LNCS 5649

Patricia M. Hill
David S. Warren (Eds.)

Logic Programming

25th International Conference, ICLP 2009
Pasadena, CA, USA, July 2009
Proceedings

Springer

Patricia M. Hill   David S. Warren (Eds.)

Logic Programming

25th International Conference, ICLP 2009
Pasadena, CA, USA, July 14-17, 2009
Proceedings

Springer

Volume Editors

Patricia M. Hill
University of Leeds, School of Computing
Leeds, LS2 9JT, UK
E-mail: hill@comp.leeds.ac.uk

David S. Warren
Stony Brook University, Department of Computer Science
Stony Brook, NY 11794-4400, USA
E-mail: warren@cs.sunysb.edu

Library of Congress Control Number: Applied for

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

LNCS Sublibrary: SL 2 – Programming and Software Engineering

ISSN	0302-9743
ISBN-10	3-642-02845-4 Springer Berlin Heidelberg New York
ISBN-13	978-3-642-02845-8 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 12705320 06/3180 5 4 3 2 1 0

Preface

This volume contains the proceedings of the 25th International Conference on Logic Programming (ICLP 2009). The conference took place in Pasadena, California during July 14–17, 2009. The ICLP series of conferences is aimed at providing a technical forum for presenting and disseminating innovative research results in the field of logic programming.

The conference, which was co-located with the International Joint Conference on Artificial Intelligence (IJCAI), featured technical presentations, tutorials, invited talks, and a number of special events, including:

- The 5th ICLP Doctoral Student Consortium
- The Second Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP)
- The 6th International Workshop on Constraint Handling Rules (CHR)
- The 9th International Colloquium on Implementation of Constraint and LOgic Programming Systems (CICLOPS)
- The 4th International Workshop on (Constraint) Logic Programming and Software Engineering (CLPSE)
- The First Workshop on Commercial Users of Logic Programming (CULP)
- Workshop on ISO Prolog — WG17
- The 19th Workshop on Logic-based methods in Programming Environments (WLPE)

Since the first conference held in Marseilles in 1982, ICLP has been the premiere international conference for disseminating research results in logic programming. The present edition of the conference received 69 submissions in three categories: application, system and tool, and technical papers. From these, the Program Committee selected 29 papers for presentation and inclusion in the proceedings. In addition, the committee selected nine short papers describing on-going research work, PhD theses and research project overviews for poster presentations and inclusion in the proceedings.

As in previous years, the Program Committee selected the best paper and the best student paper. The Best Paper Award went to Henning Christiansen and John Gallagher for their paper "Non-discriminating Arguments and Their Uses" while the Best Student Paper went to Matthias Broecheler and Gerardo Simari for their paper (co-authored with V.S. Subrahmanian) "Using Histograms to Better Answer Queries to Probabilistic Logic Programs."

ICLP 2009 included invited talks by Taisuke Sato titled "Generative Modeling by PRISM;" by Paulo Moura titled "From Plain Prolog to Logtalk Objects: Effective Code Encapsulation and Reuse;" by Chris Mungall titled "Experiences Using Logic Programming in Bioinformatics;" and by Marc Denecker titled "A Knowledge Base System Project for FO(.)". The program also featured four tutorials: "Probabilistic Logic Learning" by Luc De Raedt; "Enabling Serendipitous

Search on the Web of Data Using Prolog" by Jan Wielemaker; "(C)LP Tracing and Debugging" by Mireille Ducasse; and "Untangling Reverse Engineering with Logic and Abstraction" by Andy King.

ICLP 2009 was organized by the Association for Logic Programming (ALP) in collaboration with the IJCAI 2009 Organizing Committee and the Organizing Committees of the special events associated with ICLP 2009. ICLP 2009 was sponsored by ALP, the University of Texas at Dallas, and the University of Nebraska, Omaha.

Many people contributed to the success of the conference, to whom we hereby extend our gratitude and thanks. The IJCAI organizing staff, Carol Hamilton (Executive Director, AAAI), Richard J Doyle (Local Arrangements chair) and Craig Boutilier (PC Chair), enabled the successful collaboration and co-location with IJCAI. The General Chairs, Gopal Gupta and Hai-Feng Guo, both worked extremely hard to ensure a memorable event, providing leadership in all aspects of the organization process. The members of the Program Committee provided invaluable help in the process of selecting papers and developing the conference program. The numerous referees invested countless hours in reading submissions and providing professional reviews. Manuel Carro (the Workshop Chair) and Marcello Balduccini (a Doctoral Consortium Co-chairs); and Tom Schrijvers (a Doctoral Consortium Co-chair and the Prolog Programming Contest Chair) contributed to ICLP by the addition of exciting and well-organized events, while Ricardo Rocha, the Publicity Chair, helped to ensure its overall success. This year, with only a very short period between ICLP 2008 (held in December 2008) and the March 3 deadline for ICLP 2009 submitted papers, the biggest "thank you" of all should go to the large number of researchers in the logic programming community who rose to the challenge and submitted many excellent technical papers and posters. Last but not least, we wish to thank the developers of the EasyChair conference management system, which made our job much easier.

April 2009

Patricia M. Hill
David S. Warren

Conference Organization

Program Chairs

Patricia M. Hill
David S. Warren

Organization

ICLP 2009 was organized by the Association for Logic Programming (ALP) in collaboration with the IJCAI 2009 Organizing Committee and the Organizing Committees of the special events associated with ICLP 2009.

Organizing Committee

General Chairs	Gopal Gupta (University of Texas at Dallas)
	Hai-Feng Guo (University of Nebraska at Omaha)
Program Chairs	Patricia M. Hill (University of Leeds)
	David S. Warren (Stony Brook University)
Workshop Chair	Manuel Carro (Technical University of Madrid)
Publicity Chair	Ricardo Rocha (University of Porto)
Doctoral Consortium Chairs	Marcello Balduccini (Kodak Research Labs)
	Tom Schrijvers (Katholieke Universiteit Leuven)
Prolog Programming Contest Chair	Tom Schrijvers (Katholieke Universiteit Leuven)

Program Committee

Annalisa Bossi	Università Cà Foscari di Venezia, Italy
Pedro Cabalar	Corunna University, Spain
Mireille Ducassé	IRISA/INSA Rennes, France
Esra Erdem	Sabanci University, Turkey
François Fages	INRIA Rocquencourt, France
Thom Frühwirth	University of Ulm, Germany
Maurizio Gabrielli	University of Bologna, Italy
Maria Garcia de la Banda	Monash University, Australia

Hai-Feng Guo	University of Nebraska at Omaha, USA
Gopal Gupta	University of Texas at Dallas, USA
Michael Hanus	CAU Kiel, Germany
Patricia M. Hill Co-chair	University of Leeds, UK
Katsumi Inoue	NII, Japan
Joxan Jaffar	National University of Singapore, Singapore
Andy King	University of Kent, UK
Nicola Leone	University of Calabria, Italy
Fangzhen Lin	HKUST, Hong Kong
Paulo Moura	University of Beira Interior, Portugal
Steve Muggleton	Imperial College London, UK
Gopalan Nadathur	University of Minnesota, USA
Lee Naish	University of Melbourne, Australia
Enrico Pontelli	New Mexico State University, USA
Ricardo Rocha	University of Porto, Portugal
Torsten Schaub	University of Potsdam, Germany
Terrance Swift	Universidade Nova de Lisboa, Portugal
Péter Szeredi	Budapest University of Technology and Economics, Hungary
Mirek Truszczynski	University of Kentucky, USA
Frank Valencia	LIX École Polytechnique, Paris, France
Wim Vanhoof	University of Namur, Belgium
David S. Warren Co-chair	Stony Brook University, USA
Neng-Fa Zhou	CUNY Brooklyn College, USA

External Reviewers

Salvador Abreu
Mario Alviano
Gianluca Amato
Péter Antal
Jesus Aranda
Frédéric Benhamou
Sabrina Baselice
Hariolf Betz
Gauvain Bourgne
Bernd Brassel
Gerhard Brewka
Annamaria Bria
Francesco Calimeri
Ferdinanda Camporesi
Mats Carlsson
Luciano Caroprese
Manuel Carro
Kenil Cheng
Henning Christiansen
John Cleary
Mike Codish
Vítor Santos Costa
Susanna Cozza
Carlos Damásio
François Degrave
Bart Demoen
Jin Song Dong
Agostino Dovier
Gregory James Duck
Halit Erdogan
Wolfgang Faber
Moreno Falaschi
Raphael Finkel
Riccardo Focardi
Nuno Fonseca
Andrea Formisano
Andrew Gacek
Martin Gebser
Cinzia Di Giusto
Gianluigi Greco
Rémy Haemmerlé
Jacob Howe
Jianmin Ji
Roland Kaminski
Benjamin Kaufmann
Zeynep Kiziltan
Feliks Kluzniak
Herbert Kuchen

Joohyung Lee
Annie Liu
Lunjin Lu
Gergely Lukácsy
Damiano Macedonio
Michael Maher
Marco Maratea
Julien Martin
Thierry Martinez
Dániel Marx
Jacopo Mauro
Maria Chiara Meo
Fred Mesnard
Alessandra Mileo
Richard Min
Jorge Navas
Thang Manh Nguyen
Vitor Nogueira
Carlos Olarte
Max Ostrowski
Gergely Patai
Etienne Payet
David Pearce
Jorge Perez
Carla Piazza
Brigitte Pientka
Alexandre Miguel Pinto
Inna Pivkina
Alberto Policriti
Maurizio Proietti
Frank Raiser
C.R. Ramakrishnan
Oliver Ray
Francesco Ricca
Csaba Ringhofer
Konstantinos Sagonas
Chiaki Sakama
Andrew Santosa
Taisuke Sato
Peter Schachte
Peter Schneider-Kamp
Tom Schrijvers
Yi-Dong Shen
Fernando Silva
Jan-Georg Smaus
Sylvain Soliman
Tran Son
Harald Søndergaard
Peter Stuckey
Mahadevan Subramaniam
Sven Thiele
Paolo Torroni
Agustín Valverde
József Váncza
Yisong Wang
Stefan Woltran
Roland Yap
Jia-Huai You
Stéphane Zampelli

Table of Contents

Invited Talks

Experiences Using Logic Programming in Bioinformatics 1
Chris Mungall

A Knowledge Base System Project for FO(.) 22
Marc Denecker

From Plain Prolog to Logtalk Objects: Effective Code Encapsulation and Reuse 23
Paulo Moura

Generative Modeling by PRISM 24
Taisuke Sato

Tutorials

Enabling Serendipitous Search on the Web of Data Using Prolog 36
Jan Wielemaker

Untangling Reverse Engineering with Logic and Abstraction 37
Andy King

(C)LP Tracing and Debugging 38
Mireille Ducassé

Probabilistic Logic Learning – A Tutorial Abstract 39
Luc De Raedt

Best Papers

Using Histograms to Better Answer Queries to Probabilistic Logic Programs 40
Matthias Broecheler, Gerardo I. Simari, and V.S. Subrahmanian

Non-discriminating Arguments and Their Uses 55
Henning Christiansen and John P. Gallagher

Applications I

Preprocessing for Optimization of Probabilistic-Logic Models for Sequence Analysis 70
Henning Christiansen and Ole Torp Lassen

Stabilization of Information Sharing for Queries Answering in Multiagent Systems 84
Phan Minh Dung, Do Duc Hanh, and Phan Minh Thang

Logic Programming for Multiagent Planning with Negotiation 99
Tran Cao Son, Enrico Pontelli, and Chiaki Sakama

Implementation I

Answer Set Programming with Constraints Using Lazy Grounding 115
Alessandro Dal Palù, Agostino Dovier, Enrico Pontelli, and Gianfranco Rossi

Computing Loops with at Most One External Support Rule for Disjunctive Logic Programs 130
Xiaoping Chen, Jianmin Ji, and Fangzhen Lin

Theory

Modular Nonmonotonic Logic Programming Revisited 145
Minh Dao-Tran, Thomas Eiter, Michael Fink, and Thomas Krennwallner

Merging Logic Programs under Answer Set Semantics 160
James Delgrande, Torsten Schaub, Hans Tompits, and Stefan Woltran

Reducts of Propositional Theories, Satisfiability Relations, and Generalizations of Semantics of Logic Programs 175
Mirosław Truszczyński

Implementation II

A Tabling Implementation Based on Variables with Multiple Bindings 190
Pablo Chico de Guzmán, Manuel Carro, and Manuel Hermenegildo

A Term-Based Global Trie for Tabled Logic Programs 205
Jorge Costa, João Raimundo, and Ricardo Rocha

Analysis

A New Approach to Non-termination Analysis of Logic Programs 220
Dean Voets and Danny De Schreye

Constraints I

Constraint Answer Set Solving 235
Martin Gebser, Max Ostrowski, and Torsten Schaub

On the Implementation of Weight Constraint Rules in Conflict-Driven ASP Solvers . . . 250
Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub

System and Tool Descriptions

A Language for Large Ensembles of Independently Executing Nodes 265
Michael P. Ashley-Rollman, Peter Lee, Seth Copen Goldstein, Padmanabhan Pillai, and Jason D. Campbell

Integrating Software Testing and Run-Time Checking in an Assertion Verification Framework . . . 281
Edison Mera, Pedro Lopez-García, and Manuel Hermenegildo

Debugging for Model Expansion . . . 296
Johan Wittocx, Hanne Vlaeminck, and Marc Denecker

Applications II

Metabolic Network Expansion with Answer Set Programming . . . 312
Torsten Schaub and Sven Thiele

Answer Set Programming for Single-Player Games in General Game Playing . . . 327
Michael Thielscher

Finding Similar or Diverse Solutions in Answer Set Programming . . . 342
Thomas Eiter, Esra Erdem, Halit Erdoğan, and Michael Fink

Implementation III

Attributed Data for CHR Indexing . . . 357
Beata Sarna-Starosta and Tom Schrijvers

User Defined Indexing . . . 372
David Vaz, Vítor Santos Costa, and Michel Ferreira

Constraints II

Integration of Abductive Reasoning and Constraint Optimization in SCIFF . . . 387
Marco Gavanelli, Marco Alberti, and Evelina Lamma

Encoding Table Constraints in CLP(FD) Based on Pair-Wise AC . . . 402
Neng-Fa Zhou

Types for Secure Pattern Matching with Local Knowledge in Universal Concurrent Constraint Programming 417
Thomas Hildebrandt and Hugo A. López

Probability, Uncertainty

Logic Programming with Defaults and Argumentation Theories 432
Hui Wan, Benjamin Grosof, Michael Kifer, Paul Fodor, and Senlin Liang

Qualified Computations in Functional Logic Programming 449
Rafael Caballero, Mario Rodríguez-Artalejo, and Carlos A. Romero-Díaz

Logic Programs under Three-Valued Łukasiewicz Semantics 464
Steffen Hölldobler and Carroline Dewi Puspa Kencana Ramli

Short Papers

Execution Control for CHR 479
Leslie De Koninck

Demand-Driven Normalisation for ACD Term Rewriting 484
Leslie De Koninck, Gregory J. Duck, and Peter J. Stuckey

One More Decidable Class of Finitely Ground Programs 489
Yuliya Lierler and Vladimir Lifschitz

Optimizing Compilation and Computational Complexity of Constraint Handling Rules: Ph.D. Thesis Summary 494
Jon Sneyers

Proving Termination by Invariance Relations 499
Paolo Pilozzi and Danny De Schreye

Automating Termination Proofs for CHR 504
Paolo Pilozzi and Danny De Schreye

An Overview of FORCES: An INRIA Project on Declarative Formalisms for Emergent Systems 509
Jesús Aranda, Gerard Assayag, Carlos Olarte, Jorge A. Pérez, Camilo Rueda, Mauricio Toro, and Frank D. Valencia

An Engine for Computing Well-Founded Models 514
Terrance Swift

Incremental Answer Completion in the SLG-WAM 519
Terrance Swift, Alexandre Miguel Pinto, and Luís Moniz Pereira

Doctoral Consortium

Research Summary: Tabled Evaluation for Transaction Logic Programs 525
Paul Fodor

Research Summary: Logic Programming for Massively Distributed Systems 527
Michael P. Ashley-Rollman

Research Summary: Extending Elimination Algorithms for Functional Constraints to Solve Two Integer Variables per Inequality 530
Chendong Li

Research Summary: A Cognitive Architecture for a Service Robot: An Answer Set Programming Approach 532
Jianmin Ji

Research Summary: Termination of CHR 534
Paolo Pilozzi

Research Summary: Intelligent Natural Language Processing Techniques and Tools 536
Alessio Paolucci

Stochastic Reasoning with Models of Agent Behavior 538
Gerardo I. Simari

Research Summary: Analysing Graph Transformation Systems Using Extended Methods from Constraint Handling Rules 540
Frank Raiser

Modular Action Language $\mathcal{ALM}$ 542
Daniela Inclezan

Answering Questions from Natural Language Using A-Prolog 544
Yana Todorova

Belief Logic Programming 547
Hui Wan

Logic-Statistic Models with Constraints for Biological Sequence Analysis 549
Christian Theil Have

Fusion of Logic Programming and Description Logics 551
Mantas Šimkus

Research Summary: Non-termination Analysis of Logic Programs 553
Dean Voets

Study of Random Logic Programs 555
Gayathri Namasivayam

Locally Distributed Predicates: A Programming Facility for Distributed State Detection 557
Michael De Rosa

Capturing Fair Computations on Concurrent Constraint Language 559
Paola Campli and Stefano Bistarelli

Constraint Based Languages for Biological Reactions 561
Marco Bottalico and Stefano Bistarelli

Author Index 563

Experiences Using Logic Programming in Bioinformatics

Chris Mungall

Abstract. Reverse engineering complex biological systems requires the integration of multiple different databases using detailed background knowledge. Logic programming can provide a means of both performing integrative queries and rule-based inference to account for implicit knowledge.

The Biological Logic Programming toolkit (Blipkit) was developed as a means of doing this kind of data integration. Implemented in SWI-Prolog, Blipkit has models of different aspects of life sciences data, including genes and gene sequences, RNA structures, evolutionary relationships, phenotypes and biological interactions. These can be combined to answer complex questions spanning multiple datasources. Blipkit also has means of integrating with and combining life sciences databases and ontologies.

1 Introduction

1.1 Background

The study of biological systems is progressing at an astonishing rate. The determination of the three billion bases of the reference DNA sequence of the human genome in 2001 was a landmark event in science, but accomplishment will be dwarfed with the advent of next-generation massively parallel sequencing technologies which allow the sequencing of genomes of individual organisms or cells on a truly massive scale. The scientific and medical potential is enormous. For medical purposes we would like to know the mechanisms by which individual chemical changes in DNA molecules combine with other forces to give rise to effects of clinical importance. However, a DNA sequence is not in itself sufficient to unlock this potential: sequence data must be analyzed in the context of other rich and complex data derived from a variety of life forms. How is the gene structured in the genome? How is it related to other genes, going back to common ancestors hundreds of millions of years ago? What structures do these genes encode, and how do these structures interact with other similar structures to give rise to a functioning organism – or in the case of deleterious genetic variation, the dysfunctioning of an organism?

This richness of data and the attendant challenges in analyzing it has lead to a new multi-disciplinary endeavour: bioinformatics, the application of computer science and informatics to solving biological problems. One of the biggest challenges in bioinformatics is *semantic multi-scale data integration*, automatically combining facts from a variety of heterogeneous sources using background knowledge to answer complex questions and yield new insights. Historically this integration has been done in an ad-hoc fashion, using scripting languages to write disposable programs for gathering together data for specific purposes[19]. Recognizing the inherent problems with this approach, the community has been a move towards providing Application Programmer Interfaces (APIs) to data, both object-oriented and web services based[20]. These systems are typically successful for answering specific questions about a limited range of datatypes, but queries across databases cannot easily be combined. In addition, it is difficult to integrate knowledge and semantics into the query answering process[21].

Whereas a large portion of scientific programming comes down to "number crunching", many bioinformatics analyses come down to "symbol crunching". This can be an ideal substrate for logic programming (LP). LP has been successively applied to individual problems, programming "in the small", but there has been less in the way of using LP for integrative analyses and programming "in the large".

The Biological Logic Programming toolkit (Blipkit) was developed as an experiment in applying LP techniques to data integration and "programming in the large" in bioinformatics. It is a collection of prolog modules for integrating, modeling, querying and performing complex operations over diverse biological data. Also known as BioProlog, it comprises one of the Bio* projects under the aegis of the Open Bioinformatics Foundation (OBF), alongside BioPerl, BioJava, BioRuby and others. Each of these projects represents a community effort to provide a relatively comprehensive integrated library of code for researchers in the life sciences that takes advantages of the features of the host programming language. Most of these bioinformatics language libraries use an object-oriented approach, as is popular in software engineering today. Blipkit, written in SWI-Prolog[23], offers a radically different approach, and can be described as predicate-oriented as opposed to object-oriented. The assumption underpinning the development of Blipkit was that this would offer some unique advantages, especially when applied to complex multi-source data integration problems requiring the application of logical rules and inference.

This paper first describes the organizational principles of the library, then illustrates a subset of the domains covered, focusing on the biology of genomes. The development of this library has yielded some interesting lessons, both in bioinformatics and for the logic programming community at large. These are presented at the end of the paper.

2 A Biological Logic Programming Toolkit

2.1 Modular Organisation

Most software developers would agree that all non-trivial programming projects benefit from a modular design. Partitioning programs into modules helps make programs maintainable by ensuring a separation of concerns. Unfortunately, Prolog systems vary tremendously with respect to their module systems (if they provide them at all). This has not historically been a problem for "programming

Table 1. List of packages included in Blipkit

Package	Core Model	Description
metadata	**metadata_db**	annotations and metadata
ontol	**ontol_db**	ontologies
genomic	**genome_db** , **seqfeature_db**	DNA sequences and features (1D)
structure	**rna_db**	Secondary and tertiary structures (2D and 3D)
phylo	**phylo_db**	Phylogenetic trees and evolutionary history
pheno	**pheno_db**	Phenotypes and diseases
curation	**curation_db**	Statements about biology
sb	**sb_db** , **interaction_db**	Systems biology and interactions
sql		Relational databases
web		Accessing data and services over the web
serval		Web application framework
stats		Statistical calculations
blipcore		Miscellaneous, I/O

in the small" style research projects, but it does hamper the adoption of Prolog for "programming in the large" style projects. The variability of module systems means that Blipkit is largely restricted to its host implementation, SWI-Prolog. The implications of this decision are discussed later on.

The modular organization of blipkit is as follows. Modules are organized into *packages*, with each package consisting of multiple modules and corresponding to a particular domain of the life sciences: for example, one package for representing genes, the other for representing the evolutionary history of those genes, and yet another for the interactions those genes participate in. There are also packages that are independent of the life-sciences per-se – for example, packages for representing metadata, or for working with relational databases. Table 1 shows the packages that comprise Blipkit.

Models. Each package is centered around one or more *models* of the domain in question[1]. Model modules by convention always have the **_db** suffix. For any given package there may be multiple models, each representing complementary overlapping views on that domain. Model modules consist of modeling predicates, each corresponding to some relationship or type within that domain. These predicates are split into two disjoint categories, extensional and intensional. Extensional predicates are intended to be loaded or asserted as unit clauses or *facts* (i.e head with no body), whilst intensional predicates have a body and are never asserted. From a database perspective these can be thought of as tables and views respectively. By convention, models generally adhere to the Datalog subset of prolog (i.e. no compound terms as arguments) in order to increase interoperability with relational databases and datalog systems.

Prolog has no inherent notion of intensional and extensional predicates – clauses are largely treated equally (although systems may distinguish between

[1] The term model is here used in the sense of a schema or data model, rather than of a stable model in disjunctive datalog.

dynamic and compiled predicates, these do not strictly correspond). Blipkit has a metamodeling module **dbmeta** that each model uses to describe itself, primarily using the **extensional/1** directive/predicate to declare a predicate as being extensive. This makes it easier to perform options such as writing all facts in a model to a file. This can be thought of as partially analagous to the Data Description Language (DDL) in a relational schema.

In addition, both extensional and intensional predicates are extensively documented using the PlDoc system[24].

Utility modules. A large chunk of bioinformatics is unfortunately devoted to prosaic plumbing exercises - there is a bewildering plethora of different data formats of varying degrees of formality, and these formats must be parsed in order to be able to integrate data from a variety of source. Often it is useful to be able to export model facts conforming to these formats.

Thus blipkit provides many different parsers, translators and writers to handle these different formats. These are divided into four categories: (i) XML-based formats are parsed using the SWI-Prolog SGML package, and then translated to model assertions using a prolog XSLT-like mapping specification (ii) Tabular formats are translated to prolog facts using a uniform syntactic translation, and then the facts are translated to model assertions using prolog (iii) For some formats for which a BNF grammar exists or can be created, Blipkit uses a DCG (Definite Clause Grammar) to parse the data (iv) for other formats we leverage parsers written in other languages such as perl, and write perl programs to generate prolog facts.

Uniform naming conventions help clarify large codebase. By convention, parsers are named **parser_*format***. XML to model translation modules are named ***model*_xmlmap_*format***.

There are also modules for importing and exporting facts from ontological representations and databases - these are discussed further on.

A general purpose module called **io** deals with input and output from these formats. Each blipkit installation also has a data source registry. This means that the programmer typically just has to call the **load_biosource/1** command, giving the name of the data source, and the Blipkit system will utilize the correct parser and populate the relevant model module with facts. The system will also take care of fetching remote data dumps across HTTP, maintaining a local cache. The SWI-Prolog **qcompile/1** predicate is immensely useful for making a fast-loading pre-compiled prolog database from a data source.

2.2 Ontologies and Metadata

Regardless of the domain being represented, there are often common modeling predicates that can be reused in a number of contexts. For example, regardless of whether the elements in our domain are representations of genes, chemical entities or human patients, these elements have shared attributes such as the *primary label* by which these elements are known, *alternate labels*, *identifiers* and so on. The **metadata_db**model consists of mostly extensional predicates such as **entity_label/2**, **entity_synonym/2** and so on.

Many Blipkit models are intended to work in concert with *ontologies*, computable representations of the types of entity in some domain. The Open Biological Ontologies (OBO) project organizes and stratifies the various different ontologies used in bioinformatics[17] into orthogonal domains, and the blipkit **ontol_db**model is geared towards these ontologies, and is based on the OBO language. This model includes extensional predicates such as **subclass/2**, which represents the *is_a* relationship[16] that can hold between two ontology classes. The intensional predicate **subclassT/2** is the transitive version of this predicate, defined recursively in the standard fashion and resolved through WAM-based backward chaining. The lack of tabling in SWI-Prolog can be problematic here; certain kinds of inferences will lead to cycles. Two alternate strategies are employed - one strategy is to write the facts to a file and use a different set of intensional predicates within a Prolog such as Yap or XSB. In general it is preferable to do the inferences directly from SWI-Prolog, so Blipkit also provides a forward chaining inference engine called **ontol_reasoner**. This iteratively applies rules to the existing database, asserting new facts until no new facts can be inferred. Expressivity is limited in that rules which produce infinite or prohibitively large databases are avoided.

One of the most popular ontologies in Bioinformatics is the Gene Ontology (GO)[3], which is used to assign functions and cellular locations to the products of genes. This adds semantics to gene databases, as it allows computable statements such as “this gene encodes a product that regulates the transcription of other genes” or “this gene encodes a product that produces the neurotransmitter dopamine”. The **curation_db**model is used to represent these assignments, together with associated provenance.

The **ontol_db**module is also used to implement Obol grammars[13]. These provide a configurable means of automatically translating between biologist-friendly textual descriptions and formal logical expressions using Definite Clause Grammars (DCG), and have proved invaluable for the GO[12].

For example, the following portion of a DCG can be used to relate the tokenization of the string “permeability of mitochondrial membrane” to a prolog expression that can be reasoned over:

```
phenotype( A that attribute_of( C ) ) --> physical_attribute(A), [of],object(C).
object( membrane that surrounds(C) ) --> relational_adjective(C), [membrane].
```

2.3 Genomes

Every living organism has a genome - a blueprint for life, carried by each cell in that organism, encoded as a discrete sequence of chemicals called bases arranged along a backbone of DNA (Deoxyribonucleic Acid). There are four types of bases used, conventionally abbreviated as A, C, G and T, and these can be thought of as the symbols of a molecular alphabet life uses to feed instructions to the cellular machinery The genome sequence is passed on from generation to generation, with successive modifications and rearrangements giving rise to new phenotypes (organismal characteristics) and ultimately new species.

The layout of information along a genome sequence can be difficult to reverse engineer. The fundamental units are *genes*, genome subsequences which encode the molecular agents that give rise to myriad biological processes. In many organisms, these gene sequences are often split into subsequences called *exons*, interspersed with subsequences called *introns* (figure 1). The introns are later excised by the cellular machinery. Whilst this may seem wasteful, it serves many functions, one of which is to allow different combinations of exons of the same gene to come together in vary contexts.

The blipkit **genomic** package contains a number of modules for representing genome sequences and manipulating those representations. The architecture of a genome can be conceived of in terms of discrete intervals. The inter-relationships of these intervals can be formalized using the Allen Interval Calculus[2]. The **biorange** module contains predicates for determining the relationship between two intervals. This module extends the interval calculus as it deals with *reverse complementation* – every DNA molecule has two complementary strands going in reverse directions. A base on one strand is paired with a complementary base on the other strand.

This module also includes relationships between discontiguous sets of intervals, such as **interleaves/2**. These kinds of complex relationships are often observed in higher organisms - figure 1 shows two interleaved genes on opposite strands.

There are many other types of features encoded along a genome sequence - in fact there is an entire ontology dedicated to these, the Sequence Ontology[9][10]. Other important feature types include various classes of regulatory region - these provide cellular context for the switching on and off of genes as part of complex regulatory networks. If genes are activated at the wrong time the consequences can be disastrous - one illustration of this is in the fruitfly version of the human

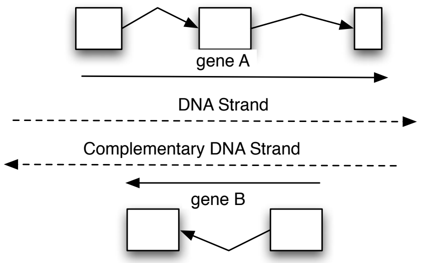


Fig. 1. Two genes, each denoted by an arrow, interleaved on opposite strands of DNA (dotted lines). When the gene is transcribed by the cell, the introns (bent lines) are excised and the exons (boxes) are joined together. Different combinations (called transcripts) are possible in different contexts (for example, the middle exon of gene A may be omitted).

PAX6 gene (implicated in eye conditions in humans). Inappropriate expression of this gene leads to the insect growing eyes in the wrong places such as on wings and legs.

Fast retrieval of features within a given range is a common operation. There are two implementations of this: the first is a pure prolog implementation, the second implementation is in C and uses the Nested Containment List argorithm[1]. The SWI-Prolog Foreign Language Interface (FLI) is used to wrap this such that it is seamlessly accessible via prolog.

The **biorange** module ignores the actual bases themselves and treats genomic features as intervals. The **bioseq** module has various predicates for performing operations on sequences, including excising subsequences and finding the complement of a sequence. These sequence operations are very standard for any kind of bioinformatics software library.

Neither of these modules are models, according to how blipkit partitions packages. The predicates do not lookup facts in the prolog database, they operate entirely on the arguments supplied as input.

There are two complementary models for representing genomic features: **genome_db**and **seqfeature_db**; in addition there is a largely orthogonal model called **seqanalysis_db**. In addition, there is a module for performing operations on genomic intervals, **range**, a module for handling sequences, **bioseq**.

The **genome_db**model contains a *direct* prolog representation of the elements of a genome, with unary predicates corresponding to the major types of feature (as represented in the Sequence Ontology), as well as extensional n-ary predicates for representing the relationships between features. The collections of facts below corresponds to the upper part of figure 1

```
gene(geneA).
exon(geneA_exon1).
exon(geneA_exon2).
exon(geneA_exon3).
dna(dnaseq1).
transcript(geneA_transcriptX).
exon_transcript_order(geneA_exon1,geneA_transcriptX,0).
exon_transcript_order(geneA_exon2,geneA_transcriptX,1).
exon_transcript_order(geneA_exon3,geneA_transcriptX,2).
exon_dnaseq_pos(geneA_exon1,dnaseq1,1000,2000,1).
exon_dnaseq_pos(geneA_exon2,dnaseq1,3000,4000,1).
exon_dnaseq_pos(geneA_exon2,dnaseq1,6000,6500,1).
```

Note that there are no facts for the introns (gaps between exons). Given any two successive exons on a transcript, we can *infer* the existence of an intron between them, as well as the position of that intron. The intensional predicate **intron/1** provides this inference, using a skolem term as intron identifier:

```
intron( intron(Seq,End,Beg,Str) ):-
        exon_transcript_order(X1,T,R1),
        exon_transcript_order(X2,T,R2),
        1 is R2-R1,
        exon_dnaseq_pos(X1,Seq,_Beg,End,Str),
        exon_dnaseq_pos(X2,Seq,Beg,_End,Str).
```

The arguments of the skolem term can be used in further intensional predicates for inferring the position of the inferred intons. A number of other feature types are inferred using intensional predicates (not shown here for brevity). This kind of inference may seem simple to a logic programming expert, but in fact it turns out to be tremendously useful for querying genome databases, where incomplete information is the norm. In bioinformatics, a little inferencing can go a long way.

One limitation with the current inference model is that the decision as to which feature types are extensional and which are intensional must be made a priori. Whilst it is more commonly the case (for example) that introns need to be inferred from exons, occasionally the situation is reversed. If we attempt to mutually define exons and introns in terms of one another, then it is difficult to use an untabled Prolog without the execution of queries getting caught in infinite loops.

A number of options are being explored here. One option is to use a Prolog which allows tabling, such as Yap or XSB. Currently this requires manual porting of code, as Blipkit is SWI-specific. So far there is an extension to the **genome_db**that works with Yap and allows for a greater variety of inferences, and allows for greater flexibility in what features are stated and which are inferred. However, at this time the additional modules of Blipkit have not been ported, which limits the potential for powerful integrative cross-domain queries.

Another option simultaneously being explored is to use an even more expressive logical modeling paradigm such as Answer Set Programming (ASP) and Disjunctive Datalog. There is a separate implementation of the **genome_db**model written for the DLV system, part of an extension of core Blipkit. Core DLV lacks the ability to have compound terms as predicate arguments, which means that the intensional predicate for intron shown above cannot be used. Fortunately there is an extension to DLV called DLV-complex which does allow for these more expressive rules. The use of DLV opens up other possibilities, such as disjunctions in the head of rules, and the ability to specific constraints on the model. This is extremely useful in the context of incomplete or incorrect genomic information (a common scenario). Of course, DLV is not a Prolog system so the rest of Blipkit cannot be used unless it too is converted.

In contrast to **genome_db**, the **seqfeature_db**model contains an *indirect* representation of the elements of a genome using a generic **feature/1** predicate, with the type of the feature represented using a binary **feature_type/2** predicate, the second argument of which is the name of the type taken from the Sequence Ontology. In other words, types are *reified* (i.e. they are predicate *arguments* and are thus part of the domain of discourse), in contrast to **genome_db**, where the types are directly *realized* in the prolog model as predicates. There is a *bridge* module for converting between these two representations.

2.4 RNA Molecules

Despite the inherent 3-dimensionality of DNA molecules, their sequences are essentially linear and 1-dimensional. The cellular machinery transcribes genes encoded along the DNA molecule into RNA molecule transcripts which assume 3-dimensional configurations. These RNA molecules were once assumed to be passive intermediaries in between protein coding genes and protein molecules. However, more recently attention has been focused on RNA genes, as these molecules are shown to play critical roles in the cell, and have been implicated in a number of diseases. In fact it has even been hypothesized that the origin of all life was the “RNA world”.

An RNA molecule, like a DNA molecule, can be modeled as a discrete sequence of bases. However, additional interactions between the bases become more significant.

The relationships between bases are modeled in **structure_db**using a hierarchy of binary relations such as **five_prime_to/2** and **paired_with/2**, derived from the RNA Ontology[4]. We can then define intensional predicates to infer the presence of higher level relationships and features such as bulges and loops.

Figure 2 shows an RNA pseudoknot, the existence of which can be inferred using the intensional predicate printed below.

```
pseudoknot(X):-
        stemloop(SL1),
        stemloop(SL2),
        stem(S2),
        loop(L1),
        has_part(SL2,S2),
        has_part(SL1,L1),
        part_of(S2,L1),
        mereological_union(SL1,SL2,X).
```

This is another area in which tabling is extremely useful. The RNA module of Blipkit works best with Yap Prolog.

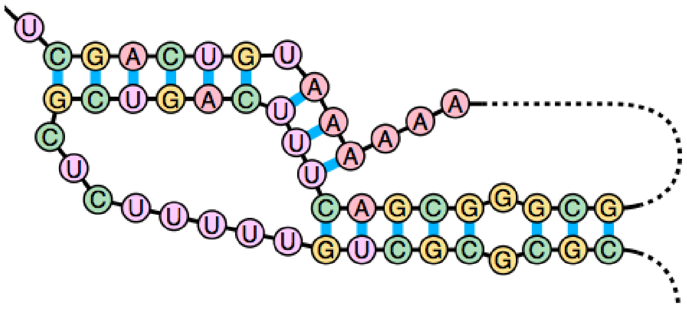


Fig. 2. RNA Pseudoknot

2.5 Phylogenetics

A phylogenetic tree is a representation of the evolutionary relationships between either a collection of species, molecules or molecular sequences. Unlike a taxonomic tree, the branches or edges of a phylogenetic tree are labeled with distance quantities.

The blipkit **phylo_db**model is used for representing phylogenetic relationships. The extensional predicates correspond to the branches of the tree and their lengths, together with additional metadata about the nodes (for example, taxonomic identifiers). Intensional predicates are used for derived information, such as transitive ancestors and cumulative branch distances. There are additional intensional predicates for comparing two phylogenetic trees.

Gene or protein trees are more complex than organismal phylogenetic trees as both speciation and duplication events have to be taken into account. The latter happens when the genome of an organism (in whole or in portion) is duplicated within the cells of that organism, and passed on to its ancestors. This can be crudely thought of as initially comprising a "backup copy" of the genome, which then affords more freedom for other genes to vary and evolve. The **phylo_db**model includes intensional predicates for inferring the relationship between two genes based on whether the splitting event was a speciation or a duplication.

Two popular formats for exchanging phylogenetic trees are New Hampshire format and PhyloXML. Blipkit supplies a DCG for the former and an XML mapping for the latter. The Nexus format is more complex, and has a pre-existing Prolog parser that is being adapted for use with Blipkit.

One limitation of the **phylo_db**model is that the structure of the tree is modeled as extensional predicates. The trees are themselves inferred, and this inference could be performed in Prolog. One advantage of doing this in Prolog is to combine information about shared features encoded using ontologies[11]. For now, it is assumed that tree inference is performed outside blipkit, which justifies the use of extensional predicates.

2.6 Systems Biology and Biological Interactions

The study of genes in isolation is by definition reductive. Systems biology aims to take a more holistic approach, looking at genes in the context of the function of other genes in the cell, and thereby study the emergent properties that underly living systems. In practice this usually takes the form of studying biological pathways and the interactions between genes, gene products and chemical entities within and between cells. The Blipkit **sb** package provides functionality for dealing with systems biology data.

Blipkit includes the **sb_db**model, strongly influenced by the SBML standard. SBML is an XML-based exchange format for representing cellular interactions in a quantitative fashion. SBML can be imported and exported by many simulation tools. SBML also includes the ability to annotate the model using OBO ontologies with RDF.

Like SBML, **sb_db**has a reaction-centric view, and thus has predicates **reaction_reactant/2**, **reaction_product/2** and **reaction_modifier/2** for representing the inputs, outputs and agents in a biochemical reaction event. In SBML terminology, these are all "species". SBML allows the expression of complex formula via embedded MathML. In **sb_db**these have their cognate representations as nested Herbrand terms.

At this time, there is no quantitive modeling built in to Blipkit. However, it is possible to perform simple qualitative modeling – for example, the **species_path/3** intensional predicate allows us to ask if there is a chain of reactions that allows species A to be produced from species B.

Another format commonly used is the RDF-based BioPAX. Data expressed in BioPAX can either be directly translated to the **sb_db**model using the provided Blipkit bridge module (which itself uses the SWI-Prolog RDF library). There is an alternative model which is a direct automated translation of the BioPAX schema (expressed in OWL) to Blipkit extensional predicates. The programmer has the option of choosing, depending on which model provides the best view for the questions they wish to ask.

Yet another model within the **sb** package is the **interaction_db**model. This provides a simplified view in terms of a binary **interacts_with** extensional predicate. Databases such as BioGRID[6] aggregate protein-protein interaction data from a variety of sources. This data can be extremely powerful when combined with other data types.

3 Integration with Relational Databases

Bioinformatics is an information science, and much of the relevant information is stored in relational databases. Sometimes these databases are supplied with an API (accessed via a specific programming language or as a web service) that allows for programs to access data. However, these APIs typically do not allow complex boolean expressive queries, as is possible with direct SQL queries. Fortunately, many of the important bioinformatics databases make their data available as database dumps, or have an open SQL port through which queries can be made remotely. For example, the ENSEMBL database system[5] has genomic annotation data for most sequenced genomes and is thus crucially important for bioinformatics analyses.

The pure subset of prolog is in part an extension of the relational model, and the methods for mapping prolog predicates to relational databases have been known for nearly twenty years[8]. Using the **sql_compiler**code written by Christoph Draxler, it is possible to automatically translate complex prolog goals into SQL queries. This stands in contrast to the more commonly used imperative languages based on object-oriented principles, in which the impedance mismatch between the two formalisms is known to be problematic[7].

Draxler's **sql_compiler**code has been adapter to work with various Vendor-specific prologs such as Ciao, YAP and XSB. This has also been adapted to SWI-Prolog as part of the Blipkit library. The adaptation also introduces numerous novel extensions to the original code, including:

- **Database connectivity.** The original **sql_compiler**writes out SQL rather than making queries to the database. This functionality is still retained, but additional functionality is added using the SWI-Prolog ODBC library to connect directly to a database and translate the result sets back into prolog terms. In addition, prolog predicates can be bound to a database handle, such that prolog execution of the goals will call the database "behind the scenes". Similar capabilities are available in some of the other adaptations such as the one in Ciao prolog.
- **Query optimization.** The original **sql_compiler**can produce some inefficient SQL queries for complex goals. By adding additional metadata about the schema, particular unique key declarations, the Blipkit **sql_compiler**can rewrite queries to avoid redundant joins, resulting in faster queries.
- **Query rewriting based on prolog clauses.** In the original **sql_compiler**, all the terminal subgoals in the compiled goal must correspond to tables in the relational schema. The Blipkit **sql_compiler**will rewrite subgoals that correspond to prolog intensional predicates. This is in fact quite simple to do due to the introspectability of prolog, and it turns out to be a powerful feature, as the same prolog rules can be re-used in both in-memory contexts and relational database contexts. The one proviso is that the prolog rules are non-recursive pure prolog.
- **Mapping to SQL Functions.** Prolog functions such as **sub_atom/5** can be translated to SQL functions such as **substr**. This increases interoperability between the prolog code and SQL.

Blipkit includes both prolog metadata on a number of bioinformatics relational schemas such as Ensembl and Chado[14], and mapping modules that translate between the schema and models. These are shown in table 2.

The presence of mapping modules means that the same prolog model can be used regardless of the underyling database schema. For example, the **genome_db**model includes an intensional predicate called **gene_overlaps/2** to test if two genes overlap on the same DNA strand. This is defined as follows:

```
gene_overlaps(G1,G2):-
        gene_dnaseq_pos(G1,Seq,Beg1,End1,Str),
        gene_dnaseq_pos(G2,Seq,Beg2,End2,Str),
        End1 >= Beg2,
        Beg1 =< End2.
```

If blipkit is configured with this predicate bound at an Ensembl instance, then calls to **gene_overlaps/2** will be translated behind the scenes to SQL queries involving joins over multiple tables and executed against a remote Ensembl server, with the results bound non-deterministically to prolog variables. The exact same predicate definition can also be used with in-memory prolog databases.

Neither of the two main open source database vendors provide means of executing recursive SQL. To get round this limitation, Blipkit allows the use of the **ontol_db**module to do some semantic query expansion. The strategy is to perform the recursive part of the query first in Prolog, and then feed the results

Table 2. Mappings between relational schemas and prolog models

Schema	Mapping	Model	Package
Chado	**seqfeature_sqlmap_chado**	**genome_db**	genomic
Ensembl-core	**genome_sqlmap_enscore**	**genome_db**	genomic
Ensembl-compara	**phylo_sqlmap_enscompara**	**phylo_db**	phylo
GODB	**homol_sqlmap_go**	**homol_db**	phylo
GODB	**ontol_sqlmap_go**	**ontol_db**	ontol

back in to the SQL query. This is all taken care of behind the scenes in the mapping module.

4 Integrating with XML and Web Services

Many biological databases and analysis programs are available as Web Services. The National Center for Biotechnology Information (NCBI) make a number of core databases and services available through eUtils[15].

The SWI-Prolog **http_client** library makes it simple to access these services programmatically. Blipkit contains a number of pre-written web wrapper modules for a variety of services. These include all the databases at NCBI, including sequence databases and the PubMed service. There are also wrappers for caBIG and Cancergrid.

In addition to these bioinformatics services, there are also wrappers for querying Google, Yahoo and Wikipedia. These wrappers also provide the ability to perform semantic query expansion using ontologies. For example, a search for "neurotransmitter" will be expanded to the various subtypes of neurotransmitter such as dopamine and serotonin.

Many web services return results in XML. In addition, many databases provided static XML dumps. SWI-Prolog provides the **sgml** module for parsing XML, which translates XML documents into nested Herbrand terms. Mapping code that translates between these XML terms and models can be quite verbose, so Blipkit has an XSLT-inspired library called **xml_transform** that can be used to quickly write mapping code. This illustrates how prolog is a suitable language for writing DSLs (Domain Specific Languages).

5 Integrating with Ontologies and the Semantic Web

The native ontology module in Blipkit is based around the OBO model, commonly used in Bioinformatics. Information is also increasingly available as OWL ontologies, or as RDF triples.

For RDF datasources, SWI-Prolog provides the **semweb** package[25] which allows for fast parsing of RDF sources. This is used for mapping some external datasets, such as those encoded using BioPAX, into Blipkit models.

Even though the Web Ontology Language OWL can be layered on RDF, it turns out that triples often provide a poor level of abstraction for complex class expressions. Here we use the Thea library, which exposes OWL axioms directly as Prolog predicates. Thea is being extended to handle OWL2[22].

In addition, Blipkit provides a parser for the Common Logic Interchange Format (CLIF), a lisp-like syntax that is used for encoding any collection of first order logic sentences.

6 Presentation Layers

6.1 Visualization of Graphs

Network-style graph views are common for visualizing complex biological data, such as interaction networks and ontological classification of genes. Blipkit provides two means of visualizing such data, the first through GraphViz, and the second through XPCE.

GraphViz takes an abstract specification of a graph and uses a layout algorithm to render it such that the nodes are placed as optimally as possible, minimizing edge-crossings. The specification is in the form of the dot language, a domain specific language for graphs. Blipkit includes a dot grammar, for generating dot files, given a prolog representation of a graph. In addition, graphs can be configured using simple configuration predicates, for example, edges of certain types can be colored or even nested.

One limitation of GraphViz is that the representation is static, the graph cannot be manipulated, unless the dot file is imported via another program. The alternate means of visualization graphs is via a bridge to XPCE, which allows nodes to be dragged, and allows the graphs to be embedded in larger XPCE applications. The main limitation here is that the XPCE canvas does not have a layout algorithm, placement of nodes is arbitrary, leading to unsightly edge crossings. One possible solution would be to use dot to perform the layout, and then use this to guide placement within XPCE.

6.2 The Serval Web Application Framework

An increasing number of applications use the web browser as a platform. For prolog applications, this has the advantage that the end user does not need to install prolog on their machine.

SWI-Prolog includes the powerful **http** package which greatly simplifies the task of writing web applications. However, this package does not include any specific capabilities for translating HTML. The general paradigm is to generate XHTML terms using a DCG, which introduces extra syntax that can obscure the relationship between the code and output.

The paradigm used by most web application frameworks, such as the popular Ruby on Rails is to use a template language. This introduces an additional language, with possible language mismatch problems. Blipkit takes a different approach with the bundled Serval module, which uses a scheme-like functional-style language with prolog syntax to specify dynamic HTML.

To give a trivial example, the following piece of code will generate a nested HTML list (built with **ul** and **li** HTML elements) for a phylogenetic tree in the **phylo_db**model, by recursively traversing down the tree.

```
phylohtml(Node) =>
  if(phylonode_leaf(Node),
      then: b(Node),
      else:
       ul(li( phylohtml(Child) ) forall phylonode_parent(Child, Node))).
```

The serval framework allows for the specification of state machine like rules for determining how transitions are made from one web page to another. The above code can be extended to allow for a dynamic tree in which nodes can be expanded and collapsed by the user.

A number of Blipkit applications are available as web applications and web services. One such application is an ontology browser and visualization tool[2].

7 Composing Models and Complex Queries

Whilst each Blipkit package is useful as a standalone piece of software, the real power comes from composing queries across multiple models and data sources. For example, using the **genome_db**model and the bridge to the Ensembl database we can query for all upstream regulatory regions for a given gene, or discoverer which genes make which proteins. Using the **ontol_db**module and an ontology of chemical entities we can find all the different kinds of neurotransmitter. Using both Gene Ontology annotations and pathway databases we can find all proteins that produces or transport those neurotransmitters. We can combine all these as follows:

```
entity_label(NT,neurotransmitter),
subclassT(Entity,NT),
(produces(Protein,Entity) ; transports(Protein,Entity)),
encodes(Gene,Protein),
regulates(Reg,Gene),
upstream_of(Reg,Gene)
```

If blipkit is configured such that the relevant predicates query the correct sources, then this will find the regulatory regions for genes whose expression determines the identity of different kinds of neurotransmitter secreting cells.

Some of the subgoals will query the in-memory prolog database using intensional predicates; others will be translated to relational queries. Unlike a relational query, the ordering of subgoals is important, as they are executed sequentially.

One possible future extension would be perform query optimization and reorder subgoals within a goal.

[2] http://berkeleybop.org/obo/

8 Discussion

8.1 Comparison with Other Bioinformatics Toolkits

The Open Bioinformatic Foundation (OBF) is a non-profit organization that acts as an umbrella for the open source Bio* projects. The Bio* projects include BioPerl, BioJava, BioPython, BioRuby and BioSQL, each representing a community effort to provide a relatively comprehensive library of code for researchers in the life sciences that takes advantages of the features of the host programming language. The longest running and perhaps most comprehensive of these projects is BioPerl[18], as perl is a popular language due to it's string matching and manipulation capabilities. Blipkit, being part of the OBF, an alias for BioProlog. It thus makes sense to compare Blipkit/BioProlog with some of the other Bio* projects such as BioPerl.

The organization and architecture of Blipkit is similar to, and influenced by BioPerl. BioPerl is divided into sub-modules each dealing with a separate subdomain of biology. These are then further divided into object-oriented modeling classes, and additional classes for parsing and writing a variety of data formats. The libraries are intended to be used in similar ways, as tools for investigators to ask questions of complex data, and also as core components within larger infrastructures.

Of course, BioPerl is far more comprehensive due to a large critical mass of developers and contributors, especially in the domain of genomics. However, Blipkit offers many capabilities absent from BioPerl, such as modules for systems biology data. For the set of capabilities in the intersection between the two systems, the use of Prolog can offer a more concise declarative way to perform operations. In many cases, prolog can offer a faster execution time, which can be surprising to some people given the reputation of Prolog as an academic language.

8.2 Comparison with Semantic Web Technology

An alternative approach to semantic data integration is to use semantic web technology to query across multiple databases. Databases can be mapped to RDF triples, either dynamically, using a system such as D2RQ, or statically, by building a native RDF triplestore. However, RDF on its own does not provide any of the "semantics" in the semantic web. For that there has to be some kind of inference, usually driven by ontologies and ontological formalisms such as RDFS or OWL.

RDFS on its own does not provide enough semantics for complex biological data. OWL is more expressive, but most OWL reasoners do not scale to large databases. Another issue is that the expressivity afforded by OWL and Description Logics in general may not be quite right for all bioinformatics applications.

For example, Description Logics are unable to represent accurately represent cyclic classes of structure, such as carbon rings, regulatory networks or RNA structures. For example, the following logic programming predicate can classify

RNA tetraloops - chains of 4 bases, each end of which is connected to bases that are themselves paired.

```
tetraloop([B,C,D,E]) :-
  chain([A,B,C,D,E,F]),
  paired_with(A,F).
```

The equivalent is not possible as an OWL class expression. One possibility is to go beyond OWL and use the Semantic Web Rules Language (SWRL), which is comparable in expressive power to Datalog. One crucial difference is that all semantic web formalisms make the open world assumption, whereas the Datalog family of languages from the relational model up through the pure subset of prolog up to answer set programming makes the closed world assumption. The open world assumption can be useful when making inferences over the web, but it can prove to be a hindrance with bioinformatics since it can often be assumed that certain classes of data are complete.

8.3 Blipkit Issues and Limitations

The main limitation of Blipkit is the lack of expressivity that comes from the lack of predicate tabling. However, for a great many uses there are workarounds, so in practice this often does not prove to be an unsurmountable problem. However, some of these workarounds are inelegant, tabling would result in a smaller codebase and cleaner, more declarative code.

One of the problems with blipkit is that it is too big. Ideally it would consist entirely of biolog-specific code, as it is there are lots of general purpose modules that belong either in independent projects or even in some kind of prolog common library. This applies especially to the **sql_compiler**code.

8.4 Expressivity and Suitability of Prolog

For the majority of the tasks within the scope of Blipkit, the expressivity provided by Prolog is mostly sufficient. The declarative nature of the language makes it well-suited to answering complex queries. In addition the programmability of Prolog means that fully fledged applications can be constructed entirely in prolog, avoiding any impedance mismatches that often result from connecting together different technologies.

As mentioned, WAM type resolution can often be a limitation. In addition there are cases where it is useful to go beyond the expressivity of Prolog, for example, disjunctive datalog or nonmonotonic reasoning, as found in systems like DLV. It would be useful to have a more seamless way of reusing portions of the same programs across logic programming systems of varying expressivity. Currently this is impeded by a lack of standards in these areas.

Currently the types of inference offered by Blipkit are purely logical. Often the final step of semantic data integration is some kind of statistical or probabilistic modeling. In the future a number of options will be explored. One is tighter integration with libraries developed using the statistical programming language R, such as BioConductor. Another is the use of formalisms that allow a more seamless integration of logical and statistical programming. This includes bayesian CLP, inductive logic programming and bayesian probabilistic logical modeling, as provided in the PRISM system.

8.5 Portability

Different prolog implementations offer different advantages. Ideally the same library code could be reused across all implementations. Unfortunately Blipkit is tied to one Prolog system, SWI-Prolog.

Historically, the initial implementation of Blipkit was in XSB Prolog. XSB offers powerful deductive database capabilities, including tabling. The decision was made to switch to SWI-Prolog due to a number of factors such as the more extensive collection of libraries and development tools, an active mailing list, frequent releases and ease of installation on a number of different operating systems. The original goal was to maintain a compatibility layer, but this fell by the wayside, and the system quickly become SWI-specific.

SWI-Prolog proved to be an excellent environment, but the lack of tabling proved to be a hindrance. Certain parts of the code had to be rewritten to avoid cycles in the deduction chain, which resulted in less compact code. Some functionality is simply unavailable. Blipkit does include a module for performing memoization on calls to goals (misleadingly called the **tabling** module, in fact much less powerful than full tabling). This can increase efficiency in certain contexts, but predicates must still be written to avoid cycles. Blipkit also has a module for forward chaining, performing the full deductive closure, but this can only be used in certain contexts, and can entail a large upfront time-penalty.

Recent efforts to improve interoperability between Yap and SWI are extremely encouraging. Hopefully it will soon be possible to use Blipkit from within Yap, which will open the possibility of using tabling, available in Yap.

One of the main sources of incompatibility between prolog systems is difference in module systems. One possibility is to refactor Blipkit to use LogTalk as a means of providing encapsulation between different modules.

8.6 Large Datasets and Relational Databases

Blipkit is in some sense a large modular multi-purpose deductive database schema. The core is largely Datalog, with additional utility layers in Prolog. Prolog and Relational Database Management Systems (RDBMSs) have many similarities, but this similarity is often under-exploited.

To say that RDBMSs are more prevalent would be a massive understatement. Yet RDBMs could benefit from the addition of logic programming features. The SQL99 standard allows for a very limited kind of deductive predicate, and this remains unimplemented in the two major open source RDBMSs. Programming with RDBMSs is frequently difficult: procedural SQL is unwieldy, and use of object oriented languages introduces an impedance mismatch. RDBMSs deserve a relational language, along the lines of Prolog. Prolog can also be a more “Agile” alternative to full fledged database systems – a prolog database can be located anywhere on the filesystem. In this respect Prolog shares some features with lightweight databases such as SQLLite.

On the other hand, prolog systems could benefit from having additional optional RDBMS like features. The lack of schemas or typing in Prolog is often a boon, but sometimes it would be useful to have optional metamodeling facilities. The programmer is of course free to roll their own (as in for example the Blipkit **dbmeta** metamodel) but it would be better to have this as a standard library.

Better ways of making seamless transitions between the two would also be welcome.

9 Conclusions

Integration of data across multiple databases remains a difficult problem, limiting the number of and scope of questions that researchers can ask. Despite having a critical mass of developers, the Blipkit library has slowly evolved to have a core set of models and utility modules that make it useful for a wide variety of powerful queries and data extraction operations. In particular, the ability to combine inference rules, relational databases and ontologies with a collection of pre-written mappings make it especially powerful.

Blipkit would not have been possible without the SWI-Prolog system. In particular, the comprehensive set of libraries offered by SWI-Prolog make it simple to develop full web applications and web services using Blipkit. The extension of Blipkit may require a hybrid solution, using different Prolog implementations and logic programming systems in general depending on the particular task at hand. The major barrier here is the lack of interoperability between logic programming systems.

Acknowledgements

Thanks to Stephen W Veitch for the development of the NCL library, Vangelis Vassiliadis for assistance with the Thea OWL library, and Jan Wielemaker for SWI-Prolog. Also many thanks to Nicola Leone and the DLV teams for assistance with the DLV system.

References

1. Alekseyenko, A.V., Lee, C.J.: Nested containment list (NCList): a new algorithm for accelerating interval query of genome alignment and interval databases. Bioinformatics, btl647 (2007)
2. Allen, J.F.: Maintaining knowledge about temporal intervals. Communications of the ACM 26, 832–843 (1983)
3. Ashburner, M., Ball, C.A., Blake, J.A., Butler, H., Cherry, J.M., Corradi, J., Dolinski, K., Eppig, J.T., Harris, M., Hill, D.P., Lewis, S., Marshall, B., Mungall, C., Reiser, L., Rhee, S., Richardson, J.E., Richter, J., Ringwald, M., Rubin, G.M., Sherlock, G., Yoon, J.: Creating the gene ontology resource: design and implementation. Genome. Res. 11(8), 1425–1433 (2001)

4. Batchelor, C., Bittner, T., Eilbeck, K., Mungall, C., Richardson, J., Knight, R., Stombaugh, J., Zirbel, C., Westhof, E., Leontis, N.: The rna ontology (rnao): An ontology for integrating rna sequence and structure data. In: Proceeeings of the First International Conference on Biomedical Ontology (2009)
5. Birney, E., Andrews, T.D., Bevan, P., Caccamo, M., Chen, Y., Clarke, L., Coates, G., Cuff, J., Curwen, V., Cutts, T., Down, T., Eyras, E., Fernandez-Suarez, X.M., Gane, P., Gibbins, B., Gilbert, J., Hammond, M., Hotz, H.R., Iyer, V., Jekosch, K., Kahari, A., Kasprzyk, A., Keefe, D., Keenan, S., Lehvaslaiho, H., McVicker, G., Melsopp, C., Meidl, P., Mongin, E., Pettett, R., Potter, S., Proctor, G., Rae, M., Searle, S., Slater, G., Smedley, D., Smith, J., Spooner, W., Stabenau, A., Stalker, J., Storey, R., Ureta-Vidal, A., Woodwark, K.C., Cameron, G., Durbin, R., Cox, A., Hubbard, T., Clamp, M.: An overview of ensembl. Genome. Res. 14(5), 925–928 (2004)
6. Breitkreutz, B.-J., Stark, C., Reguly, T., Boucher, L., Breitkreutz, A., Livstone, M., Oughtred, R., Lackner, D.H., Bhler, J., Wood, V., Dolinski, K., Tyers, M.: The biogrid interaction database: 2008 update (PMC2238873). Nucleic Acids Research 36, D637–D640 (2008)
7. Cook, W.R., Ibrahim, A.H.: Integrating programming languages and databases: What is the problem. ODBMS. ORG, Expert Article (2006)
8. Draxler, C.: Accessing Relational and Higher Databases Through Database Set Predicates. PhD thesis, PhD thesis, Zurich University (1991)
9. Eilbeck, K., Lewis, S.E., Mungall, C.J., Yandell, M.D., Stein, L.D., Durbin, R., Ashburner, M.: The sequence ontology: a tool for the unification of genome annotations. Genome. Biology 6(5) (2005)
10. Eilbeck, K., Mungall, C.: Evolution of the sequence ontology terms and relationships. In: Proceedings of the First International Conference on Biomedical Ontology (2009)
11. Mabee, P.M., Ashburner, M., Cronk, Q., Gkoutos, G.V., Haendel, M., Segerdell, E., Mungall, C., Westerfield, M.: Phenotype ontologies: the bridge between genomics and evolution. Trends. Ecol. Evol. (April 2007)
12. Mungall, C., Bada, M., Berardini, T., Deegan, J., Ireland, A., Harris, M., Hill, D., Lomax, J.: Cross-product extensions of the gene ontology. In: Proceedings of the First International Conference on Biomedical Ontology (2009)
13. Mungall, C.J.: Obol: Integrating language and meaning in bio-ontologies. Comparative and Functional Genomics 5(7), 509–520 (2004)
14. Mungall, C.J., Emmert, D.B.: A chado case study: an ontology-based modular schema for representing genome-associated biological information. Bioinformatics 23(13), 337–346 (2007)
15. Sayers, E., Wheeler, D., National Center for Biotechnology Information: Building customized data pipelines using the entrez programming utilities (eutils). In: NCBI (2004)
16. Smith, B., Ceusters, W., Kohler, J., Kumar, A., Lomax, J., Mungall, C.J., Neuhaus, F., Rector, A., Rosse, C.: Relations in biomedical ontologies. Genome. Biology 6(5) (2005)
17. Smith, B., Ashburner, M., Rosse, C., Bard, J., Bug, W., Ceusters, W., Goldberg, L.J., Eilbeck, K., Ireland, A., Mungall, C.J., Leontis, N., Rocca-Serra, P., Ruttenberg, A., Sansone, S.-A., Scheuermann, R.H., Shah, N., Whetzel, P.L., Lewis, S.: The obo foundry: coordinated evolution of ontologies to support biomedical data integration. Nat. Biotechnol. 25(11), 1251–1255 (2007)

18. Stajich, J.E., Block, D., Boulez, K., Brenner, S.E., Chervitz, S.A., Dagdigian, C., Fuellen, G., Gilbert, J.G., Korf, I., Lapp, H., Lehvaslaiho, H., Matsalla, C., Mungall, C.J., Osborne, B.I., Pocock, M.R., Schattner, P., Senger, M., Stein, L.D., Stupka, E., Wilkinson, M.D., Birney, E.: The bioperl toolkit: Perl modules for the life sciences. Genome. Res. 12(10), 1611–1618 (2002); 1088-9051 Journal Article
19. Stein, L.: How perl saved the human genome project. The Perl Journal 1(0001) (1996)
20. Stein, L.: Creating a bioinformatics nation. Nature 417(6885), 119–120 (2002)
21. Stein, L.D.: Integrating biological databases. Nature Reviews Genetics 4(5), 337–345 (2003)
22. Vassiliadis, V., Mungall, C.J.: Logic programming with owl2 using the thea prolog library (in preparation, 2009)
23. Wielemaker, J.: An overview of the SWI-Prolog programming environment. In: 13th International Workshop on Logic Programming Environments, pp. 1–16 (2003)
24. Wielemaker, J., Anjewierden, A.: PlDoc: wiki style literate programming for prolog. In: Proceedings of the 17th Workshop on Logic-Based methods in Programming Environments, p. 1630 (2007)
25. Wielemaker, J., Schreiber, G., Wielinga, B.: Prolog-based infrastructure for RDF: Scalability and performance. In: Fensel, D., Sycara, K., Mylopoulos, J. (eds.) ISWC 2003. LNCS, vol. 2870, pp. 644–658. Springer, Heidelberg (2003)

A Knowledge Base System Project for FO(.)

Marc Denecker

Department Computer Science, Katholieke Universiteit Leuven
Celestijnenlaan 200A, B-3001 Heverlee, Belgium
marc.denecker@cs.kuleuven.be

Abstract. This talk reports on a project to build a Knowledge Base System (KBS) equipped with several forms of inference able to solve different sorts of tasks using the KB. The logic FO(.) used in the KBS is an extension of classical logic (FO) with various language primitives such as inductive definitions, aggregates, arithmetic, etc. The logic is a natural integration (and further extension) of classical logic and logic programming, and is based on the view of a logic program as a definition. We discuss informal and formal semantics of definitions in FO(.) and consider the relationship with other knowledge principles such as coinduction, the closed world assumption and causality and with the LP formalisms ASP, ALP and deductive databases. On the computational level, we will report on current attempts to build finite domain inference systems for model expansion, approximate reasoning, theory debugging and model revision, with special focus on the IDP-system, a model expansion system for FO(.).

From Plain Prolog to Logtalk Objects: Effective Code Encapsulation and Reuse

Paulo Moura

Dep. of Computer Science, University of Beira Interior, Portugal
Center for Research in Advanced Computing Systems, INESC–Porto, Portugal
pmoura@di.ubi.pt

Abstract. Prolog affords concise, elegant, and clean solutions for many interesting problems, but is not immune to the software engineering challenges of large-scale application development. Code modularization, using modules or objects, is a key feature to keep projects manageable. Since most literature, instruction, and practice focus exclusively on object-oriented languages derived from imperative languages, objects are perceived as alien to logic programming while modules are considered a natural fit. Logtalk is an object-oriented logic programming language that can use most Prolog implementations as a back-end compiler. Logtalk objects are about code encapsulation and reuse, providing an alternative to Prolog module systems, and enabling natural solutions for a wide range of problems that would be awkward to solve using modules. This talk presents the Logtalk design goals, followed by a tutorial on Logtalk programming and some application examples. The talk ends with a discussion on the problems and benefits of developing Logtalk as a portable Prolog application.

The slides used in this talk are available at
`http://logtalk.org/papers/iclp2009/logtalk_iclp2009.pdf`

About the speaker. Paulo Moura, PhD, is a Researcher at the Center for Research in Advanced Computing Systems (CRACS), INESC Porto, Portugal, and an Assistant Professor at the University of Beira Interior. His research interests include design and implementation of logic programming systems, multi-paradigm programming languages, and declarative object-oriented programming. His best-known work is the Logtalk object-oriented logic programming language. He prefers coding to writing papers, makes quixotic attempts to move Prolog standardization forward, and stalks innocent Prolog implementers with bug reports.

Acknowledgements. A word of gratitude to the Logtalk users for their suggestions, words of encouragement, bug reports, and other contributions. A special thanks to the Prolog implementers for their groundwork on developing and continually improving their compilers, in particular those working on open source projects. This work is partially supported by the FCT research project MOGGY (PTDC/EIA/70830/2006).

Generative Modeling by PRISM

Taisuke Sato

Tokyo Institute of Technology, Ookayama Meguro Tokyo Japan
`http://sato-www.cs.titech.ac.jp/`

Abstract. PRISM is a probabilistic extension of Prolog. It is a high level language for probabilistic modeling capable of learning statistical parameters from observed data. After reviewing it from various viewpoints, we examine some technical details related to logic programming, including semantics, search and program synthesis.

1 Introduction

Generative modeling is a way of probabilistic modeling that describes a generating process of an outcome in a sample space. It has however different nuances in different fields. In statistics it is oftentimes used in the context of Bayesian inference, which (hierarchically) assumes prior distributions on parameters. In ML(machine learning), it means defining joint distributions $p(x, y)$ where x is an input and y an output in contrast to discriminative modeling, which defines conditional distributions $p(y \mid x)$ for the same x and y. Or in statistical natural language processing, it usually refers to language modeling by probabilistic grammars such as HMMs (hidden Markov models) and PCFGs (probabilistic context free grammars). Here we add another nuance; by generative modeling we mean a specification of a sampling process by a probabilistic program for a given distribution.

