied their cards). Unfortunately, this property does not hold. But fortunately, the counterexample generator finds the tricky man-in-the-middle attack.

In Isabelle, the hotel key card system is formalized as follows: We consider three events, a guest g checking in for a room r where he gets a card with keys (k, k') from the reception, a guest g entering a room r with a card with keys (k, k'), or a guest g leaving a room r. We denote these events as $\mathit{Checkin}\ g\ r\ (k, k')$, $\mathit{Enter}\ g\ r\ (k, k')$, and $\mathit{Exit}\ g\ r$. A trace, represented as a list of events, describes the temporal order of events taking place. Without going into details here, the set of possible and valid traces in a hotel and safety is given by the following functional description in Isabelle:

$$
\begin{array}{l}
\mathit{hotel}\ [] = \mathit{True} \\
\mathit{hotel}\ (e \cdot \mathit{evs}) = (\mathit{hotel}\ \mathit{evs} \wedge (\mathit{case}\ e\ \mathit{of} \\
\quad \mathit{Checkin}\ g\ r\ (k, k') \Rightarrow k = \mathit{currentkey}\ \mathit{evs}\ r \wedge k' \notin \mathit{issued}\ \mathit{evs}\ | \\
\quad \mathit{Enter}\ g\ r\ (k, k') \Rightarrow (k, k') \in \mathit{cards}\ \mathit{evs}\ g \wedge (\mathit{roomkey}\ \mathit{evs}\ r \in \{k, k'\})\ | \\
\quad \mathit{Exit}\ g\ r\ \Rightarrow g \in \mathit{isin}\ \mathit{evs}\ r)) \\
\mathit{safe}\ \mathit{evs}\ r = \\
\quad \exists \mathit{evs}_1\ \mathit{evs}_2\ \mathit{evs}_3\ g\ c\ c'.\ \mathit{evs} = \mathit{evs}_3\ @\ (\mathit{Enter}\ g\ r\ c \cdot \mathit{evs}_2\ @\ \mathit{Checkin}\ g\ r\ c' \cdot \mathit{evs}_1) \wedge \\
\quad \mathit{noCheckin}\ (\mathit{evs}_3\ @\ \mathit{Enter}\ g\ r\ c \cdot \mathit{evs}_2) \wedge \mathit{isin}\ (\mathit{evs}_2\ @\ \mathit{Checkin}\ g\ r\ c' \cdot \mathit{evs}_1) = \emptyset
\end{array}
$$

where $\mathit{noCheckin}\ \mathit{evs}\ r\ = \neg(\exists g\ c.\ \mathit{Checkin}\ g\ r\ c \in \mathit{evs})$ and @ denotes appending two lists.

The safety property is formally $\mathit{hotel}\ \mathit{evs} \wedge \mathit{safe}\ \mathit{evs}\ r \wedge g \in \mathit{isin}\ \mathit{evs}\ r \Longrightarrow \mathit{owner}\ \mathit{evs}\ r = g$. When checking the validity of this property, the existing counterexample generator for Isabelle [2], based on the ideas of QuickCheck, faces two problems:

Firstly, naive black-box testing would generate traces where most traces do not fulfill the necessary conditions to be a valid hotel key card trace, i.e., *hotel evs* evaluates to false for most traces *evs*. The common approach is to write a manual generator for the predicate *hotel*. But the functional description already contains all necessary information on how to construct values that fulfill the predicate *hotel*, which is *not* exploited by black-box testing.

Secondly, a predicate, such as *safe* cannot be (naively) executed because it contains unbounded existential quantifiers (over an infinite type) for evs_1, evs_2, evs_3. However, having a closer look at the description of *safe*, we see that the values for evs_1, evs_2 and evs_3 are actually restricted to be parts of the trace *evs*, which could be computed given *evs*.

The system we describe tackles the two problems, generating test data that fulfills the precondition, and detecting for which quantifier the values are bound within the computation, using an enriched mode analysis. For the hotel key card example, the new approach allows us to find the man-in-the-middle attack within a few seconds, whereas the black-box testing does not find the counterexample even after ten minutes of testing.

4 Overview of the tool

In this section, we present the overall structure of our counterexample generator, and motivate the key features and design decisions. The detached presentation of individual components is then discussed in the following three sections.

Design decisions. The Isabelle system is implemented in ML and runs in the interactive shell of the ML compiler. ML source code generated from Isabelle specifications can be passed to the underlying ML compiler, and executed in the same process.

As Isabelle runs on top of the ML compiler, it is a natural choice to test specifications in the underlying programming language ML as it adds no further system requirements and dependencies. ML as a strict functional language does not support natively the common execution mechanisms from logic programming, such as non-determinism and logic variables in terms. If we want to use these mechanisms, we must *embed them in the functional language*. But the execution in an embedding is considerably slower than native execution. We aim to find a balance between *deep embedding* and *fast execution*.

We only allow values to be *ground* during the execution, which is the native setting for purely functional languages. In other words, we do not allow logic variables in terms or partially instantiated terms. Allowing such terms would require that even purely functional specifications be embedded deeply as well, causing a tremendous overhead for purely functional execution. Restricting ourselves to ground terms has the advantage that functional specifications can be translated directly into the target functional programming language. The test data generators only construct proper ground values in the functional programming language. Only parts of the specification, i.e., predicates occurring in preconditions, are compiled to test data generators with an embedding of nondeterministic execution whereas the testing of the conclusion can be done via the fast direct execution mechanism of the functional language. E.g., consider the safety property of the hotel key card system, the set of values for evs, r, and g for the preconditions, $\mathsf{hotel}\ evs$, $\mathsf{safe}\ evs$ and $\mathsf{isin}\ evs\ r$, are computed with the deep embedding, the conclusion, $\mathsf{owner}\ evs\ r = g$, is executed as functional program directly.

The decision above burdens the compilation with a static analysis – the mode analysis – to determine a possible dataflow of ground values in the description of the precondition. However, the advantage of smarter test data generators is worth this burden. The test data generators commonly return a *set* of values – we implement this behavior using lazy sequences in ML.

In summary, our system compiles predicate preconditions to smart test data generators using ground terms, a dataflow analysis and nondeterministic execution. The conclusion is tested via direct functional execution.

Architecture. The counterexample generator performs the these steps: As the original specification can be defined using various definitional mechanisms, the specification is preprocessed by a few simple syntactic transformations (Section 5) to Horn clauses. The core component, which was previously described in [1], consists of the mode analysis (Section 6) and the code generator (Section 7). This core component only works on a syntactic subset of the Isabelle language, namely Horn clauses of the following form:

$$Q_1\ \overline{u}_1 \Longrightarrow \cdots \Longrightarrow Q_n\ \overline{u}_n \Longrightarrow P\ \overline{t}$$

In a premise $Q_i\ \overline{u}_i$, Q_i must be a predicate defined by Horn clauses and the terms $\overline{u}_i$ must be constructor terms, i.e., only contain variables or datatype constructors. Furthermore, we allow negation of atoms, assuming the Horn clauses to be stratified. If a premise obeys these

restrictions, the core compiler infers modes and compiles functional programs for the inferred modes. If a premise has a different form, e.g., the terms contain function symbols, or a predicate is not defined by Horn clauses, the core compiler will treat them as side conditions. For side conditions, the mode analysis does not infer modes, but requires all arguments as inputs. Enriching the mode analysis, we mark unconstrained values to be generated. Once we have inferred modes for the Horn clauses, these are turned into test data generators in ML using lazy sequences and type-based generators.

5 Preprocessing

In this section, we sketch how specifications in predicate logic and functions are preprocessed to Horn clauses. A definition in predicate logic is transformed to a system of Horn clauses, based on the fact that a formula of the form $P\ \overline{x} = \exists \overline{y}.\ Q_1\ u_1 \wedge \cdots \wedge Q_n\ u_n$ can be soundly underapproximated by a Horn clause $Q_1\ u_1 \Longrightarrow \cdots \Longrightarrow Q_n\ u_n \Longrightarrow P\ \overline{x}$. Predicate logic formulas in different form are transformed in the form above by a few logical rewrite rules in predicate logic. We rewrite universal quantifiers to negation and existential quantifiers, put the formula in negation normal form, and distribute existential quantifiers over disjunctions. In the process of creating Horn clauses, it is necessary to introduce new predicates for sub formulas, as our Horn clauses do not allow disjunctions within the premises and nested expressions under negations. Furthermore, we take special care of *if*, *case* and *let*-constructions.

▶ **Example 1.** The predicate *hotel* is processed to a system of Horn clauses with predicates *hotel*, $\mathit{hotel}_{\mathit{aux}}$ and $\mathit{hotel}_{\mathit{aux2}}$; the latter two are introduced during the preprocessing:

$$\mathit{hotel}\ []$$
$$\mathit{hotel}\ \mathit{evs} \Longrightarrow \mathit{hotel}_{\mathit{aux}}\ e\ \mathit{evs} \Longrightarrow \mathit{hotel}\ (e \cdot \mathit{evs})$$
$$k = \mathit{currentkey}\ \mathit{evs}\ r \Longrightarrow k' \notin \mathit{issued}\ \mathit{evs} \Longrightarrow \mathit{hotel}_{\mathit{aux}}\ (\mathit{Checkin}\ g\ r\ (k, k'))\ \mathit{evs}$$
$$(k, k') \in \mathit{cards}\ \mathit{evs}\ g \Longrightarrow \mathit{hotel}_{\mathit{aux2}}\ \mathit{evs}\ r\ k\ k' \Longrightarrow \mathit{hotel}_{\mathit{aux}}\ (\mathit{Enter}\ g\ r\ (k, k'))\ \mathit{evs}$$
$$g \in \mathit{isin}\ \mathit{evs}\ r \Longrightarrow \mathit{hotel}_{\mathit{aux}}\ (\mathit{Exit}\ g\ r)\ \mathit{evs}$$
$$\mathit{roomkey}\ \mathit{evs}\ r\ k \Longrightarrow \mathit{hotel}_{\mathit{aux2}}\ \mathit{evs}\ r\ k\ k'$$
$$\mathit{roomkey}\ \mathit{evs}\ r\ k' \Longrightarrow \mathit{hotel}_{\mathit{aux2}}\ \mathit{evs}\ r\ k\ k'$$

To enable inversion of functions, we preprocess n-ary functions to $(n+1)$-ary predicates defined by Horn clauses, which enables the core compilation to inspect the definition of the function and leads to better synthesized test data generators. This is achieved by *flattening* a nested functional expression to a flat relational expression, i.e., a conjunction of premises in a Horn clause.

▶ **Example 2.** Consider the formula $\mathit{evs} = \mathit{evs}_3\ @\ (\mathit{Enter}\ g\ r\ c \cdot \mathit{evs}_2\ @\ \mathit{Checkin}\ g\ r\ c' \cdot \mathit{evs}_1)$ used in the predicate *safe*. This formula is flattened to two premises,

$$\mathit{append}_P\ \mathit{evs}_2\ (\mathit{Checkin}\ g\ r\ c' \cdot \mathit{evs}_1)\ r_1 \text{ and } \mathit{append}_P\ \mathit{evs}_3\ (\mathit{Enter}\ g\ r\ c \cdot r_1)\ \mathit{evs},$$

and append_P is defined by two Horn clauses derived from its functional definition:

$$\mathit{append}_P\ []\ \mathit{ys}\ \mathit{ys} \text{ and } \mathit{append}_P\ \mathit{xs}\ \mathit{ys}\ \mathit{zs} \Longrightarrow \mathit{append}_P\ (x \cdot \mathit{xs})\ \mathit{ys}\ (x \cdot \mathit{zs})$$

This well-known technique is similarly described by Naish [16] and Rouveirol [21]. We also support flattening of higher-order functions, which allows inversion of higher-order functions if the function argument is invertible.

6 Mode analysis

In order to execute a predicate P, its arguments are classified as *input* or *output*, made explicit by means of *modes*. Modes can be inferred using a static analysis on the Horn clauses. Our mode analysis is based on [15]. There are more sophisticated mode analysis approaches, e.g., by Smaus et al. [23] using abstract domains and Overton et al. [18] by translating to a boolean constraint system. For our purpose, we can apply the simple mode analysis, because if the analysis does not discover a dataflow due to its imprecision, the overall process still leads to an test data generator.

Modes. For a predicate P with k arguments, we call *mode* a particular dataflow assignment by a which follows the type of the predicate and annotates all arguments as input (i) or output (o), e.g., for $append_P$, $o \Rightarrow o \Rightarrow i \Rightarrow bool$ denotes the mode where the first two arguments are output, the last argument is input.
A *mode assignment* for a given clause $Q_1\ \overline{u}_1 \Longrightarrow \cdots \Longrightarrow Q_n\ \overline{u}_n \Longrightarrow P\ \overline{t}$ is a list of modes $M, M_1, \ldots M_n$ for the predicates $P, Q_1, \ldots, Q_n$. Let $FV(t)$ denote the set of free variables in a term t. Given a vector of arguments $\overline{t}$ and a mode M, the projection expression $\overline{t}\langle M\rangle$ denotes the list of all arguments in $\overline{t}$ (in the order of their occurrence) which are input in M.

Mode consistency. Given a clause $Q_1\ \overline{u}_1 \Longrightarrow \cdots \Longrightarrow Q_n\ \overline{u}_n \Longrightarrow P\ \overline{t}$ a corresponding mode assignment $M, M_1, \ldots M_n$ is *consistent* if the chain of sets of variables $v_0 \subseteq \cdots \subseteq v_n$ defined by **(1)** $v_0 = FV(\overline{t}\langle M\rangle)$ and **(2)** $v_j = v_{j-1} \cup FV(\overline{u}_j)$ obey the conditions **(3)** $FV(\overline{u}_j\langle M_j\rangle) \subseteq v_{j-1}$ and **(4)** $FV(\overline{t}) \subseteq v_n$. Mode consistency guarantees the possibility of a sequential evaluation of premises in a given order, where v_j represents the known variables after the evaluation of the j-th premise. Without loss of generality, we can examine clauses under mode inference modulo reordering of premises. For side conditions R, condition 3 has to be replaced by $FV(R) \subseteq v_{j-1}$, i.e., all variables in R must be known when evaluating it. This definition yields a check whether a given clause is consistent with a particular mode assignment.

Generator mode analysis. To generate values that satisfy a predicate, we extend the mode analysis in a genuine way: If the mode analysis cannot detect a consistent mode assignment, i.e., the values of some variables are not *constrained* after the evaluation of the premises, we allow the use of *generators*, i.e., the values for these variables are constructed by an *unconstrained* enumeration. In other words, we combine two ways to enumerate values, either driven by the *computation* of a predicate or by *generation* based on its type.

▶ **Example 3.** Given a unary predicate R with possible modes $i \Rightarrow bool$ and $o \Rightarrow bool$ and the Horn clause $R\ x \Longrightarrow P\ x\ y$, classical mode analysis fails to find a consistent mode assignment for P with mode $o \Rightarrow o \Rightarrow bool$. To generate values for x and y fulfilling P, we combine computations and generation of values as follows: the values for variable x are built using R with $o \Rightarrow bool$; values for y are built by a generator.

This extension gives rise to a number of possible modes, because we actually drop the conditions 3 and 4 for the mode analysis. Instead, we use a heuristic to find a considerably good dataflow by locally selecting the optimal premise Q_j and mode M_j with respect to the following criteria:

1. minimize number of missing values, i.e., have $|FV(\overline{u}_j\langle M_j\rangle) - v_{j-1}|$ is minimal;
2. use functional predicates with their functional mode;
3. use predicates and modes that do not require generators themselves;
4. minimize number of output positions;
5. prefer recursive premises.

Next, we motivate and illustrate these five criteria. In general, we would like to avoid generation of values and computations that could fail, and to restrain ourselves from enumerating any values that could possibly be computed. Hence, the first priority is to use modes where the number of missing values is minimal. This way, we partly recover conditions 3 and 4 from the mode analysis.

▶ **Example 3 (continued).** For mode M_1 for $R\ x$, one has two alternatives: generating values for x and then testing R with mode $i \Rightarrow bool$, or only generating values for x using R with $o \Rightarrow bool$. The first choice generates values and rejects them by testing; the latter only generates fulfilling values and is preferable. The analysis favors $o \Rightarrow bool$ to $i \Rightarrow bool$ due to criterion 1: for $v_0 = \{\}$, $\overline{u}_1 = x$ and $M_1 = i \Rightarrow bool$, $FV(\overline{u}_1\langle M_1\rangle) - v_0 = \{x\}$; whereas for $M_1 = o \Rightarrow bool$, $FV(\overline{u}_1\langle M_1\rangle) - v_0 = \{\}$. $|FV(\overline{u}_1\langle M_1\rangle) - v_0|$ is minimal for $M_1 = o \Rightarrow bool$.

▶ **Example 4.** Consider a clause $R\ x\ y \Longrightarrow F\ x\ y \Longrightarrow P\ x\ y$ where R is a one-to-many relation and F is functional, i.e., a one-to-one relation. R and F both allow modes $i \Rightarrow o \Rightarrow bool$ and $i \Rightarrow i \Rightarrow bool$. For $M = i \Rightarrow o \Rightarrow bool$, $R\ x\ y$ and $F\ x\ y$ can be evaluated in either order. Our criterion 2 induces preference for computing y with the functional computation $F\ x\ y$ and checking $R\ x\ y$, i.e., the one value for y can either fulfill $R\ x\ y$ or not.

Criterion 3 induces avoiding the generation of values in the predicate to be invoked. Furthermore, we minimize output positions, e.g., we prefer checking a predicate (no output position) before computing some solution (one output position) as we illustrate by the following example:

▶ **Example 5.** In a clause $R\ x\ y \Longrightarrow Q\ x \Longrightarrow P\ x\ y$ with mode $i \Rightarrow o \Rightarrow bool$ for R and P, and $i \Rightarrow bool$ for Q, we prefer $Q\ x$ before $R\ x\ y$, since computing values for y would be useless if $Q\ x$ fails. This ordering is enforced by criterion 4.

Finally, we prefer recursive premises - this leads to a bottom-up generation of values. Generating larger values for predicates from smaller values for the predicate is commonly preferable because it takes advantage of the structure of the preconditions.

▶ **Example 6.** In a clause $P\ xs \Longrightarrow C\ xs \Longrightarrow P\ (x \cdot xs)$, $P\ xs$ is favored for generation of xs and $C\ xs$ for checking. Generating values for P, we apply the generator for P recursively and check the condition $C\ xs$ afterwards.

This “aggressive” mode analysis results in moded Horn clauses with annotations for generators of values. In summary, it does not only *discover* an existing dataflow, but helps *creating* a dataflow by filling the gaps with value generators.

7 Generator compilation

In this section, we discuss the translation of the compiler from moded Horn clauses to functional programs. First, we present the building blocks of the compiler, the execution mechanism and the generators. Then, we sketch the compilation scheme and show its application in the hotel key card example.

Monads for non-deterministic computations. We use lazy sequences to enumerate the (potentially infinite) set of values fulfilling the involved predicates – in other words, the lazy sequences will hold the *enumerated solutions*. As customary [25], they are implemented using the ML datatype *'a lazy*. On lazy sequences, we define *plus monad* operations describing non-deterministic computations. Depending on our enumeration scheme, we employ three

different plus monads: one for unbounded computations, and two others for depth-limited computations within positive and negative contexts, respectively.

A plus monad supports four operations: *empty*, *single*, *plus* and *bind*. They provide executable versions of basic set operations: $\mathit{empty} = \emptyset$, $\mathit{single}\ x = \{x\}$, $\mathit{plus}\ A\ B = A \cup B$ and $\mathit{bind}\ A\ f = \bigcup_{x \in A} f\ x$. Using lazy sequences results in a Prolog-like execution strategy, with a depth-first search. This strategy is fine for user-initiated evaluations, but for counterexample generation, automatically generated values cause infinite computations escaped from the control of the user. To avoid being stuck in such a computation, we also employ a plus monad with a different carrier that limits the computation by a depth-limit. Evaluating predicates with a depth-limited computation, we must take special care of negation. We implement different behaviors for queries in different contexts: for positive contexts, we compute an underapproximation; for negative contexts, an overapproximation.

For positive contexts, we implement a plus monad with the type $\mathit{int} \to \mathit{'a\ lazy}$ as carrier. The bind^+ operation checks the depth-limit and if reached, returns *empty*, which yields a sound underapproximation; otherwise it passes a decreased depth-limit to its argument. It is defined by $\mathit{bind}^+\ xq\ f = (\lambda i.\ \mathit{if}\ i = 0\ \mathit{then\ empty\ else\ bind}\ (xq\ (i - 1))\ (\lambda a.\ f\ a\ i))$.

In negative contexts, we must distinguish more explicitly failure (no solution found) from reaching the depth limit. To signal reaching the depth-limit, we include an explicit element to model an *unknown* value (as a third truth value), and continue the computation with this value. This makes the monad carrier type be $\mathit{int} \to \mathit{'a\ option\ lazy}$ where the option value *None* stands for unknown. If one computation reaches the depth-limit and another computation fails, then the overall computation fails, in other words *failure absorbs the unknown value* (which is consistent with a three-valued logic interpretation). This is witnessed by the behavior of the bind^- operator: $\mathit{bind}^-\ \mathit{single\text{-}none}\ (\lambda x.\ \mathit{empty}^-) = \mathit{empty}^-$ where *single-none* is the singleton sequence with the unknown value.

Because negative and positive occurrences of predicates are intermixed, in actual enumeration we have to combine the positive and negative monads – the bridge between them is performed by executable *not*-operations that handle the unknown value depending on the context. For instance, when applied to a solution enumeration of a negated premise, *unknown* is mapped to *false* (computation failure); this reflects the intuition that if we were not able to prove a negated premise $\neg Q\ x$ within a given depth-limit for x, then all we can soundly assume is that $Q\ x$ may hold; hence the computation cannot proceed further.

The compilation scheme builds abstractly on the monad structure interface and hence is employed for all three monads. For the rest of the presentation, we write *plus* and *bind* infix as $\sqcup$ and $\ggg=$.

Type-based generators. If values cannot be computed, we enumerate them up to a given depth. To generate values of a specific type, we make use of type classes in Isabelle. More specifically we require which the involved types τ come equipped with an operation $\mathit{gen}\ \tau$, the generator for type τ that enumerates all values as lazy sequence. For recursively-defined datatypes τ with n constructors $C_1\ \tau_1^1 \dots \tau_1^{m_1} \mid \dots \mid C_n\ \tau_n^1 \dots \tau_n^{m_n}$ we construct generators that enumerate values exhaustively up to depth d by the following scheme:

$$
\begin{array}{l}
\mathit{gen}\ \tau\ d = \\
\quad \mathit{if}\ d = 0\ \mathit{then\ empty\ else} \\
\quad\quad (\mathit{gen}\ \tau_1^1\ (d-1) \ggg= (\lambda x^1.\ \mathit{gen}\ \tau_1^2\ (d-1) \ggg= \dots \ggg= (\lambda x^{m_1-1}. \\
\quad\quad\quad \mathit{gen}\ \tau_1^{m_1}\ (d-1) \ggg= (\lambda x^{m_1}.\ \mathit{single}\ (C_1\ x^1 \dots x^{m_1})))\dots)) \ \sqcup\ \dots\ \sqcup \\
\quad\quad (\mathit{gen}\ \tau_n^1\ (d-1) \ggg= (\lambda x^1.\ \mathit{gen}\ \tau_n^2\ (d-1) \ggg= \dots \ggg= (\lambda x^{m_n-1}. \\
\quad\quad\quad \mathit{gen}\ \tau_n^{m_n}\ (d-1) \ggg= (\lambda x^{m_n}.\ \mathit{single}\ (C_n\ x^1 \dots x^{m_n})))\dots))
\end{array}
$$

Compilation of moded clauses. The central idea underlying the compilation of a predicate P is to generate a function P^M for each mode M of P that, given a list of input arguments, enumerates all tuples of output arguments. The functional equation for P^M is the union of the output values generated by the characterizing clauses. Employing the data flow from the mode inference, the expressions for the clauses are essentially constructed as chains of type-based generators and function calls for premises, connected through *bind* and *case* expressions. All functions P^M are executable in ML, because they only employ the monad operations and pattern matching. The function P^M for the mode M with all arguments as output serves as test data generator for predicate P.

▶ **Example 7.** The ML function hotel^o with mode $o \Rightarrow \mathit{bool}$ is the test data generator for the predicate *hotel*:

$$
\begin{array}{l}
\textbf{val}\ \mathit{hotel}^o = (\mathit{single}\ ()\ \ggg= (\lambda().\ \mathit{single}\ []))\\
\quad \sqcup (\mathit{single}\ ()\ \ggg= (\lambda().\ \mathit{hotel}^o\ \ggg= (\lambda \mathit{evs}.\ {\mathit{hotel}_{\mathit{aux}}}^{oi}\ \mathit{evs} \ggg= (\lambda e.\ \mathit{single}\ (e \cdot \mathit{evs})))))\\
\textbf{fun}\ {\mathit{hotel}_{\mathit{aux}}}^{oi}\ \mathit{evs} = (\mathit{single}\ \mathit{evs} \ggg= (\lambda \mathit{evs}.\ {\mathit{currentkey}_P}^{ioo}\ \mathit{evs} \ggg= (\lambda(r, k_1).\ \mathit{gen}_{\mathit{key}}\\
\qquad \ggg=(\lambda k_2.\ \mathit{not}\ (\mathit{issued}^{ii}\ \mathit{evs}\ k_2) \ggg= (\lambda().\ \mathit{gen}_{\mathit{guest}} \ggg= (\lambda g.\ \mathit{single}\ (\mathit{Checkin}\ g\ r\ (k_1, k_2))))))))\\
\quad \sqcup (\mathit{single}\ \mathit{evs} \ggg= (\lambda \mathit{evs}.\ \mathit{cards}^{ioo}\ \mathit{evs}\\
\qquad \ggg=(\lambda(g, (k_1, k_2)).\ {\mathit{hotel}_{\mathit{aux2}}}^{ioii}\ \mathit{evs}\ k_1\ k_2 \ggg= (\lambda r.\ \mathit{single}\ (\mathit{Enter}\ g\ r\ (k_1, k_2))))))\\
\quad \sqcup (\mathit{single}\ \mathit{evs} \ggg= (\lambda \mathit{evs}.\ \mathit{isin}^{ioo}\ \mathit{evs}\ \ggg= (\lambda(r, g).\ \mathit{single}\ (\mathit{Exit}\ g\ r))))\\
\textbf{fun}\ {\mathit{hotel}_{\mathit{aux2}}}^{ioii}\ \mathit{evs}\ k_1\ k_2 =\\
\quad (\mathit{single}\ (\mathit{evs}, (k_1, k_2)) \ggg= (\lambda(\mathit{evs}, (k_1, _)).\ {\mathit{roomkey}_P}^{ioi}\ \mathit{evs}\ k_1 \ggg= (\lambda r.\ \mathit{single}\ r)))\\
\quad \sqcup (\mathit{single}\ (\mathit{evs}, (k_1, k_2)) \ggg= (\lambda(\mathit{evs}, (_, k_2)).\ {\mathit{roomkey}_P}^{ioi}\ \mathit{evs}\ k_2 \ggg= (\lambda r.\ \mathit{single}\ r)))
\end{array}
$$

The generator hotel^o constructs hotel traces in a bottom-up fashion. ${\mathit{hotel}_{\mathit{aux}}}^{oi}$ adds a new event as prefix to (shorter) hotel traces. ${\mathit{hotel}_{\mathit{aux}}}^{oi}$ can either prefix a trace by *Checkin*, *Enter*, or *Exit* events; the conditions for these events, i.e., restriction on the values of these constructors, are fulfilled by either computing values using further generating functions or are generated unrestrictedly based on their type. An instance of computation is the call $\mathit{isin}^{ioo}\ \mathit{evs}$ to construct *Exit* events; an instance of generation is $\mathit{gen}_{\mathit{guest}}$ to select a guest for *Checkin* events. Applying the counterexample generator to the safety property (cf. §3) results in the following counterexample trace:

$$
\begin{array}{l}
\mathit{Enter}\ g_1\ r_0\ (k_1, k_2) \cdot \mathit{Enter}\ g_1\ r_0\ (k_0, k_1) \cdot \mathit{Checkin}\ g_0\ r_0\ (k_2, k_3)\\
\cdot\ \mathit{Checkin}\ g_1\ r_0\ (k_1, k_2) \cdot \mathit{Checkin}\ g_0\ r_0\ (k_0, k_1)
\end{array}
$$

This resembles the following situation in a hotel with one room r_0: **(1)** Joe (Guest g_0) checks in and gets a card (k_0, k_1). **(2)** Eve (Guest g_1) checks in and gets a card (k_1, k_2). **(3)** Joe checks in again and gets a card (k_2, k_3). **(4)** At this point, Joe has two cards for the room: He tries the newest card (k_2, k_3), but it does not open the door, so he gives it a try with the card from his last stay (k_0, k_1) which unlocks the door. Feeling safe in his room, he puts his wallet on the nightstand and goes to bed. **(5)** At night, Eve enters the room with card (k_1, k_2) and takes Joe's wallet. A subtle error in the key card system causes this jeopardy and can be resolved if Joe would have followed a reasonable safety policy, i.e., to only use his recent card. After understanding the counterexample and formulating this safety policy, one can prove the safety of the key card system.

8 Related work

The idea of specification-based testing was pioneered by the Haskell tool QuickCheck and has many descendants in interactive theorem provers, e.g., Agda/Alfa [9], ACL2 [10], Isabelle [2] and PVS [19], and in a variety of programming languages. QuickCheck uses test

data generators that create random values to test the propositions. Random testing can handle propositions with strong preconditions only very poorly. To circumvent this, the user must manually write a test data generator that only produces values that fulfill the precondition. SmallCheck [22] tests the propositions exhaustively for small values. It also handles propositions with strong preconditions poorly, but in practice handles preconditions better than QuickCheck because it gives preference to small values, and they tend to fulfill the commonly occurring preconditions more often. Lazy SmallCheck [22] uses partially-instantiated values and a refinement algorithm to simulate narrowing in Haskell. This is closely related to the work of Lindblad [14] and EasyCheck [7], based on the narrowing strategy in the functional logic programming language Curry [12]. This approach can cut the search space of possible values to check if partially instantiated values already violate the precondition. The three approaches, QuickCheck (without manual test data generators), SmallCheck and Lazy SmallCheck, have in common *black-box testing*, i.e., not considering the description of the precondition - they generate (partial) values and test the precondition.

Previous work [1] focused on the *verification* of the transformation of Horn clauses to functional programs, whereas the focus of this work is the extension and application of the transformation for *counterexample generation*. Our approach is a glass-box testing approach, i.e., it considers the description of the precondition and compiles a purely functional program that generates values that fulfill the precondition. Closely related to our work is the glass-box testing by Fischer and Kuchen [11] for functional logic programs, but they take advantage of narrowing and non-determinism features of Curry.

Another approach to finding values that fulfill the preconditions is to use a CLP(FD) constraint solver, as done by Carlier et al. [4]. Testing specifications using αProlog [6] is described by Cheney and Momigliano [5]. A completely different approach to finding counterexamples is translating the specification to propositional logic and invoking a SAT solver, as practiced by the Isabelle tools Refute [26] and Nitpick [3].

9 Conclusion

We described a counterexample generator that improves upon existing solutions by translating specifications into logic programs and which in turn are processed to functional programs, applying an enriched mode analysis. This counterexample generator is included in the next Isabelle release and can be invoked by Isabelle's users to validate their specifications before proving them correct.

Thus, we adopt mode analysis, a common technique from *logic programming*, and apply it in the context of *functional programming* for synthesizing test data generators. We employ the analysis in a compilation with an embedding of depth-limited non-deterministic computations in the functional language. Using these generators for preconditions allows us to find counterexamples in Isabelle specifications where type-based exhaustive and random testing have failed.

In the future, we would like to investigate counterexample generation via testing in (functional) logic programming languages, e.g., Curry, Mercury [24], αProlog [6] and XSB [20].

Acknowledgements

I would like to thank Andrei Popescua, Sascha Boehme, Tobias Nipkow, Alexander Krauss and the anonymous referees for comments on earlier versions of this paper.

References

1. Stefan Berghofer, Lukas Bulwahn, and Florian Haftmann. Turning inductive into equational specifications. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Proceedings of the 22th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2009)*, volume 5674 of *LNCS*, pages 131–146. Springer, 2009.
2. Stefan Berghofer and Tobias Nipkow. Random Testing in Isabelle/HOL. In *Proceedings of the Software Engineering and Formal Methods, Second International Conference (SEFM 2004)*, pages 230–239. IEEE Computer Society, 2004.
3. Jasmin Christian Blanchette and Tobias Nipkow. Nitpick: A counterexample generator for higher-order logic based on a relational model finder. In M. Kaufmann and L. Paulson, editors, *Interactive Theorem Proving (ITP 2010)*, volume 6172 of *LNCS*, pages 131–146. Springer, 2010.
4. Matthieu Carlier, Catherine Dubois, and Arnaud Gotlieb. Constraint Reasoning in FocalTest. In *5th International Conference on Software and Data Technologies (ICSOFT 2010)*, 2010.
5. James Cheney and Alberto Momigliano. Mechanized metatheory model-checking. In *PPDP '07: Proceedings of the 9th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 75–86. ACM, 2007.
6. James Cheney and Christian Urban. Alpha-Prolog: A Logic Programming Language with Names, Binding and Alpha-Equivalence. In *Proceedings of the International Conference on Logic Programming (ICLP 2004)*, volume 3132 of *LNCS*, pages 269–283, 2004.
7. Jan Christiansen and Sebastian Fischer. EasyCheck – Test Data for Free. In *Proceedings of the 9th International Symposium on Functional and Logic Programming (FLOPS 2008)*, volume 4989 of *LNCS*, pages 322–336. Springer, 2008.
8. Koen Claessen and John Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *Proceedings of International Conference on Functional Programming (ICFP)*, pages 268 – 279. ACM SIGPLAN, 2000.
9. Peter Dybjer, Qiao Haiyan, and Makoto Takeyama. Combining testing and proving in dependent type theory. In *Theorem Proving in Higher Order Logics*, pages 188–203, 2003.
10. Carl Eastlund. Doublecheck your theorems. In *Eighth International Workshop On The ACL2 Theorem Prover and Its Applications*, 2009.
11. Sebastian Fischer and Herbert Kuchen. Systematic generation of glass-box test cases for functional logic programs. In *Proceedings of the 9th ACM SIGPLAN international conference on Principles and practice of declarative programming*, PPDP '07, pages 63–74. ACM, 2007.
12. M. Hanus. Multi-paradigm declarative languages. In *Proceedings of the International Conference on Logic Programming (ICLP 2007)*, volume 4670 of *LNCS*, pages 45–75. Springer, 2007.
13. Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis.* The MIT Press, 2006.
14. Fredrik Lindblad. Property directed generation of first-order test data. In *The Eigth Symposium on Trends in Functional Programming*, 2007.
15. C. S. Mellish. The automatic generation of mode declarations for Prolog programs. Technical Report 163, Department of Artificial Intelligence, 1981.
16. Lee Naish. Adding equations to NU-Prolog. In *Proceedings of the 3rd Int. Symposium on Programming Language Implementation and Logic Programming*, LNCS, pages 15–26. Springer, 1991.
17. Tobias Nipkow. Verifying a Hotel Key Card System. In K. Barkaoui, A. Cavalcanti, and A. Cerone, editors, *Theoretical Aspects of Computing (ICTAC 2006)*, volume 4281 of *LNCS*. Springer, 2006. Invited paper.

18 David Overton, Zoltan Somogyi, and Peter J. Stuckey. Constraint-based mode analysis of mercury. In *Proceedings of the 4th ACM SIGPLAN international conference on Principles and practice of declarative programming*, PPDP '02, pages 109–120. ACM, 2002.

19 Sam Owre. Random testing in PVS. In *Workshop on Automated Formal Methods*, 2006.

20 Prasad Rao, Konstantinos Sagonas, Terrance Swift, David Warren, and Juliana Freire. XSB: A system for efficiently computing well-founded semantics. In Jürgen Dix, Ulrich Furbach, and Anil Nerode, editors, *Logic Programming And Nonmonotonic Reasoning*, volume 1265 of *LNCS*, pages 430–440. Springer, 1997.

21 Céline Rouveirol. Flattening and Saturation: Two Representation Changes for Generalization. *Mach. Learn.*, 14(2):219–232, 1994.

22 Colin Runciman, Matthew Naylor, and Fredrik Lindblad. Smallcheck and lazy smallcheck: automatic exhaustive testing for small values. In *Haskell '08: Proceedings of the first ACM SIGPLAN symposium on Haskell*, pages 37–48. ACM, 2008.

23 Jan-Georg Smaus, Patricia M. Hill, and Andy King. Mode analysis domains for typed logic programs. In *Selected papers from the 9th International Workshop on Logic Programming Synthesis and Transformation*, pages 82–101, London, UK, 2000. Springer.

24 Zoltan Somogyi, Fergus Henderson, and Thomas Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *The Journal of Logic Programming*, 29(1-3):17–64, December 1996.

25 Philip Wadler. How to replace failure by a list of successes. In *Proceedings of a conference on Functional programming languages and computer architecture*, pages 113–128. Springer, 1985.

26 Tjark Weber. Bounded model generation for Isabelle/HOL. In Wolfgang Ahrendt, Peter Baumgartner, Hans de Nivelle, Silvio Ranise, and Cesare Tinelli, editors, *Selected Papers from the Workshops on Disproving and the Second International Workshop on Pragmatics of Decision Procedures (PDPAR 2004)*, volume 125(3) of *Electronic Notes in Theoretical Computer Science*, pages 103–116. Elsevier, 2005.

27 Makarius Wenzel, Lawrence C. Paulson, and Tobias Nipkow. The Isabelle Framework. In Ait Mohamed, Munoz, and Tahar, editors, *Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2008)*, volume 5170 of *LNCS*, pages 33–38. Springer, 2008.

Declarative Output by Ordering Text Pieces

Stefan Brass

Martin-Luther-Universität Halle-Wittenberg, Institut für Informatik,
Von-Seckendorff-Platz 1, D-06099 Halle (Saale), Germany
`brass@informatik.uni-halle.de`

Abstract

Most real-world programs must produce output. If a deductive database is used to implement database application programs, it should be possible to specify the output declaratively. There is no generally accepted, completely satisfying solution for this. In this paper we propose to specify an output document by defining the position of text pieces (building blocks of the document). These text pieces are then ordered by their position and concatenated. This way of specifying output fits well to the bottom-up way of thinking about rules (from right to left) which is common in deductive databases. Of course, when evaluating such programs, one wants to avoid sorting operations as far as possible. We show how rules involving ordering can be efficiently implemented.

1998 ACM Subject Classification D.1.6 Logic Programming

Keywords and phrases Deductive Databases, Logic Programming, Declarative Output, Bottom-Up Evaluation, Order, Sorting, Implementation

Digital Object Identifier 10.4230/LIPIcs.ICLP.2011.151

1 Introduction

Currently, database application programs are usually developed in a combination of two or more languages, e.g. PHP for programming and SQL for the database queries and updates. SQL is declarative, but most languages used for the programming part are not.

The goal of deductive databases is that only a single, declarative language is used for programming *and* database tasks. The advantages of declarativity have been clearly shown in SQL: The productivity is higher (because the programs are shorter and there is no need to think about efficient evaluation), and new technology (parallel hardware and new data structures and algorithms) can be used for existing application programs without changing them (only the DBMS needs to be updated).

While in general, it might be difficult to reach acceptable performance for really declarative programs (Prolog is not completely declarative), database applications are quite special and usually not very difficult. For the most part, the task of an application program is to check the input and to generate an output document (e.g., a web page).

Although generating output is practically very important, it seems that there is no really good solution in logic programming yet. The standard solution in Prolog with a `write`-predicate is clearly non-declarative: It depends on the specific evaluation order used in Prolog.

The Gödel programming language, which improves upon Prolog in many ways, did also not solve this problem: "Gödels input/output facilities do not have a declarative semantics, so it is very important that input/output predicates are confined to as small a part of a program as possible." [4]. Certainly, it is a good advice to separate the generation of output documents from the more complex logic of the program. But at least for database

© Stefan Brass;
licensed under Creative Commons License ND

applications, the generation of output is a substantial part of the program, and deserves a declarative specification, too.

Another standard way is to use an accumulator pair: A prefix of the document to be generated is passed as an input argument to every predicate involved in generating output, and the current state of the document after the output of the predicate is returned. For instance, this solution is used in Mercury. The Mecury tutorial [1] contains this example:

```
main(IOState_in, IOState_out) :-
        io.write_string("Hello, ", IOState_in, IOState_1),
        io.write_string("World!", IOState_1, IOState_2),
        io.nl(IOState_2, IOState_out).
```

This is declarative, but has some problems, too:

1. It complicates the programs (many predicates have two additional arguments). In part, this problem can be solved by using special syntactic features to hide these arguments. E.g., Mecury has special state variables as a syntactic convenience [1] (only one argument is written instead of two, and the numbering of variables is done automatically).
2. Another problem is that if backtracking is possible, the actual output cannot be immediately done, and furthermore, the possibility remains that the program might sometimes produce alternative documents, which is certainly not expected. In the Mercury language, this problem is solved by checking the determinism of predicates which perform I/O, and using destructive input and unique output modes for the arguments [1].
3. This solution needs recursion already for simple tasks, e.g. printing the contents of a database table. As Molham Aref stated in his invited talk at last year's ICLP, one of the advantages of Datalog is that not so experienced users can be easily offered subsets of the language with restricted complexity. It seems unlucky that recursion is basically unavoidable if one uses this state-passing method for specifying output.
4. The situation is even worse, because one also needs lists or similar data structures: In databases, several answers are most naturally constructed as several solutions to a query (e.g. via backtracking). For instance, database tables are normally represented as sets of facts. This does not fit well with requiring deterministic code for output: One must use a predicate like `findall`, and then recurse over the resulting list. (Our proposal also contains lists, but many interesting programs do not need recursion, and do not need to inspect the constructed lists, i.e. no complex terms in body literals are needed. Furthermore, the use of lists can be hidden with some syntactic sugar.)

In functional programming, monads are used for declarative output [6, 5]. However, they depend on higher order programming, which is not common practice in logic programming. Therefore, monads are not easily understandable for logic programmers.

2 Basic Idea

The basic idea of our proposed solution is very simple: The Datalog program defines a predicate `output` with two arguments: The first argument determines the position in the output, and the second argument is a text piece. A simple example is:

```
output(1, 'Hello, ').
output(2, Name) ← name(Name).
output(3, '.').
name('Nina').
```

```
output([1], '<table>\n').
output([2], '<tr><th>Points</th><th>Student</th></tr>\n').
output([3,Points,Name,1], '<tr><td>')   ← homework(Name, Points).
output([3,Points,Name,2], Points)       ← homework(Name, Points).
output([3,Points,Name,3], '</td><td>')  ← homework(Name, Points).
output([3,Points,Name,4], Name)         ← homework(Name, Points).
output([3,Points,Name,5], '</td></tr>') ← homework(Name, Points).
output([4], '</table>\n').
homework('Ann', 5).
homework('Bob', 10).
homework('Chris', 10).
```

Figure 1 Generating an HTML Table with Homework Results

The generated output document can be computed as follows: One first uses standard bottom-up evaluation to generate all derivable facts of the form `output(P,T)`, then one sorts them by the first argument `P`, and prints the strings `T` in this sequence. Two comments must be made here:

1. Probably nobody likes to write the numbers for the output sequence, but syntactic sugar can be used to hide that (see Section 3). Basic Horn clauses are used to have a simple, common semantic framework, but different tasks might need different syntactic variants to specify the task clearly and concisely. For instance, definite clause grammar rules are a very useful notation in Prolog for syntax analysis tasks.
2. In the same way, there is not only one algorithm for the evaluation of rules, but different algorithms can be used for special cases of rules. In the above example, it would not be necessary to first generate all derivable `output`-facts and then sort them: Since the sorting argument is given explicitly in the rules, the rules might be applied in that order, and the text pieces immediately printed. But it is important to have a very simple evaluation algorithm for a directly executable semantics before discussing optimizations.

Of course, using only numbers as position specifications becomes impractical as soon as one wants to produce already slightly more complex documents. Therefore, we suggest to use a list as the first argument of `output`. Lists are sorted as usual: They are compared element by element, and the first position in which they differ decides the sequence. For instance, suppose we have stored homework points in a predicate `homework(Name, Points)`. An HTML table which contains the data sorted by points, and for equal points by name, can be generated as shown in Figure 1. Note that the sorting based on the data is for free if one uses this approach: Since the ID of the text piece contains first the points and then the name, these two sorting criteria are applied with the points having higher priority.

3 Syntactic Sugar

Of course, one wants to eliminate the ordering argument as far as possible. Furthermore, it should not be necessary to split a text whenever a parameter value must be inserted. Quotation marks and explicit commands for line breaks should be avoided as far as possible. With the pattern syntax which we propose, it is possible to write longer pieces of text, just as it appears in the output, and mark the places where something must be inserted.

```
homeworks_table(#
    <table>
    <tr><th>Points</th><th>Student</th></tr>
    <#homeworks_row>
    </table>
#).

homeworks_row([Points, Name]#
        <tr><td><$Points></td><td><$Name></td></tr>
#) ← homeworks(Name, Points).
```

Figure 2 Generating an HTML Table with Patterns

```
homeworks_table([1], '<table>\n').
homeworks_table([2], '<tr><th>Points</th><th>Student</th></tr>\n').
homeworks_table([3|X], Y) ← homeworks_row(X, Y).
homeworks_table([4], '</table>').

homeworks_row([Points, Name, 1], '<tr><td>').
homeworks_row([Points, Name, 2], Points).
homeworks_row([Points, Name, 3], '</td><td>').
homeworks_row([Points, Name, 4], Name).
homeworks_row([Points, Name, 5], '</td></tr>').
```

Figure 3 Internal Rules denoted by Patterns

Figure 2 shows the running example in this syntax, and Figure 3 shows the rules that are generated by the patterns.

Each pattern corresponds to a predicate. One writes the pattern/predicate name, then "`(`", then possibly a list as part of the position specification (if the pattern is instantiated multiple times), then a "`#`", then several lines which form the text of the pattern, then "`#)`", and then possibly a rule body. Within the pattern text, two special markers can be used: "`<#p>`" links to another pattern `p`, which is embedded here, and "`<$X>`" inserts the value of a variable which is bound in the rule body.

The generated predicates have two arguments just as the `output`-predicate (one can use pattern syntax for the `output`-predicate, too). The pattern text is internally split into as many pieces as needed, at least the inserted patterns and variables must be pieces of their own, but one can in addition split the text at line breaks. Each piece gets a sequential number. Then one fact or rule is generated for each piece:

1. The position consists of the optional list of variables (specified before the `#` in the first line of the pattern), then the piece number, and in case of an embedded pattern, the position argument of that pattern.
2. The second argument is either the text piece, or, if a variable is inserted at this point, the variable, or, if a pattern is embedded here, a variable for a text piece of that pattern.
3. If the pattern has a body, it is attached to each generated rule. If this position is for an embedded pattern, a call to the corresponding predicate is added to the body.

A predicate or pattern called "`output`" is the "main" predicate, which defines the overall output of the program (the example is only a part of the generation of an entire web page).

4 Efficient Evaluation

The goal is not having to do a lot of sorting when evaluating the rules. Sorting is a relatively expensive operation, and furthermore it is necessary to store intermediate results. Relational database management systems often work internally with tuple streams, where the next tuple is computed on demand only (using an iterator/cursor interface). In this way, the materialization of intermediate results is avoided. Only sorting needs intermediate storage, because the first result tuple can only be produced after the last input tuple was seen.

Often, already by choosing the right evaluation order of the rules, sorting can be avoided. Furthermore, if we can get the tuples for the predicates in the body in the right sorted sequence, this sorting can often be preserved. For the base predicates (the EDB predicates stored in database relations), sorting can sometimes be avoided, if they are stored in an ISAM file or index-organized table (if the required sorting sequence matches the key) or if a b-tree index with a fitting search key is available.

First note that the only way in which the predicate `output` should be treated specially is that the second argument is printed in the end, when the tuples (derived facts) are accessed ordered by the first argument. For instance, it should be possible to write

```
output([X, Y], Z) ← p(Z, X, Y).
```

Thus, in order to produce the tuples for `output` ordered by first argument, we need to generate the tuples for `p` ordered by second and third arguments (second argument with higher priority than third argument, i.e. the third argument is only important for determining the order of two tuples if the second argument is equal). So in general, we need to produce tuples for any predicate in the program ordered by any given sequence for the arguments.

In order to concentrate on the sorting problem, we make two simplifying assumptions in this paper:

- We do not consider recursive rules. There is a large body of work on evaluating recursive rules, but in most cases, sorting sequences for the predicates in the body do not give a sorting sequence for the derived tuples. Furthermore, in order to guarantee termination, the resulting tuples must be stored (materialized), so that duplicates can be eliminated. When this is anyway needed, one could of course choose a b-tree or similar data structure which gives the required sorting on output.
- While we need structured terms in the head of the rules (to build lists for the position specifications), we assume that the body literals contain only variables and constants.

4.1 Interface for Relations

As usual in relational DBMS, we use a tuple stream interface for relations (predicates), i.e. it is possible to open a cursor (scan, iterator) over the relation, which permits to loop over all tuples. We assume that for every normal predicate p, there is a class p`_cursor` with the following methods:

- `void open()`: Open a scan over the relation, i.e. place the cursor before the first tuple.
- `bool fetch()`: Move the cursor to the next tuple. This function must also be called to access the first tuple. It returns `true` if there is a first/next tuple, or `false`, if the cursor is at the end of the relation.
- T `col_`i`()`: Get the value of the i-th column (attribute) of the current tuple (T is the data type of this column). This interface permits to get access to the attribute values without actually materializing the tuple (especially, large data values do not need to be copied).

- `void close()`: Close the cursor.
- `p_cursor_pos position()`: Return current position of the cursor for a later call to `restore` (in order to return to this position). This is only used for complicated loop structures, and very seldom more than one position must be saved at the same time.
- `void restore(p_cursor_pos)`: Return to a previously saved position of the cursor. It is possible to return multiple times to the same position.

The objects of the class `p_cursor` do not guarantee any specific order of the returned tuples. Thus, we also need classes `p_cursor_`i_1`_..._`i_k, which return tuples sorted by argument i_1 with highest priority, i_2 with second highest priority, and so on.

Our goal in the section is to create code for such cursors for the derived predicates (IDB predicates), given such cursors for the base predicates (stored relations, EDB predicates). As mentioned above, the storage structures for the base predicates sometimes give the sorting more or less for free, otherwise some explicit sorting is unavoidable.

Again, in order to focus on the main problem (the sorting), and not to overload the paper with details already considered extensively in the literature, we ignore a very important optimization:

- Often, when a predicate is called, values are known for some arguments. Thus, one does not need to produce the entire extension of the predicate. One could extend the cursor interface by attaching a binding pattern to the cursor name (specifying which arguments are bound in the call, i.e. input arguments), and permit to specify values for them in the `open()` call. This is also interesting for base predicates with indexes.
- This situation is even more complicated, since some arguments are known at compile-time, others only at runtime. Furthermore, one might want to pass not only equality conditions for the arguments to the called predicate, but also other simple conditions (e.g. $\mathsf{X} \leq 5$). An extreme case in this direction is the SLDMagic-method of the author [2]. There is a large body of literature on Magic Sets that treats such problems.

4.2 Single Rule, Nested Loop Join

Consider the rule

$$\mathsf{p}(\mathsf{t}_1, \ldots, \mathsf{t}_n) \leftarrow \mathsf{B}_1 \wedge \cdots \wedge \mathsf{B}_m$$

and suppose an ordering of the derived tuples by arguments $i_1, \ldots, i_k$ is required.

Then we first determine a sequence of variables $\mathsf{X}_1, \ldots, \mathsf{X}_l$ that appear in the head such that the matches found for the body must be ordered in this sequence. These variables are simply the variables that appear in $\mathsf{t}_{i_1}, \ldots, \mathsf{t}_{i_k}$ in the order of first appearance. For instance, given the head

$$\mathsf{p}([a, \mathsf{X}, \mathsf{Y}, b], c, [\mathsf{Z}, \mathsf{Y}, \mathsf{X}])$$

and the sorting of the derived `p`-tuples by arguments $1, 2, 3$, the considered variable assignments must be sorted by values for $\mathsf{X}, \mathsf{Y}, \mathsf{Z}$ in this sequence.

The simplest implementation of the rule body is a nested loop join. It is well known that that an ordering of tuples in the outer loop is preserved by the nested loop join. This holds more generally. For instance, consider the rule body

$$\mathsf{q}(\mathsf{X}, \mathsf{Y}, a) \wedge \mathsf{r}(\mathsf{Y}, \mathsf{Z})$$

A nested loop join works as shown in Figure 4. Of course, the code shown in Figure 4 must be rewritten so that it fits itself in a `fetch()` method. Currently we would need coroutines

```
q_cursor_1_2 q;
r_cursor_2_1 r;
q.open();
while(q.fetch()) { // q(X, Y, a)
    if(q.col_3 == 'a') {
        int X = q.col_1();
        int Y = q.col_2();
        r.open();
        while(r.fetch()) { // r(Y, Z)
            if(r.col_1 == Y) {
                int Z = r.col_2();
                print(X, Y, Z); // Variable assignment satisfying body
            }
        }
        r.close();
    }
}
q.close();
```

Figure 4 Nested Loop Join for $\mathsf{q}(\mathsf{X}, \mathsf{Y}, a) \wedge \mathsf{r}(\mathsf{Y}, \mathsf{Z})$

(when a match found, instead of the `print`, one must do a "yield return", so the next call to `fetch()` starts at this place). But this restructuring of the code is a simple task for an able programmer. We have used the above code so that the nested loops are easily visible.

It is quite obvious that the program in Figure 4 produces the tuples ordered by the values for X ,Y, Z: Let $(\mathsf{X}_1, \mathsf{Y}_1, \mathsf{Z}_1)$ and $(\mathsf{X}_2, \mathsf{Y}_2, \mathsf{Z}_2)$ be two tuples that the above code yields, where the second tuple is produced later. Then there are two cases to consider:

- Suppose that the second tuple is produced in a different (later) iteration of the outer loop. The cursor over q guarantees that the tuples are considered sorted by first argument, and for equal first argument, by the second argument. There is no order for the third argument, but if we can assume that there are no duplicate tuples, the constant in the third argument means that tuples produced in different iterations of the loop must differ in at least one of the first two arguments. Therefore, we can conclude that either $\mathsf{X}_1 < \mathsf{X}_2$ or $\mathsf{X}_1 = \mathsf{X}_2$ and $\mathsf{Y}_1 < \mathsf{Y}_2$.
- Now suppose that both tuples are produced in the same iteration of the outer loop, but the second tuple is produced in a later iteration of the inner loop. Of course, we immediately get $\mathsf{X}_1 = \mathsf{X}_2$ and $\mathsf{Y}_1 = \mathsf{Y}_2$. Since the second argument is the main sorting criterion in the inner loop, it follows that $\mathsf{Z}_1 \leq \mathsf{Z}_2$.

Now, in general, suppose we need the variable assignments sorted by $\mathsf{X}_1, \ldots, \mathsf{X}_l$, and that the body $\mathsf{B}_1 \wedge \cdots \wedge \mathsf{B}_m$ is evaluated by nesting loops in this order, i.e. B_1 corresponds to the outermost loop and B_m to the innermost. Then the following condition must be satisfied for each $i = 1, \ldots, l$:

- Suppose that B_j is the first body literal, in which X_i appears.
- Then for given values for $\mathsf{X}_1, \ldots, \mathsf{X}_{i-1}$, there can be only a single match for $\mathsf{B}_1, \ldots, \mathsf{B}_{j-1}$, (i.e. a single assignment of tuples to literals such that the conditions of $\mathsf{B}_1, \ldots, \mathsf{B}_{j-1}$ are satisfied), and furthermore,
- the argument in which X_i appears in B_j is in the sorting specification for the cursor, preceded only by arguments in which one of $\{\mathsf{X}_1, \ldots, \mathsf{X}_{i-1}\}$ or a constant appears.

This ensures that the variable assignments are produced in the right order: Suppose this were not the case. Then there are tuples $(d_1, \ldots, d_n)$ and $(d'_1, \ldots, d'_n)$ of values for $\mathsf{X}_1, \ldots, \mathsf{X}_n$, where the second tuple is produced later and for some $i \in \{1, \ldots, n\}$: $d_1 = d'_1, \ldots, d_{i-1} = d'_{i-1}$, and $d_i > d'_i$. Let B_j be the first body literal in which X_i appears. Since for the given values $(d_1, \ldots, d_{i-1})$, there is only a single match for $\mathsf{B}_1 \wedge \cdots \wedge \mathsf{B}_{j-1}$, the outer loops are not switched forward between the two assignments, i.e. the problem occurs within a single run through the loop for B_j. But all arguments in B_j with higher sorting priority than the argument for X_i are filled with variables from $\{\mathsf{X}_1, \ldots, \mathsf{X}_{i-1}\}$ (which have the same value in both tuples), or constants (also the same value). Thus, the required ordering for B_j is violated.

4.3 More Interesting Loops

Of course, the above condition cannot always be reached by a permutation of the body literals. However, using only one loop per literal is not the only possibility.

Consider a rule with body

$$\mathsf{p}(\mathsf{X}, \mathsf{Y}) \wedge \mathsf{q}(\mathsf{Y}, \mathsf{Z})$$

and suppose we need the matching assignments ordered by X, Z. If the outer loop is over `p_cursor_1_2`, and the inner loop over `q_cursor_2_1`, we get a wrong sequence given the following predicate extensions:

p	
X	Y
1	3
1	4

q	
Y	Z
4	5
3	6

Result	
X	Z
1	6
1	5

Here the condition is violated that for every value for X, there is only a single match for $\mathsf{p}(\mathsf{X}, \mathsf{Y})$. The problem is that the value for Y from the first literal selects only specific facts for the second literal, and thus the ordering on Z in the second literal is not preserved, not even for a single X value.

However, it is possible to create a loop structure that preserves the ordering as required. The idea is to iterate with the outer loop only over different X-values, then to iterate with a middle loop over the second literal, and finally iterate with an innermost loop over the tuples of the first literal for a given X-value (i.e. an X-partition of p). For space reasons, the details cannot be shown here, but more information is available on the project web page.

4.4 Using Merge Joins, Explicit Sorting and Intermediate Storage

Of course, ordered tuple streams are also useful for merge joins, which can be much faster than a nested loop join. In general, the output of a merge join remains ordered only on the join columns, which sometimes might be just what is needed, but often not.

With techniques like shown above in the “interesting loops” example, one could do e.g. an outer loop over X-values, and inside do merge joins over Y. This is not as good as a single merge join, but also not as bad as a nested loop join (if there are several Y-values for a single X-value).

There are many ways to evaluate a given logic program, and especially there is always the option to require no order for the computation of the tuples for a predicate p, and then do an explicit sorting before the tuples are used. This would for instance give all options for merge joins in the evaluation of rules for p. Furthermore, if the tuples are used multiple

times (e.g. in an inner loop of a nested loop join), one has the advantage of computing them only once. The performance tests we did in [3] show that this becomes an important factor if the computation of the tuples of `p` is not easy. Thus, even if no explicit sorting is needed (because the tuples are produced in the right order using the techniques shown above), it might still be useful to materialize the tuples for some intermediate predicates. In general, one would use a cost-based optimizer which generates a number of alternative evaluation plans, estimates their costs, and chooses the cheapest.