Traditionally probabilistic models have been specified by mathematical formulas (equations) and graphs like BNs (Bayesian networks) and MRFs (Markov random fields) and programming languages were not considered as a specification language of probabilistic models. If, however, it becomes possible to use programs as probabilistic models, we will have much more flexibility in modeling because of the availability of various data structures (strings, trees, graphs) and program constructs (such as composition, if-then-else and recursion), and also a uniform mechanism (Turing machine). In addition, the expressive power of a high level programming language will reduce the coding effort to a minimum. So it seems beneficial to design a programming language and programs which denote probabilistic models. Indeed there are already a plethora of such proposals, in particular in a subfield of machine learning called PLL (probabilistic logic learning) originating in LP(logic programming)/ILP(inductive logic programming) [1,2,3,4,5,6,7,8,9,10,11,12] and SRL(statistical relational learning) originating in uncertainty reasoning by BNs [13,14,15,16,17,18,19,20,21,22,23,24].

In this talk, we examine PRISM[1] [2,5], a probabilistic extension of Prolog aiming at generative modeling by probabilistic logic programs. We will however focus on the relationship between PRISM and LP and applications to machine learning are not treated.

2 Three Viewpoints

We can see PRISM from three points of view.

[**LP view**] PRISM is a probabilistic extension of Prolog[2].

Syntactically a PRISM program $DB = F \cup R$ is a Prolog program such that F is a set of probabilistic atoms called `msw` atoms (see below) and R is a set of definite clauses whose head contains no `msw` atom. We use `msw(`i`,X)` to simulate a probabilistic choice named i (ground term) which returns in `X` a value probabilistically chosen from finite outcomes associated with i. Probabilities of `msw` atoms being true are called *parameters*. Semantically PRISM's declarative semantics, the *distribution semantics*, defines a probability measure $P_{DB}(\cdot \mid \boldsymbol{\theta})$ on Herbrand models having parameters $\boldsymbol{\theta}$ associated with `msw` atoms. It subsumes the least model semantics of definite clause programs. Practically what PRISM can do but Prolog cannot do is parameter learning. PRISM programs can learn $\boldsymbol{\theta}$ from data and change their probabilistic behavior.

[**ML view**] PRISM is a high level language for probabilistic modeling.

It is an outcome of PLL/SRL research, but unlike graphical models, it uses logical rules such as definite clauses or sometimes normal clauses to define distributions. Here is a short list of machine learning facilities supported by PRISM.[3]

Sampling: For the program $DB = F \cup R$, `sample(`G`)` executes G (atom) exactly as a Prolog goal using clauses in R except `msw(`i`,X)` which returns a probabilistically chosen value (ground term) in `X`.

Search: `probf(`G`)` returns, by searching for all SLD proofs for G with respect to DB, a boolean formula $E_1 \vee \ldots \vee E_n$ such that $E_i \wedge R \vdash G$ $(1 \leq i \leq n)$. Each E_i is a conjunction of ground `msw` atoms and called an *explanation* for G. $G \Leftrightarrow E_1 \vee \ldots \vee E_n$ holds with probability one in terms of $P_{DB}(\cdot)$.

Probability computation: `prob(`G`)` computes $P_{DB}(G)$, the probability of G by $P_{DB}(\cdot)$ whereas `chindsight(`G`,`G'`)` computes the conditional probability $P_{DB}(G' \mid G)$ of a subgoal G' that occurs in a proof of G.

Viterbi inference: `viterbif(`G`)` returns the most probable explanation for G together with its probability.

[1] `http://sato-www.cs.titech.ac.jp/prism/index.html`

[2] Currently PRISM is built on top of B-Prolog (`http://www.probp.com/`).

[3] See the PRISM manual for the complete list of available predicates.

Parameter learning: PRISM provides MLE(maximum likelihood estimation), MAP(maximum a posteriori) estimation for parameter leaning and VB (variational Bayes) for hyper parameter learning (priors are Dirichlet distributions). These are available through `learn/1` built-in predicate.

Model selection: To help structure learning, PRISM has special built-in predicates to compute criteria for model selection. They include BIC (Bayes information criterion), CS (Cheeseman-Stutz) score and VFE (variational free energy).

The primary benefit of PRISM modeling from the ML point of view is the ease of probabilistic modeling. We have only to write a program by a high level language and use it. There is no need for a laborious chain of deriving a learning algorithm, designing a data structure and implementing and debugging them. The result is a significant saving of time and energy. This is especially true when we attempt to develop a new model while going through cycles of probabilistic modeling. Think of developing some variants of HMMs for example. Once we write a basic HMM program, it is relatively easy to modify it. If the modified model goes wrong, just rewrite the program. We are free of implementing similar algorithms for similar HMMs all over again from scratch.

[**AI view**] PRISM is a system for statistical abduction.

PRISM performs search, computation and learning, all necessary elements of AI, in a unified manner under the distribution semantics. They are seamlessly integrated as *statistical abduction* [25]. In logical abduction, we abduce an explanation E for an observed goal G by search, using background knowledge $\mathbf{B}$, such that $E \wedge \mathbf{B} \vdash G$ and $E \wedge \mathbf{B}$ is consistent. Usually E is restricted to a conjunction of special atoms called abducibles. In statistical abduction, we further assume a distribution on abducibles and learn their probabilities from data. By doing so we can select E having the highest probability as the best explanation for G. In PRISM's case, the set R of definite clauses in a program $DB = F \cup R$ corresponds to $\mathbf{B}$ and `msws` in F play the role of abducibles.

Here is a small PRISM program.

```
values_x(p1,[rock,paper,scissors],fix@[0.4,0.4,0.2]).
values_x(p2,[rock,paper,scissors],[0.1,0.3,0.6]).

rps(R1,R2):-
    msw(p1,X),msw(p2,Y),
    ( X=Y -> R1=draw,R2=draw
    ; ((X=rock,Y=paper);(X=paper,Y=scissors);(X=sissors,Y=rock))
          -> R1=lose,R2=win
    ; R1=win,R2=lose ).
```

Fig. 1. Rock-paper-scissors program

This program simulates the rock-paper-scissors game. The first `values_x/3` clause introduces a probabilistic choice `msw(p1,X)` with a player `p1` and a gesture `X` being one of {`rock`, `paper`, `scissors`}, with corresponding parameters (probabilities) 0.4, 0.4 and 0.2 for each. "`fix@`" means parameters associated with `p1` do not change by learning. The second clause is understood similarly but the parameters, 0.1, 0.3 and 0.6, are temporarily set and changeable by learning. The last clause plays the rock-paper-scissors game. It first calls `msw(p1,X)` to probabilistically choose a gesture `X` for `p1` and similarly `Y` for `p2` by `msw(p2,Y)`. It then determines `win`, `lose` or `draw` by comparing `X` and `Y`.

```
?- prism(rock_paper_scissors).
...
?- get_samples(1000,rps(R1,R2),Gs),learn(Gs).
...
#em-iterations: 0.......(79) (Converged: -1091.799641688)
Statistics on learning:
       Graph size: 18
       Number of switches: 2
       Number of switch instances: 6
       Number of iterations: 79
       Final log likelihood: -1091.799641688
       Total learning time: 0.004 seconds
       Explanation search time: 0.000 seconds

?- show_sw.
Switch p1: rock (p: 0.4000) paper (p: 0.4000) scissors (p: 0.2000)
Switch p2: rock (p: 0.0641) paper (p: 0.3466) scissors (p: 0.5892)

?- viterbif(rps(win,lose)).
rps(win,lose) <= msw(p1,rock) & msw(p2,scissors)
```

Fig. 2. Learning session

Fig. 2 is a sample learning session (predicates used there are all built-ins). We first load the program in Fig. 1 on a file `rock_paper_scissors.psm` by `prism/1`. We then generate learning data `Gs` = [`rps`(`win`, `lose`), `rps`(`draw`, `draw`), . . .] by `get_samples/3` which sampled `rps(R1,R2)` 1,000 times[4]. `learn(Gs)` internally invokes a built-in EM algorithm to estimate parameters. The learning is completed after 79 iterations. The estimated values are shown by `show_sw/0`. Using the learned parameters, we compute by `viterbif/1` the most probable gestures that cause `rps(win,lose)`, i.e. `p1` wins and `p2` loses. They are `rock` for `p1` and `scissors` for `p2` with log-probability -2.1972.

[4] `p1` wins 343 times, `p2` wins 375 times, draw 282 times.

3 Inside PRISM: Three Topics

As mentioned before, PRISM can be seen from three points of view. In this section we pick up the LP view and look into some details of three topics which illustrate how PRISM is connected to LP. They are semantics [2,5], tabling [26] and program synthesis [27].

3.1 Probabilistic Semantics

The distribution semantics of PRISM is a probabilistic generalization of the least model semantics in LP. It defines a probability measure on the set of Herbrand models. Let $DB = F \cup R$ be a PRISM program. Also let $\mathtt{msw}_1, \mathtt{msw}_2, \ldots$ be an enumeration of the `msw` atoms in F. We identify an infinite 0-1 vector $\omega_F = (x_1, x_2, \ldots)$ where $x_i \in \{0,1\}$ with a Herbrand model that assigns $\mathtt{msw}_1 = x_1$, $\mathtt{msw}_2 = x_2, \ldots$ where 1 means true and 0 false. Let $P_F(\cdot)$ be an arbitrary *base measure* on such ω_Fs such that for a choice named i with possible outcomes $\{v_1, \ldots, v_k\}$, $P_F(\cdot)$ makes $\{\mathtt{msw}(i,v_1), \ldots, \mathtt{msw}(i,v_k)\}$ exhaustive and mutually exclusive. That is $P_F(\mathtt{msw}(i,v_1) \vee \cdots \vee \mathtt{msw}(i,v_k)) = 1$ and $P_F(\mathtt{msw}(i,v_h) \wedge \mathtt{msw}(i,v_{h'})) = 0$ $(h \neq h')$. It is straightforward to construct such $P_F(\cdot)$.

We now extend the $P_F(\cdot)$, using the mechanism of the least Herbrand model construction, to a probability measure $P_{DB}(\cdot)$ for the whole DB. Let $\omega_{F'}$ be a sample from $P_F(\cdot)$ and F' the set of `msw` atoms made true by $\omega_{F'}$. Construct the least Herbrand model $\mathbf{M}(F' \cup R)$ of the definite program $F' \cup R$. It uniquely determines the truth value of every ground atom and by construction every ground atom is a measurable function of $\omega_{F'}$ with respect to $P_F(\cdot)$. It follows from this fact and Kolmogorov's extension theorem that we can extend $P_F(\cdot)$ to the probability measure $P_{DB}(\cdot)$ on the set of possible Herbrand models for DB. $P_{DB}(\cdot)$ is the denotation of DB in the distribution semantics [5]. If $P_F(\cdot)$ puts all probability mass on a single interpretation F', P_{DB} puts all probability mass on the least model $\mathbf{M}(F' \cup R)$ also. Hence we can say the distribution semantics is a probabilistic generalization of the least model semantics. Hereafter for intuitiveness, we identify $P_{DB}(\cdot)$ with an infinite joint distribution $P_{DB}(A_1 = x_1, A_2 = x_2, \ldots)$ on the probabilistic ground atoms $A_1, A_2, \ldots$ in the Herbrand base of DB where $x_i \in \{0,1\}$, when appropriate.

We remark that our semantics (probability measure on possible models, or worlds) is not new. Fenstad proved a representation theorem forty years ago [28]. It states that if we assign probabilities $P(\varphi)$ to closed formulas φ in a countable language $\mathcal{L}$ without equality, respecting Kolmogorov's axioms for probability while satisfying $P(\varphi) = 1$ if $\vdash \varphi$ and $P(\varphi) = P(\phi)$ if $\vdash \varphi \Leftrightarrow \phi$, $P(\varphi)$ is given as an integration

$$P(\varphi) = \int_{\Omega} \varphi(\omega)\, \mu(d\omega)$$

where $\mu(\cdot)$ is a probability measure on a certain set Ω of models related to $\mathcal{L}$ and $\varphi(\omega) = 1$ if a model $\omega \in \Omega$ satisfies φ, else 0[5]. What is semantically new here

[5] The actual Fenstad's theorem is more complicated than stated here. We show the case of closed formulas for simplicity.

is that we construct such $\mu(\cdot) = P_{DB}(\cdot)$ concretely from a logic program DB so that $P_{DB}(G)$ is computable for a goal G. We point out some unique features of the distribution semantics.

- $P_{DB}(\cdot)$ is an infinite joint distribution on countably many random atoms. *It is definable, unconditionally, for any DB*. Other PLL formalisms often place restrictions on DB such as acyclicity [1,8] and range-restrictedness [3,6,9] for their distributions to be definable. SRL formalisms attempting to define infinite distributions also have restrictions on their programs [21,29,30].
- Probabilistic grammars such as HMMs and PCFGs that define finite stochastic processes but whose length is unbounded are formally captured by PRISM programs with the distribution semantics. Thanks to the rigor of the distribution semantics, it is even possible to write a PRISM program defining prefix probabilities for a given PCFG, though their computation requires an infinite sum and cannot be handled by the current PRISM system[6] [31].
- The distribution semantics is parameterized with a non-probabilistic semantics $\mathbf{M}$ used to extend the base measure $P_F(\cdot)$. That is, if we choose as $\mathbf{M}$ the greatest model semantics instead of the least model semantics, we will have another distribution, which is always definable but not necessarily computable, giving non-zero probability to infinite recursion. $\mathbf{M}$ may be stable model semantics [11,32] or well-founded semantics [8,33]. In such cases, we will have distributions for normal probabilistic logic programs.

3.2 Tabling and Dynamic Programming

In PRISM the probability $P_{DB}(G)$ of an atom G is computed by first reducing G logically to a disjunction $E_1 \vee \ldots \vee E_n$ of explanations and then computing $P_{DB}(G)$ by $P_{DB}(G) = \sum_{i=1}^{n} P_{DB}(E_i)$, $P_{DB}(E_i) = \prod_{k=1}^{h_i} \theta_{i,k}$ where $E_i = \mathtt{msw}_{i,1} \wedge \cdots \wedge \mathtt{msw}_{i,h_i}$ and $\theta_{i,k}$ is the parameter of $\mathtt{msw}_{i,k}$ $(1 \leq k \leq h_i)$. A computational barrier here is that usually there are exponentially many explanations. In the case of parsing where G represents a sentence and E_i a parse tree, it is not rare to have millions of parse trees. One standard way to avoid such intractable computation is applying DP (dynamic programming) to $E_1 \vee \ldots \vee E_n$ that factors out common probability computations. But the real problem is not DP but how to realize it *without* constructing $E_1 \vee \ldots \vee E_n$.

Our solution to this problem is tabling, or memoizing, which is a general technique to record what has been computed and reuse it later, thereby saving repeated computation. In addition, tabling has the side-effect of stopping infinite recursion. This makes it possible to write a DCG grammar containing left recursive rules such as `NP` $\rightarrow$ `NP S`. LP has a long history of tabling [26,34,35,36,37,38][7] and what we have found through the development of PRISM is that tabling is well-suited, or vital to probability computation in machine learning.

[6] Prefix probabilities can be computed by matrix operations [31].

[7] Our tabling is linear-tabling [26] which does not use a suspend-resume mechanism for tabled execution of logic programs but iteratively computes answers until they saturate.

By introducing tabling for all explanations search for a goal G, we can obtain a boolean formula, equivalent to $G \Leftrightarrow E_1 \vee \ldots \vee E_n$, as a descendingly ordered list of equivalences $G \Leftrightarrow W_0, A_1 \Leftrightarrow W_1, \ldots, A_M \Leftrightarrow W_M$ such that A_i $(1 \leq i \leq M)$, a tabled goal appearing in a proof of G, represents a subexpression occurring multiple times in the explanations $E_1, \ldots, E_n$ and W_i is a conjunction of `msw` atoms and tabled goals in the lower layers. We consider this list as a graph whose node are atoms and call it an *explanation graph* for G. In the explanation graph, G is a root node and subgraphs (tabled goals) at one layer are shared by subgraphs at higher layers. Hence probability computation (sum-product computation) applied to it naturally becomes DP. Thus we can realize DP by tabling while avoiding the construction of $E_1 \vee \ldots \vee E_n$. We encode the DP process as the g-IO (generalized IO) algorithm working on explanation graphs. It is a generic routine in PRISM to compute probabilities [5].

The effect of tabling is decisive. The g-IO algorithm simulates known standard probability computation/learning algorithms with the same time complexity; $O(N^2L)$ for the Baum-Welch algorithm used in HMMs [39] where N is the number of states, L input length, $O(N^3L^3)$ for the Inside-Outside algorithm used in PCFGs in Chomsky normal form [40] where N is the number of symbols and L sentence length and $O(N)$ for Pearl's $\pi\lambda$ message passing [41] used in the probability computation of singly connected BNs where is N the number of nodes in a BN [5].

Also, recently, it is discovered that the celebrated BP (belief propagation) algorithm used for the probability computation of multiply connected BNs is nothing but the g-IO algorithm applied to logically described junction trees [42]. In other words, to use BP, we have only to write a program describing a junction tree[8]. We may say PRISM subsumes both probabilistic grammars and BNs not only at the semantic level but also at the probability computation/learning level.

Tabling is useful in Bayesian inference as well. In [43] we introduced the VB (variational Bayes) approach to PRISM and implemented the VB-EM algorithm that learns hyper parameters of Dirichlet priors associated with `msw` atoms, in a dynamic programming manner using explanation graphs and the slightly extended g-IO algorithm. Hyper parameter learning is done with the same time complexity as usual parameter learning because both types of learning use the same explanation graphs and isomorphic learning algorithms. We test the left-corner parsing model and the profile-HMM model. Although there is no report on their hyper parameter learning to our knowledge, all we need to do is to write a declarative PRISM program for each model, and the rest of the task - hyper parameter learning followed by Viterbi inference based on the learned hyper parameters - is carried out automatically by the PRISM system.

3.3 Log-Linear Models and Logic Program Synthesis

The last topic is non-generative modeling. Generative modeling, typically PCFGs to us, assumes no failure in the process of generating an outcome. However

[8] The distribution of the PRISM system includes an example of logical junction tree. Querying the tree with `chindsight_agg/2` is equivalent to running BP.

logic programs may fail as we all know. The problem caused by failure to logic-based probabilistic modeling such as SLPs (stochastic logic programs) [3,44] and PRISM [45] is loss of probability mass. If the execution eventually fails after a probabilistic choice is made, the probability mass put on the choice is lost. As a result the total sum of probabilities for possible generation processes will be less than unity, implying that our probability is not mathematically correct.

Suppose there is a PRISM program DB about `q(X)` which defines a distribution $P_{DB}(\cdot)$. Let $\mathtt{t}_1, \ldots, \mathtt{t}_N$ be all answer substitutions for the query `?-q(X)`. If failure computation occurs during the search for all answers for `?-q(X)` and $Z = P_{DB}(\exists \mathtt{X} \mathtt{q}(\mathtt{X})) = \sum_{i=1}^{N} P_{DB}(\mathtt{q}(\mathtt{t}_i)) < 1$ happens, we consider a normalized distribution $Z^{-1}P_{DB}(\mathtt{q}(\mathtt{X}))$ over $\{\mathtt{q}(\mathtt{t}_1), \ldots, \mathtt{q}(\mathtt{t}_N)\}$ to recover probability.

However $Z^{-1}P_{DB}(\mathtt{q}(\mathtt{X}))$ is a log-linear model[9] and parameter learning of log-linear models is known to be much harder than BNs and PCFGs due to the computation of Z, a normalizing constant. Cussens proposed the FAM (failure-adjusted maximization) algorithm for parameter learning of SLPs whose computation may fail and hence defines log-linear models [44]. It is an EM algorithm but requires the computation of "failure probability" $1 - Z$ (Z is the probability of success computation).

We incorporated the FAM algorithm into PRISM by applying a logic program synthesis technique to PRISM programs to derive special programs called *failure programs* to compute failure probabilities $1 - Z$. Given a program DB for the target goal `q(X)` which has failed computation paths, we consider another goal $\mathtt{failure} \Leftrightarrow \forall \mathtt{X}(\mathtt{q}(\mathtt{X}) \Rightarrow \mathtt{false})$ and synthesize a failure program for this `failure` predicate so that `?-failure` faithfully traces every failed computation path for `?-q(X)` in the original program DB. Under a certain condition[10], it can be proved $P_{DB}(\mathtt{failure}) = 1 - P_{DB}(\mathtt{q}(\mathtt{X})) = 1 - Z$ [46]. The point here is not that we can compute $1 - Z$ exactly but that we are now able to compute it using DP by applying tabled search to the synthesized failure program. In [46], an example of HMMs with constraints which may fail is presented. The synthesized failure program runs by tabled execution in time linear in the length of input for the original HMM program.

The program synthesis for failure programs is done by FOC (first-order compiler) [27]. It is an unfold/fold program transformation system for logic programs with universally quantified implicational goals $\forall y(p(x, y) \Rightarrow q(y, z))$[11] in the clause body. FOC transforms the original PRISM program while considering the probabilistic semantics of `msw` atoms into a PRISM program with disequality constraints.

The program in Fig. 3 models probabilistic singular/plural agreement between nouns and verbs in some hypothetical language. `coin(a)` determines the

[9] Log-linear models take the form $\log p(x) = \sum_i w_i f_i(x)$ where $f_i(x)$ is a real-valued function called feature and w_i is a real number called weight. In the case of SLPs, $f_i(x)$ is the number of occurrences of an i-th clause in a refutation x.

[10] Roughly every computation path for `q(X)` must terminate with finite failure or success.

[11] Negation $\neg p(x, y) = (p(x, y) \Rightarrow \mathtt{false})$ is a special case.

```
failure :- not(success).  |   failure:-closure_success0(f0).
success :- agree(_).      |   closure_success0(A):-closure_agree0(A).
                          |
agree(A):-                |   closure_agree0(_):-
  msw(coin(a),A),         |     msw(coin(a),A),
  msw(coin(b),B),         |     msw(coin(b),B),
  A=B.                    |      \+A=B.
```

Fig. 3. Agreement program (left) and the synthesized failure program (right)

singularity/plurality of a noun with probability 0.4/0.6 respectively and so does `coin(b)` for a verb. If they do not agree, the sentence generation fails. `failure` predicate on the left hand side is defined as the negation of $\exists$`A agree(A)` (success of `agree(_)`). FOC compiles it into the failure program on the right hand side by removing negation while introducing new predicates `closure_success/1` and `closure_agree0/1` (see [27] for details). As you can see, the compiled program correctly computes failure probability.

4 Concluding Remarks

We have examined PRISM, an extension of Prolog with `msw/2` predicate for probabilistic choice, the distribution semantics, tabled search and generic routines for probability computation and parameter learning. We have been developing PRISM for more than a decade, to achieve generality and efficiency for probabilistic modeling, but there remains a long way to go. The future work includes an implementation of Gaussian distributions, also that of log-linear models, and removing some modeling condition (the exclusiveness condition [5]) by the introduction of BDDs.

References

1. Poole, D.: Probabilistic Horn abduction and Bayesian networks. Artificial Intelligence 64(1), 81–129 (1993)
2. Sato, T.: A statistical learning method for logic programs with distribution semantics. In: Proceedings of the 12th International Conference on Logic Programming (ICLP 1995), pp. 715–729 (1995)
3. Muggleton, S.: Stochastic logic programs. In: De Raedt, L. (ed.) Advances in Inductive Logic Programming, pp. 254–264. IOS Press, Amsterdam (1996)
4. Poole, D.: The independent choice logic for modeling multiple agents under uncertainty. Artificial Intelligence 94(1-2), 7–56 (1997)
5. Sato, T., Kameya, Y.: Parameter learning of logic programs for symbolic-statistical modeling. Journal of Artificial Intelligence Research 15, 391–454 (2001)

6. Kersting, K., De Raedt, L.: Basic principles of learning bayesian logic programs. Technical Report Technical Report No. 174, Institute for Computer Science, University of Freiburg (2002)
7. Blockeel, H.: Prolog for Bayesian networks: a meta-interpreter approach. In: Proceedings of the 2nd International Workshop on Multi-Relational Data Mining (MRDM 2003), pp. 1–13 (2003)
8. Vennekens, J., Verbaeten, S., Bruynooghe, M.: Logic programs with annotated disjunctions. In: Demoen, B., Lifschitz, V. (eds.) ICLP 2004. LNCS, vol. 3132, pp. 431–445. Springer, Heidelberg (2004)
9. Fierens, D., Blockeel, H., Bruynooghe, M., Ramon, J.: Logical Bayesian networks and their relation to other probabilistic logical models. In: Kramer, S., Pfahringer, B. (eds.) ILP 2005. LNCS, vol. 3625, pp. 121–135. Springer, Heidelberg (2005)
10. De Raedt, L., Kimmig, A., Toivonen, H.: ProbLog: A probabilistic Prolog and its application in link discoverry. In: Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI 2007), pp. 2468–2473 (2007)
11. Baral, C., Gelfond, M., Rushton, N.: Probabilistic reasoning with answer sets. Theory and Practice of Logic Programming (TPLP) 9(1), 57–144 (2009)
12. De Raedt, L., Kersting, K.: Probabilistic inductive logic programming. In: De Raedt, L., Frasconi, P., Kersting, K., Muggleton, S. (eds.) Probabilistic Inductive Logic Programming - Theory and Applications. LNCS, pp. 1–27. Springer, Heidelberg (2008)
13. Breese, J.S.: Construction of belief and decision networks. Computational Intelligence 8(4), 624–647 (1992)
14. Koller, D., Pfeffer, A.: Learning probabilities for noisy first-order rules. In: Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI 1997), pp. 1316–1321 (1997)
15. Friedman, N., Getoor, L., Koller, D., Pfeffer, A.: Learning probabilistic relational models. In: Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI 1999), pp. 1300–1309 (1999)
16. Pfeffer, A.: IBAL: A probabilistic rational programming language. In: Proceedings of the 17th International Conference on Artificial Intelligence (IJCAI 2001), pp. 733–740 (2001)
17. Jaeger, J.: Complex probabilistic modeling with recursive relational Bayesian networks. Annals of Mathematics and Artificial Intelligence 32(1-4), 179–220 (2001)
18. Getoor, L., Friedman, N., Koller, D., Taskar, B.: Learning Probabilistic Models of Relational Structure. Journal of Machine Learning Research 3, 679–707 (2002)
19. Costa, V., Page, D., Qazi, M., Cussens, J.: CLP(BN): Constraint logic programming for probabilistic knowledge. In: Proceedings of the 19th Conference on Uncertainty in Artificial Intelligence (UAI 2003), pp. 517–524 (2003)
20. Milch, B., Marthi, B., Russell, S., Sontag, D., Ong, D., Kolobov, A.: BLOG: Probabilistic models with unknown objects. In: Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI 2005), pp. 1352–1359 (2005)
21. Laskey, K.: MEBN: A logic for open-world probabilistic reasoning. C4I Center Technical Report C4I06-01, George Mason University Department of Systems Engineering and Operations Research (2006)
22. Richardson, M., Domingos, P.: Markov logic networks. Machine Learning 62, 107–136 (2006)
23. Getoor, L., Grant, J.: PRL: A probabilistic relational language. Journal of Machine Learning 62(1-2), 7–31 (2006)
24. Getoor, L., Taskar, B. (eds.): Introduction to Statistical Relational Learning. MIT Press, Cambridge (2007)

25. Sato, T., Kameya, Y.: Statistical abduction with tabulation. In: Kakas, A., Sadri, F. (eds.) Computational Logic: Logic Programming and Beyond. LNCS (LNAI), vol. 2408, pp. 567–587. Springer, Heidelberg (2002)
26. Zhou, N.F., Sato, T., Shen, Y.D.: Linear tabling strategies and optimization. Theory and Practice of Logic Programming 8(1), 81–109 (2008)
27. Sato, T.: First Order Compiler: A deterministic logic program synthesis algorithm. Journal of Symbolic Computation 8, 605–627 (1989)
28. Fenstad, J.E.: Representation of probabilities defined on first order languages. In: Crossley, J.N. (ed.) Sets, Models and Recursion Theory, pp. 156–172. North-Holland, Amsterdam (1967)
29. Milch, B., Marthi, B., Sontag, D., Russell, S., Ong, D., Kolobov, A.: Approximate Inference for Infinite Contingent Bayesian Networks. In: Proceedings of the 10th International Workshop on Artificial Intelligence and Statistics (AISTATS 2005), pp. 1352–1359 (2005)
30. Domingos, P., Singla, P.: Markov logic in infinite domains. In: De Raedt, L., Dietterich, T., Getoor, L., Kersting, K., Muggleton, S. (eds.) Probabilistic, Logical and Relational Learning - A Further Synthesis. Dagstuhl Seminar Proceedings, vol. 07161 (2008)
31. Stolcke, A.: An efficient probabilistic context-free parsing algorithm that computes prefix probabilities. Computational Linguistics 21(2), 165–201 (1995)
32. Gelfond, M., Lifshcitz, V.: The stable model semantics for logic programming, pp. 1070–1080 (1988)
33. Van Gelder, A., Ross, K., Schlipf, J.: The well-founded semantics for general logic programs. The journal of ACM (JACM) 38(3), 620–650 (1991)
34. Tamaki, H., Sato, T.: OLD resolution with tabulation. In: Shapiro, E. (ed.) ICLP 1986. LNCS, vol. 225, pp. 84–98. Springer, Heidelberg (1986)
35. Sagonas, K., Swift, T., Warren, D.: XSB as an efficient deductive database engine. In: Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, pp. 442–453 (1994)
36. Ramakrishnan, I., Rao, P., Sagonas, K., Swift, T., Warren, D.: Efficient tabling mechanisms for logic programs. In: Proceedings of the 12th International Conference on Logic Programming (ICLP 1995), pp. 687–711. The MIT Press, Cambridge (1995)
37. Guo, H.F., Gupta, G.: A simple scheme for implementing tabled logic programming systems based on dynamic reordering of alternatives. In: Codognet, P. (ed.) ICLP 2001. LNCS, vol. 2237, pp. 181–196. Springer, Heidelberg (2001)
38. Sagonas, K., Stuckey, J.: Just enough tabling. In: Proceedings of the 6th ACM SIGPLAN international conference on Principles and practice of declarative programming (PPDP 2004), pp. 78–89. ACM, New York (2004)
39. Rabiner, L.R., Juang, B.: Foundations of Speech Recognition. Prentice-Hall, Englewood Cliffs (1993)
40. Baker, J.K.: Trainable grammars for speech recognition. In: Proceedings of Spring Conference of the Acoustical Society of America, pp. 547–550 (1979)
41. Pearl, J.: Probabilistic Reasoning in Intelligent Systems. Morgan Kaufmann, San Francisco (1988)
42. Sato, T.: Inside-Outside probability computation for belief propagation. In: Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI 2007), pp. 2605–2610 (2007)

43. Sato, T., Kameya, Y., Kurihara, K.: Variational bayes via propositionalized probability computation in prism. Annals of Mathematics and Artificial Intelligence (to appear)
44. Cussens, J.: Parameter estimation in stochastic logic programs. Machine Learning 44(3), 245–271 (2001)
45. Sato, T., Kameya, Y.: PRISM: a language for symbolic-statistical modeling. In: Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI 1997), pp. 1330–1335 (1997)
46. Sato, T., Kameya, Y., Zhou, N.F.: Generative modeling with failure in PRISM. In: Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI 2005), pp. 847–852 (2005)

Enabling Serendipitous Search on the Web of Data Using Prolog

Jan Wielemaker

University of Amsterdam

Abstract. The Web of Data, also called the Semantic Web, provides a knowledge representation formalism based on a uniform triple-model: subject, predicate, object. A number of more expressive formalisms (e.g., RDFS, OWL, SKOS) are layered on top of the core triple-model. The Web of Data has been developed to represent machine readable knowledge on the internet. Designed to deal with heterogeneous knowledge, the technology underlying the Web of Data is also suitable to unify databases. We use this technology to unify collection information from multiple museums, based on diverse schemas and multiple controlled lists of terms (vocabularies). The resulting knowledge-base is enriched using automatic discovery of mappings between vocabularies. The current challenge is how to provide meaningful services for the end-user based on this knowledge, in particular, how to provide meaningful semantically enriched search?

This tutorial presents the key-components of the Prolog-based ClioPatria toolkit and shows how this infrastructure can be used to explore the opportunities of semantic search. Topics discussed are: reasoning with and editing of RDF models, web-application programming in Prolog combined with AJAX technology and issues when using Prolog for programming 'at large'. Prolog is both an RDF query language and a general purpose programming language, and therefore provides a perfect platform for a Semantic Web research and applications.

Untangling Reverse Engineering with Logic and Abstraction

Andy King

Portcullis Computer Security Limited, Pinner, HA5 2EX, UK

Reverse engineering is the filthy end of the security industry; it is the business of extracting information from a program when the source is unavailable. Reversing is often necessary when performing a security audit on a product that relies on third-party software such as a library. Security engineers also reverse to reason about the latest malicious programs and devise antivirus software. Security engineers (and malicious hackers) do not attempt to reverse assembler into, say C, which is the traditional aspiration in reversing, but merely to understand the code to sufficient depth to locate a vulnerability.

The most popular tool that is used for reversing is the IDA Pro dissembler [4]. This dissembler divides an executable into (more or less) its basic blocks, presenting them visually to the engineer in a flow diagram. Needless to say, the major impediment to reversing is the enormous effort required to understand an executable even when it is presented as a flow diagram. In fact, even of recovering the control-flow graph from a binary is more complicated than one would expect [2] and, IDA Pro often fails to reconstruct the complete control-flow graph.

One notable body of work that also aims to support the reversing is the thesis work of Balakrishnan [1]. Balakrishnan, under the direction of Reps, has developed a so-called value set analysis that attempts to uniformly track addresses and numeric values. The rationale for this approach is that it enables sets of addresses on some word alignment to be accurately represented. We consider this approach to be a major advance in the analysis of binaries, since it attempts to seamlessly support addresses and numeric values. Recently it has also been shown how logical techniques based on bit-blasting and SAT solving can also be applied to extract bit-level modulo relationships [3].

The tutorial will review this growing body of work, highlighting the potential for applying logical methods and abstraction techniques in reversing.

References

1. Balakrishnan, G.: WYSINWYX: What You See Is Not What You eXecute. Ph.D thesis, Computer Sciences Department, University of Wisconsin, Madison (2007)
2. Kinder, J., Veith, H., Zuleger, F.: An abstract interpretation-based framework for control flow reconstruction from binaries. In: Muller-Olm, M. (ed.) VMCAI 2009. LNCS, vol. 5403, pp. 214–228. Springer, Heidelberg (2009)
3. King, A., Søndergaard, H.: Inferring Congruence Equations using SAT. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 281–293. Springer, Heidelberg (2008)
4. Pennell, J.: Reverse Engineering with IDA Pro. IOActive (2008)

(C)LP Tracing and Debugging

Mireille Ducassé

IRISA-INSA of Rennes, Campus universitaire de Beaulieu,
35042 Rennes Cedex, France
ducasse@irisa.fr

Since Ehud Shapiro's "Algorithmic debugging", in 1983, there has been a continuous, even if not very abundant, flow of work on tracing and debugging for (constraint) logic programming. The tutorial presents trace production techniques, ranging from compiler instrumentations to dedicated meta-interpreters. It reviews work on trace analysis, in particular algorithmic, declarative and rational debugging. It discusses the issue of trace querying and driving. Last but not least, it describes the latest software engineering research on trace mining.

Throughout the presentation, we stress the importance of the nature of the trace data used by the techniques. We show that CLP techniques have inspired a number of work in other communities. We argue that trace mining techniques can easily be applied to CLP.

Probabilistic Logic Learning - A Tutorial Abstract[⋆]

Luc De Raedt

Department of Computer Science, Katholieke Universiteit Leuven, Belgium
luc.deraedt@cs.kuleuven.be

Probabilistic logic learning (PLL), sometimes also called statistical relational learning, addresses one of the central questions of artificial intelligence: the integration of probabilistic reasoning with first order logic representations and machine learning. It has received a lot of attention recently and a rich variety of different formalisms and learning techniques have been developed, which are now being applied to applications in network analysis, robotics, bio-informatics, intelligent agents, etc.

This tutorial starts with an introduction to probabilistic representations and machine learning, and then continues with an overview of the state-of-the-art in probabilistic logic learning. We start from classical settings for logic learning (or inductive logic programming) namely learning from entailment, learning from interpretations, and learning from proofs, and show how they can be extended with probabilistic methods. While doing so, we review state-of-the-art probabilistic logic learning approaches and formalisms, such as PRISM, ICL, stochastic logic programs, Bayesian logic programs, CP-Logic and Markov logic.

More detailed introductory overviews on probabilistic logic learning can be found in [1,2,3,4,5].

Keywords: probabilistic logic learning, statistical relational learning, logic and learning, inductive logic programming, logic programming.

References

1. De Raedt, L.: Logical and Relational Learning. Springer, Heidelberg (2008)
2. De Raedt, L., Frasconi, P., Kersting, K., Muggleton, S. (eds.): Probabilistic Inductive Logic Programming. LNCS (LNAI), vol. 4911, pp. 1–27. Springer, Heidelberg (2008)
3. Getoor, L., Taskar, B. (eds.): An Introduction to Statistical Relational Learning. MIT Press, Cambridge (2007)
4. De Raedt, L., Kersting, K.: Probabilistic logic learning. SIGKDD Explorations 5(1), 31–48 (2003)
5. De Raedt, L., Kersting, K.: Probabilistic inductive logic programming. In: Ben-David, S., Case, J., Maruoka, A. (eds.) ALT 2004. LNCS, vol. 3244, pp. 19–36. Springer, Heidelberg (2004)

⋆ This tutorial is – in part – based on joint work with Kristian Kersting.

Using Histograms to Better Answer Queries to Probabilistic Logic Programs

Matthias Broecheler, Gerardo I. Simari, and V.S. Subrahmanian

University of Maryland College Park, College Park, MD 20742, USA
{matthias,gisimari,vs}@cs.umd.edu

Abstract. Probabilistic logic programs (PLPs) define a set of probability distribution functions (PDFs) over the set of all Herbrand interpretations of the underlying logical language. When answering a query Q, a lower and upper bound on Q is obtained by optimizing (min and max) an objective function subject to a set of linear constraints whose solutions are the PDFs mentioned above. A common critique not only of PLPs but many probabilistic logics is that the difference between the upper bound and lower bound is large, thus often providing very little useful information in the query answer. In this paper, we provide a new method to answer probabilistic queries that tries to come up with a histogram that "maps" the probability that the objective function will have a value in a given interval, subject to the above linear constraints. This allows the system to return to the user a histogram where he can directly "see" what the most likely probability range for his query will be. We prove that computing these histograms is $\#P$-hard, and show that computing these histograms is closely related to polyhedral volume computation. We show how existing randomized algorithms for volume computation can be adapted to the computation of such histograms. A prototype experimental implementation is discussed.

Keywords: Probabilistic Logic Programming, Imprecise Probabilities.

1 Introduction

Since the introduction of quantitative logic programs by Shapiro [1], van Emden [2], and others, there has been extensive interest in logic programming with uncertainty. While these early frameworks were fuzzy in nature, Ng and Subrahmanian [3] introduced probabilistic logic programs by building on top of probabilistic logics studied earlier by several authors such as Hailperin [4], Fagin *et al.* [5], and Nilsson [6]. There has been much subsequent work in this vein [7,8,9].

A fundamental problem with all of these probabilistic logics is the assumption of *ignorance* — it is assumed that we do not know of any dependencies or correlations between the events represented in these logics. Given a probabilistic logic program Π over a logical language $\mathcal{L}$, we write down an associated set $LC(\Pi)$ of linear constraints. Each (ground) rule in Π generates one constraint. In addition, we have one variable in $LC(\Pi)$ for each Herbrand interpretation for language $\mathcal{L}$. While the rules in Π constrain what interpretations satisfy Π, these variables denote the probability that a Herbrand interpretation I actually represents the true state of the world. Assuming that the Herbrand Base of $\mathcal{L}$ is denoted $B_{\mathcal{L}}$, this means the linear program has $2^{B_{\mathcal{L}}}$ variables in it, and $O(|grd(\Pi)|)$ constraints. In many of these cases, $2^{B_{\mathcal{L}}}$ is significantly larger than $|grd(\Pi)|$. A consequence of this — well known to those in the field — is that $LC(\Pi)$ is vastly underconstrained as the number of variables very often significantly exceeds the number of rules. This has profound implications for the prospective utility of probabilistic logics and probabilistic logic programs. When answering a query Q (think of a query for now as a logical formula), we need to find the "lower bound" probability low_Q such that every Herbrand interpretation satisfying Π also satisfies Q with probability greater than or equal to low_Q. Likewise, we want to find the "upper bound" probability up_Q such that every Herbrand interpretation satisfying Π also satisfies Q with probability less than or equal to up_Q. To find the tightest such interval $[low_Q, up_Q]$ of this type, we minimize and maximize (respectively) an objective function associated with Q. When the problem is underconstrained as in most cases, it is often the case that low_Q is very close to 0 and up_Q is very close to 1, providing the user who wants to know the probability of Q very little information about the true probability of Q. The example below shows a very simple probabilistic logic program.

Example 1 (Stock Example). Consider a very simple probabilistic logic program Π_{stock} (using the syntax of [3]):

$$\begin{array}{lrl}
r_1 & stim\_pkg : [0.30, 0.90] & \leftarrow . \\
r_2 & home\_sales\_up : [0.25, 0.85] & \leftarrow . \\
r_3 & up\_ibm \wedge up\_goog : [0.40, 0.95] & \leftarrow home\_sales\_up : [0.65, 0.90]. \\
r_4 & up\_ibm \vee up\_goog : [0.60, 0.95] & \leftarrow home\_sales\_up : [0.65, 0.85]. \\
r_5 & up\_ibm : [0.30, 0.80] & \leftarrow stim\_pkg : [0.70, 1.0].
\end{array}$$

The first two rules intuitively say that there is a $30 - 90\%$ probability that a stimulus package will be announced (today) and that there is a $25 - 85\%$ probability that there will be an economic report released (today) that home sales are up. Rule r_3 says that if such a home sales report is released today, then IBM and Google's stock price will go up tomorrow with $40 - 95\%$ probability. Rule r_4 says that when such a home sales report is released (today), there is a $60 - 95\%$ probability that either IBM or Google's stock price will be up tomorrow. The last rule says that if an economic stimulus package is announced today, then there is a $30 - 80\%$ probability that IBM's stock price will go up tomorrow.[1] Though this example is obviously very simplistic, the reader can easily see that probabilistic logic rules that state that certain stocks go up when certain conditions are true can easily be derived from historical stock market data. Clearly, a stock analyst would like to make decisions based on such data.

According to the semantics of probabilistic logic programming [3], the probability of the conjunctive query $Q_1 = (stim\_pkg \wedge home\_sales\_up \wedge up\_ibm \wedge up\_goog)$ is given by the interval $[0, 0.8]$. This is the tightest possible interval that we can infer for this query w.r.t. Π_{stock}. A stock analyst would have very little ability to "act" based on this answer, because the probability interval $[0, 0.8]$ is so wide that it basically tells the analyst very little. Past work in the AI community has often selected some value in

[1] We don't introduce time in this paper for the sake of simplicity. But you can think of the propositional symbols in the heads of the last three rules intuitively denoting stock movements tomorrow, while all other propositional symbols in Π_{stock} refer to events today.

this interval based on some principle (*e.g.*, maximum entropy, assuming independence, etc.). Worse still, the query $Q_2 = up\_ibm$ is entailed by Π_{stock} with tightest probability interval $[0.4, 0.8]$. Without any further information, the stock analyst may think that the probability of IBM going up is greater than 0.5, which might induce him to bet on IBM stock. However, the true story is that the probability of the probability of up_ibm being in the $[0.4, 0.5]$ interval is actually 61%.

Moreover, no stock market analyst is going to want to risk millions of dollars of a mutual fund's investment based on what a probabilistic logic expert tells him (especially when that probabilistic logic expert knows nothing about the stock market and speaks in generalities about using maximal entropy, independence assumptions, etc.). The stock analyst wants to make these decisions, not rely on AI experts who do not understand the stock domain as well as he does.

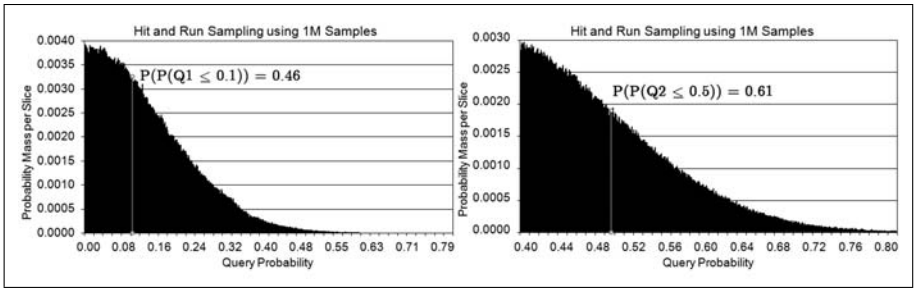


Fig. 1. Histogram answers to queries Q_1 (left) and Q_2 (right) of Stock Example

Figure 1 shows visualizations of the histograms that we can present to such a stock analyst *without making any additional assumptions* about the dependencies, correlations, etc. that may or may not exist, that the analyst may or may not believe, etc. The visualization shows a histogram for each query. The x-axis in Figure 1 (left), which corresponds to query Q_1, ranges from 0 to 0.8 (corresponding to the $[0, 0.8]$ interval associated with query Q). For a given point x in this interval, the histogram shows the probability that the probability of Q is at most x. Figure 1 (left) shows a sample value x_0 and its corresponding value $h(x_0)$. The histogram in Figure 1 (right) is similar and corresponds to query Q_2.

The stock analyst has an immediate sense, by looking at the histogram in Figure 1 (right) that he should not bet on IBM stock going up. Likewise, the probability of query Q_1 having probability 0.5 or more is low. However, there is no way for him to see this if we merely present him the interval $[0.4, 0.8]$ as the answer to the query. The histogram presents this interval (as the x-axis bounds), but it also shows far more valuable information that can enable the stock analyst to make a decision.

The goal of this paper is to show how to present answers of the kind mentioned above to the user so that we (i) present more information to the user than we did before, and (ii) so that this answer is expressed in an easy to understand graphical manner. We do this by using higher order probabilities.

The rest of this paper is organized as follows. In Section 2, we overview past work on PLPs from [3]. Then, in Section 3, we present the basic declarative semantics underlying histogram answers to PLP queries, and show that the histogram answer computation (HAC) for PLP queries is closely related to the problem of computing volumes of convex polytopes. In Section 4 we show that the HAC problem is $\#P$-hard; we also present two approximation algorithms for the HAC problem and prove appropriate complexity theorems. Section 5 contains implementation and experimental results showing that one of the approximation algorithms is far superior to the other.

2 Preliminaries

We now review (a simplified version of) the syntax and semantics of PLPs given in [10,3]; there is nothing particularly new in this section.

2.1 Syntax of PLPs

We assume the existence of a set of propositional $\mathcal{L}_{pred}$ logic symbols. Every propositional symbol is an *atom*. Formulas are defined as follows.

Definition 1 (Formula). *An atom is a formula. If F_1 and F_2 are formulas, then $F_1 \wedge F_2$, $F_1 \vee F_2$, and $\neg F_1$ are formulas. Let $Form(\mathcal{L}_{pred})$ denote the set of all formulas.*

If F is a formula and $[\ell, u]$ is a subset of the real unit interval, $F : [\ell, u]$ is called an annotated *formula.*

Returning to Example 1, we can see that $stim\_pkg : [0.3, 0.9]$, $(up\_ibm \vee up\_goog) : [0.4, 0.95]$ and $(up\_ibm \vee up\_goog) : [0.65, 0.85]$ are annotated formulas. We now define the concept of *probabilistic rule*.

Definition 2. *If $F : \mu$, $B_1 : \mu_1, \ldots, B_m : \mu_m$ are annotated formulas, then $F : \mu \leftarrow B_1 : \mu_1 \wedge \ldots \wedge B_m : \mu_m$ is a* probabilistic rule. *If this rule is named r, then $Head(r)$ denotes $F : \mu$, and $Body(r)$ denotes $B_1 : \mu_1 \wedge \ldots \wedge B_m : \mu_m$.*

Intuitively, a probabilistic rule is a statement saying that if the formulas in the body are true with their associated probabilities, then the formula in the head is also true with its associated probability.

Definition 3. *A* probabilistic logic program *(PLP) is a finite set of probabilistic rules.*

Again, it is easy to see that in Example 1, Π_{stock} is a PLP. [2]

2.2 Semantics of PLPs

PLPs are characterized by a Kripke style possible worlds semantics.

[2] The syntax presented here is, due to space constraints, simpler than that in [3]. In particular, variable annotations and function symbols over the annotation domain are eliminated; [3] also removes the assumption of propositional logic and allows predicate symbols and FOL atoms. However, the current framework can be easily extended to those cases. The definition of PLP above, however, allows more complex formulas to appear in rules compared to those of [3]; in particular, negation (not a non-monotonic form of negation though) can appear in rule heads.

Definition 4 (World). *A* world *is any set of atoms.*

We use $\mathcal{W}$ to denote the set $2^{\mathcal{L}_{pred}}$ of all possible worlds. Since a world is simply a Herbrand interpretation, it is clear what it means for a world to *satisfy* a formula. A *probabilistic interpretation* is a probability distribution over worlds.

Definition 5. *Let S be a set of annotated formulas in $\mathcal{L}$, and $\mathcal{W}$ be the set of possible worlds. A* probabilistic interpretation *is a function $I : \mathcal{W} \to [0,1]$ such that $\sum_{w \in \mathcal{W}} I(w) = 1$.*

Definition 6 (Satisfaction). *Let $F : [\ell, u]$ be an annotated formula and I be a probabilistic interpretation. I is said to* satisfy *$F : [\ell, u]$ iff $\ell \leq \sum_{w_i \in \mathcal{W}, w_i \models F} I(w_i) \leq u$.*

Let $r = F : \mu \leftarrow B_1 : \mu_1 \wedge \ldots \wedge B_m : \mu_m$ be a probabilistic rule; I is said to satisfy *r iff either I satisfies Head(r) or I does not satisfy some $B_i : \mu_i \in$ Body(r).*

A probabilistic interpretation satisfies a PLP Π if and only if it satisfies all rules in Π. A PLP Π is said to be *consistent* if and only if there exists an interpretation I that satisfies all formulas in Π, and Π *entails* an annotated formula $F : \mu$ if and only if every interpretation that satisfies all rules in Π also satisfies $F : [\ell, u]$.

The above definition naturally leads to the definition of a system of linear constraints whose solutions will correspond to satisfying interpretations. We call this set $LC(S)$, and it contains one variable p_i for each world $w_i \in \mathcal{W}$ and the following constraints:

1. For each $F : [\ell, u] \in S$, $\ell \leq \sum_{w_i \in \mathcal{W}, w_i \models F} p_i \leq u$, and
2. $\sum_{w_i \in \mathcal{W}} p_i = 1$

It follows immediately from [3], that S is consistent if and only if $LC(S)$ is solvable.

Fixpoint Operator. Via a straightforward extension of a similar procedure in [10,3], it is possible to associate a fixpoint operator T_Π with any PLP Π [3]. This operator maps sets of annotated formulas to sets of annotated formulas as follows and first involves defining an intermediate operator S_Π.

$$\begin{aligned} S_\Pi(X) = \{F : \mu \mid (F : \mu \leftarrow B_1 : \mu_1 \wedge \cdots \wedge B_m : \mu_m) \in \Pi \wedge \\ (\forall\, 1 \leq i \leq m)(\exists B_i' : \mu_i' \in X)(F_i = B_i \wedge \mu_i \subseteq \mu_i')\}. \end{aligned}$$

For *each formula*[4] F, let $[\ell_F, u_F]$ denote the result of minimizing and maximizing $\sum_{w_i \in \mathcal{W}, w \models F} p_i$ subject to $LC(S_\Pi(X))$. We now define $T_\Pi(X)$ as follows.

$$T_\Pi(X) = \{F : [\ell_F, u_F] \mid F \in Form(\mathcal{L}_{pred})\}.$$

Using results similar to those in [10,3], it is easy to show that T_Π has a least fixpoint and an annotated formula $F : [\ell, u]$ is a logical consequence of Π iff there is a formula $F : [\ell', u']$ in the least fixpoint such that $[\ell', u'] \subseteq [\ell, u]$.

[3] W.l.o.g., we assume that no rules in Π have formulas with a $[0, 1]$ annotation in the body.

[4] Many methods can be used to reduce the number of formulas in $Form(\mathcal{L}_{pred})$ we need to consider. Due to space constraints, and as this is not central to this paper, we ignore this issue.

3 Histogram Answers to a PLP Query

In classical PLPs, a query Q is an annotated formula $F : [\ell, u]$ and we want to check if Q is entailed by PLP Π (or alternatively if the least fixpoint of T_Π contains an annotated formula $F : [\ell', u']$ with $[\ell', u'] \subseteq [\ell, u]$). An alternative version says the query is a formula F and we want to find the annotated formula $F : [\ell', u']$ in the least fixpoint of T_Π. In this section, we propose a fundamentally different construct as the answer to the query that provides far more information to the user. Given a formula F as the query, we want to provide to the user a *histogram answer* to the query F w.r.t. a PLP Π. In order to do this, and in order to make our theory consistent with standard notation in (continuous) probability theory, we assume, without loss of generality, that all worlds in $\mathcal{W}$ are enumerated as $w_1, w_2, \ldots, w_{|\mathcal{W}|}$ in some total order, and that an interpretation I is represented as a vector $(p_1, \ldots, p_{|\mathcal{W}|})$ where each p_i denotes the probability of world w_i according to interpretation I, *i.e.*, $I(w_i) = p_i$. We now define the probability that a query formula will lie within a given probability interval.

Definition 7 (Higher-Order Probability of Entailment). *Suppose Π is a* PLP *and Q is a query formula. Suppose $[a, b]$ is a non-empty subset of $[0, 1]$. We define the* higher order probability that Q is entailed by Π with probability in $[a, b]$ *as:*

$$\mathbb{L}(a \leq Q \leq b \mid \Pi) = \int_{I \in \mathit{Mod}(\Pi)} \chi_{\left[a \leq \sum_{w_i \in \mathcal{W}, w_i \models Q} I(w_i) \leq b\right]} I d\mathbb{P}_\Pi I$$

where χ is the adapted set membership function, i.e., $\chi_{C(x)} = 1$ if $C(x)$ is true and 0 otherwise, for some condition C, and $\mathbb{P}_\Pi$ is the uniform probability distribution over $\mathit{Mod}(\Pi)$, the set of all interpretations that satisfy Π. Thus, $\chi_{\left[a \leq \sum_{w_i \in \mathcal{W}, w_i \models Q} I(w_i) \leq b\right]} I$ is true if interpretation I is such that $\sum_{w_i \in \mathcal{W}, w_i \models Q} I(w_i)$ lies between a and b.

We now show that the expression above yields a valid probability distribution.

Theorem 1. *Given probability distribution $\mathbb{P}_\Pi$ over the set of interpretations that satisfy Π, a PLP Π, and some query formula Q, $\mathbb{L}(Q = x \mid \Pi) = \mathbb{L}(x \leq Q \leq x \mid \Pi)$ is a proper probability distribution over* $[0, 1]$.

Proof sketch. Let $\mathit{Mod}(\Pi)$ be the set of all interpretations that satisfy Π. Then:

$$\begin{aligned}
&\textstyle\int_0^1 \mathbb{L}(Q = x \mid \Pi) \ = \ \mathbb{L}(0 \leq Q \leq 1 \mid \Pi) \\
&= \textstyle\int_{\mathit{Mod}(\Pi)} \chi_{\left[0 \leq \sum_{w_i \in \mathcal{W}', w_i \models Q} p_i \leq 1\right]} (I = (p_i)) d\mathbb{P}_\Pi I \\
&= \textstyle\int_{\mathit{Mod}(\Pi)} \chi_{[\text{true}]} (I = (p_i)) d\mathbb{P}_\Pi I = \int_{\mathit{Mod}(\Pi)} 1 d\mathbb{P}_\Pi I = 1
\end{aligned}$$

The last equality holds since $\mathbb{P}_\Pi$ is a probability distribution over $\mathit{Mod}(\Pi)$. □

We now return to Example 1 in order to illustrate the above definition of a higher order probability of entailment.

Example 2. Consider the queries Q_1 and Q_2 of Example 1:

– $\mathbb{L}(0 \leq Q_1 \leq 0.1 \mid \Pi_{\mathit{stock}})$. This represents the probability that Q_1 is entailed by Π_{stock} with probability in the range $[0, 0.1]$. We compute this using Definition 7 by solving the integral: $\int_{I \in \mathit{Mod}(\Pi)} \chi_{\left[0 \leq \sum_{w_i \in \mathcal{W}, w_i \models Q_1} I(w_i) \leq 0.1\right]} I d\mathbb{P}_\Pi I$.

– $\mathbb{L}(0.7 \leq Q_2 \leq 0.75 \mid \Pi_{stock})$. This represents the probability that Q_2 is entailed by Π_{stock} with probability in the range $[0.7, 0.75]$. Similar to the first case, we compute this by solving: $\int_{I \in Mod(\Pi)} \chi_{\left[0.7 \leq \sum_{w_i \in \mathcal{W}, w_i \models Q_2} I(w_i) \leq 0.75\right]} I d\mathbb{P}_\Pi I$.

Given a query formula Q, we can now ask for the probability that Q is entailed by PLP Π with point probability p or with a probability in the range $[a, b]$. The answer to these queries, respectively, are $\mathbb{L}(p \leq Q \leq p \mid \Pi)$ and $\mathbb{L}(a \leq Q \leq b \mid S)$. This gives us more information than simply knowing the widest interval $[\ell, u]$ of probability values for the entailment of Q. $\mathbb{L}$ gives us the entire distribution of probability values for a query formula and not just the smallest interval such that $\mathbb{L}(\ell \leq Q \leq u \mid \Pi) = 1$. *Thus, the higher order probability of entailment gives users strictly more information than answers in classical* PLP. Moreover, as shown in Figure 1, we can present the entire distribution of $\mathbb{L}$ for a given query Q, and enable a naive user (who has no in-depth knowledge of probability theory, and almost certainly no knowledge of higher order probabilities) to visualize the probability distribution for his query. There are two ways to do this. As Definition 7 provides a continuous probability distribution, we can just present an approximation of the continuous histogram as shown in Figure 1, or we can also present a *discrete* version of this answer.

Definition 8 (Histogram Answer). *Suppose Π is a PLP and Q is a query. The* histogram answer *to query Q w.r.t. PLP Π is the function $\mathbb{L}$.*

Further suppose that $k \geq 1$ is an integer and that the $[\ell, u]$ is the tightest interval such that $\Pi \models Q : [\ell, u]$. The k-discrete histogram answer *to query Q w.r.t.* PLP *Π is the set $\{\mathbb{L}(\ell + (i-1) * \frac{u-\ell}{k} \leq Q \leq \ell + i * \frac{u-\ell}{k} \mid \Pi) \mid 1 \leq i \leq \frac{u-\ell}{k}\}$.*

If the user wants a discrete (rather than a continuous) histogram answer, then he can select an integer k which specifies the desired level of discretization. The k-discrete histogram answer splits the tightest $[\ell, u]$ interval such that $\Pi \models Q : [\ell, u]$ into k equally sized sub-intervals. For each of these subintervals, it finds the probability that Q's probability lies in that sub-interval using the formula given above. The following theorem shows that computing these histograms is closely related to the problem of volume computation in convex polyhedra.

Theorem 2. *Let $\mathbb{P}_\Pi$ be the uniform probability distribution over Mod(S), Q a query formula, and $[a, b] \subseteq [0, 1]$. Then:*

$$\mathbb{L}(a \leq Q \leq b \mid \Pi) = \frac{vol\left(SOL\left(a \leq \sum_{w_i \in \mathcal{W}, w_i \models \Pi, w_i \models Q} prob(w_i) \leq b\right)\right)}{vol\left(SOL(LC(\Pi))\right)}$$

where $SOL(X)$ denotes the set of solutions of a set of constraints X, and $vol(B)$ denotes the m dimensional volume of a set of points B that form an m dimensional body in Euclidean space[5].

[5] The solutions of $LC(\Pi)$ and the models of the PLP Π are in exact one to one correspondence, so we could speak interchangeably about either solutions of $LC(\Pi)$ or models of Π. We prefer the former, as we are using geometric intuitions in computing polytope volumes.

For ease of notation, we will denote the numerator of the above expression by $Mod(\Pi)(a \leq Q \leq b) = \left\{ I \in Mod(\Pi) \mid a \leq \sum_{w_i \in \mathcal{W}, w_i \models \Pi, w_i \models Q} I(w_i) \leq b \right\}$.

Proof sketch. $\mathbb{L}(a \leq Q \leq b \mid \Pi) =$

$$= \frac{\int_{I \in Mod(\Pi)} \chi_{\left[a \leq \sum_{w_i \in \mathcal{W}, w_i \models Q} I(w_i) \leq b\right]} I d\mathbb{P}_\Pi I}{1}$$

$$= \frac{\int_{I \in Mod(\Pi)} \chi_{\left[a \leq \sum_{w_i \in \mathcal{W}, w_i \models Q} I(w_i) \leq b\right]} I d\mathbb{P}_\Pi I}{\int_{Mod(\Pi)} 1 d\mathbb{P}_\Pi I}$$

$$= \frac{vol\left(SOL\left(a \leq \sum_{w_i \in \mathcal{W}, w_i \models \Pi, w_i \models Q} prob(w_i) \leq b\right)\right)}{vol\left(SOL(LC(\Pi))\right)}$$

□

Theorem 2 shows that computing the probability distribution $\mathbb{L}$ is closely related to volume computations on the convex polytope formed by the linear constraints in $LC(S)$ in n dimensional Euclidean space.

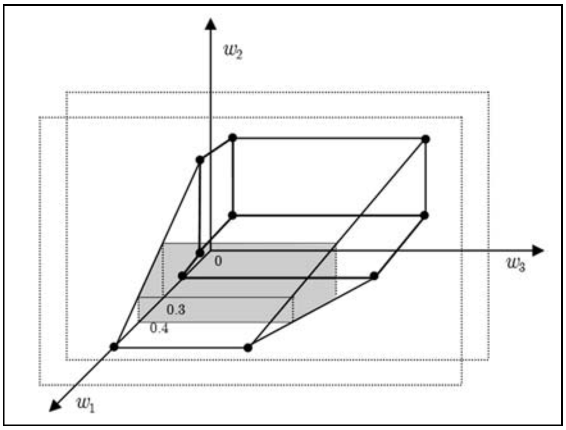


Fig. 2. The polytope from Example 3 intersected by the two hyperplanes that are determined by the query formula and its probability interval (region corresponding to Q is shown shaded)

Example 3. Suppose we have $\Pi = \{a : [0.6, 0.9], b : [0.2, 0.5]\}$, and the query formula is $Q = a \wedge \neg b$. The set of possible worlds is given by $w_0 = \{\}$, $w_1 = \{a\}$, $w_2 = \{b\}$, and $w_3 = \{a, b\}$. In the following, let p_i denote the probability of world w_i being true; $LC(\Pi)$ is given by:

$$\{0.6 \leq p_1 + p_3 \leq 0.9, \quad 0.2 \leq p_2 + p_3 \leq 0.5, \quad p_0 + p_1 + p_2 + p_3 = 1\}$$

In this case, the query formula is satisfied only by world w_1. Maximizing and minimizing the value of variable p_1 in the LP above yields as a result that Q is entailed with a probability in the interval $[0, 0.5]$. Figure 2 shows the geometric interpretation of these constraints. Here, the shaded region corresponds to the probability that Q will be true with a probability between 0.3 and 0.4.

4 Volume Computation and Answer Histograms

As shown in the preceding section, computing $\mathbb{L}(a \leq Q \leq b \,|\, \Pi)$ can be reduced to the problem of computing the ratio between the two volumes $\{I \mid I \models \Pi \wedge a \leq I(Q) \leq b\}$ and $Mod(\Pi)$. Compared to $Mod(\Pi)$, the former is also a convex polytope which is defined via the set of linear constraints $LC(\Pi)$ and two additional constraints:

$$(1) \sum_{w_i \in \mathcal{W}, w_i \models Q} p_i \geq a, \quad (2) \sum_{w_i \in \mathcal{W}, w_i \models Q} p_i \leq b$$

We use $LC(\Pi, Q, a, b)$ to refer to this modified set of constraints for a query Q.

Hence, we can build upon previous work on computing volumes of convex polytopes. A simple algorithm for the *discrete histogram answer* to PLP queries would work as follows and uses a function called vol that takes a set of linear constraints as input and returns the volume of the convex polytope generated by those constraints.

```
**Algorithm DiscreteHistoAnswer**$(\Pi, Q, k)$
1. Result $= \emptyset$;
2. Minimize and maximize $\sum_{w_i \in \mathcal{W}, w_i \models Q} p_i$ subject to $LC(\Pi)$ to get $\ell, u$ respectively;
3. Let $c = (u - \ell)/k$;
4. **for** $i = 1$ **to** $c$ **do**
    a. $V_{\ell+(i-1)*c, \ell+i*c} = \frac{vol(LC(\Pi, Q, \ell+(i-1)*c, \ell+i*c))}{vol(LC(\Pi, Q, \ell, u))}$;
    b. Add $V_{\ell+(i-1)*c, \ell+i*c}$ to $Result$;
5. **return** Result;
```

The following result states that this algorithm correctly computes the discrete histogram answer and follows immediately from Theorem 2.

Theorem 3. *Algorithm* ***DiscreteHistoAnswer***(Π, Q, k) *correctly computes the k-discrete histogram answer to this query.*

As the correctness of the above algorithm depends on volume computation algorithms, we provide a brief overview of those algorithms in Section 6. Due to the high dimensionality of the PLP histogram answer computation problem, exact volume computation algorithms are not going to work in practice. [16] study such algorithms and only consider cases with dimensionality below 20. Even in our very small stock market example, which has just 4 propositional symbols, we already have a 16-dimensional space as there are 16 possible worlds to consider! On the other hand, randomized volume computation algorithms use random walks with rapid mixing time[6] inside the polytope. Such random walks generate a Markov chain where each point in the polytope corresponds to a state in the Markov chain, and the transition probabilities denote the probability of the random walk taking you from one point to another. Sampling from this Markov Chain in accordance with the mixing time yields a uniform distribution over the polytope. Using this sampling strategy, one can compute the ratio between the

[6] The term *mixing time* refers to the number of steps the random walk must take in order to reach its stationary distribution; see [17] for a complete treatment.

volume of a known body (*e.g.*, the unit cube) and the polytope of interest. Naively applying existing volume computation algorithms to compute $\mathbb{L}(a \leq Q \leq b \,|\, \mathit{\Pi})$ as given in the **DiscreteHistoAnswer** algorithm has two serious shortcomings:

(1) We wish to plot a histogram of the distribution of $\mathbb{L}$, *i.e.*, for an interval width $\delta = \frac{u-l}{k}$. Computing each of the volumes $vol(LC(\mathit{\Pi}, Q, \ell + (i-1) * \delta, \ell + i * \delta))$ is expensive as the (already expensive) volume computation algorithm would need to be invoked $k+1$ times (once for each of the k discretized components, and once for the entire volume). This increases the running time by $O(k)$.
(2) As stated before, computing $\mathbb{L}(a \leq Q \leq b \mid \mathit{\Pi})$ requires the computation of the *ratio* between the two volumes and not the actual volume. This raises the question: can we somehow do better than volume computation algorithms?

The following theorem provides an answer to point (2).

Theorem 4. *Let K denote an arbitrary n dimensional polytope which is defined as the intersection of a set K_M of half-spaces. Let A, B be two additional half-spaces and let L denote the polytope which is the intersection of the half-spaces in $L_M = K_M \cup \{A, B\}$. Under these circumstances, computing $\frac{vol(L)}{vol(K)}$ is #P-hard.*

Proof sketch. Dyer and Frieze [13] have proven that computing the volume of a convex polytope defined by the intersection of half-spaces is $\#P$-hard. We show how convex polytope volume computation can be reduced to relative volume computation in polynomial time, thereby establishing $\#P$-hardness of relative volume computation.

We assume that an arbitrary polytope K is defined by the intersection of a set of K_M of half-spaces. To compute the volume of K using relative volumes, we proceed as follows. Firstly, we make the customary assumption that the origin o is inside K. We can determine the maximal inscribed n dimensional sphere inside K in time polynomial in the number of bounding half-spaces $|K_M|$. Let r be the radius of this maximal sphere, then we can fit a cube C of edge length $\ell = \frac{2r}{\sqrt{n}}$ centered at the origin inside this circle and hence C must be contained in K. For more details on how a contained cube can be determined in polynomial time, the interested reader is referred to Applegate and Kannan [18] who proved that one can find an affine mapping in polynomial time which maps K to K' such that the unit cube is contained in K'.