4.5 Multiple Rules for a Predicate

So far, we have only considered the evaluation of a single rule. Of course, there are often several rules about a predicate.

Again, the goal is to produce the tuples sorted by given arguments. In the most general case, we implement each rule as shown above, so that each rule produces the tuples in the required order, and then merge the tuple streams produced by the rules (i.e. we compare the current tuple of every rule, output the smallest, and fetch the next tuple from that rule).

However, often by analyzing the rule heads, we can see that all tuples produced by one rule must come before all tuples produced by another rule. In that case we can of course evaluate one rule first, and then the other. I.e. the explicit merging at runtime may be avoided or reduced, because we need to compare only tuples from rules where the order cannot be determined already at compile time.

Let us again consider the example from Figure 1. From the first list element in the head it is clear, that we must first apply the first rule, then the second, then the following block of rules, and finally the last rule.

Now the block of rules with first list element 3 is an interesting case, because they all have the same body, and each of the rules has the same variables in position 2 and 3 in the list, and position 4 is again a constant which can be ordered at compile-time. Thus, when we found one match for `homework(Name, Points)`, we can output five tuples in the order of the rules. Of course, the tuples for `homework` must be produced in sorted order, with the second argument sorted with higher priority.

Note that it is not required that the rule bodies are exactly equal. What is required is that we can get a superset for the possible values for the variables in the rule head in sorted order. Since after the variables, distinct constants follow, we can give each rule a chance in turn, whether it wants to produce facts with the given values for the variables. For instance, it would be no problem if the rule bodies contained further literals besides the common literal which determines the set of values for the variables `Points` and `Name`.

Of course, in general, the special treatment of the list-valued positioning argument should not be done only for the predicate `output`. Thus, the compiler should try to determine intervals of possible values for ordering arguments. If the intervals for two rules about a predicate do not overlap, the sequence of rule applications is again clear, and no merging at runtime is needed.

4.6 Avoiding List Construction, Example

One final point for the efficient evaluation of the `output`-rules is that the positioning argument is seldom explicitly needed, and one wants of course to avoid constructing the list terms if at all possible. In most cases, computing tuples in the right sequence and knowing intervals for the possible values is all that is needed. This is especially clear if the lists appear only in the

```
main()
{
    print("<table>\n");
    print("<tr><th>Points</th><th>Student</th></tr>\n");
    homework_cursor_2_1 h;
    h.open();
    while(h.fetch()) {
        print("<tr><td>");
        print(h.col_2()); // Points
        print("</td><td>");
        print(h.col_1()); // Name
        print("</td></tr>");
    }
    h.close();
    print("</table>\n");
}
```

Figure 5 Implementation for the Example from Figure 1

rules about `output`, or in rules about predicates which have a unique position in the final output (i.e. no explicit sorting or merging is needed for the resulting tuples).

Putting all together, we arrive at the natural program for the example shown in Figure 5. Of course, for `output`, we do not generate a cursor, but directly print the second argument of the derived facts.

5 Conclusions

In this paper, we made a simple proposal for specifying output declaratively in deductive databases. An important feature in these languages is that one thinks from right to left, i.e. in the natural direction of the rules (corresponding to bottom-up evaluation). Therefore output, updates, and other results of program execution seem to be specified most natural in the rule heads. Furthermore, it seems normal that literals have several solutions, which all need to be printed — no backtracking should be needed for this (backtracking is not really a concept of bottom-up evaluation: it is always implicitly set-oriented).

Although in its pure form, specifying output in this way is not very convenient, with the proposed pattern syntax, it seems quite reasonable. Of course, more can and should be done for special output problems (e.g., currently, putting a comma between list elements, but not at the end needs a self join to check whether there is still another element).

In the second half of the paper, we showed that the rules can often be evaluated without explicit sorting: In the end, the proposed solution should be as efficient as a standard, imperative program (regarding output). Therefore, it is good to see that the declarative specification can be translated into a C++ program, which does only as much sorting as the data really requires and any program must do.

The web page [`http://www.informatik.uni-halle.de/~brass/output/`] contains a small prototype program that does bottom-up evaluation and computes the output (currently without optimizations) and a few examples for interesting loop structures.

References

1 Ralph Becket. Mercury tutorial. Technical report, University of Melbourne, Dept. of Computer Science, 2010.
`http://www.mercury.csse.unimelb.edu.au/information/papers/book.pdf`.

2 Stefan Brass. SLDMagic — the real magic (with applications to web queries). In W. Lloyd et al., editors, *First International Conference on Computational Logic (CL'2000/DOOD'2000)*, number 1861 in LNCS, pages 1063–1077, Heidelberg, Berlin, 2000. Springer.

3 Stefan Brass. Implementation alternatives for bottom-up evaluation. In *International Conference on Logic Programming (ICLP'10): Technical Communications*, LIPIcs. Schloss Dagstuhl, 2010.

4 Patricia M. Hill and John W. Lloyd. *The Gödel Programming Language*. MIT Press, Cambridge, Massachusetts, 1994.

5 Simon Peyton Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign language calls in haskell. In Tony Hoare, Manfred Nroy, and Ralf Steinbruggen, editors, *Engineering theories of software construction*, pages 47–96, Amsterdam, 2000. IOS Press. Updated version available at:
`http://research.microsoft.com/en-us/um/people/simonpj/papers/marktoberdorf/`.

6 Philip Wadler. How to declare an imperative. *ACM Computing Surveys*, 29(3):240–263, 1997. See also: `http://homepages.inf.ed.ac.uk/wadler/topics/monads.html`.

Transaction Logic with Defaults and Argumentation Theories *

Paul Fodor and Michael Kifer

Stony Brook University
Stony Brook, NY 11794, USA
pfodor,kifer@cs.stonybrook.edu

Abstract

Transaction Logic is an extension of classical logic that gracefully integrates both declarative and procedural knowledge and has proved itself as a powerful formalism for many advanced applications, including modeling robot movements, actions specification, and planning in artificial intelligence. In a parallel development, much work has been devoted to various theories of defeasible reasoning. In this paper, we unify these two streams of research and develop Transaction Logic with Defaults and Argumentation Theories, an extension of both Transaction Logic and the recently proposed unifying framework for defeasible reasoning called Logic Programs with Defaults and Argumentation Theories. We show that this combination has a number of interesting applications, including specification of defaults in action theories and heuristics for directed search in artificial intelligence planning problems. We also demonstrate the usefulness of the approach by experimenting with a prototype of the logic and showing how heuristics expressed as defeasible actions can significantly reduce the search space as well as execution time and space requirements.

1998 ACM Subject Classification I.2.3 Artificial Intelligence. Deduction and Theorem Proving and Knowledge Processing. D.3.3 Programming Languages. Language Constructs and Features frameworks

Keywords and phrases Transaction Logic, Defeasible reasoning, Well-founded models

Digital Object Identifier 10.4230/LIPIcs.ICLP.2011.162

1 Introduction

Transaction logic (abbr., $\mathcal{TR}$) [5, 2] is a general logic for representing knowledge base dynamics. Its model and proof theories cleanly integrate declarative and procedural knowledge and the logic has been employed in domains ranging from reasoning about actions [3], to knowledge representation [1], AI planning [5], workflow management and Web services [14], and general knowledge base programming [4]. Defeasible reasoning is another important paradigm, which has been extensively studied in knowledge representation, policy specification, regulations, law, learning, and more [6, 11, 12].

In this paper we propose to combine $\mathcal{TR}$ with defeasible reasoning and show that the resulting logic language has many important applications. This new logic is called *Transaction Logic with Defaults and Argumentation Theories* (or $\mathcal{TR}^{DA}$) because it extends $\mathcal{TR}$ in the direction of the recently proposed unifying framework for defeasible reasoning called *logic programming with defaults and argumentation theories* (LPDA) [17]. Along the way we define

* This work is part of the SILK (Semantic Inference on Large Knowledge) effort within Project Halo, sponsored by Vulcan Inc. It was also partially supported by the NSF grant 0964196.

© Paul Fodor and Michael Kifer;
licensed under Creative Commons License NC-ND

a well-founded semantics [16] for $\mathcal{TR}$, which, to the best of our knowledge, has never been done before.

We show that the combined logic enables a number of interesting applications, such as specification of defaults in action theories and heuristics for pruning search in search-intensive applications such as planning. We also demonstrate the usefulness of the approach by experimenting with a prototype of $\mathcal{TR}^{DA}$ and showing that heuristics expressed as defeasible actions can drastically prune the search space together with the execution time and space requirements.

This paper is organized as follows. Section 2 motivates reasoning with defaults in $\mathcal{TR}$ with an example. Section 3 provides background on Transaction Logic to make the paper self-contained. Section 4 extends $\mathcal{TR}$ by incorporating defeasible reasoning. Section 5 specializes the logic developed in Section 4 by defining a useful argumentation theory that extends Generalized Courteous Logic Programs (GCLP) [12] and Section 7 summarizes the paper and outlines future work.

2 Motivating Example

In this section, we give an example that illustrates the advantages of extending Transaction Logic with defeasible reasoning.

The syntax of $\mathcal{TR}^{DA}$ is similar to that of standard logic programming except for the fact that literals in the rule bodies are connected via the *serial conjunction*, $\otimes$, which specifies an order of action execution. For instance, $pickup(block1) \otimes puton(block1, block2)$ says that the action $pickup(block1)$ is to be executed first and the action $puton(block1, block2)$ second. The set of predicate symbols of the program is partitioned into:

- a set of *fluents*, which are facts stored in database states or derived propositions that do not change the state of the database; and
- a set of *actions*, which represent actions that change those states.

In addition to the user defined predicate symbols, there are built-in actions called *elementary transitions* for basic manipulation of states. These include *delete(f)* and *insert(f)* for every ground fluent f. Examples of such elementary transitions include $delete(on(block1, block0)$ and $insert(clear(block0)$.

As usual in defeasible reasoning, rules in $\mathcal{TR}^{DA}$ can be *tagged* with terms. For instance, the *move* rule in the example below is tagged with the term $mv_rule(Block, To)$. The predicate **`!opposes`** is used to specify that some rules are incompatible with others. The predicate **`!overrides`** specifies that some actions have higher priority than other actions.

Following the standard convention in Logic Programming, we will be using alphanumeric symbols that begin with an uppercase letter to denote variables. Alphanumeric symbols that begin with lowercase letters will denote constant, function, and predicate symbols.

▶ Example 2.1 (Block world planning). This example illustrates the use of defeasible reasoning for heuristic optimization of planning in the blocks world. The $\mathcal{TR}^{DA}$ program below is designed to build pyramids of blocks that are stacked on top of each other so that smaller blocks are piled up on top of the bigger ones. The construction process is non-deterministic and several different blocks can be chosen as candidates to be stacked on top of the current partial pyramid. The heuristic uses defeasibility to give priority to larger blocks so that higher pyramids would tend to be constructed.[1]

[1] For more information on planning with $\mathcal{TR}$ see [5].

In this example, we represent the blocks world using the fluents $on(x, y)$, which say that block x is on top of block y; $isclear(x)$, which says that nothing is on top of block x; and $larger(x, y)$, which says that the size of x is larger than the size of y. The action $pickup(X, Y)$ lifts the block X from the top of block Y and the action $putdown(X, Y)$ puts it down on top of block Y. These actions are specified by the second and third rules, respectively. The action $move(X, From, To)$, specified by the first rule, moves block X from its current position on top of block $From$ to a new position on top of block To. This action is defined by combining the aforementioned actions *pickup* and *putdown*, if certain preconditions are satisfied. The stacking action (not included in the program) then uses the *move* action to construct pyramids.

The key observation here is that at any given point several different instances of the rule tagged with *move_action* might be applicable and several different moves might be performed. The predicate !**opposes** stipulates that two different move-actions for different block are considered to be in conflict (because only one action at a time is allowed).

```
@mv_rule(Block,To) move(Block,From,To) :-
        (on(Block,From) ∧ larger(To,Block)) ⊗
        pickup(Block,From) ⊗ putdown(Block,To).
pickup(X,Y) :- (isclear(X) ∧ on(X,Y)) ⊗
        delete(on(X,Y)) ⊗ insert(isclear(Y)).
putdown(X,table) :- (isclear(X) ∧ not on(X,Z))
        ⊗ insert(on(X,table)).
putdown(X,Y) :- (isclear(X) ∧ isclear(Y) ∧ not on(X,Z))
        ⊗ delete(isclear(Y)) ⊗ insert(on(X,Y)).
!opposes(move(B1,F1,T1),move(B2,F2,T2)) :- B1 ≠ B2.
```

Various heuristics can be used to improve construction of plans for building pyramid of blocks. In particular, we can use preferences among the rules to cut down on the number of plans that need to be looked at. For instance, the following rule says that move-actions that move bigger blocks are preferred to move-action that move smaller blocks—unless the blocks are moved down to the table surface.

```
!overrides(mv_rule(B2,To), mv_rule(B1,To)) :- larger(B2,B1) ∧ To ≠ table.
```

Consider the following configuration of blocks:

```
on(blk1,blk4). on(blk2,blk5). on(blk3,table). on(blk4,table).
on(blk5,table). isclear(blk1). isclear(blk2). isclear(blk3).
larger(blk2,blk1). larger(blk3,blk1). larger(blk3,blk2).
larger(blk4,blk1). larger(blk5,blk2). larger(blk2,blk4).
```

Although, both *blk*1 and *blk*2 can be moved on top of *blk*3, moving *blk*2 has higher priority because it is larger.

For moving blocks to the table surface, we use the opposite heuristic, one which prefers unstacking smaller blocks:

```
!overrides(mv_rule(B2,table), mv_rule(B1,table)) :- larger(B1,B2).
```

In our example, this makes unstacking `blk1` and moving it to the table surface preferable to unstacking `blk2`, since the former is a smaller block. This blocks the opportunity to then move `blk4` on top of `blk2` and subsequently put `blk1` on top of `blk4`. These preference rules can be applied to a pyramid-building program like this:

```
stack(0,Block).
stack(N,X) :- N>0 ⊗ move(Y,_,X) ⊗ stack(N-1,Y) ⊗ on(Y,X).
stack(N,X) :- (N>0 ∧ on(Y,X)) ⊗ unstack(Y) ⊗ stack(N,X).
unstack(X) :- on(Y,X) ⊗ unstack(Y) ⊗ unstack(X).
unstack(X) :- isclear(X) ∧ on(X,table).
unstack(X) :- (isclear(X) ∧ on(X,Y) ∧ Y≠ table) ⊗ move(X,_,table).
unstack(X) :- on(Y,X) ⊗ unstack(Y) ⊗ unstack(X).
```

Running this program by the interpreter described in [9] shows that the above preferences drastically reduce the number of plans that need to be considered—sometimes to just one plan. These experiments are described in Section 6. □

3 Serial-Horn Transaction Logic

In this section we describe a subset of Transaction Logic called serial-Horn $\mathcal{TR}$. This subset has been shown to be sufficiently expressive for many applications, including planning, workflow management, and action languages [5].

The syntax of $\mathcal{TR}$ is derived from that of standard logic programming. The alphabet of the language $\mathcal{L}_{\mathcal{TR}}$ of $\mathcal{TR}$ contains an infinite number of constants, function symbols, predicate symbols, and variables. The *atomic formulas* have the form $p(t_1, ..., t_n)$, where p is a predicate symbol, and t_i are terms (variables, constants, function terms). However, unlike standard logic programming, predicate symbols are partitioned into *fluents* and *actions*. Fluents are predicates whose execution does not change the state of the database, while actions are predicates that can change the state of the database. Fluents are further partitioned into *base fluents* and *derived fluents*. Base fluents correspond to the classical base predicates in relational databases; they represent stored data and may be inserted or deleted. Derived fluents correspond to derived predicates, which represent database views. An atomic formula $p(t_1, ..., t_n)$ will be also called a *fluent* or an *action atomic formula* depending on whether p is a fluent or an action symbol. Furthermore, if p is a derived or base fluent symbol then $p(t_1, ..., t_n)$ is said to be a derived or base fluent atomic formula. An expression is *ground* if it does not contain any variables.

The symbol `neg` will be used to represent the explicit negation (also called "strong" negation) and `not` will be used for default negation, that is, negation as failure. A *fluent literal* is either an atomic fluent or has one of the negated forms: `neg` α, `not` α, `not neg` α, where α is a fluent atomic formula. An *action literal* is an action atomic formula or has the form `not` α, where α is a action atomic formula. Literals of the form `neg` α, where α is an action, are not allowed. Atoms of the form `neg not` *alpha* are also not allowed.

A *database state* is a set of ground base fluents. All database states are assumed to be *consistent*, meaning that they cannot have both f and `neg` f, for any base fluent f.

Transaction Logic distinguishes a special sort of actions, called *elementary transitions* or *elementary updates*. Intuitively, an elementary transition is a "builtin" action that transforms a database from one state into another. All other actions are defined via rules using elementary transitions and fluents. In this paper, elementary transitions are deletions and insertions of base fluents. Formally, an *elementary state transition* is an action atomic formula of the form *insert*(f) or *delete*(f), where f is a ground base fluent or has the form `neg` g, where g is a ground base fluent. For any given database state $\mathbf{D}$,

- *insert*(f) causes a transition from $\mathbf{D}$ to the state $\mathbf{D} \cup \{f\} \setminus \{\texttt{neg}\, f\}$; and
- *delete*(f) causes a transition from $\mathbf{D}$ to $\mathbf{D} \setminus \{f\} \cup \{\texttt{neg}\, f\}$.

In addition to the classical connectives and quantifiers, $\mathcal{TR}$ has new logical connectives:

- $\otimes$ - the sequential conjunction
- $\Diamond$ - the modal operator of hypothetical execution

The formula $\phi \otimes \psi$ represents an action composed of an execution of ϕ followed by an execution of ψ, while the formula $\Diamond\phi$ is an action of *hypothetically* testing whether ϕ can be executed at the current state, but no actual state changes takes place. In procedural terms, executing $delete(on(blk1, table)) \otimes insert(on(blk1, blk2))$ means "first delete $on(blk1, table)$ from the database, and then insert $on(blk1, blk2)$." The current database state changes as a result. In contrast, $\Diamond move(blk1)$ is only a "hypothetical" execution: it checks whether $move(blk1)$ can be executed in the current state, but regardless of whether it can or not the current state does not change.

The semantics of Transaction Logic is such that when f_1 and f_2 are fluents, $f_1 \otimes f_2$ is equivalent to $f_1 \wedge f_2$ and $\Diamond f$ to f. Therefore, when no actions are present, $\mathcal{TR}$ reduces to classical logic. This also explains our use of $\wedge$ in Example 2.1 where it could have been replaced with $\otimes$ without changing the meaning (but, the uses of $\otimes$ in the Example 2.1 *cannot* be replaced with $\wedge$ without changing the meaning).

▶ **Definition 1** (Serial goal). Serial goals are defined recursively as follows:

- If P is a fluent or an action literal then P is a serial goal. Note that fluent literals can contain both `not` and `neg`, and action literals can contain `not`.
- If P is a serial goal, then so are `not` P and $\Diamond P$.
- If P_1 and P_2 are serial goals then so are $P_1 \otimes P_2$ and $P_1 \wedge P_2$. ◻

▶ **Definition 2** (Serial rules). A **serial rule** is an expression of the form: $H :- B.$ where H is a `not`-free literal and B is a serial goal. We will be dealing with two classes of serial rules:

- **Fluent rules:** In this case, H is a derived fluent of the form f or a fluent literal of the form `neg` f and $B = f_1 \otimes ... \otimes f_n$, where each f_i is a fluent literal (and thus $\otimes$ could be replaced with $\wedge$).
- **Action rules:** In this case, H must be an atomic action formula, while the body of the rule, B, is a *serial goal*.

A ***transaction base*** is a finite set of serial rules. ◻

An existential serial goal is a statement of the form $\exists\bar{X}\psi$ where ψ is a serial goal and $\bar{X}$ is a list of all free variables in ψ. For instance, $\exists X move(X, blk2)$ is an existential serial goal. Informally, the truth value of an existential goal in $\mathcal{TR}$ is determined over sequences of states, called *execution paths*, which makes it possible to view truth assignments in $\mathcal{TR}$'s models as executions. If an existential serial goal, ψ, defined by a program $\mathbf{P}$, evaluates to true over a sequence of states $\mathbf{D_0}, \ldots \mathbf{D_n}$, we say that it can *execute* at state $\mathbf{D_0}$ by passing through the states $\mathbf{D_1}$, ..., $\mathbf{D_{n-1}}$, and ending in the final state $\mathbf{D_n}$. Formally, this is captured by the notion of *executional entailment*, which is written as: $\mathbf{P}, \mathbf{D_0}, \ldots \mathbf{D_n} \models \psi$. Further details on $\mathcal{TR}$ can be found in [5] and [2].

4 Defeasibility in Transaction Logic

In this section we define a form of defeasible Transaction Logic, which we call *Transaction logic with defaults and argumentation theories* ($\mathcal{TR}^{DA}$). The development was inspired by our earlier work on logic programming with argumentation theories, which did not support actions [17]. Language-wise, the only difference between $\mathcal{TR}^{DA}$ and serial $\mathcal{TR}$ is that the rules in $\mathcal{TR}^{DA}$ are tagged.

▶ **Definition 3** (Tagged rules)**.** A **tagged rule** in the language $\mathcal{TR}^{DA}$ is an expression of the form: $@r\ H\ :-\ B.$ where the **tag** r of a rule is a term. The head literal, H, and the body of the rule, B, have the same restrictions as in Definition 2.
A serial $\mathcal{TR}^{DA}$ ***transaction base*** $\mathbf{P}$ is a set of rules, which can be **strict** or **defeasible**. □

▶ **Definition 4** (Transaction formula)**.** A **transaction formula** in the language $\mathcal{TR}^{DA}$ is a literal, a serial goal, a tagged or an untagged serial rule. □

We note that the rule tag in the above definition is not a rule identifier: several rules can have the same tag, which can be useful for specifying priorities among sets of rules.

Strict rules are used as *definite* statements about the world. In contrast, defeasible rules represent *defaults* whose instances can be "defeated" by other rules. Inferences produced by the defeated rules are "overridden." We assume that the distinction between strict and defeasible rules is specified in some way: either syntactically or by means of a predicate (in this paper, we consider strict rules to be non-tagged rules, as in Definition 2).

▶ **Definition 5** (Rule handle)**.** Given a tagged rule, the term $\texttt{handle}(r, H)$ is called the ***handle*** of that rule. □

$\mathcal{TR}^{DA}$ transaction bases are used in conjunction with *argumentation theories*, which are sets of rules that define conditions under which some rule instances in the transaction base may be defeated by other rules. The argumentation theory and the transaction base share the same set of fluent and action symbols.

▶ **Definition 6** (Argumentation theory)**.** An **argumentation theory**, AT, is a set of strict serial rules. We also assume that the language of $\mathcal{TR}^{DA}$ includes a unary predicate, $\texttt{\$defeated}_{AT}$, which may appear in the heads of some rules in AT but not in the transaction base. A $\mathcal{TR}^{DA}$ $\mathbf{P}$ is said to be **compatible** with AT if $\texttt{\$defeated}_{AT}$ does not appear in any of the rule heads in $\mathbf{P}$, □

The rules AT are used to specify how the rules in $\mathbf{P}$ get defeated. This is usually done using special predicates defined in $\mathcal{TR}^{DA}$, such as `!`**`opposes`** and `!`**`overrides`** used in our example. For the purpose of defining the semantics, we assume that the argumentation theories AT are grounded. This grounding can be done by appropriately instantiating the variables and meta-predicates in AT.

Although Definition 6 imposes almost no restrictions on the predicate $\texttt{\$defeated}_{AT}$, practical argumentation theories are likely to require that it is executed hypothetically, i.e., that its execution does not change the current state. This is certainly true of the argumentation theories used in this paper.

▶ **Definition 7** (Herbrand universe and base)**.** The **Herbrand universe** of $\mathcal{TR}^{DA}$, denoted $\mathcal{U}$, is the set of all ground terms built using the constants and function symbols of the language of $\mathcal{TR}^{DA}$. The **Herbrand base**, denoted $\mathcal{B}$, is the set of all ground `not`-free literals that can be constructed using the language of $\mathcal{TR}^{DA}$. □

The key concept underlying the semantics of $\mathcal{TR}$ and $\mathcal{TR}^{DA}$ is that of *execution paths*, which are sequences of database states. The truth assignment in $\mathcal{TR}$ is done using *path structures*, which are mappings from paths of states.

▶ **Definition 8** (Path and Split). A **path** of length k, or a *k-path*, is a finite sequence of states, $\pi = \langle \mathbf{D}_1 \ ... \ \mathbf{D}_k \rangle$, where $k \geq 1$. A **split** of π is any pair of subpaths, π_1 and π_2, such that $\pi_1 = \langle \mathbf{D}_1 \ ... \ \mathbf{D}_i \rangle$ and $\pi_2 = \langle \mathbf{D}_i \ ... \ \mathbf{D}_k \rangle$ for some i $(1 \leq i \leq k)$. If π has a split π_1, π_2 then we write $\pi = \pi_1 \circ \pi_2$. ☐

We extend the well-founded semantics for logic programing [16] to $\mathcal{TR}^{DA}$ using a definition in the style of [13]. In the following, we use the usual three truth values $\mathbf{t}$, $\mathbf{f}$, and $\mathbf{u}$, which stand for *true*, *false*, and *undefined*, respectively. We also assume the existence of the following total order on these values: $\mathbf{f} < \mathbf{u} < \mathbf{t}$.

▶ **Definition 9** (Partial Herbrand interpretation). A **partial Herbrand interpretation** is a mapping $\mathcal{H}$ that assigns $\mathbf{f}$, $\mathbf{u}$, or $\mathbf{t}$ to every formula L in $\mathcal{B}$. A partial Herbrand interpretation $\mathcal{H}$ is ***consistent relative to an atomic formula*** L if it is not the case that $\mathcal{H}(L) = \mathcal{H}(\mathtt{neg}\, L) = \mathbf{t}$. $\mathcal{H}$ is ***consistent*** if it is consistent relative to every formula. $\mathcal{H}$ is ***total*** if, for every ground $\mathtt{not}$-free formula L (other than $\mathbf{u}$), either $\mathcal{H}(L) = \mathbf{t}$ and $\mathcal{H}(\mathtt{neg}\, L) = \mathbf{f}$ or $\mathcal{H}(L) = \mathbf{f}$ and $\mathcal{H}(\mathtt{neg}\, L) = \mathbf{t}$. ☐

Partial Herbrand interpretations are used to define *path structures*, which tell which ground atoms (fluents or actions) are true on what paths. Path structures play the same role in $\mathcal{TR}^{DA}$ as that played by the classical semantic structures in classical logic. The semantic structures of $\mathcal{TR}^{DA}$ are *mappings* from paths to partial Herbrand interpretations.

▶ **Definition 10** (Herbrand Path Structure). A **partial Herbrand Path Structure** is a mapping $\boldsymbol{I}$ that assigns a partial Herbrand interpretation to every **path** subject to the following restrictions:

1. For every ground base fluent-literal d and every database state $\mathbf{D}$:
   $\boldsymbol{I}(\langle \mathbf{D} \rangle)(d) = \mathbf{t}$, if $d \in \mathbf{D}$;
   $\boldsymbol{I}(\langle \mathbf{D} \rangle)(d) = \mathbf{f}$, if $d \notin \mathbf{D}$;
   $\boldsymbol{I}(\langle \mathbf{D} \rangle)(d) = \mathbf{u}$, otherwise
2. $\boldsymbol{I}(\langle \mathbf{D}_1, \mathbf{D}_2 \rangle)(insert(p)) = \mathbf{t}$ *if* $\mathbf{D}_2 = \mathbf{D}_1 \cup \{p\} \setminus \{\mathtt{neg}\, p\}$ and p is a ground fluent-literal;
   $\boldsymbol{I}(\langle \mathbf{D}_1, \mathbf{D}_2 \rangle)(insert(p)) = \mathbf{f}$, otherwise.
3. $\boldsymbol{I}(\langle \mathbf{D}_1, \mathbf{D}_2 \rangle)(delete(p)) = \mathbf{t}$ *if* $\mathbf{D}_2 = \mathbf{D}_1 \setminus \{p\} \cup \{\mathtt{neg}\, p\}$ and $\mathbf{P}$ is a ground fluent-literal;
   $\boldsymbol{I}(\langle \mathbf{D}_1, \mathbf{D}_2 \rangle)(delete(p)) = \mathbf{f}$, otherwise. ☐

Without loss of generality, while defining the semantics of $\mathcal{TR}^{DA}$ we will consider ground rules only. This is possible because all variables in a rule are considered to be universally quantified, so such rules can be replaced with the set of all of their ground instantiations.

We assume that the language includes the special propositional constants: $\mathbf{u}^{\pi}$ and $\mathbf{t}^{\pi}$, for each path π. Informally, $\mathbf{t}^{\pi}$ is a propositional transaction that is true precisely over the path π and false on all other paths; $\mathbf{u}^{\pi}$ is a propositional transaction that has the value $\mathbf{u}$ over π and is false on all other paths.

▶ **Definition 11** (Truth valuation in path structures). Let $\boldsymbol{I}$ be a path structure, π a path, L a ground $\mathtt{not}$-free literal, and let F, G ground serial goals We define **truth valuations** with respect to the path structure $\boldsymbol{I}$ as follows:

- If p is a $\mathtt{not}$-free literal then $\boldsymbol{I}(\pi)(p)$ is already defined because $\boldsymbol{I}(\pi)$ is a Herbrand interpretation, by definition of $\boldsymbol{I}$.

- For any path π:
  $\boldsymbol{I}(\pi)(\mathbf{t}^\pi) = \mathbf{t}$ and $\boldsymbol{I}(\pi')(\mathbf{t}^\pi) = \mathbf{f}$, if $\pi' \neq \pi$;
  $\boldsymbol{I}(\pi)(\mathbf{u}^\pi) = \mathbf{u}$ and $\boldsymbol{I}(\pi')(\mathbf{u}^\pi) = \mathbf{f}$, if $\pi' \neq \pi$.
- If ϕ and ψ are serial goals, then $\boldsymbol{I}(\pi)(\phi \otimes \psi) = max\{\, \texttt{min}(\boldsymbol{I}(\pi_1)(\phi), \boldsymbol{I}(\pi_2)(\psi)) | \pi = \pi_1 \circ \pi_2\}$
  .
- If ϕ and ψ are serial goals then $\boldsymbol{I}(\pi)(\phi \wedge \psi) = \texttt{min}(\boldsymbol{I}(\pi)(\phi), \boldsymbol{I}(\pi)(\psi))$.
- If ϕ is a serial goal then $\boldsymbol{I}(\pi)(\texttt{not}\ \phi) = \sim \boldsymbol{I}(\pi)(\phi)$, where $\sim \mathbf{t} = \mathbf{f}$, $\sim \mathbf{f} = \mathbf{t}$, and $\sim \mathbf{u} = \mathbf{u}$.
- If ϕ is a serial goal and $\pi = \langle \mathbf{D} \rangle$, where $\mathbf{D}$ is a database state, then
  $\boldsymbol{I}(\pi)(\Diamond\phi) = \texttt{max}\{\boldsymbol{I}(\pi')(\phi) \mid \pi'$ is a path that starts at $\mathbf{D}\}$
  $\boldsymbol{I}(\pi)(\Diamond\phi) = \mathbf{f}$, otherwise.
- For a strict serial rule F **:-** G,
  $\boldsymbol{I}(\pi)(F \textbf{:-} G) = \mathbf{t}$ iff $\boldsymbol{I}(\pi)(F) \geq \boldsymbol{I}(\pi)(G)$.
- For a defeasible rule $@r$ F **:-** G,
  $\boldsymbol{I}(\pi)(@\texttt{r F :- G}) = \mathbf{t}$ iff
  $\boldsymbol{I}(\pi)(F) \geq \texttt{min}\,(\,\boldsymbol{I}(\pi)(G),\;\; \boldsymbol{I}(\langle D_0 \rangle)(\texttt{not}\ \Diamond\ \texttt{\$defeated(handle(}r, F\texttt{))}))$,
  where D_0 is the first database in the path π.

We will write $\boldsymbol{I}, \pi \models \phi$ and say that ϕ is ***satisfied*** on path π in the path structure $\boldsymbol{I}$ if $\boldsymbol{I}(\pi)(\phi)$=$\mathbf{t}$.
We will say that a path structure $\boldsymbol{I}$ is **total** if, for every path π and every serial goal ϕ, $\boldsymbol{I}(\pi)(L)$ is either $\mathbf{t}$ or $\mathbf{f}$. ☐

▶ **Definition 12** (Model of a transactional formula). A path structure, $\boldsymbol{I}$, is a *model* of a transaction formula ϕ if $\boldsymbol{I}, \pi \models \phi$ for every path π. In this case, we write $\boldsymbol{I} \models \phi$ and say that $\boldsymbol{I}$ is a **model** of ϕ or that ϕ is **satisfied** in $\boldsymbol{I}$. A path structure $\boldsymbol{I}$ is a model of a set of formulas if it is a model of every formula in the set. ☐

▶ **Definition 13** (Model of $\mathcal{TR}^{DA}$). A path structure $\boldsymbol{I}$ is a model of a $\mathcal{TR}^{DA}$ transaction base $\mathbf{P}$ if all rules in $\mathbf{P}$ are satisfied in $\boldsymbol{I}$ (i.e., $\boldsymbol{I} \models R$ for every $R \in \mathbf{P}$). Given a $\mathcal{TR}^{DA}$ transaction base $\mathbf{P}$, an argumentation theory AT, and a path structure $\boldsymbol{M}$, we say that $\boldsymbol{M}$ is a model of $\mathbf{P}$ with respect to the argumentation theory AT, written as $\boldsymbol{M} \models (\mathbf{P}, AT)$, if $\boldsymbol{M} \models \mathbf{P}$ and $\boldsymbol{M} \models AT$. ☐

Like classical logic programs, the Herbrand semantics of serial-Horn $\mathcal{TR}$ can be formulated as a fixpoint theory [3]. In classical logic programming, given two Herbrand partial interpretations σ_1 and σ_2, we write $\sigma_1 \preceq \sigma_2$ if all `not`-free literals that are true in σ_1 are also true in σ_2 and all `not`-literals that are true in σ_2 are also true in σ_1. Similarly, for partial interpretations, $\sigma_1 \leq \sigma_2$ if all `not`-free literals that are true in σ_1 are also true in σ_2 and all `not`-literals that are true in σ_1 are also true in σ_2.

▶ **Definition 14** (Order on Path Structures). If $\mathbf{M}_1$ and $\mathbf{M}_2$ are partial Herbrand path structures, then $\mathbf{M}_1 \preceq \mathbf{M}_2$ if $\mathbf{M}_1(\pi) \preceq \mathbf{M}_2(\pi)$ for every path, π. Similarly, we write $\mathbf{M}_1 \leq \mathbf{M}_2$ if $\mathbf{M}_1(\pi) \leq \mathbf{M}_2(\pi)$ for every path, π. A model $\mathbf{M}$ of $\mathbf{P}$ is **minimal** with respect to $\preceq$ iff for any other model, $\mathbf{N}$, of $\mathbf{P}$, we have that $\mathbf{N} \preceq \mathbf{M}$ implies $\mathbf{N} = \mathbf{M}$. The **least** model of $\mathbf{P}$ is a minimal model that is unique (if it exists). ☐

It is well-known that in ordinary logic programming any set of Horn rules always has a least model. In [5], it is shown that every positive serial-Horn $\mathcal{TR}$ program has a unique least total model. Theorem 15, below, shows that this property is preserved by serial `not`-free $\mathcal{TR}$ programs, but in this case the model might be a partial path structure. Serial `not`-free programs are more general than the positive $\mathcal{TR}$ programs because the undefined propositional symbol $\mathbf{u}^\pi$ for some path π may occur in the bodies of the program rules.

▶ **Theorem 15** (Unique Least Partial Model for serial `not`-free $\mathcal{TR}$ programs). *If* $\mathbf{P}$ *is a* `not`*-free* $\mathcal{TR}^{DA}$ *program, then* $\mathbf{P}$ *has a least Herbrand model, denoted* $LPM(\mathbf{P})$.[2] ☐

Next we define well-founded models for $\mathcal{TR}^{DA}$ by adapting the definition from [13]. First, we define the quotient operator, which takes a $\mathcal{TR}^{DA}$ program $\mathbf{P}$ and a path structure $\boldsymbol{I}$ and yields a serial-Horn $\mathcal{TR}$ program $\frac{\mathbf{P}}{\boldsymbol{I}}$. Despite what one might have been expecting, this adaptation is rather subtle.

▶ **Definition 16** (Quotient). Let $\mathbf{P}$ be a set of $\mathcal{TR}^{DA}$ rules and $\boldsymbol{I}$ a path structure for $\mathbf{P}$. The $\mathcal{TR}^{DA}$ **quotient of P by** $\boldsymbol{I}$, written as $\frac{\mathbf{P}}{\boldsymbol{I}}$, is defined through the following sequence of steps:

1. First, each occurrence of every `not`- literal of the form $\texttt{not}\,L$ in $\mathbf{P}$ is replaced by $\mathbf{t}^{\pi}$ for every path π such that $\boldsymbol{I}(\pi)(\texttt{not}\,L) = \mathbf{t}$ and with $\mathbf{u}^{\pi}$ for every path π such that $\boldsymbol{I}(\pi)(\texttt{not}\,L) = \mathbf{u}$.
2. For each labeled rule of the form $@r\ L :- Body$ obtained in the previous step, replace it with rules of the form: $L :- \mathbf{t}^{\langle D_t \rangle} \otimes Body$ for each database state D_t such that $\boldsymbol{I}(\langle D_t \rangle)(\texttt{not}\,(\Diamond\ \texttt{\$defeated}(\texttt{handle}(r, L)))) = \mathbf{t}$, and rules of the form $L :- \mathbf{u}^{\langle D_u \rangle} \otimes Body$ for each database state D_u such that $\boldsymbol{I}(\langle D_u \rangle)(\texttt{not}\,(\Diamond\ \texttt{\$defeated}(\texttt{handle}(r, L)))) = \mathbf{u}$.
3. Remove the labels from the remaining rules. The resulting set of rules is the quotient $\frac{\mathbf{P}}{\boldsymbol{I}}$.

☐

Note that in Step 1 of the above definition of the quotient each occurrence of $\texttt{not}\,L$ is replaced with different $\mathbf{t}^{\pi}$ and $\mathbf{u}^{\pi}$ for different π's, so every rule in $\mathbf{P}$ may be replaced with several (possibly infinite number of) `not`-free rules. All combinations of replacements for the `not`- literals in the body of the rules have to be used. Only the π's where $\boldsymbol{I}(\pi)(\texttt{not}\,L) = \mathbf{f}$ are not used, which effectively means that the rule instances that correspond to those cases are removed from consideration. Also note that, the $\mathcal{TR}^{DA}$ quotient of a $\mathcal{TR}^{DA}$ transaction base $\mathbf{P}$ with respect to an argumentation theory AT (the program union $\mathbf{P} \cup AT$) for any path structure $\mathbf{I}$, $\frac{\mathbf{P} \cup AT}{\boldsymbol{I}}$, is a negation-free $\mathcal{TR}$ program, so, by Theorem 15, it has a unique least Herbrand model, $LPM(\frac{\mathbf{P} \cup AT}{\boldsymbol{I}})$.

We will now give the definition for the immediate consequence operator Γ. For compatibility with the classical notations in logic programming, we will use the set representation of Herbrand models: $\boldsymbol{I}^{+} = \{L \mid L \in \boldsymbol{I}$ *is a* `not`*-free literal*$\}$, $\boldsymbol{I}^{-} = \{L \mid L \in \boldsymbol{I}$ *is a* `not`*-literal*$\}$ and $\boldsymbol{I} = \boldsymbol{I}^{+} \cup \boldsymbol{I}^{-}$.

▶ **Definition 17** ($\mathcal{TR}^{DA}$ immediate consequence operator). The incremental consequence operator, Γ, for a $\mathcal{TR}^{DA}$ transaction base $\mathbf{P}$ with respect to the argumentation theory AT takes as input a path structure $\boldsymbol{I}$ and generates a new path structure: $\Gamma(\boldsymbol{I}) =^{def} LPM\left(\frac{\mathbf{P} \cup AT}{\boldsymbol{I}}\right)$

Suppose $I_{\emptyset}$ is the path structure that maps each path π to the *empty* Herbrand interpretation in which all propositions are undefined (i.e., for every path π and every literal L, we have $I_{\emptyset}(\pi)(L) = \mathbf{u}$).

[2] All proofs can be found in our technical report [10].

The ordinal powers of the immediate consequence operator Γ are defined inductively as follows:

- $\Gamma^{\uparrow 0}(I_\emptyset) \;=\; I_\emptyset$;
- $\Gamma^{\uparrow \alpha}(I_\emptyset) \;=\; \Gamma(\Gamma^{\uparrow \alpha-1}(I_\emptyset))$, for α a successor ordinal;
- $\Gamma^{\uparrow \alpha}(I_\emptyset)(\pi) \;=\; \cup_{\beta<\alpha}\Gamma^{\uparrow \beta}(I_\emptyset)(\pi)$, for every path π and α a limit ordinal. ☐

The operator Γ is monotonic with respect to the $\leq$ order relation when $\mathbf{P}$ and AT are fixed (see [10]). Because Γ is monotonic, the sequence $\{\Gamma^{\uparrow n}(I_\emptyset)\}$ $(\Gamma^{\uparrow 0}(I_\emptyset),\ \Gamma^{\uparrow 1}(I_\emptyset),\ \Gamma^{\uparrow 2}(I_\emptyset),\ \ldots)$ has a least fixed point and is computable via transfinite induction.

▶ **Definition 18** (Well-founded model). The **well - founded model** of a $\mathcal{TR}^{DA}$ transaction base $\mathbf{P}$ with respect to the argumentation theory AT, written as $WFM(\mathbf{P}, AT)$, is defined as the limit of the sequence $\{\Gamma^{\uparrow n}(I_\emptyset)\}$. ☐

▶ **Theorem 19** (Correctness of the Constructive $\mathcal{TR}^{DA}$ Least Model). *$WFM(\mathbf{P}, AT)$ is the least model of the program $(\mathbf{P}, AT)$.*

The next theorem shows that $\mathcal{TR}^{DA}$ programs under the well - founded semantics reduce to ordinary $\mathcal{TR}$ programs under the same well - founded semantics. In conclusion, $\mathcal{TR}^{DA}$ can be implemented using ordinary transaction logic programming systems that support the well - founded semantics.

▶ **Theorem 20** ($\mathcal{TR}^{DA}$ Reduction). *$WFM(\mathbf{P}, AT)$ coincides with the well - founded model of the $\mathcal{TR}$ program $\mathbf{P}' \cup AT$, where $\mathbf{P}'$ is obtained from $\mathbf{P}$ by changing every defeasible rule* (`@r L :- Body`) $\in \mathbf{P}$ *to the plain rule* `L :- not (`$\Diamond$ `$defeated(handle(r,L)) )` $\otimes$ `Body` *and removing all the remaining tags.*

5 The $GCLP^{\mathcal{TR}}$ Argumentation Theory

We present here a particularly interesting argumentation theory which extends GCLP—*generalized courteous logic programs* [12]—to $\mathcal{TR}$ under the $\mathcal{TR}^{DA}$ framework. The inferences claimed in the discussion of the planning example in Section 2 assumed that particular argumentation theory. We will call this argumentation theory $GCLP^{\mathcal{TR}}$. As any argumentation theory in our framework, $GCLP^{\mathcal{TR}}$ defines a version of the predicate `$defeated` using various auxiliary concepts. We define these concepts first.

The user-defined predicates `!`**`opposes`** and `!`**`overrides`** are relations specified over rule handles. They tell the system what rule instances are in conflict with each and which rule instances are preferred over other rules.

The predicate `$defeated` is defined indirectly in terms of the predicates `!`**`opposes`** and `!`**`overrides`**. In the following definitions the variables R and S are assumed to range over rule handles, while the implicit current state identifier D is assumed to range over the possible database states. A rule is *defeated* if it is *refuted* or *rebutted* by some other rule, assuming that the first rule is defeasible and the second rule is not *compromised* or *disqualified*. We will define these notions shortly, but first we explain them informally. A rule is *refuted* if a higher-priority rule implies a conclusion that is incompatible with the conclusion implied by the refuted rule, A rule *rebuts* another rule if the two rules assert conflicting conclusions and there is no way to resolve the conflict. A rule is *compromised* if

it is defeated by some other rule, and a rule *disqualified* if that rule refutes itself.

$$
\begin{array}{lll}
\texttt{\$defeated}(R) & :- & \texttt{\$refutes}(S,R) \wedge \texttt{ not } \texttt{\$compromised}(S). \\
\texttt{\$defeated}(R) & :- & \texttt{\$rebuts}(S,R) \wedge \texttt{ not } \texttt{\$compromised}(S). \\
\texttt{\$defeated}(R) & :- & \texttt{\$disqualified}(R). \\
\texttt{\$refutes}(R,S) & :- & \texttt{\$conflict}(R,S) \wedge \texttt{!}\mathbf{overrides}(R,S). \\
\texttt{\$conflict}(R,S) & :- & \texttt{\$candidate}(R), \texttt{\$candidate}(S), \texttt{!}\mathbf{opposes}(R,S).
\end{array} \quad (1)
$$

A rule R *rebuts* another rule S if the two rules assert conflicting conclusions, but neither rule is "more important" than the other, *i.e.*, no preference can be inferred between the two rules. This intuition can be expressed in several different ways, but we selected the one below, which mimics the definition in [17].

$$
\begin{array}{lll}
\texttt{\$rebuts}(R,S) & \texttt{:-} & \texttt{\$candidate}(R) \wedge \texttt{\$candidate}(S) \wedge \\
 & & \texttt{!}\mathbf{opposes}(R,S) \wedge \texttt{not } \texttt{\$compromised}(R) \wedge \\
 & & \texttt{not } \texttt{\$refutes}(__, R) \wedge \texttt{ not } \texttt{\$refutes}(__, S).
\end{array} \quad (2)
$$

The important difference here compared to [17] is that we are dealing with state-changing actions and so all tests for refutation, rebuttal, and the like, must be hypothetical. This is reflected in the definition of a rule candidate. We say that a rule instance is a *candidate* if its body is *hypothetically* true in the current database state. The other two rules in the group below specify the symmetry of `!opposes` and the fact that literals H and `neg` H are in conflict with each other.

$$\texttt{\$candidate}(R) \texttt{ :- } \mathbf{body}(R,B) \otimes \Diamond\mathbf{call}(B). \quad (3)$$

$$\texttt{!}\mathbf{opposes}(X,Y) \texttt{ :- } \texttt{!}\mathbf{opposes}(Y,X). \quad (4)$$

$$\texttt{!}\mathbf{opposes}(\texttt{handle}(__, H), \texttt{ handle}(__, \texttt{ neg } H)). \quad (5)$$

A rule is compromised if it is defeated, and it is disqualified if it transitively refutes itself. Here the predicate $\texttt{\$refutes}_{tc}$ denotes the transitive closure of `$refutes`.

$$
\begin{array}{l}
\texttt{\$compromised}(R) \texttt{ :- } \texttt{\$refuted}(R) \wedge \texttt{\$defeated}(R). \\
\texttt{\$disqualified}(X) \texttt{ :- } \texttt{\$refutes}_{tc}(X,X). \\
\texttt{\$refutes}_{tc}(X,Y) \texttt{ :- } \texttt{\$refutes}(X,Y). \\
\texttt{\$refutes}_{tc}(X,Y) \texttt{ :- } \texttt{\$refutes}_{tc}(X,Z) \wedge \texttt{\$refutes}(Z,Y).
\end{array} \quad (6)
$$

As in [17], one can define other versions of the above argumentation theory, which differ from the above in various edge cases. However, defining such variations is tangential to the main focus of the present paper.

6 Implementation, evaluation and related work

We implemented an interpreter for $\mathcal{TR}^{DA}$ in XSB [3] and tested it on a number of examples, including Example 2.1. The goal of these tests was to demonstrate how preferential heuristics can be expressed in $\mathcal{TR}^{DA}$ and to evaluate their effects on the efficiency of planning. Table 1 shows how the preferential heuristics of Example 2.1 fare in our tests. We can see that the number of plans being searched decreases dramatically and so does the time and space.

[3] http://xsb.sourceforge.net/

Table 1 Planing in blocks world with and without preferential heuristics

World size		No heuristics	Preferential heuristics
30 blocks	Plans	4060	28
	Time(sec.)	2.390	0.438
	Space(kBs)	3730	90
40 blocks	Plans	9880	38
	Time(sec.)	7.000	1.219
	Space(kBs)	8562	120
50 blocks	Plans	19600	48
	Time(sec.)	17.109	2.938
	Space(kBs)	16347	150

However, the time spent on generation of all those plans is not proportional to their number because our implementation takes advantage of sharing of partially constructed plans among the different searches due to tabling [9] even without the heuristics.

Although a great number of works deal with defeasibility in logic programming, few have goals similar to ours: to lift defeasible reasoning from static logic programming to a logic for expressing knowledge base dynamics, such as $\mathcal{TR}$. As far as the actual chosen approach to defeasible reasoning is concerned, this work is based on [17], and extensive comparison with other works on defeasible reasoning can be found there. Although our work is *not* about planning but rather about a general language for declarative programming with defeasible actions, the closest works that we can possibly compare with are the works on planning with preferences. $\mathcal{TR}^{DA}$ is quite different from [15] in that it is a full-fledged logic that combines both declarative and procedural elements, while [15] is geared towards specifying preferences over planning *solutions*.Whereas $\mathcal{TR}^{DA}$ deals with infinite domains and allows function symbols, the approach in [15] considers only planning with complete information on finite domains and deterministic actions. Thus, although the two approaches have common applications in the area of planning, they target different knowledge representation scenarios. Both the *temporal* and the *choice* preferences presented in [7]can be expressed in the $\mathcal{TR}^{DA}$ framework, although due to the difference in the semantics the exact relationship needs further study. The framework [8] for planning with cost preferences assigns a numeric cost to each action and plans with the minimal cost are considered to be optimal. Clearly, this work uses a completely different type of preferences and tackles a different and very specific problem in planning, which we do not address.

7 Conclusions

This paper proposes a theory of defeasible reasoning in Transaction Logic, an extension of classical logic for representing both declarative and procedural knowledge. This new logic, called $\mathcal{TR}^{DA}$, extends our prior work on defeasible reasoning with argumentation theories from static logic programming to a logic that captures the dynamics in knowledge representation. We also extend the Courteous style of defeasible reasoning [12] to incorporate actions, planning, and other dynamic aspects of knowledge representation. We believe that $\mathcal{TR}^{DA}$ can become a rich platform for expressing heuristics about actions. The paper also makes a contribution directly to the development of Transaction Logic itself by defining the well-founded semantics for it and for its $\mathcal{TR}^{DA}$ extension—a non-trivial adaptation of the classical well-founded semantics of [16].

References

1. A.J. Bonner and M. Kifer. Applications of transaction logic to knowledge representation. In *Proceedings of the International Conference on Temporal Logic*, number 827 in Lecture Notes in Artificial Inteligence, pages 67–81, Bonn, Germany, July 1994. Springer-Verlag.
2. A.J. Bonner and M. Kifer. A logic for programming database transactions. In J. Chomicki and G. Saake, editors, *Logics for Databases and Information Systems*, chapter 5, pages 117–166. Kluwer Academic Publishers, Berlin, March 1998.
3. A.J. Bonner and M. Kifer. Results on reasoning about action in transaction logic. In B. Freitag, H. Decker, M. Kifer, and A. Voronkov, editors, *Transactions and Change in Logic Databases*, volume 1472 of *LNCS*. Springer-Verlag, Berlin, 1998.
4. A.J. Bonner and M. Kifer. The state of change: A survey. In B. Freitag, H. Decker, M. Kifer, and A. Voronkov, editors, *Transactions and Change in Logic Databases*, volume 1472 of *LNCS*. Springer-Verlag, Berlin, 1998.
5. A.J. Bonner and Michael Kifer. Transaction logic programming (or a logic of declarative and procedural knowledge). Technical Report CSRI-323, University of Toronto, November 1995. http://www.cs.toronto.edu/~bonner/transaction-logic.html.
6. James P. Delgrande, Torsten Schaub, and Hans Tompits. A framework for compiling preferences in logic programs. *Theory and Practice of Logic Programming*, 2:129–187, 2003.
7. James P. Delgrande, Torsten Schaub, and Hans Tompits. Domain-specific preferences for causal reasoning and planning. In Didier Dubois, Christopher A. Welty, and Mary-Anne Williams, editors, *KR*, pages 673–682, Whistler, Canada, 2004. AAAI Press.
8. Thomas Eiter, Wolfgang Faber, Nicola Leone, Gerald Pfeifer, and Axel Polleres. Answer set planning under action costs. *J. Artif. Int. Res.*, 19:25–71, August 2003.
9. Paul Fodor and Michael Kifer. Tabling for transaction logic. In *Proceedings of the 12th international ACM SIGPLAN symposium on Principles and practice of declarative programming*, PPDP '10, pages 199–208, New York, NY, USA, 2010. ACM.
10. Paul Fodor and Michael Kifer. Transaction Logic with Defaults and Argumentation Theories. Technical report, Stony Brook University, May 2011. Available from http://ewl.cewit.stonybrook.edu/trda.pdf.
11. Michael Gelfond and Tran Cao Son. Reasoning with prioritized defaults. In *Selected papers from the Third International Workshop on Logic Programming and Knowledge Representation*, pages 164–223, London, UK, 1998. Springer-Verlag.
12. B.N. Grosof. A courteous compiler from generalized courteous logic programs to ordinary logic programs. Technical Report Supplementary Update Follow-On to RC 21472, IBM, July 1999.
13. T.C. Przymusinski. Well-founded and stationary models of logic programs. *Annals of Mathematics and Artificial Intelligence*, 12:141–187, 1994.
14. Dumitru Roman and Michael Kifer. Semantic web service choreography: Contracting and enactment. In Amit P. Sheth, Steffen Staab, Mike Dean, Massimo Paolucci, Diana Maynard, Timothy W. Finin, and Krishnaprasad Thirunarayan, editors, *International Semantic Web Conference*, volume 5318 of *Lecture Notes in Computer Science*, pages 550–566, Karlsruhe, 2008. Springer.
15. Tran Cao Son and Enrico Pontelli. Planning with preferences using logic programming. *Theory Pract. Log. Program.*, 6:559–607, September 2006.
16. A. Van Gelder, K.A. Ross, and J.S. Schlipf. The well-founded semantics for general logic programs. *Journal of ACM*, 38(3):620–650, 1991.
17. Hui Wan, Benjamin Grosof, Michael Kifer, Paul Fodor, and Senlin Liang. Logic programming with defaults and argumentation theories. In *Proceedings of the 25th International Conference on Logic Programming*, ICLP '09, pages 432–448, Berlin, Heidelberg, 2009. Springer-Verlag.

Multi-agent Confidential Abductive Reasoning*

Jiefei Ma[1], Alessandra Russo[1], Krysia Broda[1], and Emil C. Lupu[1]

1 **Department of Computing, Imperial College London**
180 Queen's Gate, London, United Kingdom
{j.ma,a.russo,k.broda,e.c.lupu}@imperial.ac.uk

Abstract

In the context of multi-agent hypothetical reasoning, agents typically have partial knowledge about their environments, and the union of such knowledge is still incomplete to represent the whole world. Thus, given a global query they collaborate with each other to make correct inferences and hypothesis, whilst maintaining global constraints. Most collaborative reasoning systems operate on the assumption that agents can share or communicate any information they have. However, in application domains like multi-agent systems for healthcare or distributed software agents for security policies in coalition networks, confidentiality of knowledge is an additional primary concern. These agents are required to collaborately compute consistent answers for a query whilst preserving their own private information. This paper addresses this issue showing how this dichotomy between "open communication" in collaborative reasoning and protection of confidentiality can be accommodated. We present a general-purpose distributed abductive logic programming system for multi-agent hypothetical reasoning with confidentiality. Specifically, the system computes consistent conditional answers for a query over a set of distributed normal logic programs with possibly unbound domains and arithmetic constraints, preserving the private information within the logic programs. A case study on security policy analysis in distributed coalition networks is described, as an example of many applications of this system.

1998 ACM Subject Classification I.2.11 Distributed Artificial Intelligence

Keywords and phrases Abductive Logic Programming, Coordination, Agents

Digital Object Identifier 10.4230/LIPIcs.ICLP.2011.175

1 Introduction

In the context of multi-agent reasoning, each agent has its own *partial knowledge* about the world together with local and/or global constraints. Given a reasoning task, agents interact and compute answers that are consistent with respect to the global constraints. When the union of all the agent knowledge is still incomplete to represent the whole world, hypothetical reasoning is needed, and agents need to collaborate to make correct inferences and hypotheses given a global query. Previously, a general-purpose system called DAREC has been developed, which combines distributed problem solving and abductive logic programming, for multi-agent hypothetical reasoning. In DAREC, agent knowledge is represented as a normal logic program, and a distributed abductive logic programming algorithm is used to coordinate the agents' local reasoning tasks. Through this algorithm, agents compute local conditional answers, by assuming undefined knowledge needed to maintain their (global) constraints, and coordinate their proofs through consistency checks over their respective assumptions. DAREC is the first

* This research is continuing through participation in the International Technology Alliance sponsored by the U.S. Army Research Laboratory and the U.K. Ministry of Defence.

© Jiefei Ma, Alessandra Russo, Krysia Broda, Emil C. Lupu;
licensed under Creative Commons License NC-ND

distributed abductive system that can compute non-ground answers and handle arithmetic constraints. In DAREC, all knowledge is considered public, so, during collaboration, agents can communicate any information they may have. However, in application domains such as simulation of policy-based distributed systems for decentralised policy analysis, confidentiality is an additional primary concern – agents may contain private information that cannot be shared with others during, or after, the collaborative reasoning. This concern imposes an extra challenge to agent interactions during the reasoning process, as agents must decide what to disclose between their communications.