We can compute the volume of C in closed form as $vol(C) = \ell^n$. Using this base volume we can derive the volume of K as follows. Let $\{F_j^i\}$ for $i = 1, \ldots, n$ and $j = 0, 1$ denote the set of faces of the cube C where F_0^i, F_1^i are parallel and opposing faces, for all i. For our purposes, we consider the faces to be half-spaces which bound the cube. Then C can be considered as the intersection of the n pairs of parallel half-spaces F_0^i, F_1^i. Now, let K_d denote the polytope defined as the intersection of half-spaces $K_M \cup \{F_0^i, F_1^i \mid i = 1, \ldots, d\}$ for $0 \leq d \leq n$. Then $K_0 = K$ and $K_n = C$, since C is contained in K. We can now derive the volume of K, $vol(K) =$

$$vol(K_0) = vol(K_1)\frac{vol(K_0)}{vol(K_1)} = vol(K_n)\prod_{d=1}^{n}\frac{vol(K_{d-1})}{vol(K_d)} = \ell^n \prod_{d=1}^{n}\frac{vol(K_{d-1})}{vol(K_d)}$$

Hence, we have reduced computing the exact volume of K to the product of n relative volume computations, which completes the polynomial reduction. □

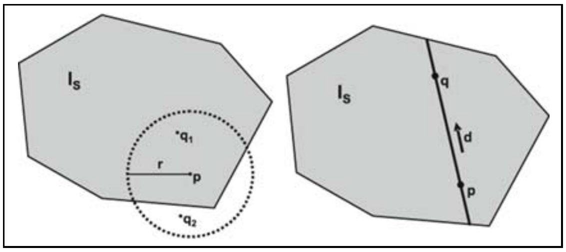


Fig. 3. Schematic Ball Walk (left) and Hit-and-Run (Right)

4.1 The Approx-HOPE Algorithm

We now present the Approx-HOPE algorithm (short for the Approximate Histogram Oriented Probabilistic Entailment algorithm) which uses randomized methods to compute the histogram answer to a query Q w.r.t. a PLP Π. The Approx-HOPE algorithm uses a function called *randomWalk* that takes $LC(\Pi)$ as input and performs a random walk through the convex polytope defined by Π. This function can be implemented in many ways, two of which we will discuss later.

```
Algorithm Approx-HOPE($\Pi, Q, k$)
1. Result $= \emptyset$;
2. Let $\delta = (u - l)/k$;
3. Sample $= randomWalk(LC(\Pi))$;
4. For $i = 1$ to $\delta$ do
    a. $V_{\ell+(i-1)*\delta,\ell+i*\delta} = \frac{|Sample \cap [\ell+(i-1)*\delta,\ell+i*\delta]|}{|textitSample|}$;
    b. Add $V_{\ell+(i-1)*\delta,\ell+i*\delta}$ to Result;
5. return Result;
```

The Approx-HOPE algorithm is quite simple. Rather than solve volume computation problems $k + 1$ times as the **DiscreteHistoAnswer** algorithm does, this algorithm basically executes one pass of the sampling stage of these randomized volume computation algorithms. All these algorithms sample from a polytope with a view to inferring the volume of the polytope. Rather than sample to determine the volume of the polytope, we try to use the random walk to estimate the part of the polytope's volume that lies within one of the k probability intervals that we are discretizing our problem into.

Though Approx-HOPE can be used with any appropriately designed random walk algorithm, we have tested it extensively with two well known ones:

(1) The **random ball walk (RBW)** starts at an arbitrarily chosen point $p \in SOL(LC(\Pi))$ where $SOL(X)$ denotes the set of solutions of a set X of constraints. It has a fixed associated parameter r which denotes the radius of a "ball" used during the random walk. To move to the next point, we uniformly sample a point q from the n dimensional sphere of radius r with center p. If q lies inside the polytope $LC(\Pi)$, the random walk moves to point q, otherwise the point is rejected and the walk stays at p. The procedure is then repeated at the selected (new or old) point. Figure 3 (left) visualizes the random ball

walk and shows the point q_1 which would be accepted as the next move and q_2 which would be rejected.
(2) The **Hit-and-Run (HAR)** walk also starts at an arbitrary point $p \in SOL(LC(\Pi))$ and has no parameters. At each step, a direction d (*i.e.*, a point on the n dimensional sphere surface) is chosen uniformly at random. We compute the segment of line l inside the polytope $Mod(\Pi)$, where l is the line through p in direction d. Finally, a point q is chosen uniformly at random from this line segment and the walk moves to q. Figure 3 (right) shows a line segment inside the polytope and the next point q. Note that the Hit-and-Run walk *never rejects any points*.

It has been shown that both RBW and HAR have a mixing time of $O^*(n^3)$; however, HAR achieves this mixing time under weaker assumptions [19]. As we will see in Section 5, our experiments show that HAR performs much better in practice as it mixes much more rapidly. This is due to the fact that the random ball walk frequently gets "stuck" for large radii and moves only very slowly for small radii.

Theorem 5. *Using either the RBW or HAR sampling strategy,* Approx-HOPE *runs in time in $O^*(n^4m)$, where m is the number of rules in Π and n is the number of worlds.*

Proof sketch. Sampling uniformly at random from a ball of radius r takes time linear in the number of dimensions n. Determining whether a point lies inside the polytope defined by $LC(\Pi)$, as required by RBW, as well as computing the line fragment for a given direction d, which is needed for HAR, can be done in time in $O(nm)$. □

5 Experiments

We implemented Approx-HOPE with both the RBW and HAR methods in Matlab 7.7.0 on a single machine with a 2.6 GHz Intel Core Duo Processor using only a single core and 3GB of RAM.

In our experiments, we randomly generated least fixpoints of PLPs. These fixpoints contained 3 to 10 annotated formulas, each with up to 4 propositional symbols in them. No fixpoint contained more than 12 propositional symbols in total. Though there should be 2^k worlds when there are k propositional symbols in such fixpoints, we eliminated some worlds using a world equivalence method described in [20], which is why the numbers of worlds in Figure 4 are not necessarily powers of two. We then recorded run times for the Approx-HOPE algorithm using the RBW and HAR sampling strategies and three different sample sizes. The running times in seconds are shown in Figure 4.

	500,000 Samples		1,000,000 Samples		2,000,000 Samples	
	Ball Walk	Hit-And-Run	Ball Walk	Hit-And-Run	Ball Walk	Hit-And-Run
3 rules, 7 worlds	13.7	23.1	28.6	46.7	56.1	91.4
4 rules, 15 worlds	14.6	23.8	29.7	47.7	58.3	95.6
5 rules, 31 worlds	15.4	26.1	30.7	52.2	62.1	102.6
6 rules, 59 worlds	16.5	29.7	32.9	60.4	65.3	119.0
7 rules, 71 worlds	17.0	31.5	34.0	63.3	68.3	127.5
8 rules, 112 worlds	19.9	38.4	40.5	76.4	76.7	153.5
9 rules, 159 worlds	25.9	46.4	52.0	93.4	100.7	180.8
10 rules, 239 worlds	38.5	65.2	77.1	130.7	149.6	259.8

Fig. 4. Running times in seconds for varying numbers of worlds and rules

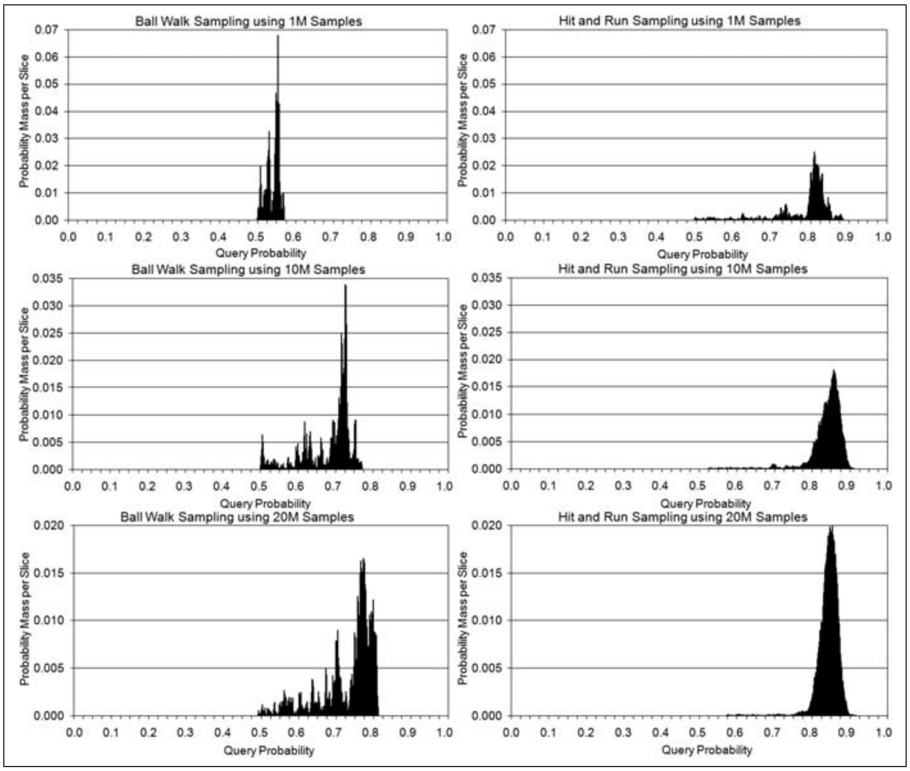


Fig. 5. Histograms output by different runs of the Ball Walk (left) and Hit and Run (right) algorithms on the same PLP with 10 rules (341 variables in the LP) for different sample sizes. Note that the y axis has different scales at different sample sizes.

As expected, the run times increase linearly with the number of samples for all rule sets. Moreover, the run time increases with the number of worlds, because the computational cost per sample depends on the number of worlds, as explained in the proof of Theorem 5. We observe that the RBW strategy outperforms the HAR strategy in running time since its cost per iteration is lower. Note that the sample sizes were identical for all rule sets, irrespective of the number of worlds and therefore of the mixing times.

In the qualitative experiments we studied the convergence of the RBW and HAR sampling strategies in detail by holding the rule set and query constant and varying the sample size between 100,000 and 40 million. Part of the results for a single experiment with 10 rules and 341 worlds are shown in Figure 5. Across all experiments we observed that HAR converges more quickly to the uniform distribution than RBW. As an example, Figure 5 shows that Approx-HOPE with HAR already clearly indicates the subinterval with the highest probability after only 1 million samples, whereas the RBW is still "walking" toward that region in the polytope. After 20 million samples, HAR has converged to the uniform distribution (*i.e.*, increasing the sample size does not change the histogram) whereas RBW is still far from convergence. We conclude that *the HAR* *sampling strategy significantly outperforms RBW*, despite its favorable cost per iteration, since HAR converges much more rapidly and requires significantly less iterations. To verify the scalability of Approx-HOPE we experimented with a set of 15 rules giving rise to 682 worlds using different random queries. The HAR strategy took about 10 hours to converge to the uniform distribution after approximately 140 million samples.

6 Related Work

Probabilistic logic programming has been studied for almost 25 years [4,5,6,3,7,8,9]. For most of this time, researchers have known that the probability intervals associated with PLP queries can be inordinately wide, often giving very little information to the user about the truth or falsity of the query and, as illustrated in our stock example, making it difficult for the user to make decisions. Past approaches to this problem have been relatively *ad hoc*, arbitrarily choosing solutions in $LC(\Pi)$ that somehow correspond to some intuition of the researcher, such as maximal entropy. Such approaches are valid when the assumptions are valid in the application domain, but little or no effort has gone into verifying whether those assumptions are valid. Presumably the user will decide, but consider the feasibility of asking a stock analyst who has no idea what entropy is to decide whether maximal entropy is the right semantics for him.

Regarding exact volume computation algorithms, Cohen and Hickey [11] were the first to propose methods based on triangulation with exponential run time complexity, followed by Khachiyan [12] a decade after. Later, Dyer and Frieze [13] proved that computing the volume of a convex polytope defined by a set of constraints is $\#P$-hard, thereby showing that this is the best time complexity one can achieve for exact algorithms. Dyer *et al.* [14] proposed a randomized algorithm to compute arbitrarily tight bounds on the volume of convex polytopes with high probability in polynomial time. [15] presented an $O^*(n^4)$ randomized polynomial time (approximation) algorithm, where n is the dimensionality of the polytope[7].

7 Conclusion

In this paper, we solve the problem of dealing with wide probability intervals *without making any assumptions*, and at the same time providing a simple, graphical output to the user in the form of an easy to understand histogram. We do this by defining, for the first time, the unique notion of a *histogram answer* to a query w.r.t. a PLP. We show that the histogram answer computation problem is $\#P$-hard, and further show a close relationship between this and volume computation in convex polytopes. We provide an exact algorithm to compute histogram answers (which is expectedly inefficient because of the $\#P$-hardness result). We further develop an approximation algorithm Approx-HOPE that can work with any sampling method and evaluate it using two types of random walk sampling strategies: *Random Ball Walk* and *Hit and Run*. We develop an initial (small) prototype and quickly discover that Approx-HOPE combined with Hit and Run is much more efficient than with Random Ball Walk.

[7] The O^* notation ignores logarithmic factors and other factors such as error bounds.

Acknowledgements. The authors gratefully acknowledge funding support for this work provided by Air Force Office of Scientific Research through grants FA95500610405 and FA95500510298, and by the Office of Naval Research through grant N000140910685.

References

1. Shapiro, E.Y.: Logic programs with uncertainties: A tool for implementing rule-based systems. In: IJCAI, pp. 529–532 (1983)
2. van Emden, M.: Quantitative deduction and its fixpoint theory. Journal of Logic Programming 4, 37–53 (1986)
3. Ng, R.: A semantical framework for supporting subjective and conditional probabilities in deductive databases. Journal of Automated Reasoning 10, 565–580 (1993)
4. Hailperin, T.: Probability logic. Notre Dame J. of Formal Logic 25(3), 198–212 (1984)
5. Fagin, R., Halpern, J.Y., Megiddo, N.: A logic for reasoning about probabilities. Information and Computation 87(1/2), 78–128 (1990)
6. Nilsson, N.: Probabilistic logic. Artificial Intelligence 28, 71–87 (1986)
7. Lukasiewicz, T.: Probabilistic logic programming. In: ECAI, pp. 388–392 (1998)
8. Lukasiewicz, T., Kern-Isberner, G.: Probabilistic logic programming under maximum entropy. In: Hunter, A., Parsons, S. (eds.) ECSQARU 1999. LNCS (LNAI), vol. 1638, p. 279. Springer, Heidelberg (1999)
9. Dekhtyar, A., Dekhtyar, M.I.: Possible worlds semantics for probabilistic logic programs. In: Demoen, B., Lifschitz, V. (eds.) ICLP 2004. LNCS, vol. 3132, pp. 137–148. Springer, Heidelberg (2004)
10. Ng, R.T., Subrahmanian, V.S.: Probabilistic logic programming. Information and Computation 101(2), 150–201 (1992)
11. Cohen, J., Hickey, T.: Two algorithms for determining volumes of convex polyhedra. Journal of the ACM 26(3), 401–414 (1979)
12. Khachiyan, L.: The problem of calculating the volume of a polyhedron is enumerably hard. Russian Mathematical Surveys 44(3), 199–200 (1989)
13. Dyer, M.E., Frieze, A.M.: On the complexity of computing the volume of a polyhedron. SIAM Journal on Computing 17(5), 967–974 (1988)
14. Dyer, M., Frieze, A., Kannan, R.: A random polynomial-time algorithm for approximating the volume of convex bodies. Journal of the ACM 38(1), 1–17 (1991)
15. Lovász, L., Vempala, S.: Simulated annealing in convex bodies and an $O^*(n^4)$ volume algorithm. Journal of Computer and System Sciences 72(2), 392–417 (2006)
16. Biieler, B., Enge, A., Fukuda, K.: Exact volume computation for polytopes: A practical study. In: Polytopes: Combinatorics and Computation. Birkhauser, Basel (2000)
17. Montenegro, R., Tetali, P.: Mathematical aspects of mixing times in markov chains. Foundations and Trends in Theoretical Computer Science 1(3), 237–354 (2006)
18. Applegate, D., Kannan, R.: Sampling and integration of near log-concave functions. In: ACM STOC, New Orleans, USA, pp. 156–163. ACM, New York (1991)
19. Lovász, L., Vempala, S.: Hit-and-run from a corner. In: ACM STOC, Chicago, IL, USA, pp. 310–314. ACM, New York (2004)
20. Khuller, S., Martinez, M.V., Nau, D., Simari, G., Sliva, A., Subrahmanian, V.: Computing most probable worlds of action probabilistic logic programs: Scalable estimation for $10^{30,000}$ worlds. AMAI 51(2–4), 295–331 (2007)

Non-discriminating Arguments and Their Uses*

Henning Christiansen[1] and John P. Gallagher[1,2]

[1] Computer Science, Building 43.2, P.O. Box 260, Roskilde University, DK-4000 Roskilde, Denmark
[2] IMDEA Software, Madrid
{henning,jpg}@ruc.dk

Abstract. We present a technique for identifying predicate arguments that play no role in determining the control flow of a logic program with respect to goals satisfying given mode and sharing restrictions. We call such arguments *non-discriminating* arguments. We show that such arguments can be detected by an automatic analysis. Following this, we define a transformation procedure, called *discriminator slicing*, that removes the non-discriminating arguments, resulting in a program whose computation trees are isomorphic to those of the original program. Finally, we show how the results of the original program can be reconstructed from trace of the transformed program with the original arguments. Thus the overall result is a two-stage execution of a program, which can be applied usefully in several contexts; we describe a case study in optimising computations in the probabilistic logic program language PRISM, and discuss applications in tabling and partial evaluation. We also discuss briefly other possible ways of exploiting the non-discriminating arguments.

1 Introduction

The first result presented here is the identification of predicate arguments that play no role in determining the control flow of a logic program computation, with respect to initial goals satisfying given mode and sharing restrictions. We call such arguments *non-discriminating* arguments. The non-discriminating arguments can be given either manually or determined by an automatic analysis.

Following this, we define a transformation procedure, called *discriminator slicing*, that removes the non-discriminating arguments, resulting in a program whose computations are isomorphic to those of the original program. The transformation can be performed on a whole program or on individual modules, assuming that mutually recursive modules do not occur.

We present a technique for decomposing the execution of a program into two stages. The first stage executes a simplified transformed program called a *mode-sliced* program, that establishes the control flow. The second stage performs computations omitted in the first stage. We present certain practical and

* This work is supported by the project "Logic-statistic modelling and analysis of biological sequence data" funded by the NABIIT programme of the Danish Strategic Research Council.

conceptual benefits of this two-stage execution. Non-discriminating arguments could be used in other ways, though we focus here on transforming a program.

Removing non-discriminating arguments generates a simpler program whose control flow mirrors that of the original program. The simpler program can be executed, yielding a trace of its execution. From that trace, together with the eliminated non-discriminating arguments, the results of executing the original computation can be reconstructed by re-running the trace but including the computations for the non-discriminating arguments.

There are various uses of this two-stage process, which might at first sight appear simply to do the same work as the original computation, and even with some additional overhead. We show how the technique can lead to overall optimisation. The simpler first stage can be of benefit in tabled computations. We show such a case in the probabilistic logic program language PRISM [18].

The paper is structured as follows. In Section 2 we define the concept of a discriminating argument, along with its relation to mode and sharing abstractions. Then the slicing of a program, cutting out non-discriminating arguments, is described. In Section 3 it is shown that slicing preserves computation traces, and a two-phase execution scheme is introduced along with an illustrative example. In Section 4 a particular application is studied, namely the calculation of the most probable sequence of states in a hidden Markov model programmed in PRISM; in this application exponential speedup can be achieved, due mainly to savings in the tabled structures constructed. Sections 5 and 6 contain a discussion on the applicability of the method and related work, and Section 7 concludes.

2 Preliminaries

We follow the standard terminology and notation for logic programs [10]. For now, we consider definite logic programs that allow calls to declarative built-in predicates.

Modes. We define *mode* abstractions $\{\mathsf{v}, \mathsf{nv}\}$ having the following interpretation given by a function mode. $\mathsf{mode}(\mathsf{v})$ is the set of variables and $\mathsf{mode}(\mathsf{nv})$ is the set of non-variables. $p(m_1, \ldots, m_n)$ is a *moded atom* if p is an n-ary predicate symbol and $m_j \in \{\mathsf{v}, \mathsf{nv}\}, 1 \leq j \leq n$. A finite set of moded atoms for predicate p is called a *mode* for p. We extend mode to atoms and set of atoms; $\mathsf{mode}(p(m_1, \ldots, m_n)) = \{p(t_1, \ldots, t_n) \mid t_i \in \mathsf{mode}(m_i), 1 \leq i \leq n\}$, and $\mathsf{mode}(M) = \bigcup\{\mathsf{mode}(p(\bar{m})) \mid p(\bar{m}) \in M\}$. We say that an atom A *respects* a mode M if $A \in \mathsf{mode}(M)$.

Sharing. We adopt a variant of a standard technique [19] for representing *possible sharing* among arguments of a predicate. A pair of terms $\{t_1, t_2\}$ is said to *share* if $vars(t_1) \cap vars(t_2) \neq \emptyset$, where $vars(t)$ denotes the set of variables occurring in term t. A pair-sharing abstraction (hereafter called simply a sharing abstraction) for an n-ary predicate p is given by a subset of $\{\{i, j\} \mid i, j \in \{1, \ldots, n\}, i \neq j\}$. A sharing abstraction S for n-ary predicate p denotes a set of atoms given by $\mathsf{share}(S) = \{p(t_1, \ldots, t_n) \mid$ if $\{t_i, t_j\}$ share then $\{i, j\} \in S\}$, or equivalently as $\mathsf{share}(S) = \{p(t_1,\ldots,t_n) \mid \{i,j\} \notin S$ implies $\{t_i,t_j\}$ do not share$\}$. Thus a sharing abstraction represents *possible* sharing between the given argument pairs, or equivalently definite *independence* of pairs of arguments that are absent. Namely, if $\{i,j\} \notin S$ for some sharing abstraction S, then for all atoms $p(t_1,\ldots,t_n) \in \mathsf{share}(S)$, t_i and t_j share no variable. We say that an atom A *respects* a sharing abstraction S for its predicate if $A \in \mathsf{share}(S)$.

Definition 1. *A* (argument) discrimination *for an n-ary predicate p is an atom $p(d_1,\ldots,d_n)$, where $d_i \in \{\mathsf{d},\mathsf{nd}\}$, where* d *stands for a* discriminating *argument and* nd *stands for a* non-discriminating *argument. An argument discrimination for a program Π is a set of discriminations, one for each p/n defined in Π.*

The intention of a discrimination is to identify which arguments in a predicate have an effect on the computation flow. A discriminating argument is one which could fail to match with the corresponding argument in at least one clause head. Conversely, a non-discriminating argument is one that does not influence the success of unification of a call with any clause head. The next definition makes this precise.

Definition 2. *Given a program Π and a goal A, a* discrimination for Π is correct *for the computation of Π with A if the following condition holds. For every call $p(t_1,\ldots,t_n)$ arising in the computation, and standardised-apart clause head $p(u_1,\ldots,u_n)$,*

– *$mgu(((t_1,\ldots,t_n),(u_1,\ldots,u_n))$ succeeds iff $mgu((t_{i_1},\ldots,t_{i_k}),(u_{i_1},\ldots,u_{i_k}))$ succeeds, where $\{i_1,\ldots,i_k\}$ is the set of discriminating arguments of p.*

Example 1. Consider the usual append program.

```
append([],Ys,Ys).
append([X|Xs],Ys,[X|Zs] :- append(Xs,Ys,Zs).
```

Then `append(d,nd,nd)` is a correct discrimination for the goals `append([a],U,V)` and `append([a],[b],V)`. That is, only the first argument determines whether a call matches a clause head, even though the second argument may also be non-variable.

Since Definition 2 is given in terms of the set of all calls arising in a computation, it is necessary to find some sufficient conditions in practice, since the set of calls is infinite in general. The next definition characterises a correct discrimination with respect to a program with a set of calls denoted by a mode abstraction and a sharing abstraction.

Definition 3. *Let Π be a program, M a mode and S a sharing abstraction. Let p be a predicate and $p(d_1,\ldots,d_n)$ a discrimination for p. Then the* discrimination is correct *with respect to M and S if for every standardised-apart clause head $p(u_1,\ldots,u_n)$ and $p(m_1,\ldots,m_n) \in M$, the set of pairs $\{\langle u_i,m_i\rangle \mid 1 \leq i \leq n\}$ satisfies the following condition;*

– *for all* $1 \leq i \leq n$, *if* $d_i = \mathsf{nd}$ *then* u_i *is a variable or* $m_i = \mathsf{v}$*; and*
– *there do not exist* $\langle u_i, m_i \rangle$ *and* $\langle u_j, m_j \rangle$, $i \neq j$, *such that* $d_i = \mathsf{nd}$, $m_i = \mathsf{v}$ *and* $\{i, j\} \in S$, *and* u_j *is non-variable; and*
– *there do not exist* $\langle u_i, m_i \rangle$ *and* $\langle u_j, m_j \rangle$, $i \neq j$, *such that* $d_i = \mathsf{nd}$, u_i *is a variable,* $m_j = \mathsf{nv}$ *and* $\mathsf{share}(u_i, u_j)$*.*

Informally, consider a call $p(t_1, \ldots, t_n)$ that satisfies the mode and sharing declarations for p, and a (standardised apart) clause head $p(u_1, \ldots, u_n)$. Then for each non-disriminating argument position i, according to Definition 3, at least one of t_i and u_i is a variable. Furthermore, if t_i is a variable and shares with some other argument t_j then u_j is a variable; and if u_i shares with some other argument u_j then t_j is a variable.

Example 2. Consider again the append program, the mode {`append(nv,nv,v)`} and sharing abstraction $\emptyset$ for `append`. Then `append(d,nd,nd)` is a correct discrimination. Note that argument 2 is non-discriminating despite the fact that it is non-variable. Although in clause head `append([],Ys,Ys)` arguments 2 and 3 share, they are not both matched to non-variables.

The following lemma states that a correct discrimination with respect to a mode and sharing abstraction (Definition 3) safely approximates the condition of Definition 2.

Lemma 1. *Let Π be a program, M a mode and S a sharing abstraction. Assume that for all A respecting M and S, every call in the computation of Π with A respects M and S. Let Δ be a discrimination for Π. Then if Δ is correct with respect to M and S for each predicate in Π then Δ is correct with respect to Π and A, for all A respecting M and S.*

Proof (Sketch). It can be verified that Definition 3 ensures that any call respecting the given mode and sharing cannot fail due to unifying the non-discriminating arguments. When unifying on a non-discriminating argument at least one of the two terms is variable. Thus the only possible cause of failure (since the two terms are standardised apart) is another occurrence of the variable that is matched to a different term. This cause is excluded by the conditions of Definition 3. It follows that every call unifies with each clause head if and only if the discriminating arguments unify, as required by Definition 2.

Constructing a discrimination from mode and sharing abstractions. Given a mode and sharing abstraction, we can construct a discrimination that satisfies the conditions of Definition 3.

Let Π be a program, M a mode and S a sharing abstraction; construct a discrimination for each predicate p as follows. The i^{th} argument of p is nd if and only if either (a) the i^{th} argument of each clause head for p is a variable, and if the i^{th} argument shares with another head argument then that argument is also a variable, or (b) the mode of the i^{th} argument is v in all mode atoms and if $\{i, j\} \in S$ then $m_j = \mathsf{v}$ also. Note also that a discrimination can be relaxed by replacing any nd by d, preserving correctness.

Example 3. Consider the usual append program, the mode {`append(nv,nv,v)`} and sharing abstraction $\emptyset$ for `append`. Then we derive `append(d,nd,nd)` as the discrimination. As there is no sharing in the calls, the 2nd argument is non-discriminating even though it is non-variable. Given the mode {`append(nv,v,v)`} and the same sharing abstraction we obtain `append(d,nd,nd)` once again. Given the mode {`append(v,v,nv)`} and the same sharing abstraction, we obtain the discrimination `append(nd,nd,d)`.

Analysis for Discrimination. Static analyses for freeness and sharing are well established, starting with [19,4] and we can apply them for automatically constructing correct discriminations. Given an initial moded call and a sharing abstraction on the call, such an analysis returns, for each predicate, a safe mode and sharing abstraction; that is, one respected by every call. An analysis is performed for a given computation rule; in this paper we assume the standard left-to-right, depth-first strategy. As discussed later, more precise analyses could be employed to derive more accurate discriminations than the one described here, which is based on very simple modes and no information on term structure.

2.1 Discriminator Slicing

Let Π be a program and Δ a discrimination for Π which is correct with respect to a mode and sharing abstraction for Π.

An argument position p_i where p is an n-ary predicate and $1 \leq i \leq n$ is *deletable* if

- that argument position is `nd` in Δ, and
- no occurrence of that argument position in the program contains a term that shares with an argument that is marked `d`.

A *slice* with respect to Δ is obtained by replacing each clause $A_0 \leftarrow A_1, \ldots, A_n$ in Π by $A'_0 \leftarrow A'_1, \ldots, A'_n$ where A'_i is obtained by deleting from A_i all deletable arguments. We call the resulting program Π^{Δ}.

Discriminator slicing with respect to a predicate. Slicing with respect to a predicate p allows the removal of body atoms in the clauses for p. A body atom can be removed if it cannot influence the choice of clause for p, or some predicate mutually recursive with p, in a computation,

Let Π be a program and Δ a discrimination for Π which is correct with respect to a mode and sharing abstraction for Π. Let G_Π be the predicate dependency graph of Π and $c_0, c_1, \ldots, c_l$ be the sequence of strongly connected components [21] of G_Π in some topologically sorted order (where c_l is the "top" component). Let $A_0 \leftarrow A_1, \ldots, A_n$ in Π be a clause whose head has predicate p and let c_k be the component containing p. Then A_i $(i \leq i \leq n)$ is deletable if

- A_i's component is c_j where $j < k$, and
- no argument of A_i shares with any `d` argument of an atom A_m, $m \neq i$, where A_m's predicate is in c_k.

Let Π^{Δ}_{p} be the program obtained by first constructing Π^{Δ} and then removing any deletable atoms (with respect to predicate p).

Discriminator slicing with respect to built-ins or imported predicates. built-in predicates are considered to be at the bottom of the predicate dependency graph. Thus slicing with respect to a predicate p allows removal of calls to built-ins from p's clauses that cannot affect p's control flow. However the same principle allows imported predicates can be handled in a similar way, assuming that mutually recursive predicates are not in separate modules. Thus predicate based slicing can be used to remove calls to imported predicates from a module if they cannot affect the control flow of the predicates of the module.

3 Discriminator Slicing and the Preservation of Traces

We now deal with the question of what properties of a program are preserved by discriminator slicing. The overall answer is that the control flow is preserved. To make this precise we introduce derivation trees and trace terms [3].

Derivations. A single moded atom is assumed for the top predicate, and a query to a given program consists, for simplicity, of a single call to the top predicate respecting its mode. The following characterisation of derivations and trace trees is adapted from [3].

Definition 4. *An* AND-tree *(for program Π) is a tree each of whose nodes is labelled by an atom and a clause, such that*

1. *each non-leaf node is labelled by a clause $A \leftarrow A_1, \ldots, A_k$ and an atom $A\theta$ (for some substitution θ), and has children $A_1\theta, \ldots, A_k\theta$,*
2. *each leaf node is labelled by a clause $A \leftarrow true$ and an atom $A\theta$ (for some θ).*

It was shown by Stärk [20] that A has answer θ in program Π if and only if there is an AND-tree (for Π) with root node labelled by $A\theta$.

Furthermore, a successful derivation with left-right depth-first computation rule (or any other computation rule) can be transformed into an AND-tree. Each AND-tree can be associated with a trace term.

Definition 5. *Let T be an AND-tree; define $\alpha(T)$ to be either*

1. *f_j, if T is a single leaf node labelled by the unit clause identified by f_j; or*
2. *$f_i(\alpha(T_1), \ldots, \alpha(T_n))$, if T is labelled by the clause identified by f_i/n, and has immediate subtrees $T_1, \ldots, T_n$.*

Adding trace-terms to programs. Trace-terms can easily be added to logic programs, so that the computation returns a trace term as well as its normal result. Let Π be a program and let the i^{th} clause be $p(\bar{t}) \leftarrow q_1(\bar{t}_1), \ldots, q_{a_i}(\bar{t}_{a_i})$. Let f_i/a_i be the function symbol associated with the i^{th} clause. Transform each such clause to $p(\bar{t}, f_i(Y_1, \ldots, Y_{a_i})) \leftarrow q_1(\bar{t}_1, Y_1), \ldots, q_{a_i}(\bar{t}_{a_i}, Y_{a_i})$, where $Y_1, \ldots, Y_{a_i}$ are distinct variables not occurring elsewhere in the clause.

Finally, transform each atomic goal $\leftarrow q(\bar{s})$ to $\leftarrow q(\bar{s}, W)$, where W is a variable not occurring elsewhere in the goal. Given a program Π we denote the program obtained by adding trace terms by Π^+.

We can modify the trace term to incorporate built-ins and imported predicates for which the clauses are not available. We simply define a unique constant for each such call and add it to the trace term. E.g. if $q_j(\bar{t}_j)$ in the clause $p(\bar{t}) \leftarrow q_1(\bar{t}_1), \ldots, q_j(\bar{t}_j), \ldots, q_{a_i}(\bar{t}_{a_i})$ is a call to a built-in, we transform the clause to $p(\bar{t}, f_i(Y_1, \ldots, b, \ldots, Y_{a_i})) \leftarrow q_1(\bar{t}_1, Y_1), \ldots, q_j(\bar{t}_j), \ldots, q_{a_i}(\bar{t}_{a_i}, Y_{a_i})$, where b is the unique identifier for that built-in.

Preservation of traces by discriminator slicing. Consider the result of discriminator slicing; some arguments are removed from the program but otherwise the structure of clauses is intact. Assume that the same clause identifiers are retained in the sliced program. Then the AND-trees of the sliced program are in one-to-one correspondence with those of the original program. This implies that, if we add trace-terms to both programs as in Definition 5, then the two programs generate exactly the same trace terms.

Proposition 1. *Let Π be a program and A a goal. Let Π^+, A^+ be the result of adding trace terms to Π and A. Let Δ be a correct discrimination for Π; Π^Δ, A^Δ the discriminator slice, and $\Pi^{\Delta^+}, A^{\Delta^+}$ the result of adding trace terms. Then there is an execution of A^+ in Π^+, yielding trace term t if and only if there is an execution of A^{Δ^+} in Π^{Δ^+} yielding t.*

Proof (Sketch). The sliced program Π^{Δ^+} uses only the discriminating arguments but a call unifies with a clause head in Π^{Δ^+} iff the corresponding call with all arguments unifies with the corresponding clause head in Π^+. Clearly the trace terms do not influence the control flow. Hence the computation follows exactly the same course and so the same trace terms are generated.

In the case of a slice with respect to a predicate, the trace term is preserved apart from the subterms corresponding to the deleted atoms. We state the corresponding correctness result.

Proposition 2. *Let Π be a program and A a goal. Let Π^+, A^+ be the result of adding trace terms to Π and A. Let Δ be a correct discrimination for Π and p a predicate; Π_p^Δ, A^Δ the discriminator slice wrt p, and $\Pi_p^{\Delta^+}, A^{\Delta^+}$ the result of adding trace terms. Then there is an execution of A^+ in Π^+, yielding trace term t if and only if there is an execution of A^{Δ^+} in $\Pi_p^{\Delta^+}$ yielding t', where t' is obtained from t by deleting the subterms corresponding to the deleted atoms.*

Re-running a trace. Having generated a trace term t (using A^{Δ^+} in $\Pi_p^{\Delta^+}$), we can then insert t into the trace term argument of the goal for the original goal A^+ in Π^+. The trace term will force the computation to re-run exactly the same path as given in the trace term. While there may have been backtracking while generating the trace term, the re-run using a ground trace term is completely deterministic.

Furthermore, some computation on failed branches is possibly avoided in the sliced program due to the deletion of atoms that do not affect the control flow. When re-running the program there is no backtracking. Thus it is possible that there is an overall saving in executing a program in two stages; the first phase to generate a successful trace using the sliced program, but doing less work in the unsuccessful branches than would have occurred in the original program. The second phase generates the full answer using the trace, with no redundant computation.

Example 4. We consider an implementation of a finite state machine which moves from states to states and consumes/emits a letter in each transition; for simplicity, we let the state machine treat all different letters alike. The top predicate `m(`*history*`,` *letter-sequence*`)` where the *history* records, for each step consecutively numbered, the states passed and letters seen. For example, the letter sequence `[a, b]` may give rise to the history `[s(0,q1), e(0,a), s(1,q2), e(1,b), s(2,end)]`. We assume the following mode pattern indicating that the program is used for mapping sequences of letters into histories, `m(v, nv)`; the program Π is as follows.

```
m(H,Ls):- m(q0,0,H,Ls).
m(end,N,[s(N,end)],[]).
m(Q,N,[s(N,Q),e(N,L)|H],[L|Ls]):-
  suc(Q,Q1), N1 is N+1, m(Q1,N1,H,Ls).
suc(q0,q1). suc(q0,q3).
...
```

A discrimination for this program is given as follows; it can be derived automatically from a sharing and freeness analysis with respect to the goal `m(v,nv)` with no sharing between the arguments.

`m(nd,nd)`, `m(d,nd,nd,d)`, `suc(d,nd)`,

We make now a two-stage transformation of this program, producing a sliced version Π^{Δ}_{m} with respect to the predicate `m/4`. Notice the call to built-in "`is`" is removed by this transformation, since `N1 is N+1` processes only variables that are non-discriminating in `m/4`. We then extend this with trace terms, corresponding to the production of $\Pi^{\Delta^+}_{m}$.

```
mDeltaT(Ls,f1(T)):- mDeltaT(q0,Ls,T).
mDeltaT(end,[],f2).
mDeltaT(Q,[L|Ls],f3(T1,T2):-
  sucDeltaT(Q,Q1,T1), mDeltaT(Q1,Ls,T2).
sucDeltaT(q0,q1,f4).  sucDeltaT(q0,q3,f5).
...
```

Finally, we generate the trace term version of the original program, corresponding to Π^{+}.

```
mPlus(H,Ls,f1(T)):- mPlus(q0,0,H,Ls,T).
mPlus(end,N,[s(N,end)],[],f2).
mPlus(Q,N,[s(N,Q),e(N,L)|H],[L|Ls],f3(T1,is,T2)):-
   sucPlus(Q,Q1,T1),
   N1 is N+1, mPlus(Q1,N1,H,Ls,T2).
sucPlus(q0,q1,f4).  sucPlus(q0,q3,f5).
...
```

Instead of querying the original program by *(i)* `m(H,`*a-list-of-letters*`)`, we can pose the query equivalently as follows,

(ii) `?- mDeltaT(`*a-list-of-letters*`,T), mPlus(H,`*the-same-list-of-letters*`,T).`

For measuring the difference in runtime, we detailed a state machine that needs to explore combinatorially many failing branches before escaping through an end state. While *(ii)* may look more complicated, it runs about 20% faster than *(i)*.[1] To explain this difference, notice firstly that the call to `mPlus` is negligible wrt. runtime as it executes deterministically for a correct trace. The unifications in `mDeltaT` all involve patterns mentioned in the head of clauses, so that the Prolog compiler can reduce them to very little work at runtime. Finally, the queries to `m` and `mPlusT` involve exactly the same number of failing and successful branches, so the speedup reflects the difference in efficiency of the single clauses.

It is clear that an arbitrarily large speedup can be demonstrated by applying this technique to suitably constructed programs with heavy use of built-ins and backtracking. In fact, as noted, the technique can be generalized to remove also calls to program defined predicates or imported predicates.

4 Discriminator Slicing in Tabling Systems, Including PRISM

We now discuss a case study in which drastic speedup is achieved using discriminator slicing in relation to tabled logic programming systems.

Very briefly, we can explain tabling [16] as a mechanism utilized in the execution of Prolog programs that maintains a table of successful calls and their answers, and whenever a call is encountered, it is checked if it (or perhaps a more general call) is known in the table already; if so, there is no need to execute it once again as the answers are ready in the table.

Comparing the use of tabling for a program Π and its reduced version Π^{Δ}, we notice that different calls of Π differing only in the non-discriminating arguments, will merge into a single call in Π^{Δ}. Thus the table can be much smaller, and there is a larger chance that the current call has a match in the table.

[1] This test was made using the optimizing compiler of SICStus Prolog 4.0.4 under Mac OS X 10.5.6, 2.4 GHz Intel Core 2 Duo with 4GB RAM.

We have applied this principle in a preprocessor for the probabilistic-logic PRISM [18] system, where in some cases, it reduces time complexity from exponential or worse to linear for applications of the system's generic Viterbi algorithm. For a detailed explanation of PRISM's facilities we refer to its manual [17]; PRISM is based on BProlog [25] from which it inherits a tabling mechanism.

Programs in PRISM are basically Prolog programs extended with random variables, called multi-valued switches. With given probabilities for the switches, a probabilistic semantics is induced that associates a probability to each true atom in the program so that a program becomes a probabilistic model. One possible application of a PRISM program is to find the set of outcomes of switches that provides the maximum probability, called the Viterbi probability, for a given observation represented as a top-level goal g; this can be done by a call `viterbif(`g`)`. Another useful option is `viterbig(`g`)`, which instantiates the variables in g to terms that provide the highest Viterbi probability. These predicates are given in alternative versions that also provide the Viterbi probability as well as an "explanation" which basically is a representation of the proof tree, including outcomes for the switches, that gives rise to the Viterbi probability.

The `viterbig` facility is interesting, among others applications, for prediction of structures in genomic sequence data. Many different models can be used and PRISM appears as a very flexible tool for developing such models. Here we will illustrate a Hidden Markov Model [14] (HMM) which can be represented as a predicate `hmm(`*annotation*`,`*sequence*`)` where *sequence* is a sequence of the letters a, c, g, t, in length between hundreds and in principle up to billions, and *annotation* is a description of those structures that the biologists find interesting (e.g., proposed positions of genes or detailed intron-exon structures).

A call such as `viterbig(hmm(A,`*sequence*`))` typically leads to a combinatorial explosion, but with our program slicing method we can achieve linear complexity which is the best possible. We will explain the details for the following PRISM program, Π_{HMM}; it defines a general HMM which records the sequence of states during a run. The most probable sequence is the so-called Viterbi path.[2]

```
values(letter(_state), [a,c,g,t]).
values(next_state(_state), [q1,q2,end]).
hmm(A,S):- hmm(q1,A,S).
hmm(end,[end],[]).
hmm(Q,[Q|Qs],[L|S]):-
   Q \= end,
   msw(letter(Q),L), msw(next_state(Q), Q1),
   hmm(Q1,Qs,S).
```

We have left out additional program lines that set the probabilities for the switches defined by the values declarations; notice that a parameterized pattern such as `letter(_state)` indicates a family of random variables. The `msw` predicate is a reference to a switch and may instantiate its second argument to any outcome of the switch.

[2] This code is inspired by similar examples in the PRISM manual [17].

The strategy applied for Viterbi computations in PRISM is to explore all possible proof trees being stored as a so-called explanation graph, but using sharing of subtrees whenever their top nodes are identical; this sharing is implemented through clever use of the underlying tabling mechanism in a way that we shall not describe here. When the program above is used for finding the best path for a given sequence, calls of the form

$$\texttt{hmm}(s_k,\ \texttt{Qs},[\ell_k,\ldots,\ell_n])$$

are made to the recursive predicate for all k and possible value of s_k. PRISM considers all possible answers for it, and they are entered in the underlying table; these answers amount to all possible instances given by substitutions of the form

$$\texttt{Qs} \rightarrow [s_k,\ldots,s_n].$$

The explanation graph needs to include all correspondingly instantiated nodes, of which there are clearly exponentially many.

On the other hand, the annotation arguments are non-discriminating and can be removed while still retaining the same set of proof trees (modulo mappings to adds/remove the annotation arguments). However, without the annotations far more nodes can share; more precisely the exponential amount of different nodes for each k and s_k reduces to a single node in the graph. To see this, consider reduced program $\Pi^{\Delta}_{\mathsf{HMM}}$ which is as follows.

```
hmm(S):- hmm(q1,S).
hmm(end,[]).
hmm(Q,[L|S]):-
   Q \= end,
   msw(letter(Q),L), msw(next_state(Q), Q1),
   hmm(Q1,S).
```

The recursive calls are now of the form $\texttt{hmm}(s_k\ ,[\ell_k,\ldots,\ell_n])$ where all arguments are grounded, thus only one possible answer, namely the empty substitution corresponding. This means that the explanation graph can be viewed as a structure a width equal to the number of different states and a length equal to the sequence length. The construction of this graph can be done in time linear in the size sequence length.

It is now so fortunate that `viterbif` can return a representation of the best proof tree extracted as a subgraph of the explanation graph. This tree can then be mapped into a desired annotation in one efficient run of a program such as our Π^{+}_{HMM} adapted for PRISM's particular proof tree format.

We have developed a little preprocessor, called `autoAnnotations` [1], which given a program Π_{HMM} as above, produces automatically $\Pi^{\Delta}_{\mathsf{HMM}}$ as well Π^{+}_{HMM}. The current version of this system requires the user to indicate the arguments and body calls to be removed. With the analysis methods described here, this can be done fully automatically from a single mode declaration for the top predicate.

We tested runtime for Viterbi computations with the reduced and non-reduced version. While the non-reduced version did not return an answer for sequences of length 20 within several hours, the reduced one $\Pi^{\Delta}_{\mathsf{HMM}}$ followed by our post-processor Π^{+}_{HMM} could produce Viterbi paths for lists of lengths up to 20,000 in few minutes on a machine with sufficient amount of RAM.

5 Discussion

It is clear that the slicing technique and two-phase execution can slow a program down. In the worst case no arguments or atoms are deletable so the program will be executed twice in its entirety. On the other hand there are examples such the one described above where spectacular speedup is achieved. Thus the technique needs to be used with care and targeted to appropriate examples. It seems hard to characterise precisely the class of programs that could benefit from its application. Applications that require explicit manipulation of computation trees or traces could be candidates; these might include online partial evaluation, where an explicit representation of computation trees are constructed and some notion of tabling is used to handle infinite branches of the tree [12,7]. In the case of the Viterbi calculation in Section 4, a key point is that one computation path (the most probable) is returned but the whole tree must be constructed first. Thus savings while constructing the tree are worthwhile. Search problems in which some structure is computed while searching for a solution are also liable to optimisation, as discussed in Example 4. The transformation can eliminate cases where partial solutions are constructed on failing branches of the search and then thrown away. Other examples in this class are non-deterministic parsers constructing a syntax tree. It is likely that the arguments of the parser constructing the syntax tree are non-discriminating; thus it could be a substantial optimisation for highly non-deterministic grammars to generate a trace of a successful parse and then deterministically construct the syntax tree afterwards.

Apart from optimisation, there could be applications of discriminator slicing for refactoring of logic programs [23]. The sliced program expresses the control part of the program. Thus two predicates p_1, p_2 that have isomorphic discriminator slices with respect to p_1, p_2 respectively could be said to share the same control. Such information could be useful for understanding, documenting or comparing programs. Furthermore the idea of recording and replaying program traces has applications in several other fields, especially debugging and understanding of concurrent programs [15].

6 Related Work

Program slicing [24,22] is a collection of techniques for removing parts of the code of a program that are irrelevant with respect to some chosen part of the program's semantics (the *slicing criterion*). Most slicing criteria concern the values of some selected variables; by contrast our slicing technique preserves control flow. However *path slicing* [5] is more similar; it concerns removing code that does not affect a given computation path; we differ in that our slicing criterion is the set of all computation paths rather than a specific one, and in that the underlying analyses are different from those used in an imperative language. We have done some initial investigation on the formulation of our approach as a value-based slicing technique using trace terms. The approach is to construct a slice of the program augmented with trace terms (Section 3) with respect to the trace term argument; the slice in principle contains only the operations needed to preserve the trace terms, that is, the control flow. Using a logic program slicing approach such as Leuschel and Vidal's [9] incorporating partial evaluation and the redundant argument filtering transformation [8] it is possible to handle simple examples but it is unclear at present whether the analyses incorporated in these tools are powerful enough.

Applications of mode and sharing analysis are many, including automatic parallelisation [13], occur-check elimination [19,4], non-failure analysis [2] and determinacy analysis [11]. The latter two applications resemble discriminator analysis in some respects and give pointers to obtaining more precise discriminations. The conditions for detecting non-failure, for example, are that each call unifies with at least one clause head, while the conditions for detecting determinacy are that each call unifies with exactly one clause head. Adapting these conditions and applying them argumentwise should lead to conditions expressing, for each argument or group of arguments, whether they unify with each clause head. If these conditions are true for some argument and every clause head, then that argument is non-discriminating. Further study is required.

We chose to apply discriminations to obtain a sliced program, but the same information could be applied dynamically during execution. In this respect there is a relation to control flow generation. Automatic generation of delay mechanisms and reordering of subgoals in clause bodies for improving efficiency and termination properties of logic programs have been considered by [6]. These techniques are somewhat orthogonal to ours, but we notice that our method may be adapted to generate delay declarations for calls which, in the construction of our reduced programs, are subject to deletion. Consider the finite state machine program of Example 4 in which the call `N1 is N+1` can be removed in the reduced program. Instead of deleting this call, we may delay it by an inline application of `freeze` or by replacing it with a call `addOne(N1,N)` where the new predicate is defined as follows.

```
:- block addOne(?,-).
addOne(N1,N):- N1 is N+1.
```

We can obtain an improved efficiency by only executing calls to `addOne` that occur in a successful execution of the program, by changing the top clause of the program into the following.

```
m(H,Ls):- m(q0,Zero,H,Ls),Zero=0.
```

(This step employs the data flow analysis of the program). An optimizing compiler and a runtime system with low overhead for delays may produce a program that runs almost as efficiently as the reduced version. It is not clear at present whether the approach of [6] will detect this opportunity for optimisation; the reordering of calls in the body for an optimised execution in their approach may possibly be utilized in order to classify more arguments as non-discriminating.

7 Conclusion

We have presented the concept of discriminating arguments and shown that they may be detected automatically given a moded goal. A slicing transformation was defined in which the slice preserves the computation tree structure. Using trace terms we then defined a two-phase execution in which the control flow is first established and then the full results are generated from the trace. Applications where this approach could be beneficial were presented and discussed.

Acknowledgements. We thank Yoshitaka Kamaya, Ole Torp Lassen, Pedro López-García and Manuel V. Hermenegildo for useful discussions.

References

1. Christiansen, H.: Efficient Viterbi for PRISM models with annotations (2009), Website with source code, http://www.ruc.dk/~henning/PRISMannotations
2. Debray, S.K., López-García, P., Hermenegildo, M.V.: Non-failure analysis for logic programs. In: Naish, L. (ed.) Proceedings of the Fourteenth International Conference on Logic Programming, ICLP 1997, pp. 48–62. MIT Press, Cambridge (1997)
3. Gallagher, J.P., Lafave, L.: Regular approximation of computation paths in logic and functional languages. In: Danvy, O., Glück, R., Thiemann, P. (eds.) Dagstuhl Seminar 1996. LNCS, vol. 1110, pp. 115–136. Springer, Heidelberg (1996)
4. Jacobs, D., Langen, A.: Accurate and efficient approximation of variable aliasing in logic programs. In: Lusk, E.L., Overbeek, R.A. (eds.) Logic Programming, Proceedings of the North American Conference 1989, NACLP 1989, pp. 154–165. MIT Press, Cambridge (1989)
5. Jhala, R., Majumdar, R.: Path slicing. In: Sarkar, V., Hall, M.W. (eds.) Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation. PLDI 2005, Chicago, IL, USA, June 12-15, pp. 38–47 (2005)
6. King, A., Martin, J.C.: Control generation by program transformation. Fundam. Inform. 69(1-2), 179–218 (2006)
7. Leuschel, M.: Advanced logic program specialisation. In: Hatcliff, J., Mogensen, T.Æ., Thiemann, P. (eds.) DIKU 1998. LNCS, vol. 1706, pp. 271–292. Springer, Heidelberg (1999)
8. Leuschel, M., Sørensen, M.H.: Redundant argument filtering of logic programs. In: Gallagher, J.P. (ed.) LOPSTR 1996. LNCS, vol. 1207, pp. 83–103. Springer, Heidelberg (1997)
9. Leuschel, M., Vidal, G.: Forward slicing by conjunctive partial deduction and argument filtering. In: Sagiv, M. (ed.) ESOP 2005. LNCS, vol. 3444, pp. 61–76. Springer, Heidelberg (2005)
10. Lloyd, J.: Foundations of Logic Programming, 2nd edn. Springer, Heidelberg (1987)
11. López-García, P., Bueno, F., Hermenegildo, M.V.: Determinacy analysis for logic programs using mode and type information. In: Etalle, S. (ed.) LOPSTR 2004. LNCS, vol. 3573, pp. 19–35. Springer, Heidelberg (2005)

12. Martens, B., Gallagher, J.P.: Ensuring global termination of partial deduction while allowing flexible polyvariance. In: Sterling, L. (ed.) Proc. International Conference on Logic Progrmaming (ICLP 1995), Tokyo, MIT Press, Cambridge (1995)
13. Muthukumar, K., Hermenegildo, M.: Determination of Variable Dependence Information at Compile-Time Through Abstract Interpretation. In: Lusk, E.L., Overbeek, R.A. (eds.) Logic Programming, Proceedings of the North American Conference 1989, NACLP 1989, October 1989, pp. 166–189. MIT Press, Cambridge (1989)
14. Rabiner, L.R.: A tutorial on hidden Markov models and selected applications in speech recognition. Proceedings of the IEEE 77(2), 257–286 (1989)
15. Ronsse, M., Bosschere, K.D., Christiaens, M., de Kergommeaux, J.C., Kranzlmüller, D.: Record/replay for nondeterministic program executions. Commun. ACM 46(9), 62–67 (2003)
16. Sagonas, K.F., Warren, D.S.: A portable compiler for integrating HiLog into Prolog systems. In: Bruynooghe, M. (ed.) Logic Programming, Proceedings of the 1994 International Symposium, p. 682. MIT Press, Cambridge (1994)
17. Sato, T.: PRISM, Programming in Statistical Modeling (PRISM web site) (checked 2009), http://mi.cs.titech.ac.jp/prism/
18. Sato, T., Kameya, Y.: Parameter learning of logic programs for symbolic-statistical modeling. Journal of Artificial Intelligence Research (JAIR) 15, 391–454 (2001)
19. Søndergaard, H.: An application of abstract interpretation of logic programs: Occur check reduction. In: Robinet, B., Wilhelm, R. (eds.) ESOP 1986. LNCS, vol. 213, pp. 327–338. Springer, Heidelberg (1986)
20. Stärk, R.F.: A direct proof for the completeness of SLD-resolution. In: Börger, E., Büning, H.K., Richter, M.M. (eds.) CSL 1989. LNCS, vol. 440, pp. 382–383. Springer, Heidelberg (1990)
21. Tarjan, R.E.: Depth-first search and linear graph algorithms. SIAM Journal of Computing 1(2), 146–160 (1972)
22. Tip, F.: A survey of program slicing techniques. J. Prog. Lang. 3(3) (1995)
23. Vanhoof, W.: Searching semantically equivalent code fragments in logic programs. In: Etalle, S. (ed.) LOPSTR 2004. LNCS, vol. 3573, pp. 1–18. Springer, Heidelberg (2005)
24. Weiser, M.: Program slicing. IEEE Trans. Software Eng. 10(4), 352–357 (1984)
25. Zhou, N.-F.: B-Prolog web site (1994–2009), http://www.probp.com/

Preprocessing for Optimization of Probabilistic-Logic Models for Sequence Analysis

Henning Christiansen and Ole Torp Lassen

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
{henning,otl}@ruc.dk

Abstract. A class of probabilistic-logic models is considered, which increases the expressibility from HMM's and SCFG's regular and context-free languages to, in principle, Turing complete languages. In general, such models are computationally far too complex for direct use, so optimization by pruning and approximation are needed. The first steps are taken towards a methodology for optimizing such models by approximations using auxiliary models for preprocessing or splitting them into submodels. Evaluation of such approximating models is challenging as authoritative test data may be sparse. On the other hand, the original complex models may be used for generating artificial evaluation data by efficient sampling, which can be used in the evaluation, although it does not constitute a foolproof test procedure. These models and evaluation processes are illustrated in the PRISM system developed by other authors, and we discuss their applicability and limitations.

1 Introduction

Models for data analysis are often based on probability theory which provides a firm theoretical basis and a catalogue of well-understood computational methods, which may be exact or approximative. Hidden Markov Models (HMM) and Stochastic Context-Free Grammars (SCFG) are well-known and popular techniques For analysis of genomic sequences in biology and other comprehensive sequential data sets. Both models consists of a logical and a probabilistic part, where the logical part of a HMM is a finite state automaton and for a SCFG, a Context-Free Grammar; see [7] for overview and detailed references. The logical part defines a space of possible analysis results, and the probabilistic part assigns a probability to each such with the understanding that high probability is a good or close-to-actual-truth result. There exists efficient Viterbi algorithms for HMM's that can find the state path of highest probability for a given sequence in linear time, whereas SCFG requires cubic times to identify a best parse tree.

There is a strong interest in the automated extraction of information from genomic and other biological sequence data due to demanding applications in medicine, biological sciences, food industry, etc..., and there are basically two complementary ways to approach this problem. Firstly, the computational power can be increased (e.g., by huge clusters of computers). Secondly, and this is the direction that we pursue, more powerful and detailed models of higher formal expressibility can be developed as a basis for novel and more sophisticated analyses. This may increase computational complexity drastically, and such new models must be accompanied by optimizations in order to gain practical value.

Our main goal with the present work is to promote the application of new strong models defined in declarative languages such as extensions to PROLOG. This provides the familiar advantages of declarative programming; it is known that standard models such as HMM and SCFG are embedded in natural ways, they can be extended and combined in a very flexible manner, and long-distance context-sensitive dependencies can be modelled using logical variables and arbitrary auxiliary data-structures and predicates. While this extends up to in principle Turing complete languages, even the cubic complexity that arises for a model that incorporate elements of a SCFG is problematic for long sequences. As a working example, we experiment with optimising and approximating an essentially context-free model to reduce its overall complexity. The method under investigation quite likely extents to even more complex problems but our current experiments does not go beyond context-free features.

While most of our constructions do not reflect the actual domain, our primary interest is on applications in computational biology, and our analysis takes into account the particular restrictions imposed by this. We focus currently on models that can be expressed in the PRISM system [16,17], but our general framework is independent of the particular formalism used.

We suggest that it would be advantageous for probabilistic-logic models to be developed by initially producing models that represent all the available knowledge about the phenomena in focus as faithfully as possible, exploiting the advantages of a powerful modelling language, and without being limited by implementation issues. Such ideal and declarative models may be used for initial testing of very small data sequences, and as representation of intellectual knowledge. Furthermore, they can be used for generating artificial samples of sequences-with-annotations which may be useful for investigating the models and for testing as we suggest below. Finally, these models can serve as specifications for, and standard of, the development of other, approximating models which can be used for efficient analysis of real data. The work described here represents the early steps taken towards this overall end, focussing on general methodological efficiency rather than on domain specific accuracy.

These efforts are part of a larger research project concerned with logic-statistical models that involves computer scientists, biologists, bioinformaticists, developers of competitive software for bioinformatic applications, and a leading company producing biological cultures to the food industry world-wide (the LoSt project [14]). Another big challenge in this project, that is not touched upon in the present paper, is to learn how to apply these models for real biological problems.

We consider in the present paper approximations based on preprocessing. For example, an efficiently implemented HMM may be used as a preprocessor to provide "steering marks" for a more advanced analysis. In fact, this is quite similar to tagging and period separation in natural language analysis.

Section 2 defines what we mean by a probabilistic-logic annotation model and states our assumptions about them; section 3 explains how such models can be defined and executed in PRISM as this is the context for the present experiments. In section 4, we define pre-annotation models and explain how they can utilized for optimized execution for prediction. Section 5 describes the particular problems involved in evaluating the quality of an approximating analysis independently of any specific domain of application. Section 6 describes an experiment that defines a simplified model for analysis of genomic sequences together with an approximating model; we consider also the suitability for these examples of evaluation based on sampling and suggest some improvements of the basic principle. Finally follows a short review of related work and conclusions.

2 Probabilistic-Logic Annotation Models

We consider probabilistic models that describe relationships between observed data and annotations that capture hidden information or "semantics" embedded in the data, and the intention is to use such models for computing the most probable annotations for given sequences.

In our motivating applications, S will be a sequence of letters, but other sorts of data structures may fit with the formal definitions as well; for simplicity of usage, we continue to refer to the S argument as a sequence. A is an annotation represented as a list identifying, say, the start and end positions for genes in S or a more detailed level of introns and exons etc., depending on the biological researcher's choice. An annotation may also represent an entire parse tree in case of a model that capture sophisticated features such as secondary or, in a proper setting, tertiary RNA-structures. In section 6 we will describe a model of a particular kind of secondary RNA structure.

Definition 1. *An* annotation model $m = \langle L, P\rangle$ *consists of a* logical part L *which is a set of ground atoms of a predicate* $m(A, S)$ *and a* probabilistic part P *which is a probability distribution over* L*, i.e.,* $0 \leq P(C) \leq 1$ *for any* $C \in L$ *and* $\sum_{C\in L} P(C) = 1$*. For given* S*, we let* $P(S) = \sum_A P(A, S)$*.*

Whenever $m(A, S) \in L$*, we say that* A *is an* annotation *of* S*; when, for no* $m(A', S) \in L$ *that* $P(m(A', S)) > P(m(A, S))$*, we say that* A *is a* best *annotation of* S*.*

We refer to the process of finding a best annotation for given sequences as *prediction.*

The use of probabilistic models for prediction is based on an assumption that there is a correlation between high probability and good score with standard measurements such as precision and recall; however, the latter is hypothetical when no authoritative test data are available, which is typically the case in computational biology.

3 Probabilistic-Logic Models in the PRISM System

PRISM is a powerful system developed by other authors [16,17] for working with a particular sort of probabilistic-logic models, based on an extension to Prolog with discrete random variables, called multi-valued switches. Consider as an example the following declaration,

```
values(letter, [a,c,g,t]).
```

It creates a switch, which means that whenever `msw(letter,X)` is called within the execution of a program, a value among `[a,c,g,t]` is assigned to `X`, understood as the outcome of a random variable which is independent of any other such variable, including other calls to the same switch. Switch declarations can also be parametrized as shown in the example below.

As shown by [15], an assignment of probabilities to the possible outcomes of each switch induces a probability distribution over program's least Herbrand model, and thus fits with our definition above; (a program needs to satisfy a few natural restrictions for this to be true, but this is of no concern here).

Example 1. We define here a simple HMM as a PRISM program; it has internal states `s1` and `s2` and a special `end` state; `s1` is also the start state. Each state emits probabilistically one of the same four letters, but potentially with different probabilities. We extend it to an annotation model that records a history of states that were visited during the creation of a given sequence.

```
values(letter(_state), [a,c,g,t]).
values(next_state(_state), [s0,s1,end]).

hmm(A,S):- hmm(s1,A,S).
hmm(end,[],[]).
hmm(Q,[Q|Qs],[L|S]):-
   msw(letter(Q),L), msw(next_state(Q), Q1),
   hmm(Q1,Qs,S).
```

Probabilities can be set explicitly or by means of learning algorithms built into PRISM; we ignore these details here and assume simply that some fixed probabilities are given. As an example, PRISM's semantics assigns the following probability to the atom `hmm([s1,s2],[a,a])`,

$$P(\texttt{msw(letter(s1),a)}) \times P(\texttt{msw(next\_state(s1),s2)}) \times$$
$$P(\texttt{msw(letter(s2),a)}) \times P(\texttt{msw(next\_state(s2),end)}).$$

Stochastic context-free grammars can be defined in an equally concise way as the HMM in example 1 above by having a switch for each nonterminal whose values represent the rules for that nonterminal. It is straightforward to extend a grammar so that it builds annotations in the shape of parse trees or mark-ups of specific subsequences. PRISM includes devices for calculating probabilities of a given atom as well as generalized Viterbi algorithms that make it possible to obtain the best annotation for given sequence. The time complexity for these varies with the inherent complexity of the given model as well as the degrees of freedom in the specific queries; up to certain sequence lengths, Viterbi computations for HMM's are linear in the length of the sequence and for SCFG's cubic which is known to be theoretically best (however involving a substantial constant factor).

Programs with annotations as the one shown in example 1 above should not be given directly to PRISM for Viterbi computations, as it will lead to a combinatorial explosion. We will not go into explaining why this is the case, but simply indicate that it is possible to run Viterbi in linear time for HMM programs with annotations taken out, and then reconstruct the annotations from a proof tree produced by PRISM; [5] explains a tool based on automatic program annotations so that the PRISM model developer does not have to care about these subtleties.

4 Optimization by Pre-annotations

A central notion in our approach is that of a pre-annotation, which restricts the class of possible annotations for a given sequence.

Definition 2. *Given an annotation model* $m = \langle L, P\rangle$*, a* pre-annotation *model is a model* $m^{pre} = \langle L^{pre}, P^{pre}\rangle$ *equipped with a* projection function π *such that for any* $m(A, S) \in L$*, there is a unique pre-annotation* A^{pre} *with* $m^{pre}(A^{pre}, S) \in L^{pre}$ *and* $\pi(A) = A^{pre}$*.*

A best annotation *of* S given pre-annotation A^{pre}*, is an annotation* A *with* $\pi(A) = A^{pre}$ *for which there is no other* A' *with* $\pi(A') = A^{pre}$ *and* $P(m(A', S)) > P(m(A, S))$*.*

In practice, a pre-annotation can be given as an additional argument to the predicate of a model or as a partially instantiated argument.

Pre-annotations may provide ways to improve the time complexity of prediction. This may be achieved when the time complexities of both

- calculating a best pre-annotation A^{pre} and
- calculating a best annotation A given A^{pre}

are strictly less than the time complexity of calculating the best annotation A^* for S in m. However, for this to be useful, the possible deviations between A and A^* as well as between $P(A, S)$ and $P(A^*, S)$ must not be too large; we return later to the question of how to characterize and estimate this accuracy.

We can make analogies to natural language processing techniques for showing different kinds of pre-annotations. Tagging means to assign most likely word classes and inflections to each word in a text, such that the parser can start from there, analyzing sentence structures without bothering about these details. Another preprocessing is identification of the beginning and end of periods. For example, some occurrences of "." indicates end of a period while others serve other purposes, e.g., as in "e.g.". It is interesting to observe that HMM's are often used for tagging and period segmentation; see, e.g., [9].

Splitting a sequence into fixed subsequences that are then analyzed separately can be an effective way to reduce time complexity.

Example 2. Sequence analysis using SCFG is known in general to be of cubic complexity, and as we noted above, this can be obtained in PRISM. Often a SCFG has a natural distinction of subsequences covered by a subset of rules; let us refer to those as periods and assume they are limited by an upper length k.

Assuming now that we have an HMM which can produce reliable preannotations that splits the sequence into (proposed) periods of lengths at most k. While analysis of a sequence of length n with the SCFG takes time $\mathcal{O}(n^3)$, the use of the HMM to fix the periods which are then analyzed separately by the SCFG, reducing the time to $\mathcal{O}(n/k \cdot k^3) = \mathcal{O}(n)$ since, in all practical cases, k is much smaller than n.

In other words, suppose we can represent a complex model as a SCFG of the form (probabilities omitted):

$S \rightarrow A\ S \mid B\ S \mid \epsilon$
$A \rightarrow$ *production rules for* A-periods
...
$B \rightarrow$ *production rules for* B-periods
...

such that only the A-rules involves features requiring context-free analysis, while the rules for B-periods in principle could be stated as a HMM instead. Then M can simply be reformulated as the interaction of three interacting submodels:

1. a HMM for S of states A and B each initiating the corresponding submodel,
2. a SCFG for A-type periods, and
3. a HMM for B-type periods.

This sort of two-stage analysis is straightforward to implement in PRISM using two successive calls to Viterbi predicates.

5 Estimating the Quality of an Approximating Model

5.1 Difficulties

Testing of tools for biological sequence analysis is a particularly difficult task, as independent test data with trusted annotations are rare, and thus standard tests based on recall and precision measurements are generally not possible.

This problem comes from the fact that it is very expensive and time consuming to perform the necessary laboratory work in order to produce a correct annotation of a single sequence. Furthermore, the sparse trusted annotated data are often used as training data for the model in question, which also makes them dubious as test data. But when data are available, we can refer to the role of thumb of splitting known annotated data into training and validation data in order to avoid over-fitting.

As hinted in the introduction, we suggest that when comparing a complex model, m^c, that we may call canonical, and its approximating (and assumed efficiently executable) model m^a, we can use m^c for generating artificial, annotated test data for m^a.

In this way, we abstract away the real world and may compare the two models in an objective way. However, this sort of probabilistic sampling of pairs, $\langle ann, seq\rangle$, of annotation and sequence introduces another problem, namely that *ann* is not necessarily the annotation of highest probability for *seq*. In our experiments, we have often seen that even the approximating model can generate annotations of a higher probability (measured in m^c for comparison). The hard truth is that we have no way of checking whether a generated annotation really is a good one or how far it is from a best one. With sampling in this very big outcome space, it is in practice impossible to hit the same sequence more than once, so it will be an illusion to refer to the law of big numbers to get a picture of the distribution for the annotations of a given sequence. To make things even more difficult, we have also observed that there is very little correlation between the length of a sequence and the probability distribution for the annotations of a given sequence. Even within the same model, some sequences have a single obvious best annotation while others are highly ambiguous and can present multitudes of almost equally good ones, with correspondingly small probability mass left for each. In other words, it seems it is not possible to produce a meaningful aggregation of collections of sampled data and their probabilities.

5.2 Considering Generated Samples for Evaluation

While samples with annotations generated from a canonical model cannot take the role of affirmative test data, we will investigate what we are able to conclude from such testing.

Sampling has the advantage that one can start the computer, have thousands of samples generated and tested with the approximating model. Let us consider a single sample $\langle ann^{samp}, seq\rangle$ generated from the canonical model m^c. We can run *seq* through the approximative model in question, m^a, and obtain its best annotation ann^{approx} as described above.

To compare the two, we can measure their probabilities $P^{samp} = P^c(ann^{samp}, seq)$ and $P^{approx} = P^c(ann^{approx}, seq)$; notice that we measure both in the distribution of the canonical model in order obtain measurements in the same scale. A ratio P^{approx}/P^{samp} close to one indicate that the quality of the two annotations are similar.

As we have discussed, using precision and recall is not always possible in a general setting, so we may instead utilize a subjective measurement of the similarity between the two annotations.

In our experiment that follows, we apply a simple principle for measuring similarity that may apply independently of actual sort of annotations in questions.

Each annotation is mapped into a sequence of symbols of the same length as the sequence, indicating the findings of interest, and similarity is defined by the fraction of all such symbol that are identical for the two sequences. When the nested structures are important, the symbol sequence may indicate the structure using brackets; for example, two tree structures may be mapped into "`[-(-){-(-)}]`" and "`[-(-){(-)-}]`" that are identical in 8 of of 12 character, i.e., a similarity measure of 0.667. When only the classification of particular subsequences is interesting, e.g., distinguishing between genes and non-genes, this sort of measurement still gives score to annotation that differs slightly in the begin and end positions of the subsequences. So the similarity between "`nnnnnnngggggggggggnn`" and "`nnnnnnnggggggggggggn`" is 0.95.

6 A Simple Genefinder in PRISM and Its Evaluation by Sampling

We show here an example of a pair of canonical and approximative model that follows the pattern of example 2. The canonical model m^c is a simplified SCFG that resembles the sort of models that one will expect for genomic sequences for prokaryotic organisms. It distinguishes between subsequences considered as coding (genes) and non-coding; the non-coding parts are modelled basically by an HMM whereas rules for the coding parts include description of a particular kind of secondary RNA-structure called *hairpins* see figure 1, these hairpins do not manifest themselves in the actual genome but in the mRNA produced by the genes; so a good match with such structures could be perceived as indicating the likeliness of a gene. Because hairpins are inherent nested structures, a SCFG is necessary to describe them.

An approximating model m^a is comprised by an HMM that fixes the boundaries between coding and noncoding regions, and then applies different submodels of m^c for the each of the two kinds of subsequences.

As seen in example 2, m^c requires cubic times whereas m^a is linear. The difference in accuracy between the models can be explained as follows: if m^c is applied (hypothetically!) for prediction, it has the degree of freedom to move the coding/non-coding boundaries in order to get the optimal hairpin structure, whereas m^a fixes these boundaries first from a shallow analysis, and then finds the best analyses for the subsequences irrespective of their context.

We indicate here the details of the models and investigate how far we can get in an evaluation using the sampling strategy.

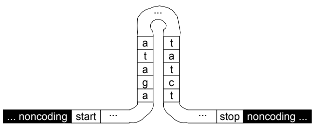


Fig. 1. *Hairpins* or *hairpin loops* consists of a *stem* of mutually abstracted letters and a *loop* of unpaired letters

6.1 Overview of the Models

The canonical model is described in two levels, a top level that is structured as a HMM with two states, for coding and non-coding regions; instead of emitting a single letter as a HMM, a call is made to a the relevant submodel. The submodel for noncoding regions is another HMM and the one for coding regions is a SCFG involving the hairpin structures indicated above.

A gene is initiated by a start codon and terminated by a stop codon[1], which is one of $\{\langle a,t,g\rangle,\langle g,t,g\rangle,\langle t,t,g\rangle\}$ and of $\{\langle t,a,a\rangle,\langle t,g,a\rangle,\langle t,a,g\rangle\}$, respectively; but there is no guarantee that a start-stop codon pair actually indicates a coding region. Furthermore, the possible lengths of coding regions are multiples of 3 as to fit with a codon structure. The hairpin structures in the coding regions of this model represent foldings in RNA that occur due to an attraction between the molecules represented by a and t, and between c and g. So for example, the subsequence $agata \ldots tatct$ may describe a hairpin, where the outermost 5+5 letters form a stem and those indicated by the dots form a loop at the top; see illustration below; for simplicity we do not allow nested (cacti-like) hairpin structures, although these occur frequently in nature.

The first phase of the approximating model m^a uses a HMM that includes start and stop codons, the multiple of 3 condition for coding regions and with potentially different distributions of letters between coding and non-coding regions. As indicated above (in section 4), this model is used for fixing a proposal for the boundaries of coding/non-coding and then the two submodels inherited from m^c takes over.

6.2 Evaluation by Sampling and Some Improvements of the Method

When generating samples with m^c, we have made an *ad hoc* improvement of the annotations which is possible for models such as m^c due to its clear subdivision into subsequences: for each such subsequence, we run a Viterbi computation with the relevant submodel and put this best sub-parse into the annotation instead of the sampled one. Be aware that this is different from the approximating model, as the sequences as well as boundary parts of the annotations are created from the unaltered m^c. We can measure, in the total probability, a clear improvement with this trick so our tests reported below may be a bit more accurate in sorting out bad annotations produced by m^c. However, the problem that sampled annotations are suboptimal is decreased but not eliminated.

We conducted four experiments, whose results are summarized in Fig. 2 and 3. Everywhere we measured the probabilities in log space, so the ratio is represented as a difference with 0 representing identity, and also the Hamming-like similarity measure (referred to as match percentages in the figures) which takes into account the deep syntactic structures of the individual parses, as indicated above. The scatterplots in Fig. 2 correlates these two measures; a dot represents a probability ratio together with match percentage; notice that dots to the left of the

[1] A *codon* corresponds to a triplet of three letters that, depending on context, may code for an amino acid, or serve a "control purpose" as start and stop codon.

zero line represent cases where the m^a found a more probable annotations than the one given by the improved m^c sampling.

Experiment a). Firstly we used uniform probabilities for all switches in the model and no selection among the samples. As could be expected, the combined plot indicates a diffuse distribution with the larger part of the results indicating that m^a provides the best annotations measured in probability. Thus we have no clear indication of how good m^a is compared with the objectively best samples (which, by nature of the setting, are unknown), and we may thus also doubt the value of the results where m^a provides a results close to the sampled one. However, the combined scatter plot is a bit misleading as dots may coincide, and the detailed plots indicate that this is the case, as most measurements are close to, or spot on, the ideal 0 resp. 100% marks.

Experiments b), c) and d). We changed the basic experiment in two directions in order to see if we could get different results when the models are made more realistic by 1) training the models using the 100 shortest annotated genes from E. Coli K12, whose lengths are between 50 and 178 letters; and/or 2) we constrained generated samples to those satisfying the inherent length constraint implied by the training data.

From Fig. 2 we see that training has a profoundly positive effect on the similarity of parses, and that constraing the samples to comply with the inherent length constraints of he domain of application affects the probability quotients of the individual analyses similarly. Thus experiment d) represents a much higher degree of correlation with far less occurrences of m^a suggesting annotations with higher probabilitites than those provided by improved sampling; this may tempt us to have more confidence to all sampled annotations and thus more confidence to an approximated annotation close to the sampled one. Both Fig. 2 and 3 indicate here that most approximated and sampled annotations correlates closely. Even with these deviations, the two meassures correlates perfectly in the vast majority of cases, coinciding in (0, 100%) coordinates, as shown in Fig 3. All in all this indicates that especially with the precautions taken in experiment d), we may trust to the approximating model produce reasonably reliable results.

These experiments showed that the sampling based tests provide a clear indication of the quality of an approximating model; it is also clear that the method works best for models with biased probabilities (so both m^a as predictor as m^c as generator are better to distinguish between good and bad, so to speak). Throwing away samples that do not respect the inherent constraints that also are expected in actual data to be analyzed, removes irrelevant observations from the statistics expressed in the diagrams; this also contributes to the improved reliability of the tests. The more critical issues of this testing method is the lack of quantitative summaries based on firm statistical considerations of how good the approximation is. In the completely general setting with no specific domain of applicaiton and sufficently annotated data, we doubt that such quantitative meassures can be devised.

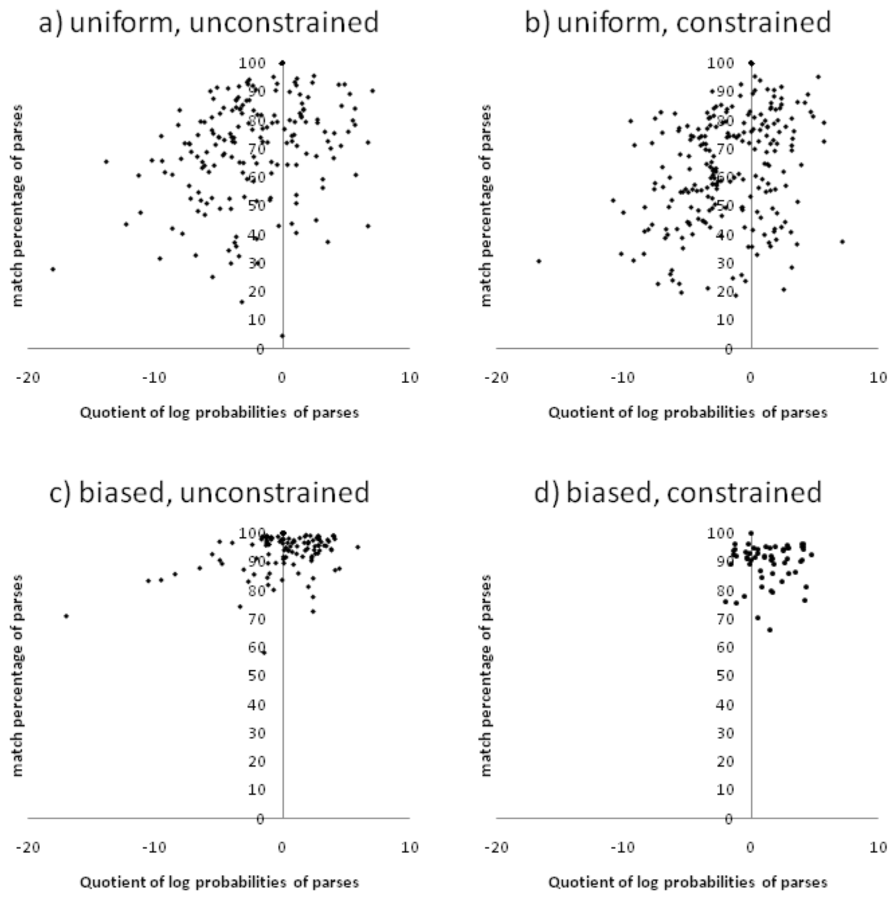


Fig. 2. Representation of the sampling distributions of the four conducted experiment. Along the X-axis are logarithmic probability quotients of the canonical vs. approximative analyses. A quotient of 0 represents identity. Along the Y-axis are the corresponding percentages of Hamming-like similarity between the two analyses (match percentages). Dots to the left of the Y-axis represent cases where the approximative analysis resulted in a more probable annotations than the one resulting from the improved canonical sampling. In *a)* is shown the plot for uniform models and unconstrained sampling. *b)* is the result of constraining samples to comply with the length range of the 100 shortest genes of E.Coli. In *c)* the models were trained using EM-learning on the aforementioned data from E.Coli. Finally, the plot in *d)* represents the analyses of samples from the trained model complying with the length constraints inherited from the training data. The plots in *a)* and *b)* illustrates the chaos of uniform parameters. The effect of training the models is easily observable in *c)*, causing the approximative and canonical analyses to agree to a much higher degree than in a) or b). In d) we see that forcing domain-specific constraints on the sampling increases the correlation between analysis and probability.

run	probability quotient = 0	match percentage = 100
a)	823/1000	822/1000
b)	839/1000	837/1000
c)	867/1000	865/1000
d)	947/1000	946/1000

Fig. 3. The degree of perfect correlation between canonical and approximative analyses according to the probability-quotient measure and the match-percentage measure respectively. This represents the number of dots that coincides in the coordinates (0,100%) of the scatter plots in Fig 2. Individually, both training and constraining increases the degree of correlation in both measures, but combining training and constraining results in a profound increase of about 10% in both measures.

7 Related Work

Bayesian networks, HMM's, and SCFG's are traditional methods for sequence analysis that can be seen as instances of probabilistic-logic models; while their flexibility for modelling and formal expressibility are far below the models we are aiming at (PRISM and similar), there exists a plethora of efficient algorithms and implemented systems; see [7] for background and overview. These provide a catalogue of possible preprocessors to be used within our approach.

More general and powerful formalisms have been suggested as extensions to logic programs or equally expressive formalisms within the last 15 years, we may mention PRISM [16,17] that we have exemplified, Stochastic Logic Programs [13], Stochastic functional Programs [10], and Relational Bayesian Networks [8].

There is a growing interest in such models for bioinformatical applications. In particular [1,3] discuss applications to Systems Biology, but also various kinds of sequence analysis have been studied (see [6] for an excellent survey.

We may refer to [4] as a precursor of the present work, where similar ideas are applied for a comparative test of three different genefinder programs [2,11,12]. A detailed PRISM model was built for parts of human genomic sequences, it was trained with known data, and then the trained model was used for producing artificial genomic sequences (complemented with manual editing for the parts not covered by the developed model). These data were used as test data, and the quality of the genefinders was evaluated with precision and recall measures. In this work, preprocessing analogously to what has been described in the present paper was used to produce auxiliary annotations for speeding up training.

8 Conclusion and Future Work

We advocate the use of probabilistic-logic models based on logic programs (or similarly expressive languages) as the basis for analysis of biological sequence data; due to flexibility and generality, such models are candidates for providing better and more detailed finds than the currently most used methods, however, this is still to be proved.

We intend that such models should be developed without consideration about performance in order to document in a formal way and as faithfully as possible, the available knowledge about the phenomena being modelled in what we called a canonical model.

Using preprocessors, e.g., based on existing and efficiently implemented technologies, as a way to reach realistic execution times, we intend to get the best of both worlds, flexibility and sophistication of the probabilistic-logic models combined with feasible execution times.

Implementation involving preprocessing to fix certain choices before a detailed analysis takes places, necessarily affects the accuracy, and we have discussed the possible ways of testing such implementation in cases when no authoritative test data is available. In the lack of such, and despite certain problems, the best thing we can do seems to be to employ the possibility of using the complex model for generation of annotated test data, and then compare with the annotations produced by the implemented approximative models. While comparing probabilities for the two annotations can give some indication, we found it best to use subjectively defined measures that compares the similarity between the sequences in a straightforward, syntactic fashion.

We noticed that the sampling method provide the best indications when the model has biased probabilities, as it is easier to distinguish between good and bad annotations. It also increases the chance that the annotations produced under sampling are of a reasonable quality. We noticed also the advantage of applying inherent constraints that are difficult to capture in probabilistic models, to sort out the relevant samples and run the comparison tests on those only. These constraints may typically concern the length of particular kinds of substrings, where sampling will produce annotated sequences that do not reflect nature (or the possible training data).

We intend to extend the methodology with a more firm statistically basis for evaluation of the measurements produced by the sampling principle.

Acknowledgement. This work is supported by the project "Logic-statistic modelling and analysis of biological sequence data" funded by the NABIIT program under the Danish Strategic Research Council, and the CONTROL project, funded by Danish Natural Science Research Council.

References

1. Biba, M., Ferilli, S., Mauro, N.D., Basile, T.M.A.: A hybrid symbolic-statistical approach to modeling metabolic networks. In: Apolloni, B., Howlett, R.J., Jain, L.C. (eds.) KES 2007, Part I. LNCS, vol. 4692, pp. 132–139. Springer, Heidelberg (2007)
2. Burge, C., Karlin, S.: Prediction of complete gene structures in human genomic DNA. Journal of Molecular Biology 268, 78–94 (1997)
3. Chen, J., Muggleton, S., Santos, J.: Abductive stochastic logic programs for metabolic network inhibition learning. In: Frasconi, P., Kersting, K., Tsuda, K. (eds.) MLG (2007)

4. Christiansen, H., Dahmcke, C.M.: A machine learning approach to test data generation: A case study in evaluation of gene finders. In: Perner, P. (ed.) MLDM 2007. LNCS, vol. 4571, pp. 742–755. Springer, Heidelberg (2007)
5. Christiansen, H., Gallagher, J.: Mode-based slicing and its applications (submitted, 2009)
6. De Raedt, L., Frasconi, P., Kersting, K., Muggleton, S. (eds.): Probabilistic Inductive Logic Programming. LNCS, vol. 4911. Springer, Heidelberg (2008)
7. Durbin, R., Eddy, S., Krogh, A., Mitchison, G.: Biological Sequence Analysis. Cambridge University Press, Cambridge (1998)
8. Jaeger, M.: Relational bayesian networks. In: Geiger, D., Shenoy, P.P. (eds.) UAI, pp. 266–273. Morgan Kaufmann, San Francisco (1997)
9. Jurafsky, D., Martin, J.H.: Speech and Language Processing, 2nd edn. Prentice-Hall, Inc., Upper Saddle River (2006)
10. Koller, D., McAllester, D.A., Pfeffer, A.: Effective bayesian inference for stochastic programs. In: AAAI/IAAI, pp. 740–747 (1997)
11. Krogh, A.: Using database matches with for HMMGene for automated gene detection in Drosophila. Genome Research 10(4), 523–528 (2000)
12. Lukashin, A., Borodovsky, M.: Genemark.hmm: new solutions for gene finding. Nucleic Acids Research 26(4), 1107–1115 (1998)
13. Muggleton, S.: Learning from positive data. In: Muggleton, S. (ed.) ILP 1996. LNCS, vol. 1314, pp. 358–376. Springer, Heidelberg (1997)
14. LoSt on the Web, `http://lost.ruc.dk`
15. Sato, T.: A statistical learning method for logic programs with distribution semantics. In: ICLP, pp. 715–729 (1995)
16. Sato, T., Kameya, Y.: Parameter learning of logic programs for symbolic-statistical modeling. J. Artif. Intell. Res (JAIR) 15, 391–454 (2001)
17. Sato, T., Kameya, Y.: Statistical abduction with tabulation. In: Kakas, A.C., Sadri, F. (eds.) Computational Logic: Logic Programming and Beyond. LNCS, vol. 2408, pp. 567–587. Springer, Heidelberg (2002)

Stabilization of Information Sharing for Queries Answering in Multiagent Systems

Phan Minh Dung, Do Duc Hanh, and Phan Minh Thang

Computer Science and Information Management
Asian Institute of Technology, Thailand
{dung,hanh,thangfm}@cs.ait.ac.th

Abstract. We consider multiagent systems situated in unpredictable environments. Agents viewed as abductive logic programs with abducibles being literals the agent could sense or receive from other agents, must cooperate to provide answers to users as they may not have the knowledge or the capabilities to sense relevant changes in their environment. As their surroundings may change unpredictably, agents may provide wrong answers to queries. Stabilization refers to a capability of the agents to eventually answer queries correctly despite unpredictable environment changes and the incapability of many agents to sense such changes.It could be viewed as the correctness criterium of communicating cooperative multiagent systems.

For efficiency, a piece of information obtained from other agents may be used to answer many queries. Surprisingly, this natural form of "information sharing" may be a cause of non–stabilization of multiagent systems. We formulate postulates and present a formal framework for studying stabilization with information sharing and give sufficient conditions to ensure it.

Keywords: Stabilization, Information Sharing, Abductive Logic Programs, Cooperative Multiagent Systems.

1 Introduction

Cooperative agents are entities that work together to achieve common objectives. To operate effectively in a changing and unpredictable environment, agents need correct information about their surroundings. Due to limited knowledge and sensing capability, agents need to cooperate with each other to get such information by sending requests and receiving replies. Stabilization, a key characteristics of cooperative multiagent systems, represents the capability of the agents to eventually get correct information ([4]).

Example 1. Consider a system of two agents A and B, where the knowledge base of A consists of the clause:

$$p \leftarrow q$$

while the knowledge base of agent B consists of the clause:

$$q \leftarrow f$$

where f is a fluent about the environment and could be sensed by only B. We assume B always notices any change on f instantly.

Suppose there is a user query from an external agent E to A on p. To answer it, A sends a query on q to B and waits for a reply. Suppose that f is true. B will send a reply *"q!" (q is true)* to A informing that q is true. In turn, after receiving "q!" from B, A sends an answer "p!" to E. E gets a correct answer for the query on p from A in this case.

The environment can change unpredictably. Suppose immediately after B sends reply "q!" to A, but before A sends reply "p!" to E, f changes to $false$. The information on q that A has received from B, becomes incorrect and so does the answer "p!" that E receives from A.

Of course if the environment (i.e. f) does not change anymore afterwards, a new query on p to A would be answered correctly because A would send a new request on q to B, then B would reply with "$\neg q$!" (*q is false*), and at last A would send "$\neg p$!" to E. Our system in this scenario is said to be **stabilizing**, i.e. **even though environment changes could cause temporarily incorrect answers to some queries, but once the environment stops changing, all queries will be answered correctly after some delay**.

Example 2 (Continuation of Example 1)

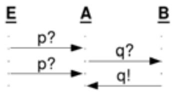


Fig. 1. Message Exchanges in Example 2

Consider the message exchanges in Figure 1. Should A send a new query "q?" to B to answer the second query "p?" from E?.

Naturally, A should not send another query "q?" to B. On receiving information about q from B in the fourth message, A uses it to answer both user queries. This **information sharing** is a natural common mechanism in real world multiagent systems (including humans) for efficiency. It is captured by the following postulate:

Postulate 1: If a request for some information has been made and an answer to it is being expected, then no request for the same information should be made now.

The environment can change unpredictably and information an agent has obtained from others may become incorrect without the agent being aware of it. To deal with this problem, the agent should not use information obtained from other agents forever. This motivates Postulate 2.

Postulate 2: No information obtained from other agents should be used forever.

When should input information be deleted? Ideally when it becomes false. The problem is that an agent may not know when the environment changes and so the agent may not know whether information obtained from other agents is false or not. For instance in the above examples there is no way agent A could know whether the information about q she has received from B is true or false.

The answer to the question could only be: "Information should be deleted some time after it has been received". Though this "some time" is domain-dependent, we could ask whether there is a "lower bound" for that time.

Come back to the scenario in Figure 1, we expect that the reply "q!" from B to A should be used to answer the two queries "p?" that have been received by A before. This motivates Postulate 3.

Postulate 3: Information obtained from other agents should be deleted only after all queries received before the information is obtained and to which the information is relevant, are answered.

The question we ask now is: "Would a multiagent system stabilize if Postulates 1 to 3 are satisfied?".

Unfortunately, in general multiagent systems do not stabilize even if Postulates 1 to 3 are all satisfied as the following example shows.

Example 3 Consider a system of two agents A and B where the knowledge base of A consists of the clause:

$$q \leftarrow \neg r(x)$$

and the knowledge base of B consists of the clauses:

$$r(x+1) \leftarrow r(x) \text{ and } r(0) \leftarrow \ .$$

Note that there is no environment change in this case.

Suppose A receives a query "q?" from some external agent. Obviously the correct answer to this query is "$\neg q$!". To answer the query "q?", A may first send a query "$\neg r(0)$?" to B and B replies with "$r(0)$!". A hence has to send another query e.g. "$\neg r(1)$?" to B and B replies with "$r(1)$!". The exchanges will continue that A will send a query "$\neg r(n+1)$?" after receiving a reply "$r(n)$!" from B and the information obtained by A is never sufficient to answer "q?". Hence A could never be able to answer the query "q?". The system is not stabilizing.

The purpose of this paper is to formalize the problem of stabilization in multiagent systems and study general conditions under which the stabilization with information sharing following Postulates 1 to 3 is guaranteed.

The rest of this paper is organized as follows. In section 2 we briefly introduce the basic notations, definitions and lemmas of acyclic and abductive logic programs and admissibility semantics that are needed through this paper. Problem formalization and results are presented in section 3. We summarize related works and conclude in section 4. Due to space limitation, proofs of lemmas and theorems are skipped in this paper.

2 Acyclic Logic Programs and Admissibility Semantics

We assume the existence of a Herbrand base HB. A normal logic program is a set of ground clauses of the form:

$$a \leftarrow l_1, \ldots, l_m$$

where a is an atom from HB, and $l_1, \ldots, l_m$ are literals (i.e. atoms or negations of atoms) over HB, $m \geq 0$. a is called the head, and $l_1, \ldots, l_m$ the body of the clause. Note that clauses with variables are considered as a shorthand for the set of all their ground instantiations.

Given a logic program P, $head(P)$ and $body(P)$ denote the sets of atoms occuring in the heads and the bodies of clauses of P respectively.

For each atom a, the **definition** of a in P is the set of all clauses in P whose head is a. A logic program is **bounded** if the definition of every atom is finite.

The **atom dependency graph** of a logic program P is a graph, whose nodes are atoms in HB and there is an edge from a to b in the graph iff there is a clause in P whose head is a and whose body contains b or $\neg b$.

A logic program P is **acyclic** iff there is no infinite directed path in its atom dependency graph.

An atom b is said to be **relevant** to an atom a in P if there is a path from a to b in the atom dependency graph of P. Further a literal l is relevant to another literal l' if the atom of l is relevant to the atom of l'.

Abusing notation we write $\neg l$ for complement of l, i.e. $\neg l$ is a if l is $\neg a$ and $\neg l = \neg a$ if l is a. Given a set S of literals, $\neg S = \{\neg l \mid l \in S\}$. A set of literals is *consistent* if it does not contain any pair of a literal and its complement.

Given a logic program P and a consistent set of literals S. We write $P \cup S \vdash l$ iff there is a sequence of literals $l_1, \ldots, l_n = l$, $n \geq 1$ such that for all $m = 1 \ldots n$: $l_m \in S$ or there exists a clause $l_m \leftarrow l'_1, \ldots, l'_k$ in P s.t. $l'_1, \ldots, l'_k \in S \cup \{l_1, \ldots, l_{m-1}\}$.

An **abductive logic program** is a tuple $\langle P, Ab \rangle$ where P is a logic program, Ab is a set of atoms in $body(P)$ called abducible atoms such that no element of Ab occurs in the head of any clause in P, i.e. $Ab \cap head(P) = \emptyset$. Literals over Ab called abducible literals or **abducibles** for short ([11], [10], [1]). $\langle P, Ab \rangle$ is **acyclic** if P is acyclic.

Let $\langle P, Ab \rangle$ be an abductive program. The set of **assumptions** $\mathbb{A}$ of $\langle P, Ab \rangle$ comprises Ab and the set of all negative literals including negative abducibles.

A set of assumptions S **attacks** a set of assumptions R if there is $\alpha \in R$ such that $P \cup S \vdash \neg\alpha$.

A set of assumptions is **admissible** if it does not attack itself and attacks every set of assumptions attacking it. It is not difficult to see that admissible sets of assumptions are consistent.

A **preferred extension** of $\langle P, Ab \rangle$ is a maximal (wrt set inclusion) admissible set of assumptions. Note that in difference to a normal acyclic logic program, an acyclic abductive logic program may have more than one preferred extensions.

Lemma 1. *Let $\langle P, Ab \rangle$ be an acyclic program. For each maximal consistent set S of abducibles there is a unique preferred extension E of $\langle P, Ab \rangle$ such that $S \subseteq E$.*

A set of assumptions $S \subseteq \mathbb{A}$ is a stable extension of $\langle P, Ab \rangle$ iff for every $a \in Ab$, either $a \in S$ or $\neg a \in S$ and for every $a \notin Ab$, either $P \cup S \vdash a$ or $\neg a \in S$

([3],[9]). Similarly to [3] we can show that each preferred extension of an acyclic abductive program is also stable.

A set S of abducibles is an **abductive solution (or explanation)** for a literal l wrt $\langle P, Ab\rangle$ iff there exists an admissible set of assumptions $S' \subseteq \mathbb{A}$ such that $S = S' \cap (Ab \cup \neg Ab)$ and $P \cup S' \vdash l$. It is not difficult to show that if there exists an abductive solution S for l wrt $\langle P, Ab\rangle$ then there exists a preferred extension E of $\langle P, Ab\rangle$ such that $S \subseteq E$ and $P \cup E \vdash l$.

An abduction solution for l wrt $\langle P, Ab\rangle$ is **non-redundant** if it contains only abducibles relevant to l in P.

As for every abductive solution S for a literal l wrt $\langle P, Ab\rangle$ there is a non-redundant abductive solution R for l wrt $\langle P, Ab\rangle$ such that $R \subseteq S$, we restrict our attention on non-redundant abductive solutions.

Example 4. Consider an abductive logic program $\langle P, Ab\rangle$ where

$$P = \{q \leftarrow r \quad q \leftarrow \neg p \quad p \leftarrow t\} \text{ and } Ab = \{r, t\}.$$

$S_0 = \{\neg t\}$ is an abductive solution for q wrt $\langle P, Ab\rangle$ as $S_0' = \{\neg p, \neg t\}$ is an admissible set and $P \cup S_0' \vdash q$. Note that $P \cup S_0 \not\vdash q$.

There are four preferred extensions $E_1 = \{r, t\}$, $E_2 = \{r, \neg p, \neg t\}$, $E_3 = \{\neg r, \neg p, \neg t\}$, $E_4 = \{\neg q, \neg r, t\}$ of $\langle P, Ab\rangle$ where $P \cup E_i \vdash q$, $i = 1 \ldots 3$.

In general an admissible set of assumptions is determined largely by its subset of abducibles as shown in the following lemmas.

Lemma 2. *If E is a preferred extension of an acyclic abductive program $\langle P, Ab\rangle$ and R is an admissible set of assumptions such that all abducibles in R are in E, i.e. $R \cap (Ab \cup \neg Ab) \subseteq E$, then $R \subseteq E$.*

Lemma 3. *Given an acyclic logic program $\langle P, Ab\rangle$ and a consistent set S of abducibles ($S \subseteq Ab \cup \neg Ab$). There is no abductive solution for a literal l wrt $\langle P, Ab\rangle$ consistent with S iff for every preferred extension $E \supseteq S$ of $\langle P, Ab\rangle$: $P \cup E \vdash \neg l$.*

There are many ALP systems and abductive proof procedures proposed in the literature (e.g. [5], [3], [7], [10], [1]). In this paper we do not consider complexity of ALP systems. We simply assume the availability of abductive solution generation algorithms.

3 Problem Formalization

Let l be a literal. A **query** whether l is true or a **reply** that l is true has a form $l?$ or $l!$ respectively.

3.1 Agent and Multiagent System

Agents are situated in environments. An agent could sense some of the changes of her surroundings though not all of them. Let ENV be a set of ground atoms representing the fluents of the environments.

A **multiagent system** is a pair $(\mathcal{A}, ENV)$ where $\mathcal{A}$ is a set of agents situated in an environment characterized by fluent atoms in ENV.

Definition 1 (Agent). *An* ***agent*** *situated in an environment ENV is represented by a quadtuple $A = (P, HBE, HBI, \Lambda)$ where*

- *P, is an acyclic logic program, representing the knowledge base of the agent.*
- *$HBE \subseteq ENV$, representing the sensing capability of the agent, is a set of environment atoms whose truth values the agent could sense. Atoms in HBE do not occur in the head of any clause of P.*
- *HBI is the set of input atoms, that occur in the body of some clause in P but not in the head of any clause of P and not in HBE, i.e. $HBI = body(P) \setminus (head(P) \cup HBE)$.*
- *Λ is the initial state of the agent and will be defined shortly.*

It is not difficult to see that $\langle P, HBE \cup HBI \rangle$ is an abductive logic program.

Definition 2 (Cooperative Multiagent System)
A multiagent system $(\mathcal{A}, ENV)$ with $\mathcal{A} = (A_1, \ldots, A_n)$, $A_i = (P_i, HBE_i, HBI_i, \Lambda_i)$ is cooperative iff the following conditions are satisfied:

- *$ENV = \bigcup_{i=1 \ldots n} HBE_i$, i.e. each environment change is sensed by some agent.*
- *For each atom a, if $a \in head(P_i) \cap head(P_j)$ then a has the same definition in P_i and P_j. In other words, agents' domain knowledge bases are consistent.*
- *For every agent A_i, for each $a \in HBI_i$, there is an agent A_j such that $a \in head(P_j) \cup HBE_j$, i.e. A_i can get the value of a or $\neg a$ from A_j.*
- *No environment atom appears in the head of clauses in the knowledge base of any agent, i.e. for all i: $ENV \cap head(P_i) = \emptyset$.*
- *A state of $(\mathcal{A}, ENV)$ is the collection of states of its agents and $(\Lambda_1, \ldots, \Lambda_n)$ is the initial state of $(\mathcal{A}, ENV)$.*

From now on, we focus solely on cooperative multiagent systems. Hence whenever we say "multiagent systems", we mean cooperative ones.

Definition 3 (Agent State). *A state of agent $A = (P, HBE, HBI, \Lambda)$ is a quintuple $\sigma = (EDB,\ RDB,\ SDB, IDB, t)$ where*

- *$EDB \subseteq HBE \cup \neg HBE$ is a maximal consistent set of environment literals containing information the agent has sensed from the environment.*
- *SDB is a database containing the send–out requests whose replies have not been received. An input literal l is inserted into SDB when A sends out a query "l?" and is removed from SDB when A receives a reply "l!" or "$\neg l$!".*
- *RDB is a database of received queries of the form $(\mathbf{sender}, \mathbf{query}, S, id)$ where*
  - **sender** *is the sender of the query;*
  - **query** *is of the form l? where l is a literal;*
  - *S is a non-redundant abductive solution for l wrt $\langle P, HBE \cup HBI \rangle$ or $\perp$ representing non-existence of abductive solutions;*
  - *id is a nonnegative integer used as the identification of the query. Each received query/reply is assigned a unique identification that is also served as a timestamp. The greater is the identification of a query/reply, the more recent the query/reply is received.*

- *IDB is a database containing input information A has obtained from other agents. It is a consistent set of input literals associated with identifications. When A receives a reply "$l!$", $\langle l, t\rangle$ is inserted into IDB and the timestamp counter t of A is increased by* 1*.*
- *t, a nonnegative integer, holds the current timestamp counter. In the initial state $t = 0$.*

Example 5. The multiagent system in Example 1 is represented by $(\mathcal{A}, ENV)$ where $\mathcal{A} = (A, B)$, $ENV = \{f\}$, $A = (P_A, HBE_A, HBI_A, \Lambda_A)$, $B = (P_B, HBE_B, HBI_B, \Lambda_B)$ and

$$\begin{array}{llll} P_A = \{p \leftarrow q\} & HBE_A = \emptyset & HBI_A = \{q\} & \Lambda_A = (\emptyset, \emptyset, \emptyset, \emptyset, 0) \\ P_B = \{q \leftarrow f\} & HBE_B = \{f\} & HBI_B = \emptyset & \Lambda_B = (\{f\}, \emptyset, \emptyset, \emptyset, 0) \end{array}$$

Let S be a set of literals and IDB be the input database in an agent state. Abusing the notation, we often say that $S \cup IDB$ is consistent meaning that $S \cup \{l \mid \langle l, id\rangle \in IDB\}$ is consistent.

Definition 4. *Given a state $\sigma = (EDB, RDB, SDB, IDB, t)$, let $\Theta = (X, l?, S, id)$, $S \neq \bot$, be a query form in RDB. Θ is* ***consistent*** *wrt σ if $S \cup EDB \cup IDB$ is consistent. Otherwise it is inconsistent wrt σ. Θ is* ***verified*** *wrt σ if $S \subseteq EDB \cup IDB$.*

3.2 Agent Actions and Environment Changes

Let $\sigma = (EDB, RDB, SDB, IDB, t)$ be the current state of an agent $A = (P, HBE, HBI, \Lambda)$. A state of A changes when the environment changes or A receives/sends a query/reply from/to another agent or deletes some inputs from IDB.

1. **Environment change**
   An environment change is represented by a pair $C = (T, F)$ where T (resp. F) contains the atoms whose truth values have changed from false (resp. true) to true (resp. false) and $T \cap F = \emptyset$. Given an environment change $C = (T, F)$, what agent A could sense of this change is a pair (T_A, F_A) where $T_A = T \cap HBE$ and $F_A = F \cap HBE$. Hence if a change $C = (T, F)$ occurs then A will update her environment database EDB to
   $$EDB' = (EDB \setminus (F_A \cup \neg T_A)) \cup T_A \cup \neg F_A$$
   The new state of A is denoted by
   $$Upe_A(\sigma, C) = (EDB', RDB, SDB, IDB, t).$$
2. **Receiving a query**
   When A receives a query "$l?$" from some agent X $(X \neq A)$, A will generate a query form $\Theta = (X, l?, S, t)$ where S is an abductive solution for l wrt $\langle P, HBE \cup HBI\rangle$ consistent with $EDB \cup IDB$ and insert Θ into RDB. If no such abductive solution exists, A will insert a query form $(X, l?, \bot, t)$ into RDB. The new received query database is denoted by RDB'.
   The new state of A is denoted by
   $$Upi_A(\sigma, l?, X) = (EDB, RDB', SDB, IDB, t + 1).$$

3. Receiving a reply

When receiving a reply "$l!$" from some agent, A updates IDB by deleting any input of the form $\langle l, id\rangle$ and $\langle \neg l, id\rangle$ from it and inserting $\langle l, t\rangle$ into it. The new input database is denoted by IDB'. A also removes l and $\neg l$ from SDB. The new sent–out database is denoted by SDB'.

The new state of A is denoted by

$$Upi_A(\sigma, l!) = (EDB, RDB, SDB', IDB', t + 1).$$

4. Sending out a query

Definition 5. *Let $\Theta = (X, l'?, S, id)$ be a query form in RDB.*
We say that A ***is ready to request information*** *$l?$ from B for Θ wrt σ, where l is an input literal, iff the following conditions are satisfied:*

1. *Θ is not verified wrt σ and*
   either *$S \cup EDB \cup IDB$ is consistent and $l \in S$*
   or *Θ is inconsistent wrt σ and there is a nonredundant abductive solution S' for l' wrt $\langle P, HBE \cup HBI\rangle$ consistent with $EDB \cup IDB$, $S' \not\subseteq EDB \cup IDB$ and $l \in S'$,*
2.
   a. *$l \notin SDB$ and $\neg l \notin SDB$ i.e. A is not waiting for replies for queries "$l?$" or "$\neg l?$" (**Postulate 1**).*
   b. *if $\langle l, id'\rangle$ or $\langle \neg l, id'\rangle$ occurs in IDB then $id' < id$ (**Postulate 1**).*[1]
3. *The atom of l is in $head(P_B) \cup HBE_B$ (queries are only sent to agents that can answer them).*

If A sends a request "$l?$" to B ($B \neq A$) for a query form $\Theta = (X, l'?, S, id) \in RDB$ in state σ then the following conditions are satisfied:

1. A is ready to request information $l?$ from B for Θ wrt σ.
2. If Θ is inconsistent wrt σ then Θ is replaced in RDB by a new query form $(X, l'?, S', id)$ where S' is a new generated abductive solution for l' wrt $\langle P, HBE \cup HBI\rangle$ consistent with $EDB \cup IDB$, $S' \not\subseteq EDB \cup IDB$ and $l \in S'$,

After sending out "$l?$" to B, A will insert l into SDB. The new received query and sent–out databases are denoted by RDB' and SDB' respectively. The new state of A is denoted by

$$Upo_A(\sigma, l?) = (EDB, RDB', SDB', IDB, t).$$

5. Sending out a reply

Definition 6. *Let $\Theta = (X, l?, S, id)$ be a query form in RDB.*
We say that A ***is ready to answer*** *Θ by "$l!$" wrt σ iff either Θ is verified wrt σ or if $S \cup EDB \cup IDB$ is inconsistent then there must be an abductive solution S' for l wrt $\langle P, HBE \cup HBI\rangle$ and $S' \subseteq EDB \cup IDB$.*

[1] Postulate 1 states that if A has been waiting for a reply $l!$ or $\neg l!$ then A should not send a query $l?$ or $\neg l?$. It implicitly implies that queries receiving before id' (with identification less than id') should use $\langle l, id'\rangle$ or $\langle \neg l, id'\rangle$ in their answers. Therefore, if a new request for l is made, it should come from queries receiving after id' (with identification greater than id').

*A **is ready to answer** Θ by "$\neg l!$" wrt σ iff either $S = \perp$ or there is no abductive solution for l wrt $\langle P, HBE \cup HBI\rangle$ consistent with $EDB \cup IDB$.*[2]

If A sends "$l!$" or "$\neg l!$" to X ($X \neq A$) in state σ then there must be a query form $\Theta = (X, l?, S, id)$ in RDB such that A is ready to answer Θ by $l!$ or $\neg l!$ wrt σ respectively.

After sending out reply "$l!$" or "$\neg l!$", A will remove Θ from RDB. The new received query database is denoted by RDB'.

The new state of A is denoted by

$$Upo_A(\sigma, l!, X) = (EDB, RDB', SDB, IDB, t).$$

6. Deleting possibly stale inputs

If A deletes an input $\langle l, id\rangle$ from IDB, then there is no query form $\Theta = (X, l'?, S, id')$ in RDB where l is relevant to l' and $id' < id$ (**Postulate 3**). A updates IDB to $IDB' = IDB \setminus \{\langle l, id\rangle\}$.
The new state of A is denoted by

$$Upd_A(\sigma, del(l)) = (EDB, RDB, SDB, IDB', t).$$

Example 6 (Continuation of Example 5). Consider the system in Example 1, 2 and 5 and the following table of changes in states of agents[3].

	Event	A					B				
		EDB	RDB	SDB	IDB	t	EDB	RDB	SDB	IDB	t
0		$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	0	f	$\emptyset$	$\emptyset$	$\emptyset$	0
1	$E, A, p?$	-	$(E, p?, \{q\}, 0)$	-	-	1	-	-	-	-	-
2	$A, B, q?$	-	-	q	-	-	-	$(A, q?, \{f\}, 0)$	-	-	1
3	$B, A, q!$	-	-	$\emptyset$	$\langle q, 1\rangle$	2	-	$\emptyset$	-	-	-

0. The initial states of A and B are shown in row 0.
1. A receives a query "$p?$" from an external agent E. A generates and adds query form $(E, p?, \{q\}, 0)$ into her received query database, and increases her timestamp counter by 1.
2. A sends out a query "$q?$" to B and adds q into her sent–out database. Receiving the query "$q?$", B generates and adds query form $(A, q?, \{f\}, 0)$ into her received query database, and increases her timestamp counter by 1.
3. B sends out a reply "$q!$" to A and removes the only query form from her received query database. Receiving the reply "$q!$", A inserts input $\langle q, 1\rangle$ into her input database and removes q from her sent–out database.

3.3 Runs

The semantics of a multiagent system is defined in terms of runs. A run of a multiagent system is an infinite sequence of transitions that occur when the environment changes or agents send out/receive queries/replies or delete possibly stale inputs from their input databases.

[2] Because by Lemma 3, for every preferred extension E of $\langle P, HBE \cup HBI\rangle$ consistent with $EDB \cup IDB$: $P \cup E \vdash \neg l$.

[3] "-" means unchanged and event "A, B, π" means that A sends π to B.

Definition 7 (Transitions). *Let $\Sigma = (\sigma_1, \ldots, \sigma_n)$ and $\Sigma' = (\sigma'_1, \ldots, \sigma'_n)$ be states of a multiagent system $(\mathcal{A}, ENV)$.*

1. *An environment transition $\Sigma \xrightarrow{C} \Sigma'$ happens when there is an environment change $C = (T, F)$ and the following conditions are satisfied:*
   - *For every $A_k \notin S_C$: $\sigma'_k = \sigma_k$ and*
   - *For each agent $A_i \in S_C$: $\sigma'_i = Upe_i(\sigma_i, C)$,*[4] *where S_C denotes the set of agents which could sense parts of C, i.e. $S_C = \{A_i \mid HBE_i \cap (T \cup F) \neq \emptyset\}$.*
2. *A query transition $\Sigma \xrightarrow{(X, A_i, l?)} \Sigma'$ happens when agent X sends a query on l to agent A_i and the following conditions are satisfied:*
   - *For every $A_k \notin \{X, A_i\}$: $\sigma'_k = \sigma_k$.*
   - *$\sigma'_i = Upi_i(\sigma_i, l?, X)$.*
   - *If $X = A_j$ then $\sigma'_j = Upo_j(\sigma_j, l?)$.*
3. *A reply transition $\Sigma \xrightarrow{(A_i, X, l!)} \Sigma'$ happens when A_i sends "$l!$" to X and the following conditions are satisfied:*
   - *For every $A_k \notin \{A_i, X\}$: $\sigma'_k = \sigma_k$.*
   - *If $X = A_j$ then $\sigma'_j = Upi_j(\sigma_j, l!)$.*
   - *$\sigma'_i = Upo_i(\sigma_i, l!, X)$.*
4. *An input delete transition $\Sigma \xrightarrow{(A_i, del(l))} \Sigma'$ happens when A_i deletes input $\langle l, id \rangle$ from IDB_i and the following conditions are satisfied:*
   - *For every A_k, $k \neq i$: $\sigma'_k = \sigma_k$.*
   - *$\sigma'_i = Upd_i(\sigma_i, del(l))$.*
5. *An empty transition $\Sigma \xrightarrow{nil} \Sigma'$ denotes that there is no change in the states of agents.*[5]

We often simply write $\Sigma \rightarrow \Sigma'$ if there is a transition from Σ to Σ'.

Definition 8 (Run). *Let $(\mathcal{A}, ENV)$ be a multiagent system. A run $\mathcal{R}$ of $(\mathcal{A}, ENV)$ is an infinite sequence of transitions*

$$\mathcal{R} = \Sigma_0 \rightarrow \Sigma_1 \rightarrow \cdots \rightarrow \Sigma_m \rightarrow \ldots$$

where $\Sigma_k = (\sigma_{1,k}, \ldots, \sigma_{n,k})$, $\sigma_{i,k} = (EDB_{i,k}, RDB_{i,k}, SDB_{i,k}, IDB_{i,k}, t_{i,k})$ such that

1. *Σ_0 is the initial state of $(\mathcal{A}, ENV)$.*
2. *There is a point h such that at every $k \geq h$ in the run, there is no more environment change.*
3. *$\mathcal{R}$ satisfies the following fairness condition.*[6] *For every agent A_i, for each k: there is no query form $\Theta = (X, l?, S, id)$ such that*

[4] We write $Upe_i(\sigma_i, C)$ for $Upe_{A_i}(\sigma_i, C)$.

[5] This transition is introduced to ensure that runs are infinite (See Definition [8]).

[6] The fairness condition ensures that actions of sending out request/reply are not delayed indefinitely if they are ready.

  - *for all $m \geq k$, $\Theta \in RDB_{i,m}$ and A_i is ready to request information from other agents for Θ wrt $\sigma_{i,m}$*[7] *or*
  - *for all $m \geq k$, $\Theta \in RDB_{i,m}$ and A_i is ready to answer Θ by $l!$ or $\neg l!$ wrt $\sigma_{i,m}$.*

4. *For every k, for every input $\langle l, id\rangle \in IDB_{i,k}$: there is a $m \geq k$ such that $\langle l, id\rangle$ is deleted at m (**Postulate 2**)*
   - *either explicitly by transition $\Sigma_m \xrightarrow{(A_i,del(l))} \Sigma_{m+1}$*
   - *or implicitly and replaced by $\langle l, t_{i,m}\rangle$ or $\langle \neg l, t_{i,m}\rangle$ by transition $\Sigma_m \xrightarrow{(A_j,A_i,l!)} \Sigma_{m+1}$ or $\Sigma_m \xrightarrow{(A_j,A_i,\neg l!)} \Sigma_{m+1}$ for some A_j in $\mathcal{R}$.*
5. *If $\Sigma_k \xrightarrow{nil} \Sigma_{k+1}$ then for all $m \geq k$: $\Sigma_m \xrightarrow{nil} \Sigma_{m+1}$.*

Example 7 (Continuation of Example 6).
The sequence in Example 6 is a part of the following run:

$$\Sigma_0 \xrightarrow{(E,A,p?)} \Sigma_1 \xrightarrow{(A,B,q?)} \Sigma_2 \xrightarrow{(B,A,q!)} \Sigma_3 \xrightarrow{(\emptyset,\{f\})} \Sigma_4 \xrightarrow{(A,E,p!)} \Sigma_5 \rightarrow \ldots$$

It is easy to see from Definitions 7, 8 and conditions for sending out queries and deleting inputs in section 3.2 that the following lemma holds.

Lemma 4. *Each run of a multiagent system satisfies Postulates 1 to 3 of information sharing.*

3.4 Superagent

The superagent of a multiagent system represents the combined capacity (both reasoning and sensing) of the multiagent system as the whole in the ideal case where all agents are instantly provided all necessary information (e.g. located at one place).

Let $(\mathcal{A}, ENV)$ be a multiagent system with $\mathcal{A} = (A_1, \ldots, A_n)$ and $A_i = (P_i,\ HBE_i, HBI_i, \Lambda_i)$, $\Lambda_i = (EDB_i, RDB_i, SDB_i, IDB_i, 0)$. The **superagent** of $(\mathcal{A}, ENV)$ is the agent $\mathbf{S}_{\mathcal{A}} = (P_{\mathcal{A}}, ENV, \emptyset, \Lambda_{\mathcal{A}})$, where $P_{\mathcal{A}} = P_1 \cup \cdots \cup P_n$ and $\Lambda_{\mathcal{A}} = (EDB, \emptyset, \emptyset, \emptyset, 0)$, $EDB = EDB_1 \cup \cdots \cup EDB_n$.

Note that as $\mathbf{S}_{\mathcal{A}}$ can answer all queries by herself without the need to send requests to other agents, her database of received queries, database of sent–out requests and input database are all empty. Her timestamp is always 0 too. Her state is therefore represented by EDB, a maximal consistent set of literals over ENV.

The answer of the superagent $\mathbf{S}_{\mathcal{A}}$ to a query "$l?$" at a state EDB is "$l!$" (resp. "$\neg l!$") iff $E \cup P_{\mathcal{A}} \vdash l$ (resp. $E \cup P_{\mathcal{A}} \vdash \neg l$) where E is the preferred extension of $\langle P_{\mathcal{A}}, ENV\rangle$ such that $EDB \subseteq E$.

Intuitively, an answer of an agent in a multiagent system is correct if it coincides with the answer of the superagent. Hence stabilization refers to the convergence of agents' answers to the answers of the superagent.

[7] We say that A is ready to request information from other agents for a query form $\Theta \in RDB$ wrt her state $\sigma = (EDB, RDB, SDB, IDB, t)$ if there is an input literal l such that A is ready to request information $l?$ from some agent B for Θ wrt σ.

3.5 Stabilization

Let $\mathcal{R} = \Sigma_0 \rightarrow \cdots \rightarrow \Sigma_h \rightarrow \ldots$ be a run of a multiagent system $(\mathcal{A}, ENV)$.

Definition 9.

– *A query "$l?$" from agent X to agent A_i at point k in $\mathcal{R}$* [8] *has an answer $l!$ or $\neg l!$ at m $(m > k)$ iff there is a reply transition of the form*

$$\Sigma_m \xrightarrow{(A_i,X,l!)} \Sigma_{m+1} \text{ or } \Sigma_m \xrightarrow{(A_i,X,\neg l!)} \Sigma_{m+1}$$

in $\mathcal{R}$ and there exist query forms $\Theta = (X, l?, S, id)$, $\Theta' = (X, l?, S', id)$ such that $RDB_{i,k+1} \setminus RDB_{i,k} = \{\Theta\}$ and $RDB_{i,m} \setminus RDB_{i,m+1} = \{\Theta'\}$ [9].

– *A query "$l?$" from X to A_i at k in $\mathcal{R}$ is said to be* ***correctly answered*** *iff*
  - *it has the answer $l!$ or $\neg l!$ at some $m > k$ and*
  - *the superagent provides the same answer at state*
  $$EDB_m = EDB_{1,m} \cup \cdots \cup EDB_{n,m}.$$

– *$\mathcal{R}$ is* ***convergent*** *if there is a point h such that every query appearing in $\mathcal{R}$ at any point $k \geq h$ is answered correctly.*

Definition 10. *A multiagent system is said to be* **stabilizing** *iff each of its runs is convergent.*

Example 3 shows that stabilization is not guaranteed in general. The following example illustrates that even if the program of each agent is finite, stabilization is not guaranteed.

Example 8. Consider a multiagent system $(\mathcal{A}, ENV)$ where $\mathcal{A} = (A, B)$, $ENV = \{f\}$ and

$$P_A = \{p \leftarrow q\} \quad HBE_A = \emptyset \quad HBI_A = \{q\} \quad \Lambda_A = (\emptyset, \emptyset, \emptyset, \emptyset, 0)$$
$$P_B = \{q \leftarrow p \quad q \leftarrow f\} \quad HBE_B = \{f\} \quad HBI_B = \{p\} \quad \Lambda_B = (\{\neg f\}, \emptyset, \emptyset, \emptyset, 0)$$

Consider the following run where A receives a query "$p?$" from an external agent E and there is no environment change.

$$\Sigma_0 \xrightarrow{(E,A,p?)} \Sigma_1 \xrightarrow{(A,B,q?)} \Sigma_2 \xrightarrow{(B,A,p?)} \Sigma_3 \xrightarrow{nil} \Sigma_4 \xrightarrow{nil} \ldots$$

To answer the query "$p?$" from E, A sends out a query "$q?$" to B. To answer the query "$q?$" from A , B sends out a query "$p?$" to A. According to Postulate 1, to answer the query "$p?$" from B, A should not send another query on q to B. As B does not receive any new query, B will not send or receive anything from A. Similarly A will not send or receive anything from B. Thus there is a deadlock and both A and B would never get information about q and p respectively. So the query "$p?$" will never be answered.

We introduce now sufficient conditions for stabilization.

[8] i.e. there is a query transition $\Sigma_k \xrightarrow{(X,A_i,l?)} \Sigma_{k+1}$ in $\mathcal{R}$.

[9] i.e. Θ is generated when A_i receives "$l?$" from X at k and Θ' is deleted when A_i sends "$l!$" at m. Θ, Θ' have the same identification.

Definition 11. *Let $(\mathcal{A}, ENV)$ be a multiagent system and $P_{\mathcal{A}}$ be its superagent's program. The* ***I/O graph*** *of $(\mathcal{A}, ENV)$ is a graph obtained from the atom dependency graph of $P_{\mathcal{A}}$ by removing all nodes that are not relevant to any input atom of agents. $(\mathcal{A}, ENV)$ is* ***IO-acyclic*** *if there is no infinite path in its I/O graph. $(\mathcal{A}, ENV)$ is* ***bounded*** *if $P_{\mathcal{A}}$ is bounded. $(\mathcal{A}, ENV)$ is* ***IO-finite*** *if its I/O graph is finite.*

Theorem 1. *IO–acyclic and IO–finite multiagent systems are stabilizing.*

Theorem 1 introduces sufficient conditions for stabilization. Unfortunately these conditions are rather strong. Could we weaken them? Are IO-acyclicity and boundedness sufficient to guarantee the stabilization of a multiagent system?

Theorem 2. *IO-acyclicity and boundedness are not sufficient to guarantee the stabilization of a multiagent system.*

Proof. We give a counterexample in Example 9.

Example 9. Consider a multiagent system $(\mathcal{A}, ENV)$ with $\mathcal{A} = (A, B)$, $ENV = \{p, q\}$ and

$$\begin{array}{ll} P_A = \{r(1) \leftarrow p \quad r(n+1) \leftarrow s(n)\} & P_B = \{s(1) \leftarrow q \quad s(n+1) \leftarrow r(n)\} \\ HBE_A = \{p\} & HBE_B = \{q\} \\ HBI_A = \{s(n) \mid n \geq 1\} & HBI_B = \{r(n) \mid n \geq 1\} \\ \Lambda_A = (\{\neg p\}, \emptyset, \emptyset, \emptyset, 0) & \Lambda_B = (\{q\}, \emptyset, \emptyset, \emptyset, 0) \end{array}$$

Obviously, $(\mathcal{A}, ENV)$ is bounded and IO–acyclic. It is easy to see that the semantics of agents' programs are as follows: If p is true (resp. false) then all $r(2n+1)$ and $s(2n+2)$, $n \geq 0$, are true (resp. false). Similarly, if q is true (resp. false) then all $s(2n+1)$ and $r(2n+2)$, $n \geq 0$, are true (resp. false).

Suppose that at the beginning p is false, q is true. Consider the following infinite sequence $\mathcal{S}$ of message exchanges between agents:

1. Steps 0 to 5 are given in Figure 2.
2. For every $n \geq 2$, steps $3n$ to $3n + 5$ in $\mathcal{S}$ follow the patterns in Figure 3.

In Figure 2 A receives two queries on $r(2)$ and sends only one request on $s(1)$ to B at step 1 (following **Postulate 1**). A uses the information in B's reply "$s(1)$!" to answer both queries on $r(2)$. Because q is true, B's answer to A's

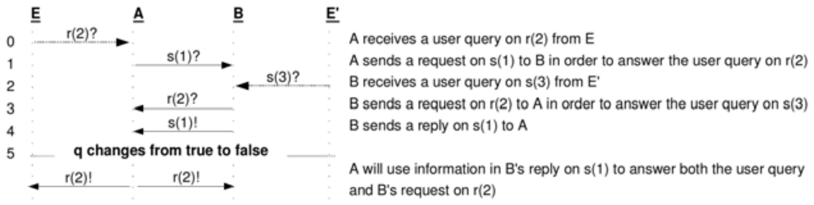
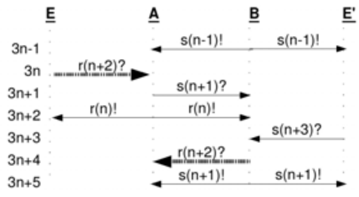
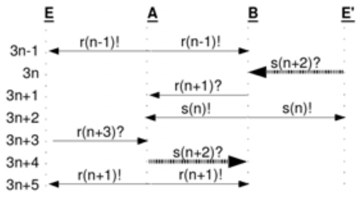

E
A
B
E'
0
1
2
3
4
5
r(2)?
s(1)?
s(3)?
r(2)?
s(1)!
q changes from true to false
r(2)!
r(2)!
A receives a user query on r(2) from E
A sends a request on s(1) to B in order to answer the user query on r(2)
B receives a user query on s(3) from E'
B sends a request on r(2) to A in order to answer the user query on s(3)
B sends a reply on s(1) to A
A will use information in B's reply on s(1) to answer both the user query and B's request on r(2)

Fig. 2. Sequence $\mathcal{S}$: Steps 0-5



(a) Sharing on $s(n+1)$ (Even n)



(b) Sharing on $r(n+1)$ (Odd n)

Fig. 3. Sequence $\mathcal{S}$: Steps $3n$ to $3n+5$

request on $s(1)$ is "$s(1)!$" ($s(1)$ is true) and A's answers to both queries on $r(2)$ are "$r(2)!$" ($r(2)$ is true).

In Figure 3(a), there are two queries on $r(n+2)$ to A (at steps $3n$ and $3n+4$) but only one query on $s(n+1)$ to B (at step $3n+1$). Similarly in Figure 3(b), there are two queries on $s(n+2)$ to B (at steps $3n$ and $3n+4$) but only one query on $r(n+1)$ to A (at step $3n+1$).

To answer all user queries, A needs to request B to provide the value of each $s(n)$, n odd, only once (at step 1 if $n = 1$ and $3n-2$ if $n > 1$) and uses the information in B's reply on $s(n)$ (at step 4 if $n = 1$ and $3n+2$ if $n > 1$) to answer both queries on $r(n+1)$ from E and B (at step $3n+5$). Similarly, B needs to request A to provide the value of each $r(n)$, n even, only once (at step 3 if $n = 2$ and $3n-2$ if $n > 2$) and uses the information in A's reply on $r(n)$ (at step $3n+2$) to answer both the queries on $s(n+1)$ from E' and A (at step $3n+5$). As a result, the answers to queries on $r(2)$, $r(4)$, $\ldots$ and $s(3)$, $s(5)$,$\ldots$ by A and B are all *true*.

As q is false at step 5 and there is no change in the environment after that, the correct answers to queries on $r(2)$, $r(4)$, $\ldots$, and $s(3)$, $s(5)$, $\ldots$ are all $false$.

Because of sharings, the wrong information in B's reply "$s(1)!''$ propagates to A's reply on $r(2)$, the wrong information in A's reply "$r(2)!''$ propagates to B's reply on $s(3)$. This propagation continues upward to replies on $r(4)$, $s(5)$, $\ldots$ and never stops. Consequently, all these replies are incorrect. There is no point in $\mathcal{S}$ where after it the user queries could be answered correctly again. Hence the system is not stabilizing.

4 Related Works and Conclusions

Stabilization of distributed protocols has been studied extensively in the literature ([2],[6],[12]) where agents are defined operationally as automata. Dijkstra ([2]) defined a system as stabilizing if it is guaranteed to reach a legitimate state after a finite number of steps regardless of the initial state. The definition of what constitutes a legitimate state is left to individual algorithms.

There are many research works on multiagent systems where logic programming is used to model agent interaction and/or dialogs/negotiations (e.g. [8],

[11]). But until now research on multiagent systems has not considered the question of stabilization.

Agent communications are either push-based or pull-based. In the push-based communication, agents periodically send information to specific recipients without being requested. Push–based communications are common in internet systems like routing systems. On the other hand, in the pull-based communication, agents have to send requests for information to other agents and wait for replies. Dung et al. ([4]) for the first time studies the stabilization of cooperative information multiagent systems for the push-based communication mode.

In this paper we study the stabilization of multiagent systems based on pull–based communication with information sharing.

Acknowledgments. We thank the referees for constructive comments and criticisms. This work was partially funded by the Sixth Framework IST program of the European Commission under the 035200 ARGUGRID project.

References

1. Denecker, M., Kakas, A.C.: Abduction in logic programming. In: Kakas, A.C., Sadri, F. (eds.) Computational Logic: Logic Programming and Beyond. LNCS, vol. 2407, pp. 402–436. Springer, Heidelberg (2002)
2. Dijkstra, E.W.: Self-stabilizing systems in spite of distributed control. Commun. ACM 17(11), 643–644 (1974)
3. Dung, P.M.: An argumentation-theoretic foundations for logic programming. Journal of Logic Programming 22(2), 151–171 (1995)
4. Dung, P.M., Hanh, D.D., Thang, P.M.: Stabilization of cooperative information agents in unpredictable environment: a logic programming approach. TPLP 6(1-2), 1–22 (2006)
5. Eshghi, K., Kowalski, R.A.: Abduction compared with negation by failure. In: ICLP, pp. 234–254 (1989)
6. Flatebo, M., Datta, A.K.: Self-stabilization in distributed systems. In: Flatebo, M., Datta, A.K. (eds.) Readings in Distributed Computer Systems, ch. 2, pp. 100–114. IEEE Computer Society Press, Los Alamitos (1994)
7. Fung, T., Kowalski, R.A.: The Iff proof procedure for abductive logic programming. Journal of Logic Programming 33(2), 151–165 (1997)
8. Gavanelli, M., Lamma, E., Mello, P., Torroni, P.: An abductive framework for information exchange in multi-agent systems. In: Dix, J., Leite, J. (eds.) CLIMA 2004. LNCS, vol. 3259, pp. 34–52. Springer, Heidelberg (2004)
9. Gelfond, M., Vladimir Lifschitz, V.: The stable model semantics for logic programming. In: ICLP/SLP, pp. 1070–1080 (1988)
10. Kakas, A.C., Robert, A., Kowalski, R.A., Toni, F.: The role of abduction in logic programming. Handbook of Logic in AI and Logic Programming, OUP 5, 235–324 (1998)
11. Satoh, K., Yamamoto, K.: Speculative computation with multi-agent belief revision. In: AAMAS, pp. 897–904 (2002)
12. Schneider, M.: Self-stabilization. ACM Computing Survey 25(1), 45–67 (1993)

Logic Programming for Multiagent Planning with Negotiation

Tran Cao Son[1], Enrico Pontelli[1], and Chiaki Sakama[2]

[1] Dept. of Computer Science, New Mexico State University, Las Cruces, NM 88003, USA
{tson,epontell}@cs.nmsu.edu
[2] Computer and Communication Sciences, Wakayama University, Wakayama 640-8510, Japan
sakama@sys.wakayama-u.ac.jp

Abstract. Multiagent planning deals with the problem of generating plans for multiple agents. It requires formalizing ways for the agents to interact and cooperate, in order to achieve their goals. One way for the agents to interact is through negotiations. Integration of negotiation in multiagent planning has not been extensively investigated and a systematic way for this task has yet to be found. We develop a generic model for negotiation in dynamic environments and apply it to generate joint-plans with negotiation for multiple agents. We identify the minimal requirements for such a model and propose a general scheme for one-to-one negotiations. This model of negotiation is instantiated to deal with dynamic knowledge of planning agents. We demonstrate how logic programming can be employed as a uniform platform to support both planning and negotiation, providing an ideal testbed for experimenting with multiagent planning with negotiations.

1 Introduction

Negotiation and planning are two important tasks that autonomous agents are frequently engaged in during their existence. Theories of *negotiation* have been developed to provide the agents with strategies and methods for doing negotiation (e.g., [1,4,13,15,19,20,17]). On the other hand *planning* research gears towards developing systems and algorithms that allow agents with a way to select appropriate courses of actions to achieve their individual goals (e.g., [11]). It is interesting to observe that there has been limited connection between these research communities. On the other hand, in many practical situations, there is a need for intelligent agents to negotiate when they are planning, especially in multiagent systems. This can be seen in the following simple example.

Example 1 (From [18]). Two home builder agents, A and B, need to hang a mirror (A's job) and a picture (B's job). A can use a screw with a screwdriver to hang the mirror. B can only use a nail and a hammer. Initially, A has a screwdriver and can buy a nail, while B has a screw and a hammer. A and B cannot achieve their goals independently.

Let us now consider the following conversation between A and B: **(1)** $\underline{A \text{ to } B}$: can you give me your screw? **(2)** $\underline{B \text{ to } A}$: yes, but only if you give me a nail; **(3)** $\underline{A \text{ to } B}$: ok, but wait for me to buy some nails. Thereafter, A buys a nail and exchanges it for the screw with B. Both can then achieve their goals independently. □

This story illustrates two important issues that agents are facing in a multiagent environment. Besides the planning capabilities, agents may also need to negotiate in order to achieve their goals—where negotiation becomes a form of cooperation between agents.

In this paper, we are interested in the problem of predicting whether a group of agents, each with her own planning problem to solve, can achieve their independent goals. In the process, the agents might need to negotiate with each others to place themselves in a position where they can achieve their goals.

We start by proposing a *generic model of negotiation*. This model is novel with respect to existing formalisms (e.g., [1,4,13,17,19,20]). Our focus is on agents in dynamic environments, whereas most of the other formalisms concentrate on agents in static environments. One of the main objectives is in predicting/generating successful negotiations within the context of the agents achieving their planning goals. The planning process is driven by the goals of all agents, while negotiations are only a means for agents to achieve their goals. This is different from other models of negotiation (e.g., [1,13,17,19,20,24]), where the focus is on providing the receiver with explanations about negotiations or on the development of languages for logic-based negotiations.

We instantiate the proposed model of negotiation to the context of *multiagent planning* and define two different notions of planning with negotiations. The first notion views complete negotiations as individual steps during planning, while the second one allows the interleaving of steps of negotiations and action executions. Our main goal is to generate a joint-plan for the agents before its execution. In this regards, our work differs from many distributed continual planning systems (e.g., [6]), which concentrate on planning and replanning or deal with unexpected events during plan execution. More significantly, we explore the use of negotiation as a means for agents to cooperate.

A key contribution of this work is to demonstrate how logic programming allows a direct and modular encoding of both negotiation and multiagent planning. To the best of our knowledge, this logic programming based solution is the first attempt to deal with negotiation in multiagent planning. The declarative nature of logic programming allows us to provide a compositional solution to the problem, by combining two orthogonally developed logic programs—one describing the planning problem and one describing the negotiation process.