This paper addresses the new challenge by extending the DAREC system to support multi-agent hypothetical reasoning with confidentiality. It provides two main contributions. At knowledge representation level the logical language and the distributed abductive framework have been extended to allow modelling of private agent knowledge. At algorithmic level, the distributed proof procedure has been customised with a *safe* yet efficient agent interaction protocol, which prevents private knowledge being passed between agents and allows some degree of concurrent computation. From the operational point of view, our new distributed abductive algorithm, called DAREC^2 is a coordinated state rewriting process, consisting of *local abductive inferences* by the agents and *coordination* of these local inferences. The local inference is a goal-directed reasoning process, where a current agent (i) solves as many sub-goals of the query as possible, using its own knowledge, and (ii) collects those sub-goals solvable only by other agents, and the constraints that must be satisfied by all agents to guarantee *global consistency* of the final answer. The latter are generated from *constructive negations* and *arithmetic constraints* during the local inference process. They can be reduced to a set of inequalities and arithmetic constraints and be handled by external Constraint Logic Programming (CLP) solvers, enabling also reasoning over unbounded domains. The collected sub-goals and constraints, together with the hypotheses made during the local inference, are encapsulated into a *token state*, which is then passed around to other agents for further processing once all private sub-goals of the current agent have been solved by the agent. This guarantees that confidential information is not included in the token state and not passed to other agents. The coordination of state-passing implements *synchronised backtracking*, whilst enabling concurrent computation between local inferences. The coordination allows two types of agent interaction: *positive* and *negative*. In the case of a positive interaction, the token state is directed to a suitable agent (i.e. who may help to solve some pending sub-goals), whereas for negative interactions, the token state is passed among all agents enforcing each to check the pending constraints. Application dependent strategies may be adopted to interleave/combine such interactions in order to reduce communication overheads. The new system is implemented in Prolog together with a benchmarking test-bed environment. The main intended use of DAREC^2 is as a decoupleable multi-purpose tool for larger multi-agent systems (MAS). For example, each DAREC^2 agent could be implemented as a reactive reasoning module of an agent in a larger MAS to support other agent/system functionalities (e.g., distributed reasoning over BDI agents' belief stores [12]). Alternatively, the whole DAREC^2 system could be implemented as a "simulator" to verify properties of a target MAS, such as the case study example described in this paper for decentralised policy analysis.

The paper is organised as follows. Section 2 discusses related work, Section 3 formalises the notion of a multi-agent abductive reasoning problem, and Section 4 describes the DAREC^2 algorithm. A case study in the context of distributed policy analysis to exemplify the confidentiality aspect in real-world applications is described in Section 5. Finally, conclusion and future work are given in Section 6.

2 Related Work

Distributed abductive reasoning has previously been studied, such as in the ALIAS system [2], the DARE system [7], MARS [1] and DAREC [8]. ALIAS focuses on *local consistency*, i.e., the global answer is consistent with each agent's knowledge individually, where the other three systems focus on *global consistency*, i.e, the global answer is consistent with the union of all the agent knowledge. Both ALIAS and DARE deploy distributed abductive algorithms that are based on the Kakas-Mancarella proof procedure [5], and can only compute ground proofs/answers. MARS uses the consequence finding algorithm [9] for local computation, and is mainly for computing a consistent superset of the agents' existing hypotheses. DAREC, whose distributed abductive algorithm is based on ASystem [10], is the first system that can compute non-ground proofs/answers for possibly unbound domains and with arithmetic constraint satisfaction support. None of these systems considers the confidential aspect of agent knowledge. Our system extends DAREC. In addition to the inherited features, it allows private and public agent knowledge to be explicitly expressed, and guarantees confidentiality during collaborative reasoning. Specifically, the new system introduces *askable literals* that can be shared between agents (i.e., to represent public knowledge), and new local inference rules for solving askable sub-goals. An agent interaction protocol, consisting of special goal selection and agent selection strategies, is enforced to preserve confidentiality while reducing communications and increasing concurrent computation.

In DAREC^2, an askable literal is $A@S$ where A is an atom and S is an agent identifier indicating where the askable sub-goal should be solved. This language feature is most closed to one proposed in *speculative computation* [11]. However, whereas in speculative computation when an askable sub-goal is selected, the agent identifier must be ground, in our system it can also be a variable with quantifier.

3 Knowledge Representation

Standard logic programming notations are employed throughout the paper. We use $\vec{t}$ to represent a vector of arguments. Constraint atoms are those formed with constraint predicates from CLP for a particular domain, such as $\{\in, <, \leq, >, \geq\}$. A clause is either a *rule* $\phi \leftarrow \phi_1, \ldots, \phi_n$ with $n \geq 0$, or a *denial* $\leftarrow \phi_1, \ldots, \phi_n$ with $n > 0$, where $\phi_1, \ldots, \phi_n$ is a conjunction of literals called the *body*, and in the case of a rule ϕ is an atom called the *head*. All variables appearing in a clause are *universally quantified* with the scope the whole clause implicitly, unless stated otherwise. A *query* is a conjunction of literals, whose variables are *existentially quantified* with the scope the whole conjunction implicitly.

In abductive logic programming, predicates of atoms that are not equality or constraint are divided into *abducible predicate* and *non-abducible predicate*. An atom with (non-)abducible predicate is called an *(non-)abducible (atom)*. An abductive (logic programming) framework is a tuple $\langle \Pi, \mathcal{AB}, \mathcal{IC} \rangle$, where Π is a finite set of rules called the *background knowledge*, $\mathcal{AB}$ is a set of abducible predicates, and $\mathcal{IC}$ is a finite set of denials, each of which contains at least one positive abducible, called the *integrity constraints*. Sometimes $\mathcal{AB}$ represents the set of all abducible atoms. Without loss of generality, it is often assumed that no abducible appears in the head of a rule in Π. Therefore, non-abducibles are also called *defined atoms* (or *defined* in brief). Given a query $\mathcal{Q}$, an abductive logic programming task consists of computing an abductive answer $\langle \Delta, \theta \rangle$, where Δ is a set of abducibles, θ is a set of variable substitutions, such that (1) $\Pi \cup \Delta\theta \models \mathcal{Q}$, (2) $\Pi \cup \Delta\theta \models \mathcal{IC}$, where $\models$ is logical entailment under a selected semantics. Δ is called the *hypotheses*, or *explanation*, for the given query (i.e. *observation*).

3.1 Multi-agent Confidential Abductive Framework

In this work, we focus on MAS's with a *fixed* set of agents satisfying the following assumptions:

- Each agent has a *unique ID* and has an *abductive framework*.
- Agents have the same set of abducible predicates, which ensures they can only generate hypotheses that are not defined or *provable* by others.
- Agents can send peer-to-peer messages, i.e., the MAS communication channels form a *fully connected* graph, eliminating the need for considering message routing.

As our current main focus is on the correctness of multi-agent hypothetical reasoning, we further assume that the agents and the communication channels are reliable (i.e. no corruption or loss of messages).

Let's denote an agent's abductive framework with the tuple $\mathcal{F} = \langle \Pi, \mathcal{AB}, \mathcal{IC} \rangle$, where Π and $\mathcal{IC}$ constitute together the agent's *local knowledge*. We may use an agent's identifier, say i, to suffix the agent's abductive framework and its components, i.e., $\mathcal{F}_i$, Π_i, $\mathcal{AB}_i$ and $\mathcal{IC}_i$. Non-abducible predicates of an agent can be *private* or *public (askable)*. The former are defined only in the agent background knowledge and *cannot* appear in the body of a rule of any other agent[1]. Public predicates can only be defined in the agent background knowledge, but *can* appear in the body of rules of other agents.

▶ **Definition 1.** An **askable** atom is $p(\vec{u})@ID$ where $p(\vec{u})$ is a non-abducible atom and ID is the agent identifier where the predicate is defined.

Askable atoms $p(\vec{u})@ID$ that appear as the head of a rule in an agent's background knowledge have their ID ground with the agent's identifier. Askable atoms that appear in the body of a rule may have an unground agent identifier. The ID of an askable atom is therefore more than just a syntactic alias. It denotes a variable that can be (appropriately) existentially and universally quantified. Syntactically, an askable can be seen as a non-abducible with an extra agent identifier argument. The negation of an askable $\neg p(\vec{u})@ag$ is read as $\neg(p(\vec{u})@ag)$, meaning "$p(\vec{u})$ should not be provable by agent ag".

▶ **Definition 2.** The **global abductive framework** is $\langle \Sigma, \widehat{\mathcal{F}} \rangle$, where Σ is the set of all agent identifiers and $\widehat{\mathcal{F}}$ is the set of agent abductive frameworks, i.e. $\{\mathcal{F}_i \mid i \in \Sigma\}$. For any pair of agents $i, j \in \Sigma$, $\mathcal{AB}_i = \mathcal{AB}_j$.

▶ **Definition 3.** **Global Abductive Answer** Given a global abductive framework $\langle \Sigma, \widehat{\mathcal{F}} \rangle$ and a query $\mathcal{Q}$, let $\widehat{\Pi} = \bigcup_{i \in \Sigma} \Pi_i$, let $\widehat{\mathcal{IC}} = \bigcup_{i \in \Sigma} \mathcal{IC}_i$, and let $\widehat{\mathcal{AB}} = \bigcup_{i \in \Sigma} \mathcal{AB}_i$. A pair $\langle \Delta, \theta \rangle$ is a global abductive answer for $\mathcal{Q}$ if and only if:

$$(1)\ \Delta\theta \subseteq \mathcal{AB};\ (2)\ \widehat{\Pi} \cup \Delta\theta \models \mathcal{Q}\theta;\ (3)\ \widehat{\Pi} \cup \Delta\theta \models \widehat{\mathcal{IC}}$$

where θ is the variable substitutions over the variables in $\mathcal{Q}$, and $\models$ is the logical entailment of a selected semantics for the logic program formed by $\widehat{\Pi} \cup \widehat{\mathcal{IC}}$.

4 Distributed Algorithm

Given a global abductive framework (of a MAS) and a query, agents must collaborate to compute global abductive answers, while keeping there private information confidential.

[1] Name clashes between private predicates of different agents are assumed to have been resolved.

▶ **Definition 4.** The collaboration of agents to compute global abductive answers for a given query is a *confidential reasoning* if and only if no private predicate and its definitions of any agent can be seen or inferred by another agent during or after the collaboration.

The DAREC^2 distributed abductive algorithm satisfies this property. Operationally, the distributed computation is a sequence of coordinated local abductive computations, i.e., *distributed abduction = local abduction + coordination*.

4.1 Local Abduction

Local abduction is a top-down (goal-directed) abductive inference, which can be described as a state rewriting process. A *state* contains intermediate computational results, and has four components: (1) a *remaining goals store* – a goal is either a literal or a denial "$\forall \vec{X} \leftarrow \phi_1, \ldots, \phi_n$" ($n > 0$), where $\vec{X}$ is the set of variables in the $\phi_1, \ldots, \phi_n$ that are universally quantified with the scope the whole denial, (2) a *hypotheses store* containing a set of collected abducibles, (3) a *constraints store* containing a set of collected and consistent inequalities and arithmetic constraints, and (4) a *denials store* containing collected denial goals whose left-most literal in the body is an abducible. These denials represent the *conditions* for collecting the instances of their left-most abducibles and are collected during the inference. A state also contains *abducible tagging* information – an abducible in the hypotheses store is *tagged* by an agent's identifier if the agent **has not** checked it against its integrity constraints. All free variables in a state are existentially quantified with the scope the whole state implicitly. A *solved* state is one that has *no remaining goal* and *no tagged abducible*. At each inference step, a goal is selected from a non-solved state, and a literal is selected from the goal if the goal is a denial. *Safe* goal selection strategies are adopted, in which the *current agent* (i.e. the one performing the local abduction) can select:

- an askable goal, only if its agent ID is the current agent's ID or a variable;
- an askable from a denial goal, only if its agent ID is the current agent's ID or a variable;
- an abducible from a denial goal, only if there is no private non-abducible literal in the denial goal;
- inequalities, arithmetic constraints and negative literals, only if they do not contain universally quantified variables;

The first three requirements guarantee the *confidentiality* (see later), whereas the last requirement avoids *floundering*. The next state is obtained after applying a *local inference rule* to the selected goal. These rules are based on the ASystem [6] inference rules, with extensions to handle askables and tagging information, where ag is the current agent's ID:

- if an askable goal $p(\vec{u})@ID$ is selected:
  - if ID is ag, it is resolved with a rule in the agent's background knowledge;
  - otherwise, ID must be an existentially quantified variable, and the goal can be replaced with either $p(\vec{u})@ag$, if $ID = ag$ is satisfiable, or with $ID \neq ag, p(\vec{u})@ID$. Semantically, this means that the askable can be either solved by the current agent, or by a different agent (later).
- if an askable $p(\vec{u})@ID$ in a denial goal is selected:
  - if ID is ag, it is reduced as being non-abducible in a denial goal;
  - if ID is an existentially quantified variable, then either the denial goal is replaced with one obtained by replacing the askable with $p(\vec{u})@ag$ if $ID = ag$ is satisfiable, or the denial goal is kept but an inequality goal $ID \neq ag$ is added. Semantically, this means that the denial can be either solved by the current agent on $p(\vec{u})@ag$, or by a different agent ag' on $p(\vec{u})@ag'$ (later);

  - if ID is a universally quantified variable with the scope the denial, then the denial goal is replaced by the set of denial goals obtained by binding ID to all the agent identifiers. Semantically, this means the denial goal must be solved by all the agents.
- if an abducible is collected to the hypotheses store by the current agent, it is tagged with all other agents' identifiers;
- if there is an abducible in the hypotheses store that is tagged by the current agent's identifier, the set of denial goals generated by resolving the abducible with the current agent's integrity constraints is added to the remaining goals store, and the tag is removed.

Each inference step may result in zero or more next states. A state is called a *transitional state* if any of the following conditions is satisfied: (1) it contains only askable goals, or denial goals consist of only askables, where none of the askables has an agent ID equal to the current agent's identifier; (2) it has no remaining goal but at least one collected abducible is tagged. A state is called a *failure state* if it is not a solved or transitional state, and no local inference rule is applicable to a selected goal or no goal can be selected. Thanks to the safe goal selection strategies, the whole local state rewriting process can be visualised as a *local abductive derivation tree*, whose leaf states can only be failure, transitional and solved states.

4.2 Coordination

When a query is received by an agent, the agent starts a local abduction with an *initial state*, whose pending goals are the literals in the query and all other stores are empty. If a transitional state is derived, it can be passed to a *suitable* agent for further processing (i.e., the recipient agent will start another local abduction with the state). If a solved state is derived, an answer $\langle \Delta, \theta \rangle$, where Δ is the set of collected abducibles and θ is the set of variable substitutions induced by the constraints store, is returned to the query issuer.

4.2.1 Transitional States and Confidentiality

A transitional state is one that can be passed between agents. It can be seen as a specification of *agent collaboration* – intuitively, the hypotheses store and the constraints store record the partial answer, the pending goals are the remaining reasoning tasks to be solved by relevant agents, the denials store includes the *global integrity constraints* that must not be violated by the agents during their reasoning, and finally the abducible tagging information is used to ensure consistency checks on the abducibles by all the agents, which may themselves result in new global integrity constraints. Under the safe goal selection strategies (1) an agent is forced to process a state as much as possible, and (2) no private non-abducible predicates can appear in transitional states, so preserving confidentiality.

4.2.2 State Recipient Agent Selection

When a transitional state is derived by an agent, a recipient agent is identified so that the state can be passed on for further processing. This uses a *helpfulness ranking* algorithm. We say "an agent ag may help with an askable $p(\vec{u})@ID$" if ID is ag's identifier or ID is a variable and $ID = ag$ is satisfiable; and we say "an agent ag may help with a (collected) abducible" if the abducible is tagged with ag's identifier. Given a transitional state, we compute the *helpfulness* of each agent in the system by summing up the total number of askable goals, denial goals (containing an askable) and abducibles that the agent may help with, and sort the agents according to the their helpfulness. One of the agents with highest value of helpfulness is then selected as the state recipient. However, it may occur that no

agent can help with a transitional state (i.e., agents' helpfulness value is 0). This may happen when the state contains an askable goal, $p(\vec{u})@ID$, with ID being a variable and has been passed around all agents once but no agent is able to solve it by bounding ID to its own identifier. In this case, the transitional state is treated as a failure state.

4.2.3 Backtracking with Concurrent Computation

Each local abduction constructs a local abductive derivation tree in search for solved states. If a transitional state is derived, this is sent out and another local abduction is initiated by a different agent. It can be shown that merging all local derivation trees would give a *global abductive derivation tree* equivalent to the one that would be constructed by the ASystem algorithm for the same query but over the merged agents' background knowledge and integrity constraints. This is, in fact, the formal basis for the correctness of our distributed algorithm. However, in practice such centralised reasoning is not always feasible and from an operational perspective we are interested in coordinations of local abductions that strike a balance between performance and communication. In our system, after an agent sends a transitional state to another agent, it can continue its local abduction while the recipient is working on a different local abductive reasoning task. Agents then essentially construct and explore different parts of the global derivation tree concurrently, providing a better performance than a fully sequential reasoning process. Each agent may derive more than one transitional state during its local abduction. If transitional states are sent as soon as they are derived, the communication channels between agents may quickly "flood" and agents become overloaded, as they may receive several states and perform several unfinished local abductions at the same time. To address these issues without sacrificing concurrent computation, a token based *synchronised backtracking* coordination strategy is adopted:

1. when the query issuer sends a query to an agent, it also sends a *token*. The agent creates the initial state and starts a local abduction;
2. during the local abduction, if the agent derives a transitional state,
   - **a.** if it has the token, then it sends out the transitional state with the token (i.e., it will no longer have the token);
   - **b.** otherwise, it buffers the transitional state;

   and in both cases it continues the local abduction;
3. once an agent receives a transitional state and the token, it initiates a local abduction and keeps the token;
4. if an agent derives a solved state, it sends the extracted answer to the query issuer regardless if it has the token or not, and continues the local abduction (i.e., to search for more solutions);
5. if an agent finishes a local abduction (i.e., finishes constructing and exploring a local abductive derivation tree), then
   - **a.** if it has the token, it returns the token as a *backtracking* signal to the agent who passed the transitional state;
   - **b.** otherwise, it waits for the backtracking signal;
6. after an agent receives a backtracking signal (i.e., it regains the token),
   - **a.** if there are buffered transitional states, then one of them is sent out with the token (i.e., the agent loses the token again);
   - **b.** otherwise, the agent stores the token and continues the local abduction if it has not yet completed, or the agent sends a backtracking signal as in 5(a) if finished;
7. if the query issuer no longer needs further answers, it sends a *discard* message to all the agents, so they will stop all relevant local abductions and remove relevant buffered states.

Note that with such a coordination strategy, an agent may still receive a transitional state while it still has some unfinished local abduction. Because of the *synchronised backtracking*, the current local abduction computation can be interrupted until a new local abduction is finished, and then resumed.

4.2.4 Soundness and Completeness

The distributed algorithm is *sound* and *complete* only with respect to a three-valued semantics [13] for which, given a global query $\mathcal{Q}$ and a global abductive answer $\langle \theta, \Delta \rangle$, the interpretation (completion) of all abducibles (i.e. $\mathcal{AB}$) is defined as $I_{\Delta\theta} = \{A^{\mathbf{t}} \mid A \in \Delta\theta \wedge A \in \mathcal{AB}\} \cup \{A^{\mathbf{f}} \mid A \notin \Delta\theta \wedge A \in \mathcal{AB}\}$.

▶ **Theorem 5.** **(Soundness)** *Given a global abductive framework $\langle \Sigma, \widehat{\mathcal{F}} \rangle$ and a global query $\mathcal{Q}$, if there is a successful global derivation for $\mathcal{Q}$ with answer $\langle \Delta, \theta \rangle$, then 1. $\bigcup_{i \in \Sigma} \Pi_i \cup I_{\Delta\theta} \models_3 \mathcal{Q}\theta$, and 2. $\bigcup_{i \in \Sigma} \Pi_i \cup I_{\Delta\theta} \models_3 \bigcup_{i \in \Sigma} \mathcal{IC}_i$, where $\models_3$ is the logical entailment under the three-valued semantics for abductive logic programs [13].*

▶ **Theorem 6.** **(Completeness)** *Let $\langle \Sigma, \widehat{\mathcal{F}} \rangle$ be a global abductive framework and $\mathcal{Q}$ a global query, suppose there is a finite global derivation tree T for $\mathcal{Q}$. If $\bigcup_{i \in \Sigma} \Pi_i \cup \bigcup_{i \in \Sigma} \mathcal{IC}_i \cup \exists \mathcal{Q}$ is satisfiable under the three-valued semantics, then T contains a successful branch.*

Our DAREC^2 system is a customisation of the DAREC system, i.e., special goal selection and agent interaction strategies are adopted. With these strategies, askable atoms can be seen as normal non-abducible atoms where the agent ID argument has the set of (the identifiers) the agents as domain. The DAREC system is sound and complete with respect to any goal selection and agent interaction strategies, and therefore our system inherits these properties (the reader is referred to [8] for outline proofs). The choice of three-valued completion semantics instead of other stronger semantics (e.g. stable model semantics [4]), is because top-down inference procedures, like abduction, may suffer from looping. In practice, looping can be avoided either by implementing a depth-bounded search strategy or by ensuring that the overall logic program satisfies certain properties (e.g., abductive non-recursive [14]).

4.2.5 Implementation

Our system has been implemented with YAP Prolog 6 [2]. An inequality solver extending the standard unification algorithm has also been implemented. For example, given $f(1, p(X)) \neq f(Y, p(2))$, the solver will answer $1 \neq Y$ or $X \neq 2$. The system uses the finite domain constraint solver (`CLP(FD)`) by YAP and the inequality solver for handling the arithmetic constraints and inequalities during the local inferences.

Agents in the system use TCP messages for peer-to-peer communications. For optimisation purposes, the system allows the option of using a "yellow page directory". When enabled, each agent maintains a copy of the directory to record *agent advertisements*. An advertisement contains an agent's identifier, the set of askables defined by the agent, and the set of abducibles *regulated* by the agent (i.e., the abducible that appears in an integrity constraint of the agent). When a new agent joins the system, it broadcasts its advertisements, so they are added to all other agents' directory. When an agent leaves, its advertisements are removed from everyone's directory. The directory can be used to further reduce communications and local computations: (1) abducibles no longer need to be checked (tagged) by agents that do

[2] `http://www.dcc.fc.up.pt/~vsc/Yap/`

not regulate them; (2) denials, with an askable whose agent ID is a universally quantified variable, no longer need to be checked by agents that do not know the askable; (3) in deciding recipient agents for a transitional state, agents that do not define the askable in question, are no longer considered. Note that the directory does not contain private atoms of any agent, and therefore its use does not affect the confidentiality property of the system.

5 Distributed Policy Analysis

A policy analysis framework has been proposed in [3]. The framework provides a specification language to represent security policies (*authorisations* and *obligations*), and system domain, as normal logic programs. Various policy analysis tasks can be solved using abduction. Modality conflict, for instance, can be defined as an abductive reasoning task where the goal is the negation of the property to analyse and the conditional answer is an example of system execution that proves this goal, i.e. a counterexample to the property. An example of analysis of *separation of duty* (SoD) in the context of security policies distributed over networks is given below.

▶ **Example 7.** In a Role-based Access Control (RBAC) system, the permission of an action depends on the role(s) assigned to the subject. Assume a RBAC corporate network with one team agent ($team1$) and two administrator agents ($admin1$ and $admin2$). The team1 is responsible for taking orders which can only be completed by an *initiator* and a *verifier* together, and who cannot be the same person, (i.e. the SoD concept). Before team1 can accept an order, it needs to check whether it can have clerks with suitable roles available to perform the two actions. The role assignments can only be performed by independent administrators, whose duties/privileges are also separated – they manage different role assignments and have without centralising the knowledge of the agents.

A distributed network, such as in the example, often consists of multiple nodes, each of which may have its own policies that are not shareable with others, but which may depend on each other. Centralised policy analysis for the whole network is not possible because of the confidentiality concern. Our system can be applied directly to this class of problems as a simulator of the distributed network.

Let's elaborate the example further. For simplicity, we only describe the policies relevant to the aforementioned analysis tasks. The team agent has local knowledge about team members and administrators, as well as the effects of roles assignments. It also has a rule specifying that role assignments are decided by the administrators. The system domain is *dynamic*, as executed actions may change its properties, and it is modelled using a set of domain independent Event Calculus axioms. Thus, Π_{team1} contains at least the following rules (by convention, Su, Ta, Ac and F are variables of the sorts *subjects*, *targets*, *actions* and *fluents* respectively, and $T, T_1, T_2, \ldots$ are variables of the sort *time*):

$$
\begin{aligned}
&holds(F,T) \leftarrow initially(F), 0 < T, \neg clipped(0,F,T).\\
&holds(F,T) \leftarrow do(Su,Ta,Ac,T1), initiates(Su,Ta,Ac,F,T1),\\
&\qquad\qquad 0 < T1, T1 < T, \neg clipped(T1,F,T).\\
&clipped(T1,F,T) \leftarrow do(Su,Ta,Ac,T2), terminates(Su,Ta,Ac,F,T2),\\
&\qquad\qquad T1 < T2, T2 < T.\\
&do(Su,Ta,Ac,T) \leftarrow Ac = assign(Role),\\
&\qquad\qquad request(Su,Ta,Ac,T), permitted(Su,Ta,Ac,T)@Su.\\
&initiates(Su,Ta,assign(Role),hasRole(Ta,Role),T) \leftarrow\\
&\qquad\qquad holds(clerk(Ta),T), holds(admin(Su),T).\\
&holds(clerk(X),T) \leftarrow X \in \{alex, bob\}.\\
&holds(admin(X),T) \leftarrow X \in \{admin1, admin2\}.\\
&holds(canInitiateOrder(X),T)@team1 \leftarrow holds(hasRole(X,initiator),T).\\
&holds(canVerifyOrder(X),T)@team1 \leftarrow holds(hasRole(X,verifier),T).
\end{aligned}
$$

One of the administrators, $admin1$, can assign the `verifier` role to users. It has a *blanket* policy rule *"a user can be assigned a role if it is permitted by at least one local policy"*. Blanket policy rules that can be used for modelling conflict resolutions are also private to the agent, e.g., in $admin1$, positive authorisation policies (rules with *permitted* as head) has higher priority than negative authorisation policies (rules with *denied* as head). In addition, $admin1$ maintains a local *trust level* database of the users and has a local positive authorisation policy *"the* `verifier` *can be assigned to a user if the user's trust level is greater than 4"*. Thus, Π_{admin1} contains at least the following rules:

$$
\begin{aligned}
&permitted(admin1, Ta, assign(Role), T)@admin1 \leftarrow \\
&\quad holds(managed_role(Role), T), permitted(admin1, Ta, assign(Role), T). \\
&permitted(admin1, Ta, assign(verifier), T) \leftarrow holds(trust_level(Ta, L), T), L > 4. \\
&holds(trust_level(alex, 3), T). \\
&holds(trust_level(bob, 5), T). \\
&holds(managed_role(verifier), T).
\end{aligned}
$$

The other administrator, $admin2$, can assign the `initiator` role to users. It has a different blanket policy rule *"a user can be assigned a role if none of the local policies denies it"* (i.e., negative authorisation has higher priority). It also tries to implement SoD by having a local negative authorisation policy *"a user cannot be assigned to the* `initiator` *role if it has been assigned the conflicting* `verifier` *role"*. Thus, Π_{admin2} contains at least of the following:

$$
\begin{aligned}
&permitted(admin2, Ta, assign(Role), T)@admin2 \leftarrow \\
&\quad holds(managed_role(Role), T), \neg denied(admin2, Ta, assign(Role), T). \\
&denied(admin2, Ta, assign(initiator), T) \leftarrow \\
&\quad holds(conflicting(initiator, Role), T), request(Su, Ta, assign(Role), T1), T1 < T. \\
&holds(managed_role(initiator), T). \\
&holds(conflicting(initiator, verifier), T).
\end{aligned}
$$

To check if team1 can fulfil a task with the administrators under the SoD constraint, we can use the query $\exists X, Y, T, Z.[holds(canInitiateOrder(X), T)@Z, holds(canVerifyOrder(Y), T)@Z \wedge X \neq Y]$. Our system can succeed the query and find one answer: $Ans_1 = request(admin1, bob, assign(verifier), T1) \wedge request(admin2, alex, assign(initiator), T2) \wedge T1 < T \wedge T2 < T \wedge Z = team1$. To check if the existing administrators' policies can guarantee the SoD security property of the overall system, i.e., *"it is not possible that someone can complete an order alone"*, we can use the (negated) query $\exists X, T, Z.[holds(canInitiateOrder(X), T)@Z, holds(canVerify-Order(X), T)@Z]$. Our system can also succeed this query and find one answer: $Ans_2 = req(admin1, bob, assign(verifier), T1) \wedge req(admin2, bob, assign(initiator), T2) \wedge T1 \leq T2 \wedge T2 < T \wedge Z = team1$. Therefore, the existing policies are not sufficient to guarantee the SoD property, and the administrators have to revise their policies.

5.1 Benchmarking

To study the scalability of DAREC[2] in distributed policy analysis, an auto-testing environment has been developed. Given a set of tunable parameters, the environment is able to generate policy rules and action effect rules with randomised conditions conforming to the language in [3]. These rules are then distributed among a specified number of agents. Note that the language in [3] guarantees the overall system model (as a logic program) is abductive acyclic and hence the executions of policy analysis queries always terminate. For each distributed computation, the total number of messages exchanged between agents, the average ping time and the total time for computing all solutions for a given query are recorded. For example, empirical results (e.g., 5 agents, each having about 250 rules; each rule body has on

average 10 conditions, 3 of which are askables) showed that for about two thirds of the tested queries, the DAREC^2 computation is about 1.26∼6.17 times faster than a computation using abduction over the centralised rules with an average ping time between agents being 0.015ms. This was expected as concurrent computation was performed during collaborative reasoning.

6 Conclusion and Future Work

Confidentiality in knowledge is one important constraint that makes a multi-agent reasoning problem challenging, and it is also a very common assumption in MAS's. The main contributions of this paper include (1) a logical framework for modelling the distributed knowledge of a multi-agent system where the agent knowledge bases are correlated and have private information, and (2) a top-down distributed abductive algorithm which allows agents to perform collaborative hypothetical reasoning without disclosing private information. By limiting the set of abducible predicates to be empty, the system becomes a general purpose distributed deductive theorem prover that performs constructive negation. This feature is very useful when dealing with logic programs with unbounded domains (e.g., Π_{admin2} in Example 7) that cannot be implemented using bottom-up algorithms like answer set programming. The system has many potential applications including distributed security policy analysis. As a future work, we aim to perform more benchmarking to investigate the performance under different safe goal selection strategies, agent selection strategies and agent interaction strategies, and extend our system to handle private abducible predicates.

References

1. G. Bourgne, K. Inoue, and N. Maudet. Abduction of distributed theories through local interactions. In *Proceedings of the 19th European Conference on Artificial Intelligence*, pages 901–906, 2010.
2. A. Ciampolini, E. Lamma, P. Mello, F. Toni, and P. Torroni. Cooperation and competition in ALIAS: a logic framework for agents that negotiate. *Annals of Mathematics and Artificial Intelligence*, 37(1–2):65–91, 2003.
3. R. Craven, R. Lobo, J. Ma, A. Russo, E.C. Lupu, and A. Bandara. Expressive policy analysis with enhanced system dynamicity. In *Proceedings of the 4th International Symposium on Information, Computer, and Communications Security*, pages 239–250, 2009.
4. M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proceedings of the 5th International Conference of Logic Programming*, 1988.
5. A.C. Kakas and P. Mancarella. Abductive logic programming. In *Proceedings of the Workshop Logic Programming and Non-Monotonic Logic*, pages 49–61, 1990.
6. A.C. Kakas, B. Van Nuffelen, and M. Denecker. A-system: Problem solving through abduction. In *Proceedings of 17th International Joint Conference on Artificial Intelligence*, pages 591–596, 2001.
7. J. Ma, A. Russo, K. Broda, and K. Clark. DARE: a system for distributed abductive reasoning. *Journal of Autonomous Agents and Multi-Agent Systems*, 16(3):271–297, 2008.
8. J. Ma, A. Russo, K. Broda, and E. Lupu. Distributed abductive reasoning with constraints. In *Post-proceedings of the 8th International Workshop on Declarative Agent Languages and Technologies*, 2010.
9. H. Nabeshima, K. Iwanuma, K. Inoue, and O. Ray. Solar: An automated deduction system for consequence finding. *AI Communications*, 23(2-3):183–203, March 2010.
10. Bert Van Nuffelen. *Abductive Constraint Logic Programming: Implementation and Applications.* PhD thesis, Department of Computer Science, K.U.Leuven, 2004.

11 K. Satoh, K. Inoue, K. Iwanuma, and C. Sakama. Speculative computation by abduction under incomplete communication environments. In *Proceedings of the 4th International Conference on Multi-Agent Systems*, pages 263–270, 2000.

12 M.P. Sindlar, M.M. Dastani, F. Dignum, and J.C. Meyer. Mental state abduction of bdi-based agents. pages 161–178, 2009.

13 Frank Teusink. Three-valued completion for abductive logic programs. *Theoretical Computer Science*, 165(1):171–200, 1996.

14 Sofie Verbaeten. Termination analysis for abductive general logic programs. In *International Conference on Logic Programming*, pages 365–379, 1999.

BAAC: A Prolog System for Action Description and Agents Coordination*

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

1 **Univ. di Udine, Dip. di Matematica e Informatica.** agostino.dovier@uniud.it
2 **Univ. di Perugia, Dip. di Matematica e Informatica.** formis@dmi.unipg.it
3 **New Mexico State University, Dept. Computer Science.** epontell@cs.nmsu.edu

Abstract

The paper presents a system for knowledge representation and coordination, where autonomous agents reason and act in a shared environment. Agents autonomously pursue individual goals, but can interact through a shared knowledge repository. In their interactions, agents deal with problems of synchronization and concurrency, and have to realize coordination by developing proper strategies in order to ensure a consistent global execution of their autonomously derived plans. This kind of knowledge is modeled using an extension of the action description language $\mathcal{B}$. A distributed planning problem is formalized by providing a number of declarative specifications of the portion of the problem pertaining a single agent. Each of these specifications is executable by a stand-alone CLP-based planner. The coordination platform, implemented in Prolog, is easily modifiable and extensible. New user-defined interaction protocols can be integrated.

1998 ACM Subject Classification I.2.3 Deduction and Theorem Proving, I.2.11 Distributed Artificial Intelligence

Keywords and phrases Knowledge Representation, Multi-Agent Systems, Planning, CLP

Digital Object Identifier 10.4230/LIPIcs.ICLP.2011.187

1 Introduction

Representing and reasoning in multi-agent domains are two of the most active research areas in *multi-agent system (MAS)* research. The literature in this area is extensive, and it provides a plethora of logics for representing and reasoning about various aspects of MAS, e.g., [13, 9, 17, 15, 7]. Several logics proposed in the literature have been designed to specifically focus on particular aspects of MAS, often justified by a specific application scenario. This makes them suitable to address specific subsets of the general features required to model real-world MAS domains. The task of generalizing these proposals to create a uniform and comprehensive framework for modeling different aspects of MAS domains is an open problem. We do not dispute the possibility of extending the existing proposals in various directions, but the task is not easy. Similarly, a variety of multi-agent programming platforms have been proposed, mostly in the style of multi-agent programming languages, e.g., Jason, ConGolog, 3APL, GOAL [1, 4, 3, 10], but with limited planning capabilities.

Our effort here is on developing a multi-agent system for knowledge representation based on a high-level action language. The starting point of this work is the action language $\mathcal{B}^{MV}$ [6]; this is a flexible single-agent action language, that generalizes the action language $\mathcal{B}$ [8], with support for multi-valued fluents, non-Markovian domains, and constraint-based

* This work is partially supported by INdAM-GNCS 2010, INdAM-GNCS 2011, and PRIN 20089M932N projects, and NSF grants 0812267, 0754525, and 0420407.

© Agostino Dovier, Andrea Formisano, and Enrico Pontelli;
licensed under Creative Commons License NC-ND
Technical Communications of the 27th International Conference on Logic Programming (ICLP'11).
Editors: John P. Gallagher, Michael Gelfond; pp. 187–197

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

formulations (which enable, for example, the formulation of costs and preferences). In this work, we propose a further extension to support MAS scenarios. The perspective is that of a distributed environment, with agents pursuing individual goals but capable of interacting through shared knowledge and concurrent actions. A first step in this direction has been described in the $\mathcal{B}^{\mathsf{MAP}}$ language [5]; $\mathcal{B}^{\mathsf{MAP}}$ extends $\mathcal{B}^{MV}$ providing a multi-agent action language with capabilities for *centralized* planning. In this paper, we embed $\mathcal{B}^{\mathsf{MAP}}$ into a truly distributed multi-agent platform. The language is extended with Communication primitives for modeling interactions among Autonomous Agents. We refer to this language as $\mathcal{B}^{\mathsf{AAC}}$. Differently from [5], agents can have private goals and are capable of developing independent plans. Agents' plans are developed in a distributed fashion, leading to replanning and/or to the introduction of coordination actions to enable a consistent global execution. The system is implemented in SICStus Prolog, using the libraries `clpfd` and `linda`.

2 Syntax of the Multi-agent Language $\mathcal{B}^{\mathsf{AAC}}$

The signature of the language $\mathcal{B}^{\mathsf{AAC}}$ consists of a set $\mathcal{G}$ of *agent* names, used to identify the agents in the system, a (unique) set $\mathcal{F}$ of *fluent* names, a set $\mathcal{A}$ of *action* names, and a set $\mathcal{V}$ of values for the fluents in $\mathcal{F}$—we assume $\mathcal{V} = \mathbb{Z}$. The behavior of each agent a is specified by an action description theory $\mathcal{D}_a$, i.e., a collection of axioms of the forms described next.

Name and priority of the agent a are specified in $\mathcal{D}_a$ by *agent declarations*:

$$\texttt{agent } a \texttt{ [priority } n \texttt{]} \tag{1}$$

where $n \in \mathbb{N}$. 0 (default value) denotes the highest priority. Priorities might be used to resolve conflicts among actions of different agents. Agent a can access only those fluents that are declared in $\mathcal{D}_a$ by axioms of the form:

$$\texttt{fluent } f_1, \dots, f_h \texttt{ valued } dom \tag{2}$$

with $f_i \in \mathcal{F}$, $h \geq 1$, and $dom \subset \mathcal{V}$ is a set of values representing the admissible values for $f_1, \dots, f_h$ (possibly represented as an interval $[v_1, v_2]$). We refer to these fluents as the "local state" of agent a. Fluents accessed by multiple agents are assumed to be defined consistently.

▶ **Example 1.** Let us specify a domain inspired by volleyball. There are two teams: *black* and *white*, with one player in each team; let us focus on the domain for the white team (Sect. 4 deals with the case that involves more players). We introduce fluents to model the positions of the players and of the ball, the possession of the ball, the score, and a numerical fluent `defense_time`. All players know the positions of all players. Since the teams are separated by the net, the x-coordinates of a black and white players must differ. This can be stated by:

```
agent player(white,X) :- num(X).
known_agents player(black,X) :- num(X).
fluent x(player(white,X)) valued [B,E] :- num(X), net(NET),B is NET+1, linex(E).
fluent x(player(black,X) valued [1,E] :- num(X), net(NET),E is NET-1.
fluent y(A) valued [1,MY] :- player(A), liney(MY).
fluent x(ball) valued [1,MX] :- linex(MX).
fluent y(ball) valued [1,MY] :- liney(MY).
fluent hasball(A) valued [0,1] :- agent(A).
fluent point(T) valued [0,1] :- team(T).
fluent defense_time valued [0,1].
team(black). team(white). num(1). linex(11). net(6). liney(5).
```

where `linex`, and `liney` are the field sizes, and `net` is the x-coordinate of the net. ◻

Fluents are used in *Fluent Expressions* (FE), which are defined as follows:

$$\texttt{FE} \;\; ::= \;\; n \mid f^t \mid f\texttt{@}r \mid \texttt{FE}_1 \oplus \texttt{FE}_2 \mid -(\texttt{FE}) \mid \texttt{abs}(\texttt{FE}) \mid \texttt{rei}(\texttt{C}) \qquad (3)$$

where $n \in \mathcal{V}$, $f \in \mathcal{F}$, $t \in \{0, -1, -2, -3, \dots\}$, $\oplus \in \{+, -, *, /, \texttt{mod}\}$, and $r \in \mathbb{N}$. FE is referred to as a *timeless expression* if it contains no occurrences of f^t with $t \neq 0$ and no occurrences of $f\texttt{@}r$. f can be used as a shorthand of f^0. The notation f^t is an *annotated* fluent expression. The expression refers to the value f had $-t$ steps in the past. An expression of the form $f\texttt{@}r$ denotes the value f has at the rth step in the evolution of the world (i.e., an *absolute* point in time). We use the expression $\texttt{pair}(\texttt{FE}_1,\texttt{FE}_2)$ to encode a pair, and the projection functions $\texttt{x}(\cdot)$ and $\texttt{y}(\cdot)$ such that $\texttt{x}(\texttt{pair}(a, b)) = a$ and $\texttt{y}(\texttt{pair}(a, b)) = b$. The reified expression rei(C) represents a Boolean value indicating the truth value of the constraint C.

A *Primitive Constraint* (PC) is formula $\texttt{FE}_1$ op $\texttt{FE}_2$, where $\texttt{FE}_1$ and $\texttt{FE}_2$ are fluent expressions, and op $\in \{=, \neq, \geq, \leq, >, <\}$. A *constraint* C is a propositional combination of PCs. As a syntactic sugar, f++ (f--) denotes the primitive constraint $f = f^{-1} + 1$ ($f = f^{-1} - 1$).

An axiom of the form action x in $\mathcal{D}_a$, declares that the action $x \in \mathcal{A}$ is available to the agent a. The same action name x can be used by different agents. A special action, nop is always executable by every agent, and it causes no changes to any of the fluents.

▶ **Example 2.** The actions for each player A of Example 1 are:

- A : $\texttt{move}(d)$ one step in direction d, where d is one of the eight directions: north, north-east, ..., west, north-west (i.e., analogous to the moves of a King on a chess-board).
- A : $\texttt{throw}(d, f)$ the ball in direction d (same eight directions as above) with a strength f varying from 1 to a maximum throw power (5 in our example).

Moreover, the player of each team is in charge of checking if a point has been scored (in such case, he whistles). We write the actions as act([A],action_name) and state these axioms:

```
action act([A],move(D)) :- whiteplayer(A),direction(D).
action act([A],throw(D,F)) :- whiteplayer(A),direction(D),power(F).
action act([player(white,1)],whistle).
```

where whiteplayer, power, and direction can be defined as follows:

```
whiteplayer(player(white,N)) :- agent(player(white,N)).
power(1). power(2). power(3). power(4). power(5).
direction(D) :- delta(D,_,_).    delta(nw,-1,1). delta(n,0,1). delta(ne,1,1).
delta(w,-1,0). delta(e,1,0). delta(sw,-1,-1). delta(s,0,-1). delta(se,1,-1).
```
□

The executability of the actions is described by axioms of the form:

$$\texttt{executable}\ x\ \texttt{if}\ \texttt{C} \qquad (4)$$

where $x \in \mathcal{A}$ and C is a constraint, stating that C has to be entailed by the current state in order for x to be executable. We assume that at least one executability axiom is present for each action x. Multiple executability axioms are treated as a disjunction.

▶ **Example 3.** In our working example, we can state executability as follows:

```
executable act([player(white,1)],whistle) if [S eq 0] :- build_sum(S).
executable act([A],move(D)) if [hasball(A) eq 0, defense_time gt 0,
      Net lt x(A)+DX, x(A)+DX leq MX, 1 leq y(A)+DY, y(A)+DY leq MY] :-
        action(act([A],move(D))), delta(D,DX,DY), net(Net), linex(MX), liney(MY).
executable act([A],throw(D,F)) if
      [hasball(A) gt 0,defense_time eq 0, 1 leq x(A)+DX*F, x(A)+DX*F leq MX,
      1 leq y(A)+DY*F, y(A)+DY*F leq MY] :-
           action(act([A],throw(D,F))), delta(D,DX,DY), linex(MX), liney(MY).
```

These axioms state that neither a player nor the ball can leave the field. build_sum is recursively defined to return the expression: $\texttt{defense_time} + \texttt{hasball}(A_1) + \cdots + \texttt{hasball}(A_n)$

where $A_1, \ldots, A_n$ are the players (i.e., `player(white,1)` and `player(black,1)`). Let us observe that $=, \neq, \leq, <$, etc. are concretely represented by `eq`,`neq`,`leq`,`lt`, respectively. ☐

The effects of an action are described by axioms (*dynamic causal laws*) of the form:

$$x \ \texttt{causes} \ \mathit{Eff} \ \texttt{if} \ \mathit{Prec} \tag{5}$$

where $x \in \mathcal{A}$, *Prec* is a constraint, and *Eff* is a conjunction of primitive constraints of the form $f = FE$, where $f \in \mathcal{F}$. The axiom asserts that if *Prec* is `true` w.r.t. the current state, then *Eff* must hold after the execution of x. Since agents share fluents, their actions may cause inconsistencies. A *conflict* happens when the effects of different actions lead to an inconsistent state; a procedure has to be applied to resolve conflicts and determine a consistent subset of the conflicting actions (Sect. 3.2). A *failure* occurs whenever an action x cannot be executed as planned by an agent a as a consequence of the above procedure.

▶ **Example 4.** Let us state the effects of the actions in the volleyball domain. When the ball is thrown, with force f, in direction d, it reaches a destination cell whose distance is as follows: a) if d is either north or south then $\Delta X = 0, \Delta Y = f$; b) if d is east or west then $\Delta X = f, \Delta Y = 0$; c) if d is any other direction, $\Delta X = f, \Delta Y = f$. As a further effect of throw, the fluent `defense_time` is set (to 1 in our example).

```
actocc([A],throw(D,F)) causes hasball(A) eq 0 :- action(act([A],throw(D,F))).
actocc([A],throw(D,F)) causes defense_time eq 1 :- action(act([A],throw(D,F))).
actocc([A],throw(D,F)) causes pair(x(ball),y(ball)) eq
      pair(x(A)^-1+ F*DX,y(A)^-1+ F*DY) :-
      action(act([A],throw(D,F))), delta(D,DX,DY).
actocc([A],throw(D,F), causes hasball(B) eq 1
      if [pair(x(B),y(B)) eq pair(x(A)+F*DX, y(A)+F*DY)] :-
      action(act([A],throw(D,F))), player(B), neq(A,B),delta(D,DX,DY).
actocc([A],throw(D,F)) causes point(black) eq 1 if [x(A)+F*DX eq Net] :-
      action(act([A],throw(D,F))), delta(D,DX,_), net(Net).
```

The effects of the other two actions `move` and `whistle` can be stated by:

```
actocc([player(white,1)],whistle) causes point(white) eq 1
      if [x(ball) lt NET] :- net(NET).
actocc([player(white,1)],whistle) causes point(black) eq 1
      if [NET lt x(ball)] :- net(NET).
actocc([A],move(D)) causes pair(x(A),y(A)) eq pair(x(A)^-1+DX,y(A)^-1+DY) :-
      action(act([A],move(D))), delta(D,DX,DY).
actocc([A],move(D)) causes defense_time-- :- action(act([A],move(D))).
actocc([A],move(D)) causes hasball(A) eq 1
      if [pair(x(ball),y(ball)) eq pair(x(A)+DX,y(A)+DY)] :-
      action(act([A],move(D))), delta(D,DX,DY).
```

Let us observe here that we concretely used `actocc` instead of `act`. This has been introduced to give the idea of the occurrence of an action. ☐

At least two perspectives can be followed, by assigning either a passive or an active role to the conflicting agents during conflict resolution. In the first case, a supervising entity is in charge of resolving the conflicts, and all the agents will adhere to the supervisor's decisions. Alternatively, the agents are in charge of reaching an agreement, possibly through negotiation. The following declarations describe basic reaction policies the agents can use:

$$\texttt{action} \ x \ [OPT] \tag{6}$$

where:

OPT ::= `on_conflict` OC $[OPT]$ | `on_failure` OF $[OPT]$
OC ::= `retry_after` T `[provided` C`]` | `forego [provided` C`]` | `arbitration`
OF ::= `retry_after` T `[if` C`]` | `replan [if` C`] [add_goal` C`]` | `fail [if` C`]`

In these axioms one can also specify policies to be adopted when a failure occurs during action execution. Reacting to a failure is a "local" activity the agent performs after the state transition has been completed. In the axioms (6), one can specify different reactions to a conflict (resp. a failure) of the same action, to be considered in their order of appearance.

Apart from the communications occurring among agents during conflict resolution, other forms of "planned" communication can be modeled in an action theory. An agent might seek help from other agents to make a constraint true. The request can be broadcast to all known agents or sent to some specific agents. The agent can optionally offer a "reward" in case of acceptance of the proposal. This allows us to model negotiations and bargaining.

$$\texttt{request}\ C_1\texttt{[to_agent}\ a'\texttt{] if}\ C_2\ \texttt{[offering}\ C_3\ \texttt{]} \tag{7}$$

Various *global constraints* can be exploited to impose control knowledge and maintenance goals representing properties that must persist. For example:

- FC `holds_at` n: the fluent constraint FC holds at the nth time step.
- `always` FC: the fluent constraint FC holds in all states of the evolution of the world.

A detailed description of these constraints and their semantics can be found in [6].

An *action domain description* consists of a collection $\mathcal{D}_a$ of axioms of the forms described so far, for each agent $a \in \mathcal{G}$. Moreover, it includes a collection $\mathcal{O}_a$ of goal axioms (objectives), of the form `goal C`, where `C` is a constraint, and a collection $\mathcal{I}_a$ of initial state axioms of the form: `initially C` where `C` is a constraint involving only timeless expressions. We assume that all the sets $\mathcal{I}_a$ are drawn from a consistent global initial state description $\mathcal{I}$, i.e., $\mathcal{I}_a \subseteq \mathcal{I}$. A specific instance of a planning problem is a triple $\left\langle \langle\mathcal{D}_a\rangle_{a\in\mathcal{G}}, \langle\mathcal{I}_a\rangle_{a\in\mathcal{G}}, \langle\mathcal{O}_a\rangle_{a\in\mathcal{G}} \right\rangle$. The problem has a solution only if $\langle\mathcal{O}_a\rangle_{a\in\mathcal{G}}$ characterizes a consistent state, i.e., there exists a consistent assignment of values to the fluents that satisfies the constraint $\bigwedge_{a\in\mathcal{G}} \bigwedge_{\texttt{goal}\,C\in\mathcal{O}_a} C$.

3 System Behavior

The behavior of $\mathcal{B}^{\mathsf{AAC}}$ can be split in two parts: the semantics of the action description languages used *locally* by each agent, ignoring the axioms (6) and (7), and the behavior of the *overall* system that deals with agents' interactions. Let us assume that there is an overall planning horizon length N. Due to space restrictions, we don't enter into the details of the "local" semantics which is given in terms of transition systems (as in [8]). The formal semantics of the language $\mathcal{B}^{MV}$, upon which $\mathcal{B}^{\mathsf{AAC}}$ is defined, is given in detail in [6].

3.1 Concurrent Plan Execution

The agents are autonomous and plan their activities independently. In executing their plans, the agents must take into account the effects of concurrent actions. We developed a basic communication mechanism among agents by exploiting a *tuple space*, realized using the Linda system [2]. Moreover, most of the interactions among concurrent agents, especially those aimed at resolving conflicts, are managed by a specific process, the *supervisor*, that also provides a global time to all agents, enabling them to execute their actions synchronously.

1. At the beginning, the supervisor acquires the initial state description $\mathcal{I} = \bigcup_{a\in\mathcal{G}} \mathcal{I}_a$.
2. At each time step the supervisor starts a new state transition:
   - Each active agent sends to the supervisor a request to perform an action specifying its effects on the (local) state.
   - The supervisor collects these requests and determines whether subsets of actions/agents are conflicting. A conflict occurs whenever agents require incompatible assignments of values to the same fluents. The transition takes place once all conflicts have

been resolved and a subset of compatible actions has been identified using one or more policies (see below). These actions are enabled while the remaining ones are inhibited.
- All the enabled actions are executed, yielding changes that define a new (global) state.
- These changes are then sent back to all agents, to update their local states. Those agents whose actions have been inhibited receive a failure message.

3. The computation stops when the time N is reached.

After each step of the local plan execution, the agents need to check if the reached state still supports their successive planned actions. If not, each agent has to reason locally and revise its plan, i.e., initiate a replanning phase. This may occur in two cases: **(a)** The proposed action was inhibited, so the agent actually executed a `nop`; this case occurs when the agent receives a failure message from the supervisor. **(b)** The interaction was successful, i.e., the planned action was executed, but the effects of the actions performed by other agents affected fluents in its local state, preventing the successful continuation of the rest of the plan—e.g., the agent a may have assumed that the fluent g maintained its value by inertia, but another agent changed such value. This might affect the executability of the next action of a's plan.

3.2 Conflicts Resolution

A conflict resolution procedure is performed by the supervisor whenever it identifies a set of incompatible actions. Different policies can be adopted in this phase and different roles can be played by the supervisor. First, the supervisor exploits the priorities of the agents to attempt a resolution of the conflict, by inhibiting the actions of low priority agents. If this does not suffice, further options are applied. Two simple options have been implemented in our prototype, assigning the active role either to the supervisor or to the conflicting agents. The architecture is modular, and can be extended with more complex policies.

- The supervisor has the *active role*—it decides which actions to inhibit. In the current prototype, the arbitration strategy is limited to **(1)** A random selection of a single action to be executed or **(2)** The computation of a maximal set of compatible actions to be executed. This computation is done by solving a dynamically generated CSP. In this strategy, the `on_conflict` policies assigned to actions by axioms (6) are ignored.
- The supervisor simply notifies the set of conflicting agents about the inconsistency of their actions. The agents involved in the conflict are completely in charge of resolving it via negotiation. The supervisor waits for a solution from the agents. In solving the conflict, each agent a makes use of one of the `on_conflict` directives (6) specified for its conflicting action x. The semantics of these directives are as follows (`[provided` C`]` is an optional qualifier; if omitted it will be interpreted as `provided true`):
  - The option `on_conflict arbitration` causes the execution of the supervisor which performs an arbitration phase to resolve the conflict, as previously described.
  - The option `on_conflict forego provided` C causes the agent a to "search" among the other conflicting agent for someone, say b, that can guarantee the condition C. In this case, b performs its action while the execution of a's action fails, and a executes a `nop` in place of its action x. Different strategies can be implemented in order to perform such a "search for help", e.g., a round-robin policy described below, but other alternatives are possible and should be considered in completing the prototype.
  - The option `on_conflict retry_after` T `provided` C, differs from the preceding one because a will execute `nop` during the following T time steps and then will try again to execute its action (provided that the preconditions of the action still hold).
  - If there is no applicable option (e.g., no option is defined or none of the agents accept to, or is able to, guarantee C), the action is inhibited and its execution fails.

The way in which agents exploit the `on_conflict` options can rely on several policies. In the current prototype we implemented a round-robin policy. Let us assume that the agents $a_1, \ldots, a_m$ aim at executing actions $z_1, \ldots, z_m$, respectively, and these actions are conflicting. The agents are sorted by the supervisor, and the agents take turn in resolving the conflict. Suppose that at a certain round j of the procedure the agent a_i is selected. It determines the j-th option for its action and tries to apply it. If the option is directly applicable or an agreement is reached with another agent on a condition C, then the two agents exit the procedure. If no arbitration is invoked, then the remaining agents will complete the procedure. If the option does not lead to an agreement, then the next agent in the sequence will start its active role in the round, while a_i will wait its next turn in round $j+1$. This procedure always ends with a solution to the conflict, since a finite number of `on_conflict` options are defined for each action. This a rigid policy, and it represents a simple example of how to realize a terminating protocol for conflict resolution. Alternative solutions can be added to the prototype thanks to its modularity.

Once all conflicts have been addressed, the supervisor applies the enabled actions, and obtains the new global state. Each agent receives a communication containing the outcome of its action execution and the changes to its local state. Moreover, further information might be sent to the participating agents, depending on the outcome of the coordination procedure. For instance, when two agents agree on an `on_conflict` option, they "promise" to execute specific actions (e.g., one agent may have to execute T consequent `nop`). This information has to be sent back to the interested agents to guide their replanning phases.

3.3 Failure Policies

Agents receiving a failure message from the supervisor need to revise their original plans to detect if the local goals can still be achieved. Different approaches can be used. For instance, one agent could avoid developing an entire plan at each step, but only produce a partial plan for the very next step. Alternatively, an agent could determine the "minimal" modifications to the existing plan in order to make it valid with respect to the new encountered state. At this time, the prototype includes only replanning from scratch at each step.

While replanning, the agent might exploit the `on_failure` options associated to the inhibited action. The intuitive semantics of these options is as follows. (The options declared for the inhibited action are considered in the given order, executing the first applicable one).

- `retry_after` T `[if` C`]`: the agent evaluates the constraint C; if C holds, then it will execute `nop` T times and then try again the failed action (provided it is executable).
- `replan [if` C_1`] [add_goal` C_2`]`: the agent evaluates C_1; if it holds, then in the replanning phase the goal C_2 will be added to the local goal. The `add_goal` C_2 is optional; if it is not present then nothing will be added to the goal, i.e., it is the same as `add_goal true`.
- `fail [if` C_1`]`: this is analogous to `replan [if` C_1`] add_goal false`. In this case the agent declares that it is impossible to reach its goal.
- If none of the above options is applicable, then the agent will proceed as if the option `replan if true` is present.

It might be the case that some global constraints (such as `holds_at` and `always`) involve fluents that are not known by any of the agents. Therefore, none of the agents is able to consider such constraints while developing his plan. These constraints have to be enforced when merging the individual plans. In doing so, the supervisor adopts the same strategies introduced to deal with conflicts and failures among actions, as described earlier.

3.4 Broadcasting and Direct Requests

Let us describe a simple protocol for implementing communications among agents, following an explicit request of the form (7). We assume that the current state is the i-th one of the plan execution. The handling of requests is interleaved with the agent-supervisor interactions; nevertheless, requests and offers are directly exchanged among agents. The main steps involved in a state transition, from the point of view of an agent a, are:

1. Agent a tries to execute its action and sends this information to the supervisor (Sect. 3.1).
2. Possibly after a coordination phase, a receives from the supervisor the outcome of its attempt to execute the action (namely, failure or success, the changes in the state, etc.)
3. If the action execution is successful, before declaring the current transition completed, the agent a starts an interaction with the other agents to handle pending requests. All the communications associated to such interactions are realized using Linda's tuple-space.

   3.a. Agent a fetches the collection H of all pending requests. For each request $h \in H$, e.g., originating from agent b, a decides whether to accept h. Such a decision might involve exploitation of the planning facilities, in order to determine if the requested condition can be achieved by a, possibly by modifying its original plan. If possible, a posts its offer into the tuple-space and waits for a rendezvous with b.

   3.b. Agent a checks whether there are replies to the requests it previously posted. For each request, a collects the set of offers/agents that expressed their willingness to help a and, by using some strategy, selects one of them, say b. The policy for choosing the responding agent can be programmed (e.g., by exploiting priorities, etc.). Once the choice has been made, a communicates with the selected agent, declares its availability to b, and communicates the fulfillment of the request to the other agents. The request is also removed from the tuple space, along with all the obsolete offers.
4. At that point, the transition is completed for agent a. By taking into account the information about the outcome of the coordination phase in solving conflicts (point (2)), the agreement reached in handling requests (point (3)), a might need to modify its plan. If the replanning phase succeeds, then a will execute the next action in its local plan.

3.5 Implementation Issues

A prototype of the system has been implemented in SICStus Prolog, using the libraries `clpfd` for reasoning (and the planners described in [5, 6]), and the libraries `system`, `linda/server`, and `linda/client` for process communication. The system is organized in modules and it is available, together with some sample domains at `www.dimi.uniud.it/dovier/BAAC`.

Each autonomous agent corresponds to an instance of the module `plan_executor`, which, in turn, relies on a planner for planning/replanning activities, and on `client` for interacting with other actors in the system. As previously explained, a large part of the coordination is guided by the module `supervisor`. Notice that both the `supervisor` and `client` act as Linda-clients. Conflict resolution functionalities are provided to the modules `client` and `supervisor` by the modules `ConflictSolver_client` and `ConflictSolver_super`, respectively. Finally, the `arbitration_opt` module implements the arbitration protocol(s).

Let us remark that all the policies exploited in coordination, arbitration, and conflict handling can be customized by providing a different implementation of individual predicates exported by the corresponding modules. For instance, to implement a conflict resolution strategy different from the round-robin described earlier, it suffices to add to the system a new implementation of the module `ConflictSolver_super` (and for `ConflictSolver_client`, if the specific strategy requires an active role of the conflicting agents). Similar extensions

can be done for `arbitration_opt`. A `settings.pl` file is available to enable specification of various parameters, e.g., the names of the files containing the action descriptions, the number of planning steps allowed, the selected conflict resolution strategies, etc.

As far as the planning module is concerned, we modified the interpreters of the $\mathcal{B}^{MV}$ and the $\mathcal{B}^{\mathsf{MAP}}$ languages [5, 6] to accept the coordination constructs described in this paper. The two planners, `sicsplan` and `bmap`, have been integrated in the system to process $\mathcal{B}^{MV}$ and $\mathcal{B}^{\mathsf{MAP}}$ theories. However, the system is open to further extensions and different planners (even not necessarily based on Prolog technology) can be easily integrated thanks to the simple interface with the module `plan_executor`, which consists of few Prolog predicates.

4 The Volleyball Domain

Let us describe a specification in $\mathcal{B}^{\mathsf{AAC}}$ of a coordination problem between two multi-agent systems. Let us extend the domains described in Examples 1–4. There are two teams: *black* and *white* whose objective is to score a point, i.e., to throw the ball in the field of the other team (passing over the net) in such a way that no player of the other team can reach the ball before it touches the ground. Each team is modeled as a multi-agent system that elaborates its own plan in a centralized manner (thus, each step in the plan consists of a set of actions).

The playing field is discretized by fixing a `linex`×`liney` rectangular grid that determines the positions where the players (and the ball) can move (see Fig. 1). The leftmost (rightmost) cells are those of the black (white) team, while the net ($x = 6$) separates the two subfields. There are p players per team ($p = 2$ in Fig. 1). The allowed actions are: `move`(d), `throw`(d, f), and `whistle`. During the defense time, the players can move to catch the ball and/or to re-position themselves on the court. When a player reaches the ball (s)he will have the ball and will throw the ball again. A team scores a point either if it throws the ball to a cell in the opposite subfield that is not reached by any player of the other team in the defense time, or if the opposite team throws the ball in the net. The captain (first player) of each team is in charge of checking if a point has been scored. In this case, (s)he `whistle`s.

Each team is modeled as a centralized multi-agent system, which acts as a singe agent in the interaction with the other team. Alternative options in modeling are also possible—for instance, one could model each single player as an independent agent that develops its own plan and interacts with all other players. The two teams have the goal of scoring a point: `goal(point(black) eq 1).` for blacks and `goal(point(white) eq 1).` for whites.

At the beginning of the execution every team has a winning strategy, developed as a local plan; these are possibly revised after each play to accommodate for the new state of the worlds reached. An execution (as printed by the system) is reported in Fig. 1, for a plan length of 9. The symbol `O` (respectively, `Y`) denotes the white (respectively, black) players, `Q` (resp. `X`) denotes a white player with the ball. The throw moves applied are:

`[player(black,1)]:throw(ne,3)`	(time 1)	`[player(black,2)]:throw(se,3)`	(time 3)
`[player(white,1)]:throw(w,5)`	(time 5)	`[player(black,1)]:throw(e,5)`	(time 7)

Let us observe that, although it would be in principle possible for the white team to reach the ball and throw it within the time allowed, it would be impossible to score a point. Therefore, players prefer to avoid to perform any move.

The complete description of the encoding of this domain is available at `http://www.dimi.uniud.it/dovier/BAAC`. The repository includes also additional domains—e.g., a domain inspired by games involving one ball and two-goals, as found in soccer. Although the encoding might seem similar to that of volleyball, the possibility of contact between two players makes this encoding more complex. Indeed, thanks to the fact that the net separates the two teams,

```
Time 0:            Time 1:            Time 2:            Time 3:            Time 4:
******|******     ******|******     ******|******     ******|******     ******|******
*     |     *     *     |     *     *     |     *     *     |     *     *     |     *
*  Y  |    O*     *  Yo |    O*     *   X |   O *     *   Y |   O *     *  Y  |  O  *
*     |     *     *     |     *     *     |     *     *     |     *     *     |     *
*     |  O  *     *     |  O  *     * Y   | O   *     * Y   | O   *     *Y    |     *
*X    |     *     *Y    |     *     *     |     *     *     |o    *     *     |Q    *
******|******     ******|******     ******|******     ******|******     ******|******
 Time 5:            Time 6:            Time 7:            Time 8:            Time 9:
******|******     ******|******     ******|******     ******|******     ******|******
*     |     *     *     |     *     *     |     *     *     |     *     *     |     *
*  Y  |  O  *     * Y   | O   *     * Y   | O   *     *Y    | O   *     *Y    | O   *
*     |     *     *     |     *     *     |     *     *     |     *     *     |     *
*Y    |     *     *     | O   *     *     | O   *     *     | O   *     *     | O   *
* o   |O    *     * X   |     *     * Y   |o    *     *Y    |o    *     *Y    |o    *
******|******     ******|******     ******|******     ******|******     ******|******
```

Figure 1 A representation of an execution of the volleyball domain

in the volleyball domain rules like the following one suffice to avoid collisions:

```
always(pair(x(A),y(A)) neq pair(x(B),y(B))) :-
      A=player(black,N),B=player(black,M), num(N), num(M), N<M.
```

In a soccer world this is not true because only the supervisor can be aware, in advance, of possible contacts between different team players originating from concurrent actions. This generates interesting concurrency problems, e.g., concerning the ball possession after a contact. A simple way to address this problem consists in assigning a fluent to each field cell, whose value can be -1 (free), 0 (resp., 1) if a white (resp. black) player is in the cell. The supervisor identifies a conflict when two opponent players move to the same cell, thus assigning to that fluent a different value. In this case, the supervisor arbitrarily enables one action, the other agent waits a turn to retry the action:

```
action act([A],move(D)) on_failure retry_after 1 on_conflict arbitrate :-
      agent(A), direction(D).
```

5 Conclusions and future work

In this paper, we designed a system for reasoning with action description languages in multi-agent domains. The language enables the description of agents with individual goals operating in shared environments. The agents can interact (by requesting help from other agents in achieving goals) and implicitly cooperate in resolving conflicts that arise during execution of their plans. The implementation is distributed, and uses Linda to enable communication.

The work is preliminary but shows strong potential and several directions of research. The immediate goal is to refine strategies and coordination mechanisms, involving, for instance, payoff, trust, etc. We intend to evaluate the performance and quality of the system in several multi-agent domains (e.g., game playing scenarios, auctions, and other domains requiring distributed planning). We will investigate the use of future references in the fluent constraints (as supported in $\mathcal{B}^{MV}$)—we believe this feature may provide a more elegant approach to handle the requests among agents, and it is necessary to enable the expression of complex interactions among agents (e.g., to model commitments). In particular, we view this platform as ideal to experiment with models of *negotiation* (e.g., as discussed in [14]) and to deal with *commitments* [11]. We will also explore the implementation of different strategies associated to conflict resolution; we are interested in investigating how to capture the notion of "trust" among agents, as a dynamic property that changes depending on how reliable agents have been in providing services to other agents.

References

1. R. H. Bordini, J. F. Hübner and M. Wooldridge. *Programming Multi-agent Systems in Agent-Speak using Jason*. J. Wiley and Sons, 2007.
2. N. Carriero and D. Gelernter. Linda in context. *CACM*, 32(4):444–458, 1989.
3. M. Dastani, F. Dignum, and J.-J. Meyer. 3APL: A programming language for cognitive agents. *ERCIM News*, 53:28–29, 2003.
4. G. De Giacomo, Y. Lespérance, and H. J. Levesque ConGolog, a concurrent programming language based on the situation calculus. *Artificial Intelligence*, 121(1–2):109–169, 2000.
5. A. Dovier, A. Formisano, and E. Pontelli. Representing multi-agent planning in CLP. *LPNMR*, LNCS 5753, pp. 423–429, Springer, 2009.
6. A. Dovier, A. Formisano, and E. Pontelli. Multivalued action languages with constraints in CLP(FD). *TPLP*, 10(2):167–235, 2010.
7. R. Fagin, J. Halpern, Y. Moses, and M. Vardi. *Reasoning about knowledge*. MIT Press, 1995.
8. M. Gelfond and V. Lifschitz. Action languages. *ETAI*, 2:193–210, 1998.
9. J. Gerbrandy. Logics of propositional control. In [12], pages 193–200.
10. K. V. Hindriks and T. Roberti. GOAL as a planning formalism. *MATES*, pp. 29–40. 2009.
11. A. U. Mallya and M. N. Huhns. Commitments among agents. *IEEE Internet Computing*, 7(4):90–93, 2003.
12. H. Nakashima, M. Wellman, G. Weiss and P. Stone editors. *Proc. of AAMAS*. ACM, 2006.
13. L. Sauro, J. Gerbrandy, W. van der Hoek, and M. Wooldridge. Reasoning about action and cooperation. In [12], pages 185–192.
14. T. Son, E. Pontelli, and C. Sakama. Logic programming for multiagent planning with negotiation. *ICLP*, pp. 99–114. Springer, 2009.
15. M. T. J. Spaan, G.J. Gordon, and N.A. Vlassis. Decentralized planning under uncertainty for teams of communicating agents. In [12], pages 249–256.
16. V. S. Subrahmanian, P. Bonatti, J. Dix. T. Eiter, S. Kraus and F. Ozcan, and R. Ross *Heterogeneous Agent Systems: Theory and Implementation*. MIT Press, 2000.
17. W. van der Hoek et al. A logic for strategic reasoning. *AAMAS*, pp. 157–164. ACM, 2005.

Declarative Processing of Semistructured Web Data

Michael Hanus

Institut für Informatik, CAU Kiel, D-24098 Kiel, Germany
mh@informatik.uni-kiel.de

Abstract

In order to give application programs access to data stored in the web in semistructured formats, in particular, in XML format, we propose a domain-specific language (DSL) for declarative processing such data. Our language is embedded in the functional logic programming language Curry and offers powerful matching constructs that enable a declarative description of accessing and transforming XML data. We exploit advanced features of functional logic programming to provide a high-level and maintainable implementation of our language. Actually, this paper contains the complete code of our implementation so that the source text of this paper is an executable implementation of our embedded DSL.

1998 ACM Subject Classification D.1.6 Logic Programming

Keywords and phrases functional logic programming, domain specific languages, XML

Digital Object Identifier 10.4230/LIPIcs.ICLP.2011.198

1 Motivation

Nowadays, huge amounts of information are available in the world-wide web. Much of this information is also available in semistructured formats so that it can be automatically accessed by application programs. The extensible markup language (XML) is often used as an exchange format for such data. Since data in XML format are basically term structures, XML data can be (in principle) easily processed with functional or logic programming languages: one has to define a term representation of XML data in the programming language, implement a parser from the textual XML representation into such terms, and exploit pattern matching to implement the specific processing task.

In practice, such an implementation causes some difficulties due to the fact that the concrete data formats are complex or evolve over time:

- For many application areas, concrete XML languages are defined. However, they are often quite complex so that it is difficult or tedious to deal with all details when one is interested in extracting only some parts of the given data.
- For more specialized areas without standardized XML languages, the XML format might be incompletely specified or evolves over time. Thus, application programs with standard pattern matching must be adapted if the data format changes.

For instance, consider the XML document shown in Fig. 1 which represents the data of a small address book. As one can see, the two entries have different information fields: the first entry contains two email addresses but no nickname whereas the second entry contains no email address but a nickname. Such data, which is not uncommon in practice, is also called "semistructured" [1]. Semistructured data causes difficulties when it should be processed with a declarative programming language by mapping the XML structures into data terms of the implementation language. Therefore, various distinguished languages for processing XML data have been proposed.

© Michael Hanus;
licensed under Creative Commons License NC-ND

Technical Communications of the 27th International Conference on Logic Programming (ICLP'11).
Editors: John P. Gallagher, Michael Gelfond; pp. 198–208

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

```
<contacts>
  <entry>
    <name>Hanus</name>
    <first>Michael</first>
    <phone>+49-431-8807271</phone>
    <email>mh@informatik.uni-kiel.de</email>
    <email>hanus@acm.org</email>
  </entry>
  <entry>
    <name>Smith</name>
    <first>William</first>
    <nickname>Bill</nickname>
    <phone>+1-987-742-9388</phone>
  </entry>
</contacts>
```

Figure 1 A simple XML document

For instance, the language XPath[1] provides powerful path expressions to select sub-documents in XML documents. Although path expressions allow flexible retrievals by the use of wildcards, regular path expressions, stepping to father and sibling nodes etc, they are oriented towards following a path through the document from the root to the selected sub-documents. This gives them a more imperative rather than a descriptive or declarative flavor. The same is true for query and transformation languages like XQuery[2] or XSLT[3] which are based on the XPath-oriented style to select the required sub-documents.

As an alternative to path-oriented processing languages, the language Xcerpt [5] is a proposal to exploit ideas from logic programming in order to provide a declarative method to select and transform semistructured data in XML format. In contrast to pure logic programming, Xcerpt proposes matching with partial term structures for which a specialized unification procedure, called "simulation unification" [6], has been developed. Since matching with partial term structures is a powerful feature that avoids many problems related to the evolution of web data over time, we propose a language with similar features. However, our language is an embedded domain-specific language (eDSL). Due to the embedding into the functional logic programming language Curry [12], our language for XML processing has the following features and advantages:

- The selection and transformation of incompletely specified XML data is supported.
- Due to the embedding into a universal programming language, the selected or transformed data can be directly used in the application program.
- Due to the use of advanced functional logic programming features, the implementation is straightforward and can be easily extended with new features. Actually, this paper contains the complete source code of the implementation.
- The direct implementation in a declarative language results in immediate correctness proofs of the implementation.

[1] http://www.w3.org/TR/xpath
[2] http://www.w3.org/XML/Query/
[3] http://www.w3.org/TR/xslt

In the following, we present our language for XML processing together with their implementation. Due to lack of space, we have to omit some details about functional logic programming in Curry and further features and properties of our eDSL. Interested readers find these in a separate technical report [10].

2 Functional Logic Programming and Curry

Curry [12] is a declarative multi-paradigm language combining features from functional, logic, and concurrent programming (recent surveys are available in [4, 9]). The syntax of Curry is close to Haskell [14]. In addition, Curry allows free (logic) variables in conditions and right-hand sides of defining rules. In contrast to functional programming and similarly to logic programming, operations can be defined by overlapping rules so that they might yield more than one result on the same input. For instance, the *choice* operation is predefined by:

```
x ? _  =  x
_ ? y  =  y
```

Thus, the expression "`0 ? 1`" has two values: `0` and `1`. If expressions have more than one value, one wants to select intended values according to some constraints, typically in conditions of program rules. A *rule* has the form "$f\ t_1 \ldots t_n$ `|` c `=` e" where the (optional) condition c is a *constraint*, like an *equational constraint* e_1 `=:=` e_2 which is satisfied if both sides are reducible to unifiable values. For instance, the rule

```
last xs | (ys++[z]) =:= xs  =  z          where ys,z free
```

defines an operation to compute the last element `z` of a list `xs` based on the (infix) operation "`++`" which concatenates two lists (in contrast to Prolog, free variables like `ys` or `z` need to be declared explicitly to make their scopes clear).

In the following, we implement an eDSL for XML processing based on functional logic programming features. To make this implementation as simple as possible, we exploit two more recent features described in the following: functional patterns and set functions.

A *functional pattern* [2] is a pattern occurring in an argument of the left-hand side of a rule containing defined operations (and not only data constructors and variables). For instance,

```
last (xs++[e]) = e
```

is a rule with the functional pattern `(xs++[e])` stating that `last` is reducible to `e` provided that the argument can be matched against some value of `(xs++[e])` where `xs` and `e` are free variables. By instantiating `xs` to arbitrary lists, the value of `(xs++[e])` is any list having `e` as its last element. As we will see in this paper, functional patterns are a powerful feature to express arbitrary selections in term structures. More details about their semantics and a constructive implementation of functional patterns by a demand-driven unification procedure can be found in [2].

Set functions [3] allow the encapsulation of nondeterministic computations in non-strict functional logic languages. For each defined function f, f_S denotes the corresponding set function. f_S encapsulates only the nondeterminism caused by evaluating f except for the nondeterminism caused by evaluating the arguments to which f is applied. For instance, consider the operation `decOrInc` defined by

```
decOrInc x = (x-1) ? (x+1)
```

Then "`decOrInc`$_{\mathcal{S}}$ `3`" evaluates to (an abstract representation of) the set {`2`,`4`}, i.e., the nondeterminism caused by `decOrInc` is encapsulated into a set. However, "`decOrInc`$_{\mathcal{S}}$ `(2?5)`" evaluates to two different sets {`1`,`3`} and {`4`,`6`} due to its nondeterministic argument, i.e., the nondeterminism caused by the argument is not encapsulated.

This paper contains the complete source code of our implementation. Actually, the paper's source text is a literate program [13] that is directly executable. In a literate Curry program, all real program code starts with the special character "`>`". Curry code not starting with "`>`", e.g., the example code shown so far, is like a comment and not required to run the program. To give an example of executable code, we show the declaration of the module `XCuery` for XML processing in Curry developed in this paper:

```
> module XCuery where
> import XML
```

Thus, we import the system module `XML` which contains an XML parser and the definition of XML structures in Curry that are explained in the next section.

3 XML Documents

There are two basic methods to represent XML documents in a programming language: a type-based or a generic representation [16]. In a type-based representation, each tagged XML structure (like `contacts`, `entry`, `name` etc) is represented as a record structure of appropriate type according to the XML schema. The advantage of this approach is that schema-correct XML structures correspond to type-correct record structures. On the negative side, this representation depends on the given XML schema. Thus, it is hardly applicable if the schema is not completely known. Moreover, if the schema evolves, the data types representing the XML structure must be adapted.

Due to these reasons, we prefer a generic representation where any XML document is represented with one generic structure. Since any XML document is either a structure with a tag, attributes and embedded XML documents (also call *child nodes* of the document), or a text string, one can define the following datatype to represent XML documents:[4]

```
data XmlExp = XText String
            | XElem String [(String,String)] [XmlExp]
```

Since it could be tedious to write XML documents with these basic data constructors, one can define some useful abstractions for XML documents:

```
xtxt s = XText s

xml t xs = XElem t [] xs
```

Thus, we can specify the second `entry` structure of the XML document shown in Fig. 1 by:

```
xml "entry" [xml "name"     [xtxt "Smith"],
             xml "first"    [xtxt "William"],
             xml "nickname" [xtxt "Bill"],
             xml "phone"    [xtxt "+1-987-742-9388"]]
```

[4] For the sake of simplicity, we ignore other specific elements like comments.

These definitions together with operations to parse and pretty-print XML documents are contained in the system module `XML` of the PAKCS programming environment for Curry [11]. In principle, these definitions are sufficient for XML processing, i.e., to select and transform XML documents. For instance, one can extract the name and phone number of an `entry` structure consisting of a name, first name and phone number by the following operation:

```
getNamePhone
  (xml "entry" [xml "name"  [xtxt name],
                _,
                xml "phone" [xtxt phone]])  =  name++": "++phone
```

Note that we use the abstractions `xml` and `xtxt` as functional patterns to provide a readable notation for matching XML documents. Nevertheless, XML processing operations as defined above have several disadvantages:

- The exact structure of the XML document must be known in advance. For instance, the operation `getNamePhone` matches only entries with three components, i.e., it fails on both entries shown in Fig. 1.
- In large XML documents, many parts are often irrelevant if one wants to select only some specific information entities. However, one has to define an operation to match the complete document.
- If the structure of the XML document changes (e.g., due to the evolution of the web services providing these documents), one has to update all patterns in the matching operations which could be tedious and error prone for large documents.

As a solution to these problems, we propose in the next section appropriate abstractions that can be used in patterns of operations for XML processing.

4 Abstractions for XML Processing

In order to define reasonable abstractions for XML processing, we start with a wish list. Since we have seen that exact matchings are not desirable to process semistructured data, we want to develop a language supporting the following features for pattern matching:

- *Partial patterns:* allow patterns where only some child nodes are known.
- *Unordered patterns:* allow patterns where child nodes can appear in any order.
- *Patterns at arbitrary depth:* allow patterns that are matched at an arbitrary position in an XML document.
- *Negation of patterns:* allow patterns defined by the absence of tags or provide default values for tags that are not present in the given XML document.
- *Transformation:* generate new structures from matched patterns.
- *Collect matchings:* accumulate results in a newly generated structure.

In the following, we show how these features can be supported by the use of carefully defined abstractions as functional patterns and other features of functional logic programming.

4.1 Partial Patterns

As we have seen in the example operation `getNamePhone` above, one would like to select some child nodes in a document independent of the availability of further components. Thus, instead of enumerating the list of *all* child nodes as in the definition above, it would be preferable to enumerate only the relevant child nodes. We support this by putting the operator “`with`” in front of the list of child nodes:

```
getNamePhone
  (xml "entry" (with [xml "name"  [xtxt name],
                      xml "phone" [xtxt phone]]))  =  name++": "++phone
```

The intended meaning of "`with`" is that the given child nodes must be present but in between any number of other elements can also occur.

We can directly implement this operator as follows:[5]

```
> with :: [a]  →  [a]
> with []     = _
> with (x:xs) = _ ++ x : with xs
```

Thus, an expression like "`with [1,2]`" reduces to any list of the form

$$x_1\texttt{:}\ldots\texttt{:}x_m\texttt{:1:}y_1\texttt{:}\ldots\texttt{:}y_n\texttt{:2:}zs$$

where the variables x_i, y_j, zs are fresh logic variables. Due to the semantics of functional patterns, the definition of `getNamePhone` above matches any `entry` structure containing a `name` and a `phone` element as children. Hence, the use of the operation `with` in patterns avoids the exact enumeration of all children and makes the program robust against the addition of further information elements in a structure.

A disadvantage of a definition like `getNamePhone` above is the fact that it matches only XML structures with an empty attribute list due to the definition of the operation `xml`. In order to support more flexible matchings that are independent of the given attributes (which are ignored if present), we define the operation

```
> xml' :: String  →  [XmlExp]  →  XmlExp
> xml' t xs = XElem t _ xs
```

For instance, the operation `getName` defined by

```
getName (xml' "entry" (with [xml' "name" [xtxt n]])) = n
```

returns the name of an `entry` structure independent of the fact whether the given document contains attributes in the `entry` or `name` structures.

4.2 Unordered Patterns

If the structure of data evolves over time, it might happen that the order of elements changes over time. Moreover, even in some given XML schema, the order of relevant elements can vary. In order to make the matching independent of a particular order, we can specify that the required child nodes can appear in any order by putting the operator "`anyorder`" in front of the list of child nodes:

```
getNamePhone
  (xml "entry"
     (with (anyorder [xml "phone" [xtxt phone],
                      xml "name"  [xtxt name]])))  =  name++": "++phone
```

[5] The symbol "`_`" denotes an anonymous variable, i.e., each occurrence of "`_`" in the right-hand side of a rule denotes a fresh logic variable.

Obviously, the operation `anyorder` should compute any permutation of its argument list. In a functional logic language, it can be easily defined as a nondeterministic operation by inserting the first element of a list at an arbitrary position in the permutation of the remaining elements:

```
> anyorder :: [a]  →  [a]
> anyorder []     = []
> anyorder (x:xs) = insert (anyorder xs)
>  where insert []     = [x]
>        insert (y:ys) = x:y:ys  ?  y : insert ys
```

Thus, the previous definition of `getNamePhone` matches both `entry` structures shown in Fig. 1.

4.3 Patterns at Arbitrary Depths

If one wants to select some information in deeply nested documents, it would be tedious to define the exact matching from the root to the required elements. Instead, it is preferable to allow matchings at an arbitrary depth in a document. Such matchings are also supported in other languages like XPath since they ease the implementation of queries in complex structures and support flexibility of the implementation w.r.t. to future structural changes of the given documents. We support this feature by an operation "`deepXml`": if `deepXml` is used instead of `xml` in a pattern, this structure can occur at an arbitrary position in the given document. For instance, if we define

```
getNamePhone
  (deepXml "entry"
     (with [xml "name"  [xtxt name],
            xml "phone" [xtxt phone]]))  =  name++": "++phone
```

and apply `getNamePhone` to the complete document shown in Fig. 1, two results are (nondeterministically) computed (methods to collect all those results are discussed later).

The implementation of `deepXml` is similar to `with` by specifying that `deepXml` reduces to a structure where the node is at the root or at some nested child node:

```
> deepXml :: String  →  [XmlExp]  →  XmlExp
> deepXml tag elems = xml tag elems
> deepXml tag elems = xml' _ (_ ++ [deepXml tag elems] ++ _)
```

Thus, an expression like "`deepXml` *t cs*" reduces to "`xml` *t cs*" or to a structure containing this element at some inner position.

4.4 Negation of Patterns

As mentioned above, in semistructured data some information might not be present in a given structure, like the email address in the second entry of Fig. 1. Instead of failing on missing information pieces, one wants to have a constructive behavior in application programs. For instance, one could select all entries with a missing email address or one puts a default nickname in the output if the nickname is missing.

In order to implement such behaviors, one could try to negate matchings. Since negation is a non-trivial subject in functional logic programming, we propose a much simpler but practically reasonable solution. We provide an operation "`withOthers`" which is similar to

"`with`" but has a second argument that contains the child nodes that are present but not part of the first argument. Thus, one can use this operation to denote the "unmatched" part of a document in order to put arbitrary conditions on it. For instance, if we want to get the name and phone number of an entry that has no email address, we can specify this as follows:

```
getNamePhoneWithoutEmail
 (deepXml "entry"
   (withOthers [xml "name" [xtxt name], xml "phone" [xtxt phone]] others))
 | "email" `noTagOf` others  =  name++": "++phone
```

The useful predicate `noTagOf` returns true if the given tag is not a tag of all argument documents (the operation `tagOf` returns the tag of an XML document):

```
> noTagOf :: String  → [XmlExp]  → Bool
> noTagOf tag = all ((/=tag) . tagOf)
```

Hence, the application of `getNamePhoneWithoutEmail` to the document in Fig. 1 returns a single value.

The implementation of `withOthers` is slightly different from `with` since we have to accumulate the remaining elements that are not part of the first arguments in the second argument:

```
> withOthers :: [a]  → [a]  → [a]
> withOthers ys zs = withAcc [] ys zs
>  where -- Accumulate remaining elements:
>    withAcc prevs [] others | others=:=prevs++suffix = suffix
>                                               where suffix free
>    withAcc prevs (x:xs) others =
>       prefix ++ x : withAcc (prevs++prefix) xs others
>                                               where prefix free
```

Thus, an expression like "`withOthers [1,2]` *os*" reduces to any list of the form

$$x_1\texttt{:}\ldots\texttt{:}x_m\texttt{:1:}y_1\texttt{:}\ldots\texttt{:}y_n\texttt{:2:}zs$$

where $os = x_1\texttt{:}\ldots\texttt{:}x_m\texttt{:}y_1\texttt{:}\ldots\texttt{:}y_n\texttt{:}zs$. If we use this expression as a pattern, the semantics of functional patterns ensures that this pattern matches any list containing the elements `1` and `2` where the variable *os* is bound to the list of the remaining elements.

4.5 Transformation of Documents

Apart from the inclusion of data selected in XML documents in the application program, one also wants to implement transformations on documents. Such transformation tasks are almost trivial to implement in declarative languages supporting pattern matching by using a scheme like "*transform pattern* = *newdoc*" and applying the *transform* operation to the given document. For instance, we can transform an `entry` document into another XML structure containing the phone number and full name of the person by

```
 transPhone (deepXml "entry" (with [xml "name"  [xtxt n],
                                    xml "first" [xtxt f],
                                    xml "phone" phone])) =
    xml "phonename" [xml "phone" phone, xml "fullname" [xtxt (f++' ':n)]]
```

If we apply `transPhone` to the document of Fig. 1, we nondeterministically obtain two new XML documents corresponding to the two entries contained in this document.

4.6 Collect Matchings

In order to collect all matchings in a given document in a single new document, we have to encapsulate the nondeterministic computations performed on the input document. For this purpose, we can exploit set functions described above. Since set functions return an unordered set of values, we have to transform this value set into an ordered list structure that can be printed or embedded in another document. This can be done by the predefined operation `sortValues`. Thus, if c denotes the XML document shown in Fig. 1, we can use our previous transformation operation to create a complete table of all pairs of phone numbers and full names by evaluating

```
xml "table" (sortValues (transPhone𝒮 c))
```

Similarly, one can also transform XML documents into HTML documents by exploiting the HTML library of Curry [8]. Furthermore, one can also nest set functions to accumulate intermediate information. As an example, we want to compute a list of all persons together with the number of their email addresses. For this purpose, we define a matching rule for an `entry` document that returns the number of email addresses in this document by a set function `emailOf`$_{\mathcal{S}}$:

```
getEmails (deepXml "entry" (withOthers [xml "name" [xtxt name]] os))
   = (name, length (sortValues (emailOf𝒮 os)))
 where emailOf (with [xml "email" email]) = email
```

In order to compute a complete list of all entries matched in a document c, we apply the set function `getEmails`$_{\mathcal{S}}$ to collect all results in a list structure:

```
sortValues (getEmails𝒮 c)
```

For our example document, this evaluates to `[("Hanus",2),("Smith",0)]`.

5 Related Work

Since the processing of semistructured data is a relevant issue in current application systems, there are many proposals for specialized languages or embedding languages in multi-purpose programming languages. We discuss some related approaches in this section.

We have already mentioned in the beginning the languages XPath, XQuery, and XSLT for XML processing supported by the W3C. These languages provide a different XML-oriented syntax and use a navigational approach to select information rather than the pattern-oriented approach we proposed. Since these are separate languages, it is more difficult to use them in application programs written in a general purpose language where one wants to process data available in the web.

The same is true for the language Xcerpt [5]. It is also a separate XML processing language without a close connection to a multi-purpose programming language. In contrast to XPath, Xcerpt proposes the use of powerful matching constructs to select information in semistructured documents. Xcerpt supports similar features as our embedded language but provide a more compact syntax due to its independence of a concrete base language. In contrast to our approach, Xcerpt requires a dedicated implementation based on a specialized

unification procedure [6]. The disadvantages of such separate developments become obvious if one tries to access the implementation of Xcerpt (which failed at the time of this writing due to inaccessible web pages and incompatible compiler versions).

HaXML [16] is a language for XML processing embedded in the functional language Haskell. It provides a rich set of combinators based on *content filters*, i.e., functions that map XML data into collections of XML data. This allows an elegant description of many XML transformations, whereas our rule-based approach is not limited to such transformations since we have no restrictions on the type of data constructed from successful matchings.

Caballero et al. [7] proposed the embedding of XPath into the functional logic language Toy that has many similarities to Curry. Similarly to our approach, they also exploit nondeterministic evaluation for path selection. Due to the use of a functional logic language allowing inverse computations, they also support the generation of test cases for path expressions, i.e., the generation of documents to which a path expression can be applied. Nevertheless, their approach is limited to the navigational processing of XPath rather than a rule-based approach as in our case. The same holds for FnQuery [15], a domain-specific language embedded in Prolog for the querying and transformation of XML data.

6 Conclusions

We have presented a rule-based language for processing semistructured data that is implemented and embedded in the functional logic language Curry. The language supports a declarative description to query and transform such data. It is based on providing operations to describe partial matchings in the data and exploits functional patterns and set functions for the programming tasks. Due to its embedding into a general-purpose programming language, it can be used to further process the selected data in application systems or one can combine semistructured data from different sources. Moreover, it is easy to extend our language with new features without adapting a complex implementation.

The simplicity of our implementation together with the expressiveness of our language demonstrate the general advantages of high-level declarative programming languages. In order to check the usability of our language, we applied it to extract information provided by our university information system[6] in XML format into a curricula and module information system[7] that is implemented in Curry. In this application it was quite useful to specify only partial patterns so that most of the huge amount of information contained in the XML document could be ignored.

Although our implementation heavily exploits nondeterministic computations, e.g., for matching in partially specified or deep structures, our initial experiments show that it is practically useful. The processing time in these tests to select or transform documents is almost equal or smaller than the time to parse the document by an already given (deterministic!) XML parser.

For future work, we intend to apply our language to more examples in order to enrich the set of useful pattern combinators. Moreover, it would be interesting to generate more efficient implementations by specializing functional patterns (e.g., by partial evaluation w.r.t. the given definitions, or by exploiting the XML schema if it is precisely known in advance).

[6] `http://univis.uni-kiel.de/`
[7] `http://www-ps.informatik.uni-kiel.de/~mh/studiengaenge/`

References

1 S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web: From Relations to Semistructured Data and XML*. Morgan Kaufmann, 2000.

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 S. Antoy and M. Hanus. Set functions for functional logic programming. In *Proceedings of the 11th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'09)*, pages 73–82. ACM Press, 2009.

4 S. Antoy and M. Hanus. Functional logic programming. *Communications of the ACM*, 53(4):74–85, 2010.

5 F. Bry and S. Schaffert. A gentle introduction to Xcerpt, a rule-based query and transformation language for XML. In *Proceedings of the International Workshop on Rule Markup Languages for Business Rules on the Semantic Web (RuleML'02)*, 2002.

6 F. Bry and S. Schaffert. Towards a declarative query and transformation language for XML and semistructured data: Simulation unification. In *Proceedings of the International Conference on Logic Programming (ICLP'02)*, pages 255–270. Springer LNCS 2401, 2002.

7 R. Caballero, Y. García-Ruiz, and F. Sáenz-Pérez. Integrating XPath with the functional-logic language Toy. Technical report sic-05-10, Univ. Complutense de Madrid, 2010.

8 M. Hanus. High-level server side web scripting in Curry. In *Proc. of the Third International Symposium on Practical Aspects of Declarative Languages (PADL'01)*, pages 76–92. Springer LNCS 1990, 2001.

9 M. Hanus. Multi-paradigm declarative languages. In *Proceedings of the International Conference on Logic Programming (ICLP 2007)*, pages 45–75. Springer LNCS 4670, 2007.

10 M. Hanus. Declarative processing of semistructured web data. Technical report 1103, Christian-Albrechts-Universität Kiel, 2011.

11 M. Hanus, S. Antoy, B. Braßel, M. Engelke, K. Höppner, J. Koj, P. Niederau, R. Sadre, and F. Steiner. PAKCS: The Portland Aachen Kiel Curry System. Available at `http://www.informatik.uni-kiel.de/~pakcs/`, 2010.

12 M. Hanus (ed.). Curry: An integrated functional logic language (vers. 0.8.2). Available at `http://www.curry-language.org`, 2006.

13 D.E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, 1984.

14 S. Peyton Jones, editor. *Haskell 98 Language and Libraries—The Revised Report.* Cambridge University Press, 2003.

15 D. Seipel, J. Baumeister, and M. Hopfner. Declaratively querying and visualizing knowledge bases in XML. In *Applications of Declarative Programming and Knowledge Management (INAP/WLP 2004)*, pages 16–31. Springer LNCS 3392, 2005.

16 M. Wallace and C. Runciman. Haskell and XML: Generic combinators or type-based translation? In *Proc. of the ACM SIGPLAN International Conference on Functional Programming (ICFP'99)*, pages 148–159. ACM Press, 1999.

CDAOStore: A Phylogenetic Repository Using Logic Programming and Web Services*

Brandon Chisham[1], Enrico Pontelli[1], Tran Cao Son[1], and Ben Wright[1]

1 **Dept. Computer Science, New Mexico State University**
{bchisham | epontell | tson | bwright}@cs.nmsu.edu

Abstract

The *CDAOStore* is a portal aimed at facilitating the storage and retrieval of data and metadata associated to studies in the field of evolutionary biology and phylogenetic analysis. The novelty of CDAOStore lies in the use of a semantic-based approach to the storage and querying of data. This enables CDAOStore to overcome the data format restrictions and complexities of other repositories (e.g., TreeBase) and to provide a domain-specific query interface, derived from studies of querying requirements for phylogenetic databases.

CDAOStore represents the first full implementation of the *EvoIO stack*, an inter-operation stack composed of a formal ontology (the Comparative Data Analysis Ontology), an XML exchange format (NeXML), and a web services API (PhyloWS). CDAOStore has been implemented on top of an RDF triple store, using a combination of standard web technologies and logic programming technology. In particular, we employed Prolog to support some of the format transformation tasks and, more importantly, in the implementation of several of the domain-specific queries, whose structure is beyond the reach of standard RDF query languages (e.g., SPARQL). CDAOStore is operational and it already hosts over 90 million RDF triples, imported from TreeBase or submitted by other domain scientists.

1998 ACM Subject Classification J.3 Life and Medical Sciences

Keywords and phrases Bioinformatics, Phylogenetic Analysis, Prolog, Ontologies

Digital Object Identifier 10.4230/LIPIcs.ICLP.2011.209

1 Introduction

The explosive growth of stock-piled information in the biological and earth sciences presents a wealth of opportunities for expanding bioinformatics-based analyses with respect to both the amount of data incorporated and the diversity of data types and sources to be integrated. While large-scale or integrative analyses of such data may use generic methods of machine learning, there is a theory-based comparative approach to the analysis of diverse types of biological data, in which the similarities and differences between compared things are interpreted as *evolved differences* that have arisen by a process of descent-with-modification from common ancestors. This evolutionary comparative approach, used throughout biology and paleobiology, depends fundamentally on "phylogenetic trees" representing paths of descent. While powerful tools exist for the inference of phylogenies, and while evolutionary approaches are increasingly recognized as effective, the lack of interoperability in tree-based data and services hinders large-scale and integrative analyses.

* The research has been partially supported by NSF grants DGE-0947465, IIS-0812267, CBET-0754525 and HRD-0420407.

© B. Chisham, E. Pontelli, T. Son, B. Wright;
licensed under Creative Commons License NC-ND
Technical Communications of the 27th International Conference on Logic Programming (ICLP'11).
Editors: John P. Gallagher, Michael Gelfond; pp. 209–219

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

To address this overarching problem, a collaboration among computational scientists and evolutionary biologists has been established—within a working group (EvoInfoWG) sponsored by the National Evolutionary Synthesis Center—leading to the development of an *interoperation stack* (*EvoIO Stack*) for the exchange of evolutionary structures. The EvoIO Stack comprises of *(i)* an ontology for the description of data (the *Comparative Data Analysis Ontology*), *(ii)* an exchange format (*NeXML*), and *(iii)* a web service interface (*PhyloWS*).

The use of components of the EvoIO Stack has been gaining momentum—e.g., NeXML is now a supported format by several analysis tools (e.g., Mesquite [11], DAMBE [22]), CDAO and PhyloWS are supported by TreeBase [1]), the largest repository of phylogenetic trees. While these efforts adopt components of the EvoIO Stack as adds-on to the existing features of the tools and repositories, the EvoInfoWG has initiated the development of a novel data repository completely built around the EvoIO Stack and capable of providing forms of access to evolutionary data that are beyond the relational access forms offered by traditional repositories, like TreeBase. The CDAOStore is the first effort in this direction.

CDAOStore is a triple store that implements the complete EvoIO Stack. As a triple store, it maintains a semantic-based repository for phylogenetic data, as RDF triples; it provides the ability to import and export the content in NeXML and other commonly used formats, and it supports querying through a PhyloWS web service interface. CDAOStore offers a domain-specific querying interface supporting classes of queries relevant to phylogenetic investigation, as identified by domain experts. The implementation of the CDAOStore combines established web services and ontology technologies with Prolog; logic programming is employed to provide an effective implementation of several classes of domain-specific queries, which are beyond the reach of traditional ontology-based query languages (e.g., SPARQL [16]). In particular, Prolog enables the elegant encoding of operations that manipulate collection of phylogenetic trees, performing selection of branches and transitive closures.

2 Background

Phylogenetics and Interoperation: Phylogenetic trees (a.k.a. phylogenies) have gained a central role in modern biology. Trees provide a systematic structure to organize evolutionary knowledge about diversity of life. Trees have become fundamental tools for building new knowledge, thanks to their explanatory and comparative-based predictive capabilities. Evolutionary relationships provide clues about processes underlying biodiversity and enable predictive inferences about future changes in biodiversity (e.g., in response to climate or anthropogenic changes). Phylogenies are used with increase frequency in several fields, e.g., comparative genomics [2], metagenomics [21], and community ecology [19]. The major obstacle hindering broad availability and repurposing of phylogenies has been, for a long time, the lack of effective standards and a community-driven process for adopting and extending them. Existing file formats allow for representation of trees using a simple string and also the molecular or morphological character data used to infer the tree. There are no widely accepted standards for annotating tips, internal nodes or branches, and different applications have adopted unique methods for modifying the tree strings, meaning that annotations from one program may generate errors or be misinterpreted when import into another program. Other types of data/metadata, such as descriptions of evolutionary models or metadata annotations for provenance, have not seen any attempts at standardization.

The EvoIO Stack: The *EvoIO Stack* [17] has been proposed as a platform that coherently combines support for exchange of data and their semantics and predictable programmatic access. The EvoIO Stack is seeded with a triplet of emerging interoperability standards—

NeXML, CDAO, and PhyloWS.

NeXML [18] is an exchange standard for phylogenetic trees, data matrices, and arbitrary metadata. NeXML is an XML schema for comparative biology that draws on the successful high-level block structure of NEXUS [10], but takes advantage of widespread support for XML, and harnesses the W3C-proposed RDFa standard to embed semantically rich metadata.

The *Comparative Data Analysis Ontology (CDAO)* [15] provides a formal ontology for describing phylogenies and their associated character state matrices. It provides a general framework for talking about the relationships between taxa, characters, states, their matrices, and associated phylogenies. The ontology is organized around four central concepts: operational taxonomic units (OTUs), characters, character states, phylogenetic trees, and transitions. A phylogenetic analysis starts with the identification of a collection of *Operational Taxonomic Units* (OTUs), representing the entities being described (e.g., species, genes). Each OTU is described, in the analysis, by a collection of properties, typically referred to as *characters*. The values that characters can assume are referred to as *character states*. In phylogenetic analysis, it is common to collect the characters and associated character states in a matrix, the *character state matrix*, where the rows correspond to the different OTUs and the columns correspond to the characters. Phylogenetic trees are used to represent paths of descent-with-modification, capturing the evolutionary process underlying the OTUs.

PhyloWS [8] is a web-services standard for searching, addressing, and accessing phylogenetic trees, data matrices, and their associated metadata in a predictable and programmable way from online phylogenetic data providers. All PhyloWS URI's begin with /phylows/ as the standard delimiter. Then based on the phylogenetic information being queried a data structure will be given, such as taxon, tree, or study. This is followed by any specific identifiers needed for the query. For example, `http://purl.org/phylo/treebase/phylows/tree/TB2:Tr3099?format=rdf` returns the tree with the TreeBase ID equal to 'Tr3099' in RDF format.

3 CDAOStore

The CDAOStore is a novel repository and portal aimed at facilitating the storage and retrieval of phylogenetic data; it is the first instance of a repository built on the EvoIO stack. The novelty of CDAOStore lies in the use of a semantic-based approach to the storage and querying of data, building on CDAO for the semantic annotation of data. This approach enables scientists to overcome the restrictions imposed by the use of specific data formats—thus, facilitating inter-operation among phylogenetic analysis applications. This is different, e.g., from the main existing repository for phylogenies, TreeBase, which requires submissions in NEXUS format; there is experimental evidence (e.g., [13]) that data reuse and inter-operation has been hard to achieve, and TreeBase has high rates of rejected submissions due to incorrect data formats (e.g., a study showed that, over a period of time, 9% of the submissions to TreeBase were complete, and 11% of them could not be parsed). Furthermore, the use of a semantic-based repository makes it possible to design and implement more meaningful domain-specific queries. The overall structure of CDAOStore is illustrated in Fig. 1. The system is organized in three modules. The *Importer* module enables the submission of new

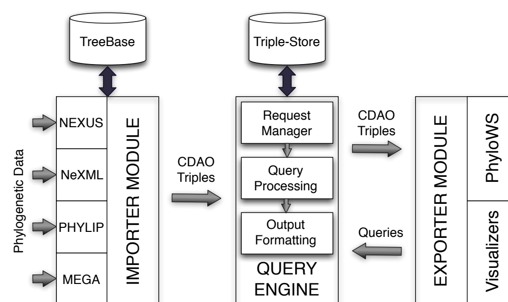

Figure 1 Overall structure of CDAOStore

phylogenies, data matrices, or metadata for existing phylogenies into the repository. The *Query* module supports all the queries to the repository. The *Exporter* module implements the user interface to the CDAOStore. In the following subsections, we review the functionalities and implementation of these three modules. The details of the *Query* modules are discussed in the following sections, as this module represents the core of the system.

Importer Module: The purpose of the Importer module is to import phylogenies and their associated data into the repository, automatically extracting their representations in terms of instances of CDAO. The data importer module can process phylogenetic data encoded in several commonly used data formats—currently, the Importer supports NEXUS [10], NeXML [18], PHYLIP [3], and MEGA [7] formats. The Importer has been realized by developing a semantic characterization of each data format in terms of CDAO concepts and relations. The Importer module includes a permanent link to TreeBase—i.e., the CDAOStore repository provides a semantic mirroring of the complete content of TreeBase, and new updates to TreeBase are immediately reflected into CDAOStore. The various parsing sub-modules have been developed from scratch, using combinations of C++, Prolog, and XSLT. In particular, Prolog has been used to implement a parser for NEXUS—accommodating for differences in the interpretations of the NEXUS specification [10] through non-determinism[6]. The Importer maps data from the input files to an object model that mirrors CDAO classes, producing RDF/XML triples that can be deposited in the CDAOStore triple store. The data importer module is also capable of mapping the object model back into any of the acceptable data formats; this enables the use of CDAOStore for conversion among data formats.

Exporter Module: The Exporter modules provides the user interfaces of CDAOStore. It consists of a Web portal, which allows users to submit and query data, and a series of visualization tools, referred to as CDAO Explorer. CDAO Explorer includes an application for visualizing phylogenetic trees, that provides different visualization formats and the ability to highlight or hide parts of the trees according to user-defined criteria, an application for displaying character data matrices, with capabilities to use color coding to highlight character patterns, and a tool to graphically add annotations to existing phylogenies. Additionally, the Exporter module includes the ability to map phylogenies encoded in the triple store to any of a number of formats (e.g., phyloXML, NeXML, Newick, RDF/XML), which can be piped to other visualization tools, such as Nexplorer 3 [4], which accepts CDAO RDF/XML input, and PhyloBox [5], which accepts phyloXML and Newick representations.

Query Module: At the core of CDAOStore we find a triple store which collects data and meta-data associated to phylogenetic analysis studies. All the data and metadata are stored as RDF triples, encoded using CDAO. The repository itself has been implemented using the RDFlib library (`www.rdflib.net`), a Python-based RDF store which uses a MySQL database to maintain the serialized RDF triples representing instances of CDAO. RDFlib was selected for its simplicity and for the flexibility offered in formatting the output of queries posed to the store. The Query module is articulated into three components—a request manager, a query processor, and an output formatter.

The request manager provides the infrastructure for the PhyloWS web service API and prepares the queries for execution on the triple store. The PhyloWS API is the basis for all the data access features of CDAOStore. The other web components and the CDAO Explorer use PhyloWS to access data. The URI's of the CDAOStore implementation of PhyloWS are divided into three conceptual parts: **(1)** the address of the store site and path prefix `www.cs.nmsu.edu/~cdaostore/cgi-bin/phylows`, **(2)** a query type (e.g., tree, matrix, msc, nca, size), and **(3)** a parameter list, dependent on the specific type of query. For example, the msc (maximum spanning clade) and nca (nearest common ancestor) query types expect a

list of taxon id's separated by '/'. The listing query takes optional limit and offset parameters to paginate results. The size query requires parameters describing a *direction* (greater, less, or equal), a *criteria* (node, internal, or leaf) and a *limit* (a numeral). The request manager handles parsing, sanitizing, interpreting request arguments, and orchestrating the remaining components. The results produced by the execution of a query are returned to the Exporter for output. For most queries, the formatting is largely determined by the format string given to the query processor, and a query-dependent header and footer. For the queries determining phylogenies, the output is filtered through a post-processing layer to reorder its components and make it easier to be processed by the exporter module—e.g., the components of the phylogeny are exported in an order that resembles a breadth-first traversal of the tree.

4 Domain-Specific Queries

4.1 Querying Biological Phylogenies

CDAOStore maps phylogenies and characters state data matrices to instances of CDAO, and stores them as RDF triples in a triple store. The triple store of CDAOStore is exposed to the users via the PhyloWS web service interface, enabling them to submit SPARQL queries [16]. These queries are processed by the SPARQL processor implemented by the RDFlib library. Nevertheless, the use of a SPARQL interface has several drawbacks; SPARQL is a relatively complex query language, which is relatively inaccessible to the average life scientist. Furthermore, there is evidence in the literature on uses of phylogenetic analysis in biological investigations that the type of queries required are often more complex than what standard SPARQL can provide. In particular, SPARQL is weak in handling queries to hierarchical structures of arbitrary depth and offers limited support for aggregate functions.

To address these issues, we developed a *domain-specific* query interface to the triple store, providing classes of queries that have been determined to be of general interest to life scientists using phylogenies in their investigations. The classes of queries have been devised through a combination of focus groups with life scientists and investigation of the relevant literature—the problem of storing and accessing phylogenies has been recognized for quite some time (e.g., [14]) and attacked by various research groups (e.g., the Evolutionary Database Interoperability group at the Natl. Evolutionary Synthesis Center). In particular, a seminal paper on desiderata for phylogenetic databases [12] provided a classification of phylogenetic investigations and the associated types of queries.

The study in [12] identifies six classes of uses of phylogenetic repositories: *(1)* Casual Uses (occasional search for a phylogeny, e.g., for informational use); *(2)* Visualization Uses (retrieval of one or more phylogeny for graphical display); *(3)* Study Development Uses (contribution of new phylogenies or update of existing ones); *(4)* Super-Tree Uses (for assembly of phylogenies into super-trees); *(5)* Simulation Studies (e.g., to assess performance of models of evolution); *(6)* Comparative Genomics Studies (use of phylogenies to relate different genes/genomes).

The investigation in [12] identified the needs of these six application leading to eleven basic types of phylogenetic queries. We revised these classes of queries through our own focus groups and refined them into the following classes of queries:

- **Q1** Determine all the phylogenies containing a given set of taxa—e.g., locate all trees containing the taxonomic units named `Ilex anomala` and `Ilex glabra`.
- **Q2** Determine the relations among a set of taxa in all phylogenies (query not supported).
- **Q3** Determine the minimum spanning tree/clade for a given set of taxa—e.g., locate the minimum spanning clade in the tree `Tree3099` (TreeBase identifier) for the taxonomic units `Ilex anomala` and `Ilex glabra`.

- **Q4** Determine all phylogenies constructed using a given inference method—e.g., locate all the phylogenies constructed using a parsimony method.
- **Q5** Determine all the phylogenies containing a set number of taxa—e.g., locate all the phylogenies with at most 25 taxa.
- **Q6** Determine all the phylogenies produced by a given tool or author—e.g., locate all the phylogenies published by W. Piel.
- **Q7** Determine all phylogenies satisfying a given geometric property—e.g., locate all the phylogenies that have diameter equal to 5.
- **Q8** Given a phylogeny P, a measure m, and a quantity q, determine all the phylogenies that are at distance q from P according to the measure m (e.g., for the purpose of clustering phylogenies that are "close" to a given tree).
- **Q9** Given a model of evolution, determine all the phylogenies that have been constructed using such model of evolution—e.g., identify all the phylogenies that have been constructed using Jukes-Cantor model for estimating distance.
- **Q10** Given a measure, return statistics about the measure in the phylogenies present in the repository—e.g., determine the distribution of tree lengths
- **Q11** Given a type of data and a set of taxa, determine all the phylogenies on the set of taxa that have been constructed using the specified type of data. sequences.

Two classes of queries are currently not supported—classes **Q2** and **Q8**. The class **Q2**, drawn from the study in [12], is only vaguely specified, and further investigation is in progress to characterize the type of relationships to be considered—the queries in this class are aimed at discovering opportunities for clustering of taxa based on how they are related by the phylogenies in the repository. Queries in class **Q8** are associated to the use of distance measures between phylogenetic trees (e.g., Robinson-Foulds distance, K tree score) to determine clusters of trees. For questions **Q7**, we support the computation of radii and diameters—defined in terms minimum and maximum eccentricity of the phylogeny, where the eccentricity is the maximum distance in the phylogeny among any two nodes.

4.2 Query Implementation

The various queries listed above have been implemented in CDAOStore using a combination of SPARQL and Prolog; some of the queries (e.g., **Q1**) can be mapped to corresponding SPARQL queries; other queries require more involved reasoning on phylogenies (e.g., computation of nearest common ancestors) and these are implemented by mapping phylogenies into Prolog terms and using Prolog to address the queries.

In order to use of Prolog in CDAOStore, we developed a pre-processor, which is used to generate a Prolog program that will produce the result of the submitted query. The first step of the pre-processor is to generate SPARQL queries to retrieve from the triple store the phylogenies of interest (i.e., those referred to by the query being executed) and convert the RDF representation of such phylogenies into a collection of ground Prolog facts. The pre-processor combines these ground facts with a set of pre-determined Prolog rules, which describe the computation of the query, and feeds the result to a Prolog engine for execution (in our case, SWI-Prolog). Finally, the pre-processor converts the result back into a RDF format for final output. The facts used to describe phylogenies include facts of the form:

```
tree(TreeName).        node(TreeName, NodeName).
edge(TreeName, EdgeName, Direction, SourceNode, DestinationNode).
```

The nodes are classified as internal, root, or leaf nodes by adding suitable simple rules. The set of rules added include those used to determine the ancestors of a node, in the form of a

predicate `ancestor_of(Tree, Ancestor, Node)`, as a simple transitive closure of the `edge` predicate. For queries that involve the entire repository (e.g., **Q10**), the process above is broken down into chunks, where the Prolog system is invoked not on all the phylogenies at once, but on groups of a given size (established as a system parameter); the results are incrementally aggregated to produce the final result.

Queries Q1: This class of queries is used to determine all the phylogenies containing a given set of taxa. This query can be implemented directly in SPARQL:

```
PREFIX study: <http://www.cs.nmsu.edu/~bchisham/study.owl#>
PREFIX contact: <http://www.w3.org/2000/10/swap/pim/contact#>
PREFIX foaf: <http://www.mindswap.org/2003/owl/foaf#>
SELECT ?tree WHERE { ?tree has_TU TU1.      . . .     ?tree has_TU TUN.  }
```

The various taxa passed as arguments refer to standardized names of taxa within the CDAOStore; work is in progress to adapt the internal nomenclature to satisfy global naming standards being developed by several working groups (e.g., the Darwin Core [20]).

Queries Q3: This class of queries is used to determine the minimum spanning clade for a given set of taxa or the nearest common ancestor of a given collection of taxa. These queries are implemented using the Prolog engine. The nearest common ancestor is an ancestor of all taxa in a given set such that none of its descendants is also an ancestor of all such taxa:

```
common_ancestor_of(Tree, Ancestor, [Node]) :- !, ancestor_of(Tree, Ancestor, Node).
common_ancestor_of(Tree, Ancestor, [Node | Nodes]) :-
     ancestor_of(Tree, Ancestor, Node), common_ancestor_of(Tree, Ancestor, Nodes).
distant_common_ancestor_of(Tree,DistantAncestor,Nodes) :-
     common_ancestor_of(Tree,Anc,Nodes),  ancestor_of(Tree,DistantAncestor,Anc).
nearest_common_ancestor_of(Tree,Nca,Nodes) :-  common_ancestor_of(Tree,Nca,Nodes),
     not(distant_common_ancestor_of(Tree, Nca, Nodes)).
```

The minimum spanning clade of a set of taxa is the set of nodes that includes the nearest-common ancestor of all the taxa in the set, and all of its descendants.

```
clade( Tree, Node, Member ):- ancestor_of( Tree, Node, Member ).
clade( Tree, Node, Node):- node( Tree, Node ).
msclade( Tree, Nodes, Clade ):-  nearest_common_ancestor_of(Tree,NCA,Nodes),
   setof(Member, clade( Tree, NCA, Member), Clade).
```

Queries Q4: This class of queries is used to determine all the phylogenies constructed using a given inference method. There are two different “methods” information that can be checked (as from the CDAO specification)—i.e., the algorithm used and the specific phylogenetic inference system (i.e., the specific software used). Both cases can be handled in SPARQL:

```
SELECT ?tree WHERE {                          SELECT ?tree WHERE {
 ?study study:has_analysis ?analysis.          ?study study:has_analysis ?analysis .
 ?analysis study:has_algorithm '$ALGO' .       ?analysis study:has_software '$ALGO' .
 ?analysis study:has_output_tree ?tree . } ?analysis study:has_output_tree ?tree . }
```

where *$ALGO* is the variable for the given algorithm/software being checked for.

Queries Q5: This queries are used to determine all phylogenies containing a given number of taxa. This query requires the use of Prolog, to count the number of leaves in the phylogenies:

```
taxa_count( Tree, Count ) :- leaf_count( Tree, Count ).
leaf_count(Tree,C) :- setof(LNode, leaf(Tree,LNode),Nodes), length(Nodes,C).
tree_with_n_taxa( N, Trees ) :- findall(T, (tree(T), taxa_count(T,N)), Trees).
```

Queries Q6: These queries determine all the phylogenies produced by a given tool or author; these can be easily addressed using SPARQL, e.g., to search for a specific author:

```
SELECT ?study WHERE {
 ?study study:has_author ?authorid.
 ?authorid foaf:last_name '$LAST_NAME'^^<http://www.w3.org/2001/XMLSchema#string>.
 ?authorid foaf:first_name '$FIRST_NAME'^^<http://www.w3.org/2001/XMLSchema#string>.}
```

where *$LAST_NAME* and *$FIRST_NAME* are the last and first name of the author, respectively. The option to search on just the last name is also available.

Queries Q7: CDAOStore currently supports only the computation of queries requesting phylogenies with a given constraint on either their radii or their diameters. The code for the radius is provided next—the predicate computes the radii of the trees (through a recursive comparison of distances among leaves) and selects those that have a radius equal to the requested value R. The code for the diameter is analogous, with the exception that we will maximize the eccentricity instead of minimizing it.

```
radius_count( Tree, R, Trees ) :- setof(T, (tree(T), radius(T,R)), Trees).
radius( Tree, R ) :- findall(Leaf, leaf(Tree, Leaf), [L|Leaves]),
   max_distance(Tree, L, Leaves, Curr), radii(Tree, Leaves, R, Curr).
radii( _, Leaf, R, R) :- length(Leaf, 1), !.
radii(Tree, [Leaf | Leaves], Radius, Curr) :-
   max_distance(Tree,Leaf,Leaves, Len),
   (Len < Curr *-> radii(Tree,Leaves,Radius,Len);  radii(Tree,Leaves,Radius,Curr)).
eccList(_,_,[],[]).
eccList(Tree, Leaf, [LeafNode | Leaves], [E | Rest]) :-
   pathlength(Tree,Leaf,LeafNode,E), eccList(Tree,Leaf,Leaves,Rest).
max_distance(Tree,Node,Nodes,D) :- eccList(Tree,Node,Nodes,Lens), max_list(Lens,D).
```

Queries Q9: This query is used to determine all phylogenies that have been constructed using a particular model of evolution; this query is mapped to a SPARQL query that filters phylogenies based on the model of evolution property. Unfortunately, the TreeBase repository is lacking this type of meta-data, preventing us from experimenting with it.