In the past, multiagent planning has been considered using refinement planning (e.g., [2,12,5]) but without negotiation. In [14], negotiation has been integrated with planning and control operations in the cycle theories, to create an agent architecture and to ensure that the agents can achieve their goals. However, negotiation is used mainly to ensure that the execution of a given plan is successful, e.g., to acquire necessary resources for the execution of a plan—in particular, the authors do not investigate the integration of negotiation within the planning phase. Negotiation using logic programming has been investigated by others (e.g., [4]), with a focus on the principles of negotiation and building new proposals, when the current one is not acceptable. Our characterization is in similar spirit to this approach.

2 Background: Answer Set Planning

In this section, we review the language $\mathcal{A}$ [9] for representing and reasoning about actions and plans in single-agent domains. To simplify the notations in the rest of the paper, we will assume that the discussion in this section is associated to an agent i. We assume the reader to be familiar with the basic concepts of answer set programming.

A *planning problem for* i is defined over a set of *fluents* $\mathtt{F}_i$ and a set of *actions* $\mathtt{A}_i$. We assume that $\mathtt{A}_i$ contains a special action `noop`, which does not have any effect on the agent's world. A *fluent literal* is either a fluent $f \in \mathtt{F}_i$ or its negation $\neg f$. A *domain specification* D_i over $\mathtt{F}_i$ and $\mathtt{A}_i$ describes the actions of an agent and consists of laws of the following forms:[1] (a **causes** ℓ **if** φ) (if $\varphi = true$ then the **if** part will be omitted) and (a **executable** φ), where a is an individual action (in $\mathtt{A}_i$), ℓ is a fluent literal and φ is a set of fluent literals (interpreted as a conjunction). The first law is a *dynamic law*, and states that if a is executed when φ is true then ℓ becomes true. The second law is an *executability condition* and it states that a can be executed only if φ is true.

The semantics of a domain specification D_i is defined by the notion of *state* and by a *transition function* Φ_{D_i}, that specifies the result of the execution of an action a in a state s. A set of literals s satisfies a fluent literal ℓ, denoted by $s \models \ell$, if $\ell \in s$. For a set of fluent literals ϕ, $s \models \phi$ if $s \models \ell$ for every $\ell \in \phi$. A *state* s is a set of fluent literals that is *consistent*—i.e., for each fluent $f \in \mathtt{F}_i$ we have that $\{f, \neg f\} \not\subseteq s$—and *complete*—i.e., for every f either $f \in s$ or $\neg f \in s$. In the following, we use $\overline{\ell}$ to denote the complement literal of ℓ, i.e., if $\ell = f$ for some $f \in \mathtt{F}_i$, then $\overline{\ell} = \neg f$; if $\ell = \neg f$ for some $f \in \mathtt{F}_i$, then $\overline{\ell} = f$. For a set of literals S, $\overline{S} = \{\overline{\ell} \mid \ell \in S\}$.

An action a is *executable* in a state s if there exists an executability condition of the form a **executable** φ in D_i such that $s \models \varphi$. Let

$$e_a(s) = \{\ell \mid \exists(a \textbf{ causes } \ell \textbf{ if } \phi) \in D_i.[s \models \phi]\}$$

The result of the execution of a in s is defined by $\Phi_{D_i}(a, s) = fails$, if a is not executable in s, and $\Phi_{D_i}(a, s) = s \cup e_a(s) \setminus \overline{e_a(s)}$ if a is executable in s. The function Φ_{D_i} can be extended to reason about the effects of a sequence of actions:

Definition 1. *Let D_i be a domain specification, s be a state, and $\alpha = [a_1; \ldots; a_n]$ be a sequence of actions. We define $\hat{\Phi}_{D_i}(\alpha, s) = s$ if $n = 0$, and $\hat{\Phi}_{D_i}(\alpha, s) = \Phi_{D_i}(a_n, \hat{\Phi}_{D_i}([a_1; \ldots; a_{n-1}], s))$, otherwise. Observe that $\Phi_{D_i}(a, fails) = fails$.*

An agent can use the transition function to reason about effects of its actions and to perform planning. A *planning problem* is a tuple $\langle D_i, I_i, O_i \rangle$ where D_i is a domain specification, I_i is a state describing the initial configuration of the world for i, and O_i is a set of literals representing the desired goal.

Definition 2. *Let $\mathcal{P}_i = \langle D_i, I_i, O_i \rangle$ be a planning problem. An action sequence α is a* plan *for $\mathcal{P}_i$ iff O_i is true in $\hat{\Phi}_{D_i}(\alpha, I_i)$.*

Example 2. The domain specification D_A for A in Ex. 1 is defined over $\mathtt{F}_A = \{h\_nail, h\_screw, mirror\_on, h\_hammer, h\_screwdriver\}$ and $\mathtt{A}_A = \{hw\_screw, buy\_nail\}$, with the set of laws:

buy_nail	**causes**	h_nail	hw_screw	**causes**	$mirror\_on$
hw_screw	**causes**	$\neg h\_screw$	hw_screw	**executable**	$h\_screw, h\_screwdriver$

[1] Originally, $\mathcal{A}$ did not include a **executable** φ. It was later introduced by the creator of $\mathcal{A}$.

The domain specification of B is defined over $\mathtt{A}_B = \{hw\_nail\}$ and $\mathtt{F}_B = \{h\_nail, h\_screw, picture\_on, h\_hammer, h\_screwdriver\}$, with the set of laws:

$$\begin{array}{llll} hw\_nail & \textbf{causes} & picture\_on & \qquad hw\_nail\ \textbf{causes}\ \neg h\_nail \\ hw\_nail & \textbf{executable} & h\_nail, h\_hammer & \end{array}$$

In all of the above, the prefix hw stands for "hang with" and h stands for "has." □

Answer set planning (e.g., [16,23]) refers to approaches to planning using logic programming with answer set semantics [8]. In these approaches, a planning problem is translated into a logic program, whose answer sets correspond one-to-one to the solutions of the original problem. As with the action language $\mathcal{A}$, answer set planning approaches have mainly focused on solving single agent planning problems. An exception is [10], dealing with multiagent systems supporting message-based coordination.

Let $\mathcal{P}_i = \langle D_i, I_i, O_i \rangle$ be a planning problem of agent i. We will now describe the logic program $\Pi^n(\mathcal{P}_i)$ that encodes $\mathcal{P}_i$. Let us denote with n the maximal length of a plan. The key predicates of $\Pi^n(\mathcal{P}_i)$ are:

- $h(i, \ell, t)$—the fluent literal ℓ holds at the time step t; and
- $o(i, a, t)$—the action a is executed (by the agent) at the time step t;
- $poss(i, a, t)$—the action a can be executed at the time step t.

$h(i, \ell, t)$ can be extended to define $h(i, \varphi, t)$ for an arbitrary fluent formula φ, which states that φ holds at the time step t. We use $h(i, \{l_1, \ldots, l_k\}, T)$ as a shorthand for $h(i, l_1, T), \ldots, h(i, l_k, T)$. In all the program rules, T denotes a time step, ranging from 0 to n. $\Pi^n(\mathcal{P}_i)$ is defined as follows:

- *Rules for declaring fluents and actions of an agent i:* For each fluent $f \in \mathtt{F}_i$ and action $a \in \mathtt{A}_i$, $\Pi^n(\mathcal{P}_i)$ contains facts of the form $fluent(i, f)$ and $action(i, a)$.
- *Rules for reasoning about effects of actions:* For each action $a \in \mathtt{A}_i$,
  - if D_i contains the law a **executable** φ then $\Pi^n(\mathcal{P}_i)$ contains the rules

$$poss(i, a, T) \leftarrow h(i, \varphi, T). \tag{1}$$

$$\leftarrow o(i, a, T), not\ \ poss(i, a, T). \tag{2}$$

  - if D_i contains the law a **causes** l **if** φ then $\Pi^n(\mathcal{P}_i)$ contains the rule

$$h(i, l, T+1) \leftarrow o(i, a, T), h(i, \varphi, T). \tag{3}$$

- *Rules describing the initial state and the goal state:* For each literal $\ell \in I_i$ and for each $\ell' \in O_i$, $\Pi^n(\mathcal{P}_i)$ contains the rules

$$h(i, \ell, 0) \leftarrow \qquad\qquad \leftarrow not\ \ h(i, \ell', n).$$

- *Rules for encoding inertia:* For each fluent $f \in \mathtt{F}_i$, $\Pi^n(\mathcal{P}_i)$ contains the rules

$$h(i, f, T+1) \leftarrow h(i, f, T), not\ \ h(i, \neg f, T{+}1). \tag{4}$$

$$h(i, \neg f, T+1) \leftarrow h(i, \neg f, T), not\ \ h(i, f, T{+}1). \tag{5}$$

$$\leftarrow h(i, f, T), h(i, \neg f, T). \tag{6}$$

- *Rules for generating action occurrences:* $\Pi^n(\mathcal{P}_i)$ contains the rule

$$1\ \{o(i, A, T) : action(i, A)\}\ 1 \leftarrow T < n. \tag{7}$$

  which states that, at any time step, the agent must execute one of its actions.

The following theorem can be proved.

Theorem 1. *The program $\Pi^n(\mathcal{P}_i)$ is consistent iff $\mathcal{P}_i$ has a plan of length n.*

Let $\mathcal{P}_A$ be the planning problem for A from Example 1. We can easily check that for every n, $\Pi^n(\mathcal{P}_A)$ is inconsistent. Likewise, $\Pi^n(\mathcal{P}_B)$ is inconsistent.

3 Multiagent Planning and Answer Set Planning

In this paper, we are interested in the planning problem in multiagent environments. We focus on situations where each agent has her own planning problem, and the agents are loosely connected—i.e., they might or might not use the same language in their representations (e.g., they might use different names to describe the same property). Furthermore, there are group actions that should be executed together for their effects to take place. Likewise, there are actions that cannot be executed by a group at the same time. Let us start with some preliminary definitions.

Definition 3. *Let $\{\mathcal{P}_i\}_{i\in\mathcal{AG}}$ be a set of planning problems of agents in $\mathcal{AG}$. A* tagged-*fluent is of the form $f[i]$ where f is a fluent in $\mathcal{P}_i$. A* tagged-*formula over $\mathcal{AG}$ is a formula constructible from the set of tagged fluents.*

We will call a sequence of states $S = \langle s_i \rangle_{i\in\mathcal{AG}}$ a *combined state* between agents in $\mathcal{AG}$. Given a tagged-formula φ over $\mathcal{AG}$ and a combined state $S = \langle s_i \rangle_{i\in\mathcal{AG}}$, the truth value of φ in $\langle s_i \rangle_{i\in\mathcal{AG}}$ is determined as follows: if φ is a tagged-fluent $f[i]$ (resp. the negation of a tagged-fluent $\neg f[i]$) then φ is true in S if f is true (resp. false) in s_i; the truth value of a complex formula is computed in the usual way.

Definition 4. *A* multiagent planning problem $\mathcal{M}$ *is a tuple $\langle \mathcal{AG}, \{\mathcal{P}_i\}_{i\in\mathcal{AG}}, \mathcal{F}, \mathcal{NC}, \mathcal{C}\rangle$ where* (i) *$\mathcal{AG}$ is a set of agents,* (ii) *$\mathcal{P}_i$ is a planning problems for agent $i \in \mathcal{AG}$,* (iii) *$\mathcal{F}$ is the set of tagged-formulas over $\mathcal{AG}$, and* (iv) *$\mathcal{NC}$ and $\mathcal{C}$ are sets of sets of pairs (i, a^i) where i is an agent and a^i is an action in $\mathtt{A}_i$.*

Intuitively, $\mathcal{F}$ is a set of constraints on the combined states, $\mathcal{NC}$ is the set of non-concurrent actions, and $\mathcal{C}$ is the set of concurrent actions within $\mathcal{AG}$. For a multiagent planning problem $\mathcal{M}$, a joint-action sequence of length k of agents in $\mathcal{AG}$ is a sequence $\langle \alpha_i \rangle_{i\in\mathcal{AG}}$ where, for each $i \in \mathcal{AG}$, $\alpha_i = [a^i_0, \ldots, a^i_k]$ is a sequence of actions in D_i, executed by i at the time steps $0, 1, \ldots, k$.

Definition 5. *A joint-action sequence $\langle \alpha_i \rangle_{i\in\mathcal{AG}}$ of length k is said to be* compatible *if, for every l, $0 \le l \le k$, the following conditions are satisfied:*

- *For each tagged-formula $\varphi \in \mathcal{F}$, φ is true in $\langle \widehat{\Phi}_{D_i}(\alpha_i[l], I_i)\rangle_{i\in\mathcal{AG}}$ where $\alpha_x[l]$ denotes the action sequence $[a^x_0, \ldots, a^x_l]$.*
- *For each $S \in \mathcal{NC}$, there exists some $(i, a) \in S$ such that $a^i_l \neq a$.*
- *For each $S \in \mathcal{C}$, either $\{a \mid (i, a) \in S \text{ and } a = a^i_l\} = \{a \mid (i, a) \in S\}$ or $\{a \mid (i, a) \in S \text{ and } a = a^i_l\} = \emptyset$.*

Intuitively, a joint-action sequence is compatible if no constraint in $\mathcal{M}$ is violated. The first item indicates that the combined states must not violate the constraints in $\mathcal{F}$, i.e., the individual sequence of actions must agree with each other on their effects in shared environment. The second and third items make sure that non-concurrent and concurrent action constraints in $\mathcal{NC}$ and $\mathcal{C}$ are maintained by the joint-action sequence.

Definition 6. *Let $\mathcal{M} = \langle \mathcal{AG}, \{\mathcal{P}_i\}_{i \in \mathcal{AG}}, \mathcal{F}, \mathcal{NC}, \mathcal{C} \rangle$ be a multiagent planning problem. A joint-action sequence of length n, $\langle \alpha_i \rangle_{i \in \mathcal{AG}}$, is a* joint plan *of length n for $\mathcal{M}$ if $\langle \alpha_i \rangle_{i \in \mathcal{AG}}$ is compatible and for each $i \in \mathcal{AG}$, α_i is a plan of $\mathcal{P}_i$.*

Intuitively, a joint plan is composed of individual plans which allow the agents to achieve their own goals and satisfy the various constraints of the problem.

Example 3. The story in Example 1 can be formalized as a multiagent planning $\mathcal{M}_{AB} = \langle \{A, B\}, \{\mathcal{P}_A, \mathcal{P}_B\}, \emptyset, \emptyset, \emptyset \rangle$ where $\mathcal{P}_A = \langle D_A, I_A, O_A \rangle$ and $\mathcal{P}_B = \langle D_B, I_B, O_B \rangle$ with D_A and D_B are given in Example 2 and

- $I_A = \{\neg h\_nail, \neg mirror\_on, h\_screwdriver, \neg h\_screw, \neg h\_hammer\}$,
- $O_A = \{mirror\_on\}$,
- $I_B = \{\neg h\_nail, \neg picture\_on, h\_screw, \neg h\_screwdriver, h\_hammer\}$, and
- $O_B = \{picture\_on\}$.

We can easily show that $\mathcal{M}_{AB}$ has no solution. □

Answer set planning can be easily extended to compute solutions of multiagent planning problems. This is achieved by defining a program $\Pi^n(\mathcal{M})$, which consists of the rules of $\Pi^n(\mathcal{P}_i)$ along with rules enforcing the constraints in $\mathcal{F}$, $\mathcal{NC}$, and $\mathcal{C}$:

- For each tagged-formula φ in $\mathcal{F}$, a set of rules defining an atom $h(tagged, n_\varphi, T)$, where n_φ is a unique name assigned to φ. Due to lack of space, we omit the set of rules defining this atom (that can be found in [22]). To make sure that the formula is satisfied by the combined state at each time point, we add to $\Pi^n(\mathcal{M})$ the constraint:
$$\leftarrow not \ \ h(tagged, n_\varphi, T).$$
- For each set $\{(i_1, a_1), \ldots, (i_k, a_k)\}$ in $\mathcal{C}$, the constraint
$$\leftarrow 0 \; \{o(i_1, a_1, T), \ldots, o(i_k, a_k, T)\} \; k-1.$$
which makes sure that if a part of S is executed, i.e., $o(i_j, a_j, T)$ belongs to an answer set, then the whole set S is executed.
- For each set $\{(i_1, a_1), \ldots, (i_k, a_k)\}$ in $\mathcal{NC}$, the constraints
$$\leftarrow o(i_1, a_1, T), \ldots, o(i_k, a_k, T).$$
This guarantees that not all actions $a_1, \ldots, a_k$ are executed at the same time.

We can extend Th. 1 and prove that the program $\Pi^n(\mathcal{M})$ is consistent iff $\mathcal{M}$ has a solution of length n.

4 Negotiations between Agents in Dynamic Environments

We consider one-to-one negotiations between agents in a dynamic environment and assume that the negotiation is related to the world representation of each agent. Each agent maintains her own world representation and has her own means to affect the world. Exchanges between agents can be characterized by logical formulae constructible from their representation languages. The acceptance by an agent i of an exchange coming from agent j will affect i's state of the world, and possibly that of j as well.

We assume that each agent i uses her own representation language $\mathcal{L}_i$ and a knowledge base KB_i, whose set of models $\mathcal{W}_i$ represents the acceptable states that she could be in. We assume that there exists a relation $\mathcal{R}_{i,j} \subseteq \mathcal{W}_i \times \mathcal{W}_j$ which encodes the set of *compatible* models between agents i and j—i.e., a pair $(w_i, w_j) \in \mathcal{R}_{i,j}$ is a possible combined state of the agents i and j. We will assume that $\mathcal{R}_{j,i}$ is the inverse relation of $\mathcal{R}_{i,j}$. We will not worry about how $\mathcal{R}_{i,j}$ could be defined or what properties it should have. For example, if the agents in Example 1 have only one hammer, then any pair of possible worlds between them should indicate that exactly one of them has the hammer.

Since negotiation entails an exchange between agents, and the agents can potentially rely on distinct languages, we introduce partial functions $\rho_{i,j}$, called *language matching* functions, that map formulae of $\mathcal{L}_i$ to formulae of $\mathcal{L}_j$. We will assume that these functions are unambiguous w.r.t. equivalence of formulas, i.e., $\rho_{i,j}(\varphi) = \rho_{i,j}(\psi)$ if $\varphi \equiv \psi$. In the case of planning agents, the function $\rho_{A,B}$ could be used to map fluents of A to corresponding fluents of B—e.g., $\rho_{A,B}(f) = g$ states that when A refers to f, it will mean g for B. We require that for each pair of agents i and j there are two functions $\rho_{i,j}$ and $\rho_{j,i}$ that are used by the agents in their communications. As in the case of the compatible relation $\mathcal{R}$, we will not worry about how ρ is defined. Before we continue, let us illustrate these notions using the agents A and B from Ex. 1.

Example 4. A and B can use the languages constructible from $\mathtt{F_A}$ and $\mathtt{F_B}$ as $\mathcal{L}_A$ and $\mathcal{L}_B$ respectively. Thus, we have that $\mathcal{W}_A$ (resp. $\mathcal{W}_B$) is the set of possible states of A (resp. B). Since there are no constraints on the combined states, we have that $\mathcal{R}_{A,B} = \mathcal{W}_A \times \mathcal{W}_B$. The language matching functions between A and B are identities. □

A negotiation originates from an agent i (*originator*), who is trying to have agent j (*recipient*) to establish for her a property (ψ). In turn, i may have to agree to establish φ for j. Such an exchange is captured by the notion of a conditional proposal.

Definition 7. *A* conditional proposal *(or, simply,* proposal*) from i to j has the form $\varphi \overset{i,j}{\Rightarrow} \psi$, where φ and ψ are formulae in $\mathcal{L}_i$ s.t. $\rho_{i,j}(\varphi)$ and $\rho_{i,j}(\psi)$ are both defined.*

A proposal $\varphi \overset{i,j}{\Rightarrow} \psi$ says that i is willing to establish φ for j (i.e., j can consider that $\rho_{i,j}(\varphi)$ is true in her state) and, in exchange, i requires j to establish $\rho_{i,j}(\psi)$ for her. For example, the conditional proposal $h\_nail \overset{A,B}{\Rightarrow} h\_screw$ from A to B in Ex. 1 states that A wants to exchange her $nail$ for B's screw. A can make this offer if she has a nail, and she will have a screw in the resulting state after the proposal is accepted by B.

Agents will negotiate by exchanging proposals. Each agent has her own way of evaluating and assimilating proposals within her knowledge base. We will assume that each agent i is associated with three functions, $RPre_i$, $RPost_i$, and $OPost_i$, which map models and proposals to sets of models. The use of separate originators/receivers functions allows us to formalize various types of negotiations, e.g., asymmetric ones.

$OPost_i$ describes the possible states i will be in if her proposal is accepted. $RPre_i$ and $RPost_i$ represent the conditions for i to consider a received proposal and the consequences of accepting it. These functions satisfy the following conditions:

- $RPre_i(w, \varphi \overset{j,i}{\Rightarrow} \psi) \subseteq \mathcal{W}_i$ and for each $w' \in RPre_i(w, \varphi \overset{j,i}{\Rightarrow} \psi)$, $w' \models \rho_{j,i}(\psi)$.
- $RPost_i(w, \varphi \overset{j,i}{\Rightarrow} \psi) \subseteq \mathcal{W}_i$.
- $OPost_i(w, \varphi \overset{i,j}{\Rightarrow} \psi) \subseteq \mathcal{W}_i$ and for each $w' \in OPost_i(w, \varphi \overset{i,j}{\Rightarrow} \psi)$, $w' \models \psi$.

The condition on $RPre_i$ indicates that if i wants to accept the proposal $\varphi \overset{j,i}{\Rightarrow} \psi$ then she should (somehow) have $\rho_{j,i}(\psi)$ to satisfy the proposal. $OPost_i$ requires that if i made the proposal $\varphi \overset{i,j}{\Rightarrow} \psi$ then she should have ψ if the proposal is accepted by j. Finally, for all functions, it is required that the agent considers only acceptable states.

Example 5. Consider the agents in Example 4. A possible definition for $RPre_A$ w.r.t. the proposal $h\_screw \overset{B,A}{\Rightarrow} h\_nail$ is

$$RPre_A(w, h\_screw \overset{B,A}{\Rightarrow} h\_nail) = \begin{cases} \{w\} & \text{if } w \models h\_nail \\ \emptyset & \text{otherwise} \end{cases}$$

(i.e., to accept the proposal, A should have a nail). A possible definition of $RPost_A$ w.r.t. the proposal $h\_screw \overset{B,A}{\Rightarrow} h\_nail$ is

$$RPost_A(w, h\_screw \overset{B,A}{\Rightarrow} h\_nail) = \begin{cases} \{w' \mid w' = w \setminus \{\neg h\_screw, h\_nail\} \cup \\ \qquad \{h\_screw, \neg h\_nail\}\} \text{ if } w \models h\_nail \\ \emptyset \qquad \text{otherwise} \end{cases}$$

A possible definition of $OPost_B$ w.r.t. the proposal $h\_screw \overset{B,A}{\Rightarrow} h\_nail$ is given next:

$$OPost_B(w, h\_screw \overset{B,A}{\Rightarrow} h\_nail) = \begin{cases} \{w' \mid w' = w \setminus \{h\_screw, \neg h\_nail\} \cup \\ \qquad \{\neg h\_screw, h\_nail\}\} \text{ if } w \models h\_screw \\ \emptyset \qquad \text{otherwise} \end{cases}$$

Let us define when an agent can make a proposal and what she can do if a proposal was made to her. If a proposal $\varphi \overset{i,j}{\Rightarrow} \psi$ is made, j can either accept the proposal, reject it, or continue with the negotiation. First, if j were to accept the proposal, then $RPre_j(w, \varphi \overset{i,j}{\Rightarrow} \psi)$ should not be empty, since this set tells j that she can satisfy the request of i (e.g., she has the formula that is being requested by i). Furthermore, $RPost_j(w, \varphi \overset{i,j}{\Rightarrow} \psi)$ should not be empty as this set indicates that the consequence of accepting the proposal is acceptable to j. Otherwise, j can make a counter proposal or reject the offer. A counter proposal can only be made if j thinks that she can offer $\rho_{i,j}(\psi)$ to i in exchange for φ', i.e., $\rho_{j,i}(\varphi') \overset{i,j}{\Rightarrow} \psi$ should be a possible proposal. If j cannot accept the proposal and cannot make a counter proposal to i, then she will reject the proposal. We formulate these notions in the next definitions—where $\alpha = \varphi \overset{i,j}{\Rightarrow} \psi$ is a proposal from i to j and w_i and w_j are the current states of i and j, respectively.

Definition 8. *α is acceptable to both i and j w.r.t. w_i and w_j if*
- ○ *$w_i \models \varphi$ and $OPost_i(w_i, \alpha) \neq \emptyset$ (α is O-acceptable w.r.t. w_i).*
- ○ *$RPre_j(w_j, \alpha) \neq \emptyset$ and $RPost_j(w_j, \alpha) \neq \emptyset$ (α is R-acceptable w.r.t. w_j).*

Observe that the definitions of $RPre_j$ and $RPost_j$ take care of converting the formulae in the proposal (that are in the language of i) to the local language of j.

Definition 9. *α is R-negotiable w.r.t. w_j if α is not R-acceptable w.r.t. w_j, and there is some φ' (in the language of j) s.t. $RPre_j(w_j, \beta) \neq \emptyset$ and $RPost_j(w_j, \beta) \neq \emptyset$ where $\beta = \rho_{j,i}(\varphi') \overset{i,j}{\Rightarrow} \psi$. α is R-rejectable if it is not R-acceptable and not R-negotiable.*

Definition 10. *α is O-negotiable w.r.t. w_i if α is not O-acceptable w.r.t. w_i, and there exists some φ' such that $w_i \models \varphi'$ and $OPost_i(w_i, \varphi' \overset{i,j}{\Rightarrow} \psi) \neq \emptyset$. α is O-rejectable if it is not O-acceptable and not O-negotiable.*

Example 6. Let us assume that $RPre_B$, $RPost_B$, and $OPost_B$ are defined similarly to $RPre_A$, $RPost_A$, and $OPost_A$ in Example 5. Given two states $w_A = I_A$ and $w_B = I_B$ (Example 3) of A and B, we can easily check the following: **(a)** $h\_nail \overset{A,B}{\Rightarrow} h\_screw$ is O-negotiable in w_A; **(b)** $true \overset{A,B}{\Rightarrow} h\_screw$ is O-acceptable w.r.t. w_A and R-acceptable to B; **(c)** $h\_nail \overset{A,B}{\Rightarrow} h\_screw$ is R-acceptable in w_B. □

We will next define the notion of a negotiation. An *exchange* between i and j is either a formula in $\mathcal{L}_i$ or a *critique*, which is either `accept` or `reject`.

Definition 11. *Let i and j be two agents and w_i and w_j be their current states. A sequence of exchanges $m_0, \ldots, m_n, \ldots$ is a (i, j)*-negotiation for ψ w.r.t. w_i and w_j *if*

- *for every k, $m_{2k} \overset{i,j}{\Rightarrow} \psi$ is O-negotiable w.r.t. w_i and $m_{2k+1} \overset{i,j}{\Rightarrow} \psi$ is R-negotiable w.r.t. w_j;*
- *if m_n is a critique then the sequence is finite and*
  - *if $m_n =$ `accept` then $m_{n-1} \overset{i,j}{\Rightarrow} \psi$ is acceptable w.r.t. w_i and w_j.*
  - *if $m_n =$ `reject` then $m_{n-1} \overset{i,j}{\Rightarrow} \psi$ is O-rejectable w.r.t. w_i if n is even or $m_{n-1} \overset{i,j}{\Rightarrow} \psi$ is R-rejectable w.r.t. w_j if n is odd.*

Since acceptance of a proposal leads two agents to possibly change states, states compatibility becomes an issue:

Definition 12. *Let i and j be two agents and w_i and w_j be their states. A finite (i, j)*-negotiation $(m_0, \ldots, m_n)$ for ψ w.r.t. w_i and w_j is *practical* if $m_n =$ `accept` and there exists a pair $(w'_i, w'_j) \in \mathcal{R}_{i,j}$ such that $w'_i \in OPost_i(w_i, m_{n-1} \overset{i,j}{\Rightarrow} \psi)$ and $w'_j \in RPost_j(w_j, m_{n-1} \overset{i,j}{\Rightarrow} \psi)$. We say that $m_{n-1} \overset{i,j}{\Rightarrow} \psi$ is the *outcome* of the negotiation.

Definition 13. *A (i, j)-negotiation $m_0, \ldots$ for ψ w.r.t. w_i and w_j is non-repeating if, for every pair of $k \neq t$, m_k is not logically equivalent to m_t (i.e., $\not\models m_k \Leftrightarrow m_t$).*

The following theorem is an immediate consequence of the finiteness of the languages of i and j and the definition of non-repeating negotiation.

Theorem 2. *Any non-repeating (i, j)-negotiation for ψ w.r.t. w_i and w_j is finite.*

5 Integration of Negotiation in Multiagent Planning

In this section, we will integrate the proposed method for negotiation in planning in presence of multiple agents. We have seen (e.g., Ex. 3) that a planning problem may not have a solution (e.g., agents A and B cannot achieve their goals). It is easy to see that, if A purchases a nail and exchanges it with B for a screw, then both A and B can achieve their goals. Thus, negotiation can provide the interaction between multiple agents necessary to achieve success. In order for the negotiation to be used during planning, we will need to instantiate our model of negotiation to the case of multiagent planning.

5.1 Negotiation in Multiagent Planning

Let us consider a multiagent planning problem $\mathcal{M} = \langle \mathcal{AG}, \{\mathcal{P}_i\}_{i \in \mathcal{AG}}, \mathcal{F}, \mathcal{NC}, \mathcal{C} \rangle$. In this case, the language $\mathcal{L}_i$ for negotiation used by agent i is the propositional language built using the set of fluents $\mathtt{F}_i$ in D_i. The set $\mathcal{W}_i$ of permissible states of i is the set of all possible states in $\mathcal{P}_i$.[2] In this paper, we are concerned with the case where each successful (i,j)-negotiation with outcome $\varphi \overset{i,j}{\Rightarrow} \psi$, where φ and ψ are conjunctions of literals, will result in (i) agent i having ψ and $\overline{\varphi}$ in the next state; and (ii) agent j having φ and $\overline{\psi}$ in the next state (recall that $\overline{\psi}$ denotes the $\{\overline{\ell} \mid \ell \in \psi\}$). In order to accept a proposal $\varphi \overset{i,j}{\Rightarrow} \psi$, an agent j should have ψ. This means that the functions $RPre_j$, $RPost_j$, and $OPost_i$ are defined as follows.

$$RPre_j(w, \varphi \overset{i,j}{\Rightarrow} \psi) = \begin{cases} \{w\} & \text{if } w \models \rho_{i,j}(\psi) \\ \\ \emptyset & \text{otherwise} \end{cases}$$

$$RPost_j(w, \varphi \overset{i,j}{\Rightarrow} \psi) = \begin{cases} \{w \cup e \setminus \bar{e}\} & \text{if } w \models \rho_{i,j}(\psi) \\ \\ \emptyset & \text{otherwise} \end{cases}$$

where $e = \rho_{i,j}(\varphi) \cup \overline{\rho_{i,j}(\psi)}$. Furthermore,

$$OPost_i(w, \varphi \overset{i,j}{\Rightarrow} \psi) = \begin{cases} \{w \cup e' \setminus \overline{e'}\} & \text{if } w \models \varphi \\ \emptyset & \text{otherwise} \end{cases}$$

where $e' = \psi \cup \overline{\varphi}$. The model compatibility relation $\mathcal{R}_{i,j}$ consists of (s, s') if there exists a combined state $\langle w_t \rangle_{t \in \mathcal{AG}}$ such that $s = w_i$, $s' = w_j$, and the formulas in $\mathcal{F}$ are satisfied by $\langle w_t \rangle_{t \in \mathcal{AG}}$. As there is no explicit requirement on the languages used in formalizing $\mathcal{M}$, we will keep assuming the existence of a language matching function ρ. In our examples, ρ will simply correspond to the identity function.

5.2 Planning with Non-interleaved Negotiations

The first approach we consider is the case where agents participating in a negotiation are prevented from performing any other activities until the negotiation is complete.

Let us assume that each finite length negotiation between any two distinct agents in $\mathcal{AG}$ is assigned a unique name, and let us denote with $N_{i,j}$ the set of the names of all finite (i,j)-negotiations. A *joint-action sequence with negotiation* of length k for the agents in $\mathcal{AG}$ is a sequence $\langle \alpha_i \rangle_{i \in \mathcal{AG}}$ where, for each $i \in \mathcal{AG}$, $\alpha_i = [a_0^i, \ldots, a_k^i]$ and, for each $0 \leq l \leq k$, a_l^i is either an action in D_i or an element of $N_{i,j} \cup N_{j,i}$. A joint-action sequence $\langle \alpha_i \rangle_{i \in \mathcal{AG}}$ of length k is said to be *compatible* if: *(i)* It is compatible w.r.t. Definition 5, and *(ii)* If $a_l^i \in N_{i,j} \cup N_{j,i}$ then $a_l^j = a_l^i$. This is illustrated next.

Example 7. Consider the multiagent planning problem $\mathcal{M}_{AB}$ in Ex. 3, where $\mathcal{M}_{AB}$ has no solution. It is easy to see that the following *joint-action sequence* can achieve the goals of both A and B: *(1)* A buys a nail. *(2)* A proposes to B to exchange a screw for a nail. *(3)* B accepts the proposal of A and the exchange is made. *(4)* A hangs the mirror

[2] This set could exclude some interpretations, e.g., because of the $\mathcal{F}$ constraints of Def. 4.

with her screwdriver and the screw. *(5)* B hangs the picture with the nail and the hammer. This can be represented as $\langle \alpha_i \rangle_{i \in \{A,B\}}$, where $\alpha_A = [buy\_nail, N_1, hw\_screw]$ and $\alpha_B = [\mathtt{noop}, N_1, hw\_nail]$, where $N_1 = h\_nail \overset{A,B}{\Rightarrow} h\_screw$, $\mathtt{accept}$. □

Let us extend the definition of the transition function Φ to encompass negotiations.

Definition 14. *Let $N \in N_{i,j}$ be an (i,j)-negotiation and s_i and s_j be the states of i and j, respectively. We define $\Phi(N, s_i)$ and $\Phi(N, s_j)$ as follows.*

- *If N ends with* `reject`*, then $\Phi(N, s_i) = s_i$ and $\Phi(N, s_j) = s_j$.*
- *If N ends with* `accept` *and the outcome is $\varphi \overset{i,j}{\Rightarrow} \psi$, then* $\Phi(N, s_i) = OPost_i(s_i, \varphi \overset{i,j}{\Rightarrow} \psi)$ *and* $\Phi(N, s_j) = RPost_j(s_j, \varphi \overset{i,j}{\Rightarrow} \psi)$. [3]

The function $\widehat{\Phi}$ can be extended to the case of sequences of actions with negotiations. Definition 6 can then be used to define the notion of a joint-plan with negotiation for multi-agent planning problems. For example, it is easy to see that $\langle \alpha_i \rangle_{i \in \{A,B\}}$ (Example 7) is a joint-plan with negotiation for A and B in the problem $\mathcal{M}_{AB}$.

5.3 Planning with Interleaved Negotiations

A joint-plan with negotiation as defined in the previous subsection does not consider the case where agents may align themselves to make a proposal acceptable. For example, if B makes the proposal $h\_screw \overset{B,A}{\Rightarrow} h\_nail$ to A in the initial state, A—by virtue of having no nail—will reject it. On the other hand, A can accept the proposal after it purchases a nail, i.e., the following could be considered a joint-plan for A and B:

$$\alpha_A = [\mathtt{noop}, buy\_nail, \mathtt{accept}, hw\_screw]$$
$$\alpha_B = [h\_screw \overset{B,A}{\Rightarrow} h\_nail, \mathtt{noop}, \mathtt{noop}, hw\_nail]$$

In the above joint-plan, individual exchanges of a negotiation act like individual actions. To accommodate this, we introduce *negotiation actions* of the following forms:

a. $starts(i, j, \varphi, \psi)$: i starts a negotiation with j, by making the proposal $\varphi \overset{i,j}{\Rightarrow} \psi$;
b. $proposes(i, j, \varphi, \psi)$: φ is a non-critique exchange in an (i, j)-negotiation for ψ;
c. $accepts(i, j, \varphi, \psi)$: i and j accept the proposal $\varphi \overset{i,j}{\Rightarrow} \psi$;
d. $rejects(i, j)$: i and j reject the last exchange made and terminate the negotiation.

These actions are referred to as (i, j)-*negotiation actions*.

The notion of a compatible joint-action sequence has to be modified to account for negotiation actions. Different views may lead to different definitions of a joint-action with negotiation actions. In the following, we will require the following:

- at any time, one agent is engaged in at most one negotiation; and
- agents must finish one negotiation before they can start a new one.

We extend the definition of transition function to account for the negotiation actions:

$$\Phi_{D_i}(starts(i, j, \varphi, \psi), w) = w$$
$$\Phi_{D_j}(starts(i, j, \varphi, \psi), w) = \mathit{fails}$$
$$\Phi_{D_x}(proposes(i, j, \varphi, \psi), w) = w$$
$$\Phi_{D_x}(rejects(i, j), w) = w$$

[3] Note that we slightly abuse the notation since $OPost$ and $RPost$ return singleton sets.

$$\mathit{\Phi}_{D_i}(accepts(i,j,\varphi,\psi),w) = OPost_i(w, \varphi \overset{i,j}{\Rightarrow} \psi)$$
$$\mathit{\Phi}_{D_j}(accepts(i,j,\varphi,\psi),w) = RPost_j(w, \varphi \overset{i,j}{\Rightarrow} \psi)$$

where $x \in \{i,j\}$. Let $\alpha_i = [a^i_1, \ldots, a^i_k]$ and $\alpha_j = [a^j_1, \ldots, a^j_k]$ be two action sequences, containing ordinary actions from D_i and D_j and/or negotiation actions. Let $\alpha_i \oplus \alpha_j = [C_1, \ldots, C_k]$ where C_l is the set of negotiation actions among $\{a^i_l, a^j_l\}$. Let $Ag(C_l) = \{x \mid a^x_l \in C_l\}$. We say that (α_i, α_j) is *syntactically correct* if

- C_l is either an empty set or a singleton, i.e., $|C_l| \leq 1$;
- for each $1 \leq l < l' \leq k$ s.t. $C_l \neq \emptyset$ and $C_{l'} \neq \emptyset$, if $C_{l+1} = \cdots = C_{l'-1} = \emptyset$ then $Ag(C_l) \cup Ag(C_{l'}) = \{i,j\}$.

The pair (α_i, α_j) is (i,j)-*syntactically correct* if it is syntactically correct and

- either no (i,j)-negotiation action occurs in $\alpha_i \oplus \alpha_j$, or
- for each (i,j)-negotiation action a^i_x or a^j_x in $\alpha_i \oplus \alpha_j$ there exists $l \neq l'$ such that **(a)** $l \leq x \leq l'$, **(b)** $C_l = \{a^i_l\} = \{starts(i,j,\varphi_l,\psi)\}$, **(c)** for every $l < t < l'$, $C_t = \emptyset$ or $C_t = \{proposes(i,j,\varphi_t,\psi)\}$, and **(d)** either $a^i_{l'} = a^j_{l'} = accepts(i,j,\varphi_{l'},\psi)$ or $a^i_{l'} = a^j_{l'} = rejects(i,j)$.

We say a joint-action sequence $\langle \alpha_i \rangle_{i \in \mathcal{AG}}$ of length k, where each a^i_l is an action in $\mathcal{P}_i$ or one of the negotiation actions, is *compatible* if:

- it satisfies the conditions in Definition 5; and
- for every $i \neq j$, (α_i, α_j) is (i,j)-syntactically correct and, for every C_t in $\alpha_i \oplus \alpha_j$ containing an (i,j)-negotiation action a we have that:
  - if $a = starts(i,j,\varphi,\psi)$ then $\varphi \overset{i,j}{\Rightarrow} \psi$ is O-negotiable w.r.t. $\hat{\mathit{\Phi}}_{D_i}(\alpha_i[t-1], I_i)$;
  - if $a = proposes(i,j,\varphi,\psi)$ and $Ag(C_t) = i$ then $\varphi \overset{i,j}{\Rightarrow} \psi$ is O-negotiable w.r.t. $\hat{\mathit{\Phi}}_{D_i}(\alpha_i[t-1], I_i)$;
  - if $a = proposes(i,j,\varphi,\psi)$ and $Ag(C_t) = j$ then $\varphi \overset{i,j}{\Rightarrow} \psi$ is R-negotiable w.r.t. $\hat{\mathit{\Phi}}_{D_j}(\alpha_j[t-1], I_j)$;
  - if $a = accepts(i,j,\varphi,\psi)$ then $\varphi \overset{i,j}{\Rightarrow} \psi$ is acceptable w.r.t. $\hat{\mathit{\Phi}}_{D_i}(\alpha_i[t-1], I_i)$ and $\hat{\mathit{\Phi}}_{D_j}(\alpha_j[t-1], I_j)$;
  - if $a = rejects(i,j)$ and $proposes(i,j,\varphi,\psi)$ or $starts(i,j,\varphi,\psi)$ is the last occurrence of an (i,j)-negotiation action in $\alpha_i \oplus \alpha_j$ before t, then $\varphi \overset{i,j}{\Rightarrow} \psi$ is O-rejectable w.r.t. $\hat{\mathit{\Phi}}_{D_i}(\alpha_i[t-1], I_i)$ or R-rejectable w.r.t. $\hat{\mathit{\Phi}}_{D_j}(\alpha_j[t-1], I_j)$.

Joint-plans with negotiation are defined accordingly.

5.4 Computing Plan with Negotiation Using Logic Programming

In the rest of this section, we will present a set of rules that, when added to $\mathit{\Pi}^n(\mathcal{M})$, will generate joint-plans with negotiation. We refer to the new program as $\mathit{\Gamma}^n(\mathcal{M})$. Due to lack of space, we omit some of the more technical details and we make use of a rather informal logic programming syntax. In the following rules, i, j, and k denote possible agents. As in the previous discussion, we will consider the case where formulae involved in negotiations are composed only of sets of literals. We assume the existence of negotiation formula atoms in the program `formula_name`(.), naming the possible set of literals. The composition of the actual formula φ can be described as a collection of facts `in_formula`(φ, ℓ) for each literal $\ell \in \varphi$ (with a slight abuse of notation, we will use φ as the name of the formula itself). We also assume that the language matching functions are identities. For each agent i, we introduce

$$NA_i = \left\{ \begin{array}{l} starts(i,j,\varphi,\psi), proposes(i,j,\varphi,\psi), \\ accepts(i,j,\varphi,\psi), rejects(i,j) \end{array} \middle| \; j \in \mathcal{AG}, \varphi, \psi \textit{ are fomulas names} \right\}.$$

A predicate $\texttt{na}(i,a)$ is used to identify the elements $a \in NA_i$. Since the construction of an exchange requires hypothetical reasoning, we assume a predicate $hyp\_h(i,\varphi,\psi,\ell,T)$ that is true if ℓ is true in $OPost_i(w, \varphi \overset{i,j}{\Rightarrow} \psi)$, where w is the state for i described by $h(i,\cdot,T)$. The definition of hyp_h is straightforward. This allows us to describe states that are not acceptable; in our case:

$$\texttt{bad}(i,\varphi,\psi,T) \leftarrow fluent(i,f), hyp\_h(i,\varphi,\psi,f), hyp\_h(i,\varphi,\psi,neg(f))$$

The rules used to describe the effects of negotiation can be summarized as follows.

• *Generation rules:* The rule for generating action occurrences is expanded to:

$$1\{o(i,a,T) : \texttt{na}(i,a), o(i,A,T) : action(i,A)\}1 \leftarrow agent(i), time(T), T < n$$

• *Negotiation rules:* These rules control the ability to perform steps of negotiation; the predicate `wait` is used to indicate that it is not the agent's turn to respond to a negotiation, allowing to enforce the exchange protocol described earlier. Each non-critique exchange will prompt `wait` to become true; each generated exchange will also invalidate the `wait` of the other party. In the following, $x \in \{i,j\}$ and $\bar{i} = j$ and $\bar{j} = i$. The variable T denotes the time parameter.

$$\begin{array}{ll}
\%\% \;\; \textit{i starts} & \\
h(i, wait(i,j,\varphi,\psi), T+1) & \leftarrow o(i, starts(i,j,\varphi,\psi), T). \\
h(j, neg(wait(i,j,\varphi,\psi)), T+1) & \leftarrow o(i, starts(i,j,\varphi,\psi), T). \\
\%\% \;\; \textit{x exchanges} & \\
h(x, wait(i,j,\varphi,\psi), T+1) & \leftarrow o(x, proposes(i,j,\varphi,\psi), T). \\
h(\bar{x}, neg(wait(i,j,\varphi,\psi)), T+1) & \leftarrow o(x, proposes(i,j,\varphi,\psi), T). \\
\%\% \;\; \textit{Suspend waiting on termination} & \\
h(x, neg(wait(i,j,\varphi,\psi)), T+1) & \leftarrow o(x, accepts(i,j,\varphi,\psi), T). \\
h(x, neg(wait(i,j,\varphi,\psi)), T+1) & \leftarrow o(x, rejects(i,j), T).
\end{array}$$

A successful completion of a negotiation will be achieved when an acceptable exchange is reached. The `bad` predicate will be used to validate acceptability:

$$\begin{array}{l}
\texttt{acceptable}(i,j,T) \leftarrow h(i, wait(i,j,\varphi,\psi),T), h(i,\varphi,T), not \;\; bad(i,\varphi,\psi,T). \\
\texttt{acceptable}(i,j,T) \leftarrow h(j, wait(i,j,\varphi,\psi),T), h(j,\psi,T), not \;\; bad(j,\psi,\varphi,T).
\end{array}$$

If the negotiation is not acceptable, then an attempt to generate a new exchange is made. We use a *repeated* predicate to avoid repeated exchanges (the code is simple but tedious, and omitted).

$$\begin{array}{ll}
\texttt{valid\_proposal}(i,j,\varphi',T) & \leftarrow h(i, wait(i,j,\varphi,\psi),T), formula\_name(\varphi'), \\
 & \quad not \;\; repeated(i,j,\varphi',T), h(i,\varphi',T), not \;\; bad(i,\varphi',\psi,T). \\
\texttt{valid\_proposal}(i,j,\varphi',T) & \leftarrow h(j, wait(i,j,\varphi,\psi),T), formula\_name(\varphi'), \\
 & \quad not \;\; repeated(i,j,\varphi',T), h(j,\psi,T), not \;\; bad(j,\psi,\varphi',T). \\
\texttt{negotiable}(i,j,T) & \leftarrow h(i, wait(i,j,\varphi,\psi),T), not \; \texttt{acceptable}(i,j,T), \\
 & \quad \texttt{valid\_proposal}(i,j,\varphi',T). \\
\texttt{negotiable}(i,j,T) & \leftarrow h(j, wait(i,j,\varphi,\psi),T), not \; \texttt{acceptable}(i,j,T), \\
 & \quad \texttt{valid\_proposal}(i,j,\varphi',T).
\end{array}$$

Using the above characterizations of what is acceptable and negotiable, we can introduce constraints that will avoid generation of unsuitable negotiation actions:

$$\begin{array}{l}
\%\% \;\; \textit{Protocol: actions must be done in exchange way} \\
\quad \leftarrow o(x, proposes(i,j,\varphi',\psi'),T), h(x, neg(wait(i,j,\varphi,\psi)),T). \\
\quad \leftarrow o(x, accepts(i,j,\varphi',\psi'),T), h(x, neg(wait(i,j,\varphi,\psi)),T). \\
\quad \leftarrow o(x, rejects(i,j)), h(x, neg(wait(i,j,\varphi,\psi)),T). \\
\%\% \;\; \textit{Ensure only valid actions are performed} \\
\quad \leftarrow o(i, starts(i,j,\varphi,\psi),T), not\;\; h(i,\varphi,T). \\
\quad \leftarrow o(i, proposes(i,j,\varphi,\psi),T), not\;\; \texttt{valid\_proposal}(i,j,\varphi,T). \\
\quad \leftarrow o(j, proposes(i,j,\varphi,\psi),T), not\;\; \texttt{valid\_proposal}(i,j,\varphi,T). \\
\quad \leftarrow o(i, accepts(i,j,\varphi,\psi),T), not\;\; \texttt{acceptable}(i,j,T). \\
\quad \leftarrow o(j, accepts(i,j,\varphi,\psi),T), not\;\; \texttt{acceptable}(i,j,T). \\
\quad \leftarrow o(i, rejects(i,j),T), not\;\; negotiable(i,j,T), not\;\; acceptable(i,j,T).
\end{array}$$

Finally, we introduce rules to describe the state changes produced by a negotiation:

$$\begin{array}{l}
h(i,\ell,T+1) \leftarrow o(i, accepts(i,j,\varphi,\psi),T), \texttt{in\_formula}(\psi,\ell). \\
h(i,\overline{\ell},T+1) \leftarrow o(i, accepts(i,j,\varphi,\psi),T), \texttt{in\_formula}(\varphi,\ell). \\
h(j,\ell,T+1) \leftarrow o(j, accepts(i,j,\varphi,\psi),T), \texttt{in\_formula}(\varphi,\ell). \\
h(j,\overline{\ell},T+1) \leftarrow o(j, accepts(i,j,\varphi,\psi),T), \texttt{in\_formula}(\psi,\ell).
\end{array}$$

Theorem 3. *For a multiagent planning problem $\mathcal{M}$ and an integer n,*

- *if I is an answer set of $\Gamma^n(\mathcal{M})$ and $\alpha_i = [a_0^i, \ldots, a_{n-1}^i]$ such that $o(i, a_t^i, t) \in I$, then $\langle \alpha_i \rangle_{i \in \mathcal{AG}}$ is a joint-plan.*
- *if $\langle \alpha_i \rangle_{i \in \mathcal{AG}}$ where $\alpha_i = [a_0^i, \ldots, a_{n-1}^i]$ is a joint-plan of length n, then there is an answer set I of $\Gamma^n(\mathcal{M})$ such that $o(i, a_t^i, t) \in I$ for $i \in \mathcal{AG}$ and $0 \le t \le n-1$.*

Observe that the program $\Gamma^n(\mathcal{M})$ can be easily modified to generate plans with non-interleaved negotiations by adding constraints forbidding an occurrence of an ordinary action for an agent i when the fluent $wait(i,j,.,.)$ or $wait(j,i,.,.)$ is true.

6 Discussion and Conclusions

In this paper, we presented a preliminary investigation of the use of logic programming technology to address the composite problem of multiagent planning and negotiation. We developed a generic model of negotiation, suitable for dynamic environments, and instantiate it to the case of multiple planning agents with independent goals. We defined different notions of planning with negotiations. We illustrated how logic programming provides an elegant and modular encoding of the different aspects of the problem. Observe that the use of logic programming allows for a simple integration between planning and negotiation. The generation of an answer set satisfying the goal of the planning problem drives the generation of any negotiation that needs to occur between agents.

This preliminary work offers several directions for future research. Part of our discussion relies on the use of identity function as a language matching functions. This restriction was imposed for the sake of simplicity and it should be lifted to allow more complex scenarios. For example, in planning with resources, two agents—one using British pounds and one using Dollars—would need matching functions that convert between the two currencies. Other examples include agents using ontologies with different granularities; e.g., agent i uses $has\_nail\_brand\_x(foo)$ to describe nail named foo, while agent j uses separate predicates to describe individual properties of foo (e.g., $is\_nail(foo)$ and $is\_brand\_x(foo)$). Thus, $\rho_{i,j}$ applied to $has\_nail\_brand\_x(foo)$ should result in the formula $is\_nail(foo) \wedge is\_brand\_x(foo) \wedge has(foo)$. The language matching functions are domain-dependent but still expressible in logic programming.

The proposed model of negotiation is grounded and used in the most simple way. It is suitable for negotiations involving exchanges of consumable resources, such as the nail or the screw. Non-consumable resources may require different definitions of the functions $OPost$, $RPre$, and $RPost$. In practice, there could be situations in which the owner of a resource does not lose it after an exchange has happened. For example, a student, agreeing to give another student a ride to school in exchange for the solution of a homework, does not lose her car after the exchange.

There are also situations in which an agent may need to take into consideration what other agents offer before deciding to accept or reject an offer. For example, a student with a car without gasoline could agree to drive some friends to school if the friends give her enough money to buy gasoline. In this case, the student has to take into consideration what was offered before accepting the offer.

The implementation in planning assumes that agents negotiate on sets of literals. This is not a limitation of the general planning framework, but the consequence of the language restrictions of several answer set solvers. This restriction could be lifted by adopting a more general logic programming framework e.g., [3,7].

Further generalizations include the use of negotiation models involving groups of agents, preferences, more expressive action languages (e.g., with static causal laws, concurrent actions), and more complex planning scenarios (e.g., joint-goals).

Acknowledgement

The first two authors were partially supported by NSF grants IIS-0812267, MII-0220590, and CREST-0420407.

References

1. Amgoud, L., Dimopoulos, Y., Moraitis, P.: A unified and general framework for argumentation-based negotiation. In: AAMAS, pp. 1018–1025. ACM Press, New York (2006)
2. Boutilier, C., Brafman, R.I.: Partial-order planning with concurrent interacting actions. J. Artif. Intell. Res. 14, 105–136 (2001)
3. Castro, L., et al.: XASP: Answer Set Programming with XSB and Smodels (2007)
4. Chen, W., Zhang, M., Foo, N.: Repeated negotiation of logic programs. In: Workshop on Nonmonotonic Reasoning, Action and Change (2006)
5. Cox, J.S., Durfee, E.H.: An efficient algorithm for multiagent plan coordination. In: AAMAS, pp. 828–835. ACM, New York (2005)
6. desJardins, M., Durfee, E.H., Ortiz, C.L., Wolverton, M.: A survey of research in distributed, continual planning. AI Magazine 20(4), 13–22 (1999)
7. El-Khatib, O., Pontelli, E., Son, T.: ASP-Prolog: A System for Reasoning about Answer Set Programs in Prolog. In: Jayaraman, B. (ed.) PADL 2004. LNCS, vol. 3057, pp. 148–162. Springer, Heidelberg (2004)

8. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: Fujisaki, T., Furukawa, K., Tanaka, H. (eds.) Logic Programming 1988. LNCS, vol. 383, pp. 1070–1080. Springer, Heidelberg (1989)
9. Gelfond, M., Lifschitz, V.: Representing actions and change by logic programs. JLP 17(2,3,4), 301–323 (1993)
10. Gelfond, G., Watson, R.: Modeling Cooperative Multi-Agent Systems. In: ASP 2007 (2007)
11. Ghallab, M., Nau, D., Traverso, P.: Automated Planning. Morgan Kaufmann, CA (2004)
12. Cox, J.S., Durfee, E.H., Bartold, T.: A Distributed Framework for Solving the Multiagent Plan Coordination Problem. In: AAMAS, pp. 821–827. ACM Press, New York (2005)
13. Kakas, A., Moraitis, P.: Adaptive agent negotiation via argumentation. In: AAMAS, pp. 384–391. ACM Press, New York (2006)
14. Kakas, A., Torroni, P., Demetriou, N.: Agent planning, negotiation and control of operation. In: ECAI, pp. 28–32. IOS Press, Amsterdam (2004)
15. Kraus, S.: Negotiation and cooperation in multi-agent environments. AI J. 94(1-2) (1997)
16. Lifschitz, V.: Answer set programming and plan generation. Artificial Intelligence 138(1–2), 39–54 (2002)
17. Rahwan, I., et al.: Argumentation-based negotiation. Knowledge Engineering Review 18, 343–375 (2003)
18. Parsons, S., et al.: Agents that reason and negotiate by arguing. JLC 8(3), 261–292 (1998)
19. Sadri, F., Toni, F., Torroni, P.: An abductive logic programming architecture for negotiating agents. In: Flesca, S., Greco, S., Leone, N., Ianni, G. (eds.) JELIA 2002. LNCS (LNAI), vol. 2424, pp. 419–431. Springer, Heidelberg (2002)
20. Sakama, C., Inoue, K.: Negotiation by abduction and relaxation. In: AAMAS, pp. 1018–1025. ACM Press, New York (2007)
21. Simons, P., Niemelä, N., Soininen, T.: Extending and Implementing the Stable Model Semantics. Artificial Intelligence 138(1–2), 181–234 (2002)
22. Son, T.C., Baral, C., Tran, N., McIlraith, S.: Domain-dependent knowledge in answer set planning. ACM Trans. Comput. Logic 7(4), 613–657 (2006)
23. Subrahmanian, V., Zaniolo, C.: Relating stable models and AI planning domains. In: Proceedings of the International Conference on Logic Programming, pp. 233–247 (1995)
24. Wooldridge, M., Parsons, S.: Languages for negotiation. In: Proceedings of ECAI (2000)

Answer Set Programming with Constraints Using Lazy Grounding

Alessandro Dal Palù[1], Agostino Dovier[2], Enrico Pontelli[3], and Gianfranco Rossi[1]

[1] Dip. Matematica, Univ. Parma
{alessandro.dalpalu,gianfranco.rossi}@unipr.it
[2] Dip. Matematica e Informatica, Univ. Udine
dovier@dimi.uniud.it
[3] Dept. Computer Science, New Mexico State Univ.
epontell@cs.nmsu.edu

Abstract. The paper describes a novel methodology to compute stable models in Answer Set Programming. The proposed approach relies on a bottom-up computation that does not require a preliminary grounding phase. The implementation of the framework can be completely realized within the framework of Constraint Logic Programming over finite domains. The use of a high level language for the implementation and the clean structure of the computation offer an ideal framework for the implementation of extensions of Answer Set Programming. In this work, we demonstrate how non-ground arithmetic constraints can be easily introduced in the computation model. The paper provides preliminary experimental results which confirm the potential for this approach.

1 Introduction

The recent literature has shown a booming interest towards the *Answer Set Programming (ASP)* paradigm [2]. ASP builds on the theoretical foundations of normal logic programs under stable model semantics, and it provides a programming paradigm that elegantly integrates traditional logic programming, non-monotonic reasoning, and some forms of constraint-based reasoning.

The popularity of ASP has been fueled by the realization that ASP offers compact, elegant, and provably correct solutions for problems in a variety of application domains (e.g., phylogenetic inference [4], planning [13], bioinformatics [9]); significant effort has also been invested in the design of knowledge building blocks and methodologies (e.g., [2]). The development of novel applications has also stretched to the limits both the traditional *languages* supported by ASP as well as system implementations, emphasizing some of the limitations of the currently used technology. This has been, for example, highlighted in a recent study concerning the use of ASP to solve complex planning problems (drawn from recent international planning competitions) [23]. A problem like Pipeline (from IPC-5), whose first 9 instances can be effectively solved by state-of-the-art planners like FF, can be solved only in its first instance using Lparse and

Smodels; instances 2 through 4 do not terminate within several hours of execution, while instance 5 leads Lparse to generate a ground image that is beyond the input capabilities of Smodels.

We have witnessed a flourishing of new proposals for language extensions (e.g., aggregates, domain-specific constraints, functions), to enable the declarative encoding of complex relationships. In turn, also these extensions have proved challenging for the implementors, often leading to unnecessarily complex machineries to integrate extensions within the rigid framework of existing ASP solvers (e.g., [7,19]).

The majority of the existing ASP systems rely on a two-stage computation model. The actual computation of the answer set is performed only on propositional programs—either directly (as in Smodels [21], DLV [12] and Clasp [8]) or appealing to the use of a SAT solver (as in ASSAT [14] and Cmodels [1]). On the other hand, the convenience of ASP vitally builds on the use of *first-order* constructs. This introduces the need of a grounding phase, typically performed by a grounding module (e.g., a separate program, like Lparse or Gringo, or an integrated module as in DLV). The presence of grounding represents a significant obstacle to applications and extensions—it has the potential (often observed in practice) of leading to extremely large ground programs and it may force developers to unnatural solutions to circumvent the grounding of certain components of the program (e.g., as observed in some implementations of aggregates [7] and domain-specific constraints [19]).

In this manuscript, we propose a different perspective on this problem, aimed at creating a framework which executes ASP programs *without* preliminary grounding and which enables ease integration of extensions like domain-specific constraints. The proposed framework is called *Grounding-lazy ASP (*GASP*)*. The spirit of our effort can be summarized as follows:

- The framework is completely developed in a declarative language (Constraint Logic Programming over finite domains)—where finite domain sets are employed for the compact representation of predicates in an ASP program.
- The execution model is bottom-up and does not require preliminary grounding of the program.

This combination of ideas provides a novel system with significant potentials:

- It enables the simple integration of new features in the solver, such as domain-specific constraints (e.g., numerical constraints). With a preliminary grounding stage, these features would have to be encoded as ground programs, thus reducing the capability to devise general strategies to optimize the search, and often leading to exponential growth in the size of the ground program.
- The adoption of a non-ground search allows the system to effectively control the search process at a higher level, enabling the adoption of Prolog-level implementations of search strategies and the use of static analysis techniques.
- It reduces the negative impact of grounding the whole program before execution; grounding is lazily applied to the rules being considered during the

construction of an answer set, and the ground rules are not kept beyond their needed use.

GASP has been implemented in a prototype, implemented in SICStus Prolog (using the `clpfd` library) and available at `www.dimi.uniud.it/dovier/GASP`. GASP supports the use of numerical constraints in the ASP programs (providing language capabilities comparable to that of the system presented in [19]). In spite of the overheads introduced by the intermediate Prolog layer, GASP is performance-wise competitive; it is capable of outperforming systems like Smodels and Clasp especially in benchmarks where the ground image is large.

The ideas presented in this paper expand our preliminary work on comparing ASP and CLP methodologies [6] and development of computation-based characterizations of answer sets [15]. The work is also similar in spirit to the concurrently proposed ASPeRiX system [11]. Both GASP and ASPeRiX have their theoretical roots in the same notion of computation-based characterization of answer sets [15]. ASPeRiX is implemented in C++ and develops heuristics aimed at enhancing the choice of the rules when more of them are applicable. Models for non-ground computation based on alternative execution schemes (e.g., top-down computations) have also been recently proposed (e.g., [3]).

2 The Language GASP

Syntax. The signature $\Sigma = \langle \Pi_C \cup \Pi_U, \mathcal{F}, \mathcal{V} \rangle$ of the language is defined as follows. $\mathcal{V}$ is a denumerable set of variables. $\mathcal{F} = \mathbb{Z} \cup F_Z \cup F_U \cup \{'..'/2\}$ is the set of constant and function symbols of the language, where

- $\mathbb{Z} = \{0, -1, 1, -2, 2, \dots\}$ is a set of constants for the integer numbers
- F_Z is a set of function symbols representing operations over integer numbers, such as $+, -, *, \mathsf{div}, \mathsf{mod}$, etc.;
- F_U is a (possibly empty) set of user-defined constant symbols, with the property that $F_U \cap (\mathbb{Z} \cup F_Z \cup \{`..'\}) = \emptyset$.
- '..' is a binary function symbol used to build intervals.

Π_U is the set of user-defined predicate symbols, while Π_C is the set of constraint predicate symbols (we assume that $\Pi_C \cap \Pi_U = \emptyset$).

Each Σ-term of the form $a..b$ is well-formed iff a, b are integer constants and $a \leq b$; we will refer to this type of terms as *interval terms*. Each Σ-term of the form $f(t_1, \dots, t_k)$, $f \in F_Z$, is well-formed iff $t_1, \dots, t_k$ are either variables, or integer constants, or (recursively) well-formed compound terms of the same form. Well-formed Σ-terms of the above form are called *compound integer terms*. User-defined function symbols are not allowed in our language.

$\langle \Pi_U, \mathcal{F}, \mathcal{V} \rangle$-atoms are *user-defined atoms*, while $\langle \Pi_C, \mathcal{F}, \mathcal{V} \rangle$-atoms are *constraint atoms* (or *primitive constraints*). We assume that interval terms can occur only in user-defined atoms (namely, in rule's head atoms—see below), while compound integer terms can occur only in constraint atoms. Negated literals have the form **not** A, where A is a (positive) Σ-atom.

The set of constraint predicate symbols Π_C of our language includes $=$, $\neq$ and $\leq$. $t_1 = t_2$, $t_1 \neq t_2$ are well-formed iff t_1 and t_2 are Σ-terms, while $t_1 \leq t_2$ is well-formed iff t_1 and t_2 are integer terms (either constants, variables, or compound integer terms). These symbols represent, respectively, the (syntactic) equality and inequality over $\mathbb{Z} \cup F_U$ and the natural order relation over $\mathbb{Z}$.

A GASP-*constraint* is a conjunction of constraint atoms. Other integer predicates (e.g., $<$, $\geq$, and $>$) can be defined as GASP-constraints using $=$, $\neq$, and $\leq$. Examples of well-formed terms and atoms are: `p(1..10)`, `p` $\in \Pi_U$ and `X` $\neq$ `Y + 1`. Hereafter, we will consider only well-formed terms and atoms.

Let us observe that our approach is parametric w.r.t. the constraint domain considered. In the paper, however, we focus on integer constraints.

A GASP-*rule* has the form $H \leftarrow B_1, \ldots, B_k$, where H is a user-defined atom or false, and $B_1, \ldots, B_k$ are either user-defined literals or constraint atoms or true. A GASP-rule of the form $H \leftarrow$ true (abbreviated H) is called a *fact*. A GASP-rule of the form false $\leftarrow B_1, \ldots, B_k$ (abbreviated $\leftarrow B_1, \ldots, B_k$) is called an *integrity constraint*. Intuitively, an integrity constraint $B_1, \ldots, B_k$ expresses the fact that we want to discard all models of the given program that entail $B_1 \wedge \ldots \wedge B_k$. A GASP-*program* is a collection of GASP-rules.

Given a GASP-rule $H \leftarrow B_1, \ldots, B_k$, let us denote with $\mathcal{U}\_body$ the collection of user-defined literals in $B_1, \ldots, B_k$, and with $\mathcal{C}\_body$ the collection of constraint atoms in $B_1, \ldots, B_k$. Hence, $H \leftarrow B_1, \ldots, B_k$ can be written as $H \leftarrow \mathcal{U}\_body, \mathcal{C}\_body$. Moreover we define with $body^+$ the collection of positive literals in $\mathcal{U}\_body \cup \mathcal{C}\_body$ and with $body^-$ the collection of atoms that appear in negative literals in $\mathcal{U}\_body$.

We assume that our language provides also special atoms called *cardinality constraints* [22]. Accordingly, Π_C includes the symbol $\{\}/3$ which is used to construct cardinality constraints of the form $h\{\varphi\}k$. $h\{\varphi\}k$ is well-formed iff h and k are integer s.t. $0 \leq h \leq k$, and φ is a sequence of atoms of the form $A : B_1, \ldots, B_n$, $n \geq 0$ (written A, if $n = 0$), where $A, B_1, \ldots, B_n$ are user-defined atoms and $B_1, \ldots, B_n$ occur as head atoms in some rules of the program. Furthermore, $vars(B_1, \ldots, B_n) \subseteq vars(A)$. Cardinality constraints can occur both in the head and in the body of a rule. When occurring in the head, their intuitive meaning is the following. $h\{A[\bar{X}, \bar{Y}_1, \ldots, \bar{Y}_n] : B_1[\bar{Y}_1], \ldots, B_n[\bar{Y}_n]\}k$ forces models of the given program to contain, for each tuple of ground terms $\bar{t}$ for $\bar{X}$, a set R such that $R \subseteq \{A : \bar{X} = \bar{t}, \exists \bar{Y}_1 \ldots \exists \bar{Y}_n (B_1, \ldots, B_n)\} \wedge h \leq |R| \leq k$. For example, given the program

$$r(1..3). \quad q(a). \quad q(b). \quad 1\{p(X,Y) : r(Y)\}1 \leftarrow q(X).$$

its models are required to contain exactly one among $p(a,1)$, $p(a,2)$, $p(a,3)$ and exactly one among $p(b,1)$, $p(b,2)$, $p(b,3)$.

When cardinality constraints occur in the body of a rule, they will be entailed by models that meet the above- mentioned property.

We assume, as done in several existing ASP systems, that programs satisfy the *range restriction* property, suitably adapted to account for constraints. A GASP-rule is range restricted if all variables occurring in its head (except those which are "local" to cardinality constraints) occur also in at least one positive atom of $\mathcal{U}\_body$. A GASP-program is range restricted iff every rule in it is range restricted. In this way, all variables in the program are guaranteed to have a finite set of possible values associated with.

Semantics. A GASP-program can be seen as a syntactic shorthand for an ASP program where any non-ground GASP-rule represents a family of ground ASP rules. Let $\mathcal{A}$ be a collection of propositional atoms. An *ASP rule* has the form:

$$p \leftarrow p_0, \ldots, p_n, \textbf{not}\, p_{n+1}, \ldots, \textbf{not}\, p_m$$

where $\{p, p_0, \ldots, p_n, p_{n+1}, \ldots, p_m\} \subseteq \mathcal{A}$. An ASP program is a collection of ASP rules. The process of replacing each non-ground rule with an equivalent finite set of ground rules is called *grounding*.