Queries Q10: This class of queries is used to determine statistical information about the phylogenies present in the repository (e.g., distribution of tree lengths). The code has been developed to support the computation of the mean, median, mode, and standard deviation of size of trees, edge lengths, radii, and diameters of all phylogenies in the repository. These queries are implemented in Prolog. Here below we sketch its main aspects.

```
stat_measures(Type, Mean, Median, Mode, Dev) :-
   select_data(Type, List), msort(List, Sorted),
   find_mean(Sorted, Mean), find_median(Sorted, Median),
   find_mode(Sorted, Mode), find_stddev(Sorted, Dev).
select_data(size, List) :- findall(C, node_count(Tree,C), List).
select_data(edge_length, List) :-
   findall(C, (edge(Tree,Name,_,_,_), edge_length(Name,C)), List).
...
findMean(List, Mean):- sum(List, Sum), length(List, N), Mean is Sum / N.
```

Queries Q11: This class of queries is used to determine, given a type of data and a set of taxa, all the phylogenies containing all the given taxa and whose construction involved data of the given type (e.g., DNA). This type of queries can be addressed using SPARQL, since CDAO provides properties describing all these features; the overall structure of the query is

```
SELECT ?tree  WHERE {
   ?study has_analysis ?analysis.   ?analysis has_output_tree ?tree.
   ?analysis has_input_matrix ?matrix.
   ?tree has_TU <TU1>.   . . .   ?tree has_TU <TUN>.
   ?matrix has_Character ?character.
   ?character rdf:type cdao:AminoAcidResidueCharacter. }
```

5 Preliminary Evaluation

The CDAOStore has been implemented and it is now available for access. The implementation has been realized using a collection of publicly available tools. In particular, the triple store has been implemented on top of the RDFlib Python library, the Prolog components in SWI-Prolog, and various libraries have been adopted to deal with specialized data formats. The store has been populated by importing into CDAOStore the complete content of the TreeBase [1] repository, encoded using CDAO and enhanced with semantic annotations drawn from several associated repositories. At this time, CDAOStore contains over 4,000 phylogenies, and the overall size of the repository is now at more than 90 million RDF triples.

We performed some preliminary experiments to evaluate the performance of the system on some sample queries, drawn from scientific publications. Let us observe that performance was not one of the main driving criteria in initial design and implementation of CDAOStore—we focused more on providing an environment that is flexible and provides querying capabilities that are beyond what supported in existing repositories. Furthermore, CDAOStore is viewed as a service provider that will be used by other clients for various types of applications.

The performance in answering the queries has been measured in terms of time to respond. Since our Web portal uses cgi-bin, we evaluated performance by performing a direct call with a specific URL, instead of going through the actual HTML form, in order to get a more accurate time measurement. The queries have been performed on a server running on a HP Intel Core i7 860 machine, with 8GB of memory and making use of SuSE Linux.

Table 1 Execution Times for Sample Queries (time in sec.)

Query	PhyloWS Time	Web Portal Time
T1	2.44	1.20
T2	1.82	3.18
T3	0.91	4.19
T4	6.12	6.18
T5	6.19	6.26
T6	32.58	35.22
T7	5.42	5.04
T8	15.91	*

Table 2 summarizes the queries and results, while Table 1 summarizes some of the execution times. The timings are expressed in seconds. The different rows correspond to different queries. The first time column reports the response time while issuing the query through the PhyloWS API, while the second made use of the web portal. Note that occasionally the web portal provides a faster response time as some of the argument parsing is pre-determined by the specific fields in the portal. We do not report results for queries of type 9 because meta-data about models of evolution are missing from the current repository. Queries 10 and 11 have been only very recently integrated in the system. A particular note for query of type 7—the pre-processing of the query required excessive time, leading to a time-out of the web server; we are working on addressing this issue.

Table 2 Sample queries

Query	Description	Query Type	Answer
T1	Phylogenies containing taxa `Ilex anomala` and `Ilex glabra`	**Q1**	16 phylogenies
T2	Minimum Spanning Clade for taxa `Ilex anomala` and `Ilex glabra` in tree `Tree3099`	**Q3**	1 clade, 14 nodes
T3	Nearest Common Ancestor of taxa `Ilex anomala` and `Ilex glabra` in tree `Tree3099`	**Q3**	1 node
T4	Phylogenies constructed using Parsimony algorithm	**Q4**	3,636 phylogenies
T5	Phylogenies constructed using PAUP*	**Q4**	4,091 phylogenies
T6	Phylogenies with less than 25 nodes	**Q5**	3,120 phylogenies
T7	Phylogenies co-authored by W.H. Piel	**Q6**	3 phylogenies
T8	Phylogenies with radius equal to 10	**Q7**	1 phylogeny

Some of the queries require a larger execution times (in the order of several tens of seconds), due to the fact that the queries check all possible phylogenies in the triple store, rather than looking only at nodes from a particular phylogeny. Some ways to improve these execution times may include maintaining summary information of the various phylogenies. An asynchronous query mechanism would also be useful for running queries such as radius and diameter, avoiding the web server timeout issue mentioned earlier.

6 Conclusion and Future Work

The CDAOStore is a collaborative effort to implement a repository of results from phylogenetic analysis studies in the field of life sciences, built on a formally specified inter-operation stack, the EvoIO stack, composed of a formal ontology, a standard exchange format, and a web service API. The first deployment of CDAOStore has been completed with success, embedding a sophisticated domain-specific querying API and Web interface, implemented using a combination of SPARQL and Prolog. The CDAOStore platform is open-source and is available as a SourceForge project, at `sourceforge.net/projects/cdaotools`. The portal to CDAO-Store is available at `http://www.cs.nmsu.edu/~cdaostore`.

The novelty of CDAOStore lies in the use of a semantic-based approach to the storage and querying of data, building on established ontologies for the semantic annotation of data and on a query language which is domain-specific. These are features that are absent from related existing repositories (e.g., TreeBase [1], Tree of Life Web project [9], Dryad `datadryad.org`). This approach enables us to overcome restrictions imposed by the use of specific data formats (facilitating interoperation among phylogenetic analysis applications) and makes it possible to formulate more meaningful domain-specific queries.

We are currently working on extending the set of domain-specific queries supported by CDAOStore, paying particular attention at queries used to discover clustering of taxa and clustering of phylogenies, according to different types of distance measures. We are also exploring ways of enhancing the speed of some queries, through better representations and pre-computation of additional meta-data. An important component of our future work will include the evaluation of the suitability of the current platform (based on RDFlib) to sustain the growing size of the triple store. Alternative platforms are available (e.g., Jena, RAP, AllegroGraph), with probably better performance but also with steeper amount of work required to integrate the various system components.

References

1. Treebase. `http://www.treebase.org`, 2010.
2. H. Ellegren. Comparative genomics and the study of evolution by natural selection. *Molecular Ecology*, 17(21):4586–4596, 2008.
3. J. Felsenstein. PHYLIP: Phylogeny Inference Package. *Cladistics*, 5:164–166, 1989.
4. V. Gopalan, W. Qiu, M. Chen, and A. Stoltzfus. Nexplorer: phylogeny-based exploration of sequence family data. *Bioinformatics*, 22(1):120–121, 2006.
5. A. Hill, S. Pick, and R. Guralnick. PhyloBox, 2010.
6. J.R. Iglesias, G. Gupta, E. Pontelli, D. Ranjan, and B. Milligan. Interoperability between bioinformatics tools: A logic programming approach. In *PADL*, pages 153–168. Springer Verlag, 2001.
7. S. Kumar, J. Dudley, M. Nei, and K. Tamura. MEGA: A Biologist-centric Software for Evolutionary Analysis of DNA and Protein Sequences. *Briefings in Bioinformatics*, 9:299–306, 2008.
8. H. Lapp and R. Vos. Phyloinformatics web services api: Overview. `https://www.nescent.org/wg/evoinfo/index.php?title=PhyloWS`, National Evolutionary Synthesis Center, 2009.
9. D. Maddison, K. Schulz, and W. Maddison. The Tree of Life Web Project. *Zootaxa*, 1668:19–40, 2007.
10. D. Maddison, D. Swofford, and W. Maddison. NEXUS: an Extensible File Format for Systematic Information. *Syst. Biol.*, 46(4):590–621, 1997.
11. W. Maddison and D. Maddison. Mesquite: a modular system for evolutionary analysis. `http://mesquiteproject.org`, 2010.
12. L. Nakhleh, D. Miranker, and F. Barbancon. Requirements of phylogenetic databases. In *Third IEEE Symposium on Bioinformatics and Bioengineering*, pages 141–148. IEEE, 2003.
13. NESCent. Supporting nexus. `https://www.r-phylo.org/wg_phyloinformatics/Supporting_NEXUS`, National Evolutionary Synthesis Center, 2008.
14. W.H. Piel. Phyloinformatics and tree networks. In *Computational Biology and Genome Informatics*, pages 239–252. World Scientific Press, 2003.
15. F. Prosdocimi, B. Chisham, E. Pontelli, J.D. Thompson, and A. Stoltzfus. Initial implementation of a comparative data analysis ontology. *Evolutionary Bioinformatics*, 5:47–66, July 2009.
16. E. Prud'hommeaux and A. Seaborne. SPARQL Query Language for RDF. Technical Report REC-rdf-sparql-query-20080115, W3C, 2008.
17. A. Stoltzfus, N. Cellinese, K. Cranston, H. Lapp, S. McKay, E. Pontelli, and R. Vos. The evoio interop project. `http://www.evoio.org/wiki/Main_Page`, National Evolutionary Synthesis Center, 2009.
18. R. Vos. `NeXML`: Phylogenetic data in xml. `http://www.nexml.org`, 2008.
19. C. Webb, D. Ackerly, M. McPeek, and M. Donoghue. Phylogenies and communtiy ecology. *Annu. Rev. Ecol. Syst.*, 33(1), 2002.
20. J. Wieczorek, M. Döring, R. De Giovanni, T. Robertson, and D. Vieglais. Darwin core. `http://rs.tdwg.org/dwc/index.htm`, Darwin Core Task Group / TDWG, 2009.
21. M. Wu and J. Eisen. A simple, fast, and accurate method of phylogenomic inference. *Genome Biology*, 9(10):R151, 2008.
22. X. Xia and Z. Xie. DAMBE: Data analysis in molecular biology and evolution. *Journal of Heredity*, 92:371–373, 2001.

Bayesian Annotation Networks for Complex Sequence Analysis

Henning Christiansen, Christian Theil Have, Ole Torp Lassen and Matthieu Petit

Research group PLIS: Programming, Logic and Intelligent Systems
Department of Communication, Business and Information Technologies
Roskilde University, P.O.Box 260, DK-4000 Roskilde, Denmark
E-mail: {henning,cth,otl,petit}@ruc.dk

Abstract

Probabilistic models that associate annotations to sequential data are widely used in computational biology and a range of other applications. Models integrating with logic programs provide, furthermore, for sophistication and generality, at the cost of potentially very high computational complexity. A methodology is proposed for modularization of such models into sub-models, each representing a particular interpretation of the input data to be analysed. Their composition forms, in a natural way, a Bayesian network, and we show how standard methods for prediction and training can be adapted for such composite models in an iterative way, obtaining reasonable complexity results. Our methodology can be implemented using the probabilistic-logic PRISM system, developed by Sato *et al*, in a way that allows for practical applications.

1998 ACM Subject Classification I.2.6 Learning

Keywords and phrases Probabilistic Logic Bayesian Sequence Analysis

Digital Object Identifier 10.4230/LIPIcs.ICLP.2011.220

1 Introduction

Analysis of DNA is an important example of a complex sequence annotation task which has motivated our approach. The sheer size of data instances and the degree of ambiguity in such tasks pose great challenges for efficient probabilistic analysis. Furthermore, most systems for DNA-analysis used in practice are implemented in low-level programming languages, optimized and tweaked for very specific procedures, thus leading to systems with an unclear semantics and lack of flexibility for the modeling part. A possible shift to using probabilistic-logic systems and languages provides obvious benefits in terms of clear semantics and flexibility, but also introduces potential problems concerning complexity and scalability. We present here a modular approach, in which complex probabilistic-logic models are defined in terms of separate sub-models, each representing a particular interpretation (or "signal") of the input data to be analyzed. The dependencies among the results of analyses performed by these sub-models are described in terms of edges in a Bayesian network. This allows for an implementation based on incremental application of standard methods for prediction and training, one sub-model at a time, thus possibly leading to acceptable complexity. We refer to such modularized models for sequence analysis as *Bayesian Annotation Networks*. We demonstrate an implementation based on PRISM [16], which is a probabilistic extension of Prolog.

© Henning Christiansen, Christian Theil Have, Ole Torp Lassen and Matthieu Petit;
licensed under Creative Commons License NC-ND
Technical Communications of the 27th International Conference on Logic Programming (ICLP'11).
Editors: John P. Gallagher, Michael Gelfond; pp. 220–230

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

2 Probabilistic Annotation Models

Probabilistic-logic models for sequence annotations will be presented in two steps, first the logical part, and then probabilities are added. Notice also, that we abstract away the details of any actual modeling language and the format of probability parameters.

▶ **Definition 1.** An *annotation program*, or just *a program*, is a logic program *prog*, that defines a set of atoms, each of the form:

$$prog(s, a, parents),$$

where

- s is called *the sequence*, and represents the data sequence to be annotated by the program.
- a is called an *output annotation*, and
- *parents* represents zero or more *conditioning annotations*.

The name "*parents*" anticipates the introduction of Bayesian Annotations Networks in section 3 below. They represent annotations produced by other sub-models, serving as conditions for the analysis associated with *prog*.

▶ **Definition 2.** A *probabilistic annotation model*

$$m = \langle prog, \theta \rangle$$

consists of a probabilistic annotation program *prog* and a *parameter* θ. The parameter is element of some data domain which is not specified further, but which gives rise to a well-defined conditional probability distribution for atoms $prog(s, a, parents)$ as follows:

$$P(a \mid s, parents, \theta)$$

The intuition is that θ that associates probabilities to the detailed choices made within *prog* to produce the output annotation a, given a specific sequence s and parent annotations. Notice that our framework captures also analyses that are not necessarily written in a probabilistic-logic language. Notice that our framework captures also analyses that are not necessarily written in a probabilistic-logic language.

▶ **Definition 3.** A *deterministic annotation model* is a program

$$prog(s, a, parents)$$

where, for specific sequence s^0 and $parents^0$, there exists exactly one output annotation a^0, i.e.,

$$P(a^0 \mid s^0, parents^0, \theta) = 1,$$

where θ, in this case, refers to an (empty) parameter which is ignored.

The empty parameter is included for uniformity of notation only. A deterministic annotation model with empty parents may represent an analysis provided by an external tool that, e.g., searches for similarities in a database of related sequence data.

3 Organizing Annotation Models as a Bayesian Network

Our overall idea for prediction is to evaluate one model at a time, fix its output annotation to a single "best" one which, then, is used as parent for subsequent analyses. This is very similar to the way forward analysis takes place in Bayesian networks, which we thus take as our central paradigm for putting sub-models together to a whole. A Bayesian Network (BN) is defined as a directed acyclic graph as follows [15].

- Its *nodes* are random variables.
- An *edge* from node A to node B indicates that B is directly dependent on A, and A is called a *parent* of B; the notation $\mathit{parents}(B)$ refers to the sequence of parent nodes of B.
- Each node A has an associated *conditional probability distribution*, *CPD*, $P(A \,|\, \mathit{parents}(A))$.

For many applications of BNs, the CPDs are given in the form of tables, but since the random variables in our case range over huge sets of alternative annotations, this is infeasible, and we use probabilistic models instead.

▶ **Definition 4.** A *Bayesian annotation network* (BAN) is a set of probabilistic annotation models $\{M_i \,|\, i = 1,\ldots,n\}$, with $M_i = \langle m_i(s, a_i, \mathit{parents}_i), \theta_i\rangle$, numbered in such a way that $\mathit{parents}_i \subseteq \{a_1,\ldots,a_{i-1}\}$.
The model M_n is a designated *top model*, and it is assumed that the parent relationship induces a path from any other M_i to M_n.

A BAN in itself is not a BN, but it induces a BN in the following way.
- Nodes are labelled a_i, $i = 1,\ldots,n$ and s.
- Whenever $a_j \in \mathit{parents}_i$, there is an edge from a_j to a_i, and there is an edge from s to any a_i.
- The CPD associated with a_i is given by the model M_i, i.e., $P(a_i \,|\, s, \mathit{parents}_i, \theta_i)$.

For ease of terminology, we refer to a suitable set of annotation programs as a BAN, when below, we talk about training a BAN, i.e., finding parameters such that it actually becomes a BAN, as per the present definition. When presenting a BAN as a graph, we typically leave out s and the n edges going out from it. When doing predictive inference below, the sequence is always fixed, so we can leave it out, assuming instead a particular BN for each sequence.

4 Predictive inference

Predictive inference refers here to the process of identifying a best proposal for top output annotation that characterizes a given sequence. The fundamental assumption when using probabilistic models is that quality of a solution is intimately coupled to its probability, in other words, we should be searching for a top output annotation with a relatively high probability, ideally the one with highest probability.

Below, we give first a precise, declarative characterization of the best top output annotation, and then an approximative calculation method which, under certain circumstances, may reduce computational complexity drastically. Examples and detailed arguments for this claim will be given later.

We assume a BAN $\{M_i \,|\, i = 1,\ldots,n\}$ with $M_i = \langle m_i(s, a_i, \mathit{parents}_i), \theta_i\rangle$ and a fixed sequence s^0 to be analysed. We use Θ to refer to the set of all parameters in the BAN, $\{\theta_1,\ldots,\theta_n\}$. Considering the BAN as an entire model, we can describe the best solution as follows.

$$\mathit{ideal}_n(s^0, \Theta) =_{\mathrm{def}} \operatorname*{argmax}_{a_n} P(a_n \,|\, s^0, \Theta)$$

where the term inside the argmax can be unfolded as follows.

$$P(a_n \,|\, s^0, \Theta) = \sum_{\langle a_1,\ldots,a_{n-1}\rangle} P(a_1,\ldots,a_n \,|\, s^0, \Theta) \quad (1)$$

$$= \sum_{\langle a_1,\ldots,a_{n-1}\rangle} \prod_{i=1}^{n} P(a_i \,|\, s^0, \mathit{parents}_i, \theta_i) \quad (2)$$

Standard methods for reasoning in Bayesian networks, see, e.g., [6], is of very little use here due to the unmanageable size of the random variables' outcome spaces, which in practice are impossible to iterate over.

We are not aware of any reasonable way to reduce this formula, although we do not have a formal proof that this is not possible. Instead, we propose an approximative, iterative algorithm that fixes one particular best annotation $a_i = approx_i(s_0, \Theta)$ for each sub-model and applies it subsequently in the prediction of those a_j with $a_i \in parents(a_j)$.

$$approx_i(s^0, \Theta) = \operatorname*{argmax}_{a_i} P(a_i \mid s^0, approx_{parents_i}(s^0, \Theta), \Theta), i = 1, \dots, n \tag{3}$$

where $approx_{parents_i}(s, \Theta)$, for some sequence s, stands for the sequence of parent annotations $approx_j(s, \Theta)$ for all $a_j \in parents_i$.

Specifically, we take $approx_n(s^0, \Theta)$ as an approximated value for $ideal_n(s^0, \Theta)$; the possible conditions under which this may be considered a good approximation will be discussed among our conclusions, section 8.

Notice, that there is no circularity in this definition and $approx_n(\cdots)$ can be calculated in a single sweep calculating $approx_1(\cdots)$, $approx_2(\cdots)$, ... in that order. The "argmax" in (3) may be calculated by existing algorithms as we demonstrate below.

In practical applications of our methodology, we expect the number of sub-models in a BAN to be a relatively small number (say, arbitrarily, < 10), but lengths of sequences and their annotations are expected to be huge. Measured in sequence length, the complexity of approximate prediction with the entire BAN coincides with the complexity for the most complex sub-model.

5 Training the network

In order to obtain the probabilistic parameters Θ for a BAN, we rely on existing training algorithms for supervised learning, e.g., as built into the PRISM system [7], [17]. Such algorithms require a sufficiently large and representative collection of ground atoms for each sub-model, each representing a sequence with its correct annotation, which in our motivating application domain means annotations verified in the lab by the biologists.

To this end, we assume the availability of some state-of-the-art training algorithm $T^{supervised}$, described as a function mapping a particular program together with its training data into a parameter. Notice that we are not interested here in the actual details of how the training algorithm works.

For doing supervised training of any sub-model in a BAN, we need in principle ground data that exemplifies the relation between sequence, parent annotations, and output annotation. We define, thus, a *conditional training data set* for program m_i as a set

$$CTD_i = \{m_i(s_i^j, a_i^j, parents_i^j) \mid j = 1, \dots\}.$$

It is called "conditional" since it includes parent annotations $parents_i^j$ for each output annotation a_i^j.

In practice, however, we cannot expect such conditional training sets always be available as this assumes that the signals represented by the different sub-models has been analyzed consistently for the same set of sequences. In other words, we can only assume that the following sorts of training data are available in a more traditional format without explicit parent annotations.

$$TD_i = \{\langle s_i^j, a_i^j \rangle \mid j = 1, \dots\}$$

However, if we train the different models one by one in the order M_1, M_2, …, we can use the already trained models to supply parent annotations. We can thus specify an iterative BAN training algorithm as follows.

$$\theta_i = T^{supervised}(m_i, CTD_i)$$

where

$$CTD_i = \{m(s_i^j, a_i^j, approx_{parents_i^j}(s_i^j, \{\theta_1, \ldots, \theta_{i-1}\})) \mid \langle s_i^j, a_i^j \rangle \in TD_i\}$$

There is no circularity in these equations which may be evaluated in one sweep θ_0, θ_1, ….

This strategy can be adapted to handle cases where training data TD_i are unavailable for some non-top model M_i, i.e., $i < n$. Here we may use unsupervised training, or even set the parameters manually, and still hope for good results. It is not essential that model M_i is a faithful mirror of some physically measurable signal (call this M_i^{true}): the necessary property is whether M_i represents *some* annotation that can help the models M_j of which M_i is a parent to discriminate the details of the sequence under consideration. To see this, notice that such an M_j consistently applies annotations produced by M_i (rather than M_i^{true}) for its own training *and* prediction.

We postulate the following rule of thumb for checking the relevance of a specific model M_i within a given BAN.

(*) – Whether a model M_i contributes an interesting signal to M_j can be checked by inspection of the parameter to check whether different values for a_i provide any significant variation in the magnitude of $P(a_j \mid s^0, a_1, \ldots, a_i, \ldots, a_{j-1})$.[1]

However, we expect that models designed according to biological expert knowledge, that are trained using a sufficient set of authoritative data, and whose position in the hierarchy is based on the same biological expert knowledge, will have the best chance to constitute an interesting signal according to (*). In case of a biologically justified model, for which sufficient amounts of data are available, it will be natural also to check it with standard precision and recall methods.

We can summarize some of the practical consequences of these arguments as follows.

- M_i may for reasons of performance, or to avoid over-training, be programmed in a rather coarse way, which gives only a very rough approximation of M_i.
- We may introduce an arbitrary sub-model in a BAN, be it based on only little or no biological insights; it may be trained unsupervised or the parameter may be set by hand, and we can apply (*) to check whether it is of any use.
- We may introduce alternative models for the same biological signal, and use another model as a voting mechanism to combine the different signals and check its contribution according to (*).
- Having a collection of candidate sub-models, we can experiment with different topologies for dependencies, and validate it internally according to (*) as well as using precision and recall tests for the top model.

We will discuss some these points below in relation to our experiments.

[1] For a trained PRISM model, we may compare the different conditional `msw` probabilities produced by the training of M_j.

6 Implementation in PRISM – the LoSt Framework

The methodology described so far is supported by an implementation built on top of the PRISM system [16], which is a probabilistic extension to Prolog which provides a wide range of learning and prediction algorithms.

In this section, we first explain our own system, called the *LoSt-framework* [9], which is basically a collection of scripts that control the ordering of different runs of PRISM for prediction and training plus a file management system that keeps track of the different models, their parameters, the connections that tie them together to a BAN as well as all data files involved (catalogues of sequences, training data, files of predicted annotations, etc.). We also show a simplified, implemented example that illustrates different aspects of our methodology.

6.1 Embedding BANs in PRISM

The PRISM system [16] realizes a probabilistic extension to Prolog and is equipped with a comprehensive collection of facilities for prediction and training.

The PRISM language extends Prolog with so-called multi-valued switches: a call `msw(`*name*`, X)` represents a probabilistic choice of a value to be assigned to `X`.[2] The semantics of a PRISM program is given as a probabilistic Herbrand model, determined by a parameter which is a file of probability declarations for the individual switches. For this semantics to be well-defined, any choice point in the program must be governed by an msw.

The program part of a sub-model $m(s, a, parents)$ may be represented by a PRISM program with a main predicate

`m(`s`, `a`, `$parents$`)`

where *parents* are a arguments corresponding to the number of parents of m_i. A typical sequence model is implemented as a recursive predicate which relates the s and a arguments in a probabilistic fashion conditioned by given parent annotations and involving myriads of `msw` calls.

PRISM contains algorithms for training based on suitable generalizations of EM learning and Variational Bayesian learning [17] which can be used for both supervised and unsupervised learning; the LoSt environment keeps track of training data and generated parameter files for the individual sub-models.

Prediction using a PRISM program, representing a trained sub-model, can be performed using one of PRISM's generalized Viterbi algorithms. Specifically, we use a minor extension to PRISM, described in [3], which makes it possible to analyse longer sequences in reasonable time. The following call,

```
S= ··· , A1= ··· , A2= ··· , viterbiAnnot( m(S,A,A1,A2, ···)),
```

will instantiate `A` to the annotation that provides the highest probability of the goal `m(S,A,A1,A2, ···)`, thus implementing the argmax in equation (3) above.

The scripts in the LoSt environment implement the correct ordering of sub-model processing as prescribed by our incremental prediction and training algorithms described in sections 4 and 5 above, however, avoiding computations that have been made before and whose results are available on files.

[2] To be exact, a switch introduced by a declaration `values(name, [···` *outcomes* `···])` defines a family of random variables, one for each execution of `msw(`*name*`, ···)` in a program run.

6.2 Example: Gene-finding in DNA

We illustrate our methodology showing experiments with BANs that represent gene-finders for DNA sequences. A piece of DNA is a sequence of letters $\{a, c, g, t\}$; it can be viewed as a sequence of triplets, each called a codon. Codons are separated into specific start-codons, stop-codons and other codons; a gene is a specific subsequence matching the codon structure such that it must begin with a start codon and it will definitely end at the next stop-codon; such a syntactic pattern is called an open reading frame (ORF). Our BAN models are designed to annotate ORFs, where the annotation task is to find out whether an ORF contains a gene and, if so, where in the sequence the gene starts.

We define sub-models for different signals — codon preference, gene length and conservation — which are are expected to have influence on whether a sequence is a gene or not. The resulting annotation from such models is a sequence that for each position in the original sequence contains a `1` if the position is predicted as part of a gene and a `0` if it is not.

All our probabilistic models are output HMMs with a gene-state and a non-gene state, which can emit symbols of the annotations of the parent nodes. The transitions between the states reflect the described ORF pattern.

The *codon preference* model **m1** reflects preferential codon usage in the gene and non-gene state. The states can each emit one of the 64 possible codons.

The *gene length* annotation is obtained by using a deterministic model **m2** that annotates each potential start codon with a symbol representing the distance to the upstream stop codon.

Conservation describes a degree to which the codons of a DNA sequence are conserved across species. To detect conservation, each ORF matched to a database of genome sequences of distantly related organisms [3] using the tblastn tool, which produce a gapped alignment of the matches. Only statistically significant matches (evalue $< 10^{-34}$) and only one match per organism are reported. The conservation model **m3** emits identity positions of reported matches to ORFs.

In the following we discuss and assess a number of BAN topologies built using these three signals as basic building blocks. The considered models are **m1**, **m3**, **m1** conditioned on **m2** – **m1(m2)**, **m1** conditioned on **m3** – **m1(m3)**, and **m1** conditioned on both **m2** and **m3** – **m1(m2,m3)**.

We train and predict on the well-annotated *Escherischia Coli* genome and its curated gene annotations from refseq (NC_000913). We have randomly divided the ORFs of the genome into a training and a test set. Supervised training is done using only the former and the method for supervised training algorithm described in section 5. We report prediction accuracy results for both sets. Accuracy is measured as $\mathit{Sensitivity}(SN) = \frac{TP}{TP+FN}$ and $\mathit{Specificity}(SP) = \frac{TP}{TP+FP}$, with respect to annotation of start and stop codons. The results are summarized in table 1.

It can be observed from table 1 that all our models have good generalization capabilities, since the performance is very similar on both the training and test set.

The best model seems to be **m1(m2)**, which achieve a significant increase in specificity with only slightly degraded sensitivity, e.g. it predicts fewer genes but its predictions are more reliable. By them selves, both **m1** and **m3** have reasonable stop specificity, but **m3** has consistent tendency to predict too long genes, leading to severely decreased start specificity.

---

[3] The sequences are from refseq: NC_004547, NC_008800, NC_009436, NC_009792, NC_010067, NC_010694 and NC_011283.

Table 1 Accuracy of predictions using different BAN topologies.

	Training set (114429 ORFs, 2075 genes)				Test set (114404 ORFs, 2065 genes)			
BAN	SN_{start}	SP_{start}	SN_{stop}	SP_{stop}	SN_{start}	SP_{start}	SN_{stop}	SP_{stop}
m1	0.7701	0.2935	0.9711	0.3701	0.7564	0.2920	0.9719	0.3751
m3	0.0636	0.0322	0.8255	0.4183	0.0140	0.0072	0.8412	0.4298
m1(m2)	0.6723	0.5011	0.9345	0.6965	0.6489	0.4896	0.9433	0.7117
m1(m3)	0.4405	0.2243	0.8255	0.4204	0.4315	0.2216	0.8416	0.4323
m1(m2,m3)	0.4361	0.2228	0.8255	0.4217	0.4174	0.2149	0.8416	0.4333

Interestingly, conditioning **m1** on the conservation additional signal **m3** does not improve prediction accuracy much. It does lead to slightly better stop specificity but it tends to degrade the start specificity. Additionally, conditioning on the length signal as done in **m1(m2,m3)** does not seem to help, even though the impact observed in **m1(m2)** was quite significant. It seems that the **m3** signal dominates decisions about which ORFs should be predicted as coding. This effect is apparent from model parameters and it is possible to get an intuition of the problem from inspection of the prediction accuracies.

The **m3** model has a (stop) false negative rate of $1 - SP_{stop} = 1 - 0.4183 \approx 0.58$. The vast majority of ORFs $\sim 98\%$ does not contain genes. The probability that an ORF contains a gene but **m3** classifies it as non-gene is thus relatively small, $1 - 0.98 \times 0.58 \approx 0.11$. In the conditional distribution defined by **m1(m3)** (given predictions of **m3**), it becomes virtually impossible for the viterbi algorithm to classify an ORF as a gene if **m3** has not, since the probability of the gene hypothesis is scaled by ~ 0.11 and the non-gene hypothesis by ~ 0.89.

Part of the explanation is that maximizing the likelihood of observed data (as we do in training) is not equivalent to maximizing prediction accuracy; it may have an adverse effect when selecting predictions as most probable explanations as done by the viterbi algorithm. An other part of the explanation is in our model assumptions; namely, **m1(m3)** is an output HMM that has joint emissions of both codon and the signal from **m3**, and these are dominated by **m3** as explained above. Alternative HMM structures with different constraints and independence tradeoffs might avoid the dominating effect of **m3**. We are still investigating how this is best done.

7 Related work

Our method is closely related to *Dynamic Bayesian Networks* (DBNs) of [10]. By our definition of a BAN, the detailed dependencies between individual models in the network are left abstract, but a concrete instantiation of a BAN may indeed be a DBN. However, as the nodes in a BAN may be arbitrary probabilistic models, for instance context-free grammars, not all BAN instantiations can be represented as DBNs. Oppositely, we only define BANs for discrete models but DBNs may include continuous-valued nodes.

In the realm of classification techniques, it is common to combine the results of different classifiers of the same phenomena in ways such that the combined classifications outperform the individual constituent classifiers. Such methods are generally known by the name *ensemble methods*, which covers a wide range of different ways to the combine classifiers [14]. Our method is related but quite different; this is not just because we consider sequence annotation rather than classification, but also because constituent models of a BAN may model very different phenomena.

In biological sequence analysis, the most successful genome annotation programs are

combiners [4]; programs which combines different sources of annotation evidence using some sort of weighting scheme. Evidence may come in diverse forms, including comparative analysis sources [12], but are typically predictions (e.g. annotations) from other annotation programs (e.g. gene finders). Brent [1] makes a distinction between combiners and joint models, where joint models are described as models which consider the full joint probability distributions evidence and combiners as probabilistic models of the the relative accuracy of evidence sources they are combining. Using our approximate inference algorithm we have a situation similar to combiners in that predictions of parents are combined by child nodes.

While many combiners use non-probabilistic combinations methods, several are explicitly based on principles of (dynamic) Bayesian networks [11, 8]. A main difference is that our framework allows multi-layered and branching topologies where the combiners are usually just single layered probabilistic models.

Our approach also has analogies to *annotation pipelines* [13, 2] where a complex sequence of analysis steps are performed in a possibly branching topology and perhaps synthesized (e.g. by a combiner) in a final annotation as the last step. Opposed to combiners, such pipelines usually allows complex topologies like our framework. However, such pipelines are usually just practical and pragmatic ways of combining existing tools and incorporate probabilistic modeling only to a very limited degree.

There are other declarative approaches to combining evidence in biological sequence analysis. In GAZE [5], a configurable XML-based specification describes a particular composition of evidence sources. However, GAZE integrates existing tools, where our PRISM based approach allows for much more modelling flexibility and have clear and well-defined semantics.

8 Conclusions

We have proposed a Bayesian framework, *Bayesian Annotation Networks*, which allows the representation and composition of models for complex sequence analysis. In a modular way, it supports experimentation with and evaluation of models and signals and it is a practically useful tool for modeling and analyzing sequences. In particular, its applicability to biological sequence analysis has been motivated. We have shown that reasonable complexity can be achieved by the use of tractable, incremental algorithms for inference and training, which can be implemented by successive calls to PRISM, and shown that these algorithms may produce useful annotations.

In general, we have no good analytical or sampling-based principles for analyzing the quality of the approximated annotations compared with the ideal ones. By assumption, the ideal annotations provided by a BAN for a given sequence is too complex to be evaluated, so we need to rely on standard validation techniques based on authoritative test data. However, we will list a few observations which may be used as guidelines.

The crux in our approximate inference algorithm is, in each iteration step, to select a most probable annotation $approx_i$ for each annotation node a_i and take it as a representative for the distribution of all possible a_i values. In the detailed calculations, this means that we use $P(a_j \mid s, \cdots approx_i \cdots)$, for some a_j with $a_i \in parents_j$, as a replacement of a weighted sum over all possible a_i values of $P(a_j \mid s, \cdots a_i \cdots)$.

In the trivial case, where all freedom of choice is implemented in the top node of the Bayesian Network, the approximate algorithm coincides to the ideal. Beyond the trivial case, however, it is difficult (impossible in general) to give sufficient conditions for which the approximate inference method will yield good results.

The relation between the quality of an annotation and its probability is assumed implied by the purpose of a probabilistic annotation model; e.g. it should assign high probability to good annotations. In our definitions of BANs, we define a child model to be dependent on its parent model. In a concrete BAN, however, individual models typically have more fine grained interdependencies, e.g. enforce their inherent independence assumptions. If these assumptions are not faithful to the actual data dependencies, a discordance between annotation quality and probability may arise. Similarly, in the case of approximate inference, we are concerned in particular with the degree of validity of the assumption about independence of parent distributions given the most probable individual elements of those distributions.

Perhaps surprisingly, the annotation quality achieved by the approximate method may be positively affected by correlation between assumed independent nodes of the network. Redundant (correlated) signals does not generally result in better annotations, if the ideal inference method is used. However, such overlapping signals may indeed compensate for the information lost due to the (possibly) unjustified independence assumptions imposed by the approximation method or inherent in constituent models. For instance, information contained in the distribution of a particular parent node, but not reflected by the best annotation from that distribution, may be reflected through the best annotation of some other (correlated) parent.

In practice, we are satisfied with the approximation if the annotations are judged as good using an external measure of quality (e.g. sensitivity/specificity) and we have used cross-validation to build confidence about generality, as demonstrated in section 6.2. Obviously, this may require a considerable amount of, possibly unavailable, labelled training data. A second consequence, also observed in section 6.2, is that the measure optimized by the training algorithm does not necessarily coincide with the external measure of quality. Model constraints and independence assumptions play a key role affecting the correlation between these measures.

References

1. Michael R Brent. Steady progress and recent breakthroughs in the accuracy of automated genome annotation. *Nature Reviews Genetics*, 9(1):62–73, 2008.
2. Brandi L Cantarel, Ian Korf, Sofia M C Robb, Genis Parra, Eric Ross, Barry Moore, Carson Holt, Alejandro Sánchez Alvarado, and Mark Yandell. MAKER: an easy-to-use annotation pipeline designed for emerging model organism genomes. *Genome Research*, 18(1):188–196, 2008.
3. Henning Christiansen and John Gallagher. Non-discriminating Arguments and their Uses. In *Logic Programming, 25th International Conference, ICLP 2009, Lecture Notes in Computer Science 5649*, pages 55–69. Springer, 2009.
4. Roderic Guigó, Paul Flicek, Josep F Abril, Alexandre Reymond, Julien Lagarde, France Denoeud, Stylianos Antonarakis, Michael Ashburner, Vladimir B Bajic, Ewan Birney, Robert Castelo, Eduardo Eyras, Catherine Ucla, Thomas R Gingeras, Jennifer Harrow, Tim Hubbard, Suzanna E Lewis, and Martin G Reese. EGASP: the human ENCODE Genome Annotation Assessment Project. *Genome Biology*, 7(Suppl 1):S2, 2006.
5. Kevin L Howe, Tom Chothia, and Richard Durbin. GAZE: A Generic Framework for the Integration of Gene-Prediction Data by Dynamic Programming. *Genome Research*, 12(9):1418–1427, 2002.
6. Finn V. Jensen and Thomas D. Nielsen. *Bayesian Networks and Decision Graphs*. Springer, 2 edition, 2007.

7 Yoshitaka Kameya and Taisuke Sato. Efficient em learning with tabulation for parameterized logic programs. In John W. Lloyd, Verónica Dahl, Ulrich Furbach, Manfred Kerber, Kung-Kiu Lau, Catuscia Palamidessi, Luís Moniz Pereira, Yehoshua Sagiv, and Peter J. Stuckey, editors, *Computational Logic*, volume 1861 of *Lecture Notes in Computer Science*, pages 269–284. Springer, 2000.

8 Qian Liu, Aaron J Mackey, David S Roos, and Fernando C N Pereira. Evigan: a hidden variable model for integrating gene evidence for eukaryotic gene prediction. *Bioinformatics*, 24(5):597–605, 2008.

9 LoSt. The lost project, 2007. http://lost.ruc.dk.

10 K P Murphy. Dynamic bayesian networks. *Probabilistic graphical models*, 41(November):515–29, 2003.

11 Vladimir Pavlović, Ashutosh Garg, and Simon Kasif. A Bayesian framework for combining gene predictions. *Bioinformatics*, 18(1):19–27, 2002.

12 Maria S. Poptsova and J. Peter Gogarten. Using comparative genome analysis to identify problems in annotated microbial genomes. *Microbiology*, 156(7):1909–1917, 2010.

13 Simon C Potter, Laura Clarke, Val Curwen, Stephen Keenan, Emmanuel Mongin, Stephen M J Searle, Arne Stabenau, Roy Storey, and Michele Clamp. The Ensembl Analysis Pipeline. *Genome Research*, 14(5):934–941, 2004.

14 Lior Rokach. Ensemble-based classifiers. *Artificial Intelligence Review*, 33(1-2):1–39, 2009.

15 Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall Series In Artificial Intelligence. Prentice Hall, 2003.

16 Taisuke Sato. Generative Modeling by PRISM. *Proceedings of the International Conference on Logic Programming*, LNCS 5649:24–35, 2009.

17 Taisuke Sato, Yoshitaka Kameya, and Kenichi Kurihara. Variational Bayes via propositionalized probability computation in PRISM. *Annals of Mathematics and Artificial Intelligence*, 54(1-3):135–158, 2009.

Improving the Outcome of a Probabilistic Logic Music System Generator by Using Perlin Noise

Colin J. Nicholson[1], Danny De Schreye[1], and Jon Sneyers[1]

1 **Katholieke Universiteit Leuven**
Celestijnenlaan 200A
B-3001 Heverlee
Belgium
`<FirstName.LastName>@cs.kuleuven.be`

Abstract

APOPCALEAPS is a logic-based music generation program that uses high level probabilistic rules. The music produced by APOPCALEAPS is controlled by parameters that can be customized by a user to create personalized songs. Perlin noise is a type of gradient noise algorithm which generates smooth and controllable variations of random numbers. This paper introduces the idea of using a Perlin noise algorithm on songs produced by APOPCALEAPS to alter their melody. The noise system modifies the song's melody with noise values that fluctuate as measures change in a song. Songs with more notes and more elaborate differences between the notes are modified by the system more than simpler songs. The output of the system is a different but similar song. This research can be used for generation of music with structure where one would need to generate variants on a theme.

1998 ACM Subject Classification D.1.6 Logic Programming

Keywords and phrases Probabilistic logic, Music generation, Perlin noise

Digital Object Identifier 10.4230/LIPIcs.ICLP.2011.231

1 Introduction

Automatic music generation is a broad area in the field of artificial intelligence, even the subfield of declarative logic programming contains work on the subject [1] [2]. Probabilistic methodologies for generating music are less explored. Research in the field is often specific to certain domains, such as certain types of piano music [5]. The focus of this paper is on music generation in the area of probabilistic logic programming; that is, logic programming with probabilistic features added on. APOPCALEAPS (Automatic Pop Composer And Learner of Parameters) [10] is a music generation program written in the CHRiSM language [11] which builds on a music classification/generation program [12] written in PRISM [9]. CHRiSM is an extension of Constraint Handling Rules [3] that contain multi-headed rules which are triggered only if certain probabilistic conditions are met. Rules can lead to different outcomes for the same input based on set probabilities.

In this paper, the APOPCALEAPS system will be extended by adding a noise component that takes the music files generated by the system and alters the melody to produce new music files that maintain some features of the original piece while showing diversity. The noise feature is based on a variation of Perlin Noise [7] (commonly used in the field of computer graphics to generate natural-looking patterns) which was introduced in [14]. Perlin noise has benefits over random noise when altering a pattern, such as the melody of a song. Perlin noise is controllable: we can decide what sort of a design the noise should take.

© Colin J. Nicholson, Danny De Schreye, and Jon Sneyers;
licensed under Creative Commons License NC-ND
Technical Communications of the 27th International Conference on Logic Programming (ICLP'11).
Editors: John P. Gallagher, Michael Gelfond; pp. 231–239

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

This paper will first introduce the CHRiSM language (Section 2), then go into detail about how APOPCALEAPS generates music (Section 3). Noise generation will be explained in Section 4. In Section 5 we will describe how noise can interact with the current APOPCALEAPS system to produce interesting output. We end with conclusions and future work ideas.

2 CHRiSM

CHRiSM [11] is a programming language based on a combination of concepts of two logic programming languages: CHR [3] and PRISM [11]. The rules of CHRiSM act on constraints, which are similar to Prolog atoms. During execution, all constraints are stored as a multiset. This multiset is referred as the constraint store. An initial store (query) is given to the program, which applies all possible rules before ending with a final store (the result for the query). CHRiSM combines the benefits of CHR with those of PRISM, while maintaining a more user-friendly syntax than a CHR(PRISM) system.

2.1 Constraint Handling Rules

Constraint Handling Rules (CHR) is a programming language introduced by Frühwirth [3]. The name comes from the original intention of adding constraint solvers to a host language. CHR extends a host language. For example, CHR(Prolog) allows one to use Prolog as a host language but with the added benefit of CHR's multiheaded rules. A CHR program is a sequence of CHR rules which can be of the following types:

- Simplification: $h_1, ..., h_n <=> g_1, ..., g_m \mid b_1, ..., b_k$
- Propagation: $h_1, ..., h_n ==> g_1, ..., g_m \mid b_1, ..., b_k$
- Simpagation: $h_1, ..., h_l \setminus h_1, ..., h_n <=> g_1, ..., g_m \mid b_1, ..., b_k$

When the head of a simplification rule matches with corresponding constraints in a constraint store and the (optional) guard conditions are met, then the rule adds constraints corresponding to the body to the constraint store. A propagation rule is the same only the constraints matching with its head are kept in the store. To illustrate the difference between simplification and propagation rules, consider the following example:

Listing 1 CHR example code

```
rain ==> wet.

rain ==> umbrella.
```

The query "rain" will give the result "rain, wet, umbrella". If the two propagation rules were replaced with simplification rules, however, the same query would give the result "wet" or "umbrella," non-deterministically. A simpagation rule removes the constraints matching with the part of the head after the backslash, but keeps those corresponding to the part before it. Consider the following example:

Listing 2 CHR example code

```
male(X) \ female(Y) <=> pair(X,Y).

pair(X,Y) :- write(X), write(' dances with '), write(Y), nl.
```

Here, the simpagation rule at the top leads to a PROLOG rule that outputs "X dances with Y" for variables X and Y. If a male and several females are provided as queries to the program, the output will be the male dancing with each female.

2.2 PRISM

PRISM is an extension of Prolog developed by Sato which includes probabilistic rules [9]. PRISM extends Prolog by adding the msw/2 (multiarity random switch) predicate. The probabilistic switch msw(id,v) enables one to choose a value v from a set labeled with *id*. For example, *id* could be blood type and v could be a, b, or o depending on which value is probabilistically chosen. Below is a sample PRISM program.

Listing 3 PRISM Coin Flip program

```
values(coin,[head,tail]).

direction(D):-
msw(coin,Face),
( Face == head -> D=left ; D=right).
```

In this program, the switch labeled "coin" has two possible values (head or tail) each with a 50% probability of occurrence. The query "direction" will return a value of left if head is chosen and right if tail is chosen. The parameters of the values can be learned from examples or set manually. Learning from examples entails providing a list of observed atoms to an expectation maximization algorithm to find the parameters with the greatest likelihood.

2.3 CHRiSM syntax

The following example from [10] shows a game of "rock, paper, scissors" represented by CHRiSM code:

Listing 4 CHRiSM Rock, Paper, Scissors program

```
player(P) <=> choice(P) ?? rock(P) ; paper(P);  scissors(P).

rock(P1), scissors(P2) ==> winner(P1).
scissors(P1), paper(P2) ==> winner(P1).
paper(P1), rock(P2) ==> winner(P1).
```

Here each player leads to a choice. The "??" introduces three choices (rock, paper, and scissors) of equal probability. Apart from just simulating games for groups of players and returning lists of winners and losers, the constraints can be used to learn playing styles of individual players. [10] has a more detailed analysis of this program and the statistical experiments CHRiSM can perform on it.

2.4 CHRiSM to CHR(PRISM)

A CHRiSM program maintains the ability to use the multiheaded rules of CHR while including the ability to use the probabilities, statistical sampling, and expectation maximization learning of PRISM. A CHRiSM program is translated to a CHR(PRISM) program. As shown in [10], a simplification rule such as

Listing 5 CHRiSM rule

```
player(P) <=> choice(P) ?? rock(P) ; scissors(P); paper(P)
```

is translated to

Listing 6 CHR(PRISM) translation

```
values(choice(_), [1,2,3]).

player(P) <=> msw(choice(P),X),
```

```
(X = 1 -> rock(P); X = 2 -> scissors(P);  X = 3 -> paper(P)).
```

in CHR(PRISM).

3 APOPCALEAPS

APOPCALEAPS is a music generation program written in CHRiSM. APOPCALEAPS generates a song in the form of a text file, and uses the program LilyPond [4] to make a MIDI music file and a PDF file of the sheet music. The program is unique in that it is the first music generation system that is both probabilistic and constraint-based. APOPCALEAPS can use the voices: melody, bass, chords, and drums, or any combination of the four. The user can specify properties of the song to be produced, such as number of measures or the shortest possible duration for the notes of a particular voice.

In its current state, APOPCALEAPS can generate all notes, where X flat is written as (X - 1) sharp. Only the chords C, F, G, A minor, E minor, and D minor (and their corresponding four-note seventh chords) are possible (this is an arbitrary limitation but covers the most common chords in pop music). APOPCALEAPS outputs one chord (or three and four-note chords of the same root) per measure. For example: one measure may have C and C7 as its chords (pauses, called rests, are always possible as well).

Listing 7 Probabilitistic choices in APOPCALEAPS

```
values(chord_choice(C), [c,g,f,am,em,dm]).
values(note_choice(V,C,B), [c,d,e,f,g,a,b,r]).
values(octave_choice(mid), [-2,-1,0,+1,+2]).
values(octave_choice(low), [0,+1,+2]).
values(octave_choice(high), [-2,-1,0]).
values(drum_choice(B), [bd,sn,hh,cymc,r]).
values(chord_type(_,B), [0,7,r]).
values(split_beat(V), [no,yes]).
values(join_notes(V,_,_), [no,yes]).
```

3.1 Chord Generation

APOPCALEAPS uses the chord C major for the first and last measure of a piece if the piece is in major key, and the chord A minor for the first and last measure if the piece is in minor key. The remaining chords are generated by looking at the current chord and choosing a chord to follow it based on preset probabilities for chord transitions (the probabilities were empirically chosen).

Listing 8 CHRiSM rules for Chord Generation

```
key(major), measure(1) ==> mchord(1,c).
key(major), measures(N) ==> mchord(N,c).
key(minor), measure(1) ==> mchord(1,am).
key(minor), measures(N) ==> mchord(N,am).

measures(N) ==> make measures(N).
make_measures(N) <=> N>0 | measure(N), N1 is N-1, next measure(N1,N),
                         make measures(N1).

mchord(X,C), next_measure(X,Y), measures(M) ==> Y < M |
                         msw(chord choice(C),NextC), mchord(Y,NextC).
```

3.2 Rhythm Generation

To generate rhythm, first APOPCALEAPS generates beats in each measure. Each voice has "beat(V,M,N,X,D)" constraints (V = voice, M = measure, N = beat number, X = sub-beat position, D = duration). A chance rule (with probabilities specific to each voice) determines if a beat will be split or not. Beat splitting entails taking a note and turning it into two notes with half the duration of the original note (beat splitting can only occur if the duration of the original beat is longer than the shortest duration the user has specified). Also, beats are sometimes joined together. This occurs with different frequencies depending on whether the notes are in the same measure and belong to the same beat. Like beat splitting, beat joining depends on probabilities unique to each voice.

Listing 9 CHRiSM rules for Splitting and Joining Beats

```
split_beat(V) ??
meter(_,OD), shortest_duration(V,SD) \  beat(V,M,N,X,D),
next_beat(V,M,N,X,Mn,Nn,Z)
<=>  D<SD | D2 is D*2,  Y is X+1/(D2/OD), next_beat(V,M,N,X,M,N,Y),
next_beat(V,M,N,Y,Mn,Nn,Z), beat(V,M,N,X,D2), beat(V,M,N,Y,D2).

join_notes(V,cond Ma=Mb,cond Na=Nb) ??
next_beat(V,Ma,Na,Xa,Mb,Nb,Xb), note(V,Mb,Nb,Xb,SameNote)
\ note(V,Ma,Na,Xa,SameNote) <=> note(V,Ma,Na,Xa,SameNote + '~').
```

4 Perlin Noise

Perlin noise was introduced in 1984 by Ken Perlin [7] as a way of bringing the controlled randomness of nature to computer graphics simulations. For example: one might want to have several patterns in a large cloud of smoke that are similar in shape and movement but not exactly the same. The algorithm behind Perlin noise has been revised slightly over time [8] but remains largely the same. Perlin noise can add variety to many domains besides computer graphics. Perlin noise is bounded, band-limited, non-periodic, stationary and isotropic.

Using a traditional random number generator has a few disadvantages over a Perlin noise algorithm. True randomness (also called white noise) lacks the smoothness of a Perlin noise algorithm, that is more continuous in its output. Also, a Perlin noise algorithm can be controllable. Figure 1 shows a 1-dimensional example of noise. The input must be an integer and the outputs will be random numbers that show no pattern. Figure 2 shows the noise in Figure 1 after it has been smoothed with an interpolation function. Now we can give real input values and the change in them will be much more gradual.

The noise function takes in a coordinate and returns a real number between -1 and 1. The coordinate can be of any dimension. In this paper we will be using 1-dimensional coordinates.

In 2005 Yoon et al. introduced the idea of using Perlin noise to modify melody of an existing song to produce a new song [14] using a Perlin noise variation they had developed earlier [13]. This paper will use the same noise algorithm used in [14]. The steps for the algorithm (with x as a real input value) are as follows:

- 1: Generate M pseudorandom numbers (PRNs) between -1 and 1 and store them in a list G. Make another list (P) which is a random permutation of the set of integers from 0 to M - 1.
- 2: Calculate the integer interval $[q_0, q_1]$, where $q_0 = \lfloor x \rfloor \bmod M$, and $q_1 = (q_0 + 1) \bmod M$.

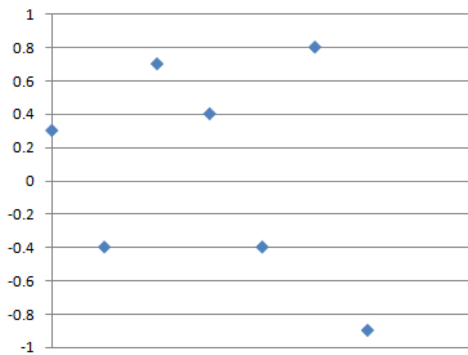

Figure 1 Scattered graph points, representing white noise.

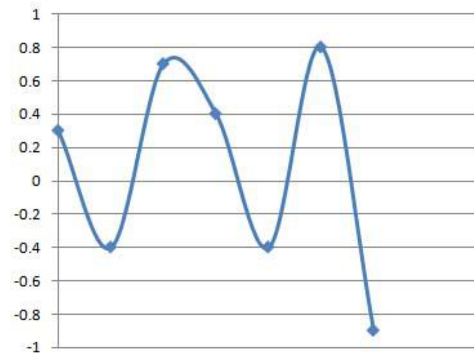

Figure 2 Smoothed graph points, representing Perlin noise.

- 3: Obtain two PRNs $g_j = G[P[q_j]]$, where j = 0, 1.
- 4: The noise value is computed by: Noise(x) = $(1 - s(d_0))g_0d_0+s(d_0)g_1d_1$, where $d_j = q_j - x$, j = 0, 1, and $s(t) = 6t^5 - 15t^4 + 10t^3$.

Here, a pseudorandom number is a number that is returned by a function that appears to be random, but will always return the same number for the same input. The M value can vary; M values of 200 are usual. Also, s(t) is an ease curve. The ease curve in the Perlin noise algorithm takes in a number and returns a number that exaggerates the numbers proximity to zero or one; an input number close to zero would return a number much closer to zero while an input number close to one would return a number much closer to one. This has a smoothing effect on the numbers and will produce waves similar to the ones in Figure 2. The integer interval corresponds to the two integer grid points surrounding our real input value. Each integer point has a PRN (obtained in Step 3 from a list of PRNs generated in Step 1). The d value represents the distance between our input value and the two surrounding grid points. Each grid point's PRN value has an influence on the value of the input point in between. The closer grid point will have greater influence. So, if our grid points were 1 and 2 and our input 1.99, the algorithm would output a PRN that is much closer to 2's PRN than 1's PRN. If the input were 1.5, the output value would be in the middle of 1 and 2's PRN values, and so on.

5 Noise Component in APOPCALEAPS

Until this point, APOPCALEAPS generated one sound file. In this paper, we take files generated by APOPCALEAPS and modify the melodies using the Perlin noise algorithm presented earlier to produce new but similar songs. Figure 3 shows an example of how noise values can modify a melody by altering notes. In this example, a note will stay the same if the noise value is between -0.1 and 0.1; a value between 0.1 and 0.2 will increase the note's position on the scale by one place, above 0.2 will increase the position by two places. Similarly, a noise value between -0.1 and -0.2 will decrease the note's position by one place while a value below -0.2 will decrease the note's position by two places.

5.1 Noise on Notes

As can be seen in Figure 3, each measure contains noise values that are similar. This has been proven to work well for songs with a small amount of notes in each measure. It enables

consistent changes to notes that are grouped together, while enabling diversity for the entire song. Generating similar noise values for notes within a measure is done simply by providing close input values to the Perlin noise algorithm. So real input values are fed to the noise algorithm which are close, but change drastically, every time the measure changes.

However, we must also consider cases where there are several notes within a certain measure. In fact, it is possible for APOPCALEAPS to produce music that has many notes all in one measure; we would not want to use very similar noise values for all of these notes just because they are in the same measure, the output would not show much diversity. So, if a measure contains too many notes, we split the measure into two measures (each with distinct noise values) and distribute the notes equally among the new measures. Determination of whether or not to split a measure depends on the ratio of the shortest note duration in the measure and the duration of the measure itself (the maximum number of notes we can fit in a measure if it was exclusively filled with notes of the shortest duration). That raio must have an upper bound of six, or we split measures until that ratio is below six. It is not necessary to enforce a lower bound since we do not need to group notes in terms of noise that were not in the same measure in the original song.

If a note is a rest in the original melody, it will remain a rest in the new melody. Apart from just changing notes, we can also change the duration (the length of time a note is held). The main aspects of the piece (meter, tempo, length) will remain the same.

5.2 Duration Modification

The duration of notes must be changed in a precise manner. The sum of the duration for all the notes in each measure of the melody must be the same in a changed song as it was in the original song or the melody will lose connection with the rest of the song. Experiments with changes in duration seem to indicate that large changes can cause a melody to lose smoothness and become unlistenable.

One method that works well for duration changes is to produce an altered melody that uses a random permutation of the duration values for the notes in each measure. This ensures each measure maintains the same duration values (though in a different order) as the original song contained. Also, some simple songs (such as "Twinkle Twinkle Little Star") just have a few notes with the same duration values in each measure. These values are kept the same as altering them can be too dramatic a change for a song with a just few notes in each measure. So, while more simple songs are more protected from atonality, they are also less diverse when changed by the system, in terms of note duration.

It is also possible to use Perlin noise to change the duration values. Noise can alter the

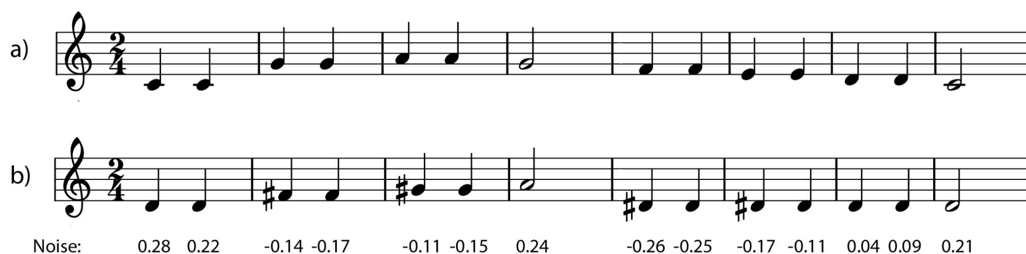

Figure 3 Example of Noise Effect on Notes: a) original melody b) melody modified with noise on each note

duration of a few notes while the durations of the rest of the notes can be altered to balance out the difference from the change brought by the noise. At this point, using noise to alter duration produces limited results that are not better than using permutations. However, as the noise feature is extended in later work to include possibilities such beat splitting or joining, using noise to alter durations as well as note values could become the better option.

5.3 Conclusions From Noise Experiments

Songs generally sound better when the melody rounds the new (noise altered) notes to the nearest chord note (dictated by the chord in the corresponding measure) as opposed to just changing to the note returned by the noise function. However, using chord rounding with every melody can produce output which is too similar. It is more desirable for the purpose of machine learning to have distinct melodies created each time the noise function is run on the same input melody. Chord rounding should be done for some of our output melodies, but not all. Songs with more notes can generally handle more variation in the frequency of chord rounding. Also, chord rounding is more important for long notes on strong beat positions than for short notes on intermediate positions.

It is possible for APOPCALEAPS to produce a rest instead of a chord for a given measure. In cases like these, it is best to use the chord from the previous measure when performing chord rounding on the notes (if the chords are all rests, no rounding will be done). This can be seen especially in more simple songs with few notes and not much variation, where the difference in perception between using chord rounding and not can be dramatic.

6 Future Work

Future work for this system involves two main areas: using the information that can be obtained from the new melodies for new purposes and extending the current noise system. For the first area it is clear that the various outputs produced by the noise algorithm pave the way for future work on a machine learning component for the APOPCALEAPS system. The user can hear different melodies produced by APOPCALEAPS and indicate to the system which sound preferable and which do not. A forthcoming machine learning algorithm can use this data sample to improve its rules for melody generation. First, a distance function is needed to show differences between different LilyPond files output by the noise system. At this point, the differences would be related to note and duration changes. The user can then classify the noise variations of a song based on taste. The parameters used by APOPCALEAPS to produce melody can be updated by the new data. Research already exists where previous notes are used to predict ideal new notes for a music system [6].

The second area of research shows much promise as well. The noise component is currently specialized for the melody. It could also be extended to the other voices of the APOPCALEAPS system. Furthermore, instead of just changing notes, the noise function could split or join beats in a similar way that APOPCALEAPS does in rhythm generation. As the APOPCALEAPS system develops, the noise function can be used to modify features not currently present in the system. In addition, the noise patterns could be used to change the volume of a song (to fade in and out).[1]

[1] Examples of APOPCALEAPS songs that were altered by the noise function in this paper can be found online: `http://dooz.myweb.uga.edu/noisecolin/music.html`

Acknowledgements We wish to thank the anonymous reviewers for their helpful comments and suggestions.

References

1 Georg Boenn, Martin Brain, Marina De Vos, and John Fitch. ANTON: Composing Logic and Logic Composing. LPNMR 2009: 542-547.
2 Georg Boenn, Martin Brain, Marina De Vos, and John Fitch. Automatic Composition of Melodic and Harmonic Music by Answer Set Programming. ICLP 2008: 160-174.
3 Thom Frühwirth. Constraint simplification rules. Tech. Rep. ECRC-92-18, European Computer-Industry Research Centre, Munich, Germany, July 1992.
4 Han-Wen Nienhuys and Jan Nieuwenhuizen. LilyPond, a system for automated music engraving. Proceedings of the XIV Colloquium on Musical Informatics (XIV CIM 2003), Firenze, Italy, May 2003.
5 Tae Hun Kim, Satoru Fukayama, Tanuya Nishimoto, and Shigeki Sagayama. Performance rendering for polyphonic piano music with a combination of probabilistic models for melody and harmony. Proceedings of the 7th International Conference of Sound and Music Computing (SMC2010), 23-30, 2010.
6 Tomasz Oliwa and Markus Wagner. Composing Music with Neural Networks and Probabilistic Finite-State Machines. Proceedings of the Sixth European Workshop on Evolutionary and Biologically Inspired Music, Sound, Art and Design (EvoMUSART 2008), Springer Berlin / Heidelberg, Springer, 503-508, 2008.
7 Ken Perlin. ACM SIGGRAPH 84 conference, course in Advanced Image Synthesis, 1984.
8 Ken Perlin. Improving noise. SIGGRAPH 02, Proceedings 729735, 2002.
9 Taisuke Sato. A glimpse of symbolic-statistical modeling by PRISM. Journal of Intelligent Information Systems, 31(2):161176, 2008.
10 Jon Sneyers and Danny De Schreye. APOPCALEAPS: Automatic Music Generation with CHRiSM. 22nd Benelux Conference on Artificial Intelligence (BNAIC'10), Luxembourg, October 2010.
11 Jon Sneyers, Wannes Meert, Joost Vennekens, Yoshitaka Kameya, and Taisuke Sato. CHR(PRISM)-based probabilistic logic learning. 26th International Conference on Logic Programming, Edinburgh, UK, July 2010.
12 Jon Sneyers, Joost Vennekens, and Danny De Schreye. Probabilistic-logical modeling of music. Proceedings of the 8th International Symposium on Practical Aspects of Declarative Languages (PADL06), 6072, Charleston, SC, USA, January 2006.
13 Jong-Chul Yoon, In-Kwon Lee, and Jung-Ju Choi. Editing Noise. Journal of Computer Animation and Virtual Worlds, 15:3, 277287, 2004.
14 Yong-Woo Jeon, In-Kwon Lee, and Jong-Chul Yoon. Generating and modifying melody using editable noise function. In CMMR, volume 3902 of Lecture Notes in Computer Science, 164168. Springer, 2005.

Abduction in Annotated Probabilistic Temporal Logic*

Cristian Molinaro, Amy Sliva, and V. S. Subrahmanian

Department of Computer Science and UMIACS, University of Maryland
College Park, MD 20742, USA
`{molinaro,asliva,vs}@umiacs.umd.edu`

Abstract

Annotated Probabilistic Temporal (APT) logic programs are a form of logic programs that allow users to state (or systems to automatically learn) rules of the form "formula G becomes true K time units after formula F became true with L to U% probability." In this paper, we develop a theory of abduction for APT logic programs. Specifically, given an APT logic program Π, a set of formulas H that can be "added" to Π, and a goal G, is there a subset S of H such that $\Pi \cup S$ is consistent and entails the goal G? In this paper, we study the complexity of the Basic APT Abduction Problem (BAAP). We then leverage a geometric characterization of BAAP to suggest a set of pruning strategies when solving BAAP and use these intuitions to develop a sound and complete algorithm.

1998 ACM Subject Classification I.2.3 Logic Programming, Probabilistic Reasoning

Keywords and phrases Probabilistic Reasoning, Imprecise Probabilities, Temporal Reasoning, Abductive Reasoning

Digital Object Identifier 10.4230/LIPIcs.ICLP.2011.240

1 Introduction

Statements of the form "formula G becomes true K time units after formula F became true with L to U% probability" occur naturally in many different domains. For example, it is common to talk about the probability that certain stock prices will increase or decrease within a given amount of time after an economic announcement is made by the US Federal Reserve. In the same vein, it is normal to say that there is a probability that certain diseases will occur at specific levels within some amount of time after a natural disaster has occurred. And in an all too common scenario for air-travelers, there is a certain probability that a plane will take off within a time T after pushing back from the gate. All of these examples can naturally be expressed as Annotated Probabilistic Temporal (APT) rules [19].

In this paper, we consider the problem of *abduction* in APT logic programs (APT-LPs). There are many cases where abduction in such logic programs is critically needed. For instance, [19] describes how APT logic programs describing the temporal and probabilistic behavior of over 30 terrorist groups were automatically extracted from data about those groups. In those APT logic programs, there are two types of predicate symbols—*action* predicates describing actions the group takes (e.g., suicide bombing, kidnappings), and *environmental* predicates describing the environment in which the group operates (e.g., whether the group gets financial support from a foreign state, whether the group can participate in the electoral process in its country, etc.). In such an application, APT-rules have action atoms in the rule

* Some of the authors were partly supported by ARO grants W911NF0910206 and W911NF0910525.

© C. Molinaro, A. Sliva, V. S. Subrahmanian;
licensed under Creative Commons License NC-ND

1) $interOrgConflict(Group_1) \overset{efr}{\rightsquigarrow} armedAttack(Group_1) : [2, 0.85, 0.95]$
$Group_1$ will use armed attacks as a strategy within two time units of being involved in inter-organizational conflict with a probability between 0.85 and 0.95.

2) $militaryWing(Group_1) \overset{pfr}{\hookrightarrow} bomb(Group_1) : [2, 0.86, 0.91, 0.7, 0.8]$
$Group_1$ will carry out a bombing exactly two time units after establishment of a standing military wing with a probability between 0.86 and 0.91.

Figure 1 A sample APT program based on automatically extracted rules from [19] modeling the behavior of a terrorist organization $Group_1$.

head and environmental atoms in the rule body. Figure 1 provides a small set of such rules. Clearly, defense experts are interested in understanding how they can modify the true set of environmental atoms (subject to constraints specifying what can and cannot be made true) so that a given goal (e.g., reduction of suicide attacks by 10% or more in the next quarter) can be achieved, if possible. In a second application we are building, we are studying the relationship between numerous social-cultural-economic and policy variables with educational outcomes in over 220 countries.[1] Here again, APT rules can describe the temporal relationship between environmental atoms (e.g., percentage of education budget spent on primary education, student-teacher ratio, provision of childcare facilities) and outcomes (e.g., percentage of females enrolled in primary education), together with the probability that those relationships hold. Here, policy makers want to understand what environmental atoms they can make true (subject to some constraints) so that certain desirable outcomes occur during a given time frame with some probability.

In this paper, we consider triples of the form $\langle \Pi, H, g \rangle$ where Π is an APT logic program, H is a set of formulas that can be added (e.g., adding some childcare facilities might be possible, but increasing the share of the education budget spent on primary education might not), and g is a goal we wish to achieve (e.g., improve primary education female enrollment by 10%). The *Basic APT Abduction Problem* (BAAP) tries to find (if possible) a set $S \subseteq H$ such that $\Pi \cup S$ is consistent and entails the goal. In the case of the terrorism and education applications mentioned above, such sets S correspond to things we could do to try to ensure the desired goal occurs at the desired time (with some probability).

The organization and contributions of this paper are as follows. Section 2 briefly reviews the syntax and semantics of APT-LPs from [19]. Section 3 defines the basic APT abduction problem and shows the complexity of checking for the existence of a solution (which is Σ_2^P-complete) and the complexity of checking whether a given set is a solution (which is D^P-complete). Section 4 provides a geometric intuition behind APT-LPs and derives conditions to prune the search space for a solution to BAAP. Section 5 derives a sound and complete algorithm for BAAP. Finally, Section 6 describes related work.

2 Preliminaries

We now briefly recall the syntax and semantics of APT-LPs from [19]. We assume the existence of a finite set $\mathcal{L}_{cons}$ of constant symbols, a finite set $\mathcal{L}_{pred}$ of predicate symbols,

[1] `http://www.aaas.org/news/releases/2011/0113rwanda.shtml?sa_campaign=Internal_Ads%2fAAAS%2fAAAS_News%2f2011-01-13%2fjump_page`, Jan. 13, 2011.

and an infinite set $\mathcal{L}_{var}$ of variable symbols. Each predicate symbol $p \in \mathcal{L}_{pred}$ has an *arity*. A *term* is any member of $\mathcal{L}_{cons} \cup \mathcal{L}_{var}$. If $t_1, \ldots, t_n$ are terms and $p \in \mathcal{L}_{pred}$ is a predicate symbol with arity n, then $p(t_1, \ldots, t_n)$ is an *atom*. Every atom is a formula. If F and G are formulas, then $F \wedge G$, $F \vee G$ and $\neg F$ are formulas.[2] A formula is *ground* iff no variables occur in it. $\mathcal{B}$ denotes the Herbrand base, i.e., the set of all ground atoms. We assume an arbitrarily large, but fixed size window of time $\tau = \{1, \ldots, t_{max}\}$.

▶ **Definition 1** (Annotated formula). If F is a (ground) formula, $t \in \tau$ is a time point, and $[\ell, u] \subseteq [0, 1]$, then $F : [t, \ell, u]$ is an *(ground) annotated formula.*

Intuitively, $F : [t, \ell, u]$ says that F will be true at time t with a probability in the range $[\ell, u]$. We assume the existence of a finite set $\mathcal{F}$ of symbols called *frequency function symbols* denoting "frequency functions," which will be defined shortly.

▶ **Definition 2** (APT rule/program). Let F and G be two (ground) formulas, Δt be a time interval, $[\ell, u] \subseteq [0, 1]$, $\mathsf{fr} \in \mathcal{F}$ be a frequency function symbol, and $[\alpha, \beta] \subseteq [0, 1]$.
1. $F \overset{\mathsf{fr}}{\rightsquigarrow} G : [\Delta t, \ell, u]$ is called an *(ground) unconstrained APT rule.*
2. $F \overset{\mathsf{fr}}{\hookrightarrow} G : [\Delta t, \ell, u, \alpha, \beta]$ is called a *(ground) constrained APT rule.*

An (ground) *APT program* is a finite set of (ground) APT rules and annotated formulas.

We now define the semantics of APT-programs.

▶ **Definition 3** (World). A *world* is any set of ground atoms. We use $2^{\mathcal{B}}$ to denote the set of all worlds.

As a world is just an ordinary Herbrand interpretation, satisfaction of a formula F by a world w, denoted $w \models F$, is defined in the usual way [14].

▶ **Definition 4** (Thread). A *thread* is a mapping $Th : \tau \to 2^{\mathcal{B}}$.

Intuitively, $Th(t)$ is the set of ground atoms which are true at time t according to Th. We use $\mathcal{T}$ to denote the set of all possible threads and now define a tp-interpretation.

▶ **Definition 5** (Temporal Probabilistic Interpretation). A temporal probabilistic (tp) interpretation I is a probability distribution over the set of all possible threads, i.e., $I : \mathcal{T} \to [0, 1]$ and $\sum_{Th \in \mathcal{T}} I(Th) = 1$.

A tp-interpretation I assigns a probability to each thread.

▶ **Definition 6** (Satisfaction of an Annotated Formula). Let $F : [t, \ell, u]$ be a ground annotated formula, and I be a tp-interpretation. We say that I *satisfies* $F : [t, \ell, u]$, denoted $I \models F : [t, \ell, u]$, iff $\ell \leq \sum_{Th \in \mathcal{T}, Th(t) \models F} I(Th) \leq u$.

A tp-interpretation satisfies an annotated formula iff it satisfies all its ground instances. Each frequency function symbol in an APT rule denotes a frequency function (FF) which tries to capture (within a thread) how often G is true Δt units after F. FFs are defined axiomatically by [19].

▶ **Definition 7** (Frequency Function). Let Th be a thread, F and G be ground formulas, and $\Delta t > 0$ be an integer. A *frequency function* fr is a mapping of quadruples $(Th, F, G, \Delta t)$ to $[0, 1]$ such that:

[2] Throughout this paper, negation $\neg$ is treated in a classical manner and not as non-monotonic negation.

(FF1) If G is a tautology, then $\mathsf{fr}(Th, F, G, \Delta t) = 1$.
(FF2) If F is a tautology and G is a contradiction, then $\mathsf{fr}(Th, F, G, \Delta t) = 0$.
(FF3) If F is a contradiction, then $\mathsf{fr}(Th, F, G, \Delta t) = 1$.
(FF4) If G is not a tautology, and either F or $\neg G$ is not a tautology, and F is not a contradiction, then there exist threads $Th_1, Th_2 \in \mathcal{T}$ such that $\mathsf{fr}(Th_1, F, G, \Delta t) = 0$ and $\mathsf{fr}(Th_2, F, G, \Delta t) = 1$.

The rationale behind the above axioms is given in [19], which also introduces two useful FFs, the *point frequency function* (pfr) and the *existential frequency function* (efr):

$$pfr(Th, F, G, \Delta t) = \frac{|\{t : Th(t) \models F \wedge Th(t + \Delta t) \models G\}|}{|\{t : (t \leq t_{max} - \Delta t) \wedge Th(t) \models F\}|}$$

If there is no $t \in [0, t_{max} - \Delta t]$ such that $Th(t) \models F$ then we define *pfr* to be 1. Intuitively, pfr expresses how frequently G follows F in exactly Δt time points.

$$efr(Th, F, G, \Delta t) = \frac{efn(Th, F, G, \Delta t, 0, t_{max})}{|\{t : (t \leq t_{max} - \Delta t) \wedge Th(t) \models F\}| + efn(Th, F, G, \Delta t, t_{max} - \Delta t, t_{max})}$$

Here $efn(Th, F, G, \Delta t, t_1, t_2) = |\{t : (t_1 \leq t \leq t_2)$ and $Th(t) \models F$ and there exists $t' \in [t + 1, \min(t_2, t + \Delta t)]$ such that $Th(t') \models G\}|$. If the denominator is zero (if there is no $t \in [0, t_{max} - \Delta t]$ such that $Th(t) \models F$ and $efn(Th, F, G, \Delta t, t_{max} - \Delta t, t_{max}) = 0$) then we define *efr* to be 1. Intuitively, efr indicates the frequency that G follows F within Δt time points.

We now define satisfaction of APT rules by tp-interpretations.

▶ **Definition 8** (Satisfaction of APT rules)**.** Let r be a ground APT rule with FF fr and I be a tp-interpretation.
1) If $r = F \overset{\mathsf{fr}}{\rightsquigarrow} G : [\Delta t, \ell, u]$, then we say that I *satisfies* r (denoted $I \models r$) iff $\ell \leq \sum_{Th \in \mathcal{T}} I(Th) \cdot \mathsf{fr}(Th, F, G, \Delta t) \leq u$.
2) If $r = F \overset{\mathsf{fr}}{\hookrightarrow} G : [\Delta t, \ell, u, \alpha, \beta]$, then we say that I *satisfies* r (denoted $I \models r$) iff $\ell \leq \sum_{Th \in \mathcal{T}, \alpha \leq \mathsf{fr}(Th, F, G, \Delta t) \leq \beta} I(Th) \leq u$.

Intuitively, the unconstrained APT rule $F \overset{\mathsf{fr}}{\rightsquigarrow} G : [\Delta t, \ell, u]$ evaluates the probability that F leads to G in Δt time units as follows: for each thread, it finds the probability of the thread according to I and then multiplies that by the frequency (in terms of fraction of times) with which F is followed by G in Δt time units according to frequency function fr. It then sums up these products across all threads in much the same way as an expected value computation. For constrained rules, the probability is computed by first finding all threads such that the frequency of F leading to G in Δt time units is in the $[\alpha, \beta]$ interval, and then summing up the probabilities of all such threads. This probability is the sum of probabilities assigned to threads where the frequency with which F leads to G in Δt time units is in $[\alpha, \beta]$. To satisfy the constrained APT rule $F \overset{\mathsf{fr}}{\hookrightarrow} G : [\Delta t, \ell, u, \alpha, \beta]$, this probability must be within the probability interval $[\ell, u]$. A tp-interpretation I satisfies an APT-rule iff it satisfies all its ground instances, and it satisfies an APT-program Π, denoted $I \models \Pi$, iff I satisfies all rules and annotated formulas in it.

An APT program Π is *consistent* iff there exists a tp-interpretation I s.t. $I \models \Pi$. Π *entails* an annotated formula f (resp., an APT rule r), denoted $\Pi \models f$ (resp., $\Pi \models r$), iff every tp-interpretation satisfying Π satisfies f (resp., r) as well.

3 Abduction in APT Logic

Given an APT program modeling a particular situation, we may want to know how to act to induce some goal condition at a given time in the future and within specified probability bounds.

▶ **Definition 9** (Basic APT Abduction Problem (BAAP)). An instance of the basic APT abduction problem is a triple $\langle \Pi, H, g \rangle$, where Π is an APT program, H is a finite set of annotated formulas and g is an annotated formula. $S \subseteq H$ is a solution to BAAP iff $\Pi \cup S$ is consistent and $\Pi \cup S \models g$.

Intuitively, Π models an environment, g is a goal we want to achieve, and H represents possibilities for what we can do in order to achieve the goal. Note that g is an annotated formula of the form $G : [t, \ell, u]$, where G is an *arbitrary* boolean combination.

▶ **Example 10.** Suppose we have an instance $\langle \Pi, H, g \rangle$ of BAAP, where Π is the simple APT program consisting of the following rules:

$$b \overset{pfr}{\rightsquigarrow} a : [1, 0.68, 0.82]$$
$$c \overset{pfr}{\rightsquigarrow} a : [1, 0.47, 0.7]$$

the set of abducibles is $H = \{b : [1, 0.3, 0.5], c : [1, 1, 1]\}$, and the goal is $g = a : [2, 0.6, 0.85]$. In addition, assume that for this problem $t_{max} = 2$. We must find a subset S of formulas from H such that $\Pi \cup S$ is consistent and entails g, that is, a solution where a will be true at time 2 with a probability in the range $[0.6, 0.85]$. Consider the subset $S = \{c : [1, 1, 1]\}$. When this formula is added to Π, we have that the resulting program $\Pi \cup \{c : [1, 1, 1]\}$ is consistent and entails $a : [2, 0.68, 0.7]$, which entails the goal formula $a : [2, 0.6, 0.85]$. Thus, S is a solution to this instance of BAAP.

Different preference criteria among solutions might be expressed in order to identify "preferred" solutions. Due to space constraints, in this paper we restrict ourselves to the Basic APT Abduction Problem defined above, thus no preference relation is considered.

Checking for the existence of a solution is Σ_2^P-complete even for restricted classes of APT programs.

▶ **Theorem 11** (BAAP Existence). *Let $P = \langle \Pi, H, g \rangle$ be an instance of BAAP. Deciding whether a solution exists for P is Σ_2^P-complete. Σ_2^P-hardness holds even if Π is empty.*

Checking if a given set is a solution to BAAP is D^P-complete even for restricted classes of APT programs.

▶ **Theorem 12** (BAAP Checking). *Let $P = \langle \Pi, H, g \rangle$ be an instance of BAAP. Deciding whether $S \subseteq H$ is a solution for P is D^P-complete. Hardness holds even if Π is restricted to be a program containing only annotated formulas, only unconstrained rules, or only constrained rules.*

It is easy to see that $\langle \Pi, H, g \rangle$ does not have solutions if at least one of Π and g is inconsistent; thus, in the rest of this paper we assume that both Π and g are consistent. A consistency checking algorithm is given in [19].

4 Geometric Characterization of APT Abduction

In this section, we present a geometric characterization of abduction in APT logic. As shown in [19], given an APT program Π, we can derive a linear program $\mathsf{LC}(\Pi)$ s.t. the solutions of $\mathsf{LC}(\Pi)$ are in $1-1$ correspondence with the interpretations satisfying Π. Clearly, $\mathsf{LC}(\Pi)$ defines a convex polytope denoted $\mathit{Polytope}(\Pi)$. For any two polytopes P_1 and P_2, we write $P_1 \subseteq P_2$ iff P_1 is contained in P_2, whereas $P_1 \cap P_2$ denotes the intersection of P_1 and P_2.

We can apply this geometric interpretation to BAAP as follows: given an instance $P = \langle \Pi, H, g \rangle$ of the basic APT abduction problem, $S \subseteq H$ is a solution of P iff $\mathit{Polytope}(\Pi \cup S) \neq \emptyset$ and $\mathit{Polytope}(\Pi \cup S) \subseteq \mathit{Polytope}(\{g\})$. Intuitively, we can think of BAAP as the problem of "cutting" $\mathit{Polytope}(\Pi)$ by using half-spaces in H so that we obtain a non-empty polytope which fits inside $\mathit{Polytope}(\{g\})$.

The following simple proposition provides a sufficient condition for the intersection of the polytope of an APT program and the polytope of an annotated formula to be empty. This proposition will effectively provide a pruning condition that we can exploit when looking for a solution.

▶ Proposition 4.1. Let Π be a consistent APT program and $F : [t, \ell, u]$ a consistent annotated formula. If $[\ell', u']$ is a probability interval s.t. $\mathit{Polytope}(\Pi) \subseteq \mathit{Polytope}(\{F : [t, \ell', u']\})$ and $[\ell', u'] \cap [\ell, u] = \emptyset$, then $\mathit{Polytope}(\{F : [t, \ell, u]\}) \cap \mathit{Polytope}(\Pi') = \emptyset$ for any $\Pi' \supseteq \Pi$.

By leveraging the geometric characterization of the APT abduction problem, we can make the following observations that play an important role in developing an algorithm for BAAP.

- If $\mathit{Polytope}(\Pi) \cap \mathit{Polytope}(\{g\}) = \emptyset$, then the problem does not have a solution—no matter how we slice $\mathit{Polytope}(\Pi)$, it will never fit inside $\mathit{Polytope}(\{g\})$. Alternatively, if $\mathit{Polytope}(\Pi) \subseteq \mathit{Polytope}(\{g\})$, then $\emptyset$ is a solution—we do not need to cut $\mathit{Polytope}(\Pi)$ as it already fits within $\mathit{Polytope}(\{g\})$, i.e., $\Pi \models g$.
- If neither of the above conditions holds, then we iterate over all possible subsets of H. There are two possible reasons why a subset S of H may *not* be a solution:
  1. $\mathit{Polytope}(\Pi \cup S) = \emptyset$, that is, $\Pi \cup S$ is inconsistent. In this case, any superset of S is also not a solution.
  2. $\mathit{Polytope}(\Pi \cup S) \neq \emptyset$, but $\mathit{Polytope}(\Pi \cup S) \not\subseteq \mathit{Polytope}(\{g\})$. In this case, any subset of S is also not a solution.

     In addition, Proposition 4.1 allows us to say even more. If $g = G : [t, \ell, u]$ and $[\ell', u']$ is a probability interval s.t. $\mathit{Polytope}(\Pi \cup S) \subseteq \mathit{Polytope}(\{G : [t, \ell', u']\})$ but $[\ell', u'] \cap [\ell, u] = \emptyset$, then $\mathit{Polytope}(\Pi \cup S)$ and $\mathit{Polytope}(\{g\})$ are not overlapping. Thus, any additional cuts to $\Pi \cup S$ would be useless, that is, any superset of S is also not a solution. Intuitively, this is the case where Π has been cut too much by S.

5 Basic APT Abduction Algorithm

We now present a sound and complete algorithm for BAAP. We note that the search space of possible solutions $\mathcal{H} = 2^H$ is a lattice under $\subseteq$. A brute-force algorithm to solve BAAP would traverse this lattice, inspecting one element at a time and checking whether or not that element is a solution. The algorithm would halt when it arrived at a solution (or when all lattice elements had been considered).

▶ **Example 13.** Suppose we have an arbitrary instance $\langle \Pi, H, g \rangle$ of BAAP, where $|H| = 4$. $\mathcal{H} = 2^H$ has 16 elements and is a complete lattice under the $\subseteq$ ordering. All 16 elements need to be considered, either implicitly or explicitly.

Instead of blindly choosing an element from this lattice, we can find a better way of choosing an element $S \in \mathcal{H}$ such that (i) S has a good chance of being a solution, and (ii) if it is not a solution, then we are able to prune the search space to avoid inspecting elements of $\mathcal{H}$ that are definitely not solutions.

As for the first point, we note that bigger subsets of H make more cuts to Π, but they also have a greater chance of being inconsistent with Π. For example, an extreme case would be the entire set H. If $\Pi \cup H$ is consistent, then we can determine whether or not the abduction problem has a solution without looking at any other subset of H; however, $\Pi \cup H$ has a high likelihood of being inconsistent. On the other hand, smaller subsets of H are more likely to be consistent, but we risk not cutting Π enough to entail the goal. For example, $\emptyset$ is consistent, but it is unlikely to be a solution unless $Polytope(\Pi) \subseteq Polytope(\{g\})$. Thus, a "medium-sized" set might be a good compromise between these two extremes.

Regarding the second point above, as already mentioned in the previous section, when we find that a subset $S \in \mathcal{H}$ is not a solution, we can prune the search space by discarding either all supersets or all subsets (or even both when Proposition 4.1 applies) of S. However, as the following example will demonstrate, the amount of potential pruning depends on the size of the subset chosen.

▶ **Example 14.** Consider again the instance of Example 13. Suppose we choose a "small" $S_1 \subseteq H$, e.g., where $|S_1| = 1$. If S_1 is not a solution because $Polytope(\Pi \cup S_1) \not\subseteq Polytope(\{g\})$, then we can discard only the empty set from the search space. On the other hand, suppose we choose a "big" $S_2 \subseteq H$, e.g., where $|S_2| = 3$. If S_2 is not a solution because $Polytope(\Pi \cup S_2) = \emptyset$, then we can discard only H from the search space. Note that if S_1 fails to be a solution because $Polytope(\Pi \cup S_1) = \emptyset$, or S_2 is not a solution because $Polytope(\Pi \cup S_2) \not\subseteq Polytope(\{g\})$, then the search space is pruned more, but these cases are less likely to occur. Thus, in the worst case we will end up pruning only one element. If we choose an $S \subseteq H$ having cardinality 2 and it fails to be a solution, then it is guaranteed that at least 3 elements from the search space will be discarded, regardless of why S is not a solution. A "medium-sized" set looks like a promising choice—because it is located in the center of the lattice surrounded by many elements, it can provide the largest amount of pruning in the worst case.

In the previous example, we saw that a subset S of H having cardinality 2 would be a good choice. In addition, proceeding by choosing medium-sized, centrally-located sets will move towards a solution in a binary search fashion, providing faster convergence. Identifying such a medium-sized set is easy in the first iteration, but as we move through the search space looking for a solution and pruning parts of the lattice, it becomes more difficult to determine how to choose a good element to inspect. Following the same intuitions introduced above, we would like to choose an element of the lattice from an area where there are many other elements, and in addition, choose one that has a medium size among those.

To identify the medium-sized subsets from a populous region of the search space, we use the scoring function defined below.

▶ **Definition 15.** Consider an instance $\langle \Pi, H, g \rangle$ of BAAP. Let $\mathcal{P} \subseteq 2^H$, $S \in \mathcal{P}$, $v_{max} = max\{|S'| \ : \ S' \in \mathcal{P} \wedge S \subseteq S'\}$, and $v_{min} = min\{|S'| \ : \ S' \in \mathcal{P} \wedge S' \subseteq S\}$. We define $score(S, \mathcal{P}) = (v_{max} - v_{min}) - abs(|S| - (v_{max} + v_{min})/2)$.

In the previous definition, $(v_{max} - v_{min})$ gives a measure of how many elements are around S whereas $abs(|S| - (v_{max} + v_{min})/2)$ gives a measure of the distance from S to a medium-sized element among those surrounding S. Note that in this definition, $\mathcal{P}$ is any subset of 2^H, and represents the remaining portions of the original search space $\mathcal{H}$ that have not yet been pruned.

▶ **Example 16.** Once again, consider the instance of Example 13 and let $\mathcal{P}$ be the whole search space $\mathcal{H} = 2^H$. For the top element of the lattice, $v_{max} = 4$, since the largest superset in $\mathcal{P}$ is H itself, and $v_{min} = 0$, since the smallest subset in $\mathcal{P}$ is the empty set. Using the scoring function from Definition 15, the score of the top element is $(4-0)-abs(4-(\frac{4+0}{2})) = 2$. Likewise, we can compute the scores of the remaining elements in the lattice: all sets of cardinality 3 have score 3, all sets of cardinality 2 have score 4, sets of cardinality 1 have score 3, and the score of the empty set is 2. As described in Example 14, the medium-sized elements of size 2 in the middle of the lattice are the most preferable choices for pruning, and have thus been assigned the highest score.

The observations made so far lead us to the algorithm reported below for solving the basic APT abduction problem.

Algorithm 1 Basic Abduction

Input: Instance $P = \langle \Pi, H, g = G : [t, \ell, u] \rangle$ of the basic APT abduction problem
Output: A solution for P if it exists, *false* otherwise

1: **if** $Polytope(\Pi) \cap Polytope(\{g\}) = \emptyset$ **then**
2:    **return** *false*
3: **if** $Polytope(\Pi) \subseteq Polytope(\{g\})$ **then**
4:    **return** $\emptyset$
5: $\mathcal{H} := 2^H$
6: **while** $\mathcal{H} \neq \emptyset$ **do**
7:    Choose $S \in \mathcal{H}$ having maximum $score(S, \mathcal{H})$
8:    $[\ell', u'] = Tightest(\Pi \cup S, G, t)$
9:    **if** $[\ell', u'] \cap [\ell, u] = \emptyset$ **then**
10:       $\mathcal{H} := \mathcal{H} - \{S' \mid S' \in \mathcal{H} \wedge S' \supseteq S\}$
11:    **if** $[\ell', u'] \neq \emptyset$ **then**
12:       **if** $[\ell', u'] \subseteq [\ell, u]$ **then**
13:          **return** S
14:       **else**
15:          $\mathcal{H} := \mathcal{H} - \{S' \mid S' \in \mathcal{H} \wedge S' \subseteq S\}$
16: **return** *false*

The algorithm first checks whether no solution exists because $Polytope(\Pi) \cap Polytope(\{g\}) = \emptyset$ or if the empty set is a trivial solution (lines 1–4). These checks can be done by solving a linear program as described in [19].

After that, in line 6 we begin to traverse the search space $\mathcal{H}$, pruning as the algorithm proceeds. In line 7 we choose a set $S \in \mathcal{H}$ with the maximum score. The function $Tightest(\Pi \cup S, G, t)$ in line 8 returns the tightest interval $[\ell', u']$ s.t. $Polytope(\Pi \cup S) \subseteq Polytope(\{G : [t, \ell', u']\})$ if $\Pi \cup S$ is consistent, otherwise it returns $\emptyset$. Again, [19] shows how this function can be computed by solving a linear program, providing different optimizations that can be applied to significantly reduce the size of these constraints. In lines 9 and 10 the search space is pruned by discarding every superset of S. Note that here we handle both the case where $\Pi \cup S$ is inconsistent and the case where $Polytope(\Pi \cup S)$ and $Polytope(\{g\})$ do not overlap, i.e., when Proposition 4.1 can be applied. If S is a solution, then it is returned

(lines 11–13). Line 15 examines the case where $\Pi \cup S$ is consistent but not a solution—here every subset of S is discarded. If no solution can be found, then the algorithm returns *false*.

▶ **Theorem 17.** *Let P be an instance of BAAP. If P has a solution, then Algorithm* Basic Abduction *returns one solution of P, otherwise the algorithm returns false.*

▶ **Example 18.** Suppose we have an instance $\langle \Pi, H, g\rangle$ of BAAP that we want to solve using Algorithm Basic Abduction, where Π is the simple APT program given in Example 10, $H = \{f_1 = b : [1, 0.4, 0.8], f_2 = c : [1, 0.8, 1.0], f_3 = c : [1, 1, 1], f_4 = b : [1, 1, 1]\}$, and $g = a : [2, 0.6, 0.85]$. First, we check whether the problem has no solution because $Polytope(\Pi) \cap Polytope(\{g\}) = \emptyset$, or a trivial solution if $Polytope(\Pi) \subseteq Polytope(\{g\})$. Since neither of these conditions are true, we begin searching $\mathcal{H}$ in line 6. As described in Example 16, in the initial lattice, elements of cardinality 2 have the maximum score of 4, so we choose the first such set $S = \{f_1, f_2\}$. Next, computing $Tightest(\Pi \cup S, G, t)$ on line 8 returns $[\ell', u'] = [0.68, 0.9]$. We now check to see if S is a solution, and if not, what elements we can prune from $\mathcal{H}$. Because $[\ell', u'] \cap [0.6, 0.85] \neq \emptyset$ on line 9, we cannot prune any supersets of S. However, because $[\ell', u'] \not\subseteq [0.6, 0.85]$ we can prune all subsets of S on line 14, removing the sets $\{f_1\}, \{f_2\}$, and $\emptyset$ from $\mathcal{H}$. We now begin the loop on line 6 again with our pruned $\mathcal{H}$. This time, the maximum score is 2.5 for subsets of cardinality 2 or 3, so we choose one of these elements—the next subset of cardinality 2—$S = \{f_1, f_3\}$. This time, $Tightest(\Pi \cup S, G, t)$ on line 8 returns $[\ell', u'] = [0.68, 0.7]$. Again, $[\ell', u'] \cap [0.6, 0.85] \neq \emptyset$ on line 9, so we continue to the next test on line 12. Since $[0.68, 0.7] \subseteq [0.6, 0.85]$, $Polytope(\Pi \cup S) \subseteq Polytope(\{g\})$, so we return $S = \{b : [1, 0.4, 0.8], c : [1, 1, 1]\}$ as a BAAP solution.

6 Related Work and Conclusion

Though abduction has been extensively studied [3, 7, 16, 8], there is no work that we are aware of that studies abduction in the context of both probabilities and time.

There has been important work on abduction in temporal logic. [2] presents a non-deterministic algorithm to find explanations for temporal phenomena based on a framework called STP. [5] is another important paper that uses constraint checking methods and compilation methods to achieve greater efficiency. [6] develops an SLDNF-based procedure to perform temporal abduction in logic programs based on the Abductive Event Calculus [6, 9, 21, 12] using a notion of partial plans. [1] provides an abduction mechanism that allows the definition of preferences over abductive explanations. However, none of these frameworks involves uncertainty.

Abduction has also been studied in the context of uncertainty. However, all past work on abduction in such settings has been devised under various independence assumptions [18, 17, 4]. The only exceptions are [24, 23] which perform abduction in possible worlds-based probabilistic logic systems such as those of [11], [15], and [10] where independence assumptions are not made. In this paper, no independence assumption is made, and moreover, none of the above frameworks include time in addition to probabilities. The first implementation of abductive logic programs without independence assumptions is contained in [23, 22].

In this paper, we have shown how abduction can be performed in the context of logic programs containing both probabilistic and temporal information. We have formally defined the Basic APT Abduction Problem (BAAP). We have shown that the problem of determining existence of a solution to BAAP is Σ_2^P-complete whereas checking if a given set is a solution to BAAP is D^P-complete. We have developed strategies to prune the search space for a solution and have given an algorithm to compute a solution to BAAP.

Much work remains to be done. It is only recently that practical algorithms to compute consistency and entailment in logic programs with probability and time have been developed based on randomized algorithms [20]. Likewise, it is only recently that algorithms to compute abductive explanations to probabilistic logic programs *without independence assumptions* have been devised [23, 22]. Scalable implementations of algorithms for BAAP are necessary. We are developing randomized and sampling-based algorithms for BAAP in the spirit of [13] which scale well to very large numbers of possible worlds and threads. We are also developing an application of APT Abduction to the problems described in this paper.

References

1. C. Baral. Abductive reasoning through filtering. *Artif. Intell.*, 120(1):1–28, June 2000.
2. V. Brusoni, L. Console, P. Terenziani, and D. Theseider Dupré. An efficient algorithm for temporal abduction. *AI*IA*, pages 195–206, 1997.
3. T. Bylander, D. Allemang, M. C. Tanner, and J. R. Josephson. The computational complexity of abduction. *Artif. Intell.*, 49((1–3)):25–60, 1991.
4. H. Christiansen. Implementing probabilistic abductive logic programming with constraint handling rules. In *Constraint Handling Rules*, pages 85–118. 2008.
5. T. Console, P. Terenziani, and D.T. Dupré. Local reasoning and knowledge compilation for temporal abduction. *IEEE Trans. Knowl. Data Eng.*, 14(6):1230–1248, 2002.
6. M. Denecker, L. Missiaen, and M. Bruynooghe. Temporal reasoning with abductive event calculus. In *In Proc. of the European Conference on Artificial Intelligence*, pages 384–388. John Wiley & Sons, 1992.
7. T. Eiter and G. Gottlob. The complexity of logic-based abduction. *Journal of the ACM*, 42(1):3–42, 1995.
8. T. Eiter, G. Gottlob, and N. Leone. Semantics and complexity of abduction from default theories. *Artif. Intell.*, 90:90–1, 1997.
9. U. Endress, P. Mancarella, F. Sadri, G. Terreni, and F. Toni. The ciff proof procedure for abductive logic programming with constraints. *Logics in Artificial Intelligence*, 3229:31–43, 2004.
10. R. Fagin, J. Y. Halpern, and N. Megiddo. A logic for reasoning about probabilities. *Information and Computation*, 87(1/2):78–128, 1990.
11. T. Hailperin. Probability logic. *Notre Dame Journal of Formal Logic*, 25 (3):198–212, 1984.
12. A. Kakas, R. Kowalski, and F. Toni. Abductive logic programming. *Journal of Logic and Computation*, 2(6):719–770, 1992.
13. S. Khuller, M. V. Martinez, D. S. Nau, A. Sliva, G. I. Simari, and V. S. Subrahmanian. Computing most probable worlds of action probabilistic logic programs: scalable estimation for $10^{30,000}$ worlds. *Ann. Math. Artif. Intell.*, 51(2-4):295–331, 2007.
14. J. W. Lloyd. *Foundations of Logic Programming*. Springer, 2nd edition, 1987.
15. N. Nilsson. Probabilistic logic. *Artif. Intell.*, 28:71–87, 1986.
16. Y. Peng and J. Reggia. *Abductive Inference Models for Diagnostic Problem Solving*. Springer-Verlag, 1990.
17. D. Poole. Probabilistic horn abduction and bayesian networks. *Artif. Intell.*, 64(1):81–129, 1993.
18. D. Poole. The independent choice logic for modelling multiple agents under uncertainty. *Artif. Intell.*, 94(1-2):7–56, 1997.
19. P. Shakarian, A. Parker, G. I. Simari, and V. S. Subrahmanian. Annotated probabilistic temporal logic. *ACM Trans. Comput. Log.*, 12(2), 2011.
20. P. Shakarian, G.I. Simari, and V.S. Subrahmanian. Annotated probabilistic temporal logic: Approximate fixpoint implementation. *ACM Trans. Comput. Log.*, to appear, 2011.

21 M. Shanahan. An abductive event calculus. *Journal of Logic Programming*, 44(1–3):207–240, 2000.

22 G. I. Simari, J. Dickerson, A. Sliva, and V.S. Subrahmanian. Parallel abductive query answering in probabilistic logic programs. *submitted to a technical journal*, Dec. 2010.

23 G. I. Simari, J. P. Dickerson, and V. S. Subrahmanian. Cost-based query answering in action probabilistic logic programs. In *SUM*, pages 319–332, 2010.

24 G. I. Simari and V. S. Subrahmanian. Abductive inference in probabilistic logic programs. In *ICLP'10 (Technical Communications)*, volume 7 of *LIPIcs*, pages 192–201. Schloss Dagstuhl, 2010.

Automatic Parallelism in Mercury

Paul Bone*

Department of Computer Science and Software Engineering
The University of Melbourne
pbone@csse.unimelb.edu.au

National ICT Australia (NICTA)

Abstract

Our project is concerned with the automatic parallelization of Mercury programs. Mercury is a purely-declarative logic programming language, this makes it easy to determine whether a set of computations may be performed in parallel with one-anther. However, the problem of how to determine *which* computations should be executed in parallel in order to make the program perform optimally is unsolved. Therefore, our work concentrates on building a profiler-feedback automatic parallelization system for Mercury that creates programs with very good parallel performance with as little help from the programmer as possible.

1998 ACM Subject Classification D.3.4 Processors

Keywords and phrases Program Optimization, Automatic Parallelism, Mercury

Digital Object Identifier 10.4230/LIPIcs.ICLP.2011.251

1 Introduction

The rate at which computers are becoming faster at sequential execution has dropped significantly. Instead their parallel processing ability is increasing, and multicore computers are now common. It is now necessary to use parallelism to get the most out of a modern processor. However, parallelization in imperative languages is very difficult, logic and functional languages make parallelism easier by supporting *deterministic parallelism*, which can prevent deadlocks and race conditions, and will always compute the same answer for the same inputs regardless of how execution is scheduled.

Unfortunately most programmers are not very good parallelizing their programs. It is very easy to slow a program down by creating a lot of fine-grained parallelism, in which the overheads are more expensive than the benefit of parallel evaluation. In other cases communicating threads may block waiting for one-another to produce some value, to minimize the runtime of the program, the programmer must understand how their program's threads will be scheduled and when during their execution values will be produced and consumed.

Our work attempts to solve these problems by automatically parallelizing programs written in the Mercury programming language. Mercury is a pure logic programming language, it already supports explicit parallelism of dependent conjunctions, as well as powerful profiling tools which generate data for our analysis.

* Paul is supported by an Australian Postgraduate Award and a NICTA top-up scholarship.

© Paul Bone;
licensed under Creative Commons License NC-ND
Technical Communications of the 27th International Conference on Logic Programming (ICLP'11).
Editors: John P. Gallagher, Michael Gelfond; pp. 251–254

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

2 Background

Mercury supports dependant AND parallelism [4, 13], allowing programmers to request parallel evaluation of a conjunction by using an ampersand to separate conjuncts. We are extending this work to enable feedback directed automatic parallelization of Mercury programs.

Mercury's deep profiler [5] gathers detailed information about a programs execution. In particular, it records statistics not just for each predicate but for the chain of ancestor calls that represent the predicate's invocation. For example, if list.map/3 is used in multiple places within a program the deep profiler will record separate data for each use. This enables us to gather accurate information that we use to automatically parallelize the program.

3 Related Work

Mercury has strong mode and determinism systems [12], this makes it easy to detect how many solutions a goal may have as well as the locations within a predicate where variables are bound. Mercury only allows parallelization of goals that never fail, and never produce more than one solution, which is the most common type of goal in Mercury programs. Therefore, discovering producer-consumer relationships is easy compared to Prolog systems supporting AND-parallelism such as [6].

Most research in parallel logic programming so far has focused on trying to solve these problems of getting parallel execution to *work* well, with only a small fraction trying to find when parallel execution would actually be *worthwhile*. Almost all previous work on automatic parallelization has focused on granularity control: parallelizing only computations that are expensive enough to make parallel execution worthwhile [7, 9], and properly accounting for the overheads of parallelism itself [11]. Most of the rest has tried to find opportunities to exploit independent AND-parallelism during the execution of otherwise-dependent conjunctions [10, 3].

We have found that this is far from enough: the majority of conjunctions with two or more goals that are expensive enough to parallelize are dependant conjunctions, and most of these are dependant in such a way that they must almost be executed sequentially, usually because one goal's execution blocks on a variable that won't be produced until much later.

4 Goals

Our first goal is to detect all the parallelism implicit in a Mercury program — this information can be used by a parallelizing compiler to create efficient parallel Mercury by detecting all the profitable parallelism in a Mercury program. The more opportunities for parallelism that can be found, then the more optimally a program can be parallelized.

Given this, our second goal is to choose which set of these opportunities to take advantage of in order to automatically parallelize a program. This will involve parallelizing opportunities that have the best cost-benefit ratio, those that have the most parallelism due to their dependencies. When doing this, it is important to account for the effects that parallel evaluation of one opportunity will have on the benefits of parallelizing other opportunities.

5 Current Status

We have been able to automatically parallelize a number of small programs with abundant parallelism, including those with only dependant parallelism. For these programs we have

recorded speed-ups that meet our expectations. [2]

We have also modified Mercury's deep profiler, making it possible to extract coverage information for every goal in a Mercury program. This is used by our analysis for measuring the parallel overlap between dependant conjuncts.

Based on ThreadScope [8] we are building tools to profile parallel Mercury programs. Note that this profiler is not related to the deep profiler. This enables us to improve Mercury's parallel runtime system and to improve our automatic parallelization tools. This will also help other developers profile their parallel Mercury programs.

We have also made a number of contributions to Mercury's parallel runtime system, making it more efficient.

6 Preliminary Results

Please see [2] for recent benchmarks. This paper was accepted into the ICLP 2011 conference as a full paper, and has been accepted for publication in TPLP.

Please also see [1] For a description of how we intend to modify ThreadScope to support the profiling of parallel Mercury programs. This paper was accepted into the WLPE 2011 workshop associated with ICLP.

I have also presented this research at the Multicore Miniconference associated with the Linux Conference Australia 2010 in Wellington, New Zealand. I have also been invited to speak at The University of New South Wales, and Google Australia.

7 Open issues

There are many ways in which we can improve our work. Firstly, we can use a best-first traversal of the call tree rather than a depth-first traversal. We can also use this to revisit nodes in the call tree after deciding to parallelize their siblings. We can also use parallelization as a specialization.

Often a loop may do very little work per iteration but may iterate many times. In these cases we should use granularity control to create fewer, larger parallel tasks. This should be tied to automatic parallelization so that the cost-benefit ratio of a granularity-controlled loop can be calculated.

We should also implement parallelization of dependant but commutative operations. When operations are commutative we can re-order them provided that the commutative operations are the only ones that are dependant. There are other cases where code can be re-ordered or transformed to improve the parallel speedup.

References

1 Paul Bone and Zoltan Somogyi. Profiling parallel Mercury programs with ThreadScope. In *Proceedings of the 21st Workshop on Logic-based methods in Programming Environments*, Lexington, Kentucky, 2011.

2 Paul Bone, Zoltan Somogyi, and Peter Schachte. Estimating the overlap between dependent computations for automatic parallelization. In *Proceedings of the 27th International Conference on Logic Programming*, Lexington, Kentucky, 2011.

3 Amadeo Casas, Manuel Carro, and Manuel V. Hermenegildo. Annotation algorithms for unrestricted independent and-parallelism in logic programs. In *Proceedings of the 17th International Symposium on Logic-based Program Synthesis and Transformation*, pages 138–153, Lyngby, Denmark, 2007.

4 Thomas Conway. *Towards parallel Mercury.* PhD thesis, Department of Computer Science and Software Engineering, The University of Melbourne, Melbourne, Australia, July 2002.
5 Thomas Conway and Zoltan Somogyi. Deep profiling: engineering a profiler for a declarative programming language. Technical report, Department of Computer Science and Software Engineering, University of Melbourne, Melbourne, Australia, July 2001.
6 Daniel Cabeza Gras and Manuel V. Hermenegildo. Non-strict independence-based program parallelization using sharing and freeness information. *Theoretical Computer Science*, 410(46):4704–4723, 2009.
7 Tim Harris and Satnam Singh. Feedback directed implicit parallelism. *SIGPLAN Notices*, 42(9):251–264, 2007.
8 Don Jones, Jr., Simon Marlow, and Satnam Singh. Parallel performance tuning for haskell. In *Haskell '09: Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*, pages 81–92, New York, NY, USA, 2009. ACM.
9 P. Lopez, M. Hermenegildo, and S. Debray. A methodology for granularity-based control of parallelism in logic programs. *Journal of Symbolic Computation*, 22(4):715–734, 1996.
10 Kalyan Muthukumar, Francisco Bueno, Maria J. García de la Banda, and Manuel V. Hermenegildo. Automatic compile-time parallelization of logic programs for restricted, goal level, independent AND-parallelism. *Journal of Logic Programming*, 38(2):165–218, 1999.
11 Kish Shen, Vïtor Santos Costa, and Andy King. Distance: A new metric for controlling granularity for parallel execution. *Journal of Functional and Logic Programming*, 1999(1), 1999.
12 Zoltan Somogyi, Fergus Henderson, and Thomas Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming*, 26(1-3):17–64, October-December 1996.
13 Peter Wang and Zoltan Somogyi. Minimizing the overheads of dependent AND-parallelism. In *Proceedings of the 27th International Conference on Logic Programming*, Lexington, Kentucky, 2011.

Consistency Techniques for Hybrid Simulations

Marco Bottalico[1]

1 **Dipartimento di Scienze,**
Università "G. d'Annunzio" di Chieti-Pescara, Italy
bottalic@sci.unich.it

Abstract

The goal of this paper is to show consistency techniques methods and hybrid stochastic/deterministic models to describe biochemical systems and their behaviour through the ordinary differential equations.

1998 ACM Subject Classification I.6.1 Simulation Theory

Keywords and phrases Consistency techniques, deterministic approach, stochastic approach

Digital Object Identifier 10.4230/LIPIcs.ICLP.2011.255

1 Introduction and problem description

In this paper, we investigate hybrid methods based on simulation of stochastic and deterministic models for biochemical systems, with consistency techniques in ordinary differential equations to have a preliminary vision on dissimilar methods to simulate different biochemical systems in Biocham.

2 Background and overview of the existing literature

System biology is an interdisciplinary science, integrating experimental activity and mathematical modeling, which studies the dynamical behaviors of biological systems. An important problem in the modeling these systems is to characterize the dependence of certain properties on time and space. One frequently applied strategy is the description of the change of state variables by differential equations. If only temporal changes are considered, *ordinary differential equations* (ODEs) are used; for changes in time and space, partial differential equations are appropriate [3].

A variety of formalisms for modeling biological systems has been proposed in literature but in this paper we want to investigate only the consistency techniques in ordinary differential equations [2] and a new hybrid stochastic and deterministic model for biochemical systems [1]. There are two formalisms for mathematically describing the time behavior of a spatially homogeneous chemical system: the *deterministic approach* and the *stochastic approach.* The deterministic approach regards the time evolution as a continuous, wholly predictable process which is governed by a set of coupled, ordinary differential equations (the "reaction-rate equations"). The stochastic approach regards the time evolution as a kind of random-walk process which is governed by a single differential-difference equation (the "master equation"). Fairly simple kinetic theory arguments show that the stochastic formulation of chemical kinetics has a firmer physical basis than the deterministic formulation, but unfortunately the stochastic master equation is often mathematically intractable [7].

There is also a way to make exact numerical calculations within the framework of the stochastic formulation without having to deal with the master equation directly. We are

© Marco Bottalico;
licensed under Creative Commons License NC-ND
Technical Communications of the 27th International Conference on Logic Programming (ICLP'11).
Editors: John P. Gallagher, Michael Gelfond; pp. 255–260

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

talking about the Monte Carlo procedure to numerically simulate the time evolution of the given chemical system. Like the master equation, this "stochastic simulation algorithm" correctly accounts for the inherent fluctuations and correlations that are necessarily ignored in the deterministic formulation. Moreover this algorithm never approximates infinitesimal time increments dt by finite time steps Δt. The feasibility and utility of the simulation algorithm are demonstrated by applying it to several well-known model chemical systems, including the Lotka model, the Brusselator, and the Oregonator [7].

3 Goal of the research

How we have explained in the previous section, the ordinary differential equations (ODEs) play a crucial role in the deterministic model. A first order (ODE) system $\mathcal{O}$ is a system of the form

$$\begin{array}{rcl} u_1^{'}(t) & = & f_1(t, u_1(t), ..., u_n(t)) \\ u_2^{'}(t) & = & f_2(t, u_1(t), ..., u_n(t)) \\ & \vdots & \\ u_n^{'}(t) & = & f_n(t, u_1(t), ..., u_n(t)) \end{array}$$

In [2] the author uses the vector representation $u^{'}(t) = f(t, u(t))$ or more simply $u^{'} = f(t, u)$. At this point, there are two assumptions:

(1) the function f is sufficiently smooth;

(2) the existence and uniqueness of a solution.

Now, given an initial condition $u(t_0) = u_0$ and for the second assumption, the solution of $\mathcal{O}$ is a function $s^* : R \rightarrow R^s$ satisfying $\mathcal{O}$ and the initial condition $s^*(t_0) = u_0$.

Although for some classes of ODEs the solution can be represented in closed form, most ODE systems cannot be solved explicitly [2]. The *discrete variable method* aim at approximating the solution $s^*(t)$ of any ODE system, not over a continuous range of t, but only at some points $t_0, t_1, ..., t_m$. This method include *one-step methods* and *multi-step methods*; in general these methods do not guarantee the existence of a solution within a given bound.

The *interval analysis method* instead, was introduced by Moore [16] in 1966. These methods provide numerically reliable enclosures of the exact solution at points $t_0, t_1, ..., t_m$. To achieve the result, they typically apply a one-step Taylor interval method and make extensive use of automatic differentiation to obtain the Taylor coefficients[2].

The major problem of interval analysis methods on ODE systems is the explosion of the size of resulting boxes at point $t_0, t_1, ..., t_m$. For the author, there are two reasons for this explosion: at first this method has a tendency to accumulate errors from point to point, second the approximation of an arbitrary region by a box (wrapping effect) may introduce considerable loss of accuracy after a number of steps.

For all these reasons, in[17, 2] they show how to provide a unifying framework to extend traditional numerical techniques to intervals providing reliable enclosures. The first contribution is to extend explicit and implicit, one-step and multi-step methods to intervals. The second one is to generalize interval techniques into a two-step process: a forward process (to compute an enclosure) and a backward process (to reduce this enclosure).

4 Current status of the research

The stochastic effects play an important role in biological processes leading to an increase in stochastic modelling attempts. The main problem related to the stochastic simulations regards times and computations which are very expensive [1].
The stochastic models have gained considerable attention when experiments conducted at the level of single cells showed the existence of a non-negligible level of noise in intracellular processes, like transcriptions and translation [4]. The dynamics of a stochastic system is described by the *chemical master equation* and in the 1976 Gillespie devised two exact algorithms to numerically simulate the stochastic time evolution of coupled chemical reactions, which are equivalent to solving the chemical master equation [7]. Only recently, modifications to the original chemical master equation have been proposed to further speed up simulations. The most important methods involve the averaging over fast reactions [8], application of quasi-steady-state theory [9], grouping together reactions that occur in fast succession [10].

Another strategy is to model those processes that either involve large number of particles or have fast rates, in a deterministic way, keeping stochastic the remaining ones [1]. There are two recent algorithms to simulate biochemical systems in such hybrid framework that have been proposed [11, 12]. In both cases, the main idea is to first predict the time in which a stochastic event should occur and then evolve the system of ordinary differential equations. At specific instant in time, the system is updated, and it is checked whether the stochastic event has to be performed or not. Instead in [1] the authors propose a rigorous mathematical ground for hybrid stochastic and deterministic modelling in a natural way. There are three different algorithms: the direct hybrid method, the first reaction hybrid method and the next reaction hybrid method. The main difference between the first two approaches and the second one is essentially one: they are based on a prediction correction heuristic for the realization of the stochastic part that can be seen as an approximation to the simultaneous solution of the system of ODEs which in [1] are precisely calculated.

Consider N chemical species $S_1, ..., S_N$ involved in M reactions $R_1, ..., R_M$. Chemical species are modelled in terms of number of molecules $X(t) = (X_1(t), ...,$
$X_N(t))$. The reaction rate for each reaction R_j is specified by a so-called propensity function $a_j = a_j(X(t), t)$, which is equal to the product rate constant c_j and the number of possible combinations of reactant molecules involved in reaction R_j. Once a reaction R_j is performed, the number of molecules for each species is updated according to the state change vector v_j, i.e., $X(t) \leftarrow X(t) + v_j$ [1].