A *ground instance* of a rule R of P is obtained from R by replacing all variables in it by ground terms built using the symbols in $\mathcal{F} \setminus (\{'..'\} \cup F_Z)$, respecting well-formedness of the resulting ground atoms. In addition, each variable $v \in \mathcal{V}$ that appears in a Σ-term whose functor is in $F_Z \cup \{'..'\}$ or that appears in GASP-constraints based on $\leq$ has to be grounded using an element of $\mathbb{Z}$. Additionally, note that:

- We omit compound integer terms from the grounding process—as these are meant to be evaluated and replaced with the constants representing the values of the compound terms (elements of $\mathbb{Z}$);
- We omit intervals. Instead, we expect the grounding process to replace each rule of the form $p(\bar{t}, a..b, \bar{s}) \leftarrow body$, with the set of rules

$$p(\bar{t}, a, \bar{s}) \leftarrow body \quad p(\bar{t}, a+1, \bar{s}) \leftarrow body \cdots p(\bar{t}, b, \bar{s}) \leftarrow body$$

- each ground constraint atom C is replaced with true or false depending on whether C is entailed or not in the traditional theory of integer arithmetic.

Let us note that $\mathcal{C}\_body$ disappears as soon as the program is grounded.

A ground program ground(P) is obtained from P by replacing all rules in P by all ground instances of all rules in P.

Integrity constraints are always removed from the generated program: an ASP integrity constraint $\leftarrow p_0, \ldots, p_n, \textbf{not}\, p_{n+1}, \ldots, \textbf{not}\, p_m$ is equivalent, for stable models, to $p \leftarrow \textbf{not}\, p, p_0, \ldots, p_n, \textbf{not}\, p_{n+1}, \ldots, \textbf{not}\, p_m$, where p is a new propositional atom. Similarly, rules containing cardinality constraints are replaced by a collection of rules that precisely capture their semantics.

Therefore we can use all definitions and results usually adopted in ASP to provide a semantics characterization of GASP-programs. In particular, we consider here the semantics based on the notion of *well-founded model* [24]. The *well-founded model* [24] of a general program P is a *3-interpretation* I, i.e., a pair $\langle I^+, I^- \rangle$ such $I^+ \cup I^- \subseteq \mathcal{A}$ and $I^+ \cap I^- = \emptyset$. I^+ denotes the atoms that are known to be true while I^- denotes those atoms that are known to be false.

The union between two 3-interpretations $I \cup J$, where $I = \langle I^+, I^- \rangle$ and $J = \langle J^+, J^- \rangle$, is defined as $\langle I^+ \cup J^+, I^- \cup J^- \rangle$. The intersection is defined similarly.

If $I^+ \cup I^- = \mathcal{A}$, then the interpretation I is said to be *complete*. Given two 3-interpretations I, J, we use $I \subseteq J$ to denote the fact that $I^+ \subseteq J^+$ and $I^- \subseteq J^-$. The notion of entailment for 3-interpretations can be defined as follows. If $p \in \mathcal{A}$, then $I \models p$ iff $p \in I^+$; $I \models \mathbf{not}\, p$ iff $p \in I^-$; $I \models A \wedge B$ iff $I \models A$ and $I \models B$, and $I \models H \leftarrow A_1 \wedge \ldots \wedge A_n$ iff $I \models H$ or there is $i \in \{1, \ldots, n\}$ such that $I \models \mathbf{not}\, A_i$.

Intuitively, the well-founded model of P contains only (possibly not all) literals that are necessarily true and the ones that are necessarily false in all stable models of P. The remaining literals are undefined. It is well-known that a general program P has a unique well-founded model $\mathsf{wf}(P)$ [24]. If $\mathsf{wf}(P)$ is complete then it is also a stable model (and it is the unique stable model of P).

3 Computation-Based Characterization of Stable Models

The computation model adopted in GASP has been derived from recent investigations about alternative models to characterize answer set semantics for various extensions of ASP—e.g., programs with *abstract constraint atoms* [17].

The work described in [15] provides a *computation-based* characterization of answer sets for programs with abstract constraints. One of the side-effects of that research is the development of a computation-based view of answer sets for general logic programs. The original definition of answer sets [10] requires guessing an interpretation and successively validating it—through the notion of reduct (P^I) and the ability to compute minimal models of a definite program (e.g., via repeated iterations of the immediate consequence operator [16]).

The characterization of answer sets derived from [15] does not require the initial guessing of a complete interpretation; instead it combines the guessing process with the construction of the answer set.

We provide our formalization of computation in terms of GASP-*computation* a nd show that it is an instance of [15]. We begin with the following notion:

Definition 1 (Applicable rule). *We say that a ground rule $a \leftarrow body$ is* applicable *w.r.t. an interpretation I, if $body^+ \subseteq I^+$ and $body^- \cap I^+ = \emptyset$.*

We extend the definition of applicable to a non-ground rule R w.r.t. I iff there exists a grounding r of R that is applicable w.r.t. I. Note that $\mathcal{C}\_body$ is replaced by true during the local grounding stage.

Given a program P and an interpretation I, we denote with $P \cup I$ the program

$$P \cup I = (P \setminus \{r \in P \mid head(r) \in I^-\}) \cup I^+.$$

Intuitively, $P \cup I$ is the program P modified in such a way to guarantee that all elements in I^+ are true and all elements in I^- are false.

Definition 2 (GASP-computation). *A* GASP-computation *of a program P is a sequence of 3-interpretations $I_0, I_1, I_2, \ldots$ that satisfies the following properties:*

- $I_0 = \mathsf{wf}(P)$
- $I_i \subseteq I_{i+1}$ *for all* $i \geq 0$ *(*Persistence of Beliefs*)*

- *if* $I = \bigcup_{i=0}^{\infty} I_i$*, then* $\langle I^+, \mathcal{A} \setminus I^+ \rangle$ *is a model of* P *(*Convergence*)*
- *for each* $i \geq 0$ *there exists a rule* $a \leftarrow body$ *in* P *that is applicable w.r.t.* I_i *and* $I_{i+1} = \mathsf{wf}(P \cup I_i \cup \langle body^+, body^- \rangle)$ *(*Revision*)*
- *if* $a \in I_{i+1}^+ \setminus I_i^+$ *then there is a rule* $a \leftarrow body$ *in* P *which is applicable w.r.t.* I_j*, for each* $j \geq i$ *(*Persistence of Reason*).*

The proofs of correctness and completeness of GASP-computation w.r.t. the answer sets of a program P can be found at `http://sole.dimi.uniud.it/~agostino.dovier/GASP/`.

4 A CLP Approach for Stable Models Computation

In this section we show how the GASP-computation can be implemented within a CLP(FD) framework. The use of a Prolog based implementation allows fast prototyping of the search techniques and heuristics. The CLP environment allows non-ground computation of arithmetic constraints to be easily embedded into the implemented system.

4.1 Representation of Interpretations Based on FDSETs

Instances of a predicate that are true and false within an interpretation are encoded as sets of tuples, and handled using FD techniques. In order to compute the set of ground applicable rules, a local grounding phase is performed according to the definition of applicable rule, i.e. only compatible assignments of the rule w.r.t. the current interpretations are considered. During the construction of a model, the effect of this strategy is to ground only those rules that effectively contribute in supporting each stable model. Moreover, when arithmetic constraints are present in a rule, the ability to treat them in their non-ground version, allows to save on the enumeration of all possible admissible combinations of their ground instances.

The computation of applicable rules is at the basis of the GASP-computation and it is performed very frequently, i.e. at every node of the computation tree. From a relational algebra point of view, this can be seen as a set of join and projection operations on a set of tuples. If performed naively, these operations may become inefficient, especially when the number of tuples increases. We cope with this problem by introducing a *compact* and dynamic representation of the interpretations based on FDSETs. This allows us to efficiently handle large sets of tuples with respect to memory usage and query time. The compact representation is withdrawn to build a CSP whose solutions correspond to the applicable ground rules.

FDSETs are a data structure available in the clpfd library of SICStus Prolog that allows to efficiently store and compute on sets of integer numbers. Basically, a set $\{a_1, a_2, \ldots, a_n\}$ is interpreted as the union of a set of intervals $[a_{b_1}..a_{e_1}], \ldots, [a_{b_k}..a_{e_k}]$ and stored consequently as $[[a_{b_1}|a_{e_1}], \ldots, [a_{b_k}|a_{e_k}]]$. A library of built-in predicates for dealing with this data structure is made available.

We identify with p^n a predicate p with arity n. In the program, a predicate p^n appears as $p(X_1, \dots, X_n)$ where, in place of some variables, a constant can occur (e.g., $p(a, X, Y, d)$). The interpretation of the predicate p^n can be modeled as a set of tuples $(a_1, a_2, \dots, a_n)$, where $a_i \in \mathit{Consts}(P)$—where $\mathit{Consts}(P)$ denotes the set of constants in the language used by the program P. The explicit representation of the set of tuples has the maximal cardinality $|\mathit{Consts}(P)|^n$. The idea is to use a more compact representation based on FDSETs, after a mapping of tuples to integers. Without loss of generality, we assume that $\mathit{Consts}(P) \subseteq \mathbb{N}$. Each tuple $\boldsymbol{a} = (a_0, \dots, a_{n-1})$ is mapped to the *unique* number $\mathsf{map}(\boldsymbol{a}) = \sum_{i \in [0..n-1]} a_i \mathbb{M}^i$, where $\mathbb{M}$ is a "big number", $\mathbb{M} \geq |\mathit{Consts}(P)|$. In case of predicates without arguments (predicates of arity 0), for the empty tuple () we set $\mathsf{map}(()) = 0$. We also extend the map function to the case of non-ground tuples, using FD variables. If $\boldsymbol{Y} = (y_1, y_2, \dots, y_n)$, where $y_i \in \mathit{Vars}(P) \cup \mathit{Consts}(P)$, then $\mathsf{map}(\boldsymbol{Y})$ is the FD expression that represent the sum defined above. For instance, if $\boldsymbol{Y} = (3, X, 1, Y)$ and $\mathbb{M} = 10$, then $\mathsf{map}(\boldsymbol{Y}) = 3 + X * 10 + 1 * 10^2 + Y * 10^3$. Moreover, all variables possibly occurring in $\boldsymbol{Y}$ are constrained to have domain $0..\mathbb{M} - 1$.
A 3-interpretation $\langle I^+, I^- \rangle$ is represented by a set of 4-tuples $(p, n, \mathtt{POS}_{p,n}, \mathtt{NEG}_{p,n})$, one for each predicate symbol, where p is the predicate name, n its arity, and

$$\mathtt{POS}_{p,n} = \{\mathsf{map}(\boldsymbol{x}) : I^+ \models p(\boldsymbol{x})\} \qquad \mathtt{NEG}_{p,n} = \{\mathsf{map}(\boldsymbol{x}) : I^- \models \textbf{not}\, p(\boldsymbol{x})\}$$

If clear from the context, we drop the subscript n from the notation. These sets are represented and handled efficiently, by using FDSETs. For instance, if

$$\mathtt{POS}_{p,3} = \{\mathsf{map}(0,0,1), \mathsf{map}(0,0,2), \mathsf{map}(0,0,3), \mathsf{map}(0,0,8), \mathsf{map}(0,0,9), \mathsf{map}(0,1,0), \mathsf{map}(0,1,1), \mathsf{map}(0,1,2)\}$$

and $\mathbb{M} = 10$, then its representation as FDSETs is simply: [[1|3], [8|12]], in other words, the disjunction of two intervals.

4.2 Computing Applicable Rules

We briefly show now how local grounding is performed, highlighting the role of arithmetic constraints in the rule.

The idea is to build a CSP where the variables appearing in the rule correspond to FD variables. Solutions to the CSP correspond to ground rules that are applicable. According to our definition, the applicable rule has its *body*$^+$ that is completely contained in I^+ and its *body*$^-$ that has no ground predicate that appears in I^+. These requirements can be encoded in terms of FD constraints, by linking the FD variables appearing in a predicate to the values associated to the predicate by the function map and contained in the FDSET representation of I. More formally, let us assume a rule

$$r \equiv p_0(\boldsymbol{X}_0) \leftarrow C(\boldsymbol{X}), p_1(\boldsymbol{X}_{p_1}), \dots, p_k(\boldsymbol{X}_{p_k}), \textbf{not}\ \ p_{n+1}(\boldsymbol{X}_{p_{n+1}}), \dots, \textbf{not}\ \ p_m(\boldsymbol{X}_{p_m}),$$

where C is $\mathcal{C}$*_body* of r (namely a conjunction of arithmetic constraints), $\boldsymbol{X}_i$ is a list of variables and/or ground integers that are compatible to the arity of p_i.

For each variable in the rule, a corresponding FD variable is defined. Moreover, for each predicate p_i another FD variable V_i is created.

Every variable V_i is bounded to the corresponding variables $\boldsymbol{X}_i$, according to the $\mathsf{map}(\boldsymbol{X}_i)$ function, i.e. $V_i = \mathsf{map}(\boldsymbol{X}_i)$. Moreover, in order to implement the semantics of applicable rule, we require that each predicate p_i in *body*$^+$ has domain $\texttt{POS}_{p_i}$ and each predicate p_i in *body*$^-$ can not take values from $\texttt{POS}_{p_i}$. In addition we require that the head does not appear in I, since in that case it would be already supported and the rule would be applied redundantly. Finally, the constraint C is added to the CSP by introducing the corresponding FD constraints. This choice allows GASP to be easily extensible and to support a wide range of constraints in a modular way.

The solutions of this CSP are all the ground instances of V_0, computed through labeling, that represent the possible head values to be added to the model.

The same technique is adapted to compute T_P and the single steps of the alternating fixpoint procedure for computing the well-founded models.

4.3 The Overall Algorithm

We describe now the methodology followed in the implementation of the GASP-computation. We distinguish among three cases: the program is positive, the program admits a well-founded model, the program does not admit a complete well-founded model (it can have zero or more stable models).

In the first case, the computation of T_P operator fixpoint is performed, and the resulting interpretation is the only stable model. The computation of the fixpoint uses similar techniques to the CSP-based computation of the applicable rule, and it makes use of a dependency graph in order to select the rules to activate during the fixpoint.

In the second case, the idea of alternating fixpoint [25] is coded in Prolog. The implementation boils down to controlling the alternating fixpoint computation and to encode the $T_{P,J}$ operator (see e.g., [25]). Once again, similar CSPs to applicable rule computations are added in order to compute the $T_{P,J}$ operators.

In the third case, the GASP-computation is launched starting from that model. Instead of starting from an empty model, literals that are necessarily true and false respectively in each stable model are included in the starting model and lesser application of rules are required.

The GASP-computation is implemented through a chronological backtracking search where choice points contain the option whether to apply an applicable rule or not. The key ingredients of the main loop are: the computation of an extension of the T_P operator fixpoint, the handling of some specific cardinality constraint and the implementation of some rule-based propagators.

In Figure 1, we summarize in pseudocode the algorithm. Each applicable rule represents a non-deterministic choice in the computation of a stable model. The computation explores the first of these choices (line 4), and acts depending on the head a of the rule. In case the head is a cardinality constraint (we currently support exactly one, but this can be extended in the future), a non-deterministic assignment is added to the model, where one literal out of the possible candidates

```
(1)   rec_search(P,I)
(2)      $R$ = applicable_rules(I)
(3)      if ($R = \emptyset$ and I is a model) output: I is a stable model
(4)      else select $a \leftarrow body^+, \textbf{not}\, body^- \in R$
(5)         if ($a =$ 1{...}1)
(6)            ND-choice: I = $\langle$assignment, $body^-\rangle \cup$ I
(7)            I = fixpoint(propagation(P,$T_P$($P \cup (\langle \emptyset, body^- \rangle \cup$ I))))
(8)         else
(9)            ( I = $\langle \{a\}, body^- \rangle \cup$ I,
(10)             I = fixpoint(propagation(P,$T_P$($P \cup (\langle \emptyset, body^- \rangle \cup$I))))
(11)           OR (Non-deterministic)
(12)             P= P $\cup \{\leftarrow body^-\}$,
(13)             I = fixpoint(propagation(P,$T_P$($P \cup$I))))
(14)        if (I not failed) rec_search(P,I)
```

Fig. 1. The answer set computation

is added to I^+ and all the remaining to I^- (line 6). After the assignment, a fixpoint over the computation of T_P and propagators is performed before entering the recursive call. The propagation phase will be discussed in the next section.

If the head is a normal literal (line 9) then a non-deterministic choice is opened (lines 9 or 12). In the first part, the rule is applied and thus a and $body^-$ are added to I. After the fixpoint (line 10) the recursive call (line 14) is performed. In the second part, we consider the case in which the rule is never chosen in the subtree and to ensure this a new integrity constraint is added to the program (line 12). After the fixpoint the recursive call is made.

Let us recall that every time the local grounding in invoked, a CSP is built. We believe that the enhancement of this step (e.g., building CSPs less often and/or incremental CSP) could reduce the search time significantly. In line 14 "I not failed" means that $I^+ \cap I^- \neq \emptyset$.

The process may encounter a contradiction while adding new facts to the interpretation, and consequently the computation may encounter failures. Whenever there are no more applicable rules, a leaf in the search tree is reached (line 3) and the corresponding stable model is obtained (convergence property).

From the implementation point of view, it turns out that computing well-founded models at every non-deterministic application of a rule is time consuming. In particular, the computation of the extension of P with new facts from the positive interpretation is inefficient.

To gain efficiency, we substitute the call to the well-founded computation with a variant of the T_P operator. The extension of T_P to ASP considers rules where $body^+ \in I^+$ and $body^- \in I^-$. The T_P operator adds new positive atoms as stated by the head of the rule. Using well-founded computation involves the alternating fixpoint procedure which is not efficient enough to be included at each level of the search. The combination of T_P fixpoint and our propagators provide better results. In future work we plan to improve the well-founded model computation algorithm and to use it in place of the T_P fixpoint.

4.4 Non-ground Propagation

A propagation step is launched before each leaf expansion, in order to deduce additional literals that can be safely introduced in the current interpretation and that neither T_P nor well-founded fixpoints are able to infer.

The ideas presented below represent a generalization of some of the techniques that drive the search in Smodels. In particular, we deal with non-ground rules and therefore we introduce a CSP-based analysis similar to the computation of the applicability of rules. The resolution of the CSP is designed to avoid the complete grounding of the rules involved. We address three settings where negative literals can be deduced: inferring a literal that appears (i) in the body, (ii) in the head and (iii) in a cardinality constraint in the head.

Let $I = \langle I^+, I^- \rangle$ be the current model, R be a non-ground ASP rule of the form $head(R) : -body^+(R), \mathbf{not}\ \ body^-(R)$ and R' a grounding of R.

The case (i) applies when there exists a grounding R' such that $head(R') \in I^-$. In this case the rule R' should not become applicable, otherwise $head(R')$ would be added in I^+ and generate a failure. We consider the specific situation in which $body(R')$ is completely satisfied except for exactly one undetermined literal $l \in body^+$ ($l \notin I^+ \cup I^-$). To prevent the rule R' to fire, the literal l is added to I^-.

The case (ii) applies when it is possible to deduce that an undetermined literal $l \notin I^+ \cup I^-$ may not be introduced in I^+ in any subsequent computation. The (ground) literal l can be introduced in I^+ only if there is at least one (potentially) applicable rule R' such that $head(R') = l$. If some literals $p \in body(R')$ are undetermined, we assume that they can potentially contribute to satisfy the body: i.e., if $p \in body^+(R')$ then p is assumed to be true and if $p \in body^-(R')$ then p is assumed to be false. If there is no such rule R' then the literal l can be safely added to I^-.

The case (iii) applies when a positive ground literal in I^+ and the predicate matches the cardinality constraint $(1\{...\}1)$ in the head of an applicable rule. In this case, every other literal in the same range can be safely set to false.

Note that the inference of positive literals is possible as well, however they can not be introduced in the model, unless a test for unfoundedness is performed (they must be supported by some chains of applicable rules). We plan as future work to investigate this kind of propagation that resembles a mixed top-down approach in the computation of stable models.

5 Experiments

The prototype implementing the ideas described above and all the tests described in this section are available at `www.dimi.uniud.it/dovier/GASP`. The prototype has been developed using SICStus Prolog 4.0 (`www.sics.se/isl/sicstuswww/site/`), chosen for its rich library of FDSET primitives. Although faster constraint solvers are available (e.g., Gecode), we prefer to stay in the realm of declarative programming.

Table 1. Timings. '-' means that computation was killed after 24 hours

Test	Param	Lparse	Smodels	Clasp	GASP
non_wf_graph	40	4.035	0.735	1.720	1.82
(all sol)	80	29.824	6.874	7.8	13.55
	160	235.039	61.568	45.6	120.81
	320	1,885	-	-	1,380.77
Send More Money (all)	none	55.69	0.01	0.01	3.43
Queens	22	0.310	172.5	0.05	0.62
(1st sol)	23	0.360	395.9	0.06	1.68
	24	0.415	220.0	0.08	1.20
	25	0.464	2,067.0	0.09	8.38
Squares	(19,7)	1.76	2.99	0.17	1.43
(1st sol)	(6,24)	88	371.71	17.37	0.49
	(6,45)	1,140	-	-	1.48
p2	100	0.84	0.286	.200	0.75
(all sol)	200	3.346	1.172	1.45	3.00
	300	8.338	2.691	5.54	7.72
	400	13.242	4.819	15.27	14.81

We performed some preliminary experiments, using different classes of ASP programs, and we report the execution times in Table 1. All the experiments have been performed on an AMD Opteron 2.2 GHz Linux Machine. For the ASP tests, we used Lparse 1.1.5, Smodels 2.33 (`www.tcs.hut.fi/Software/smodels/`), Clasp 1.1.0 (`www.cs.uni-potsdam.de/clasp`) and ASPeRiX 0.1 (`http://www.info.univ-angers.fr/pub/claire/asperix/`).

In Table 1 we report on the benchmarks we run to compare the performances of GASP and Lparse+Smodels and Lparse+Clasp. Times are in seconds.

The first set of benchmarks (non_wf_graph) is based on a non well-founded program inspired by a graph problem, where the parameter determines the size of the graph. The program admits two distinct stable models and basically computes a transitive closure h of a binary predicate p, then add the predicate r: $r(X,Y)$:- $h(X,Y)$, **not** $p(X,Y)$. Depending on the stable model, the predicate p is slightly modified. The preliminary computation of well-founded model returns a sub-model and the non-deterministic GASP computation procedure must be used. The grounding time (and size of the program—with $q = 320$ the file is 1.9GB) are not negligible. However, in large instances GASP outperforms Lparse+Smodels and Lparse+Clasp even removing the time spent for grounding.

The second benchmark is the classical Send + More = Money problem, coded with ASP. Here constraint propagation performed by CLP in GASP is the key for solving the problem efficiently. Compared to Lparse, it is interesting to note that even if the size of the ground program is small (53K and 1300 rules), it takes almost a minute to produce the file.

Table 2. GASP and ASPeRiX

Test	Param	GASP	ASPeRiX	GASP nodes	ASPeRiX nodes
p2	100	0.75	0.22	0	10,000
(all sol)	200	3.00	1.20	0	40,000
	300	7.72	3.53	0	90,000
	400	14.81	7.92	0	160,000
non_wf_graph	40	1.82	0.096	2	1
(all sol)	80	13.55	0.671	2	1
	160	120.81	5.315	2	1
	320	1,380.77	42.918	2	1

The third set of benchmarks is the N queens problem. Here, GASP outperforms Smodels, while the performances of Clasp are only biased by the grounding time.

The fourth set is taken from the CSPLIB (`www.csplib.org`). The problem # 9 is the perfect square placement problem, where a set of non-overlapping squares must be placed inside a larger square. We designed 3 test sets, where (N, S) indicates the number N of squares and S the master side: the set (19,7) contains 1 square (side 4), 5 squares (side 2) and 13 squares (side 1); the set (6,24) contains 1 square (side 16) and 5 squares (side 8); the set (6,45) contains 1 square (side 30) and 5 squares (side 15). In this test, the performances of GASP are impressive, since the non-ground computation takes advantage of FD constraint solving during the search. The time spent by Lparse increases dramatically with the size of the master square as well as the size of the ground program (for (6,24) we have 71MB and 3.5M rules, for (6,45) 945MB and 44M rules), thus making it impossible for the solvers to find a solution.

Finally we included last test p2 taken from [11] and used by authors to prove the effectiveness of ASPeRiX in a case with large grounding when one is interested in a single solution. The program admits a stable model that contains a complete graph of a number of nodes that is provided as parameter. We can see that the GASP is capable of finding the solutions in the time needed to ground the program. Note that the ground program for 400 nodes has 22MB of size and contains 880K rules.

In Table 2 we compare the performances of GASP and ASPeRiX on programs that are supported by the latter (ASPeRiX does not support cardinality constraints). Since the approach is similar, we can compare the size of the search trees (number of choice points nodes). For the time comparisons, recall that ASPeRiX has a C++ implementation, while GASP is written in Prolog.

In the p2 program, despite the penalty for running into a Prolog environment, GASP timings are comparable and linearly scaled to ASPeRiX. Moreover, the rule-based propagators of GASP are able to reduce the search tree to a single node, while ASPeRiX develops a quadratic sized tree. On the other hand, the pruning of the tree in GASP represents the principal cost of the search.

In the second test (non_wf_graph), GASP is much slower than ASPeRiX, suggesting that the propagators used by GASP perform unnecessary work. This issue will be considered in future work. In the current implementation, when propagation is performed, CSPs are built on the current interpretation and they ignore the partial work performed in previous runs. Considering incremental versions of these CSPs could save the largest fraction of time currently used.

6 Conclusions

In this paper, we provided the foundation for a bottom-up construction of stable models of a program P without preliminary program grounding. The notion of GASP-computation has been introduced; this model does not rely on the explicit grounding of the program. Instead, the grounding is local and performed on-demand during the computation of the answer sets. The GASP language handles cardinality constraints and arithmetic constraints that can be implemented in a non-ground fashion and provide significant enhancements in the computation. We believe this approach can provide an effective avenue to achieve greater efficiency in space and time w.r.t. a complete program grounding.

We illustrated our Prolog implementation of GASP using CSP on FD variables and FDSETs. The performances of GASP show that, notwithstanding the Prolog overhead and naive data structures used, the computations are comparable and often better than traditional ground-based approaches.

We plan to investigate how to handle incremental CSPs in order to save redundant work, to reimplement efficiently the well-founded computation and include it in the main loop, to study a mixed goal-driven resolution (top-down approach) that should guide the non-deterministic choices.

Acknowledgments. The work has been partially supported by MIUR FIRB RBNE03B8KK and PRIN projects, and NSF grants HRD0420407, CNS0220590 and IIS0812267. We really thank Andrea Formisano for the several useful discussions.

References

1. Babovich, Y., Maratea, M.: Cmodels-2: SAT-based Answer Sets Solver Enhanced to Non-tight Programs. In: Lifschitz, V., Niemelä, I. (eds.) LPNMR 2004. LNCS, vol. 2923, pp. 346–350. Springer, Heidelberg (2003)
2. Baral, C.: Knowledge Representation, Reasoning, and Declarative Problem Solving. Cambridge University Press, Cambridge (2003)
3. Bonatti, P., Pontelli, E., Son, T.: Credulous Resolution for ASP. In: AAAI (2008)
4. Brooks, D., Erdem, E., Erdogan, S., Minett, J., Ringe, D.: Inferring Phylogenetic Trees Using Answer Set Programming. JAR 39(4), 471–511 (2007)
5. Codognet, P., Diaz, D.: A Minimal Extension of the WAM for clp(fd). In: ICLP, pp. 774–790. MIT Press, Cambridge (1993)
6. Dovier, A., Formisano, A., Pontelli, E.: A Comparison of CLP(FD) and ASP Solutions to NP-Complete Problems. In: Gabbrielli, M., Gupta, G. (eds.) ICLP 2005. LNCS, vol. 3668, pp. 67–82. Springer, Heidelberg (2005)

7. Elkabani, I., Pontelli, E., Son, T.: A System for Computing Answer Sets of Logic Programs with Aggregates. In: Baral, C., Greco, G., Leone, N., Terracina, G. (eds.) LPNMR 2005. LNCS, vol. 3662, pp. 427–431. Springer, Heidelberg (2005)
8. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Clasp: a Conflict-driven Answer Set Solver. In: Baral, C., Brewka, G., Schlipf, J. (eds.) LPNMR 2007. LNCS, vol. 4483, pp. 260–265. Springer, Heidelberg (2007)
9. Gebser, M., Schaub, T., Thiele, S., Usadel, B., Veber, P.: Detecting Inconsistencies in Large Biological Networks with ASP. In: Garcia de la Banda, M., Pontelli, E. (eds.) ICLP 2008. LNCS, vol. 5366, pp. 130–144. Springer, Heidelberg (2008)
10. Gelfond, M., Lifschitz, V.: The Stable Model Semantics for Logic Programs. In: ICLP, pp. 1070–1080. MIT Press, Cambridge (1988)
11. Lefevre, C., Nicolas, P.: Integrating Grounding in Search Process for Answer Set Computing. In: Work. on Integrating ASP and Other Computing Paradigms (2008)
12. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Perri, G.S., Scarcello, F.: The DLV System for Knowledge Representation and Reasoning. ACM Transactions on Computational Logic 7(3), 499–562 (2006)
13. Lifschitz, V.: Answer Set Planning. In: Ganzinger, H., McAllester, D., Voronkov, A. (eds.) LPAR 1999. LNCS, vol. 1705, pp. 373–374. Springer, Heidelberg (1999)
14. Lin, F., Zhao, Y.: ASSAT: Computing answer sets of a logic program by SAT solvers. Artificial Intelligence 157(1-2), 115–137 (2004)
15. Liu, L., Pontelli, E., Tran, S., Truszczynski, M.: Logic Programs with Abstract Constraint Atoms: the Role of Computations. In: Dahl, V., Niemelä, I. (eds.) ICLP 2007. LNCS, vol. 4670, pp. 286–301. Springer, Heidelberg (2007)
16. Lloyd, J.W.: Foundations of Logic Programming. Springer, Heidelberg (1987)
17. Marek, V., Remmel, J.: Set Constraints in Logic Programming. In: Lifschitz, V., Niemelä, I. (eds.) LPNMR 2004. LNCS, vol. 2923, pp. 167–179. Springer, Heidelberg (2003)
18. Marek, V.W., Truszczyński, M.: Stable Models and an Alternative Logic Programming Paradigm. In: Apt, K.R., Marek, V.W., Truszcziński, M., Warren, D.S. (eds.) The Logic Programming Paradigm. Springer, Heidelberg (1999)
19. Mellarkod, V., Gelfond, M.: Integrating Answer Set Reasoning with Constraint Solving Techniques. In: Garrigue, J., Hermenegildo, M.V. (eds.) FLOPS 2008. LNCS, vol. 4989, pp. 15–31. Springer, Heidelberg (2008)
20. Niemelä, I.: Logic Programs with Stable Model Semantics as a Constraint Programming Paradigm. Annals of Mathematics and AI 25(3-4), 241–273 (1999)
21. Niemelä, I., Simons, P.: Smodels - An Implementation of the Stable Model and Well-Founded Semantics for Normal LP. In: Fuhrbach, U., Dix, J., Nerode, A. (eds.) LPNMR 1997. LNCS, vol. 1265, pp. 421–430. Springer, Heidelberg (1997)
22. Simons, P., Niemelä, I., Soininen, T.: Extending and implementing the stable model semantics. Artificial Intelligence 138(1-2), 181–234 (2002)
23. Son, T., Pontelli, E.: Planning for Biochemical Pathways: a Case Study of Answer Set Planning in Large Planning Problem Instances. In: First International Workshop on Software Engineering for Answer Set Programming, pp. 116–130 (2007)
24. Van Gelder, A., Ross, K.A., Schlipf, J.S.: The Well-Founded Semantics for General Logic Programs. Journal of the ACM 38(3), 620–650 (1991)
25. Zukowski, U., Freitag, B., Brass, S.: Improving the Alternating Fixpoint: The Transformation Approach. In: Fuhrbach, U., Dix, J., Nerode, A. (eds.) LPNMR 1997. LNCS, vol. 1265, pp. 4–59. Springer, Heidelberg (1997)

Computing Loops with at Most One External Support Rule for Disjunctive Logic Programs

Xiaoping Chen[1], Jianmin Ji[1], and Fangzhen Lin[2]

[1] School of Computer Science and Technology,
University of Science and Technology of China, P.R. China
xpchen@ustc.edu.cn, jizheng@mail.ustc.edu.cn
[2] Department of Computer Science and Engineering
Hong Kong University of Science and Technology
flin@cs.ust.hk

Abstract. We extend to disjunctive logic programs our previous work on computing loop formulas of loops with at most one external support. We show that for these logic programs, loop formulas of loops with no external support can be computed in polynomial time, and if the given program has no constraints, an iterative procedure based on these formulas, the program completion, and unit propagation computes the least fixed point of a simplification operator used by DLV. We also relate loops with no external supports to the unfounded sets and the well-founded semantics of disjunctive logic programs by Wang and Zhou. However, the problem of computing loop formulas of loops with at most one external support rule is NP-hard for disjunctive logic programs. We thus propose a polynomial algorithm for computing some of these loop formulas, and show experimentally that this polynomial approximation algorithm can be effective in practice.

1 Introduction

This paper is about Answer Set Programming (ASP) where the main computational task is to compute the answer sets of a logic program. In this context, consequences of a logic program, those that are true in all answer sets, are of interest as they can be used to simplify the given program and help computing its answer sets. The best known example is the well-founded model for normal logic programs, which always exists and can be computed efficiently. All literals in the well-founded model are consequences, and in all current ASP solvers, a logic program is first simplified by its well-founded model. A natural question then is whether there are other consequences of a logic program that can be computed efficiently and used to simplify the given logic program. Motivated by this, Chen *et al.* [1] proposed an iterative procedure of computing consequences of a non-disjunctive logic program based on unit propagation, the program's completion and its loop formulas. They showed that when restricted to loops with no external support, their procedure basically computes the well-founded model. They also considered using loops with at most one external support, and showed that the loop formulas of these loops can be computed in polynomial time.

In this paper, we consider extending this work to disjunctive logic programs. As expected, loops with no external supports are closely related to well-founded models and greatest unfounded sets in disjunctive logic programs as well. However, many other issues are more complicated in disjunctive logic programs. In particular, the problem of computing the loop formulas of loops with at most one external support rule is NP-hard.

This paper is organized as follows. We briefly review the basic notions of logic programming in the next section. We then define loops with at most one external support rule under a given set of literals, and consider how to compute their loop formulas. We then consider how to use these loop formulas to derive consequences of a disjunctive logic program using unit propagation, and discuss related work, especially the greatest unfounded sets, the pre-processing step in DLV, and the well-founded semantics for disjunctive logic programs proposed by Wang and Zhou [2].

2 Preliminaries

In this paper, we consider only fully grounded finite disjunctive logic programs. A *disjunctive logic program* is a finite set of (disjunctive) rules of the form

$$a_1 \vee \cdots \vee a_k \leftarrow a_{k+1}, \ldots, a_m, not\, a_{m+1}, \ldots, not\, a_n, \tag{1}$$

where $n \geq m \geq k \geq 0$ and $a_1, \ldots, a_n$ are atoms. If $k = 0$, then this rule is called a *constraint*, if $k \neq 0$, it is a *proper rule*, and if $k = 1$, it is a *normal rule*. In particular, a *normal logic program* is a finite set of constraints and normal rules.

We will also write rule r of form (1) as

$$head(r) \leftarrow body(r), \tag{2}$$

where $head(r)$ is $a_1 \vee \cdots \vee a_k$, $body(r) = body^+(r) \wedge body^-(r)$, $body^+(r)$ is $a_{k+1} \wedge \cdots \wedge a_m$, and $body^-(r)$ is $\neg a_{m+1} \wedge \cdots \wedge \neg a_n$, and we identify $head(r)$, $body^+(r)$, $body^-(r)$ with their corresponding sets of atoms, and $body(r)$ the set $\{ a_{k+1}, \ldots, a_m, \neg a_{m+1}, \ldots, \neg a_n \}$ of literals obtained from the body of the rule with "*not*" replaced by "$\neg$".

Given a disjunctive logic program P, we denote by $Atoms(P)$ the set of atoms in it, and $Lit(P)$ the set of literals constructed from $Atoms(P)$:

$$Lit(P) = Atoms(P) \cup \{\neg a \mid a \in Atoms(P)\}.$$

Given a literal l, the *complement* of l, written $\bar{l}$ below, is $\neg a$ if l is a and a if l is $\neg a$, where a is an atom. For a set L of literals, we let $\overline{L} = \{ \bar{l} \mid l \in L \}$.

2.1 Answer Sets

The *answer sets* of a disjunctive logic program is defined as in [3]. Given a disjunctive logic program P and a set S of atoms, the Gelfond-Lifschitz transformation of P on S, written P_S, is obtained from P by deleting:

1. each rule that has a formula *not p* in its body with $p \in S$, and
2. all formulas of the form *not p* in the bodies of the remaining rules.

Clearly for any S, P_S is the set of rules without any negative literals, so that P_S has a set of minimal models, denoted by $\Gamma_P(S)$. Now a set S of atoms is an answer set of P iff $S \in \Gamma_P(S)$.

2.2 Completions

The *completion* of a disjunctive logic program P [4], $Comp(P)$, is defined to be the set of propositional formulas that consists of the implication

$$body(r) \supset head(r), \tag{3}$$

for every rule r in P, and the implication

$$a \supset \bigvee_{r \in P,\ a \in head(r)} \left(body(r) \wedge \bigwedge_{p \in head(r) \setminus \{a\}} \neg p \right), \tag{4}$$

for each atom $a \in Atoms(P)$. Note that, if P is a normal logic program without constraints, $Comp(P)$ is equivalent to the Clark's completion of P [5].

We will convert the completion into a set of clauses, and use unit propagation as the inference rule. Since unit propagation is not logically complete, it matters how we transform the formulas in the completion into clauses. In the following, let $comp(P)$ be the set of following clauses:

1. for each $a \in Atoms(P)$, if there is no rule in P with a in its head, then add $\neg a$;
2. if r is not a constraint, then add $head(r) \vee \bigvee \overline{body(r)}$;
3. if r is a constraint, then add the clause $\bigvee \overline{body(r)}$;
4. if a is an atom and $r_1, \ldots, r_t$, $t > 0$, are all the rules in P with a in their heads, then introduce t new variables $v_1, \ldots, v_t$, and add the following clauses:

$$\begin{aligned}
&\neg a \vee v_1 \vee \cdots \vee v_t,\\
&v_i \vee \bigvee \overline{body(r_i)} \vee \bigvee_{p \in head(r_i) \setminus \{a\}} p, \text{ for each } 1 \le i \le t,\\
&\neg v_i \vee l, \text{ for each } l \in body(r_i) \cup \overline{head(r_i) \setminus \{a\}} \text{ and } 1 \le i \le t.
\end{aligned}$$

2.3 Loops and Loop Formulas

We now briefly review the notions of loops and loop formulas in disjunctive logic programs [4]. Given a disjunctive logic program P, the *positive dependency graph* of P, written G_P, is the directed graph whose vertices are atoms in P, and there is an arc from p to q if there is a rule $r \in P$ such that $p \in head(r)$ and $q \in body^+(r)$. A set L of atoms is said to be a loop of P if for any p and q

in L, there is a non-empty path from p to q in G_P such that all the vertices in the path are in L, i.e. the L-induced subgraph of G_P is strongly connected.

Given a loop L, a rule r is an *external support* of L if $head(r) \cap L \neq \emptyset$ and $L \cap body^+(r) = \emptyset$. In the following, let $R^-(L)$ be the set of external support rules of L. Then the *loop formula* of L under P, written $LF(L, P)$, is the following implication

$$\bigvee_{p \in L} p \supset \bigvee_{r \in R^-(L)} \left(body(r) \wedge \bigwedge_{q \in head(r) \setminus L} \neg q \right). \tag{5}$$

2.4 Unfounded Sets

The notion of unfounded sets for normal logic programs, which provide the basis for negative conclusions in the well-founded semantics [6], has been extended to disjunctive logic programs [7].

Let P be a disjunctive logic program, A be a set of literals. A set of atoms X is an *unfounded set* for P w.r.t. A if, for each $a \in X$, for each rule $r \in P$ such that $a \in head(r)$, at least one of the following conditions holds:

1. $\overline{A} \cap body(r) \neq \emptyset$, that is, the body of r is false w.r.t. A.
2. $body^+(r) \cap X \neq \emptyset$, that is, some positive body literal belongs to X.
3. $(head(r) \setminus X) \cap A \neq \emptyset$, that is, an atom in the head of r, distinct from elements in X, is true w.r.t. A.

Note that if P is a normal logic program, unfounded sets defined here coincide with the definition given for normal logic programs in [6]. For normal logic programs, the union of all unfounded sets w.r.t. A is also an unfounded set w.r.t. A (called the *greatest unfounded set*). But this is not generally true for disjunctive logic programs, thus for some disjunctive logic program P and set of literals A, the union of two unfounded sets is not an unfounded set and the greatest unfounded set of P w.r.t. A does not exist. From Proposition 3.7 in [7], the greatest unfounded set exists for any P if A is unfounded-free. Formally, a set of literals A is *unfounded-free* for a disjunctive logic program P, if $A \cap X = \emptyset$ for each unfounded set X for P w.r.t. A. If A is unfounded-free for P, then the greatest unfounded set exists. In the following, we use $GUS_P(A)$ to denote the greatest unfounded set for P w.r.t. A.

Loops and unfounded sets are closely related [8,9]. In this paper, we show that the greatest unfounded sets (if exist) can be computed from loops that have no external support rules.

2.5 Unit Propagation

We use unit propagation as the inference rule for deriving consequences from the completion and loop formulas of a logic program. Given a set Γ of clauses, we let $UP(\Gamma)$ be the set of literals that can be derived from Γ by unit propagation. Formally, it can be defined as follows:

```
Function $UP(\Gamma)$
    if $(\emptyset \in \Gamma)$ then return $Lit$;
    $A := unit\_clause(\Gamma)$;
    if $A$ is inconsistent then return $Lit$;
    if $A \neq \emptyset$ then return $A \cup UP(assign(A, \Gamma))$ else return $\emptyset$;
```

where $unit\_clause(\Gamma)$ returns the union of all unit clauses in Γ, and $assign(A, \Gamma)$ is $\{ c \mid \text{ for some } c' \in \Gamma, c' \cap A = \emptyset, \text{ and } c = c' \setminus \overline{A} \}$.

3 Loops with at Most One External Support

The basic theorem about loop formulas says that a set of atoms is an answer set of a logic program iff it is a model of the program's completion and loop formulas[1]. This is the case for normal logic programs [10] as well as disjunctive logic programs [4]. This means that a sentence is a consequence of a logic program iff it is a logical consequence of the program's completion and loop formulas. The problem is that logical entailment in propositional logic is coNP-complete, and that in the worst case, there may be an infinite number of loops and loop formulas. In [1], Chen *et al.* considered using unit propagation as the inference rule, and some special classes of loops whose loop formulas can be computed efficiently. In general terms, their procedure is as follows:

Input: a logic program P.
1. Initialize $U = \emptyset$, and convert $Comp(P)$ to a set C of clauses.
2. Based on U, compute a set of loop formulas, and convert them into a set L of loop formulas.
3. Let $K = \{\varphi \mid U \cup C \cup L \vdash_P \varphi\}$, where $\vdash_P$ is a sound inference rule in propositional logic (such as unit propagation).
4. If $K \setminus U = \emptyset$, then return K, else let $U = K$ and go back to step 2.

They showed that when $\vdash_P$ is unit propagation, and the class of loops under U is those that have no external support under U, then the above procedure basically computes the well-founded model for normal logic programs. They also considered loops with at most one external support and showed that their loop formulas can be computed efficiently.

Our main objectives are to extend these results to disjunctive logic programs. We consider first these loops can be computed in disjunctive logic programs.

3.1 Loops with No External Support

It is easy to see that if a loop L has no external support rules, i.e. $R^-(L) = \emptyset$, then its loop formula (5) is equivalent to $\bigwedge_{p \in L} \neg p$, if L has only one external support rule, i.e. $R^-(L) = \{r\}$, then its loop formulas (5) is equivalent to

$$\bigwedge_{p \in L} \neg p \vee \left(body(r) \wedge \bigwedge_{q \in head(r) \setminus L} \neg q \right),$$

[1] Or the program and its loop formulas if singletons are always considered loops.

which is equivalent to a set of binary clauses.

More generally, if we already know that A is a set of literals that are true in all answer sets, then for any loop L that has at most one external support rule whose body is *active* under A w.r.t. L, its loop formula is still equivalent to either a set of literals or a set of binary clauses under A. A rule r is *active* under A w.r.t. L if $\overline{A} \cap \mathit{body}(r) = \emptyset$ and $A \cap (\mathit{head}(r) \setminus L) = \emptyset$.

Thus we extend the notion of external support rules, and have it conditioned on a given set of literals. Let P be a disjunctive logic program, and A a set of literals. We say that a rule r is an *external support rule of* L *under* A if $r \in R^-(L)$ is active under A w.r.t. L. In the following, we denote by $R^-(L, A)$ the set of external support rules of L under A. Note that if P is a normal logic program, $R^-(L, A)$ defined here coincides with the same notion defined in [1].

Now given a disjunctive logic program P and a set A of literals, let

$$\begin{aligned} \mathit{loop}_0(P, A) &= \{\, L \mid L \text{ is a loop of } P \text{ such that } R^-(L, A) = \emptyset \,\}, \\ \mathit{floop}_0(P, A) &= \{\, \neg a \mid a \in L \text{ for a loop } L \in \mathit{loop}_0(P, A) \,\}. \end{aligned}$$

Then $\mathit{loop}_0(P, A)$ is the set of loops that do not have any external support rules under A, and $\mathit{floop}_0(P, A)$ is equivalent to the set of loop formulas of these loops. In particular, the set of loop formulas of loops without any external support rules is equivalent to $\mathit{floop}_0(P, \emptyset)$.

We now consider how to compute $\mathit{floop}_0(P, A)$. For normal logic programs, Chen *et al.* showed that $\mathit{floop}_0(P, A)$ can be computed in quadratic time. However, for disjunctive logic programs, the problem is NP-hard in the general case.

Proposition 1. *Given a disjunctive logic program P, a set A of literals, and an atom a, deciding whether $\neg a \in \mathit{floop}_0(P, A)$ is NP-complete.*

Fortunately, if the set A is unfounded-free[2], then $\mathit{floop}_0(P, A)$ can be computed in quadratic time. As we shall see, this restriction is enough for computing consequences of a logic program using the procedure outlined above when $\vdash_P$ is unit propagation and the class of loops is that of loops without external support.

Our algorithm below for computing $\mathit{floop}_0(P, A)$ is similar to the corresponding one in [1], and is through maximal loops.

Let $\mathit{ml}_0(P, A)$ be the set of maximal loops that do not have any external support rules under A: a loop is in $\mathit{ml}_0(P, A)$ if it is a loop of P such that $R^-(L, A) = \emptyset$, and there does not exist any other such loop L' such that $L \subset L'$. Clearly,

$$\mathit{floop}_0(P, A) = \bigcup_{L \in \mathit{ml}_0(P, A)} \overline{L}.$$

If P is a normal logic program, loops in $\mathit{ml}_0(P, A)$ are pair-wise disjoint [1]. For disjunctive logic programs the property is not true in general, thus the reason that $\mathit{floop}_0(P, A)$ is intractable. However, if A is unfounded-free, then loops in $\mathit{ml}_0(P, A)$ are pair-wise disjoint. This follows from the following proposition:

[2] Recall that A is unfounded-free if $A \cap X = \emptyset$ for each unfounded set X of P w.r.t. A.

Proposition 2. *Let P be a disjunctive logic program, A be a set of literals such that $A \cap (L_1 \cup L_2) = \emptyset$. If L_1 and L_2 are two loops of P that do not have any external support rules under A, and $L_1 \cap L_2 \neq \emptyset$, then $L_1 \cup L_2$ is also a loop of P that does not have any external support rules under A.*

The following example shows that the condition $A \cap (L_1 \cup L_2) = \emptyset$ in Proposition 2 is necessary.

Example 1. Consider the following disjunctive logic program P:

$$a \vee b \vee c \leftarrow . \qquad a \leftarrow b,\ c. \qquad b \leftarrow a. \qquad c \leftarrow a.$$

Let $A = \{b, c\}$, $L_1 = \{a, b\}$ and $L_2 = \{a, c\}$. L_1 and L_2 are belong to $loop_0(P, A)$, but $L_1 \cup L_2 = \{a, b, c\}$ is a loop of P that has one external support under A.

Now consider the following algorithm:

```
Function $ML_0(P, A, S)$: $P$ a program, $A$ and $S$ sets of literals of $P$
    $ML := \emptyset$; $G :=$ the $S$ induced subgraph of $G_P$;
    For each strongly connected component $L$ of $G$:
        if $R^-(L, A) = \emptyset$ then add $L$ to $ML$
        else append $ML_0(P, A, L \setminus \bigcup_{r \in R^-(L,A)} H(r, A))$ to $ML$.
    return $ML$,
```

where G_P is the positive dependency graph of P, and

$$H(r, A) = \begin{cases} head(r) & \text{if } head(r) \cap A = \emptyset \\ head(r) \cap A & \text{if } head(r) \cap A \neq \emptyset. \end{cases}$$

Theorem 1. *Let P be a disjunctive logic program, A and S sets of literals in P.*

1. *The function $ML_0(P, A, S)$ runs in $O(n^2)$, where n is the size of P as a set.*
2. *$ML_0(P, A, Atoms(P)) \subseteq loop_0(P, A)$.*
3. *If A is unfounded-free, then $ML_0(P, A, Atoms(P)) = ml_0(P, A)$.*

3.2 Loops with at Most One External Support

Similarly, we can consider the set of loops that have exactly one external support rule under a set A of literals, and the set of loop formulas of these loops:

$$loop_1(P, A) = \{\, L \mid L \text{ is a loop of } P \text{ such that } R^-(L, A) = \{r\}\,\},$$
$$floop_1(P, A) = \{\neg a \vee l \mid a \in L, l \in body(r) \cup \overline{head(r) \setminus L}, \text{ for some loop } L \text{ and rule } r \text{ such that } R^-(L, A) = \{r\}\,\}.$$

In particular, $floop_1(P, \emptyset)$ is equivalent to the set of loop formulas of the loops that have exactly one external support rule in P.

Like $floop_0(P, A)$, $floop_1(P, A)$ is intractable.

Proposition 3. *Given a disjunctive logic program P, a set A of literals, an atom a, and a literal l, deciding whether $\neg a \vee l \in \mathit{floop}_1(P, A)$ is NP-complete.*

While there is a polynomial algorithm for computing $\mathit{floop}_0(P, A)$ when A is unfounded-free, this is not the case for $\mathit{floop}_1(P, A)$. Proposition 3 holds even when we restrict A to be unfounded-free.

Notice that for normal logic programs, the complexity of $\mathit{floop}_1(P, A)$ is left as an open question in [1]. Instead, a polynomial algorithm is proposed for computing $\mathit{floop}_0(P, A) \cup \mathit{floop}_1(P, A)$[3] which corresponds to the set of loop formulas of loops with at most one external support [1]. For disjunctive logic programs, $\mathit{floop}_0(P, A) \cup \mathit{floop}_1(P, A)$ is still intractable even when A is unfounded-free[4].

Given this negative results about computing loop formulas of loops with at most one external support in disjunctive logic programs, we turn our attention to polynomial algorithms that can compute as many loop formulas from $\mathit{floop}_0(P, A) \cup \mathit{floop}_1(P, A)$ as possible. We propose one such approximation algorithm below. It is based on the observation that if a loop has one external support rule, then it often has no external support when this rule is deleted. This would reduce the problem of computing loops with one external support rule to that of loops with no external support, and for the latter we can use the function $ML_0(P, A, S)$ when A is unfounded-free (Theorem 1).

Proposition 4. *For any disjunctive logic program P and a set A of literals that is unfounded-free for P. $\mathit{floop}_0(P, A)$ and $\mathit{floop}_1(P, A)$ imply the following theory*

$$\bigcup_{\overline{A} \cap body(r) = \emptyset, L \in ML_0(P \setminus \{r\}, A, Atoms(P \setminus \{r\}))} \{\neg a \vee l \mid a \in L, l \in body(r) \cup \overline{head(r) \setminus L}\}. \tag{6}$$

In the following, we use $\mathit{fLoop}_1(P, A)$ to denote (6). According to Proposition 2 of [1], if P is a normal logic program, then $\mathit{floop}_0(P, A) \cup \mathit{floop}_1(P, A)$ is equivalent to $\mathit{floop}_0(P, A) \cup \mathit{fLoop}_1(P, A)$ for any A. However, for disjunctive logic programs, this two theories are not equivalent, even when A is unfounded-free, as the following example illustrates.

Example 2. Consider the following logic program P:

$$a \vee b \vee c \leftarrow d. \qquad a \leftarrow b, c. \qquad b \leftarrow a. \qquad c \leftarrow b.$$

Let $A = \emptyset$, $loop_1(P, A) = \{\, \{a, b, c\}, \{a, b\} \,\}$, both loops have one external support rule: $a \vee b \vee c \leftarrow d$, thus $\neg a \vee \neg c, \neg b \vee \neg c \in \mathit{floop}_1(P, A)$, but they can not be computed from $\mathit{fLoop}_1(P, A)$.

[3] Not exactly this set, but $\mathit{floop}_0(P, A) \cup \mathit{fLoop}_1(P, A)$, which is logically equivalent to $\mathit{floop}_0(P, A) \cup \mathit{floop}_1(P, A)$, and especially $UP(\mathit{floop}_0(P, A) \cup \mathit{fLoop}_1(P, A)) = UP(\mathit{floop}_0(P, A) \cup \mathit{floop}_1(P, A))$.

[4] For normal logic programs, we need to compute $T(P, A)$, we show that $floop_0(P, A) \supset (\mathit{floop}_1(P, A) \equiv \mathit{fLoop}_1(P, A))$ and furthermore $UP(\mathit{floop}_0(P, A) \cup \mathit{floop}_1(P, A)) = UP(\mathit{floop}_0(P, A) \cup \mathit{fLoop}_1(P, A))$. For disjunctive logic programs, deciding whether a literal $l \in T(P, A)$ or even $l \in UP(\mathit{floop}_0(P, A) \cup \mathit{floop}_1(P, A))$, A is unfounded-free, is NP-hard.

So to summarize, while we can not efficiently compute $floop_0(P, A) \cup floop_1(P, A)$, we can compute $floop_0(P, A) \cup fLoop_1(P, A)$ which is still helpful for computing consequences of a logic program. To compute $floop_0(P, A) \cup fLoop_1(P, A)$, we first call $ML_0(P, A, Atoms(P))$, and then for each proper rule $r \in P$ such that $\overline{A} \cap body(r) = \emptyset$, we call $ML_0(P \setminus \{r\}, A, Atoms(P \setminus \{r\}))$. The worse case complexity of this procedure is $O(n^3)$, where n is the size of P.

4 Computing Consequences of a Program

Let's now return to the iterative procedure given in the beginning of last section. When $\vdash_P$ is unit propagation UP, and the loop formulas are those from ML_0 (maximal loops with no external support), it becomes the following one:

```
**Function** $T_0(P)$ - $P$ is a disjunctive logic program;
$X := \emptyset$; $Y := comp(P) \cup \{$loop formulas of loops in $ML_0(P, \emptyset, Atoms(P))\}$;
**while** $X \neq UP(Y)$ **do**
    $X := UP(Y)$; $Y := Y \cup \{$loop formulas of loops in $ML_0(P, X, Atoms(P))\}$;
**return** $X \cap Lit(P)$.
```

Clearly $T_0(P)$ runs in polynomial time and returns a set of consequences of P. It is also easy to see that at each iteration, the set X computed by the procedure is also a set of consequences of P. Thus by the following proposition and Theorem 1, if P has at least one answer set, then at each iteration, the set of literals added to Y, $\{$loop formulas of loops in $ML_0(P, X, Atoms(P))\}$, equals to $floop_0(P, X)$, the set of loop formulas with no external support under X.

Proposition 5. *Let P be a disjunctive logic program that has an answer set. If A is a set of literals that are consequences of P, then A is unfounded-free for P.*

Similarly, using $floop_0(P, A) \cup floop_1(P, A)$, we get the following procedure:

```
**Function** $T^1(P)$ - $P$ is a disjunctive logic program;
$Y := comp(P) \cup floop_0(P, \emptyset) \cup floop_1(P, \emptyset)$; $X := \emptyset$;
**while** $X \neq UP(Y)$ **do**
    $X := UP(Y)$; $Y := Y \cup floop_0(P, X) \cup floop_1(P, X)$;
**return** $X \cap Lit(P)$.
```

Again it is easy to see that at each iteration, X is a set of consequences of P, and in particular, $T^1(P)$ returns a set of consequences of P. As we have shown in the last section, even for unfounded-free A, computing $floop_0(P, X) \cup floop_1(P, X)$ is intractable. Thus we cannot show that the above procedure is polynomial. However, this still leaves open the question of whether $T^1(P)$ can be computed by some other methods that hopefully can be shown to run in polynomial time. Unfortunately, this does not seem to be likely as we can show that computing $T^1(P)$ is also intractable.

Proposition 6. *For any disjunctive logic program P, deciding whether a literal is in $T^1(P)$ is NP-hard.*

In the last section, we propose to use $fLoop_1(P, A)$ as a polynomial approximation of $floop_1(P, A)$. We can thus make use of this operator:

```
**Function** $T_1(P)$ - $P$ is a disjunctive logic program;
$Y := comp(P) \cup floop_0(P, \emptyset) \cup fLoop_1(P, \emptyset)$; $X := \emptyset$;
**while** $X \neq UP(Y)$ **do**
    $X := UP(Y)$; $Y := Y \cup floop_0(P, X) \cup fLoop_1(P, X)$;
**return** $X \cap Lit(P)$.
```

This is the function that we have implemented and used in our experiments. See Section 6 for details.

5 Related Work

Here we relate our work to the preprocessing procedure in DLV [11] and the well-founded semantics of disjunctive programs proposed by Wang and Zhou [2].

5.1 DLV Preprocessing Operator

We now show that $T_0(P)$ coincides with the least fixed point of the operator $\mathcal{W}_\mathcal{P}$ used in DLV for preprocessing a given disjunctive logic program. First, we show that the greatest unfounded set of a disjunctive logic program (if exists) can be computed from loop formulas of loops that have no external support rules.

Given a disjunctive logic program P and A a set of literals. The function $M(P, A)$, the least fixed point of the operator M_P^A defined as follows:

$loop_0^A(P, X) = \{ a \mid$ there is a loop L of P s.t. $a \in L$ and $R^-(L, A \cup \overline{X}) = \emptyset \}$,
$F_2^A(P, X) = \{ a \mid a \in Atoms(P)$ and for all $r \in P$, if $a \in head(r)$ then $\overline{A} \cap body(r) \neq \emptyset$, $X \cap body(r) \neq \emptyset$, or $(head(r) \setminus \{a\}) \cap A \neq \emptyset \}$,
$M_P^A(X) = X \cup loop_0^A(P, X) \cup F_2^A(P, X)$.

Theorem 2. *For any disjunctive logic program P and any $A \subseteq Lit(P)$ such that the greatest unfounded set of P w.r.t. A exists. $M(P, A) = GUS_P(A)$.*

From the above theorem, we can compute $GUS_P(A)$ by $M(P, A)$. We do not yet know any efficient way of computing $loop_0(P, A)$ for any possible A, but if A is restricted to be unfounded-free, then $GUS_P(A)$ always exists, and $loop_0^A(P, X) = \bigcup_{L \in ML_0(P, A \cup \overline{X}, Atoms(P))} L$, which can be computed in polynomial time. Furthermore, $F_2^A(P, X)$ can be computed in linear time. So, if A is unfounded-free, we have proposed a loop-oriented approach for computing $GUS_P(A)$ in polynomial time. Note that, different from other current approaches, $GUS_P(A)$ is computed directly here, avoiding the computation of the complement of it.

Now we introduce the $\mathcal{W}_\mathcal{P}$ operator proposed in [7].

$\mathcal{T}_\mathcal{P}(X) = \{ a \in Atoms(P) \mid$ there is a rule $r \in P$ such that $a \in head(r)$, $head(r) \setminus \{a\} \subseteq \overline{X}$, and $body(r) \subseteq X \}$,
$\mathcal{W}_\mathcal{P}(X) = \mathcal{T}_\mathcal{P}(X) \cup \overline{GUS_P(X)}$.

From Proposition 5.6 in [7], $\mathcal{W}_{\mathcal{P}}$ has a least fixed point, denoted $\mathcal{W}_{\mathcal{P}}^{\omega}(\emptyset)$, is the consequence of the program. $\mathcal{W}_{\mathcal{P}}^{\omega}(\emptyset)$ can also be computed efficiently, thus it is considered as a good start point to compute answer sets and is implemented in DLV.

In the following, a disjunctive logic program P is said to be *simplified* if for any $r \in P$, $head(r) \cap (body^+(r) \cup body^-(r)) = \emptyset$. Notice that any disjunctive logic program is strongly equivalent to a simplified program: if $head(r) \cap body^+(r) \neq \emptyset$, then $\{r\}$ is strongly equivalent to the empty set, thus can be safely deleted from any logic program, and if $head(r) \cap body^-(r) \neq \emptyset$, then $\{r\}$ is strongly equivalent to $\{r'\}$ such that $head(r') = head(r) \setminus body^-(r)$ and $body(r') = body(r)$ (cf. [12]).

The following theorem relates $T_0(P)$ and $\mathcal{W}_{\mathcal{P}}^{\omega}(\emptyset)$.

Theorem 3. *For any disjunctive logic program P, $\mathcal{W}_{\mathcal{P}}^{\omega}(\emptyset) \subseteq T_0(P)$. If P is simplified and without constraints, then $\mathcal{W}_{\mathcal{P}}^{\omega}(\emptyset) = T_0(P)$.*

Note that, [7] proved that, if P does not contain constraints, $\mathcal{W}_{\mathcal{P}}^{\omega}(\emptyset)$ coincides with the well-founded model of a normal logic program P' obtained by "shifting" some head atoms to the bodies of the rules. Thus, if P is simplified, then $T_0(P)$ coincides with the well-founded model of P' as well.

Given a disjunctive logic program P, we denote by $sh(P)$ the normal program obtained from P by substituting every rule of form (1) by the k rules

$$a_i \leftarrow a_{k+1}, \ldots, a_m, not\, a_{m+1}, \ldots, not\, a_n, not\, a_1, \ldots, not\, a_{i-1}, not\, a_{i+1}, \ldots, not\, a_k. \qquad (1 \leq i \leq k)$$

It is worth to note that, $fLoop_1(sh(P), A)$ may be not a consequence of a disjunctive logic program, even when A is unfounded-free for P or $sh(P)$.

Example 3. Consider the following logic program P:

$$\begin{array}{llll} d \leftarrow not\, e. & e \leftarrow not\, d. & a \vee c \leftarrow d. & a \vee b \leftarrow e. \\ a \leftarrow b. & b \leftarrow a. & \leftarrow not\, a. & \leftarrow not\, b. \end{array}$$

Clearly, $\{a, b, d\}$ and $\{a, b, e\}$ are the only two answer sets of P, $\{a, b, d\}$ is the only answer set of $sh(P)$. Let $A = \{a, b\}$, A is unfounded-free for P and $sh(P)$. $\neg a \vee d \in fLoop_1(sh(P), A)$ which is false for $\{a, b, e\}$, thus not a consequence of P.

A disjunctive logic program P is *head-cycle free*, if there does not exist a loop L and a rule r, s.t. $a, b \in L$ and $a, b \in head(r)$. If P is head-cycle free, then a set of atoms is an answer of P iff it is an answer set of $sh(P)$.

Proposition 7. *For any head-cycle free disjunctive logic program P and a set A of literals:*

$$floop_0(P, A) = ML_0(P, A, Atoms(P)) = ML_0(sh(P), A, Atoms(P)),$$

$$fLoop_1(P, A) = \bigcup_{\overline{A} \cap body(r) = \emptyset, L \in ML_0(sh(P \setminus \{r\}), A, Atoms(P))} \{\neg a \vee l \mid a \in L, \; l \in body(r) \cup \overline{head(r) \setminus L}\},$$

and $floop_0(P, A)$ implies that $floop_1(P, A)$ is equivalent to $fLoop_1(P, A)$.

5.2 Wang and Zhou's Well-Founded Semantics for Disjunctive Logic Programs

It is proved in [1] that T_0 computes the well-founded model when the given normal logic program is simplified and has no constraints. However, there have been several competing proposals for extending the well-founded semantics to disjunctive logic programs [7,13,2]. It is interesting that with a slight change of unit propagation, the procedure computes the same results as the well-founded semantics proposed in [2]. We now make this precise, first, we give one of the definitions of the well-founded semantics proposed by Wang and Zhou.

Given a disjunctive logic program P, a *positive* (*negative*) disjunction is a disjunction of atoms (negative literals) of P. A *pure disjunction* is either a positive one or a negative one. If A and $B = A \vee A'$ are two disjunctions, then we say A is a *subdisjunction* of B, denoted $A \subseteq B$. Let S be a set of pure disjunctions, we say $body(r)$ of $r \in P$ is *true* w.r.t. S, denoted $S \models body(r)$, if $body(r) \subseteq S$; $body(r)$ is *false* w.r.t. S, denoted $S \models \neg body(r)$ if either (1) the complement of a literal in $body(r)$ is in S or (2) there is a disjunction $a_1 \vee \cdots \vee a_n \in S$ such that $\{\, not\, a_1, \ldots, not\, a_n \,\} \subseteq body(r)$.

Now we extend the notion of unfounded set to under a set of pure disjunctions. Let S be a set of pure disjunctions of a disjunctive logic program P, a set of atoms X is an *unfounded set* for P w.r.t. S if, for each $a \in X$, $r \in P$ such that $a \in head(r)$, at least one of the following conditions holds:

1. the body of r is false w.r.t. S;
2. there is $x \in X$ such that $x \in body^+(r)$;
3. if $S \models body(r)$, then $S \models (head(r) - X)$. Here $(head(r) - X)$ is the disjunction obtained from $head(r)$ by removing all atoms in X, $S \models (head(r) - X)$ means there is a subdisjunction $A' \subseteq (head(r) - X)$ such that $A' \in S$.

Note that, if S is just a set of literals, then the above definition is equivalent to the definition in Preparation. If P has the greatest unfounded set w.r.t. S, we denote it by $\mathcal{U}_P(S)$. However, $\mathcal{U}_P(S)$ may be unfounded for some S.

Now we are ready to define the well-founded operator $\mathcal{W}'_P$ for any disjunctive logic program P and set of pure disjunctions S:

$$\mathcal{T}'_P(S) = \{\, A \text{ a pure disjunction} \mid \text{there is a rule } r \in P\text{: } A \vee a_1 \vee \cdots \vee a_k \leftarrow body(r), \text{ such that } S \models \underline{body(r)} \text{ and } not\, a_1, \ldots, not\, a_k \in S \,\},$$
$$\mathcal{W}'_P(S) = \mathcal{T}'_P(S) \cup \overline{\mathcal{U}_P(S)}.$$

Note that $\mathcal{T}'_P(S)$ is a set of positive disjunctions rather than a set of atoms.

From [2], the operator $\mathcal{W}'_P$ always has the least fixed point, denoted by $lfp(\mathcal{W}'_P)$, and the well-founded semantics $U\text{-}WFS$ is defined as $U\text{-}WFS(P) = lft(\mathcal{W}'_P)$.

Now we extend T_0 to treat about pure disjunctions. First, we extend the notion $floop_0(P, A)$ to under a set of pure disjunctions S.

A rule r is *active* under S w.r.t. a loop L, if $S \not\models \neg body(r)$ and $S \not\models (head(r) \setminus L)$. A rule r is an *external support rule* of L under S, if $r \in R^-(L)$ is active under S w.r.t. L. We use $R^-(L, S)$ to denote the set of external support rules of L under S.

Given a disjunctive logic program P and a set S of pure disjunctions, let

$$floop_0(P,S) = \{\, \neg a \mid a \in L \text{ for a loop } L \text{ of } P \text{ such that } R^-(L,S) = \emptyset \,\}.$$

Then $floop_0(P,S)$ is equivalent to the set of loop formulas of the loops that do not have any external support rules under S. Clearly, if S is just a set of literals, the above definition of $floop_0$ is equivalent to the definition in Section 3.

Now we extend unit propagation to return positive disjunctions. Given a set Γ of clauses, we use UP^* to denote the set of positive disjunctions returned by the extended unit propagation:

```
**Function** $UP^*(\Gamma)$
    **if** $(\emptyset \in \Gamma)$ **then return** $Lit$;
    $S := positive\_clause(\Gamma)$;
    **if** $S$ is inconsistent **then return** $Lit$;
    **if** $S \neq \emptyset$ **then return** $S \cup UP^*(assign(S,\Gamma))$ **else return** $\emptyset$;
```

where $positve\_clause(\Gamma)$ returns the union of all positive clauses (disjunctions) in Γ, let A is the union of all unit clauses in S, then $assign(S,\Gamma)$ is $\{\, c \mid$ for some $c' \in \Gamma, c' \cap A = \emptyset,$ and $c = c' \setminus \overline{A} \,\}$.