The **deterministic model** is based on the law of mass action, where a system of coupled ordinary differential equations (ODEs) is established for the time evolution of the number of molecules $X(t) \in R_+^N$

$$\frac{d}{dt}X(t) = \sum_{j=1}^{M} v_j a_j(X(t), t) \tag{1}$$

with some initial value $X(t_0) \in R_+^N$. While the system should be described as a vector of integers, this model needs real values for $X(t)$. This is however acceptable under the assumption of large number of molecules ($X_i(t) >> 1$) so that the relative error can be neglected [1].

The **stochastic model** is based on physical laws and the idea that chemical reactions are essentially random processes, the stochastic formulation of chemical reactions is given

in terms of a Markov jump process $X(t) \in N^N$ [13]. Its characterization is based on the probability $a_j(X(t), t)dt$ of a reaction R_j occurring in the next infinitesimal time interval $[t, t+dt]$. Denoting by $T_j(t)$ the time at which reaction R_j first occur after t, this amounts to write that

$$\mathbf{P}[T_j(t) \in [t, t+dt] | X(t)] = a_j(X(t), t)dt. \tag{2}$$

In [1] the authors consider a partition of the reactions $R_1, ..., R_M$ into those modelled stochastically (labeled with index $j \in \mathcal{S}$) and those modelled deterministically (labeled with index $j \in \mathcal{D}$). Now we can write the evolution equation for $X(t) \in R^N$ which is given by the following hybrid system

$$dX(t) = \sum_{j \in \mathcal{D}} v_j a_j(X(t), t)dt + \sum_{j \in \mathcal{S}} v_j dN_j(t) \tag{3}$$

To partition the reactions the authors suggest some methods:

- run a fully stochastic realization and analyze the frequencies/propensities of each reaction;
- use biological insight (i.e. in [1] the authors say that seems reasonable to model gene regulatory parts stochastically, while metabolic reactions deterministically);
- for each reaction choose adaptively between two approaches using a criterion based on the number of the molecules and its propensity function.

To check if the algorithms based on hybrid model (direct hybrid method, first and next reaction methods) obtained good results they tested them in a intracellular growth of bacteriophage T7 derived by [14]. From the experiment appears that the hybrid simulations are about 100 times as fast as the fully stochastic ones without compromising the results accuracy (fig. 1).

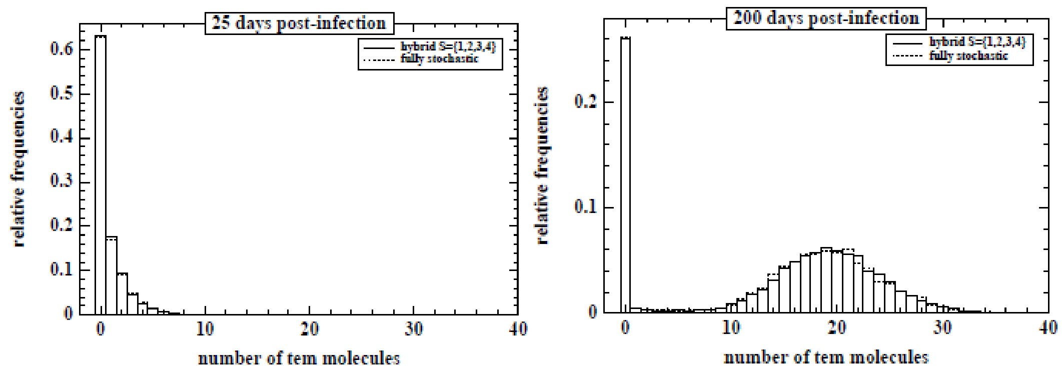

Figure 1 Hybrid kinetics for the bacteriophage T7 model (reaction R_1, R_2, R_3 and R_4 modelled stochastically, reactions R_5 and R_6 modelled deterministically) compared to the the reference fully stochastic model (based on 10^4 realizations) [1].

5 Preliminary results accomplished

The goal is to implement in Biocham some techniques to realize hybrid simulation, combining different kinds and different nature models, in a qualitative and quantitative optical, with discrete and continue dynamics. The solution is to provide the specific language with a

multi level description mechanism for the modelization; the second step is to distinguish in the formalism, the common characteristics from the details. At last we want to specify the criteria to change, during the simulation, the formalism.

6 Open issues and expected achievements

The importance of precise analysis to study and comprise biological phenomena involve different kind of models. On the one hand, it is necessary to describe some parts in a rigorous and accurate numerical method (for example methods based on ordinary differential equations or stochastic methods). On the other hand, the lack of evidence, drives the analysis on purely qualitative models (boolean or discrete models).

References

1 A. Alfonsi and E. Cancès and G. Turinici and B. Di Ventura and W. Huisinga, Exact simulation of hybrid stochastic and deterministic models for biochemical systems, English, MICMAC - INRIA Rocquencourt - INRIA - Ecole Nationale des Ponts et Chaussées , 20, Research Report, INRIA, RR-5435, 2004

2 Y. Deville and M. Janssen and P. Van Hentenrycy, Consistency Techniques in Ordinary Differential Equations, Constraints, 7, 3-4, 2002, 289-315

3 K. Edda and H. Ralf and K. Axel and W. Christoph and L. Hans, 1, Hardcover, 3527310789, Systems Biology in Practice: Concepts, Implementation and Application 2005

4 M.B. Elowitz and A.J. Levine and E.D. Siggia and P.S. Swain, Science, 5584, 1183-1186, Stochastic gene expression in a single cell, 297, 2002

5 V. Gupta and R. Jagadeesan and V.A. Saraswat and DG. Bobrow, Programming in Hybrid Constraint Languages, Hybrid Systems, 1994 ,226-251, P.J. Antsaklis and W. Kohn and A. Nerode and S. Sastry, Hybrid Systems II, Springer, Lecture Notes in Computer Science, 999, 1995, 3-540-60472-3

6 V.A. Saraswat, Concurrent Constraint Programming Languages, The MIT Press, Cambridge, MA, 1993

7 D.T. Gillespie, J. Phys. Chem., 25,, 2340-2361, Exact stochastic simulation of coupled chemical reactions, 81, 1977

8 E.L. Haseltine and J.B. Rawlings, The Journal of Chemical Physics, gillespie_algorithm, kinetics, simulation, stochastic, 6959–6969, 2007-12-11 22:31:52, 2, AIP, Approximate simulation of coupled fast and slow reactions for stochastic chemical kinetics, 117, 2002

9 C.V. Rao and A.P. Arkin, The Journal of Chemical Physics, 4999–5010, 2007-10-23, AIP, Stochastic chemical kinetics and the quasi-steady-state assumption: Application to the Gillespie algorithm, 118, 2003

10 K. Burrage and T. Tian and P. Burrage, Progress Biophys Mol Biol, 217-234, A multi-scale approach for simulation chemical reaction systems, 85, 2004

11 T.R. Kiehl and R.M. Mattheyses and M.K. Simmons, Hybrid simulation of cellular behavior, Bioinformatics, 20, 3, 2004, 316-322

12 K. Takahashi and K. Kaizu and B. Hu and M. Tomita, A Multi-Algorithm, Multi-Timescale Method for Cell Simulation, Bioinformatics, 2004, 20, 538–546

13 D.T. Gillespie, Markov Processes-An Introduction for Physical Scientists, Academic Press, 1992

14 Srivastava, R. and You, L. and Summers, J. and Yin, J., Journal of Theoretical Biology, deterministic, modeling, stochastic_simulation, 3, 309–321, 2008-10-20 , 2, Stochastic vs. Deterministic Modeling of Intracellular Viral Kinetics, 218, 2002

15 L. Calzone and F. Fages and S. Soliman, BIOCHAM: an environment for modeling biological systems and formalizing experimental knowledge, Bioinformatics, 22, 14, 2006, 1805-1807

16 R.E. Moore, Interval Analysis, Prentice-Hall, Englewood Cliffs, N.J., 1966

17 Y. Deville and M. Janssen and P. Van Hentenryck, Multistep Filtering Operators for Ordinary Differential Equations, CP, 1999, 246-260, J. Jaffar, Principles and Practice of Constraint Programming - CP'99, 5th International Conference, Alexandria, Virginia, USA, October 11-14, 1999, Proceedings, CP, Springer, Lecture Notes in Computer Science, 1713, 1999

Extensions of Answer Set Programming

Alex Brik[1]

1 Department of Mathematics, UC San Diego, U.S.A

Abstract

This paper describes a doctoral research in three areas: Hybrid ASP – an extension of Answer Set Programming for reasoning about dynamical systems, an extension of Set Constraint atoms for reasoning about preferences, computing stable models of logic programs using Metropolis type algorithms. The paper discusses a possible application of all three areas to the problem of maximizing total expected reward.

1998 ACM Subject Classification I.2.3 Deduction and Theorem Proving

Keywords and phrases answer set programming, hybrid systems, modeling and simulation, preferences

Digital Object Identifier 10.4230/LIPIcs.ICLP.2011.261

1 Introduction and Problem Description

The research investigates three areas related to ASP.

Area 1: H-ASP. The main motivation for this area is the question of how to reason about dynamical systems that exhibit both discrete and continuous behavior. The unique feature of Hybrid ASP (H-ASP) is that H-ASP rules can be thought of as general input-output devices. In particular, H-ASP programs allow the user to include ASP type rules that act as controls for when to apply a given algorithm to advance the system to the next position. This feature allows H-ASP to be used with Partial Differential Equation (PDE) solvers and Ordinary Differential Equation (ODE) solvers.

Area 2: preferences as SC atoms. The notion of a set constraint atom (SC atom) was introduced by Marek and Remmel [18]. This notions is extended to the notion of an extended set constraint atom (ESC atom) to model preferences.

Area 3: computing stable models of logic programs using Metropolis type algorithms. The Metropolis algorithm introduced by Metropolis et al. [16] in 1953, is a widely applicable procedure for drawing samples from a specified distribution on a large finite set. It was later generalized to the Metropolis-Hastings algorithm [8]. Since its introduction the Metropolis algorithm has found many applications in statistical physics, biology, statistics, and other areas of science [5]. The subject of the research is to produce algorithms that use Metropolis type algorithms for the following two tasks:

1. Given a finite propositional logic program P which has a stable model, find a stable model M of P.

2. Given a finite propositional logic program P which has no stable model, find a maximal program $P' \subseteq P$ which has a stable model and find a stable model M' of P'.

Finding maximal subprograms that have stable models is important for certain extensions of ASP where arbitrary set constraints are used to model both hard and soft preferences. In such situations, one may not be able to satisfy all soft preferences so that stable models may not exist that satisfy all preferences. However, if certain soft preferences are dropped, then the subprograms that do have stable models may be found.

© Alex Brik;
licensed under Creative Commons License NC-ND

The three areas can be combined when one considers certain problems. For instance: what is a next action that an agent acting in a dynamical system has to perform in order to maximize the total expected reward. The idea is to describe a dynamical system in H-ASP, define optimal strategy as a set of preferred stable models and then perform computations using Metropolis type algorithm.

2 Background and Overview of Existing Literature

The following is a review of the definitions of normal propositional logic programs and stable model semantics. A *normal propositional logic program* P consists of clauses of the form $C = a \leftarrow a_1, \ldots, a_m, \neg b_1, \ldots, \neg b_n$ where $a, a_1, \ldots, a_m, b_1, \ldots, b_n$ are atoms. Here $a_1, \ldots, a_n$ are called the *premises* of clause C, $b_1, \ldots, b_m$ are called the *constraints* of clause C, and a is called the *conclusion* of clause C. For any clause C as above, let $prem(C) = \{a_1, \ldots, a_n\}$, $cons(C) = \{b_1, \ldots, b_m\}$, and $c(C) = a$. Either $prem(C)$, $cons(C)$, or both may be empty. C is said to be a Horn clause if $cons(C)$ is empty. Let $mon(P)$ denote the set of all Horn clauses of P and $nmon(P) = P \setminus mon(P)$. The elements of $nmon(P)$ will be called *nonmonotonic* clauses. Let $H(P)$ denote the Herbrand base of P. A subset $M \subseteq H(P)$ is called a **model** of a clause C if whenever $prem(C) \subseteq M$ and $cons(C) \cap M = \emptyset$, then $c(C) \in M$. M is a model of a program P if it is a model of every clause $C \in P$. The Gelfond-Lifschitz reduct of P with respect to M denoted P^M is obtained by removing every clause C such that $cons\,(C) \cap M \neq \emptyset$ and then removing the constraints from all the remaining clauses. M is called a **stable model** of P if M is the least model of P^M.

H-ASP

Modern computational models and simulations such as the model of dog's heart described in [11], or the model of internal tides within Monterey Bay and the surrounding area described in [10] rely on PDE solvers and ODE solvers to determine the values of parameters. Such simulations proceed by invoking appropriate algorithms to advance a system to the next state, which is often distanced by a short time interval into the future from the current state. In this way, a simulation of continuously changing parameters is achieved. The parameter passing mechanisms and the logic for making decisions regarding what algorithms to invoke and when are part of the ad-hoc control algorithm. Thus the laws of a system are implicit in the ad-hoc control software.

Action languages [7] which are also used to model dynamical systems allow the users to describe the laws of a system explicitly. Initially action languages did not allow simulation of the continuously changing parameters. Recently, Chintabathina introduced an action language H [4] where he proposed an elegant approach to modeling continuously changing parameters. However, the implementation of H discussed in [4] cannot use PDE solvers nor ODE solvers. This means that parameters governed by physical processes such as the distribution of heat or air flow, that cannot be described explicitly as functions of time and realistic simulations of which require sophisticated numerical methods, cannot be modeled using the current implementations of H.

Hybrid ASP [3] is an extension of ASP that allows users to combine the strength of the ad-hoc approaches, i.e. the use of numerical methods to faithfully simulate physical processes, and the expressive power of ASP, which provides the ability to elegantly model laws of a system. Hybrid ASP provides mechanisms to express the laws of the modeled system via hybrid ASP rules which can control execution of algorithms relevant for simulation.

Let S be a parameter space and let At be a set of atoms. The universe is $At \times S$. Given $M \subseteq At \times S$ and $B_i = a_1^{(i)}, \ldots, a_{n_i}^{(i)}, \neg b_1^{(i)}, ..., \neg b_{m_i}^{(i)}$, where $a_1^{(i)}, ...a_{n_i}^{(i)}$, $b_1^{(i)}, ..., b_{m_i}^{(i)} \in At$

and $\mathbf{p} \in S$, M satisfies B_i at $\mathbf{p}$, if $(a_j^{(i)}, \mathbf{p}) \in M$ for $j = 1, \ldots, n_i$, and $(b_j^{(i)}, \mathbf{p}) \notin M$ for $j = 1, \ldots, m_i$. **Advancing Rules** are of the form

$$\frac{B_1; B_2; \ldots; B_r : A, O}{a}$$

where A is an algorithm, each B_i is as above, $a \in At$, and $O \subseteq S^r$ is such that if $(\mathbf{p}_1, \ldots, \mathbf{p}_r) \in O$, then $t(\mathbf{p}_1) < \ldots < t(\mathbf{p}_r)$, $A(\mathbf{p}_1, \ldots, \mathbf{p}_r) \subseteq S$, and for all $\mathbf{q} \in A(\mathbf{p}_1, \ldots, \mathbf{p}_r)$, $t(\mathbf{q}) > t(\mathbf{p}_r)$. The idea is that if for each i, B_i is satisfied at $\mathbf{p}_i$, then the algorithm A can be applied to $(\mathbf{p}_1, \ldots, \mathbf{p}_r)$ to produce a set $O' \subseteq S$ such that if $\mathbf{q} \in O'$, then $t(\mathbf{q}) > t(\mathbf{p}_r)$ and $(a, \mathbf{q})$ holds. **Stationary rules** are of the form

$$\frac{B_1; B_2; \ldots; B_r : H, O}{a}$$

where each B_i is as above, $a \in At$, $O \subseteq S^r$ such that if $(\mathbf{p}_1, \ldots, \mathbf{p}_r) \in O$, then $t(\mathbf{p}_1) < \cdots < t(\mathbf{p}_r)$, and H is a Boolean algorithm such that for all $(\mathbf{p}_1, \ldots, \mathbf{p}_r) \in O$, $H(\mathbf{p}_1, \ldots, \mathbf{p}_r)$ is defined. The idea is that if $(\mathbf{p}_1, \ldots, \mathbf{p}_r) \in O$ and for each i, B_i is satisfied at $\mathbf{p}_i$, and $H(\mathbf{p}_1, \ldots, \mathbf{p}_r)$ is true, then $(a, \mathbf{p}_r)$ holds.

Modeling Preferences as SC Atoms

SC atoms were first introduced by Marek and Remmel in [18]. A SC atom is a pair $\langle X, \mathcal{F} \rangle$ where X is a finite set and $\mathcal{F} \subseteq 2^X$. A *set constraint clause* (SC clause) is a string of the form $\langle X, \mathcal{F} \rangle \leftarrow \langle Y_1, \mathcal{G}_1 \rangle, \ldots, \langle Y_n, \mathcal{G}_n \rangle$, where $\langle X, \mathcal{F} \rangle$, $\langle Y_1, \mathcal{G}_1 \rangle, \ldots, \langle Y_n, \mathcal{G}_n \rangle$ are SC atoms. A *set constraint program* (SC program) is a finite set of SC clauses. Let M be a set of atoms and $K = \langle X, \mathcal{F} \rangle$ be a SC atom. Define $M \models K$ if $X \cap M \in \mathcal{F}$. Marek and Remmel in [18] defined the stable model semantics for SC program using the notion of NSS transform.

The basic idea of using SC atoms to define preferences is to consider triples of the from $\langle X, \mathcal{F}, wt \rangle$ or $\langle X, \mathcal{F}, \preccurlyeq \rangle$ where X is a finite set of atoms, $\mathcal{F} \subseteq 2^X$, $wt : \mathcal{F} \to [0, \infty) \subseteq \mathbb{R}$, $\preccurlyeq$ is a partial order in $\mathcal{F}$.

The triples of the form $\langle X, \mathcal{F}, wt \rangle$ are called *weight preference set constraint atoms* and triples of the form $\langle X, \mathcal{F}, \preccurlyeq \rangle$ are called *partially ordered preference set constraint atoms.* A set of atoms M is satisfies $\langle X, \mathcal{F}, wt \rangle$ or $\langle X, \mathcal{F}, \preccurlyeq \rangle$ if and only if M satisfies $\langle X, \mathcal{F} \rangle$. Now suppose that we have a SC program P and an additional finite set of clauses T of the form $\langle X_i, \mathcal{F}_i, wt_i \rangle \leftarrow$, $i \in \{1, \ldots, n\}$. Suppose that M is a stable model of $P \cup T$. Then we can define the weight of the model M as $W(M) = \sum_{i=1}^{n} wt_i(X_i \cap M)$. The weight functions can be used to describe user's preferences for what the user wants $M \cap X_i$ to be by saying that for $F_1, F_2 \in \mathcal{F}_i$, F_1 is preferred over F_2 if $wt_i(F_1) < wt_i(F_2)$. A stable model M_1 of $P \cup T$ is preferred over the stable model M_2 of $P \cup T$ if $W(M_1) < W(M_2)$. Thus the introduction of weight preference set constraint atoms can lead to a natural weighting of stable models which can be used to model preferences. Similarly, suppose that there is an SC program P which in addition has a finite set of clauses T of the form $\langle X_i, \mathcal{F}_i, \preccurlyeq_i \rangle \leftarrow$ for $i \in \{1, \ldots, n\}$. Now suppose that two stable models M_1 and M_2 of $P \cup T$ are given. Then $M_1 \preccurlyeq M_2$ if and only if $M_1 \cap X_i \preccurlyeq_i M_2 \cap X_i$ for $i = 1, \ldots, n$. Thus the introduction of partial order preference set constraint atoms can lead to a natural partial order on stable models which can be used to model preferences.

Computing Stable Models of Logic Programs Using Metropolis Type Algorithms

The use of Metropolis type algorithms to compute stable models of logic programs is based on the Forward Chaining (FC) algorithm introduced by Marek et al.[17]. The FC algorithm provides two ingredients necessary for any procedure to be used with the Metropolis type algorithms: a way to produce a "next" candidate given a current candidate, and a measure of

merit that assigns a numeric score to a candidate (based on how closely it corresponds to a stable model).

Given a Markov chain $K(x,y)$ and a probability distribution $\pi(x)$, let $G(x,y) = \pi(y) K(y,x) / \pi(x) K(x,y)$. The Metropolis-Hastings algorithm defines a new Markov chain $M(x,y)$, where the probability $M(x,y)$ is equal to the probability of drawing $x_{t+1} = y$ given $x_t = x$ using the following procedure:

1. given current state $x_t = x$, draw y based on the Markov chain $K(x,\cdot)$;
2. draw U from the uniform distribution on $[0,1]$;
3. set $x_{t+1} = y$ if $U \leq G(x_t, y)$ and set $x_{t+1} = x_t$ otherwise.

The key result about the Metropolis-Hasting algorithm is the following.

▶ **Theorem 1.** *Let X be a finite set and $K(x,y)$ be a Markov chain on X such that $\forall x,y \in X$ $K(x,y) > 0$ iff $K(y,x) > 0$. Let $\pi(x)$ be a probability distribution on X. Let $M(x,y)$ be the Metropolis-Hastings chain as defined above. Then $\pi(x) M(x,y) = \pi(y) M(y,x)$ for all x, y. In particular, for all x, $y \in X$ $\lim_{n\to\infty} M^n(x,y) = \pi(y)$.*

The relevance of this result to the task of finding stable models of normal propositional logic programs is in the following: after sampling from M for sufficiently many steps the probability of being at y is $\pi(y)$ regardless of the starting state. If $\pi(y)$ is defined to be relatively large whenever y corresponds to a stable model then for a normal propositional logic program P which has a stable model, samples generated from M will eventually include those corresponding to the stable models and moreover the sampling from M will be biased towards those samples that correspond to the stable models of P.

There are various approaches to computing stable models of logic programs. Systems such as *smodels* [20] and *clasp* [6] use complete algorithms that will either find stable models if they exist or will report that stable models do not exist. These systems are not based on the Metropolis type algorithms. The use of the Metropolis type algorithm for the purpose of finding stable models of logic programs was investigated in [13], [14], and [15]. The Metropolis algorithm is also used in SAT solvers [19].

3 Goal of the Research

Each of the 3 areas studied has its own goal. Thus the goal of the research in the area of Hybrid ASP is to produce theoretical machinery necessary to build an H-ASP software system that is able to reason about dynamical systems that exhibit both discrete and continuous behavior, and to produce a prototype of such a software system.

In action languages like H, the goal is to compile an H program into a variant ASP program that can be processed with variant ASP solvers. A long term goal of this research is to develop extensions of ASP solvers that can process Hybrid ASP programs. This would allow the development of extensions of action languages like H that could be compiled to Hybrid ASP programs which, in turn would be processed by Hybrid ASP solvers.

The goal of the research in the area of modeling preferences is to produce theoretical machinery necessary to build a software system for finding preferred stable models of SC logic programs and to produce a prototype of such a software system.

The goal of the research in the area of computing stable models of logic programs using Metropolis type algorithms is to produce a theoretical machinery necessary to build a software system for finding stable models of logic programs using Metropolis type algorithms and to produce a prototype of such a software system.

Finally, it would be interesting to combine the research to produce a theoretical foundation and software system for solving the following problem: what is a next action that an agent acting in a dynamical system has to perform in order to maximize the total expected reward.

4 Current Status of the Research

Hybrid ASP. A paper on H-ASP [3] was accepted as a technical communication to ICLP 2011. Proof of concept software prototypes are developed and successfully tested to research the feasibility of combining H-ASP with numerical algorithms for solving PDEs and with decision making algorithms (see [22] for a review of decision making algorithms).

Preferences as SC atoms. The framework of theoretical definitions is created and various examples of the use of SC atoms to model preferences are being investigated.

Computing stable models of logic programs using Metropolis type algorithms. The subject is discussed in [2]. The paper discusses Metropolized Forward Chaining (MFC) algorithm and some computer experiments performed using the algorithm.

5 Preliminary Results Accomplished

Hybrid ASP. see section 4.

Preferences as SC atoms. Preliminary results include being able to successfully model preferences as defined in [21], as well as to model preferences in various toy examples.

Computing stable models of logic programs using Metropolis type algorithms. Preliminary results include software prototypes, both single processor and parallel versions, as well as a size 300 $(2,6)$ Van der Waerden certificate found by the software as discussed in [2]. The certificate was found using a 288 processor parallel run in under 2 weeks. To illustrate the difficulty of finding size 300 $(2,6)$ Van der Waerden certificate - a single processor version of smodels has failed to find size 210 certificate (while running for over 3 weeks), and a single processor version of clasp 1.3.3 has failed to produce size 240 certificate (while running for over 2 weeks). The experiments demonstrate that the method is feasible for the problem of finding stable models of logic programs and merits additional research. A different set of experiments was used to validate the use of MFC for the problem of finding maximal subprograms that have stable models of the programs that don't have stable models. Two Metropolis type algorithm were tested: Metropolis algorithm and Stochastic Approximation Monte Carlo (SAMC) algorithm [1], [12]. Preliminary experiments indicate that for difficult problems SAMC performs significantly better than the Metropolis algorithm.

6 Open Issues and Expected Achievements

Hybrid ASP. It is expected that the theoretical foundation necessary for computations with H-ASP will be produced as well as a software proof of concept prototype.

Preferences as SC atoms. It is expected that a theoretical foundation necessary for modeling preferences using SC atoms will be produced. It is not clear whether a software proof of concept prototype will be produced as part of the dissertation.

Computing stable models of logic programs using Metropolis type algorithms. The initial goal of this area is already mostly achieved. However research produced many open issues. Here are some of them: 1. what are the relative merits of using various Metropolis type algorithms? 2. What other approaches to using Metropolis type algorithms for finding

stable models of logic programs are there? 3. How does MFC compare in performance to the existing algorithms?

Regarding combining the 3 areas: it is not clear whether a fusion of 3 areas will be accomplished, and a proof of concept prototype will be produced as part of the dissertation.

Acknowledgments. The author is grateful to his thesis advisor Jeffrey Remmel for collaboration and guidance. The author is grateful to Adriano Garsia for helpful discussion and for providing a summer research assistantship from NSF grant DMS-0800273 in order to conduct some of the computer experiments. This research was supported in part by NSF MRI Award 0821816 and by other computing resources provided by the Center for Computational Mathematics (http://ccom.ucsd.edu/).

References

1 Y. F. Atchade, J. S. Liu. The Wang-Landau Algorithm in General State Spaces: Applications and Convergence Analysis. *Technical Report, Department of Statistics, University of Michigan*, (2007).

2 A. Brik and J.B. Remmel. Computing Stable Models of Logic Programs Asing Metropolis Type Algorithm. Preprint, ASPOCP ICLP 2011, (2011).

3 A. Brik, J.B. Remmel. Hybrid ASP. Preprint, technical communications ICLP 2011, (2011).

4 Sandeep Chintabathina. Towards Answer Set Programming Based Architectures for Intelligent Agents. *PhD thesis, Texas Tech University*, (2010).

5 P. Diaconis. The Markov Chain Monte Carlo Revolution. *Bulletin of the American Mathematical Society*, Nov. (2008).

6 M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. Conflict-Driven Answer Set Solving. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence*, (2007), 386–392.

7 M. Gelfond and V. Lifschitz. Action languages. *Electronic Transactions on AI*, 3(16), (1998).

8 W. Hastings, Monte Carlo sampling methods using Markov chains and their applications. *Biometrica*, 57 (1970), 97-109.

9 P.R. Herwig, M. J. H. Heule, P. M. van Lambalgen, and H. van Maaren. A New Method to Construct Lower Bounds for van der Waerden Numbers. *Electronic Journal of Combinatorics*, **14** (2007), #R6.

10 S. M. Jachec, O. B. Fringer, M. G. Gerritsen, R. L. Street. Numerical Simulation of Internal Tides and the Resulting Energetics within Monterey Bay and the Surrounding Area. *Geophysical Research Letters,* (2006) Vol. 33, L12605, doi: 10.1029/2006GL026314.

11 R. Kerckhoffs, M. Neal, Q. Gu, J. Bassingthwaighte, J. Omens, A. McCulloch. Coupling of a 3D Finite Element Model of Cardiac Ventricular Mechanics to Lumped Systems Models of the Systemic and Pulmonic Circulation. *Annals of Biomedical Engineering*, 35 (1), (2007), 1-18.

12 F. Liang, C. Liu, and R. J. Carroll. Stochastic Approximation in Monte Carlo Computation. *Journal of the American Statistical Association*, 102 (2007), 305-320.

13 L. Liu and M. Truszczyński. Local-search techniques in propositional logic extended with cardinality atoms. *Proceedings of the 9-th International Conference on Principles and Practice of Constraint Programming*, LNCS 2833 (2003), 495-509.

14 L. Liu and M. Truszczyński. Local search techniques for Boolean combinations of psuedo-Boolean constraints. *Proceedings of the Twentieth National Conference on Artificial Intelligence*, AAAI Press (2006), 98-103.

15 L. Liu and M. Truszczyński. Statisfiability testing of Boolean combinations of pseudo-boolean constraints using local search techniques. *Constraints*, **12**(3), (2007), 345-369.
16 N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth. A. H. Teller, and E. Teller. Equation of State Calculations by Fast Computing Machines. *Journal of Chemical Physics*, 21 (1953), 1087-1092.
17 W. Marek, A. Nerode, and J.B. Remmel. Logic Programs. Well Orderings, and Forward Chaining. *Annals of Pure and Applied Logic*, 96 (1999), 231-276.
18 W. Marek and J. B. Remmel. Set Constraint in Logic Programming. *Logic Programming and Nonmonotonic Reasoning, Proceedings of the 7th International Conference*, LNCS 2923, (2004), 154–167.
19 B. Selman, H. Levesque, D. Mitchel. A New Method for Solving Hard Satisfiability Problems. *Proceedings of the 10th National Conference on Artificial Intelligence*, (1992), 440-446.
20 P. Simons, I. Niemelä, and T. Soininen. Extending and Implementing the Stable Model Semantics. *Artificial Intelligence*, 138 (2002), 181–234.
21 T. C. Son and E. Pontelli. Planning with Preferences using Logic Programming. *TPLP*, 6 (5), (2006), 559-608.
22 R. S. Sutton, A. G. Barto. Reinforcement Learning. 1998. *The MIT Press.*

A Semiring-based framework for fair resources allocation

Paola Campli

University G.D'Annunzio - Pescara, Italy
campli@sci.unich.it

Abstract

In this paper a general framework (based on soft constraints) to model and solve the fair allocation problem is proposed. Our formal approach allows to model different allocation problems, ranging from goods and resources allocation to task and chore division. Soft constraints are employed to find a fair solution by respecting the agents's preferences; indeed these can be modeled in a natural fashion by using the Semiring-based framework for soft constraints. The fairness property is considered following an economical point of view, that is, taking into account the notions of *envy-freeness* (each player likes its allocation at least as much as those that the other players receive, so it does not envy anybody else) and *efficiency* (there is no other division better for everybody, or better for some players and not worse for the others).

1998 ACM Subject Classification F.4.1 Mathematical Logic

Keywords and phrases soft constraints fairness allocation resources

Digital Object Identifier 10.4230/LIPIcs.ICLP.2011.268

1 Introduction and problem description

The problem of "fair division", that is, fairly dividing resources or costs among a set of people, is an important issue in real life scenarios; it can refer to several situations, such as inheritance and divorce settlements, division of health resources, computer networking resources, voting power, intellectual property licenses, costs for environmental improvements, etc. In these cases, formal protocols for division are needed. Many variations of the basic problem exist, for example, the situation with divisible resources is quite different from the situation with indivisible objects; the items to be divided can be goods or sometimes "bads" like chores or other burdens; some problems can involve the division of money to compensate a "non fair share", or a payoff in exchange for performing a chore. Other aspects to consider are the number of objects with respect to the number of people. If goods are scarce, an *auction* is needed and the items are assigned to (usually) one winner; in this case fairness methods are studied in repeated auctions to guarantee that not always the same player will be the winner.

But most of the variation comes from the fact that there are many reasonable ways to formalize "fairness" including *max-min fairness*, *proportional fairness*, *envy-free* fairness, etc. which may or may not lead to the same optimal allocation; if we take into account a *global view* this means looking at the overall allocation in terms of social welfare, while a *local view* focus on the agents preferences.

In this paper we investigate on the allocation of indivisible objects which can be either goods or bads; thus, given a set of items and a set of people, each person states a weight for each object which, depending on the cases, can represent preferences or costs (such as time, money, resources etc.). According our model, the solution will be an *envy-free* allocation of objects to the agents, reminding that envy-freeness is a fairness property that guarantees

© Pola Campli;
licensed under Creative Commons License NC-ND
Technical Communications of the 27th International Conference on Logic Programming (ICLP'11).
Editors: John P. Gallagher, Michael Gelfond; pp. 268–273

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

that no agent would rather have one of the bundle allocated to any of the other agents, that is, each player prefers his bundle.

The paper is organized as follows: in Sec. 2.1 the various aspects of fair division problems are illustrated; in Sec. 2.2 we give a background on Soft Constraint Satisfaction Problems and semirings; in Sec. 4.1 we show how to use the SCSP framework to model allocation problems; in Sec. 4.2 the mapping between SCSP and allocation problems is provided; finally, Sec. 5 contains references to similar works in the field of fair division and allocation problems, a short summary and our future works.

2 Background and overview of the existing literature

In the indivisible resource area, most of the issues are based on the *Santa Claus problem* [4], in which the goal is to distribute n presents among k kids, in such a way that the least lucky kid is as happy as possible; a linear programming is used with the *Max-min fairness* objective function. Another variant is represented by the "housemate problem" [5], where goods are associated with bads. The problem consists in assigning different rooms to people sharing a house according the bid they submit, but also to determine a price to be paid by each roommate for his assigned room. Concerning Chore division, the problem of dividing an undesirable object has been investigated only for divisible goods, where cake cutting algorithms have been adapted in order to deal with chores instead of desirable goods. It is supposed that chores are infinitely divisible [3] and valuations over bundles are additive. Other works view the problem from a computational perspective and are based on approximation algorithms with the purpose of finding a solution closest to the optimum [6].

2.1 The problem of fair division

Fair division [2] is the problem of dividing one or several goods amongst two or more agents in a way that satisfies a suitable fairness criterion. Fair division has been studied in philosophy, political science, economics and mathematics for a long time, but is also relevant to computer science and multiagent systems (MAS), in which resource allocation is a central topic since the application or agents need resources to perform tasks. It is assumed that agents are autonomous. A solution needs to respect and balance their individual preferences; fairness definitions are required and once we have a well-defined fair division problem, we require an algorithm to solve it.

The elements of a Fair Division Problem are a set P of n players: $p_1, p_2, ..., p_n$ and a set of m objects O to be divided. The problem is to divide the set O into n shares ($o_1, o_2, ..., o_n$) so that each player gets a fair share of O. A *fair share* is any share that, in the opinion of the player receiving it, has a value that is at least $\frac{1}{n}$ of the total value of the set of goods O. It is crucial to understand that share value is subjective, and that each player may even have a different notion of how much the set to be divided is worth.

There are three types of fair division schemes: the *Continuous Fair Division Schemes*, in which the set O is infinitely divisible (cake, land, etc.) and shares can be adjusted by arbitrarily small amounts; the *Discrete Fair Division Schemes* where the set O is made up of indivisible objects (cars, houses, etc) and the *Mixed Fair Division Schemes*. In this paper, since we are dealing with indivisible objects, we will focus on the discrete case.

2.2 Constraint Satisfaction Problems, Semirings and Soft Constraints

The classic definition of a Constraint Satisfaction Problem (CSP) is as follows [10]. A CSP P is a triple $P = \langle X, D, C\rangle$ where X is an n-tuple of variables $X = \langle x_1, x_2, \ldots, x_n\rangle$, D is a corresponding n-tuple of domains $D = \langle D_1, D_2, \ldots, D_n\rangle$ such that x_i can assume values within a determined domain D_i, C is a t-tuple of constraints $C = \langle C_1, C_2, \ldots, C_t\rangle$. A constraint C_j is a pair $\langle Rl_{S_j}, S_j\rangle$ where Rl_{S_j} is a relation on the variables in $S_i = scope(C_i)$. A solution to the CSP P is an n-tuple $A = \langle a_1, a_2, \ldots, a_n\rangle$ where $a_i \in D_i$ and each C_j is satisfied in that Rl_{S_j} holds on the projection of A onto the scope S_j . In a given task one may be required to find the set of all solutions, *sol(P)*, to determine if that set is non-empty or just to find any solution, if one exists. If the set of solutions is empty the CSP is unsatisfiable. A *c-semiring* [8, 9] S (or simply semiring in the following) is a tuple $\langle A, +, \times, \mathbf{0}, \mathbf{1}\rangle$ where A is a set with two special elements $\mathbf{0}, \mathbf{1} \in A$ (respectively the bottom and top elements of A) and with two operations $+$ and $\times$ that satisfy certain properties: $+$ is defined over (possibly infinite) sets of elements of A and is commutative, associative and idempotent; it is closed, $\mathbf{0}$ is its unit element and $\mathbf{1}$ is its absorbing element; $\times$ is closed, associative, commutative and distributes over $+$, $\mathbf{1}$ is its unit element and $\mathbf{0}$ is its absorbing element (for the exhaustive definition, please refer to [8]). The $+$ operation defines a partial order $\leq_S$ over A such that $a \leq_S b$ iff $a + b = b$; intuitively $a \leq_S b$ if b represents a value *better* than a. Other properties related to the two operations are that $+$ and $\times$ are monotone on $\leq_S$, $\mathbf{0}$ is its min and $\mathbf{1}$ its max, $\langle A, \leq_S\rangle$ is a complete lattice and $+$ is its lub. A *soft constraint* [8, 9] may be seen as a constraint where each instantiation of its variables has an associated preference. Given $S = \langle A, +, \times, \mathbf{0}, \mathbf{1}\rangle$ and an ordered set of variables V over a finite domain D, a soft constraint is a function which, given an assignment $\eta : V \to D$ of the variables, returns a value of the semiring. Using this notation $\mathcal{C} = \eta \to A$ is the set of all possible constraints that can be built starting from S, D and V.

Given the set $\mathcal{C}$, the combination function $\otimes : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$ is defined as $(c_1 \otimes c_2)\eta = c_1\eta \times c_2\eta$ [8, 9]. The $\otimes$ builds a new constraint which associates with each tuple of domain values for such variables a semiring element which is obtained by multiplying the elements associated by the original constraints to the appropriate sub-tuples. Given a constraint $c \in \mathcal{C}$ and a variable $v \in V$, the *projection* [8, 9, 11] of c over $V - \{v\}$, written $c \Downarrow_{(V \setminus \{v\})}$ is the constraint c' such that $c'\eta = \sum_{d \in D} c\eta[v := d]$. Informally, projecting means eliminating some variables from the support. A SCSP [9] is a tuple $P = \langle X, D, C, A\rangle$ where X is a set of variables, D is the domain of the variables and C is a set of constraints over X associating values form a c-semiring A. The *best level of consistency* notion defined as $blevel(P) = Sol(P) \Downarrow_{\emptyset}$, where $Sol(P) = \bigotimes C$ [9]. A problem P is α-consistent if $blevel(P) = \alpha$ [9]; P is instead simply "consistent" iff there exists $\alpha >_S \mathbf{0}$ such that P is α-consistent [9]. P is inconsistent if it is not consistent.

3 Goal of the research

Although there is wide literature on fair division within the fields of economics, game theory, political science, mathematics, operations research and computer science, it seems to lack a unified and general framework which allows to solve the different kinds of problems, each one with different objects (desirable or undesirable items), weights or preferences. Another issue encountered in previous works is that often it is not possible to find a solution and the problem remains unsolved; our approach might be applied in all these cases because the use of soft constraints allows to always find a solution; moreover we provide a general framework which can model several cases by choosing the appropriate semiring (see Sec. 2.2).

4 Preliminary results accomplished

4.1 The SCSP framework for allocations problems

In this section we define a quantitative framework to model fair division problems, where each assignment of objects to people have an associated preference/weight and, consequently, modeling this kind of problems as Soft CSPs (see Sec. 2.2) leads to an allocation of goods to people that optimize the criteria defined by the chosen semiring. For instance, the *Weighted* semiring $\langle \mathbb{R}^+, min, \hat{+}, 0, 1\rangle$, where $\hat{+}$ is the arithmetic plus ($\mathbf{0} = \infty$ and $\mathbf{1} = 0$), can be used to model the undesirable objects case (such as chore division) by expressing the (e.g. money) cost for performing a chore; the optimum solution in this scenario corresponds to an allocation with minimum total cost. The *Fuzzy* semiring $\langle [0..1], max, min, 0, 1\rangle$ is well suited for modeling the players's preferences with respect to each good; the solution we obtain with this semiring corresponds in choosing the highest among the minimum preferences. The *Probabilistic* semiring $\langle [0..1], max, \hat{\times}, 0, 1\rangle$ can be used when preferences are unknown, thus, weights corresponds to probabilities; we can express for instance, that person p_1 prefers object o_3 with probability 0.4. The arithmetic $\hat{\times}$ is used to compose the probability values together (since we assume that preferences and thus probabilities are independent). By using the *Boolean* semiring $\langle \{true, false\}, \vee, \wedge, false, true\rangle$ we can solve the non weighted allocation problems, that is, each person states only the goods he/she desires (or the tasks he is able to perform).

4.2 Mapping Allocation Problems to SCSPs

In this section we show a mapping from the allocation problem to SCSPs. An allocation problem is formed by a set of m indivisible objects (or items) $O = \{o_1, o_2, \ldots, o_m\}$ and a set of people (or players) $P = \{p_1, \ldots, p_n\}$. Each player has their own preferences or costs regarding the allocation of goods/tasks to be selected. The problem consists in partitioning the set of objects in n subsets (or bundles) in a way that each person receives a (non-empty) bundle that satisfies a suitable fairness criterion. In order to model this problem with a SCSP, we define a variable for each object $o_i \in O$, i.e. $V = \{o_1, o_2, \ldots, o_m\}$ and the domain of each variable is the set of people in P: $D = \{p_1, \ldots, p_n\}$, meaning that an object can be assigned to a person in the set P; for example $o_1 = p_2$ means that player p_2 receives object o_1. A soft constraint associates a semiring value for each assignment of the variables in its scope, which represent the preference of the player for a given item; if no weights are considered , the corresponding variable assignment is *not admitted* or *admitted* and the values $\mathbf{0}$ or $\mathbf{1}$ of the boolean semiring set are respectively returned.

▶ **Example 1.** As a simple example, suppose we must assign 3 objects (o_1, o_2, o_3) to 2 players (p_1, p_2). The corresponding SCSP, by using (for instance) a Fuzzy semiring, has three variables: o_1, o_2 and o_3, each with the domain $D = \{p_1, p_2\}$; we define the following unary constraints: $C_{o1} := \{(p_1, 0.7); (p_2, 0.2); \}$ meaning that object 1 can be assigned either to person 1 (who has a preference of 0.7 for this objects) or person 2 (with preference 0.2); $C_{o2} := \{(p_1, 0.3); (p_2, 0.1); \}$ that is, object 2 can be assigned either to person 1 or 2 with preferences 0.3 and 0.1 respectively; $C_{o3} := \{(p_2, 0.7)\}$ meaning that object 3 can only be assigned to person 2 who desires the object with preference 0.7;

the solution is illustrated in the table below:

o1	o2	o3		$Sol(P)$
p1	p1	p1	not allowed	
p1	p1	p2	$0.7 \times 0.3 \times 0.7$	**0.3**
p1	p2	p1	not allowed	
p1	p2	p2	$0.7 \times 0.1 \times 0.7$	0.1
p2	p1	p1	not allowed	
p2	p2	p1	not allowed	
p2	p2	p2	$0.2 \times 0.1 \times 0.7$	0.1
p2	p1	p2	$0.2 \times 0.3 \times 0.7$	0.2

The (unique) optimal solution of this problem is $o_1 = p_1, o_2 = p_1 o_3 = p_2$ (with preference 0.3).

Depending on the cases, the solution provided might not be fair if we only use the previous method. For example, with different preferences, the solution (p_2, p_2, p_2) could be returned, which is certainly unfair, since player 1 does not get any object and envies player 2. For this reason, we need to specify additional constraints in order to solve the allocation problem and guarantee the *envy-freeness* property of fairness. Let x_{ij} be a boolean variable which is equal to 1 if item j is assigned to player i and 0 otherwise and let $u_i(B_i)$ be the value for person i of the set of objects (B_i) assigned to him; this value represents the valuation of the bundle (that is, the subset of items) for each person; an issue encountered in this case is that requesting an input to the agents for every possible combination of goods is NP-hard, in fact for m object there are 2^m valuations for each of the n players. In order to reduce the size of the problem, we can automatically calculate the value of the bundle, by specifying in the problem if the valuations are *additive* (thus, the value is obtained by summing the weights of the single objects in the bundle), *super-additive* (the value of the bundle is greater than the values of the single objects), *sub-additive* (the value of the bundle is lower than the values of the single objects) or *maximal* (the value of the bundle corresponds to the maximum weight among the objects in the bundle); in this way we can compute the value of the entire bundle $u_i(B_i)$; the type of valuation depends on the kind of goods; for example if the items considered are *complementary* (e.g. printers and ink cartridges) the valuation of the bundle might be super-additive, or conversely, if the goods are *substitute* (e.g. petroleum and natural gas), the valuation might be sub-additive. The defined constraints are the following: [1]

1. Each object must be assigned to at most one person $\forall j \quad \sum_i x_{ij} = 1$;
2. Each person must receive at least one item: $\forall i \quad \sum_j x_{ij} \geq 1$;
3. *Envy-freeness constraint.* Each person does not prefer the set of objects assigned to the other players: $\forall i \quad u_i(Bi) \geq u_i(Bj) \quad$ for each $j \neq i$;

Moreover, since we are assuming that the number of objects is greater (or equal) than the number of people, our solution is also efficient, as shown in [12], which proves that when $m \geq n$ envy-freeness implies efficiency.

5 Open issues and expected achievements

We investigated on the use of the semiring-based framework for soft constraints in order to model and solve the fair allocation of objects problem. According the chosen semiring, we can easily represent the different set of preferences, their combination and the various kind of objects. In the future we plan to use the framework for the Stable Marriage Problem, which can be casted in a particular fair allocation problem involving the same number of objects and people.

[1] constraints 1 and 2 are based on those used in the Santa Claus Problem's paper [4]

References

1 B. Ivona and D. Varsha, Allocating indivisible goods, SIGecom Exch., volume 5, number 3, 2005, pages 11–18, doi: http://doi.acm.org/10.1145/1120680.1120683, ACM, NY, USA
2 U. Endriss, Lecture Notes on Fair Division,2009, ILLC, University of Amsterdam
3 Elisha Peterson and Francis Edward Su, Four-Person Envy-Free Chore Division, Mathematics Magazine, 2002
4 B. Nikhil and S. Maxim, The Santa Claus problem, STOC '06: Proceedings of the thirty-eighth annual ACM symposium on Theory of computing, 2006, isbn 1-59593-134-1, pages 31–40, Seattle, WA, USA, doi: http://doi.acm.org/10.1145/1132516.1132522, NY, USA
5 R.F. Potthoff, Use of Linear Programming to Find an Envy-Free Solution Closest to the BramsKilgour Gap Solution for the Housemates Problem, Group Decision and Negotiation, vol 11, pages 405–414, 2002
6 F. Uriel, On allocations that maximize fairness, SODA '08: Proceedings of the nineteenth annual ACM-SIAM symposium on Discrete algorithms, 2008, pages 287–293, San Francisco, California, Society for Industrial and Applied Mathematics, PA, USA
7 I. P. Gent, Principles and Practice of Constraint Programming - CP 2009, 15th International Conference, CP 2009, Lisbon, Portugal, September 20-24, 2009, Proceedings, CP, Springer, Lecture Notes in Computer Science, vol: 5732, 2009, isbn: 978-3-642-04243-0, ee: http://dx.doi.org/10.1007/978-3-642-04244-7, DBLP, http://dblp.uni-trier.de
8 S. Bistarelli and U. Montanari and F. Rossi, Semiring-based Constraint Solving and Optimization, Journal of the ACM, vol 44, 2, pages 201-236, 1997
9 S. Bistarelli, Semirings for Soft Constraint Solving and Programming, Springer, LNCS, vol 2962, 2004, isbn 3-540-21181-0, DBLP, http://dblp.uni-trier.de
10 F. Rossi and P. Van Beek and T. Walsh, Handbook of Constraint Programming, 2006, isbn 0444527265, Elsevier Science Inc., NY, USA
11 S. Bistarelli and U. Montanari and F. Rossi, Soft concurrent constraint programming, ACM Trans. Comput. Logic, vol 7, num 3, 2006, issn 1529-3785, pages 563–589, doi: http://doi.acm.org/10.1145/1149114.1149118, ACM Press,NY, USA
12 A. Ahmet and D. Gabrielle and G. David, Fair Allocation of Indivisible Goods and Criteria of Justice,Econometrica, vol 59, num 4, 1991, pag 1023–1039, http://www.jstor.org/stable/2938172, ISSN 00129682, The Econometric Society, Copyright © 1991 The Econometric Society

Promoting Modular Nonmonotonic Logic Programs*

Thomas Krennwallner

Institut für Informationssysteme, Technische Universität Wien
Favoritenstraße 9-11, A-1040 Vienna, Austria
tkren@kr.tuwien.ac.at

Abstract

Modularity in Logic Programming has gained much attention over the past years. To date, many formalisms have been proposed that feature various aspects of modularity. In this paper, we present our current work on Modular Nonmonotonic Logic Programs (MLPs), which are logic programs under answer set semantics with modules that have contextualized input provided by other modules. Moreover, they allow for (mutually) recursive module calls. We pinpoint issues that are present in such cyclic module systems and highlight how MLPs addresses them.

1998 ACM Subject Classification I.2.3 Deduction and Theorem Proving

Keywords and phrases Knowledge Representation, Nonmonotonic Reasoning, Modular Logic Programming, Answer Set Programming

Digital Object Identifier 10.4230/LIPIcs.ICLP.2011.274

1 Introduction and Problem Description

Answer set programming (ASP) is an approach for declarative problem solving geared towards search problems. More specifically, problems are represented by nonmonotonic logic programs, such that the *stable models* (or *answer sets*) [19] of the program represent the solutions to a given problem instance. ASP has many applications in knowledge representation and problems in artificial intelligence including planning, diagnosis, and configuration.

A natural way to design software for solving problems is to identify easier to handle subproblems that can be solved independently from each other, and then based on this analysis to craft corresponding software components that solve the subproblems: the modules. The combination of these components then gives an implementation for the whole problem. Most general-purpose programming languages have their own way to introduce modularity, a key concept that helps developing software artifacts. Techniques like information hiding, abstraction, and structured programming are well-established principles for breaking down sub-tasks in an imperative program, and essentially any standard programming language has amenities that allow to define input/output interfaces to modules for easy code-reuse in implementations of possibly unrelated problems. Testing software greatly benefits from structured programs, since it involves defining well-suited interfaces to the components, which in turn assists writing testcases. When many programmers are working on a project, the strict component-wise building of software is the only way to success. In contrast, it is customary to view logic programs as monolithic entities, i.e., one program is tailored to solve a particular problem without a clear separation of the sub-tasks, albeit the same principle of

* This research has been supported by the Austrian Science Fund project P20841, by the Vienna Science and Technology Fund project ICT 08-020, and by the EC project OntoRule (IST-2009-231875).

© Thomas Krennwallner;
licensed under Creative Commons License NC-ND
Technical Communications of the 27th International Conference on Logic Programming (ICLP'11).
Editors: John P. Gallagher, Michael Gelfond; pp. 274–279

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

creating manageable pieces will help users of logic programming systems building knowledge bases. Having an explicit way to modularize knowledge in logic programs is thus needed and adding modularity principles to ASP has several advantages like easy knowledge base reuse by clean input/output interfaces and helping to model complex problem domains by focusing on smaller parts first. This issue has been identified and various notions for modularizing logic programs have been proposed to support testing logic programs, reusing and abstracting components, and maintaining program code.

However, there are obstacles that impede to bring such characteristics to ASP. Traditional answer set semantics has no module concept and there is no straightforward way that would allow that. It is not clear how a semantics should be defined that caters for modules, as the declarative nature of ASP does not distinguish between knowledge stored in different logic programs (when viewed as modules). Another issue is to allow for cyclic module systems, i.e., when modules mutually refer to each other. Modules that have such cyclic dependencies may bring in semantic issues like unfounded models that would not be present when viewing logic programs as single unit. Both of these problems are related to the declarative nature of ASP, and any prospective model-theoretic semantics for modular ASP has to deal with unwanted semantic deficits. Methods that bring modularity aspects closer to ASP have not yet stood the test of time, and no single semantics has gained general acceptance.

The aim of this paper is to recall existing approaches in modular logic programming and to present work and results on a novel formalism to modular ASP: Modular Nonmonotonic Logic Programs (MLPs) [10]. We pinpoint peculiar issues that exist in modular frameworks for ASP and highlight how the MLP formalism addresses them. We conclude with prospective future work and open research issues.

2 Background and Overview of Existing Literature

There is a long history of research in investigating modularity principles in logic programming. A good overview provides [5, 7], which studies modularity in the context of traditional definite Horn logic programming. In general, they identify two directions for investigating modularity aspects in logic programming: (i) *Programming-in-the-large*, which introduce compositional operators to combine separate and independent modules; and (ii) *Programming-in-the-small*, which builds upon abstraction and scoping mechanisms. Early influential work on modularity in logic programming include [17] and [18], where the former can be seen as an approach for (ii), while the latter is a prototypical instance of (i).

In the context of answer set semantics, whose focus lies in the treatment of negation-as-failure and disjunctive rules, several important proposals have been put forward. Representatives for (i) are DLP-functions [21] and modular SMODELS programs [25], which has recently been generalized to a module-based framework for multi-language constraint modeling [22], and to modular P-log programs [9] that combines probabilistic reasoning with logic programs. Another proponent [28] is concerned with operator splitting similar in the vein of splitting sets [24]. Exponents in (ii) are modular logic programs with generalized quantifiers [15], macros [3], templates [8], and web rule bases [1]. On a broader scale, multi-agent scenarios with logic programs has been studied in social logic programs [6] and communicating ASP [4].

3 Goal of the Research

As described above, several semantics exist that deal with modularity in ASP. Virtually all semantics are defined such that mutual recursion between modules is disallowed. While

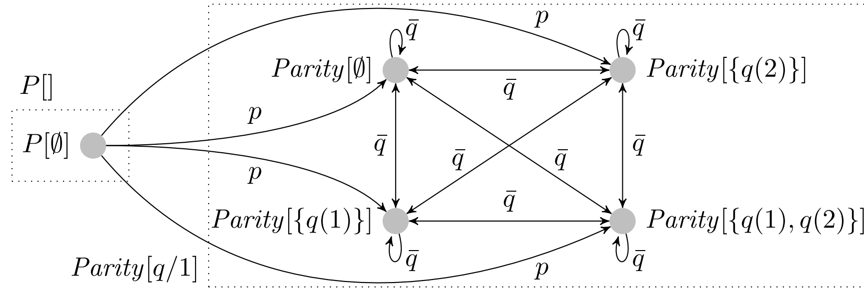

Figure 1 Call graph of instantiated modules in Example 1

this helps to simplify the definitions of a semantics for modular ASP, in general this may bring issues when different, possibly independently developed modules are combined. Many natural problems exist that have an inherent cyclic flavor, and ruling out the chance to model problems using modules that depend on each other may be too restrictive in practice, or even force to use unintuitive encodings. We aim at defining a model-theoretic semantics that caters for this situation, investigate its semantic properties and computational complexity, and develop novel evaluation algorithms for such modular nonmonotonic logic programs. The next example illustrates cycles in modular logic programming using Modular Nonmonotonic Logic Programs (MLP) as defined in [10], a formalism that admits arbitrary non-ground disjunctive nonmonotonic logic programs as modules. MLPs can be seen as a proponent of the programming-in-the-small approach to modular programming, as it is using module atoms as a language construct to access knowledge encoded in other modules. We sketch the basic building blocks of MLPs and refer to [10] for proper formal definitions.

▶ **Example 1.** Consider the following recursive module $Parity[q/1]$ consisting of four rules, which determines whether a set has an even respectively odd number of elements:

$$\bar{q}(X) \vee \bar{q}(Y) \leftarrow q(X), q(Y), X \neq Y \qquad odd \leftarrow skip(X), Parity[\bar{q}].even$$
$$skip(X) \leftarrow q(X), not\ \bar{q}(X) \qquad even \leftarrow not\ odd$$

Here, $q/1$ is a (formal) unary input predicate that stores the set. The first two rules on the left have the effect, by stability of answer sets, that q becomes $\bar{q}$ with one element randomly removed (for which *skip* is true, as defined in the lower left rule). The third rule top right determines recursively whether q stores an odd number of elements using the *module atom* $Parity[\bar{q}].even$, while the last rule bottom right defines *even* as the complement of *odd*. Intuitively, if we call the module *Parity* with a predicate p for input, then *even* is computed true, which is expressed by $Parity[p].even$, whenever p stores an even number of elements. Note that *Parity* is recursive, and for empty input p it calls itself with the same input.

We demonstrate the use of *Parity* in an MLP with the (main) module $P[]$ with empty input, which calls *Parity* with a set p of two elements:

$$p(1) \leftarrow \qquad p(2) \leftarrow \qquad pev \leftarrow Parity[p].even$$

The combination of both modules gives the cyclic MLP $\mathbf{P} = (P[], Parity[q/1])$. On the surface, $\mathbf{P}$ can be seen as an "uninstantiated" modular program, whose semantics is given by characterizing models at modules which have been instantiated with a set of input facts: the *value calls*. Figure 1 depicts the *call graph* (the principle dependencies) of $\mathbf{P}$ with value calls as nodes and edges labeled with input predicates; e.g., value call $P[\emptyset]$ calls $Parity[\{q(1), q(2)\}]$

on input p. The dotted boxes highlight the modules from which the value calls on the inside have been generated. Loosely speaking, MLPs encode schematic dependencies between modules, and instantiated modules then can be used to define a semantics that takes module input into account which is defined over possibly cyclic modules. Different interpretations of an MLP select different subgraphs of its call graph, and answer sets are defined based on the selected subgraphs. For instance, **P** has two answer sets in which *pev* is true at the main instantiation $P[\emptyset]$ and *even* is true at $\mathit{Parity}[\{q(1), q(2)\}]$ and $\mathit{Parity}[\emptyset]$, whereas *odd* is satisfied at $\mathit{Parity}[\{q(1)\}]$ and $\mathit{Parity}[\{q(2)\}]$. Both answer sets are symmetric on the guess of $\bar{q}$ at $\mathit{Parity}[\{q(1), q(2)\}]$, but otherwise equal.

4 Current Status

We have an advanced understanding of peculiar issues that arise when we allow for module cycles in MLPs. One key aspect is the use of the FLP-reduct [16] instead of the traditional GL-reduct [19] to cure semantic issues when dealing with negation-as-failure over potential nonmonotonic module atoms. Roughly, given an interpretation of a program, the GL-reduct first removes each rule whose negative body is false in the interpretation, and then cut offs the negative literals from remaining rules. On the other hand, the FLP-reduct just removes rules whose body is unsatisfied in a given interpretation, which leaves negative literals in the result of this translation. Applied to traditional answer set programs, both reducts are equivalent, but FLP-semantics is beneficial for language extensions of ASP such as logic programs with aggregates. In the context of MLPs, the FLP-semantics guarantees that models are minimal, thus we retain groundedness of the semantics and prohibit unfounded answer sets. Another aspect of MLP is to contextualize module instantiation. Here, relevant instantiations are a concept to concentrate on the important part of all instantiated modules. In general, module instantiation plays a key role for the definition of a semantics for MLPs. Akin to the call semantics of imperative programming languages, the module instantiation employed in MLPs can be seen as *call-by-value* mechanism, where module instantiation calls other instantiations with explicit input facts. This is in contrast to the module framework of DLP-functions [21], which can be classified as *call-by-reference* mechanism; input here is given implicitly by the models of each module.

Further results show that MLPs have an increase in computational complexity compared to standard ASP: propositional Horn-MLPs with unrestricted cyclic input over modules are EXP-complete, and non-ground ones are 2EXP-complete. If we restrict propositional MLPs such that modules have no input predicates, we obtain for instance that checking satisfiability of normal propositional MLPs is NP-complete, and for disjunctive MLP it is Σ_2^p-complete. In general, checking answer set existence of arbitrary normal non-ground MLPs is 2NEXP-complete, and $2\mathrm{NEXP}^{\mathrm{NP}}$-complete for the disjunctive case.

5 Preliminary Results

The work in [10] devised a novel semantics for MLPs that allows for mutual recursion between modules. We have studied the semantic properties of MLPs, their computational complexity, and compared it to DLP-functions [21]; interestingly, DLP-functions can be seen as MLPs that have no module input parameters. MLPs conservatively extend ordinary logic programs, and many semantic properties of answer set programs generalize to MLPs. For instance, the important property that every answer set of an MLP is a minimal model implies that answer sets in the MLP setting are grounded (see discussion above).

In [13], we investigated the relationships between various semantics for modular logic programs and other nonmonotonic formalisms. We have provided a more systematic view of approaches in combining nonmonotonic knowledge bases and classified formalisms based on the program reduct and on the environment view, i.e., whether their semantics is defined in terms of local models for each individual knowledge base that implicitly converge to a semantics for the combined system, or whether the formalism has a global state using a collection of explicitly accessible local models.

We developed a novel evaluation algorithm for MLPs in [11]. Here, we concentrated on an MLP fragment called input- and call-stratified MLPs, whose stratification can be evaluated in a top-down fashion starting from uninstantiated modules. This way we could generalize the splitting sets technique to MLPs and develop an evaluation algorithm that traverses the call graph and instantiates modules on-the-fly. Example 1 above is input-call-stratified, and the techniques developed in [11] are applicable to it.

We worked on two characterizations of MLPs in terms of classical models by investigating the notions of loop formulas [23] and ordered completion [2] in MLPs [12]. The results include (i) *modular loop formulas* based on loops over module instantiations, and (ii) *ordered completion for MLPs* without using explicit loop formulas. We generalized Clark's completion and positive dependency graph to MLPs with respect to different module instantiations. Based on these results, we defined modular loop formulas that capture MLP semantics. The second contribution was to explore ordered completion in the realm of MLPs. Here, fresh predicates ensures a derivation order, and program completion is only active for those predicates that do not participate in a positive loop, possibly involving module instantiations.

6 Open Issues

Future work includes to find further useful fragments of MLPs and characterize their computational complexity. Based on first results on loop formulas and ordered completion for MLPs [12] we seek to develop new algorithms that interweave conflict-driven model building with module instantiation. Related to this is to investigate first-order theorem proving techniques in the context of MLPs. Another line of research is to improve the understanding of MLP semantics and give it a logical foundation using (generalized) equilibrium logic [26] and applying results on FLP-semantics in [27]. Furthermore, we want to relax the restriction to minimal models in non-relevant instantiations and use semi-equilibrium models [14] instead. As a prospective application we want to investigate dl-programs with Datalog-rewritable Description Logics [20]. Intuitively, the Description Logic knowledge base can be rewritten to a module, and dl-atoms that appear in the logic program of the dl-program can be rewritten as module atoms that refer to this module. Moreover, we are currently developing a prototype implementation to evaluate input-call stratified MLPs.

Acknowledgments. I would like to thank my supervisor Prof. Dr. Thomas Eiter and Dr. Michael Fink for their ongoing support, as well as my fellow co-author Minh Dao-Tran.

References

1. A. Analyti, G. Antoniou, and C. V. Damásio. MWeb: a principled framework for modular web rule bases and its semantics. *ACM Trans. Comput. Logic*, 12(2):17:1–17:46, 2011.
2. V. Asuncion, F. Lin, Y. Zhang, and Y. Zhou. Ordered completion for first-order logic programs on finite structures. In *AAAI'10*, pp. 249–254. AAAI Press, 2010.
3. C. Baral, J. Dzifcak, and H. Takahashi. Macros, macro calls and use of ensembles in modular answer set programming. In *ICLP'06*, pp. 376–390. Springer, 2006.

4 K. Bauters, S. Schockaert, D. Vermeir, and M. De Cock. Communicating ASP and the polynomial hierarchy. In *LPNMR'11*, pp. 67–79. Springer, 2011.

5 A. Brogi, P. Mancarella, D. Pedreschi, and F. Turini. Modular logic programming. *ACM Trans. Prog. Lang. Syst.*, 16(4):1361–1398, 1994.

6 F. Buccafurri and G. Caminiti. Logic programming with social features. *Theo. Pract. Logic Progr.*, 8(5-6):643–690, 2008.

7 M. Bugliesi, E. Lamma, and P. Mello. Modularity in logic programming. *J. Logic Prog.*, 19/20:443–502, 1994.

8 F. Calimeri and G. Ianni. Template programs for disjunctive logic programming: An operational semantics. *AI Comm.*, 19(3):193–206, 2006.

9 C. V. Damásio and J. Moura. Modularity of P-Log programs. In *LPNMR'11*, pp. 13–25. Springer, 2011.

10 M. Dao-Tran, T. Eiter, M. Fink, and T. Krennwallner. Modular nonmonotonic logic programming revisited. In *ICLP'09*, pp. 145–159. Springer, 2009.

11 M. Dao-Tran, T. Eiter, M. Fink, and T. Krennwallner. Relevance-driven evaluation of modular nonmonotonic logic programs. In *LPNMR'09*, pp. 87–100. Springer, 2009.

12 M. Dao-Tran, T. Eiter, M. Fink, and T. Krennwallner. First-order encodings of modular nonmonotonic logic programs. In *Datalog 2.0*. Springer, 2011. `http://datalog20.org/`, to appear.

13 T. Eiter, G. Brewka, M. Dao-Tran, M. Fink, G. Ianni, and T. Krennwallner. Combining nonmonotonic knowledge bases with external sources. In *FroCos'09*, pp. 18–42. Springer, 2009.

14 T. Eiter, M. Fink, and J. Moura. Paracoherent answer set programming. In *KR'10*, pp. 486–496. AAAI Press, 2010.

15 T. Eiter, G. Gottlob, and H. Veith. Modular logic programming and generalized quantifiers. In *LPNMR'97*, pp. 290–309. Springer, 1997.

16 W. Faber, N. Leone, and G. Pfeifer. Semantics and complexity of recursive aggregates in answer set programming. *Artif. Intell.*, 175(1):278–298, 2011.

17 M. Fitting. Enumeration operators and modular logic programming. *J. Logic Prog.*, 4(1):11–21, 1987.

18 H. Gaifman and E. Shapiro. Fully abstract compositional semantics for logic programs. In *POPL'89*, pp. 134–142, ACM, 1989.

19 M. Gelfond and V. Lifschitz. Classical negation in logic programs and deductive databases. *New Generat. Comput.*, 9:365–385, 1991.

20 S. Heymans, T. Eiter, and G. Xiao. Tractable reasoning with dl-programs over datalog-rewritable description logics. In *ECAI'10*, pp. 35–40. IOS Press, 2010.

21 T. Janhunen, E. Oikarinen, H. Tompits, and S. Woltran. Modularity aspects of disjunctive stable models. *J. Artif. Intell. Res.*, 35:813–857, 2009.

22 M. Järvisalo, E. Oikarinen, T. Janhunen, and I. Niemelä. A module-based framework for multi-language constraint modeling. In *LPNMR'09*, pp. 155–168. Springer, 2009.

23 F. Lin and Y. Zhao. ASSAT: computing answer sets of a logic program by SAT solvers. *Artif. Intell.*, 157(1-2):115–137, 2004.

24 V. Lifschitz and H. Turner. Splitting a logic program. In *ICLP'94*, pp. 23–37. MIT, 1994.

25 E. Oikarinen and T. Janhunen. Achieving compositionality of the stable model semantics for smodels programs. *Theo. Pract. Logic Prog.*, 8(5–6):717–761, 2008.

26 D. Pearce. A new logical characterisation of stable models and answer sets. In *Non-Monotonic Extensions of Logic Programming*, pp. 57–70. Springer, 1997.

27 M. Truszczynski. Reducts of propositional theories, satisfiability relations, and generalizations of semantics of logic programs. *Artif. Intell.*, 174(16–17):1285–1306, 2010.

28 J. Vennekens, D. Gilis, and M. Denecker. Splitting an operator: algebraic modularity results for logics with fixpoint semantics. *ACM Trans. Comput. Logic*, 7(4):765–802, 2006.

Correct Reasoning about Logic Programs

Jael Kriener

School of Computing, University of Kent, CT2 7NF, UK

Abstract

In this PhD project, we present an approach to the problem of determinacy inference in logic programs with cut, which treats cut uniformly and contextually. The overall aim is to develop a theoretical analysis, abstract it to a suitable domain and prove both the concrete analysis and the abstraction correct in a formal theorem prover (Coq). A crucial advantage of this approach, besides the guarantee of correctness, is the possibility of automatically extracting an implementation of the analysis.

1998 ACM Subject Classification D.1.6 Logic Programming, D.2.4 Software/Program Verification, F.3.1Specifying and Verifying and Reasoning about Programs, F.3.2 Semantics of Programming Languages

Keywords and phrases Prolog, cut, determinacy inference, abstract interpretation, denotational semantics, automated theorem proving, Coq

Digital Object Identifier 10.4230/LIPIcs.ICLP.2011.280

1 Introduction and problem description

The focus of my work lies on provably correct static analysis of logic programs, in particular on determinacy inference for logic programs containing *cut*, and on verified implementations thereof.

For reasons which hardly need spelling out here, the question of whether a goal is deterministic or not is central in logic programming. In practice, logic programmers often use in-build control mechanisms, like the *cut* in Prolog, to ensure determinacy of certain goals. Yet, todate methods for inferring determinacy conditions on goals have not addressed this close connection between the *cut*s in a program and the determinacy of its goals. One of the problems I address in my PhD work is to apply well-known techniques in program analysis (abstract interpretation) to develop and prove correct a method for determinacy inference that takes account of this.

Furthermore, I would like to build on the work of Cachera, Pichardie and others ([3], [4], etc.), who observe that "[i]n spite of the nice mathematical theory of program analysis and the solid algorithmic techniques available one problematic issue persists, *viz.*, the *gap* between the analysis that is proved correct on paper and the analyser that actually runs on the machine" [3]. Thanks to advances in theorem proving, and in particular to the Coq system, this gap can be bridged. Coq provides a framework in which the development of a well-defined determinacy inferrence, its correctness proof with respect to an underlying program semantics and its implementation can be part of one integrated process. I would like to use Coq to prove correct the determinacy analysis discussed above and to obtain a verified implementation of it.

Technical Communications of the 27th International Conference on Logic Programming (ICLP'11).
Editors: John P. Gallagher, Michael Gelfond; pp. 280–283

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

2 Background and overview of the existing literature

Logic Programming

[11], [13], [15] are standard works on logic programming and as such have been most helpful so far and will in all likelihood continue to be so.

Determinacy Inference in Logic Programs

[5] present a method for inferring determinacy information from a program by adding constraints to the clauses of a predicate which allow the inference of mutual exclusion conditions between these clauses rather than determinacy conditions for a whole predicate. [14] presents a method for determinacy analysis, based on a partial evaluation technique for full Prolog which detects whether there are none, one or more than one ways a goal can succeed. [10] present a top-down framework for abstract interpretation of Prolog which is based on sequences of substitutions and can be instantiated to derive an analysis equivalent to that of [14]. Finally, my own supervisor has worked on the problem of determinacy inference before (see [6], [7], [12]). These works form the starting point of my own research.

Coq and Automated Abstract Interpretaion

I have spent less time aquiring background in on Coq and Automated Abstract Interpretation. [1] is a standard reference for Coq and should provide a good starting point. David Cachera, David Pichardie and others have published on abstract interpretation in Coq (see [3], [4]).

3 Goal of the research

My goal is to develop a determinacy analysis for logic programs including *cut*, that is as tight as possible, to prove it correct using a framework for automated theorem proving and to obtain a verified implementation of the same.

4 Current status of the research

During the first six months of my PhD work I have focused on the following:

- Acquiring the necessary background and techniques in program analysis, in particular in abstract interpretation.
- Researching previous work on determinacy in logic programs and on formal semantics for logic programs including *cut*.
- Applying the understanding of these two areas to develop and prove correct a determinacy inference for logic programs including *cut*.

This work has led to a paper submitted to ICLP 2011 (see Section 5 below). There are some issues arising from this work, that will need to be addressed, before I can move to the next stage in my overall project (see Section 6 below).

5 Preliminary results accomplished

In collaboration with my supervisor Dr Andy King, I have written a paper presenting and manually proving correct a method for inferring determinacy conditions for Prolog with *cut*

which has been accepted to ICLP 2011 and for publication in a special issue of the journal "Theory and Practice of Logic Programming" [9, 8].

In the process of implementing this method, I found one difficulty in computational elimination of existential quantifiers in constraint systems. In addressing this problem I have developed, in collaboration with Jörg Brauer and my supervisor, a method for reducing existential quantifier elimination to incremental SAT, a paper on which has just been accepted for publication in the conference Computed Aided Verification (see [2]).

6 Open issues and expected achievements

In the short term, I am addressing a further difficulty arising from the manual implementation of the determinacy inference, other than the issue of existential quantifier elimination mentioned in the last section, namely how to efficiently compute a mutual exclusion condition for two sets of constraints. I am reasonably confident that this problem can be addressed in a similar fashion by reformulating it as an incremental SAT problem.

In the long term, to achieve the goals outlined in the previous sections, I plan to reformulate and implement the determinacy inferrence mentioned above and its underlying semantics for Prolog with *cut* in the Coq system and mechanize my manual correctness prove. For this, I will need to gain considerable background in automated theorem proving.

References

1 Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science. Springer Verlag, 2004.

2 J. Brauer, A. King, and J. Kriener. Existential Quantification as Incremental SAT. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Twenty-third International Conference on Computer Aided Verification*, Lecture Notes in Computer Science. Springer-Verlag, July 2011. To appear.

3 David Cachera, Thomas P. Jensen, David Pichardie, and Vlad Rusu. Extracting a Data Flow Analyser in Constructive Logic. *Theor. Comput. Sci.*, 342(1):56–78, 2005.

4 David Cachera and David Pichardie. A Certified Denotational Abstract Interpreter. In Matt Kaufmann and Lawrence C. Paulson, editors, *ITP*, volume 6172 of *Lecture Notes in Computer Science*, pages 9–24. Springer, 2010.

5 S. Dawson, C. R. Ramakrishnan, I. V. Ramakrishnan, and R. C. Sekar. Extracting Determinacy in Logic Programs. In *Proceedings of the Tenth International Conference on Logic Programming*, pages 424–438. MIT Press, 1993.

6 S. Genaim and A. King. Inferring Non-Suspension Conditions for Logic Programs with Dynamic Scheduling. *ACM Transactions on Computational Logic*, 9(3), November 2008.

7 A. King, L. Lu, and S. Genaim. Detecting Determinacy in Prolog Programs. In *Proceedings of the Twenty-second International Conference on Logic Programming*, volume 4079 of *Lecture Notes in Computer Science*, pages 132–147. Springer, 2006.

8 Jael Kriener and Andy King. Appendix for RedAlert: Determinacy Inference for Prolog. Technical Report 1-11, School of Computing, University of Kent, CT2 7NF, UK, 2011. Available from: `http://www.cs.kent.ac.uk/pubs/2011/3107`.

9 Jael Kriener and Andy King. RedAlert: Determinacy Inference for Prolog. *Theory and Practice of Logic Programming*, July 2011. To appear.

10 B. Le Charlier, S. Rossi, and P. Van Hentenryck. An Abstract Interpretation Framework which Accurately Handles Prolog Search-Rule and the Cut. In *Symposium on Logic Programming*, pages 157–171. MIT Press, 1994.

11 John W. Lloyd. *Foundations of Logic Programming, 2nd Edition*. Springer, 1987.

12 L. Lu and A. King. Determinacy Inference for Logic Programs. In Shmuel Sagiv, editor, *Fourteenth European Symposium on Programming*, volume 3444 of *Lecture Notes in Computer Science*, pages 108–123. Springer, 2005.

13 Richard A. O'Keefe. *The Craft of Prolog*. MIT Press, Cambridge, MA, USA, 1990.

14 D. Sahlin. Determinacy Analysis for Full Prolog. In *Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 23–30. ACM, 1991.

15 Leon Sterling and Ehud Y. Shapiro. *The Art of Prolog - Advanced Programming Techniques, 2nd Ed*. MIT Press, 1994.

Accepting the natural order of rules in a logic program with preferences

Alexander Šimko

Department of Applied Informatics
Faculty of Mathematics, Physics and Informatics
Comenius University
Mlynská dolina, 824 48 Bratislava, Slovakia
simko@fmph.uniba.sk

Abstract

Preference is a natural part of common sense reasoning. It allows us to select preferred conclusions from broader range of alternative conclusions. It is typically specified on parts of conclusions or on rules. Different semantics have been proposed that deal with preference on rules. None fully meets our requirements.

We are interested in a descriptive approach to preference handling in logic programs under answer set semantics that always selects preferred answer set when standard one exists. Existing semantics that meet this criterion also give non intuitive conclusions on some programs. We think this kind of problem is related to the problem of not accepting natural order of rules induced by underlying answer set semantics.

Our goal is to define semantics that would always select preferred answer set when standard one exists, accept natural order on rules, and satisfy principles for preference handling.

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

Keywords and phrases non-monotonic reasoning, knowledge representation, answer set semantics, preference handling, preferred answer set

Digital Object Identifier 10.4230/LIPIcs.ICLP.2011.284

1 Introduction and problem description

In common sense reasoning some form of preference is usually used. One can prefer some conclusion over another, e.g., doctors and patients tend to prefer non invasive procedures over invasive ones. Preferences are also used to solve conflicts among rules. Having two applicable rules with contradictory effects we want to apply only preferred one.

In the last two decades logic programming has emerged as a favourite framework for knowledge representation. Especially answer set semantics of logic programming is widely used. Logic program consists of rules of the form "If a is true, and there is no evidence that b is true then also c is true". In such a program, answer set semantics gives us answer sets – sets of alternative conclusions. Existence of multiple answer sets is due to the use of default negation.

Natural questions arise: **1.** How to encode preference in logic program? **2.** How to extend answer set semantics in presence of preference?

These two questions have already been explored in literature. In next section we give an overview of existing approaches and identify places for improvement. We mainly focus on approaches that deal with preference on rules.

© Alexander Šimko;
licensed under Creative Commons License NC-ND
Technical Communications of the 27th International Conference on Logic Programming (ICLP'11).
Editors: John P. Gallagher, Michael Gelfond; pp. 284–289

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

2 Background and overview of the existing literature

As mentioned earlier, one can consider preference on literals (e.g., [7], [2], [6]). Another option is to consider preference on rules (e.g., [1], [3], [14], [12]). [7] also provides a way for handling preferences on rules – via transformation to preference on literals.

Meaning of a logic program without preferences is a set of answer sets. Similarly, meaning of a logic program with preferences is a set of preferred answer sets. Difference between an answer set and a preferred answer set is only that in later one preferences are considered.

In a selective approach, we select some standard answer sets to be preferred answer sets [8]. We only pick from existing answer sets and do not generate new ones. In a non selective approach a preferred answer set does not have to be a standard answer set. We see such a thing as a step outside of the answer set semantics. For the purpose of this summary we restrict our focus only to selective approaches.

Approaches that deal with preference on rules are traditionally divided into two groups: prescriptive and descriptive. Prescriptive approaches (e.g., [1], [3], [14]) view preference on rules as an order in which rules are to be applied in process of generating answer set. Descriptive approaches (e.g., [15], [7], [10], [12]) do not follow this view. Instead, they see preferences as a wish list one tries to satisfy [4].

Principles for preference handling relate a preference on rules with a preference on answer sets. First two principles were proposed in [1]. [1] also considers additional condition for preference handling: If a program has a standard answer set then it also has a preferred one. If we assume this condition the second principle is violated [1]. This additional condition is not satisfied in semantics of [1], [3], [14]. [1] also propose a relaxation of their approach, that always yields a preferred answer set when a standard one exists. But this approach satisfies none of the principles from [1].

A problem with existence of a preferred answer set is related to the view prescriptive approaches adapt. They see preference as an order in which rules are to be applied. But the answer set semantics already induces an order on rules. It is well known that stratified program induces the natural order in which rules must to be applied [9]. But the order specified by preference on rules can be in conflict with the natural order. In such case, prescriptive approaches can have difficulties to select a preferred answer sets.

[7] deals with a preference on literals. It also provides a way one can transfer a preference on rules to a preference on literals. Use of this transformation leads to the comparison of generating sets of answer sets. But the transformation is unable to track blocking between rules. If head of a rule r_1 is default negated in the body of a rule r_2 we say that r_1 blocks r_2. As shown in [11] concept of blocking is important when we compare generating sets. Preference alone is not sufficient.

[10] uses preference on rules to select "best" extended answer set. Selection of a preferred extended answer set is done by comparing program reducts – set of rules. When the comparison is made, only preference on rules is used . This semantics does not satisfy the first principle from [1]. It is also on the edge of our focus as it introduces preference in a modified semantics (extended answer set semantics).

[15] always selects a preferred answer set when a standard one exists [1]. On the other hand it gives some results that we consider counterintuitive:

▶ **Example 1.** Consider the program from Example 6 from [15].

$r_1 : p \leftarrow not\ q, not\ r$
$r_2 : q \leftarrow not\ p$
$r_3 : r \leftarrow not\ p$ $\qquad$ r_1 is preferred over r_2

It has two answer sets, $A_1 = \{p\}$ and $A_2 = \{q, r\}$, generated by $R_1 = \{r_1\}$ and $R_2 = \{r_2, r_3\}$, respectively. According to [15] both A_1 and A_2 are preferred.

We argue that only A_1 should be preferred. $r_1 \in R_1$ blocks both $r_2, r_3 \in R_2$. Also every $r \in R_2$ blocks r_1. But r_1 is preferred over r_2 and on its own generates A_1. R_2 contains one less preferred rule and no preferred one.

3 Goal of research

We also do not adopt the view of prescriptive approaches. It is well known that a stratified program induces a natural order on rules [9]. It tells us which rule must be applied before another. Moreover, every non stratified program is transformed to stratified one during the Gelfond-Lifschitz transformation – the guess phase of the answer set semantics. Rules that are filtered out in this transformation are not applicable in an answer set candidate. Note that rules, which pass this transformation for a given answer set, form a set of generating rules.

We understand this in the following way. There is no need to consider order on rules from the same generating set of an answer set. Such order is already defined. We know that every rule from generating set is applicable, and none of default negated literals in the body of rule is derivable (in a corresponding answer set). A rule can only be applied after we have derived its prerequisites. Hence, rules deriving prerequisites must be applied first. On the other hand, rules not being part of any generating set will never be applied. “Order in which they will be applied” has no meaning. Similarly, order of application between rules from different generating sets has no meaning. When we are generating answer set, we work with one generating set.

There is also another view. In a selective approach, a set of preferred answer sets is a subset of a set of standard answer sets. When we accept the principle that there must be a preferred answer set when a standard one exists and a program has only one answer set, it clearly must be a preferred one. In such a program there is no need to consider preference on rules since there is only one candidate we can choose from. Preference turns to be interesting if we have rules that produce alternative conclusions – multiple answer sets.

In accordance with this we see another interpretation of preference. We understand preference “r_2 is preferred over r_1” as follows. When rules r_1 and r_2 lead to alternative conclusions (answer sets), prefer one that uses rule r_2. But when rules participate on same conclusion or one of them is inapplicable at all (in every answer set), preference does not matter.

Condition “when rules lead to alternative conclusions” fits well into the concept of generating sets. Rules from the same generating set produce the same answer set. Different answer sets are represented by different generating sets. Same answer sets can also be represented by different generating sets. It does not cause any complication. In fact, it enables us to reason about alternative derivations. That is why we understand selection of preferred answer sets as a comparison of generating sets.

As shown in [11], preference alone is not sufficient to compare generating sets. Blocking between rules is important to determine which preference is more important.

Goal of our current research is to develop a descriptive approach to preference handling in logic programming that would: **1.** handle preference on rules, **2.** be selective, **3.** always give a preferred answer set when a standard one exists, **4.** accept natural order in which rules must to be applied, **5.** satisfy principles for preference handling (first principle from [1], fourth principle from [12], and sixth principle from [13]).

In a long term we also plan to provide an implementation and a detailed comparison to existing approaches.

4 Preliminary results accomplished

In our previous work [12], we have tried to refine approach from [11]. We have proposed an approach to preference handling that always selects a preferred answer set. It is inspired by some form of argumentation. Rules are seen as an argumentation structures. The key point is not to consider all preferences but only those among blocking rules. Such preferences are called attacks. Next, there are nine rules to combine argumentation structures into answer sets and to derive attacks on argumentation structures. Roughly speaking, preferred answer sets are then defined in terms of attacks on answer sets. We have also proposed new principles to preference handling (based on notion of attack) that enable us to correctly solve problematic examples from literature. We submitted this work to ICLP 2011. Detailed description of this approach is technically complicated and it is beyond the scope of this paper.

In order to be able to provide implementation for [12] we needed to simplify technical aspects of our approach. We have also realized that our argumentation structures represent subsets of generating rules of an answer set. We have focused on translating attacks on argumentation structures to attacks on generating sets. In [13], we have proposed a descriptive approach to preference handling that is based on the concept of attacks on generating sets. We did not try to propose equivalent approach to the one in [12]. We have proposed similar approach that tries to be compatible, satisfies principles from [12], and is based on the same understanding of preference on rules. The main idea of our approach is that a generating set being under attack cannot generate a preferred answer set. We have also proposed a new principle that expresses our understanding of preference. Preference on rules that do not generate any answer set should not matter. We have submitted [13] to ŠVK 2011, a student science conference at our university. We have also presented it there and applied it for publication in conference proceeding.

Both our approaches share a common drawback. The way they handle mutual attacks of argumentation structures/generating sets is technically oriented. It lacks intuitive interpretation and ignores a natural order of rules in logic program.

Work on approaches [12] and [13] have helped us to better understand the connection between preference on rules and blocking of rules. It is clear that not all preferences have the same importance. Preference on blocking rules should be more important.

5 Current status of the research and expected achievements

Our current research is focused on figuring out the details around the concept of preference importance. We consider the concept of preference importance to be important for our goals: to accept natural order in which rules have to be applied, and to produce intuitive results.

We see a promising direction. Splitting [5] shows that we can consider rules in a iterative manner, similar to the one we use to compute the answer set of a stratified program. The important difference is that there exist decision points where we must decide which of mutually blocking rules we want to use. These decision points are responsible for multiple answer sets. In other words, splitting creates a decision tree of rule application. In order to decide which rule to use in a decision point we have to decide which rules to apply in all former decision points. If we do not choose a particular branch of a decision tree there is no

need to consider rules in it. Hence, preference on rules on former decision points is more important than on later ones.

Next example demonstrates the concept of preference importance and sketches solution to selection of preferred answer sets.

▶ **Example 2.** Consider the following program with one decision point:

$r_1 : a \leftarrow not\ b$ $\qquad$ $r_2 : b \leftarrow not\ a$
$r_3 : c \leftarrow a, not\ d$ $\qquad$ $r_4 : d \leftarrow b$

r_1 is preferred over r_2
r_4 is preferred over r_3

It has two answer sets, $A_1 = \{a, c\}$ and $A_2 = \{b, d\}$, generated by $R_1 = \{r_1, r_3\}$ and $R_2 = \{r_2, r_4\}$, respectively.

Splitting sequence $\langle \{a, b\}, \{a, b, d\}, \{a, b, c, d\} \rangle$ divides rules into three groups $\Pi_0 = \{r_1, r_2\}$, $\Pi_1 = \{r_4\}$ and $\Pi_2 = \{r_3\}$. Due to the literal b in the body of rule r_4, group Π_1 depends on group Π_0. So there is natural order. Rules from Π_0 must be considered before rules in Π_1, and similarly Π_1 before Π_2.

In the first place, we must settle down the question of rule application for rules from Π_0. Answer sets of Π_0 are $\{a\}$ and $\{b\}$. It means that the first (also the only) decision point is whether to use rule r_1 or r_2. Since r_1 is preferred over r_2, we select to use rule r_1. We do not use r_2, so there is no way to generate A_2. Thus, A_2 is not preferred. And since r_2 is not used, also r_4 cannot be used. Consequently, there is no need to consider preference of rule r_4 over any other rule. Preference of r_4 over r_3 would be considered only if r_1 is not preferred over r_2 and r_2 is not preferred over r_1.

To sum up, we expect to propose a descriptive approach to preference handling for extended logic programs under answer set semantics with preference on rules. Our semantics should be selective, and conceptually based on comparison of generating sets. We hope to meet Principle I, III, IV and VI (from [1], [12], [13]). More importantly, we think that the use of preference importance as described above, will allow us to accept natural order of a program, and to have a semantics with intuitive conclusions. Once the semantics is defined, we plan to provide an implementation and a comparison to existing approaches.

References

1 G. Brewka and T. Eiter. Preferred answer sets for extended logic programs. *Artificial Intelligence*, 109(1-2):297–356, 1999.

2 Gerhard Brewka. Logic programming with ordered disjunction. In *Proceedings of AAAI-02*, pages 100–105. AAAI Press, 2002.

3 J. Delgrande, T. Schaub, and H. Tompits. A framework for compiling preferences in logic programs. *Theory and Practice of Logic Programming*, 3(2):129–187, 2003.

4 James P. Delgrande and Torsten Schaub. Expressing preferences in default logic. *Artificial Intelligence*, 123(1-2):41–87, 2000.

5 Vladimir Lifschitz and Hudson Turner. Splitting a logic program. In *Proceedings of the eleventh international conference on Logic programming*, pages 23–37, Cambridge, MA, USA, 1994. MIT Press.

6 A. Mileo and T. Schaub. Extending Ordered Disjunctions for Policy Enforcement: Preliminary report. In *The 2006 Federated Logic Conference*, page 45, 2006.

7 Chiaki Sakama and Katsumi Inoue. Prioritized logic programming and its application to commonsense reasoning. *Artificial Intelligence*, 123(1-2):185–222, 2000.
8 Torsten Schaub and Kewen Wang. A comparative study of logic programs with preference. In *IJCAI*, pages 597–602, 2001.
9 Torsten Schaub and Kewen Wang. A semantic framework for preference handling in answer set programming. *Theory and Practice of Logic Programming*, 3(4):39, 2003.
10 D. Van Nieuwenborgh and D. Vermeir. Preferred answer sets for ordered logic programs. *Theory and Practice of Logic Programming*, 6(1-2):107–167, 2006.
11 Ján Šefránek. Preferred answer sets supported by arguments. In *Proc. of the Workshop NMR*, 2008.
12 Ján Šefránek and Alexander Šimko. Warranted derivation of preferred answer sets, http://kedrigern.dcs.fmph.uniba.sk/reports/. Technical Report TR-2011-027, Comenius University, Faculty of Mathematics, Physics, and Informatics, 2011.
13 Alexander Šimko. Preferred answer sets - banned generating set approach. 2011.
14 Kewen Wang, Lizhu Zhou, and Fangzhen Lin. Alternating fixpoint theory for logic programs with priority. In *Computational Logic*, pages 164–178, 2000.
15 Y. Zhang and N. Foo. Answer sets for prioritized logic programs. In *Proceedings of the 1997 international symposium on Logic programming*, pages 69–83, 1997.

Implementation of Axiomatic Language

Walter W. Wilson[1]

1 **Dept. of Computer Science & Engineering**
The University of Texas at Arlington
Arlington, Texas 76019, USA
`wwwilson@acm.org`

Abstract

This report summarizes a PhD research effort to implement a type of logic programming language called "axiomatic language". Axiomatic language is intended as a specification language, so its implementation involves the transformation of specifications to efficient algorithms. The language is described and the implementation task is discussed.

1998 ACM Subject Classification D.1.6 Logic Programming, I.2.2 Automatic Programming

Keywords and phrases axiomatic language, specification, program transformation, unfold/fold

Digital Object Identifier 10.4230/LIPIcs.ICLP.2011.290

1 Introduction and Problem Description

This research project investigates a type of logic programming language called "axiomatic language" [1,2]. Axiomatic language is intended as a specification language where the user defines the external behavior of a program without giving an algorithm. The language implementation has the task of transforming this input specification into an equivalent efficient algorithm. This research project will attempt to make progress on this difficult problem. A secondary goal will be to make a software engineering case for axiomatic language as a specification language through example applications. Section 2 describes axiomatic language and its attributes and sections 3-6 discuss the transformation problem.

2 Background and Existing Literature

This section defines axiomatic language and discusses its novel aspects in comparison to other languages. Axiomatic language has three goals:

1. a pure specification language - what, not how
2. minimal, but extensible - as small and simple as possible
3. a meta-language - able to imitate and thus subsume other languages

We also have the goal of beauty and elegance for the language.

Axiomatic language is based on the idea that the external behavior of a function or program – even an interactive program – can be specified by a static infinite set of symbolic expressions that enumerate all possible inputs – or sequences of inputs – along with the corresponding outputs. The language is just a formal system for defining this symbolic expression set.

© Walter W. Wilson;
licensed under Creative Commons License NC-ND
Technical Communications of the 27th International Conference on Logic Programming (ICLP'11).
Editors: John P. Gallagher, Michael Gelfond; pp. 290–295

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

2.1 An Informal Overview

Axiomatic language can be described as pure definite Prolog with Lisp syntax, HiLog [3] higher-order generalization, and "string variables", which match a substring in a sequence. A typical Prolog predicate is represented in axiomatic language as follows,

```
father(bob,X)    ->    (father Bob %x)
```

where (expression) variables start with % and both upper and lowercase letters (as well as many special characters) can be used for symbols.

Natural numbers and their addition can be defined by the following "axioms":

```
(number 0).                     ! set of natural numbers
(number (s %n))< (number %n).

(plus % 0 %)< (number %).       ! natural number addition
(plus %1 (s %2) (s %3))< (plus %1 %2 %3).
```

These axioms generate "valid expressions" such as `(plus (s 0) (s 0) (s (s 0)))`, which is interpreted as the statement "$1 + 1 = 2$". Comments start after !.

String variables, which start with $, match a string of elements within a sequence. They enable more concise definitions of predicates on lists:

```
(concat ($1) ($2) ($1 $2))        ! list concatenation

(member % ($1 % $2)).             ! member of a sequence

(reverse () ()).                  ! reversing a sequence
(reverse (% $) ($rev %))< (reverse ($) ($rev)).
```

Some example valid expressions are `(concat (a b) (c d) (a b c d))` and `(member c (a b c))`. String variables can be considered a generalization of Prolog's list tail variables.

2.2 The Core Language

In axiomatic language a finite set of axioms generates a (usually) infinite set of valid expressions. An **expression** is:

an **atom** - a primitive, indivisible element,
an **expression variable**,
or a **sequence** of zero or more expressions and **string variables**.

Atoms are represented syntactically by symbols starting with the backquote ` (so the symbols seen previously are not atoms). A sequence is represented by a string of expressions and string variables separated by blanks and enclosed in parentheses: ``(`abc %n)``, ``(`M $ ())``.

An **axiom** consists of a **conclusion** expression and zero or more **condition** expressions, represented as follows:

<conclu> < *<cond1>*, . . . , *<condn>*.
<conclu>.   `!  an unconditional axiom`

An axiom generates an **axiom instance** by the substitution of values for the expression and string variables. An expression variable can be replaced by an arbitrary expression, the same value replacing the same variable throughout the axiom. A string varable can be replaced by a string of expressions and string variables. For example, the axiom,

```
(`a %x $w)< (`b %y $w), (`c (%x %y) $1).
```

has the instance,

```
(`a (` %) () `v)< (`b `y () `v), (`c ((` %) `y)).
```

by the substitution of `(` %)` for %x, `` `y `` for %y, ‘`() `v`’ for $w, and ‘’ (the null string) for $1.

The conclusion expression of an axiom instance is a **valid expression** if all the condition expressions of the axiom instance are valid expressions. By default, the conclusion of an unconditional axiom instance is a valid expression. For example, the two axioms,

```
(`a `b).
((%) $ $) < (% $).
```

generate the valid expressions ``(`a `b)``, ``((`a) `b `b)``, ``(((`a)) `b `b `b `b)``, etc.

2.3 Syntax Extensions

The expressiveness of axiomatic language is enhanced by adding certain syntax extensions to the core language. We let a single character in single quotes be syntactic shorthand for an expression that gives the 8-bit character code:

```
'A'   =   (`char (`0 `1 `0 `0 `0 `0 `0 `1))
```

A string of characters in single quotes within a sequence is equivalent to writing those single characters separately:

```
(... 'abc' ...)  =  (... 'a' 'b' 'c' ...)
```

A string of characters in double quotes is equivalent to the sequence of those characters:

```
"abc"  =  ('abc')  =  ('a' 'b' 'c')
```

(A quote character is repeated when it occurs in a character string enclosed by the same quote character: `"'"""` = `("'"')`.)

A symbol that does not begin with one of the special characters `` ` % $ ( ) ' " ! `` is equivalent to an expression that contains the symbol as a character string:

```
abc   =   (` "abc")
```

2.4 Specification by Enumeration [4]

We want to specify the external behavior of a program using a set of valid expressions. A program that maps an input file to an output file can be specified by an infinite set of symbolic expressions of the form,

```
(Program <input> <output>)
```

where <*input*> is a symbolic expression for a possible input file and <*output*> is the corresponding output file. For example, a program that sorts the lines of a text file could be represented by valid expressions such as the following:

```
(Program ("dog" "horse" "cow")     ! 3-line input file
         ("cow" "dog" "horse"))    ! sorted output file
```

Axioms would define these valid expressions for all possible input files.

An interactive program where the user types lines of text and the computer types lines in response could be represented by valid expressions such as,

```
(Program <out> <in> <out> <in> ... <out> <in> <out>)
```

where <*out*> is a sequence of zero or more output lines typed by the computer and <*in*> is a single input line typed by the user. Each Program expression gives a possible execution history. Valid expressions would be generated for all possible execution histories.

2.5 Novelty and Existing Work

This section lists some of the novel aspects of axiomatic language:

1. goal of pure specification – Axiomatic language has the ambitious and idealistic goal of completely freeing the user from thinking about implementation and efficiency.
2. specification by enumeration – We specify the external behavior of programs by enumerating all possible inputs/outputs represented as static symbolic expressions. This is a completely pure approach to the awkward problem of i/o in declarative languages [5].
3. definition vs. computation semantics – Axiomatic language is just a formal system for defining infinite sets of symbolic expressions, which are then interpreted. Prolog semantics, in contrast, are based on a model of computation.
4. minimal language – We see elegance in having minimal size with maximum expressiveness. Axiomatic language shares this goal with minimal Lisp systems.
5. Lisp syntax – Axiomatic language, like some other LP languages (MicroProlog [6], Allegro [7]) uses Lisp syntax instead of Edinburgh syntax. Predicate and function names are moved inside the parentheses and commas are replaced with blanks. Data lists also use this notation, which supports representing code as data.
6. higher-order syntax – Predicate and function names can be arbitrary expressions, including variables, and entire predicates can be represented by variables. This is the same as in HiLog, but with Lisp syntax.
7. non-atomic characters – Axiomatic language separates the definitions and rules of its semantics, which are fixed, from its syntax and syntax extensions such as character sets and their representation, which are likely to evolve. Non-atomic characters make character representations more explicit, but should be hideable with a well-designed library.
8. non-atomic symbols – Non-atomic symbols eliminate the need for built-in decimal numbers, since they can be easily defined through library utilities.
9. string variables – These provide pattern matching and meta-language support.
10. meta-language – Axiomatic language has the goal of being a single language that can provide the user with the features and expressiveness of any other language. Its flexible syntax and higher order capability should make it well-suited to metaprogramming, language-oriented programming [8], and embedded domain-specific languages [9].

11. no built-in arithmetic or other functions – The minimal nature and extensibility of axiomatic language means that basic arithmetic and other functions are provided through a library rather than built-in. But this also means that such functions have explicitly defined semantics and can be formally treated like regular code.
12. explicit definition of approximate arithmetic – Since there is no built-in floating point arithmetic, approximate arithmetic must also be defined in a library. But this means that symbolically defined numerical results would always be identical down to the last bit, regardless of future floating point hardware.
13. no built-in inequality between distinct symbols – Symbol inequality is easily defined from the non-atomic nature of symbols and characters.
14. no built-in negation – Axiomatic language defines recursively enumerable sets but not their complement. One can, however, define negation-as-failure on encoded axioms.
15. no non-logical operations such as cut – This follows from there being no procedural interpretation in axiomatic language.
16. no meta-logical operations such as var, set_of, find_all – These would have to be defined on encoded axioms.
17. no assert/retract - A set of axioms is static. Modifying this set must be done "outside" the language.

3 Goal of this Research

The ultimate goal of this research is the efficient implementation of axiomatic language. This means the automatic transformation of specifications to efficient algorithms. Unlike Prolog, where the user is careful to write clauses that will execute efficiently, we want the axiomatic language user to write specifications without concern about efficiency. A further goal is that this transformation be proven correct — the implementation algorithm is guaranteed to be equivalent to the specification.

One can argue that no finite system can successfully transform all possible user specifications. Just knowing whether or not a specification defines no output is, of course, undecidable. The best we can hope for is a system that can successfully transform the specifications of most "typical" programs.

In addition to their purpose of specification, axioms should be able to define an implementation algorithm, either by executing them in a Prolog-like manner or by modeling, say, a C subset. Thus, this transformation problem can be reduced to transforming one set of axioms to an equivalent set. Unfold/fold transformations [10] can be used here and will provide the guarantee of correctness for the resulting implementation.

4 Current Status and Preliminary Results

My initial work on proof involves proving "valid clauses". We define a "clause" to be the same as an axiom – a conclusion expression and zero or more condition expressions. A clause is "valid" with respect to a set of axioms if no additional valid expressions are generated when the clause is added to the set of axioms. For example, consider the set of axioms consisting of the number and plus axioms along with the following equality axiom:

```
(== % %).             ! identical expressions
```

The following clause,

```
(== %3a %3b)< (plus %1 %2 %3a), (plus %2 %1 %3b).
```

would be valid since no new equality expressions would be generated if this clause were added to the set of axioms. This clause states the commutativity of addition. Unfold/fold rules for proving valid clauses are being worked out and should be a basis for proving equivalent sets of axioms. Proving valid clauses may also be useful for proving assertions about specifications.

The initial work on transformation has involved developing a framework for manipulating axioms as data.

5 Open Issues and Expected Achievements

The automatic transformation of specifications to efficient algorithms is widely considered unsolvable [11]. Other work, however, shows more positive results [12]. The axiomatic language implementation problem has the advantage that the specifications will be completely detailed and the specification language is an extremely simple formal system. Unlike an interactive transformation system [13], we don't expect the user to manually transform his program; instead the transformation system will have to do its job on its own. We acknowledge that the system may not be able to do a good job on a given input specification, in which case an expert will be called on to add new knowledge. We hope that as knowledge continues to be added and generalized, the need for an expert's intervention will decline.

I wish to thank my advisor, Dr. Jeff Lei, for our discussions.

References

1 Wilson, W.W., Beyond Prolog: Software Specification by Grammar, ACM SIGPLAN Notices, Vol. 17, #9, pp. 34–43, Sept. (1982)
2 http://www.axiomaticlanguage.org
3 Chen, W., Kifer, M., Warren, D.S., HiLog: A Foundation for Higher-Order Logic Programming, J. of Logic Programming, Vol. 15, #3, pp. 187–230 (1993)
4 Wilson, W.W., Lei, Y., Specifying Input/Output by Enumeration, http://csl.stanford.edu/~christos/pldi2010.fit/wilson.specio.pdf (2010)
5 Peyton Jones, S., Tackling the Awkward Squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell, in Engineering Theories of Software Construction, ed Tony Hoare, Manfred Broy, Ralf Steinbruggen, IOS Press, pp. 47–96 (2001)
6 Clark, K.L., Micro-Prolog: Programming in Logic, Prentice Hall (1984)
7 Allegro Prolog, http://www.franz.com/support/documentation/8.2/doc/prolog.html
8 Ward, M., Language Oriented Programming, Software Concepts and Tools, 15, pp 147–161 (1994)
9 Hudak, P., Building Domain-Specific Embedded Languages, ACM Computing Surveys, Vol. 28, #4es, Dec. (1996)
10 Pettorossi, A., Proietti, M., Giugno, R., Synthesis and Transformation of Logic Programs Using Unfold/Fold Proofs, J. Logic Programming, Vol. 41 (1997)
11 Rich, C., Waters W., Automatic Programming: Myths and Prospects, IEEE Computer, vol. 21, pp. 40–51 (1988)
12 Binoux, G., et al, DESCARTES: An Automatic Programming System for Algorithmically Simple Programs, IWSSD '98 Proceedings of the 9th International Workshop on Software Specification and Design, IEEE (1988)
13 Renault, S., A System for Transforming Logic Programs, RR-97.04, Dept. of Computer Science, University of Rome – Tor Vergata (1997)

Two Phase Description Logic Reasoning for Efficient Information Retrieval

Zsolt Zombori[1]

1 **Department of Computer Science and Information Theory**
Budapest University of Technology and Economics
Budapest, Magyar tudósok körútja 2. H-1117, Hungary
zombori@cs.bme.hu

Abstract

Description Logics are used more and more frequently for knowledge representation, creating an increasing demand for efficient automated DL reasoning. However, the existing implementations are inefficient in the presence of large amounts of data. This paper summarizes the results in transforming DL axioms to a set of function-free clauses of first-order logic which can be used for efficient, query oriented data reasoning. The described method has been implemented in a module of the DLog reasoner openly available on SourceForge to download.

1998 ACM Subject Classification I.2.3: Deduction and Theorem Proving

Keywords and phrases reasoning, Description Logics, DLog, resolution

Digital Object Identifier 10.4230/LIPIcs.ICLP.2011.296

1 Overview

Description Logics (DLs) [1] is family of logic languages designed to be a convenient means of knowledge representation. They can be embedded into FOL, but – contrary to the latter – they are decidable which gives them a great practical applicability. A DL knowledge base consists of two parts: the TBox (terminology box) and the ABox (assertion box). The TBox contains general background knowledge in the form of rules that hold in a specific domain. The ABox stores knowledge about individuals. For example, let us imagine an ontology about the structure of a university. The TBox might contain statements like "Every department has exactly one chair", "Departments are responsible for at least 4 courses and for each course there is a department responsible for it". In contrast, the ABox might state that "The Department of Computer Science is responsible for the course Information Theory" or that "Andrew is the chair of the the Department of Music".

As DL languages are being used more and more frequently, there is an increasing demand for efficent automated reasoning services. Some reasoning tasks involve the TBox only. This is the case, for example, when we want to know what rules follow from the ones that we already know, or we want to verify that the model of a certain domain does not contain obvious mistakes in the form of contradictions and unsatisfiable concepts. We might want to make sure that there are not so many restrictions on the chair that it is impossible to be one (which is the case if he has to spend 70 percent of his time on research an another 70 percent on teaching). Other reasoning problems use both the ABox and the TBox: in such cases we might ask if a certain property holds for a certain individual (*instance check* – Is Andrew a chair?) or we might want to collect all individuals satisfying a given property (*instance retrieval* – What are the courses taught by the Department of Music?).

© Zsolt Zombori;
licensed under Creative Commons License NC-ND
Technical Communications of the 27th International Conference on Logic Programming (ICLP'11).
Editors: John P. Gallagher, Michael Gelfond; pp. 296–300

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

The Tableau Method [1] has long provided the theoretical background for DL reasoning and most existing DL reasoners implement some of its variants. Typical DL reasoning tasks can be reduced to concept consistency checking and this is exactly what the Tableau Method provides. While the Tableau itself has proven to be very efficient, the reduction to consistency check is rather costly for some ABox reasoning tasks. In particular, instance retrieval (i.e., to enumerate those individuals that belong to a given concept) requires running the Tableau Method for every single individual that appears in the knowledge base. Several techniques have been developed to make tableau-based reasoning more efficient on large data sets, (see e.g. [3]), that are used by the state-of-the-art DL reasoners, such as RacerPro [4] or Pellet [9].

Other approaches use first-order resolution for reasoning. A resolution-based inference algorithm is described in [5] which is not as sensitive to the increase of the ABox size as the tableau-based methods. The system KAON2 [8] is an implementation of this approach, providing reasoning services over the description logic language $\mathcal{SHIQ}$. The algorithm used in KAON2 in itself is not any more efficient for instance retrieval than the Tableau, but several steps that involve only the TBox can be performed before accessing the ABox, after which some axioms can be eliminated because they play no further role in the reasoning. This yields a qualitatively simpler set of axioms which then can be used for an efficient, query driven data reasoning. For the second phase of reasoning KAON2 uses a disjunctive datalog engine and not the original calculus. Thanks to the preprocessing, query answering is very focused, i.e., it accesses as little part of the ABox as possible. However, in order for this to work, KAON2 still needs to go through the whole ABox once at the end of the first phase.

2 Research Direction

In my PhD work I try to develop algorithms that can be used for reasoning over large ABoxes while the TBox is relatively small. These assumptions do not hold for all ontologies, but there are some very important examples when this is the case: one can, for instance, think of searching the WEB in the context of a specific, well characterized domain.

It seems that the complexity comes from two sources: on one hand the TBox contains complex background knowledge that requires sophisticated reasoning, and on the other the size of the ABox makes the sophisticated algorithm too slow in practice. An important lesson to be learned from KAON2 is that we might be able to cope with these two sources separately: let us perform the complex reasoning on the TBox – which we assume to be small – and turn it into a syntactically simpler set of rules before accessing the ABox. Afterwards, the simpler rules can be used for a focused, query driven ABox reasoning.

It is not clear how to separate the reasoning for the Tableau. This algorithm tries to build a model of the knowledge base, but a model of a small part of the knowledge base is not necessarily useful for constructing a model of the whole. Resolution approaches are more suitable: we can deduce implicit consequences of the axioms in one way at the beginning and then deduce further consequences in another way. In particular, we will be interested in solutions where we start with a bottom-up strategy and finish with a focused top-down strategy.

To perform first-order resolution, we need to transform the initial axioms to first-order clauses. While the initial knowledge base does not contain function symbols (there are no functions in DLs), existential restrictions and minimum restrictions in the TBox translate to existential quanifiers, which are eliminated by introducing new function symbols, called

skolem functions. This is problematic, because termination is hard to guarantee if we can obtain terms of ever increasing depth. Furthermore, some top-down reasoning algorithms (top-down reasoning is a must if the ABox is really large), such as datalog only work if there are no function symbols. For this reason, it is very important to find some way to eliminate function symbols before performing the data reasoning. Note that this is intuitively very possible: the ABox does not contain any knowledge about functions since they were introduced by us during clausifying the axioms from the TBox. Hence, everything that is to know about function symbols is in the clauses derived from the TBox and whatever role they play, they should be able to play it at the beginning of the reasoning.

3 Two Phase Reasoning

The above considerations motivate a two phase reasoning algorithm. In the first phase we only work with the clauses derived from the TBox. We use a bottom-up algorithm, deduce lots of consequences of the TBox, in particular all the important consequences of the clauses containing function symbols. By the end of the first phase, function symbols can play no further role and hence the clauses containing them can be eliminated. The second phase now begins and the reduced clause set can be used for a focused, top-down reasoning on the ABox.

This separation of TBox and ABox reasoning is only partially achieved in [8]. By the end of the first phase, we can only eliminate clauses with term depth greater than one. So, while function symbols persist, there is no more nesting of functions into each other. In order for the second phase to work, all function symbols are eliminated using a *syntactic* transformation: for every function symbol and every constant in the ABox a new constant is introduced. Note that this step involves scanning through the ABox and results in adding new constants whose number is linear in the size of the ABox.

Reading the whole ABox even once is not a feasible option in case the ABox contains billions of assertions or the content of the ABox changes so frequently that on-the-fly ABox access is an utmost necessity. Such scenarios include reasoning on web-scale or using description logic ontologies directly on top of existing information sources, such as in a DL based information integration system.

4 Results

I started my PhD at Budapest University of Technology in September 2009. I work as member of a team developing the DLog DL data reasoner [7], available to download at `http://www.dlog-reasoner.org`. This is a resolution based reasoner, built on principles similar to KAON2. One difference is that instead of a datalog engine, we use the reasoning mechanism of the Prolog language [2] to perform the second phase [6]. Reasoning with function symbols using Prolog is possible, unlike the datalog engine, but for considerations about termination it is equally important to eliminate function symbols during the first phase.

I work to provide DLog with a purely two phase reasoning algorithm. In [10] I presented a modified resolution calculus for the $\mathcal{SHIQ}$ language that allows us to perform more inferences in the first phase (compared with KAON2), yielding a simpler TBox to work with in the second phase. Namely, the new calculus ensures that no function symbols remain at all, without the need to go through the ABox. The modification makes the first phase somewhat slower, however, the speed of the second phase becomes independent of the amount of data

that is irrelevant to the query. The greater the ABox the better DLog performs compared to its peers. Another great advantage of DLog is that its architecture allows for storing the ABox in an external database that is assessed through direct database queries.

Afterwards, I worked on a new DL calculus ([12] and [11]) where we move resolution from first-order clauses to DL axioms, saving many intermediary transformation steps. Even if the speed of the first phase is not as critical as that of the second, this optimisation is important. With the increase of the TBox the first phase can become hopelessly slow, such that DLog is impossible to use. Making the first phase faster slightly increases the critical TBox size within which it is still worth reasoning with DLog. On the other hand, the DL calculus is a complete algorithm for TBox reasoning. It is novel in that the metodology is still resolution, but the inference rules are given directly for DL expressions. It is not as fast for TBox reasoning as the Tableau, but it provides an alternative and I hope that it will motivate research in the area. I tried to extend the DL calculus to ABox reasoning, but I have not yet been successful in doing that.

I managed to make DLog support the $\mathcal{RIQ}$ language which extends $\mathcal{SHIQ}$ with complex role hierarchies. I also tried to incorporate nominals into the reasoner, i.e., move from $\mathcal{RIQ}$ to $\mathcal{ROIQ}$, however, this work yielded only negative results. It turned out that the presence of nominals blurs the distinction between TBox and ABox (the whole content of the ABox can be rephrased using TBox axioms), hence there is no possibility of separating terminology and data reasoning into two separate phases. These last two results have not yet been published.

5 Current Work

I would like to fully explore the relation between DLog and the $\mathcal{SROIQ}$ language on which the new web ontology language OWL 2 is based. This exploration involves partly to extend DLog towards more expressive language constructs and partly to understand its limitations.

I also am working to better explore the complexity of our algorithms. Bottom up reasoning in the first phase is very costly: it is at most triply exponential in the size of the TBox, although our experiments indicate that there could be a better upper bound. We also need to better explore the clauses that are deduced from the TBox. While our main interest is to eliminate function symbols, we deduce other consequences as well. Some of them make the data reasoning faster, some of them do not, and we cannot yet well characterize them.

6 Concluding Remarks

With the proliferation of knowledge intensive applications, there is a vivid research in the domain of knowledge representation. Description Logics are designed to be a convenient means for such representation task. One of the main advantages over other formalisms is a clearly defined semantics. This opens the possibility to provide reasoning services with mathematical rigorousness.

My PhD work is concerned with Description Logic reasoning. I am particularly interested in ABox reasoning when the available data is really large. This domain is much less explored than TBox reasoning. Nevertheless, reasoning over large ABoxes is useful for problems like web-based reasoning.

I am one of the developers of the DLog data reasoner which implements a two phase reasoning: the first phase uses complex reasoning to turn the TBox into simple rules, while the second phase is geared towards fast query answering over large ABoxes. DLog currently

supports the $\mathcal{SHIQ}$ DL language, but we plan to extend it as far as $\mathcal{SROIQ}$, the logic behind OWL 2.

References

1. F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation and Applications.* Cambridge University Press, 2004.
2. Alain Colmerauer and Philippe Roussel. *The birth of Prolog.* ACM, New York, NY, USA, 1996.
3. V. Haarslev and R. Möller. Optimization techniques for retrieving resources described in OWL/RDF documents: First results. In *Ninth International Conference on the Principles of Knowledge Representation and Reasoning, KR 2004, Whistler, BC, Canada, June 2-5*, pages 163–173, 2004.
4. V. Haarslev, R. Möller, R. van der Straeten, and M. Wessel. Extended Query Facilities for Racer and an Application to Software-Engineering Problems. In *Proceedings of the 2004 International Workshop on Description Logics (DL-2004), Whistler, BC, Canada, June 6-8*, pages 148–157, 2004.
5. Ullrich Hustadt, Boris Motik, and Ulrike Sattler. Reasoning for Description Logics around SHIQ in a resolution framework. Technical report, FZI, Karlsruhe, 2004.
6. Gergely Lukácsy and Péter Szeredi. Efficient description logic reasoning in Prolog: the DLog system. *Theory and Practice of Logic Programming*, 09(03):343–414, May 2009.
7. Gergely Lukácsy, Péter Szeredi, and Balázs Kádár. Prolog based description logic reasoning. In Maria Garcia de la Banda and Enrico Pontelli, editors, *Proceedings of 24th International Conference on Logic Programming (ICLP'08),Udine, Italy*, pages 455–469, December 2008.
8. Boris Motik. *Reasoning in Description Logics using Resolution and Deductive Databases.* PhD thesis, Universität Karlsruhe (TH), Karlsruhe, Germany, January 2006.
9. Evren Sirin, Bijan Parsia, Bernardo Cuenca Grau, Aditya Kalyanpur, and Yarden Katz. Pellet: A practical OWL-DL reasoner. *Web Semant.*, 5(2):51–53, 2007.
10. Zsolt Zombori. Efficient two-phase data reasoning for description logics. In Max Bramer, editor, *IFIP AI*, volume 276 of *IFIP*, pages 393–402. Springer, 2008.
11. Zsolt Zombori. A resolution based description logic calculus. *Submitted to Acta Cybernetica*, 2009.
12. Zsolt Zombori and Gergely Lukácsy. A resolution based description logic calculus. In Boris Motik Bernardo Cuenca Grau, Ian Horrocks and Ulrike Sattler, editors, *Proceedings of the 22nd International Workshop on Description Logics (DL 2009), Oxford*, volume 477 of *CEUR Workshop Proceedings*, Oxford, UK, July 27-30 2009.