We use the new unit propagation in T_0, formally, the procedure computes the least fixed point of the following operator:

$$T_0^*(P,S) = UP^*(comp(P) \cup S \cup floop_0(P,S)) \cap DB(P),$$

where $DB(P)$ denotes the set of pure disjunctions formed by the literals in $Lit(P)$. We use $T_0^*(P)$ to denote such least fixed point.

The following theorem relates $U\text{-}WFS(P)$ and $T_0^*(P)$.

Theorem 4. *For any disjunctive logic program P, $U\text{-}WFS(P) \subseteq T_0^*(P)$. If P is simplified and without constraints, then $U\text{-}WFS(P) = T_0^*(P)$.*

6 Some Experimental Results

We have implemented a program that for any given disjunctive logic program P, it first computes $T_1(P)$, and then adds $\{ \leftarrow \bar{l} \mid l \in T_1(P) \}$ to P.

We tried our program on a number of benchmarks. First, for the disjunctive logic programs at the First Answer Set Programming System Competition, $T_1(P)$ does not return anything beyond the well-founded model of P. Next we tried the disjunctive encoding of the Hamiltonian Circuit (HC) problem,[5] and consider graphs with the same structure proposed in [1]. Specifically, we create some copies of a complete graph, and then randomly add some arcs to connect these copies into a strongly connected graph such that any HC for this graph must go through these special arcs. None of these "must in" arcs can be computed using the $\mathcal{W}_{\mathcal{P}}$ operator, except one of them, others can be computed from $T_1(P)$, thus adding the corresponding constraints to P should help ASP solvers in computing the answer sets.

[5] From the website of DLV,
http://www.dbai.tuwien.ac.at/proj/dlv/examples/hamcycle

Table 1. Run-time Data for cmodels and DLV

Problem	cmodels	cmodels$_{T_1}$	DLV	DLV$_{T_1}$	T_1	Problem	cmodels	cmodels$_{T_1}$	DLV	DLV$_{T_1}$	T_1
10x10.1	58.29	22.01	43.96	1.04	25.65	9x11.1	>1h	24.05	384.83	1.32	25.71
10x10.2	227.52	22.86	43.90	1.04	24.63	9x11.2	>1h	24.04	385.07	1.32	26.34
10x10.3	361.62	21.77	43.09	1.03	24.46	9x11.3	959.18	24.51	389.44	1.34	27.05
10x10.4	447.36	28.98	44.16	1.05	24.50	9x11.4	797.76	23.29	385.89	1.33	26.07
10x10.5	66.62	21.19	1.28	1.05	21.63	9x11.5	1276.01	21.64	391.15	1.33	26.00
10x10.6	344.12	21.20	43.97	1.03	24.92	9x11.6	1339.06	27.19	1.79	1.34	22.76
10x10.7	289.98	21.32	43.78	1.03	24.80	9x11.7	206.85	23.58	386.94	1.31	25.83
10x10.8	508.95	21.63	43.42	1.04	25.45	9x11.8	2803.17	22.89	389.94	1.34	25.85
10x10.9	246.04	20.86	44.11	1.03	24.87	9x11.9	1837.58	20.70	1.79	1.29	26.16
10x10.10	1481.17	20.78	44.24	1.05	25.45	9x11.10	>1h	21.76	385.01	1.34	27.01

Table 1 contains the running times for these programs.[6] In this table, MxN.K stands for a graph with M copies of the complete graph with N nodes: $C_1, ..., C_M$, and with exactly one arc from C_i to C_{i+1} and exactly one arc from C_{i+1} to C_i, for each $1 \leq i \leq M$ (C_{M+1} is defined to be C_1). The extension K stands for a specific way of adding these arcs. The numbers under "cmodels$_{T_1}$" and "DLV$_{T_1}$" refer to the run times (in seconds) of cmodels (version 3.77 [14]) and DLV (Oct 11 2007 [11]) when the results from $T_1(P)$ are added to the original program as constraints, and those under "T_1" are the run times of our program for computing $T_1(P)$. As can be seen, information from $T_1(P)$ makes cmodels and DLV run much faster when looking for an answer set. In addition to cmodels, we also tried claspD [15], which is very fast on these programs, on average it returned a solution in a few seconds.

7 Conclusion

We have extended the work of Chen *et al.* [1] on computing loops with at most one external support from normal logic programs to disjunctive logic programs. Our main results are that the set of loop formulas of loops that do not have any external support under an unfounded-free set of literals can be computed in polynomial time, and an iterative procedure using these loop formulas, program completion and unit propagation outputs the same set of consequences as computed by the preprocessing step of DLV, and is basically the same as Wang and Zhou's well-founded model semantics of disjunctive logic programs. However, the problem of computing loop formulas of loops with at most one external support is intractable. As a result, we consider a polynomial time algorithm for computing some of these loop formulas, and our experimental results show that this algorithm is sometimes useful for simplifying a disjunctive logic program beyond that can be done by the preprocessing step of DLV.

For future work, we plan to conduct more experiments with our algorithms and to consider more effective ways of using consequences of a logic program.

[6] Our experiments were done on an AMD Athlon(tm) 64 X2 Dual Core Processor 3600+ and 1GB RAM. The reported times are in CPU seconds as reported by Linux "/usr/bin/time" command.

Acknowledgments. This work has been supported in part by the Natural Science Foundations of China under grants 60745002, 60573009, and 60703095, the National High-Tech Project under grant 2008AA01Z150, and by the Hong Kong RGC CERG 616806. We thank Yisong Wang for useful discussions related to the topic of this paper.

References

1. Chen, X., Ji, J., Lin, F.: Computing loops with at most one external support rule. In: Proceedings of the 11th International Conference on Principles of Knowledge Representation and Reasoning, pp. 401–410 (2008)
2. Wang, K., Zhou, L.: Comparisons and computation of well-founded semantics for disjunctive logic programs. ACM Trans. Comput. Logic 6(2), 295–327 (2005)
3. Lifschitz, V., Tang, L., Turner, H.: Nested expressions in logic programs. Annals of Mathematics and Artificial Intelligence 25(3), 369–389 (1999)
4. Lee, J., Lifschitz, V.: Loop formulas for disjunctive logic programs. In: Proceedings of the 19th International Conference on Logic Programming, pp. 451–465 (2003)
5. Clark, K.L.: Negation as failure. In: Gallaire, H., Minker, J. (eds.) Logic and Databases, pp. 293–322. Plenum Press, New York (1978)
6. Van Gelder, A.: The alternating fixpoint of logic programs with negation. In: Proceedings of the eighth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems, pp. 1–10. ACM, New York (1989)
7. Leone, N., Rullo, P., Scarcello, F.: Disjunctive Stable Models: Unfounded Sets, Fixpoint Semantics, and Computation. Information and Computation 135(2), 69–112 (1997)
8. Lee, J.: A model-theoretic counterpart of loop formulas. In: Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence, pp. 503–508 (2005)
9. Anger, C., Gebser, M., Schaub, T.: Approaching the core of unfounded sets. In: Proceedings of the International Workshop on Nonmonotonic Reasoning (NMR 2006), pp. 58–66 (2006)
10. Lin, F., Zhao, Y.: ASSAT: computing answer sets of a logic program by SAT solvers. Artificial Intelligence 157(1-2), 115–137 (2004)
11. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The dlv system for knowledge representation and reasoning. ACM Transactions on Computational Logic 7(3), 499–562 (2006)
12. Lin, F., Chen, Y.: Discovering Classes of Strongly Equivalent Logic Programs. Journal of Artificial Intelligence Research 28, 431–451 (2007)
13. Brass, S., Dix, J.: Semantics of (disjunctive) logic programs based on partial evaluation. The Journal of Logic Programming 40(1), 1–46 (1999)
14. Giunchiglia, E., Lierler, Y., Maratea, M.: Answer set programming based on propositional satisfiability. J. Autom. Reasoning 36(4), 345–377 (2006)
15. Drescher, C., Gebser, M., Grote, T., Kaufmann, B., König, A., Ostrowski, M., Schaub, T.: Conflict-driven disjunctive answer set solving. In: Proceedings of the 11th International Conference on Principles of Knowledge Representation and Reasoning, pp. 422–432 (2008)

Modular Nonmonotonic Logic Programming Revisited*

Minh Dao-Tran, Thomas Eiter, Michael Fink, and Thomas Krennwallner

Institut für Informationssysteme, Technische Universität Wien
Favoritenstraße 9-11, A-1040 Vienna, Austria
{dao,eiter,fink,tkren}@kr.tuwien.ac.at

Abstract. Recently, enabling modularity aspects in Answer Set Programming (ASP) has gained increasing interest to ease the composition of program parts to an overall program. In this paper, we focus on modular nonmonotonic logic programs (MLP) under the answer set semantics, whose modules may have contextually dependent input provided by other modules. Moreover, (mutually) recursive module calls are allowed. We define a model-theoretic semantics for this extended setting, show that many desired properties of ordinary logic programming generalize to our modular ASP, and determine the computational complexity of the new formalism. We investigate the relationship of modular programs to disjunctive logic programs with well-defined input/output interface (DLP-functions) and show that they can be embedded into MLPs.

Keywords: Knowledge Representation, Answer Set Programming, Modular Logic Programming.

1 Introduction

In the recent years, there has been an increasing interest in studying modularity aspects of Answer Set Programming (ASP), in order to ease the composition of program parts to an overall program. Since the conception of Splitting Sets [1], which generalize stratification and proved to be a useful tool to decompose programs, a number of approaches to enhance ASP and LP in general with modularity have been made [2,3,4,5,6,7,8].

However, compared to the area of logic programming (LP) in general (see [4] for a historic account), the work on modular ASP is still less developed. As in general LP, there are two directions, namely Programming-in-the-large and Programming-in-the-small. In the former, compositional operators are provided for combining separate and independent modules based on standard semantics. This direction has been followed, e.g., with answer set programs with Gaifman-Shapiro-style module architecture [2,3]. Programming-in-the-small aims at enhancing ASP with abstraction and scoping mechanisms similar as in other programming paradigms. This direction has been more widely considered, and modular extensions of ASP based on generalized quantifiers [4], macros [5], and templates [6] have been proposed.

The two directions are quite divergent, as Programming-in-the-large requires to introduce new operators in the language. Modular ASP Programs [4] were an early attempt to narrow the gap between them a bit, using general quantifiers as a device

* This research has been supported by the Austrian Science Fund (FWF) projects P20841, the Vienna Science and Technology Fund (WWTF) project ICT08-020, and the EC ICT Integrated Project Ontorule (FP7 231875).

to access from one module P_1 another module P_2 using *module atoms* of the form $P_2[\boldsymbol{p}].q(X)$ (in slightly different syntax), where $\boldsymbol{p}$ is a list of predicates and q is a predicate; intuitively, the module atom evaluates to true for X if, on input of the values of the predicates in $\boldsymbol{p}$ to the module P_2, the atom $q(X)$ will be concluded by P_2 (under skeptical semantics). For a system $P_1[\boldsymbol{q_1}], \ldots, P_n[\boldsymbol{q_n}]$ of such modules, where $\boldsymbol{q_i}$ is a (list of) formal input predicates, answer sets have been defined using a generalization of the Gelfond-Lifschitz reduct. As it has been shown, the resulting framework is quite expressive, as it is EXPSPACE-complete in general.

However, the proposal in [4] has limitations and, due to the use of the Gelfond-Lifschitz reduct, suffers from similar anomalies as answer sets for other extensions of logic programs defined in this way. As for the former, an important restriction that was made in [4] is that calls of modules must be acyclic; that is, following the call chain, one may not return to the same call of a module. In fact, this condition was already imposed at the syntactic level, and does not allow the use of recursion in modules, which is a common and natural technique. Also other approaches exclude (mutually) recursive calls (e.g., disjunctive logic programs with a well-defined input/output interface (DLP-functions) exclude positive such calls [2]; see also Section 6).

Example 1. Consider the following recursive module $P[q/1]$, which determines whether a set has an even number of elements:

$$\begin{array}{rr} q'(X) \vee q'(Y) \leftarrow q(X), q(Y), X \neq Y. & \mathit{skip}(X) \leftarrow q(X), \text{not } q'(X). \\ \mathit{odd} \leftarrow \mathit{skip}(X), P[q'].\mathit{even}. & \mathit{even} \leftarrow \text{not } \mathit{odd}. \end{array}$$

Here, $q/1$ is a (formal) unary input predicate that stores the set. The first two rules in the top line effect, by stability of answer sets, that q becomes q' with one element randomly removed (for which skip is true). In the last line, the left rule determines recursively whether q stores an odd number of elements, while the right rule defines even as the complement of odd. Intuitively, if we call P with a predicate p for input, then even is computed true, which is expressed by $P[p].\mathit{even}$, if p stores an even number of elements. Note that P is recursive, and for empty input p it calls itself with the same input (one can easily rewrite this to mutual recursion between two modules for odd and even).

While a main motivation for the proposal in this paper is to allow for recursive calls of program modules with input, another objective is to provide a global semantics for a collection of modules. Comparatively, [4] was more concerned with defining local models of a single module, by importing conclusions of other modules rather than giving a model based semantics to a collection $P_1, \ldots, P_n$ of modules.

Concerning semantics, the use of the Gelfond-Lifschitz reduct effected that local models were in the same vein as Nash equilibria, viz., that a model is (locally) stable if assuming that all modules behave in the same way there is no need for the local program to switch to another model. Specifically, a program P_0 consisting of the clause $q \leftarrow P_1.p[q]$, where $P_1[q_1]$ consists of the single clause $p \leftarrow q_1$, has two answer sets, viz., $\emptyset$ and $\{q\}$. The reason is that q can be concluded in a self-stabilizing way from the call $P_1.p[q]$; however, arguably $\emptyset$ may be considered as the single answer set of P_0.

Such behavior can be excluded using alternative reducts, like the Faber-Leone-Pfeifer (FLP) reduct [9], which has been proposed in the context of ASP with aggregates to ensure that answer sets are minimal models. This reduct formed also the basis for defining the semantics of HEX-programs [10], which generalized the semantics of logic programs with generalized quantifiers to the HiLog setting; however, the setting has been module-centric like [4], and no global semantics for a collection of modules is evident. Motivated by these shortcomings, we reconsider modular ASP and make the following main contributions.

• We define a model theoretic semantics of a system $P_1[\boldsymbol{q_1}], \ldots, P_n[\boldsymbol{q_n}]$ of program modules, which are divided into one or multiple main modules P_i that have no input (i.e., $\boldsymbol{q_i}$ is void), and library modules which may have input (i.e., $\boldsymbol{q_i}$ can be non-void). Informally, the semantics assigns an answer set to each main module and module instance that is called by the program under a call-by-value mechanism as in [4]; the answer set must be reproducible from the rules along its recursive computation.

Example 2 (cont'd). In Example 1 above, an answer set for the module instance of $P[q]$, whose input q stores $S = \{c_1, \ldots, c_n\}$, would have q' storing $S_1 = S \setminus \{c_{\pi(1)}\}$ and call the instance of $P[q]$ with q storing S_1, whose answer set in turn stores $S_2 = S_1 \setminus \{c_{\pi(2)}\} = S \setminus \{c_{\pi(1)}, c_{\pi(2)}\}$ in q', etc., where π is any permutation of $\{1, \ldots, n\}$. The value of $even$ and odd in the answer sets of the instances is determined bottom up from the ground: for the instance of $P[q]$ where $q = \emptyset$, q' and $skip$ are void, and thus odd must be necessarily false; hence, $even$ is true. On the way back, $even$ and odd are complemented with their values at the next recursion level.

While a naive definition of the semantics is straightforward, a more difficult question is to delineate the *relevant* instances of modules for the computation. Intuitively, many (instances of) modules $P_i[\boldsymbol{q_i}]$ in a library might be completely irrelevant for determining the semantics of a particular collection of modules, but prevent the existence of a global semantics if locally, for some input value of $\boldsymbol{q_i}$, the instance has no answer set.

Example 3 (cont'd). Suppose in the module P in Example 1 there would also be a fact $r(a)$ and a rule $ok \leftarrow P'[r].nonempty$ where the module $P'[q/1]$ consists of the rules $nonempty \leftarrow \text{not } nonempty$ and $nonempty \leftarrow q(X)$. Then, an instance P' has an answer set precisely if its input is nonempty. Thus, the call $P'[r].nonempty$ in the rule will always lead to an answer set in which $nonempty$ is true, and hence we expect an answer set for the instance of P with input S. However, as P' has for empty input no answer set, there is no global answer set; intuitively, the instance of P' with empty input is irrelevant, and should not be considered.

To remedy this situation and keep the semantics simple, we use here minimal models as an approximation of answer sets in module instances that are outside of a *context* for which stability of models is strictly required; this context contains always the modules instances along the call graph of the program; the smaller the context, the more permissive is the semantics.

• We analyze semantic properties of the approach, and show that many of the desired properties of ordinary logic programs generalize to our modular ASP. This includes that the answer sets of a positive modular ASP are its minimal models; that Horn programs have a model intersection property, and thus a least model, which can be computed by least fixpoint iteration; that the latter can be extended to stratified programs, which have a canonical model modulo the relevant part.

• We characterize the computational complexity of the new formalism. Our modular ASP programs have the same complexity as ordinary ASP programs if the modules have no input, i.e., deciding answer set existence is Σ_2^p-complete in the propositional case and $\mathrm{NEXP}^{\mathrm{NP}}$-complete in the non-ground (Datalog) case. For programs with arbitrary inputs, the complexity is exponentially higher, viz. $\mathrm{NEXP}^{\mathrm{NP}}$-complete and $2\mathrm{NEXP}^{\mathrm{NP}}$-complete, respectively; note that EXPSPACE is believed to be strictly contained in $2\mathrm{NEXP}^{\mathrm{NP}}$. The picture is analogous for deciding membership of an atom in the least model of a Horn program, which is P-complete resp. EXP-complete without inputs and EXP-complete resp. 2EXP-complete with arbitrary inputs. However, if the inputs are naturally bounded, then the complexity is the same as in the case without inputs, and thus as in ordinary ASP.
• We analyze the relationship between our modular ASP programs and DLP-functions, which are one of the premier formalisms for combining ASP modules. As it turns out, DLP-functions can be very naturally embedded into our formalism, and vice versa a fragment of our modular ASP programs can be embedded into DLP functions. Since our approach admits mutual recursion of calls and also input to modules in terms of call by value, it can be viewed as a generalization of DLP-functions.

We believe that the approach presented in this paper contributes to modular ASP in which modules can be used in an unrestricted and natural way for problem solving, and looping recursion is handled by the very means of logic programming semantics.

2 Modular Nonmonotonic Logic Programs

In this section, we present our framework of modular ASP programs, and define first syntax and then semantics of such programs. We assume that the reader is familiar with basic notions of logic programming and the answer set semantics of nonmonotonic logic programs [11]. The syntax is based on disjunctive logic programs; our modular logic programs (MLPs) consist of modules as a way to structure logic programs. Moreover, such modules allow for input provided by other modules; it is safe to say that one module may call other modules and additionally provide input.

We pose no essential restriction on the rules, and modules may mutually call each other in a recursive way, and, on top of that, provide mutual input. The semantics we provide for MLPs caters for this situation and is thus not straight-forward. By the very notion of module input, it is apparent that modules must be instantiated before they can be "used." To this end, we delineate contexts of models that carry instantiations of modules and serve to define answer sets for modular programs. As noted in [4], answer sets of modular programs based on a Gelfond-Lifschitz-style reduct may be weaker than those of ordinary logic programs, we thus use the FLP-reduct in order to gain the desired property of minimality in answer sets.

Syntax of Modular Nonmonotonic Logic Programs. We consider programs in a function-free first-order (Datalog) setting (this restriction is not essential from a conceptual point of view, but convenient for the purposes of this work).

Let $\mathcal{V}$ be a vocabulary $\mathcal{C}$, $\mathcal{P}$, $\mathcal{X}$, and $\mathcal{M}$ of mutually disjoint sets whose elements are called *constants*, *predicate*, *variable*, and *module names*, respectively, where each $p \in \mathcal{P}$ has a fixed associated arity $n \geq 0$, and each module name in $\mathcal{M}$ has a fixed associated list $\boldsymbol{q} = q_1, \dots, q_k$ ($k \geq 0$) of predicated names $q_i \in \mathcal{P}$ (the formal input parameters). Unless stated otherwise, elements from $\mathcal{X}$ (resp., $\mathcal{C} \cup \mathcal{P}$) are denoted with first letter in upper case (resp., lower case).

Elements from $\mathcal{C} \cup \mathcal{X}$ are called *terms*. Ordinary atoms (simply atoms) are of the form $p(t_1, \dots, t_n)$, where $p \in \mathcal{P}$ and $t_1, \dots, t_n$ are terms; $n \geq 0$ is the *arity* of the atom. A *module atom* is of the form

$$P[p_1, \dots, p_k].o(t_1, \dots, t_l) \ , \tag{1}$$

where $p_1, \dots, p_k$ is a list of predicate names $p_i \in \mathcal{P}$, called *module input list*, such that p_i has the arity of the formal input parameter q_i, $o \in \mathcal{P}$ is a predicate name with arity l such that for the list of terms $t_1, \dots, t_l$, $o(t_1, \dots, t_l)$ is an ordinary atom, and $P \in \mathcal{M}$ is a module name.

Intuitively, a module atom provides a way for deciding the truth value of a ground atom $o(\boldsymbol{c})$ in a program P depending on the extension of a set of input predicates.

A *rule* r is of the form

$$\alpha_1 \vee \dots \vee \alpha_k \leftarrow \beta_1, \dots, \beta_m, \text{not}\, \beta_{m+1}, \dots, \text{not}\, \beta_n \ , \tag{2}$$

where $k \geq 1$, $m, n \geq 0$, $\alpha_1, \dots, \alpha_k$ are atoms, and $\beta_1, \dots, \beta_n$ are either atoms or module atoms. We define $H(r) = \{\alpha_1, \dots, \alpha_k\}$ and $B(r) = B^+(r) \cup B^-(r)$, where $B^+(r) = \{\beta_1, \dots, \beta_m\}$ and $B^-(r) = \{\beta_{m+1}, \dots, \beta_n\}$. If $B(r) = \emptyset$ and $H(r) \neq \emptyset$, then r is a (disjunctive) *fact*; r is *ordinary*, if it contains only ordinary atoms.

We now formally define the syntax of modules.

Definition 1 (module). *A* module *is a pair $m = (P[\boldsymbol{q}], R)$, where $P \in \mathcal{M}$ with associated formal input $\boldsymbol{q}$, and R is a finite set of rules. It is* ordinary*, if all rules in R are ordinary, and* ground*, if all rules in R are ground. A module m is either a* main module *or a* library module*; if it is a main module, then $|\boldsymbol{q}| = 0$.*

Recall that the formal input $\boldsymbol{q}$ is given by a list of predicate names $p_i \in \mathcal{P}$. We refer with $R(m)$ to the rule set of m. When clear from the context, we omit empty [] and () from (main) modules and module atoms. E.g., the module $P[q]$ in Example 1 is a library module; further examples are given below.

Based on modules, we define modular logic programs as follows.

Definition 2 (modular logic program). *A* modular logic program *(MLP)* $\mathbf{P}$ *is an n-tuple of modules*

$$(m_1, \dots, m_n) \ , n \geq 1, \tag{3}$$

consisting of at least one main module, where $\mathcal{M} = \{P_1, \dots, P_n\}$. We say that $\mathbf{P}$ is ground*, if each module is ground.*

Example 4 (cont'd). Suppose that we have besides a module $m_2 = (P[q], R_2)$, where R_2 is taken from the rules in Example 1, a further module $m_1 = (Q[], R_1)$, in which

$$R_1 = \left\{ \begin{array}{ll} s(a).\ s(b).\ s(c).\ s(d). & s_1(X) \vee s_2(X) \leftarrow s(X). \\ ok \leftarrow P[s_1].even, P[s_2].even. & ok \leftarrow \text{not}\ ok. \end{array} \right\} .$$

Informally, the disjunctive rule splits the predicate s into two predicates s_1 and s_2; the subsequent rules check that they both store sets of even cardinality. Formally, $\mathbf{P} = (m_1, m_2)$ forms the respective MLP; here, m_1 is the (single) main module.

Example 5. Take an MLP $\mathbf{P} = (m_1, m_2, m_3)$, where both $m_1 = (P_1[], \{a \leftarrow P_2.b.\})$, $m_2 = (P_2[], \{b \leftarrow P_1.a.\})$ are main modules, and $m_3 = (P_3[c], \{c \leftarrow \text{not } c.\})$ is a library module. Intuitively, m_1 and m_2 amount to the logic program $\{a \leftarrow b.\ b \leftarrow a.\}$, while m_3 is a simple constraint with formal input c.

Semantics of Modular Nonmonotonic Logic Programs. We now define the semantics of modular logic programs. It is defined in terms of Herbrand interpretations and grounding as customary in traditional logic programming and ASP.

The *Herbrand base* w.r.t. vocabulary $\mathcal{V}$, $HB_{\mathcal{V}}$, is the set of all possible ground ordinary and module atoms that can be built using $\mathcal{C}$, $\mathcal{P}$ and $\mathcal{M}$; if $\mathcal{V}$ is implicit from an MLP $\mathbf{P}$, it is the *Herbrand base of* $\mathbf{P}$ and denoted by $HB_{\mathbf{P}}$. The grounding of a rule r is the set $gr(r)$ of all ground instances of r w.r.t. $\mathcal{C}$; the grounding of rule set R is $gr(R) = \bigcup_{r \in R} gr(r)$, and the one of a module m, $gr(m)$, is defined by replacing the rules in $R(m)$ by $gr(R(m))$; the grounding of an MLP $\mathbf{P}$ is $gr(\mathbf{P})$, which is formed by grounding each module m_i of $\mathbf{P}$.

The semantics of an arbitrary MLP $\mathbf{P}$ is given in terms of $gr(\mathbf{P})$.

Let $S \subseteq HB_{\mathbf{P}}$ be any set of atoms. For any list of predicate names $\boldsymbol{p} = p_1, \ldots, p_k$ and $\boldsymbol{q} = q_1, \ldots, q_k$, we use the notation $S|_{\boldsymbol{p}} = \{p_i(\boldsymbol{c}) \in S \mid i \in \{1, \ldots, k\}\}$ and $S|_{\boldsymbol{p}}^{\boldsymbol{q}} = \{q_i(\boldsymbol{c}) \mid p_i(\boldsymbol{c}) \in S, i \in \{1, \ldots, k\}\}$.

Next, we define module instantiations. Therefore, we need to index a module with a particular, fixed set of input facts it receives, which is termed a value call.

Definition 3 (value call). *For a $P \in \mathcal{M}$ with associated formal input $\boldsymbol{q}$ we say that $P[S]$ is a* value call with input S*, where $S \subseteq HB_{\mathbf{P}}|_{\boldsymbol{q}}$. Let $VC(\mathbf{P})$ denote the set of all value calls $P[S]$ with input S such that $P \in \mathcal{M}$.*[1]

Instantiating an MLP $\mathbf{P}$ is more complex than instantiating $R(m)$ for every module m of $\mathbf{P}$, since all possible inputs for the modules need to be taken into account, yielding different sets of ground rules. Rule bases indexed by value calls account for this.

Definition 4 (rule base). *A* rule base *is an (indexed) tuple $\mathbf{R} = (R_{P[S]} \mid P[S] \in VC(\mathbf{P}))$ of sets of ground rules $R_{P[S]}$.*

Definition 5 (instantiation). *For a module $m_i = (P_i[\boldsymbol{q_i}], R_i)$ from $\mathbf{P}$, its* instantiation *with $S \subseteq HB_{\mathbf{P}}|_{\boldsymbol{q_i}}$, is $I_{\mathbf{P}}(P_i[S]) = R_i \cup S$. For an MLP $\mathbf{P}$, its* instantiation *is the rule base $I(\mathbf{P}) = (I_{\mathbf{P}}(P_i[S]) \mid P_i[S] \in VC(\mathbf{P}))$.*

Loosely speaking, a module instantiation is given by the rules of the module together with particular, additional input facts. Intuitively, rule bases collect all possible such instantiations with all possible inputs, and can be referenced by $VC(\mathbf{P})$.

We next define (Herbrand) interpretations and models of an MLP.

Definition 6 (interpretation). *An* interpretation $\mathbf{M}$ *of an MLP $\mathbf{P}$ is an (indexed) tuple $(M_i/S \mid P_i[S] \in VC(\mathbf{P}))$, where all $M_i/S \subseteq HB_{\mathbf{P}}$ contain only ordinary atoms.*

An interpretation provides an assignment for every module instance, and thus is likewise indexed, i.e., M_i/S is an interpretation of the module instance referenced by $P_i[S]$.

[1] Note that $VC(\mathbf{P})$ is also used as index set here.

Definition 7 (model). *An interpretation* $\mathbf{M}$ *of an MLP* $\mathbf{P}$ *is a* model *of*

- *a ground atom* $\alpha \in HB_{\mathbf{P}}$ *at* $P_i[S]$*, denoted* $\mathbf{M}, P_i[S] \models \alpha$*, if in case* α *is an ordinary atom,* $\alpha \in M_i/S$*, and if* $\alpha = P_k[\boldsymbol{p}].o(\boldsymbol{c})$ *is a module atom,* $o(\boldsymbol{c}) \in M_k/((M_i/S)|^{\boldsymbol{q_k}}_{\boldsymbol{p}})$*;*
- *a ground rule* r *at* $P_i[S]$ *(*$\mathbf{M}, P_i[S] \models r$*), if* $\mathbf{M}, P_i[S] \models H(r)$ *or* $\mathbf{M}, P_i[S] \not\models B(r)$*, where (i)* $\mathbf{M}, P_i[S] \models H(r)$*, if* $\mathbf{M}, P_i[S] \models \alpha$ *for some* $\alpha \in H(r)$*, and (ii)* $\mathbf{M}, P_i[S] \models B(r)$*, if* $\mathbf{M}, P_i[S] \models \alpha$ *for all* $\alpha \in B^+(r)$ *and* $\mathbf{M}, P_i[S] \not\models \alpha$ *for all* $\alpha \in B^-(r)$*;*
- *a set of ground rules* R *at* $P_i[S]$ *(*$\mathbf{M}, P_i[S] \models R$*) iff* $\mathbf{M}, P_i[S] \models r$ *for all* $r \in R$*;*
- *a ground rule base* $\mathbf{R}$ *(*$\mathbf{M} \models \mathbf{R}$*) iff* $\mathbf{M}, P_i[S] \models R_{P_i[S]}$ *for all* $P_i[S] \in VC(\mathbf{P})$.

Finally, $\mathbf{M}$ *is a* model *of an MLP* $\mathbf{P}$*, denoted* $\mathbf{M} \models \mathbf{P}$*, if* $\mathbf{M} \models I(\mathbf{P})$ *in case* $\mathbf{P}$ *is ground resp.* $\mathbf{M} \models gr(\mathbf{P})$*, if* $\mathbf{P}$ *is nonground. An MLP* $\mathbf{P}$ *is* satisfiable*, if it has a model.*

Intuitively, an interpretation $\mathbf{M}$ satisfies a ground module atom $P_k[\boldsymbol{p}].o(\boldsymbol{c})$ appearing in an instantiation $I_{\mathbf{P}}(P_i[S])$, if the ordinary atom $o(\boldsymbol{c})$ holds for the instantiation of the module m_k with the input which is given by the interpretation of $\boldsymbol{p}$ in M_i/S. On top of this, satisfaction of ordinary atoms, rules, etc., is straightforward.

Example 6. Consider $\mathbf{P}$ from Example 5, then $\mathbf{M} = (M_1/\emptyset, M_2/\emptyset, M_3/\emptyset, M_3/\{c\})$ is a model of $\mathbf{P}$, where $M_1/\emptyset = \{a\}$, $M_2/\emptyset = \{b\}$, and $M_3/\emptyset = M_3/\{c\} = \{c\}$. We have $\mathbf{M}, P_1[\emptyset] \models a$; $\mathbf{M}, P_2[\emptyset] \models b$; $\mathbf{M}, P_1[\emptyset] \models P_2.b$; $\mathbf{M}, P_2[\emptyset] \models P_1.a$; hence $\mathbf{M}, P_1[\emptyset] \models a \leftarrow P_2.b$; $\mathbf{M}, P_2[\emptyset] \models b \leftarrow P_1.a$. Moreover, $\mathbf{M}, P_3[\emptyset] \models c$; $\mathbf{M}, P_3[\emptyset] \models c \leftarrow \text{not } c$ (and similar for $\mathbf{M}$ at $P_3[\{c\}]$); thus $\mathbf{M}, P_1[\emptyset] \models I_{\mathbf{P}}(P_1[\emptyset])$, $\mathbf{M}, P_2[\emptyset] \models I_{\mathbf{P}}(P_2[\emptyset])$, $\mathbf{M}, P_3[\emptyset] \models I_{\mathbf{P}}(P_3[\emptyset])$, and $\mathbf{M}, P_3[\{c\}] \models I_{\mathbf{P}}(P_3[\{c\}])$; therefore $\mathbf{M} \models I(\mathbf{P})$, where $I(\mathbf{P}) = (I_{\mathbf{P}}(P_1[\emptyset]), I_{\mathbf{P}}(P_2[\emptyset]), I_{\mathbf{P}}(P_3[\emptyset]), I_{\mathbf{P}}(P_3[\{c\}]))$. Finally, $\mathbf{M} \models \mathbf{P}$.

We next proceed to define answer sets of an MLP $\mathbf{P}$. To this end, we need to compare models and single out minimal models. Furthermore, in order to focus on relevant modules, we introduce the formal notion of a call graph.

Definition 8 (minimal models). *For any interpretations* $\mathbf{M}$ *and* $\mathbf{M}'$ *of* $\mathbf{P}$*, we define that* $\mathbf{M} \leq \mathbf{M}'$*, iff for every* $P_i[S] \in VC(\mathbf{P})$ *it holds that* $M_i/S \subseteq M'_i/S$*, and* $\mathbf{M} < \mathbf{M}'$*, if both* $\mathbf{M} \neq \mathbf{M}'$ *and* $\mathbf{M} \leq \mathbf{M}'$*. A model* $\mathbf{M}$ *of* $\mathbf{P}$ *(resp., a rule base* $\mathbf{R}$*) is* minimal*, if* $\mathbf{P}$ *(resp.,* $\mathbf{R}$*) has no model* $\mathbf{M}'$ *such that* $\mathbf{M}' < \mathbf{M}$*. The set of all minimal models of* $\mathbf{P}$ *(resp.,* $\mathbf{R}$*) is denoted by* $MM(\mathbf{P})$ *(resp.,* $MM(\mathbf{R})$*).*

Definition 9 (call graph). *The* call graph of an MLP $\mathbf{P}$ *is a labeled digraph* $CG_{\mathbf{P}} = (V, E, l)$ *with vertex set* $V = VC(\mathbf{P})$ *and an edge* e *from* $P_i[S]$ *to* $P_k[T]$ *in* E *iff* $P_k[\boldsymbol{p}].o(\boldsymbol{t})$ *occurs in* $R(m_i)$*; furthermore,* e *is labeled with an input list* $\boldsymbol{p}$*, denoted* $l(e)$*. Given an interpretation* $\mathbf{M}$*, the* relevant call graph $CG_{\mathbf{P}}(\mathbf{M}) = (V', E')$ of $\mathbf{P}$ w.r.t. $\mathbf{M}$ *is the subgraph of* $CG_{\mathbf{P}}$ *where* E' *contains all edges from* $P_i[S]$ *to* $P_k[T]$ *of* $CG_{\mathbf{P}}$ *such that* $(M_i/S)|^{\boldsymbol{q_k}}_{l(e)} = T$*, and* V' *contains all* $P_i[S]$ *that are main module instantiations or induced by* E'*; any such* $P_i[S]$ *is called* relevant w.r.t. $\mathbf{M}$.

Example 7. Consider $\mathbf{P}$ and $I(\mathbf{P})$ from Example 6. The call graph of $\mathbf{P}$ is $CG_{\mathbf{P}} = (VC(\mathbf{P}), E, l)$, where $E = \{(P_1[\emptyset], P_2[\emptyset]), (P_2[\emptyset], P_1[\emptyset])\}$, and l maps each edge to the void input list. Both $P_1[\emptyset]$ and $P_2[\emptyset]$ are relevant, since they are main modules, while $P_3[\emptyset]$ and $P_3[\{c\}]$ are irrelevant (never called). Thus, we obtain that $CG_{\mathbf{P}}(\mathbf{M}) = (\{P_1[\emptyset], P_2[\emptyset]\}, E, l)$, for any interpretation $\mathbf{M}$ of $\mathbf{P}$.

We refer to the vertex and edge set of a graph G by $V(G)$ and $E(G)$, resp. For defining answer sets, we use a reduct of the instantiated program as customary in ASP. A suggestive way is to apply a traditional reduct to each module instance of $\mathbf{P}$; however, this is not fully satisfactory, as in practice $\mathbf{P}$ might contain module instantiations which have no answer sets for certain inputs, which compromises the existence of an answer set of $\mathbf{P}$. For this reason, we contextualize the notions of reduct and answer sets.

Definition 10 (context). *Let* $\mathbf{M}$ *be an interpretation of an MLP* $\mathbf{P}$. *A* context for $\mathbf{M}$ *is any set* $C \subseteq VC(\mathbf{P})$ *such that* $V(CG_{\mathbf{P}}(\mathbf{M})) \subseteq C$.

Then, the reduct of an instantiated program is built w.r.t. a given context.

Definition 11 (context-based reduct). *Let* $\mathbf{M}$ *be an interpretation of an MLP* $\mathbf{P}$ *and* C *be a context for* $\mathbf{M}$. *The* reduct of $\mathbf{P}$ at $P[S]$ w.r.t. $\mathbf{M}$ and C, *denoted* $f\mathbf{P}(P[S])^{\mathbf{M},C}$, *is the rule set* $I_{gr(\mathbf{P})}(P[S])$ *from which, if* $P[S] \in C$, *all rules* r *such that* $\mathbf{M}, P[S] \not\models B(r)$ *are removed. The* reduct of $\mathbf{P}$ w.r.t. $\mathbf{M}$ and C *is the rule base* $f\mathbf{P}^{\mathbf{M},C} = (f\mathbf{P}(P[S])^{\mathbf{M},C} \mid P[S] \in VC(\mathbf{P}))$.

That is, outside C the module instantiations of $\mathbf{P}$ resp. $gr(\mathbf{P})$ remain untouched, while inside C the FLP-reduct [9] is applied.

Example 8. Consider $\mathbf{P}$ and $\mathbf{M}$ from Example 6, and a context $C = \{P_1[\emptyset], P_2[\emptyset]\}$. The context-based reduct of $\mathbf{P}$ w.r.t. $\mathbf{M}$ and C is given by the rule base $f\mathbf{P}^{\mathbf{M},C} = (f\mathbf{P}(P_1[\emptyset])^{\mathbf{M},C}, f\mathbf{P}(P_2[\emptyset])^{\mathbf{M},C}, f\mathbf{P}(P_3[\emptyset])^{\mathbf{M},C}, f\mathbf{P}(P_3[\{c\}])^{\mathbf{M},C})$, which is equal to $I(gr(\mathbf{P}))$, i.e., $f\mathbf{P}(P_1[\emptyset])^{\mathbf{M},C} = \{a \leftarrow P_2.b.\}$, $f\mathbf{P}(P_2[\emptyset])^{\mathbf{M},C} = \{b \leftarrow P_1.a.\}$, $f\mathbf{P}(P_3[\emptyset])^{\mathbf{M},C} = \{c \leftarrow \text{not } c.\}$, and $f\mathbf{P}(P_3[\{c\}])^{\mathbf{M},C} = \{c; c \leftarrow \text{not } c.\}$.

Definition 12 (answer set). *Let* $\mathbf{M}$ *be an interpretation of a ground MLP* $\mathbf{P}$. *Then* $\mathbf{M}$ *is an* answer set of $\mathbf{P}$ w.r.t. a context C for $\mathbf{M}$, *if* $\mathbf{M}$ *is a minimal model of* $f\mathbf{P}^{\mathbf{M},C}$.

Note that C is a parameter that allows to select a degree of overall-stability for answer sets of $\mathbf{P}$. The extreme case $C = VC(\mathbf{P})$ requires that all module instances have answer sets. On the other end, the minimal context $C = V(CG_{\mathbf{P}}(\mathbf{M}))$ is the relevant call graph of $\mathbf{P}$; we consider this as the default context and omit C from notation.

Example 9. Consider $\mathbf{P}$ from Example 4. We have that $\mathbf{P}$ has answer sets of four different shapes, each of them having exactly two instances of s_1 and two instances of s_2 for the model $M_Q/\emptyset$ of instantiation $I_{\mathbf{P}}(Q[\emptyset])$. A particular answer set is the indexed tuple with the entries $(M_Q/\emptyset, M_P/\emptyset, M_P/\{q(a)\}, M_P/\{q(b)\}, M_P/\{q(c)\}, M_P/\{q(d)\}, M_P/\{q(a), q(c)\}, M_P/\{q(b), q(d)\}, \dots)$, where $M_Q/\emptyset = \{s_1(a), s_2(b), s_1(c), s_2(d), ok, s(a), s(b), s(c), s(d)\}$, $M_P/\emptyset = \{even\}$, all models for instantiations with singletons $M_P/\{q(a)\}$, $M_P/\{q(b)\}$, $M_P/\{q(c)\}$, $M_P/\{q(d)\}$ contain odd and the resp. $skip$'d element, and both $M_P/\{q(a), q(c)\}$ and $M_P/\{q(b), q(d)\}$ contain $even$.

Example 10. Consider $\mathbf{P}$ and $\mathbf{M}$ from Example 6. Let $\mathbf{M}_0 = (M_1^0/\emptyset, M_2^0/\emptyset, M_3^0/\emptyset, M_3^0/\{c\})$, such that $M_1^0/\emptyset = M_2^0/\emptyset = \emptyset$, $M_3^0/\emptyset = M_3^0/\{c\} = \{c\}$, be another interpretation for $\mathbf{P}$. One can verify that $\mathbf{M}_0$ is also a model of $\mathbf{P}$. Since we fixed the context C to $\{P_1[\emptyset], P_2[\emptyset]\}$, the reduct w.r.t. $\mathbf{M}_0$ is $f\mathbf{P}^{\mathbf{M}_0,C} = (f\mathbf{P}(P_1[\emptyset])^{\mathbf{M}_0,C}, f\mathbf{P}(P_2[\emptyset])^{\mathbf{M}_0,C}, f\mathbf{P}(P_3[\emptyset])^{\mathbf{M}_0,C}, f\mathbf{P}(P_3[\{c\}])^{\mathbf{M}_0,C}) = (\emptyset, \emptyset, I_{\mathbf{P}}(P_3[\emptyset]), I_{\mathbf{P}}(P_3[\{c\}]))$, and $f\mathbf{P}^{\mathbf{M},C}$ is as in Example 8. The minimal model of $f\mathbf{P}^{\mathbf{M}_0,C}$ is $\mathbf{M}_0$, hence it is an answer set of $\mathbf{P}$ w.r.t. C, whereas the minimal model of $f\mathbf{P}^{\mathbf{M},C}$ is also $\mathbf{M}_0$, i.e., $\mathbf{M}$ is not an answer set of $\mathbf{P}$ w.r.t. C.

3 Semantic Properties

We now consider some properties of modular logic programs. Obviously, they conservatively generalize ordinary logic programs.

Proposition 1. *Let R be an ordinary logic program. Then M is an answer set of R iff $\mathbf{M} = (M_1/\emptyset)$ with $M_1/\emptyset = M$ is an answer set of the MLP (m_1), where $m_1 = (P_1[], R)$ is a main module and P_1 is a module name.*

Some well-known properties from standard answer set programming carry over to the semantics of modular logic programs. This is of avail not only to encompass underlying intuitions, but also for characterizing computational aspects. Two straightforward consequences from the definition of FLP-reduct are the following.

Lemma 1. *If $\mathbf{M} \models f\mathbf{P}^{\mathbf{M},C}$ for some context C for $\mathbf{M}$, then $\mathbf{M} \models \mathbf{P}$.*

Lemma 2. *If $\mathbf{M} \models \mathbf{P}$, then $\mathbf{M} \models f\mathbf{P}^{\mathbf{M}',C}$ for any interpretation $\mathbf{M}'$ and context C.*

Consequently, we obtain that answer sets are minimal models of $\mathbf{P}$.

Proposition 2. *If $\mathbf{M}$ is an answer set of $\mathbf{P}$ w.r.t. context C, then $\mathbf{M} \in MM(\mathbf{P})$.*

Furthermore, the semantics is a proper refinement of a naive semantics that would require stability w.r.t. all possible module instantiations disregarding their relevance. This is a simple consequence of the following property.

Proposition 3. *If $\mathbf{M}$ is an answer set of $\mathbf{P}$ w.r.t. context $C \subseteq VC(\mathbf{P})$, then $\mathbf{M}$ is an answer set of $\mathbf{P}$ w.r.t. every context $C' \subseteq C$ for $\mathbf{M}$, i.e., $V(CG_{\mathbf{P}}(\mathbf{M})) \subseteq C' \subseteq C$.*

We next consider answer sets that, in a sense, face no inconsistency in the scope of instantiations that are relevant to them. Let $ord(\mathbf{P})$ denote the result of deleting from an MLP $\mathbf{P}$ all rules containing module atoms in $R(m)$ in all modules m of $\mathbf{P}$. We call an answer set $\mathbf{M}$ of $\mathbf{P}$ w.r.t. *C fully stable*, if $V(CG_{\mathbf{P}}(\mathbf{M}')) \subseteq C$ for all $\mathbf{M}' \leq \mathbf{M}$ such that $\mathbf{M}' \models ord(\mathbf{P})^{\mathbf{M},C}$. Then the following holds.

Proposition 4. *Every answer set of $\mathbf{P}$ w.r.t. $C = VC(\mathbf{P})$ is fully stable, and if $\mathbf{M}$ is an answer set of $\mathbf{P}$ w.r.t. C and fully stable w.r.t. $C' \subseteq C$, then $\mathbf{M}$ is fully stable w.r.t. C.*

Obviously, answer sets coincide with the naive semantics if $V(CG_{\mathbf{P}}(\mathbf{M})) = VC(\mathbf{P})$ for all interpretations $\mathbf{M}$ of $\mathbf{P}$, in particular, when all modules are main. Moreover, also for positive MLPs the semantics coincides with the naive semantics. Just like in ordinary logic programs, it behaves like the minimal model semantics in absence of negation.

Proposition 5. *Let $\mathbf{P}$ be positive. Then, the answer sets of $\mathbf{P}$ coincide with $MM(\mathbf{P})$.*

By monotonicity of all module instances, one can easily show that the models of a Horn MLP $\mathbf{P}$ are closed under a suitable notion of intersection. Given interpretations $\mathbf{M}$ and $\mathbf{N}$ of $\mathbf{P} = (m_1, \ldots, m_n)$, let their intersection be the interpretation denoted $\mathbf{M} \cap \mathbf{N}$ such that $(M \cap N)_i/S = \bigcap_{S' \supseteq S}(M_i/S' \cap N_i/S')$, for every $S \subseteq HB_{\mathbf{P}}|_{q_i}$ and $i = 1, \ldots, n$. Then:

Proposition 6. *Suppose* $\mathbf{M} \models \mathbf{P}$ *and* $\mathbf{N} \models \mathbf{P}$*, where* $\mathbf{P}$ *is Horn. Then* $\mathbf{M} \cap \mathbf{N} \models \mathbf{P}$.

As a consequence, a Horn MLP has a canonical answer set.

Corollary 1. *If* $\mathbf{P}$ *is Horn, then it has a unique answer set, which coincides with its least model.*

Like for ordinary programs, we can compute the answer set of a Horn MLP by means of a bottom up fixed-point computation.

Definition 13 **($T_{\mathbf{P}}$-operator).** *Given a Horn MLP* $\mathbf{P}$ *and an interpretation* $\mathbf{M}$ *of* $\mathbf{P}$*, we define the operator* $T_{\mathbf{P}}(\mathbf{M})$ *point-wise as follows:*

$$T_{\mathbf{P}}(M_i/S) = \{H(r) \mid r \in I_{\mathbf{P}}(P_i[S]),\ \mathbf{M}, P_i[S] \models B(r)\}.$$

Since the operator is continuous, it has a least fixed-point $\mathit{lfp}(\mathbf{P})$ that results, starting from the empty interpretation $\mathbf{M}_\emptyset$, i.e., where $M_i/S = \emptyset$ for every $P_i[S] \in \mathit{VC}(\mathbf{P})$ in ω steps, i.e., $\mathit{lfp}(\mathbf{P}) = T_{\mathbf{P}}{\uparrow}_\omega(\mathbf{M}_\emptyset)$. We obtain the following result.

Proposition 7. *For a Horn MLP* $\mathbf{P}$*,* $\mathit{lfp}(\mathbf{P})$ *is the unique answer set of* $\mathbf{P}$.

For normal MLPs, we generalize the notion of stratification as follows. Intuitively, the usual notion of the dependency graph of a program is extended by nodes $\mathcal{E}$ for the module atoms appearing in $\mathbf{P}$, which serve to take care of the dependencies between input to the module and module output. Furthermore, we assume that each predicate occurs in ordinary atoms of at most one module.

Let $\mathbf{P} = (m_1, \dots, m_n)$ be an MLP. The *dependency graph of* $\mathbf{P}$ is the following digraph $G_{\mathbf{P}} = (V, E)$. The vertex set V contains all $p \in \mathcal{P} \cup \mathcal{E}$, with p appearing somewhere in $\mathbf{P}$, and $\mathcal{E}$ is the set of module atoms in $\mathbf{P}$. The edge set E is as follows:

Let $r \in R(m_i)$. There is a $\star$-edge $p \rightarrow^\star q$ in $G_{\mathbf{P}}$, $\star \in \{+, -\}$, if either (i) $p(\boldsymbol{t_1}) \in H(r)$ and $q(\boldsymbol{t_2}) \in B^\star(r)$; (ii) $p(\boldsymbol{t_1}), q(\boldsymbol{t_2}) \in H(r)$ and $\star = +$; or (iii) $p(\boldsymbol{t_1}) \in H(r)$ and q is a module atom in $B^\star(r)$. Moreover, for $\alpha = P_j[\boldsymbol{p}].o(\boldsymbol{t}) \in B(r)$, the set E contains all edges $a \rightarrow^+ b$, where (iv) $a = \alpha$ and b appears in $\boldsymbol{q_j}$ of $P_j[\boldsymbol{q_j}]$; (v) $a = \alpha$ and $b = o$; or (vi) $a = q_\ell$ and $b = p_\ell$, where q_ℓ is in $\boldsymbol{q_j}$ of $P_j[\boldsymbol{q_j}]$ and p_ℓ is in $\boldsymbol{p}$.

Definition 14. *We say that an MLP* $\mathbf{P}$ *is* stratified *if no cycle in* $G_{\mathbf{P}}$ *has* $-$*-edges.*

As for ordinary logic programs, given a stratified MLP $\mathbf{P}$, there exists a labelling function l from $\mathit{HB}_{\mathbf{P}}$ to the nonnegative integers, such that $l(\alpha) \geq l(\beta)$ if $a \rightarrow^+ b$ in $G_{\mathbf{P}}$, and $l(\alpha) > l(\beta)$ if $a \rightarrow^- b$ in $G_{\mathbf{P}}$, where $\alpha = a(\boldsymbol{t})$, or $a \in \mathcal{E}$ and a unifies with α, respectively for β and b.

Let k be the maximal value assigned by a particular labelling function, and let $\mathit{Strat}_i = \{a \in \mathit{HB}_{\mathbf{P}} \mid l(a) = i\}$ for $0 \leq i \leq k$, then $\mathit{Strat}_0, \dots, \mathit{Strat}_k$ is a stratification, i.e., a partitioning of $\mathit{HB}_{\mathbf{P}}$.

Towards an iterated fixed-point computation of answer sets for stratified MLPs, we define the following operator.

Definition 15 **($T^L_{\mathbf{P}}$-operator).** *Given a normal MLP* $\mathbf{P}$*, a subset* L *of* $\mathit{HB}_{\mathbf{P}}$*, and an interpretation* $\mathbf{M}$ *of* $\mathbf{P}$*, we define the operator* $T^L_{\mathbf{P}}(\mathbf{M})$ *point-wise as follows:*

$$T^L_{\mathbf{P}}(M_i/S) = M_i/S \cup \{H(r) \mid r \in I_{\mathbf{P}}(P_i[S]),\ \mathbf{M}, P_i[S] \models B(r),\ B(r) \subseteq L\}.$$

By $T^L_{\mathbf{P}}\uparrow_\omega(\mathbf{M})$, we denote the application of $T^L_{\mathbf{P}}$ in ω steps, starting with $\mathbf{M}$. Furthermore, let $\mathbf{M}^0 = \mathbf{M}_\emptyset$ be the empty interpretation, i.e., where $M_i/S = \emptyset$ for every value call $P_i[S] \in VC(\mathbf{P})$, and let $L_i = \bigcup_{0\leq j\leq i} Strat_j$.We inductively define $\mathbf{M}^{i+1} = T^{L_{i+1}}_{\mathbf{P}}\uparrow_\omega(\mathbf{M}^i)$, for $0 \leq i < k$.

Proposition 8. *Let $\mathbf{P}$ be normal and stratified. Then $\mathbf{M}^k$ is an answer set of $\mathbf{P}$, for any stratification $Strat_0, \ldots, Strat_k$ of $HB_{\mathbf{P}}$.*

A further consequence of stratification is that the relevant call graph is unique.

Proposition 9. *Let $\mathbf{P}$ be normal and stratified. Then $V(CG_{\mathbf{P}}(\mathbf{M})) = V(CG_{\mathbf{P}}(\mathbf{M}^k))$, for any answer set $\mathbf{M}$ of $\mathbf{P}$ and any stratification $Strat_0, \ldots, Strat_k$ of $HB_{\mathbf{P}}$.*

Therefore, answer sets of stratified, normal MLPs coincide on relevant instances. The answer set obviously is unique if all value calls of $VC(\mathbf{P})$ are relevant, or if all irrelevant instances have a unique minimal model.

4 Computational Complexity

To begin with, let us restrict our attention to Horn MLPs. Considering the propositional case, if the modules $m_i = (P_i[\boldsymbol{q}_i], R_i)$ in $\mathbf{P}$ have no input (i.e., $\boldsymbol{q}_i$ is void), then $I(\mathbf{P})$ has polynomial size and $lfp(\mathbf{P})$ is computable in polynomial time. For arbitrary propositional $\mathbf{P}$ with no inputs, we can guess and verify an answer set $\mathbf{M}$ of $\mathbf{P}$ in polynomial time with an NP oracle. As MLPs (Proposition [1]) subsume ordinary logic programs, we thus obtain by known results (cf. [12]) the same complexity. With slight abuse of notation, for a ground atom α and an interpretation $\mathbf{M}$ of $\mathbf{P}$, we write $\alpha \in \mathbf{M}$ if $\alpha \in M_i/S$ for a given $P_i[S] \in VC(\mathbf{P})$.

Theorem 1. *Given a propositional MLP $\mathbf{P} = ((P_1[], R_1), \ldots, (P_n[], R_n))$, (i) if $\mathbf{P}$ is Horn, the unique answer set $\mathbf{M} = lfp(\mathbf{P})$ of $\mathbf{P}$ is computable in polynomial time and to decide whether $\alpha \in \mathbf{M}$ for a ground atom α is P-complete; (ii) to decide whether $\mathbf{P}$ has an answer set is Σ^p_2-complete.*

These results generalize to the case where the module inputs in $\mathbf{P}$ have bounded length, i.e., $|\boldsymbol{q}_i| \leq k$ for some constant k, as $I(\mathbf{P})$ and $\mathbf{M}$ have polynomial size. For unrestricted inputs, however, $I(\mathbf{P})$ and $\mathbf{M}$ are exponential and we get a blowup.

Theorem 2. *Given a propositional MLP $\mathbf{P}$ (i) if $\mathbf{P}$ is Horn, the unique answer set $\mathbf{M} = lfp(\mathbf{P})$ of $\mathbf{P}$ is computable in exponential time and to decide whether $\alpha \in \mathbf{M}$ for a ground atom α is EXP-complete; (ii) to decide whether $\mathbf{P}$ has an answer set is $\mathrm{NEXP}^{\mathrm{NP}}$-complete.*

The hardness parts can be shown e.g. by encodings of Turing machines, which adapt constructions in [12]. Superficially, one uses modules $P[\boldsymbol{c}, \boldsymbol{t}]$, where $\boldsymbol{c}$ amounts to a tape cell index and $\boldsymbol{t}$ to a time stamp during a computation; with $|\boldsymbol{c}| = |\boldsymbol{t}| = n$, 2^n cells and 2^n time stamps can be modeled. Further atoms store the cell contents, state of the machine, and the position of the read-write head. The transition function is encoded by rules with access to the contents of neighboring cells, which is realized by respective (recursive) module calls; neighboring cells and time stamps are computed using local rules.

Table 1. Complexity of MLPs ($\mathbf{P}$ is Horn in the first two columns, α is a ground atom)

MLP $\mathbf{P}$	Computing $\mathit{lfp}(\mathbf{P})$	Deciding $\alpha \in \mathit{lfp}(\mathbf{P})$	Answer set existence
prop. $\mathbf{P}$, empty inputs	polynomial time	P-complete	Σ_2^p-complete
prop. $\mathbf{P}$	exponential time	EXP-complete	$\mathrm{NEXP}^{\mathrm{NP}}$-complete
non-ground $\mathbf{P}$	double exponential time	2EXP-complete	$2\mathrm{NEXP}^{\mathrm{NP}}$-complete

In the Datalog setting, we get for MLPs a similar picture as for ordinary logic programs, where the complexity of Datalog programs is exponentially higher than the one of propositional programs. Intuitively, the process of grounding may introduce exponentially many ground atoms for an atom, which in turn may result in double exponentially many module instances; thus, $I(\mathbf{P})$ and interpretations $\mathbf{M}$ have double exponential size in general. Computing $\mathit{lfp}(\mathbf{P})$ for Horn MLPs $\mathbf{P}$ may thus take double exponential time, and a guess for an answer set has double exponential size. We get the following results.

Theorem 3. *Given a non-ground MLP $\mathbf{P}$, (i) if $\mathbf{P}$ is Horn, the unique answer set $\mathbf{M} = \mathit{lfp}(\mathbf{P})$ of $\mathbf{P}$ is computable in double exponential time and to decide whether $\alpha \in \mathbf{M}$ for a ground atom α is* 2EXP*-complete; (ii) to decide whether $\mathbf{P}$ has an answer set is* $2\mathrm{NEXP}^{\mathrm{NP}}$*-complete.*

The hardness parts an be shown by lifting the constructions for the propositional case. Here, n-ary predicates $p(X_1, \ldots, X_n)$ are used to store 2^n bits of a number, such that a range of 2^{2^n} tape cells and time stamps can be spanned via module inputs $\boldsymbol{q}$.

Finally, we note that the complexity drops by an exponential to the one of ordinary logic programs, if the arities of input predicates are bounded by a constant (as then $I(\mathbf{P})$ and $\mathbf{M}$ have single exponential size). Our results are compactly summarized in Table 1.

5 Relationship to DLP-Functions

DLP-functions [2] are a proposal for modular logic programs under answer set semantics in conformance with Programming-in-the-large. The approach creates a semantics for a sequence of modules by defining a suitable input-output interface, and allows combining compatible answer sets between joinable modules.

More specifically, a DLP-function has form $\Pi = \langle R, I, O, H \rangle$, where R is a set of propositional disjunctive rules and I, O, H are sets of propositional atoms defining input, output, and hidden atoms, respectively. An operator $\oplus$ forms a new DLP-function from two DLP-functions that respect hidden atoms of each other. In addition, if two such DLP-functions Π_1 and Π_2 are not mutually (positive) dependent, their join $\Pi_1 \sqcup \Pi_2$ is defined. Joinability allows negative loops between DLP-functions but not positive ones; one can use $\oplus$ to generate the join. On top of joinable DLP-functions, the Module Theorem is the basis for computing the answer sets of a sequence of DLP-functions by taking the union of mutually compatible answer sets of each member; hence joinable DLP-functions qualify for having a compositional semantics.

We now show a translation from DLP-functions to MLP modules, and briefly outline a translation from a fragment of MLPs without input to an equivalent sequence of DLP-functions. For space reasons, we must omit recalling the formal machinery of DLP-functions here, but stick to definitions of [2] as much as possible. To be in line with [2], we consider only the propositional case.

Translation from DLP-Functions to MLPs. We now define a translation ∇ which maps sequences of DLP-functions to MLPs. To this end, we map input atoms a appearing in bodies of rules in some DLP-function to module atoms of MLPs, whenever there is an output of another DLP-function which contains a. Other atoms remain unchanged. Then, we add further guessing rules to the modules; intuitively, they guess the truth value for input atoms which have not been fixed by some output.

Let $\Pi = (\Pi_1, \ldots, \Pi_n)$ be a sequence of DLP-functions, where Π_i is a DLP-function $\langle R_i, I_i, O_i, H_i \rangle$, and the join $\bigsqcup_{i=1}^{n} \Pi_i$ is defined. For a propositional atom a in $\mathrm{At}(R_i)$, if $a \in I_i$ and there exists another DLP-function Π_j in Π such that $a \in \mathrm{At_o}(\Pi_j)$, then $\nabla(a) = P_j.a$ (note that such a Π_j is unique due to the condition $\mathrm{At_o}(\Pi_k) \cap \mathrm{At_o}(\Pi_\ell) = \emptyset$ for every $k \neq \ell$); otherwise $\nabla(a) = a$.

Let r be a propositional rule in Π_i of the form (2). We create $\nabla(r)$ by replacing each β_i by $\nabla(\beta_i)$;[2] for Π_i, let $\nabla(\Pi_i) = (P_i, \nabla(R_i))$ where P_i is a module name and $\nabla(R_i) = \{\nabla(r) \mid r \in R_i\} \cup Q_i$, where $Q_i = \{a \vee \bar{a} \mid a \in \mathrm{At_i}(\Pi_i) \setminus \bigcup_{j \neq i} \mathrm{At_o}(\Pi_j)\}$ and all $\bar{a}$ are fresh propositional atoms. Finally $\nabla(\Pi) = (\nabla(\Pi_1), \ldots, \nabla(\Pi_n))$, where each $\nabla(\Pi_i)$ is a main module.

Example 11. Let $\Pi = (\Pi_1, \Pi_2)$ be a sequence of DLP-functions consisting of $\Pi_1 = \langle \{a \leftarrow \text{not } b\}, \{b\}, \{a\}, \emptyset \rangle$ and $\Pi_2 = \langle \{b \leftarrow \text{not } a\}, \{a\}, \{b\}, \emptyset \rangle$. The translation of Π to MLP is $\nabla(\Pi) = (\nabla(\Pi_1), \nabla(\Pi_2))$, where $\nabla(\Pi_1)$ and $\nabla(\Pi_2)$ are the main modules whose associative sets of rules are $\{a \leftarrow \text{not } P_2.b\}$ and $\{b \leftarrow \text{not } P_1.a\}$, resp. Here, both Π and $\nabla(\Pi)$ possess two answer sets: Π has $\{a\}$ and $\{b\}$, while $\nabla(\Pi)$ has $(\{a\}, \emptyset)$ and $(\emptyset, \{b\})$.

Now, let Π_1 be from above and $\Pi = (\Pi_1)$. In this case, $\nabla(\Pi) = (\nabla(\Pi_1))$, where $\nabla(\Pi_1) = (P_1, \nabla(R_1))$ and $\nabla(R_1) = \{a \leftarrow \text{not } b;\ b \vee \bar{b}\}$. Both Π and $\nabla(\Pi)$ have two answer sets; Π has $\{a\}$ and $\{b\}$, while $\nabla(\Pi)$ has $(\{a, \bar{b}\})$ and $(\{b\})$.

The following proposition shows that ∇ is correct.

Proposition 10. *Let $\Pi = (\Pi_1, \ldots, \Pi_n)$ be a sequence of DLP-functions whose join $\bigsqcup_{i=1}^{n} \Pi_i$ is defined. Then, the answer sets of $\nabla(\Pi)$ correspond 1-1 to those of Π.*

Translation from MLPs to DLP-Functions. Compared to DLP-functions, MLPs have a fine-grained input mechanism. DLP-functions import atoms from other DLP-functions by means of an explicit input/output interface; an atom, whose truth value originates from a different DLP-function, can be seen as a call-by-reference. To clarify, take an MLP with library modules $m_k = (P[q], R_k)$ and $m_\ell = (Q[p], R_\ell)$. Consider a module atom $Q[b].a$ appearing in R_k; we are confronted with two different types of input:

(1) m_ℓ retrieves input b from m_k explicitly in form of an additional fact p whenever b holds in some instantiation of $P[q]$, which can be seen as call-by-value, and
(2) m_k retrieves input from m_ℓ implicitly in form of a, which plays a similar role to call-by-reference input in DLP-functions.

Here, we restrict our attention to MLPs with input of type (2). By complexity arguments, translating MLPs with inputs of type (1) into sequences of DLP-functions is likely to cause an exponential blowup in general.

[2] Constraints are allowed in [2]; they can be emulated by adding *fail* (not *fail*) to the head (body) of $\nabla(r)$, where *fail* is a fresh propositional atom.

Given a propositional MLP $\mathbf{P} = (m_1, \ldots, m_n)$, where each $m_i = (P_i[], R_i)$ has no formal input parameter, we can do the translation by mapping in R_i each ordinary atom a to $\Delta(a) = a_{P_i}$, each module atom $P_j.b$ to $\Delta(P_j.b) = b_{P_j}$. For each module m_i, the input (output) atoms of the corresponding DLP-function are determined by applying Δ to module atoms occurring in R_i (resp., module atoms $P_i.a$ occurring in $\mathbf{P}$). Based on this idea, $\mathbf{P}$ can be translated into a sequence $\Delta(\mathbf{P})$ of DLP-functions where the composition operator $\oplus$ is defined. However, to have the join operator $\sqcup$ defined and thus answer sets of $\Delta(\mathbf{P})$, the modules in $\mathbf{P}$ must respect a condition akin to "*not mutually dependent*" [2], which is based on the sharing of strongly connected components in the positive dependency graph. On top of this condition, our translation gives a variant of the Module Theorem in [2]. Technical details and proofs are given in an accompanying technical report.

6 Related Work and Conclusion

In the ASP context, several modular logic programming formalisms have been proposed We already discussed the modular logic programs of [4] and DLP-functions [2].

Towards code reusability in ASP, [5] defines modules in terms of macros. On top of this, the authors define ensembles, which group modules comparable to the way classes keep their methods together in object-oriented programming languages, and an inheritance mechanism for ensembles. In a similar way but more focused on aggregates, [6] defines "template" predicates to quickly introduce new predefined constructs and to deal with compound data structures. The DLP^T language based on this notion was implemented on top of DLV. Both [5] and [6] have the restriction that no cycle is allowed between macros/templates.

A different approach is used in [13]. Here, the modules allow to import answer sets from other modules to compute the overall solution. However, this approach considers only modular ASP programs with acyclic dependency graph. Another system called RSig [14] allows to specify modules and provides an information hiding mechanism. Direct communication between modules was not addressed; instead, modules exchange information with a global state via import/export declarations. The semantics of such a system is given by a (polynomial) compilation into an ordinary ASP program.

Another formalism with multiple nonmonotonic logic programs is [15], targeting a Semantic Web environment. It allows to interlink logic programs that may refer to remote knowledge bases distributed on the Web. The authors propose a context-aware form of negation as failure to deal with the inherent incompleteness of data on the Web. The `MWeb` framework [16] is a further attempt to enhance the Semantic Web with scope and context for modular web rule bases. However, it is mainly concerned with support for hidden knowledge and the safe use of strong and weak negation, and modular rule bases are translated into ordinary logic programs, respecting different reasoning modes.

While we have presented the basic approach, several issues remain for further work. An interesting issue is to further analyze contexts and, e.g., to determine conditions for contexts that are fully stable, which desirably should be small. Some (less effective) conditions may be determined by syntactic analysis.

Another issue is extensions of MLP to richer classes of programs, including constructs like strong negation, constraints, external functions, nesting, etc. On the semantical side, we can imagine alternative ways of tolerating violations of stability outside the context. This could be done, e.g., by using partial FLP-reducts (where not all rules with false bodies are dropped, leading to a superset of the answer sets), or by genuine approximations. Variants of stratification and splitting sets would also be interesting.

On the computational side, a detailed complexity study of MLPs that considers various fragments is of interest, where in particular the interplay of major classes of ordinary logic programs with dependency information through module calls deserves attention; various notions similar as in [4] might be considered here. Furthermore, efficient methods and algorithms to compute answer sets of MLPs remain to be developed, as well as implementations. To this end, methods based on reductions to ordinary logic programs and extensions are under investigation.

References

1. Lifschitz, V., Turner, H.: Splitting a Logic Program. In: ICLP 1994, pp. 23–37. MIT Press, Cambridge (1994)
2. Janhunen, T., Oikarinen, E., Tompits, H., Woltran, S.: Modularity Aspects of Disjunctive Stable Models. In: Baral, C., Brewka, G., Schlipf, J. (eds.) LPNMR 2007. LNCS, vol. 4483, pp. 175–187. Springer, Heidelberg (2007)
3. Oikarinen, E., Janhunen, T.: Achieving compositionality of the stable model semantics for Smodels programs. Theory Pract. Log. Program. 8(5–6), 717–761 (2008)
4. Eiter, T., Gottlob, G., Veith, H.: Modular Logic Programming and Generalized Quantifiers. In: Fuhrbach, U., Dix, J., Nerode, A. (eds.) LPNMR 1997. LNCS, vol. 1265, pp. 290–309. Springer, Heidelberg (1997)
5. Baral, C., Dzifcak, J., Takahashi, H.: Macros, Macro calls and Use of Ensembles in Modular Answer Set Programming. In: Etalle, S., Truszczyński, M. (eds.) ICLP 2006. LNCS, vol. 4079, pp. 376–390. Springer, Heidelberg (2006)
6. Calimeri, F., Ianni, G.: Template programs for Disjunctive Logic Programming: An operational semantics. AI Commun. 19(3), 193–206 (2006)
7. Bugliesi, M., Lamma, E., Mello, P.: Modularity in Logic Programming. J. Logic Progr. 19/20, 443–502 (1994)
8. Brogi, A., Mancarella, P., Pedreschi, D., Turini, F.: Modular logic programming. ACM Trans. Program. Lang. Syst. 16(4), 1361–1398 (1994)
9. Faber, W., Leone, N., Pfeifer, G.: Recursive Aggregates in Disjunctive Logic Programs: Semantics and Complexity. In: Alferes, J.J., Leite, J. (eds.) JELIA 2004. LNCS, vol. 3229, pp. 200–212. Springer, Heidelberg (2004)
10. Eiter, T., Ianni, G., Schindlauer, R., Tompits, H.: Effective Integration of Declarative Rules with external Evaluations for Semantic Web Reasoning. In: Sure, Y., Domingue, J. (eds.) ESWC 2006. LNCS, vol. 4011, pp. 273–287. Springer, Heidelberg (2006)
11. Gelfond, M., Lifschitz, V.: Classical negation in logic programs and deductive databases. New Gener. Comput. 9, 365–385 (1991)
12. Dantsin, E., Eiter, T., Gottlob, G., Voronkov, A.: Complexity and Expressive Power of Logic Programming. ACM Comput. Surv. 33(3), 374–425 (2001)
13. Tari, L., Baral, C., Anwar, S.: A Language for Modular Answer Set Programming: Application to ACC Tournament Scheduling. In: ASP 2005. CEUR WS, pp. 277–293 (2005)
14. Balduccini, M.: Modules and Signature Declarations for A-Prolog: Progress Report. In: Workshop on Software Engineering for Answer Set Programming (SEA 2007) (2007)
15. Polleres, A., Feier, C., Harth, A.: Rules with Contextually Scoped Negation. In: Sure, Y., Domingue, J. (eds.) ESWC 2006. LNCS, vol. 4011, pp. 332–347. Springer, Heidelberg (2006)
16. Analyti, A., Antoniou, G., Damásio, C.V.: A principled framework for modular web rule bases and its semantics. In: KR 2008. AAAI Press, Menlo Park (2008)

Merging Logic Programs under Answer Set Semantics

James Delgrande[1], Torsten Schaub[2,⋆], Hans Tompits[3], and Stefan Woltran[3]

[1] Simon Fraser University, Burnaby, B.C., Canada V5A 1S6
jim@cs.sfu.ca
[2] Universität Potsdam, August-Bebel-Str. 89, D–14482 Potsdam, Germany
torsten@cs.uni-potsdam.de
[3] Technische Universität Wien, Favoritenstraße 9–11, A–1040 Vienna, Austria
tompits@kr.tuwien.ac.at,
woltran@dbai.tuwien.ac.at

Abstract. This paper considers a semantic approach for merging logic programs under answer set semantics. Given logic programs $P_1, \ldots, P_n$, the goal is to provide characterisations of the merging of these programs. Our formal techniques are based on notions of relative distance between the underlying SE models of the logic programs. Two approaches are examined. The first informally selects those models of the programs that vary the least from the models of the other programs. The second approach informally selects those models of a program P_0 that are closest to the models of programs $P_1, \ldots, P_n$. P_0 can be thought of as analogous to a set of database integrity constraints. We examine formal properties of these operators and give encodings for computing the mergings of a multiset of logic programs within the same logic programming framework. As a by-product, we provide a complexity analysis revealing that our operators do not increase the complexity of the base formalism.

Keywords: answer set programming, belief merging, strong equivalence.

1 Introduction

Answer set programming [1] is an appealing approach for representing problems in knowledge representation and reasoning: It has a conceptually simple theoretical foundation, while at the same time it has found application in a wide range of practical problems. As well, there are now efficient and well-studied implementations. However, as is the case with any large program or body of knowledge, a logic program is not a static object in general, but rather it will evolve and be subject to change, whether as a result of correcting information in the program, adding to the information already present, or in some other fashion modifying the knowledge represented in the program.

In the past, research on the evolution of logic programs mostly focussed on *updating logic programs* [2,3,4,5,6,7]. In such approaches, the issue was to characterise the answer sets of a sequence $\langle P_1, \ldots, P_n \rangle$ of programs, where for $j > i$, program P_j has higher priority, in some sense, over P_i. However, seemingly the nonmonotonic nature of extended logic programs makes the problem of belief change intrinsically harder compared to a monotonic setting, often leading to subtle effects. In previous work [8],

⋆ Affiliated with Simon Fraser University, Canada, and Griffith University, Australia.

we addressed this challenge by defining an approach for revising logic programs under answer set semantics based on the notion of an *SE model* [9]. The key point of this undertaking is that SE models provide a *monotonic semantic foundation* of answer set programs. More specifically, SE models derive from models in the logic of here-and-there, which is intermediate between classical logic and intuitionistic logic, representing the logical underpinning of strong equivalence [10]. Indeed, the latter notion can be seen as the logic programming analogue of ordinary equivalence in classical logic, in the sense that both equivalence notions adhere to a substitution principle. With our revision approach for logic programs based on SE models we thus phrased the problem of belief revision in logic programs in terms analogous to those of revision in classical logic. Additionally, the approach possesses appealing features as it satisfies all but one of the established postulates for belief revision [11].

In this paper, we employ these techniques to address the *merging* of logic programs. The problem of merging multiple, potentially conflicting bodies of information arises in different contexts. For example, an agent may receive reports from differing sources of knowledge, or from sets of sensors that need to be reconciled. As well, an increasingly common phenomenon is that collections of data may need to be combined into a coherent whole. In these cases, the problem is that of combining knowledge sets that may be jointly inconsistent in order to get a consistent set of merged beliefs.

In characterising the merging of logic programs, the central idea is that the SE models of the merged program are those that are in some sense "closest" to the SE models of the programs to be merged. However, as with merging knowledge bases expressed in classical logic, there is no single preferred notion of distance nor closeness, and consequently different approaches have been defined for combining sources of information. We introduce two merging operators for logic programs under answer set semantics. Both operators take an arbitrary (multi-)set of logic programs as argument. The first operator can be regarded an instance of *arbitration* [12]. Basically (SE) models are selected from among the SE models of the programs to be merged; in a sense this operator is a natural extension of our belief revision operator, presented in previous work [8]. The second merging operator can be regarded as an instance of Konieczny and Pino Pérez's merging operator [13]. Here, models of a designated program (representing information analogous to database integrity constraints) are selected that are closest to (or perhaps, informally, represent the best compromise among) the models of the programs to be merged.

2 Background

Answer Set Programming. A (*generalised*) *logic program*[1] (GLP) over an alphabet $\mathcal{A}$ is a finite set of rules of the form

$$a_1; \dots; a_m; \sim b_{m+1}; \dots; \sim b_n \leftarrow c_{n+1}, \dots, c_o, \sim d_{o+1}, \dots, \sim d_p, \tag{1}$$

where $a_i, b_j, c_k, d_l \in \mathcal{A}$ are *atoms*, for $1 \leq i \leq m \leq j \leq n \leq k \leq o \leq l \leq p$. Operators ';' and ',' express disjunctive and conjunctive connectives. A *default literal* is an atom a or its (default) negation $\sim a$. A rule r as in (1) is called a *fact* if $p = 1$,

[1] Such programs were first considered by Lifschitz and Woo [14].

normal if $n = 1$, *disjunctive* if $m = n$, and an *integrity constraint* if $n = 0$, yielding an empty disjunction denoted by $\bot$. Accordingly, a program is called *disjunctive*, or a *DLP*, if it consists of disjunctive rules only. Likewise, a program is *normal* if it contains normal rules only. We furthermore define $H(r) = \{a_1, \ldots, a_m, {\sim}b_{m+1}, \ldots, {\sim}b_n\}$ as the *head* of r and $B(r) = \{c_{n+1}, \ldots, c_o, {\sim}d_{o+1}, \ldots, {\sim}d_p\}$ as the *body* of r, for r as in (1). Moreover, given a set X of literals, $X^+ = \{a \in \mathcal{A} \mid a \in X\}$, $X^- = \{a \in \mathcal{A} \mid {\sim}a \in X\}$, and ${\sim}X = \{{\sim}a \mid a \in X \cap \mathcal{A}\}$. For simplicity, we sometimes use a set-based notation, expressing r as in (1) as $H(r)^+; {\sim}H(r)^- \leftarrow B(r)^+, {\sim}B(r)^-$.

In what follows, we restrict ourselves to a finite alphabet $\mathcal{A}$. An interpretation is represented by the subset of atoms in $\mathcal{A}$ that are true in the interpretation. A (*classical*) *model* of a program P is an interpretation in which all of the rules in P are true according to the standard definition of truth in propositional logic, and where default negation is treated as classical negation. By $\mathit{Mod}(P)$ we denote the set of all classical models of P. An *answer set* Y of a program P is a subset-minimal model of

$$\{H(r)^+ \leftarrow B(r)^+ \mid r \in P, H(r)^- \subseteq Y, B(r)^- \cap Y = \emptyset\}.$$

The set of all answer sets of a program P is denoted by $AS(P)$. For example, the program $P = \{a \leftarrow, \quad c; d \leftarrow a, {\sim}b\}$ has answer sets $AS(P) = \{\{a, c\}, \{a, d\}\}$.

As defined by Turner [9], an *SE interpretation* is a pair (X, Y) of interpretations such that $X \subseteq Y \subseteq \mathcal{A}$. An SE interpretation (X, Y) is an *SE model* of a program P if $Y \models P$ and $X \models P^Y$. The set of all SE models of a program P is denoted by $\mathit{SE}(P)$. Note that Y is an answer set of P iff $(Y, Y) \in \mathit{SE}(P)$ and no $(X, Y) \in \mathit{SE}(P)$ with $X \subset Y$ exists. Also, we have $(Y, Y) \in \mathit{SE}(P)$ iff $Y \in \mathit{Mod}(P)$.

A program P is *satisfiable* just if $\mathit{SE}(P) \neq \emptyset$. Two programs P and Q are *strongly equivalent*, symbolically $P \equiv_s Q$, iff $\mathit{SE}(P) = \mathit{SE}(Q)$. Alternatively, $P \equiv_s Q$ holds iff $AS(P \cup R) = AS(Q \cup R)$, for every program R [10]. We also write $P \models_s Q$ iff $\mathit{SE}(P) \subseteq \mathit{SE}(Q)$. For simplicity, we often drop set-notation within SE interpretations and simply write, e.g., (a, ab) instead of $(\{a\}, \{a, b\})$.

A set S of SE interpretations is *well-defined* if, for each $(X, Y) \in S$, also $(Y, Y) \in S$. A well-defined set S of SE interpretations is *complete* if, for each $(X, Y) \in S$, also $(X, Z) \in S$, for any $Z \supseteq Y$ with $(Z, Z) \in S$.

We have the following properties: (i) for each GLP P, $\mathit{SE}(P)$ is well-defined; and (ii) for each DLP P, $\mathit{SE}(P)$ is complete. Furthermore, for each well-defined set S of SE interpretations, there exists a GLP P such that $\mathit{SE}(P) = S$, and for each complete set S of SE interpretations, there exists a DLP P such that $\mathit{SE}(P) = S$. Programs meeting these conditions can be constructed thus [15,16]: In case S is a well-defined set of SE interpretations over a (finite) alphabet $\mathcal{A}$, define P by adding

1. the rule $r_Y : \bot \leftarrow Y, {\sim}(\mathcal{A} \setminus Y)$, for each $(Y, Y) \notin S$, and
2. the rule $r_{X,Y} : (Y \setminus X); {\sim}Y \leftarrow X, {\sim}(\mathcal{A} \setminus Y)$, for each $X \subseteq Y$ such that $(X, Y) \notin S$ and $(Y, Y) \in S$.

In case S is complete, define P by adding

1. the rule r_Y, for each $(Y, Y) \notin S$, as above, and
2. the rule $r'_{X,Y} : (Y \setminus X) \leftarrow X, {\sim}(\mathcal{A} \setminus Y)$, for each $X \subseteq Y$ such that $(X, Y) \notin S$ and $(Y, Y) \in S$.

We call the resulting programs *canonical*.

For illustration, consider $S = \{(p,p),(q,q),(p,pq),(q,pq),(pq,pq),(\emptyset,p)\}$ over $\mathcal{A} = \{p,q\}$.[2] Note that S is not complete. The canonical GLP is as follows:

$$\begin{aligned} r_{\emptyset}: &\qquad\quad \bot \leftarrow {\sim}p, {\sim}q; \\ r_{\emptyset,q}: &\qquad q; {\sim}q \leftarrow {\sim}p; \\ r_{\emptyset,pq}: &\ p; q; {\sim}p; {\sim}q \leftarrow . \end{aligned}$$

For obtaining a complete set, we have to add $(\emptyset, pq)$ to S. Then, the canonical DLP is as follows:

$$r_{\emptyset} : \bot \leftarrow {\sim}p, {\sim}q; \qquad r_{\emptyset,q} : q \leftarrow {\sim}p.$$

One feature of SE models is that they contain "more information" than answer sets, which makes them an appealing candidate for problems where programs are examined with respect to further extension (in fact, this is what strong equivalence is about). We illustrate this point with the following well-known example, involving programs $P = \{p; q \leftarrow\}$ and $Q = \{p \leftarrow {\sim}q,\ q \leftarrow {\sim}p\}$. Here, we have $AS(P) = AS(Q) = \{\{p\},\{q\}\}$. However, the SE models (we list them for $\mathcal{A} = \{p,q\}$) differ:

$$\begin{aligned} SE(P) &= \{(p,p),(q,q),(p,pq),(q,pq),(pq,pq)\}; \\ SE(Q) &= \{(p,p),(q,q),(p,pq),(q,pq),(pq,pq),(\emptyset,pq)\}. \end{aligned}$$

This is to be expected, since P and Q behave differently with respect to program extension (and thus are not strongly equivalent). Consider $R = \{p \leftarrow q, q \leftarrow p\}$. Then, $AS(P \cup R) = \{\{p,q\}\}$, while $AS(Q \cup R)$ has no answer set.

Belief Merging. This section reviews previous work in belief merging. We survey related work, first in logic programming and then in the belief merging literature.

With respect to merging logic programs, we have already mentioned updating logic programs, which can also be considered as *prioritised logic program merging*. With respect to combining logic programs, Baral et al. [17] describe an algorithm for combining a set of normal, stratified logic programs in which the union of the programs is also stratified. In their approach the combination is carried out so that a set of global integrity constraints, which is satisfied by individual programs, is also satisfied by the combination. Buccafurri and Gottlob [18] present an interesting approach whereby rules in a given program encode desires for a corresponding agent. A predicate *okay* indicates that an atom is acceptable to an agent. Answer sets of these *compromise logic programs* represent acceptable compromises between agents. While it is shown that the joint fixpoints of such logic programs can be computed as stable models, and complexity results are presented, the approach is not analysed from the standpoint of properties of merging. Sakama and Inoue [19] address what they call the *generous* and *rigorous coordination* of logic programs in which, given a pair of programs P_1 and P_2, a program Q is found whose answer sets are equal to the union of the answer sets of P_1 and P_2 in the first case, and their intersection in the second. As the authors note, this approach and its goals are distinct from program merging.

Earlier work on merging operators includes approaches by Baral et al. [20] and Revesz [21]. The former authors propose various theory merging operators based on

[2] We assume henceforth that the alphabet in an example consists of just the mentioned atoms.

the selection of maximum consistent subsets in the union of the belief bases. The latter proposes an "arbitration" operator (see below) that, intuitively, selects from among the models of the belief sets being merged. Lin and Mendelzon [22] examine *majority* merging, in which, if a plurality of knowledge bases hold ϕ to be true, then ϕ is true in the merging. Liberatore and Schaerf [12] address arbitration in general, while Konieczny and Pino Pérez [13] consider a general approach in which merging takes place with respect to a set of global constraints, or formulas that must hold in the merging. We examine these latter two approaches in detail below.

Konieczny, Lang, and Marquis [23] describe a very general framework in which a family of merging operators is parameterised by a distance between interpretations and aggregating functions. More or less concurrently, Meyer [24] proposed a general approach to formulating merging functions based on ordinal conditional functions [25]. Booth [26] also considers the problem of an agent merging information from different sources, via what is called *social contraction*. Last, much work has been carried out in merging possibilistic knowledge bases; we mention here, e.g., the method by Benferhat et al. [27].

We next describe the approaches by Liberatore and Schaerf [12] and by Konieczny and Pino Pérez [13], since we use the intuitions underlying these approaches as the basis for our merging technique. First, Liberatore and Schaerf [12] consider merging two belief bases built on the intuition that models of the merged bases should be taken from those of each belief base closest to the other. This is called an an *arbitration operator* (Konieczny and Pino Pérez [13] call it a *commutative revision operator*). They consider a propositional language over a finite set of atoms; consequently their merging operator can be expressed as a binary operator on formulas. The following postulates characterise this operator:

Definition 1. *$\diamond$ is an* arbitration operator (*or a* commutative revision operator) *if $\diamond$ satisfies the following postulates.*

$(LS1)$ $\vdash \alpha \diamond \beta \equiv \beta \diamond \alpha$.
$(LS2)$ $\vdash \alpha \wedge \beta \supset \alpha \diamond \beta$.
$(LS3)$ *If $\alpha \wedge \beta$ is satisfiable then $\vdash \alpha \diamond \beta \supset \alpha \wedge \beta$.*
$(LS4)$ *$\alpha \diamond \beta$ is unsatisfiable iff α is unsatisfiable and β is unsatisfiable.*
$(LS5)$ *If $\vdash \alpha_1 \equiv \alpha_2$ and $\vdash \beta_1 \equiv \beta_2$ then $\vdash \alpha_1 \diamond \beta_1 \equiv \alpha_2 \diamond \beta_2$.*

$$(LS6)\quad \alpha \diamond (\beta_1 \vee \beta_2) = \begin{cases} \alpha \diamond \beta_1 & \textit{or} \\ \alpha \diamond \beta_2 & \textit{or} \\ (\alpha \diamond \beta_1) \vee (\alpha \diamond \beta_2). \end{cases}$$

$(LS7)$ $\vdash (\alpha \diamond \beta) \supset (\alpha \vee \beta)$.
$(LS8)$ *If α is satisfiable then $\alpha \wedge (\alpha \diamond \beta)$ is satisfiable.*

The first postulate asserts that merging is commutative, while the next two assert that, for mutually consistent formulas, merging corresponds to their conjunction. $(LS5)$ ensures that the operator is independent of syntax, while $(LS6)$ provides a "factoring" postulate, analogous to a similar factoring result in (AGM-style) belief revision and contraction. Postulate $(LS7)$ can be taken as distinguishing $\diamond$ from other such operators; it asserts that the result of merging implies the disjunction of the original formulas. The last postulate informally constrains the result of merging so that each operator "contributes to" (i.e., is consistent with) the final result.

Next, Konieczny and Pino Peréz [13] consider the problem of merging possibly contradictory belief bases. To this end, they consider finite multisets of the form $\Psi = \{K_1, \ldots, K_n\}$. They assume that the belief sets K_i are consistent and finitely representable, and so representable by a formula. K^n is the multiset consisting of n copies of K. Following Konieczny and Pino Peréz [13], let $\Delta^\mu(\Psi)$ denote the result of merging the multi-set Ψ of belief bases given the entailment-based integrity constraint expressed by μ. The intent is that $\Delta^\mu(\Psi)$ is the belief base closest to the belief multiset Ψ. They provide the following set of postulates (multiset union is denoted by $\cup$):

Definition 2 ([13]). *Let Ψ be a multiset of sets of formulas, and ϕ, μ formulas (all possibly subscripted or primed). Then, Δ is an* IC merging operator *if it satisfies the following postulates.*

$(IC0)$ $\Delta^\mu(\Psi) \vdash \mu$.
$(IC1)$ *If* $\mu \nvdash \bot$ *then* $\Delta^\mu(\Psi) \nvdash \bot$.
$(IC2)$ *If* $\bigwedge \Psi \nvdash \neg\mu$ *then* $\Delta^\mu(\Psi) \equiv \bigwedge \Psi \wedge \mu$.
$(IC3)$ *If* $\Psi_1 \equiv \Psi_2$ *and* $\mu_1 \equiv \mu_2$ *then* $\Delta^{\mu_1}(\Psi_1) \equiv \Delta^{\mu_2}(\Psi_2)$.
$(IC4)$ *If* $\phi \vdash \mu$ *and* $\phi' \vdash \mu$ *then* $\Delta^\mu(\phi \cup \phi') \wedge \phi \nvdash \bot$ *implies* $\Delta^\mu(\phi \cup \phi') \wedge \phi' \nvdash \bot$.
$(IC5)$ $\Delta^\mu(\Psi_1) \wedge \Delta^\mu(\Psi_2) \vdash \Delta^\mu(\Psi_1 \cup \Psi_2)$.
$(IC6)$ *If* $\Delta^\mu(\Psi_1) \wedge \Delta^\mu(\Psi_2) \nvdash \bot$ *then* $\Delta^\mu(\Psi_1 \cup \Psi_2) \vdash \Delta^\mu(\Psi_1) \wedge \Delta^\mu(\Psi_2)$.
$(IC7)$ $\Delta^{\mu_1}(\Psi) \wedge \mu_2 \vdash \Delta^{\mu_1 \wedge \mu_2}(\Psi)$.
$(IC8)$ *If* $\Delta^{\mu_1}(\Psi) \wedge \mu_2 \nvdash \bot$ *then* $\Delta^{\mu_1 \wedge \mu_2}(\Psi) \vdash \Delta^{\mu_1}(\Psi) \wedge \mu_2$.

$(IC2)$ states that, when consistent, the result of merging is simply the conjunction of the belief bases and integrity constraints. $(IC4)$ asserts that when two belief bases disagree, merging does not give preference to one of them. $(IC5)$ states that a model of two mergings is in the union of their merging. With $(IC5)$ we get that if two mergings are consistent then their merging is implied by their conjunction. Note that merging operators are trivially commutative. $(IC7)$ and $(IC8)$ correspond to the extended AGM postulates $(K * 7)$ and $(K * 8)$ for revision, but with respect to the integrity constraints.

3 Merging Logic Programs

We denote (generalised) logic programs by $P_1, P_2, \ldots$, reserving P_0 for a program representing global constraints, as described later. For logic programs P_1, P_2, we define $P_1 \sqcap P_2$ to be a program with SE models equal to $SE(P_1) \cap SE(P_2)$ and $P_1 \sqcup P_2$ to be a program with SE models equal to $SE(P_1) \cup SE(P_2)$. By a *belief profile*, Ψ, we understand a sequence $\langle P_1, \ldots, P_n \rangle$ of (generalised) logic programs. For $\Psi = \langle P_1, \ldots, P_n \rangle$ we write $\sqcap\Psi$ for $P_1 \sqcap \cdots \sqcap P_n$. We write $\Psi_1 \circ \Psi_2$ for the (sequence) concatenation of belief profiles Ψ_1, Ψ_2; and for logic program P_0 and $\Psi = \langle P_1, \ldots, P_n \rangle$ we abuse notation by writing $\langle P_0, \Psi \rangle$ for $\langle P_0, P_1, \ldots, P_n \rangle$. A belief profile Ψ is *satisfiable* just if each component logic program is satisfiable. The set of SE models of Ψ is given by $SE(\Psi) = SE(P_1) \times \cdots \times SE(P_n)$. For $\overline{S} \in SE(\Psi)$ such that $\overline{S} = \langle S_1, \ldots, S_n \rangle$, we use $\overline{S}_i$ to denote the i^{th} component S_i of $\overline{S}$. Thus, $\overline{S}_i \in SE(P_i)$. Analogously, the set of classical propositional models of Ψ is given by $Mod(\Psi) = Mod(P_1) \times \cdots \times Mod(P_n)$; also we use $\overline{X}_i$ to denote the i^{th} component of $\overline{X} \in Mod(\Psi)$.

Let $\ominus$ denote the symmetric difference operator between sets, i.e., $X \ominus Y = (X \setminus Y) \cup (Y \setminus X)$ for every set X, Y. We extend $\ominus$ so that it can be used with SE interpretations as follows: For every pair (X_1, X_2), (Y_1, Y_2),

$$(X_1, X_2) \ominus (Y_1, Y_2) = (X_1 \ominus Y_1, X_2 \ominus Y_2).$$

Similarly, $(X_1, X_2) \subseteq (Y_1, Y_2)$ iff $X_1 \subseteq Y_1$ and $X_2 \subseteq Y_2$, and, moreover, $(X_1, X_2) \subset (Y_1, Y_2)$ iff $(X_1, X_2) \subseteq (Y_1, Y_2)$ and either $X_1 \subset Y_1$ or $X_2 \subset Y_2$.

3.1 Arbitration Merging

For the first approach to merging, called *arbitration*, we consider models of Ψ and select those models in which, in a global sense, the constituent models vary minimally. The result of arbitration is a logic program made up of SE models from each of these minimally-varying tuples. Note that, in particular, if a set of programs is jointly consistent, then there are models of Ψ in which all constituent SE models are the same. That is, the models that vary minimally are those $\overline{S} \in SE(\Psi)$ in which $\overline{S}_i = \overline{S}_j$ for every $1 \leq i, j \leq n$; and merging is the same as simply unioning the programs.

The first definition provides a notion of distance between models of Ψ, while the second then defines merging in terms of this distance.

Definition 3. *Let $\Psi = \langle P_1, \dots, P_n \rangle$ be a satisfiable belief profile and let $\overline{S}, \overline{T}$ be two SE models of Ψ (or two classical models of Ψ).*
Then, define $\overline{S} \leq_a \overline{T}$, if $\overline{S}_i \ominus \overline{S}_j \subseteq \overline{T}_i \ominus \overline{T}_j$ for every $1 \leq i < j \leq n$.

Clearly, $\leq_a$ is a partial pre-order. In what follows, let $Min_a(N)$ denote the set of all minimal elements of a set N of tuples relative to $\leq_a$, i.e.,

$$Min_a(N) = \{\overline{S} \in N \mid \overline{T} \leq_a \overline{S} \text{ implies } \overline{S} \leq_a \overline{T} \text{ for all } \overline{T} \in N\} .$$

Preparatory for our central definition to arbitration merging, we furthermore define, for a set N of n-tuples,

$$\cup N = \{S \mid S = \overline{S}_i \text{ for some } \overline{S} \in N \text{ and some } i \in \{1, \dots, n\}\}.$$

Definition 4. *Let Ψ be a belief profile. Then, the* arbitration merging*, or simply* arbitration*, of Ψ, is a logic program $\nabla(\Psi)$ such that*

$$SE(\nabla(\Psi)) = \{(X, Y) \mid Y \in \cup Min_a(Mod(\Psi)), X \subseteq Y, \\ \text{and if } X \subset Y \text{ then } (X, Y) \in \cup Min_a(SE(\Psi))\} ,$$

providing $\Psi = \langle P_1, \dots, P_n \rangle$ is satisfiable, otherwise, if P_i is unsatisfiable for some $1 \leq i \leq n$, define $\nabla(\Psi) = \nabla(\langle P_1, \dots, P_{i-1}, P_{i+1}, \dots, P_n \rangle)$.

For illustration, consider the belief profile

$$\langle P_1, P_2 \rangle = \langle \{p \leftarrow , u \leftarrow\}, \{\leftarrow p , v \leftarrow\} \rangle . \tag{2}$$

Given that $SE(P_1) = \{(pu, pu), (pu, puv), (puv, puv)\}$ and $SE(P_2) = \{(v, v), (v, uv), (uv, uv)\}$, we obtain nine SE models for $SE(\langle P_1, P_2 \rangle)$. Among them, we find

Table 1. Examples on Arbitration Merging

P_1	P_2	$SE(\nabla(\langle P_1, P_2\rangle))$	$\nabla(\langle P_1, P_2\rangle)$
$\{p \leftarrow\}$	$\{q \leftarrow\}$	$\{(pq, pq)\}$	$\{p \leftarrow , q \leftarrow\}$
$\{p \leftarrow\}$	$\{\leftarrow p\}$	$\{(p, p), (\emptyset, \emptyset)\}$	$\{p; {\sim}p \leftarrow\}$
$\{p \leftarrow {\sim}p\}$	$\{\leftarrow p\}$	$\{(\emptyset, p), (p, p), (\emptyset, \emptyset)\}$	$\{\}$
$\{p \leftarrow , q \leftarrow\}$	$\{\leftarrow p, q\}$	$\{(pq, pq), (p, p), (q, q)\}$	$\{p; q \leftarrow, \ p; {\sim}p \leftarrow, \ q; {\sim}q \leftarrow\}$
$\{\bot \leftarrow {\sim}p , \bot \leftarrow {\sim}q\}$	$\{\leftarrow p, q\}$	$\{S \in SE(\emptyset) \mid S \neq (\emptyset, \emptyset)\}$	$\{\bot \leftarrow {\sim}p, {\sim}q\}$
$\{\bot \leftarrow p , \bot \leftarrow q\}$	$\{p; q \leftarrow\}$	$\{(\emptyset, \emptyset), (p, p), (q, q)\}$	$\{\leftarrow p, q, \ p; {\sim}p \leftarrow, \ q; {\sim}q \leftarrow\}$

a unique $\leq_a$-minimal one, yielding $Min_a(SE(\langle P_1, P_2\rangle)) = \{\langle (puv, puv), (uv, uv)\rangle\}$. Similarly, $\langle P_1, P_2\rangle$ has a single $\leq_a$-minimal collection of pairs of classical models, viz. $Min_a(Mod(\langle P_1, P_2\rangle)) = \{\langle puv, uv\rangle\}$. Accordingly, we get

$$\begin{aligned} \cup Min_a(Mod(\langle P_1, P_2\rangle)) &= \{puv, uv\}, \\ \cup Min_a(SE(\langle P_1, P_2\rangle)) &= \{(puv, puv), (uv, uv)\}, \text{ and} \\ SE(\nabla((P_1, P_2))) &= \cup Min_a(SE(\langle P_1, P_2\rangle)) . \end{aligned}$$

We thus obtain the program $\nabla(\langle P_1, P_2\rangle) = \{p; {\sim}p \leftarrow , u \leftarrow , v \leftarrow\}$ as the resultant arbitration of P_1 and P_2.

For further illustration, consider the technical examples given in Table 1.

We note that merging normal programs often leads to disjunctive or generalised programs. Although plausible, this is also unavoidable because merging does not preserve the model intersection property of the reduced program satisfied by normal programs.

Moreover, we have the following general result.

Theorem 1. *Let $\Psi = \langle P_1, P_2\rangle$ be a belief profile, and define $P_1 \diamond P_2 = \nabla(\Psi)$. Then, $\diamond$ satisfies the following versions of the postulates of Definition 1.*

$(LS1')$ $P_1 \diamond P_2 \equiv_s P_2 \diamond P_1$.
$(LS2')$ $P_1 \sqcap P_2 \models_s P_1 \diamond P_2$.
$(LS3')$ *If $P_1 \sqcap P_2$ is satisfiable then $P_1 \diamond P_2 \models_s P_1 \sqcap P_2$.*
$(LS4')$ *$P_1 \diamond P_2$ is satisfiable iff P_1 is satisfiable and P_2 is satisfiable.*
$(LS5')$ *If $P_1 \equiv_s P_2$ and $P_1' \equiv_s P_2'$ then $P_1 \diamond P_2 \equiv_s P_1' \diamond P_2'$.*
$(LS7')$ $P_1 \diamond P_2 \models_s P_1 \sqcup P_2$.
$(LS8')$ *If P_1 is satisfiable then $P_1 \sqcap (P_1 \diamond P_2)$ is satisfiable.*

3.2 Basic Merging

For the second approach to merging, programs $P_1, \ldots, P_n$ are merged with a target logic program P_0 so that the SE models in the merging will be drawn from models of P_0. This operator will be referred to as the (*basic*) *merging* of $P_1, \ldots, P_n$ with respect to P_0. The information in P_0 *must* hold in the merging, and so can be taken as *necessarily* holding. Konieczny and Pino Pérez [13] call P_0 a set of *integrity constraints*, though this usage of the term differs from its usage in logic programs. Note that in the case where $SE(P_0)$ is the set of all SE models, the two approaches do not coincide, and that merging is generally a weaker operator than arbitration.

Definition 5. *Let $\Psi = \langle P_0, \ldots, P_n \rangle$ be a belief profile and let $\overline{S}, \overline{T}$ be two SE models of Ψ (or two classical models of Ψ).*

Then, define $\overline{S} \leq_b \overline{T}$, if $\overline{S}_0 \ominus \overline{S}_j \subseteq \overline{T}_0 \ominus \overline{T}_j$ for every $1 \leq j \leq n$.

As in the case of arbitration merging, $\leq_b$ is a partial pre-order. Accordingly, let $Min_b(N)$ be the set of all minimal elements of a set N of tuples relative to $\leq_b$. In extending our notation for referring to components of tuples, we furthermore define $N_0 = \{\overline{S}_0 \mid \overline{S} \in N\}$. We thus can state our definition for basic merging as follows:

Definition 6. *Let Ψ be a belief profile. Then, the* basic merging*, or simply* merging*, of Ψ, is a logic program $\Delta(\Psi)$ such that*

$$SE(\Delta(\Psi)) = \{(X, Y) \mid Y \in Min_b(Mod(\Psi))_0, X \subseteq Y, \text{ and if } X \subset Y \text{ then } (X, Y) \in Min_b(SE(\Psi))_0\} ,$$

providing $\Psi = \langle P_1, \ldots, P_n \rangle$ is satisfiable, otherwise, if P_i is unsatisfiable for some $1 \leq i \leq n$, define $\Delta(\Psi) = \Delta(\langle P_0, \ldots, P_{i-1}, P_{i+1}, \ldots, P_n \rangle)$.

Let us reconsider Programs P_1 and P_2 from (2) in the context of basic merging. To this end, we consider the belief profile $\langle \emptyset, \{p \leftarrow, u \leftarrow\}, \{\leftarrow p, v \leftarrow\}\rangle$. We are now faced with twenty-seven SE models for $SE(\langle \emptyset, P_1, P_2 \rangle)$. Among them, we get the following $\leq_b$-minimal SE models

$$Min_b(SE(\langle \emptyset, P_1, P_2 \rangle)) = \{\langle (uv, uv), (puv, puv), (uv, uv)\rangle, \langle (uv, puv), (puv, puv), (uv, uv)\rangle, \langle (puv, puv), (puv, puv), (uv, uv)\rangle\}$$

along with $Min_b(Mod(\langle \emptyset, P_1, P_2 \rangle)) = \{\langle uv, puv, uv\rangle, \langle puv, puv, uv\rangle\}$. We get:

$$\begin{aligned} Min_b(Mod(\langle \emptyset, P_1, P_2 \rangle))_0 &= \{puv, uv\}, \\ Min_b(SE(\langle \emptyset, P_1, P_2 \rangle))_0 &= \{(uv, uv), (uv, puv), (puv, puv)\}, \text{ and} \\ SE(\Delta(\langle \emptyset, P_1, P_2 \rangle)) &= Min_b(SE(\langle \emptyset, P_1, P_2 \rangle))_0 . \end{aligned}$$

While arbitration resulted in $\nabla(\langle P_1, P_2 \rangle) = \{p; {\sim}p \leftarrow, u \leftarrow, v \leftarrow\}$, the more conservative approach of basic merging yields $\Delta(\langle \emptyset, P_1, P_2 \rangle) = \{u \leftarrow, v \leftarrow\}$.

We have just seen that basic merging adds "intermediate" SE models, viz. (uv, puv), to the ones obtained in arbitration merging. This can also be observed on the examples given in Table 1, where every second merging is weakened by the addition of such intermediate SE models. This is made precise in Theorem 3 below. We summarise the results in Table 2 (but omit programs due to limited space). In fact, the programs $\Delta(\langle \emptyset, P_1, P_2 \rangle)$ are obtained from $\nabla(\langle P_1, P_2 \rangle)$ in Table 1 by simply dropping all rules of form $p; {\sim}p \leftarrow$ and $q; {\sim}q \leftarrow$, respectively.

The next example further illustrates the difference between arbitration an basic merging. Take $P_1 = \{p \leftarrow, q \leftarrow\}$ and $P_2 = \{{\sim}p \leftarrow, {\sim}q \leftarrow\}$. Then, we have that $SE(\nabla(\langle P_1, P_2 \rangle)) = \{(pq, pq), (\emptyset, \emptyset)\}$ and $SE(\Delta(\langle \emptyset, P_1, P_2 \rangle)) = SE(\emptyset)$. That is, in terms of programs, we obtain

$$\nabla(\langle P_1, P_2 \rangle) = \{p; {\sim}p \leftarrow, q; {\sim}q \leftarrow, \leftarrow p, {\sim}q, \leftarrow {\sim}p, q\} \text{ and } \Delta(\langle \emptyset, P_1, P_2 \rangle) = \emptyset .$$

Table 2. Examples on Basic Merging

P_1	P_2	$SE(\Delta(\langle\emptyset, P_1, P_2\rangle))$
$\{p \leftarrow\}$	$\{q \leftarrow\}$	$\{(pq, pq)\}$
$\{p \leftarrow\}$	$\{\leftarrow p\}$	$\{(p,p), (\emptyset,\emptyset)\} \cup \{(p,\emptyset)\}$
$\{p \leftarrow \sim p\}$	$\{\leftarrow p\}$	$\{(\emptyset,p), (p,p), (\emptyset,\emptyset)\}$
$\{p \leftarrow\, , q \leftarrow\}$	$\{\leftarrow p, q\}$	$\{(pq,pq), (p,p), (q,q)\} \cup \{(p,pq), (q,pq)\}$
$\{\bot \leftarrow \sim p\, , \bot \leftarrow \sim q\}$	$\{\leftarrow p, q\}$	$\{S \in SE(\emptyset) \mid S \neq (\emptyset,\emptyset)\}$
$\{\bot \leftarrow p\, , \bot \leftarrow q\}$	$\{p; q \leftarrow\}$	$\{(\emptyset,\emptyset), (p,p), (q,q)\} \cup \{(p,\emptyset), (q,\emptyset)\}$

Theorem 2. *Let Ψ be a belief profile, P_0 a program representing global constraints, and Δ as given in Definition 6. Then, Δ satisfies the following versions of the postulates of Definition 2:*

$(IC0')$ $\Delta(\langle P_0, \Psi\rangle) \models_s P_0$.
$(IC1')$ *If P_0 is satisfiable then $\Delta(\langle P_0, \Psi\rangle)$ is satisfiable.*
$(IC2')$ *If $\sqcap(\Psi)$ is satisfiable then $\Delta(\langle P_0, \Psi\rangle) \equiv_s P_0 \sqcap (\sqcap(\Psi))$.*
$(IC3')$ *If $P_0 \equiv_s P_0'$ and $\Psi \equiv_s \Psi'$ then $\Delta(\langle P_0, \Psi\rangle) \equiv_s \Delta(\langle P_0', \Psi'\rangle)$.*
$(IC4')$ *If $P_1 \models_s P_0$ and $P_2 \models_s P_0$ then:*
if $\Delta(\langle P_0, P_1, P_2\rangle) \sqcap P_1$ is satisfiable, then $\Delta(\langle P_0, P_1, P_2\rangle) \sqcap P_2$ is satisfiable.
$(IC5')$ $\Delta(\langle P_0, \Psi\rangle) \sqcap \Delta(\langle P_0, \Psi'\rangle) \models_s \Delta(\langle P_0, \Psi \circ \Psi'\rangle)$.
$(IC7')$ $\Delta(\langle P_0, \Psi\rangle) \sqcap P_0' \models_s \Delta(\langle P_0 \sqcap P_0', \Psi\rangle)$.
$(IC9')$ *Let Ψ' be a permutation of Ψ. Then, $\Delta(\langle P_0, \Psi\rangle) \equiv_s \Delta(\langle P_0, \Psi'\rangle)$.*

We also obtain that arbitration merging is stronger than (basic) merging in the case of tautologous constraints in P_0.

Theorem 3. *Let Ψ_a and $\Psi_b = \langle\emptyset, \Psi_a\rangle$ be belief profiles. Then, $\nabla(\Psi_a) \models_s \Delta(\Psi_b)$.*

As well, for belief profile $\Psi = \langle P_1, P_2\rangle$, we can express our merging operators in terms of the revision operator defined in previous work [8].

Theorem 4. *Let $\langle P_1, P_2\rangle$ be a belief profile.*

1. $\nabla(\langle P_1, P_2\rangle) = (P_1 * P_2) \sqcup (P_2 * P_1)$.
2. $\Delta(\langle P_1, P_2\rangle) = P_2 * P_1$.

Note that in the second part of the preceding result, P_1 is regarded as a set of constraints (usually with name P_0) according to our convention for basic merging.

4 Computational Issues

In this section, we provide encodings of arbitration and basic merging into logic programs. Since our encodings can be computed efficiently from a given belief profile, we are able to provide complexity results for decision problems typically associated to merging operators. We start, however, with the formal machinery required for the encodings.

In what follows, for basic merging, we consider the program representing integrity constraints to be part of a belief profile, and conventionally have it as the first element of the belief profile. Thus, we write $\Psi = \langle P_\alpha, \ldots, P_n \rangle$, and depending on the merging operator, we have $\alpha = 0$ or $\alpha = 1$. Moreover, we restrict ourselves to *satisfiable* belief profiles here. In fact, a generalisation of the subsequent encodings to the general case is possible but requires some further technical efforts, which we omit in order to provide a more succinct presentation of the basic ideas.

We let A be the set of all atoms occurring in Ψ and require mutually disjoint atoms

$$\{a_h^i, a_t^i, a_m^i, \hat{a}_h^i, \hat{a}_t^i, \hat{a}_m^i \mid a \in A, \alpha \leq i \leq n\}, \tag{3}$$

which are used as follows: atoms a_h^i, a_t^i $(1 \leq i \leq n)$ characterise $SE(P_i)$, and likewise a_m^i is used to characterise $Mod(P_i)$. In other words, an assignment to the atoms $\{a_h^i, a_t^i \mid a \in A, 1 \leq i \leq n\}$ represents a candidate for $\overline{S} \in SE(\Psi)$ and an assignment to the atoms $\{a_m^i \mid a \in A, 1 \leq i \leq n\}$ represents a candidate for $\overline{X} \in Mod(\Psi)$. The atoms $\hat{a}_h^i, \hat{a}_t^i, \hat{a}_m^i$ play analogous roles and are used to range over further SE models (resp., classical models) $\overline{T}$ of Ψ. In particular, we compare $\overline{T}$ to $\overline{S}$ (resp., to $\overline{X}$) to make the necessary checks for the merging operators. We give the formal details below.

To "guess" assignments, we need each atom a from the set (3) also in a "negated way", $\tilde{a}$. Moreover, we use further atoms $O = \{a_h^o, a_t^o \mid a \in A\}$ to carry our final result, $SE(\nabla(\Psi))$ (resp., $SE(\Delta(\Psi))$), and atoms H for particular technical programming issues, which we introduce as we go along. For the moment, we just have to assume that our encodings are given over an alphabet A^Ψ which contains each atom from the set (3) and its negation, the output atoms O and further atoms H.

We use sub- and superscripts also as renaming functions: Given a set $Y \subseteq A$ of atoms, $x \in \{h, t, m\}$, and an index i, Y_x^i denotes the set $\{y_x^i \mid y \in Y\}$, $\hat{Y}_x^i$ denotes the set $\{\hat{y}_x^i \mid y \in Y\}$, etc. Likewise for a rule r, r_x^i denotes the rule r after replacing each of its atom y by y_x^i, and $\hat{r}_x^i$ denotes the rule r after replacing each atom y by $\hat{y}_x^i$, etc.

We are now able to formally associate an interpretation $I \subseteq A^\Psi$ to several SE and classical interpretations over A as follows: Let $I \subseteq A^\Psi$ and i an index. Then,

$$\sigma^i(I) = \{(X, Y) \mid X, Y \subseteq A, X_h^i = I \cap A_h^i, Y_t^i = I \cap A_t^i\} \text{ and}$$
$$\pi^i(I) = \{X \mid X \subseteq A, X_m^i = I \cap A_m^i\}.$$

Moreover, let

$$\sigma(I) = \langle \sigma^\alpha(I), \ldots, \sigma^n(I) \rangle \text{ and } \pi(I) = \langle \pi^\alpha(I), \ldots, \pi^n(I) \rangle.$$

Likewise, for a set $\mathcal{I}$ of interpretations over A^Ψ, we define $\Sigma^i(\mathcal{I}) = \bigcup_{I \in \mathcal{I}} \sigma^i(I)$, $\Pi^i(\mathcal{I}) = \bigcup_{I \in \mathcal{I}} \pi^i(I)$, $\Sigma(\mathcal{I}) = \Sigma^\alpha(\mathcal{I}) \times \cdots \times \Sigma^n(\mathcal{I})$, and $\Pi(\mathcal{I}) = \Pi^\alpha(\mathcal{I}) \times \cdots \times \Pi^n(\mathcal{I})$.

We define the following module for an index i:

$$\begin{aligned} G[i] = \{ & a_x^i; \tilde{a}_x^i \leftarrow, \quad \bot \leftarrow a_x^i, \tilde{a}_x^i \mid a \in A, x \in \{h, t, m\}\} \cup \\ & \{\bot \leftarrow a_h^i, \tilde{a}_t^i \mid a \in A\} \cup \\ & \{\bot \leftarrow H^+(\tilde{r}_y^i), H^-(r_y^i), B^+(r_y^i), B^-(\tilde{r}_y^i) \mid r \in P_i, y \in \{t, m\}\} \cup \\ & \{\bot \leftarrow H^+(\tilde{r}_h^i), H^-(r_t^i), B^+(r_h^i), B^-(\tilde{r}_t^i) \mid r \in P_i\}. \end{aligned}$$

We note that $\Sigma^i(AS(G[i])) = SE(P_i)$ and $\Pi^i(AS(G[i])) = Mod(P_i)$. Consequently, $\Sigma(AS(G[\alpha] \cup \cdots \cup G[n])) = SE(\Psi)$ and $\Pi(AS(G[\alpha] \cup \cdots \cup G[n])) = Mod(\Psi)$.

The next module guesses the remaining atoms which are used to check minimality of the guess above. However, we use now a *spoiling technique* rather than constraints, to exclude (SE) interpretations which are not (SE) models of the respective program. The new atom z indicates whether we have to spoil. Moreover, this spoiling is also activated below where we compare the new guess with the guess from above. We define:

$$\begin{aligned} H[i] = \{&\hat{a}^i_x; \tilde{\hat{a}}^i_x \leftarrow, \quad z \leftarrow \hat{a}^i_x, \tilde{\hat{a}}^i_x, \quad \hat{a}^i_x \leftarrow z, \quad \tilde{\hat{a}}^i_x \leftarrow z \mid a \in A, x \in \{h,t,m\}\} \cup \\ \{&z \leftarrow \hat{a}^i_h, \tilde{\hat{a}}^i_t \mid a \in A\} \cup \\ \{&z \leftarrow H^+(\tilde{\hat{r}}^i_y), H^-(\hat{r}^i_y), B^+(\hat{r}^i_y), B^-(\tilde{\hat{r}}^i_y) \mid r \in P_i, y \in \{t,m\}\} \cup \\ \{&z \leftarrow H^+(\tilde{\hat{r}}^i_h), H^-(\hat{r}^i_t), B^+(\hat{r}^i_h), B^-(\tilde{\hat{r}}^i_t) \mid r \in P_i\}. \end{aligned}$$

For the moment, we can assume that the $H[i]$ modules act in the same way as the $G[i]$ modules. In particular, assuming that each P_i has at least one SE model, there exists a situation where z is not derived. Below, on the one hand, we derive z to indicate the outcome of several checks, and finally force z to be included in an answer set. However, for the moment, we can use the operators $\hat{\sigma}$, $\hat{\pi}$, $\hat{\Sigma}$, $\hat{\Pi}$ in an analogous way as above. Hence, for instance, given $I \subseteq A^{\Psi}$ and an index i, we have $\hat{\pi}^i(I) = \{X \mid X \subseteq A, \hat{X}^i_h = I \cap \hat{A}^i_h\}$, and so on.

Next, we want to compare different models, e.g., $\Sigma(I)$ with $\hat{\Sigma}(I)$, for some $I \subseteq A^{\Psi}$. By the considerations above, this allows us to compare two SE models $\overline{S}, \overline{T}$ of Ψ.

We require the following property:

Lemma 1. *For a belief profile $\Psi = \langle P_\alpha, \ldots, P_n \rangle$, we have $\overline{S} \in Min_a(SE(\Psi))$ iff*

(i) *for each $\overline{T} \in SE(\Psi)$, $\overline{S} \leq_a \overline{T}$, and*
(ii) *there exist $\alpha \leq i < j \leq n$ such that $\overline{S}_i \ominus \overline{S}_j \neq \overline{T}_i \ominus \overline{T}_j$.*

An analogous result holds for $Mod(\Psi)$ instead of $SE(\Psi)$.

Exploiting a somewhat dual method to this lemma, the following module derives, for given i, j,

- the atom z iff $\overline{S}_i \ominus \overline{S}_j \not\subseteq \overline{T}_i \ominus \overline{T}_j$, and
- the atom $z_{i,j}$ iff $\overline{S}_i \ominus \overline{S}_j = \overline{T}_i \ominus \overline{T}_j$.

For the latter, we require further new atoms $a^{i,j}_{x,\delta}$, for $x \in \{h,t,m\}$. Indeed, the compared models $\overline{S}$ and $\overline{T}$ are characterised via $I \subseteq A^{\Psi}$ by $\Sigma(I) = \overline{S}$ and $\hat{\Sigma}(I) = \overline{T}$, resp., $\Pi(I) = \overline{S}$ and $\hat{\Pi}(I) = \overline{T}$. We define

$$\begin{aligned} C[i,j] = \{&z \leftarrow a^i_x, \tilde{a}^j_x, \hat{a}^i_x, \hat{a}^j_x, & &z \leftarrow a^i_x, \tilde{a}^j_x, \tilde{\hat{a}}^i_x, \tilde{\hat{a}}^j_x, \\ &z \leftarrow \tilde{a}^i_x, a^j_x, \hat{a}^i_x, \hat{a}^j_x, & &z \leftarrow \tilde{a}^i_x, a^j_x, \tilde{\hat{a}}^i_x, \tilde{\hat{a}}^j_x, \\ &a^{i,j}_{x,\delta} \leftarrow a^i_x, \tilde{a}^j_x, \hat{a}^i_x, \tilde{\hat{a}}^j_x, & &a^{i,j}_{x,\delta} \leftarrow a^i_x, \tilde{a}^j_x, \tilde{\hat{a}}^i_x, \hat{a}^j_x, \\ &a^{i,j}_{x,\delta} \leftarrow \tilde{a}^i_x, a^j_x, \hat{a}^i_x, \tilde{\hat{a}}^j_x, & &a^{i,j}_{x,\delta} \leftarrow \tilde{a}^i_x, a^j_x, \tilde{\hat{a}}^i_x, \hat{a}^j_x, \\ &a^{i,j}_{x,\delta} \leftarrow a^i_x, a^j_x, \hat{a}^i_x, \hat{a}^j_x, & &a^{i,j}_{x,\delta} \leftarrow a^i_x, a^j_x, \tilde{\hat{a}}^i_x, \tilde{\hat{a}}^j_x, \\ &a^{i,j}_{x,\delta} \leftarrow \tilde{a}^i_x, \tilde{a}^j_x, \hat{a}^i_x, \hat{a}^j_x, & &a^{i,j}_{x,\delta} \leftarrow \tilde{a}^i_x, \tilde{a}^j_x, \tilde{\hat{a}}^i_x, \tilde{\hat{a}}^j_x \mid a \in A, x \in \{h,t,m\}\} \cup \\ \{&z_{i,j} \leftarrow A^{i,j}_{h,\delta} \cup A^{i,j}_{t,\delta}, & &z_{i,j} \leftarrow A^{i,j}_{m,\delta}\}. \end{aligned}$$

In other words, if we have guessed $\overline{S}$ and $\overline{T}$ in such a way that $\overline{S}_i \ominus \overline{S}_j \not\subseteq \overline{T}_i \ominus \overline{T}_j$, then $\overline{S} \leq_a \overline{T}$ cannot hold and we derive the spoiling atom z. In case $\overline{S}_i \ominus \overline{S}_j = \overline{T}_i \ominus \overline{T}_j$, we store this result by deriving an intermediate spoiling atom $z_{i,j}$. Below, we spoil if all relevant $z_{i,j}$'s have been derived.

Arbitration Merging. For a belief profile $\Psi = \langle P_1, \dots, P_n \rangle$, we put things together as follows, where Z is the set $\{z_{i,j} \mid 1 \leq i < j \leq n\}$:

$$\begin{aligned} E(\Psi) = \textstyle\bigcup_{i=1}^n (G[i] \cup H[i]) \cup \bigcup_{i=1}^n \bigcup_{j=i+1}^n C[i,j] \cup \\ \{z \leftarrow Z, \quad \bot \leftarrow {\sim}z\} \cup \{g_1 \vee \dots \vee g_{2n} \leftarrow\} \cup \\ \{a_t^o \leftarrow g_i, a_m^i, \quad a_h^o \leftarrow g_i, a_m^i, \\ a_t^o \leftarrow g_{n+i}, a_t^i, \quad a_h^o \leftarrow g_{n+i}, a_h^i \mid a \in A, 1 \leq i \leq n\} \cup \\ \{f_j \leftarrow g_{n+i}, a_m^j, \tilde{a}_t^i, \quad f_j \leftarrow g_{n+i}, \tilde{a}_m^j, a_t^i \mid 1 \leq i,j \leq n, a \in A\} \cup \\ \{\bot \leftarrow f_1, \dots, f_n\}. \end{aligned}$$

Roughly speaking, the guess via the g_i's selects from which P_i we now add a pair (X,Y) into $SE(\nabla(\Psi))$. More precisely, if a g_i is selected, with $1 \leq i \leq n$, we add $(\overline{X}_i, \overline{X}_i)$ for the currently guessed $\overline{X} \in Mod(\Psi)$. Otherwise, i.e., when g_{n+i} is selected $(1 \leq i \leq n)$, we add $\overline{S}_i = (X,Y)$, where $\overline{S} \in SE(\Psi)$ is the current guess, provided that Y matches *some* $\overline{X}_j$.

Let us now define, for a set $\mathcal{I}$ of interpretations over A^Ψ,

$$\Sigma^o = \bigcup_{I \in \mathcal{I}} \{(X,Y) \mid X,Y \subseteq A, X_h^o = I \cap A_h^o, Y_t^o = I \cap A_t^o\}.$$

We obtain the following result:

Theorem 5. $SE(\nabla(\Psi)) = \Sigma^o(AS(E(\Psi)))$.

Basic Merging. We now continue with the encoding for basic merging. We already have most ingredients at hand. In fact, for a belief profile $\Psi = \langle P_0, \dots, P_n \rangle$, we define

$$\begin{aligned} F(\Psi) = \textstyle\bigcup_{i=0}^n (G[i] \cup H[i]) \cup \bigcup_{i=1}^n C[0,i] \cup \\ \{z \leftarrow z_{0,1}, \dots, z_{0,n}, \quad \bot \leftarrow {\sim}z\} \cup \{g_0 \vee g_1 \leftarrow\} \cup \\ \{a_t^o \leftarrow g_0, a_m^0, \quad a_h^o \leftarrow g_0, a_m^0, \quad a_t^o \leftarrow g_1, a_t^0, \quad a_h^o \leftarrow g_1, a_h^0 \mid a \in A\} \cup \\ \{\bot \leftarrow g_1, a_m^0, \tilde{a}_t^0, \quad \bot \leftarrow g_1, \tilde{a}_m^0, a_t^0\}. \end{aligned}$$

$F(\Psi)$ follows the same ideas as used in $E(\Psi)$ but significantly simplifies due to the special role of P_0 in basic merging. Note that we require much less comparisons $C[0,i]$ here. As well, we only have to select classical and SE models of P_0 to become output atoms. Our result is thus as follows:

Theorem 6. $SE(\Delta(\Psi)) = \Sigma^o(AS(F(\Psi)))$.

Complexity. In our previous work [8], the following decision problem has been studied with respect to the revision operator $*$: Given GLPs P, Q, R, does $P * Q \models_s R$ hold? This problem was shown to be Π_2^P-complete. Accordingly, we give here results for the following problems:

Given a belief profile Ψ and a program R: (1) Does $\nabla(\Psi) \models_s R$ hold? (2) Does $\Delta(\Psi) \models_s R$ hold?

By Theorem 4, it can be shown that the hardness result for the revision problem also applies to the respective problems in terms of merging. On the other, hand Π_2^P-membership can be obtained by a slight extension of the above encodings such that these extensions possess an answer set iff the respective problem (1) or (2) does *not* hold. Since checking whether a program has at least one answer set is a problem on the second of layer of the polynomial hierarchy and our (extended) encodings are polynomial in the size of the encoded problems, the desired membership results follow.

Theorem 7. *Given a belief profile Ψ and a program R, deciding $\nabla(\Psi) \models_s R$ (resp., $\Delta(\Psi) \models_s R$) is Π_2^P-complete.*

5 Discussion

We have addressed the problem of merging logic programs under the answer set semantics. Unlike related work in updating logic programs, but similar to our work in logic program revision [8], our approach is based on a monotonic characterisation of logic programs, given in terms of the set of SE models of a sequence of programs. We defined and examined two operators for logic program merging, the first following intuitions from arbitration [12], the second being closer to IC merging [13]. Notably, since these merging operators are defined via a semantic characterisation, the results of merging are independent of the particular syntactic expression of a logic program. As well as giving properties of these operators, we also considered the complexity and an encoding scheme for both.

This work is original, given that it addresses merging in terms familiar to researchers in belief change. However, it applies these concepts in the context of logic programs. While we considered set-containment-based merging here, cardinality-based merging (which in fact would be closer to the specific operators proposed by Konieczny and Pino Pérez [13]) can also easily be defined.

Acknowledgements. The first author acknowledges support from a NSERC Discovery Grant. The second author was supported by the German Science Foundation (DFG) under grant SCHA 550/8-1. The third author was supported by the Austrian Science Fund (FWF) under project P18019.

References

1. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press, Cambridge (2003)
2. Zhang, Y., Foo, N.: Updating logic programs. In: ECAI 1998, pp. 403–407. IOS Press, Amsterdam (1998)
3. Alferes, J., Leite, J., Pereira, L., Przymusinska, H., Przymusinski, T.: Dynamic updates of non-monotonic knowledge bases. Journal of Logic Programming 45(1–3), 43–70 (2000)
4. Leite, J.: Evolving Knowledge Bases: Specification and Semantics. IOS Press, Amsterdam (2003)

5. Inoue, K., Sakama, C.: Updating extended logic programs through abduction. In: Gelfond, M., Leone, N., Pfeifer, G. (eds.) LPNMR 1999. LNCS, vol. 1730, pp. 147–161. Springer, Heidelberg (1999)
6. Eiter, T., Fink, M., Sabbatini, G., Tompits, H.: On properties of update sequences based on causal rejection. Theory and Practice of Logic Programming 2(6), 711–767 (2002)
7. Delgrande, J., Schaub, T., Tompits, H.: A preference-based framework for updating logic programs. In: Baral, C., Brewka, G., Schlipf, J. (eds.) LPNMR 2007. LNCS, vol. 4483, pp. 71–83. Springer, Heidelberg (2007)
8. Delgrande, J., Schaub, T., Tompits, H., Woltran, S.: Belief revision of logic programs under answer set semantics. In: KR 2008, pp. 411–421. AAAI Press, Menlo Park (2008)
9. Turner, H.: Strong equivalence made easy: Nested expressions and weight constraints. Theory and Practice of Logic Programming 3(4-5), 609–622 (2003)
10. Lifschitz, V., Pearce, D., Valverde, A.: Strongly equivalent logic programs. ACM Transactions on Computational Logic 2(4), 526–541 (2001)
11. Gärdenfors, P.: Knowledge in Flux. MIT Press, Cambridge (1988)
12. Liberatore, P., Schaerf, M.: Arbitration (or how to merge knowledge bases). IEEE Transactions on Knowledge and Data Engineering 10(1), 76–90 (1998)
13. Konieczny, S., Pino Pérez, R.: Merging information under constraints: A logical framework. Journal of Logic and Computation 12(5), 773–808 (2002)
14. Lifschitz, V., Woo, T.: Answer sets in general nonmonotonic reasoning (preliminary report). In: KR 1992, pp. 603–614. Morgan Kaufmann, San Francisco (1992)
15. Eiter, T., Tompits, H., Woltran, S.: On solution correspondences in answer set programming. In: IJCAI 2005, pp. 97–102. Professional Book Center (2005)
16. Cabalar, P., Ferraris, P.: Propositional theories are strongly equivalent to logic programs. Theory and Practice of Logic Programming 7(6), 745–759 (2007)
17. Baral, C., Kraus, S., Minker, J.: Combining multiple knowledge bases. IEEE Transactions on Knowledge and Data Engineering 3, 208–220 (1991)
18. Buccafurri, F., Gottlob, G.: Multiagent compromises, joint fixpoints, and stable models. In: Kakas, A.C., Sadri, F. (eds.) Computational Logic: Logic Programming and Beyond. LNCS, vol. 2407, pp. 561–585. Springer, Heidelberg (2002)
19. Sakama, C., Inoue, K.: Coordination in answer set programming. ACM Transactions on Computational Logic 9, 1–30 (2008)
20. Baral, C., Kraus, S., Minker, J., Subrahmanian, V.: Combining multiple knowledge bases consisting of first order theories. Computational Intelligence 8(1), 45–71 (1992)
21. Revesz, P.: On the semantics of theory change: Arbitration between old and new information. In: ACM Principles of Database Systems, pp. 71–82 (1993)
22. Lin, J., Mendelzon, A.: Knowledge base merging by majority. In: Dynamic Worlds: From the Frame Problem to Knowledge Management, pp. 195–218. Kluwer, Dordrecht (1999)
23. Konieczny, S., Lang, J., Marquis, P.: Distance-based merging: A general framework and some complexity results. In: KR 2002, pp. 97–108 (2002)
24. Meyer, T.: On the semantics of combination operations. Journal of Applied Nonclassical Logics 11(1-2), 59–84 (2001)
25. Spohn, W.: Ordinal conditional functions: A dynamic theory of epistemic states. In: Causation in Decision, Belief Change, and Statistics, pp. 105–134. Kluwer, Dordrecht (1988)
26. Booth, R.: Social contraction and belief negotiation. In: KR 2002, pp. 375–384 (2002)
27. Benferhat, S., Dubois, D., Kaci, S., Prade, H.: Possibilistic merging and distance-based fusion of propositional information. Annals of Mathematics and AI 34(1-3), 217–252 (2003)

Reducts of Propositional Theories, Satisfiability Relations, and Generalizations of Semantics of Logic Programs

Mirosław Truszczyński

Department of Computer Science, University of Kentucky, Lexington, KY 40506-0046, USA
mirek@cs.uky.edu

Abstract. Over the years, the stable-model semantics has gained a position of the correct (two-valued) interpretation of default negation in programs. However, for programs with aggregates (constraints), the stable-model semantics, in its broadly accepted generalization stemming from the work by Pearce, Ferraris and Lifschitz, has a competitor: the semantics proposed by Faber, Leone and Pfeifer, which seems to be *essentially* different. Our goal is to explain the relationship between the two semantics. Pearce, Ferraris and Lifschitz's extension of the stable-model semantics is best viewed in the setting of arbitrary propositional theories. We propose an extension of the Faber-Leone-Pfeifer semantics, or *FLP semantics*, for short, to the full propositional language, which reveals both common threads and differences between the FLP and stable-model semantics. We establish several properties of the FLP semantics. We apply a similar approach to define supported models for arbitrary propositional theories.

Keywords: Stable models, answer-set programming, logic here-and-there.

1 Introduction

The stable-model semantics, introduced by Gelfond and Lifschitz [1], is the foundation of answer-set programming [2,3,4], a paradigm for modeling and solving search problems. From its inception, developing theoretical underpinnings of the stable-model semantics has been a major research objective. In particular, a contribution by Pearce [5] explained the stable-model semantics in terms of models of theories in the logic *here-and-there* (*HT*, for short), introduced by Heyting [6].

Pearce's work had two important consequences. First, it resulted in a generalization of the stable-model semantics, originally limited to a restricted syntax of program rules, to *arbitrary* theories in the language of propositional logic. Second, it brought about the notion of *strong equivalence* of programs, fundamental to modular program development [7].

The original definition of stable models [1] was based on the *reduct* of a program with respect to a set of atoms. The characterization in terms of the logic HT makes no reference to reducts but employs a form of model minimization. Lifschitz and Ferraris [8,9] extended the notion of reduct to propositional theories, and developed the reduct-based definition of stable models equivalent to that provided by the logic HT.

The question motivating the present work is whether there are other generalizations of the stable-model semantics to the case of arbitrary logic theories. An indication that it might be so comes from the work by Faber, Leone and Pfeifer [10] on programs with aggregates. Aggregates, in the form of weight aggregates, were introduced to answer-set programming by Niemelä and Simons [11], who extended the stable-model semantics to that class of programs. Ferraris and Lifschitz [9] cast that generalization in terms of stable models of propositional theories. Stable models of programs with weight constraints are no longer guaranteed to be minimal models. From the perspective of the Ferraris and Lifschitz's result, it is not unexpected. Stable models of propositional theories in general do not have the minimal-model property.

However, as minimization is an important knowledge-representation principle, Faber et al. [10] sought an alternative semantics for programs with constraints, one that would have the minimal-model property. Naturally, they also wanted it to coincide with the original semantics on the class of programs without aggregates. They came up with a solution that satisfied both requirements by modifying the concept of the reduct.

In the setting with aggregates, the Faber-Leone-Pfeifer stable-model semantics, or *FLP* semantics, is different than the extension of the original stable-model semantics based on the logic HT (throughout the paper, whenever we speak about the stable-model semantics, we have this specific semantics in mind). Thus, the question we raised earlier is relevant.

In this paper, we have the following goals: (1) To extend the semantics of Faber et al. [10] to the language of propositional logic. We do so in two equivalent ways: by means of a generalization of the reduct introduced by Faber et al., as well as in terms of a certain satisfiability relation similar to the one that defines the logic HT. We show that the FLP semantics generalizes several properties of the stable-model semantics of logic programs and so, it can be regarded as its legitimate generalization, alongside with the extension based on the logic HT. We derive several additional properties of the FLP semantics, including a characterization of strong equivalence under that semantics, and a normal-form result. (2) To relate the FLP and stable-model semantics of propositional theories. We show that each can be expressed in each other in the sense that there are modular translations that do not use any auxiliary atoms and such that FLP-stable models of a theory are stable models of its image under the translation (and *vice versa*). (3) To apply a similar two-pronged approach, exploiting both some notion of reduct and a certain satisfiability relation, to the supported model semantics. We show that also supported models can be defined for arbitrary propositional theories. We generalize to propositional language some well-known properties of supported models, as well as the results connecting stable and supported models of programs.

2 Preliminaries

We consider the language of propositional logic determined by an infinite countable set At of atoms, and *boolean connectives* $\bot$, $\wedge$, $\vee$, and $\rightarrow$. A Backus-Naur Form expression $\varphi ::= \bot \mid A \mid (\varphi\wedge\varphi) \mid (\varphi\vee\varphi) \mid (\varphi \rightarrow \varphi)$, where $A \in At$, provides a concise definition of a formula. The parentheses are used only to disambiguate the order of binary operations. Whenever possible, we omit them. Generalizing the concept of the head of a program rule, we say that an occurrence of an atom is a *head* occurrence if it does not occur in the antecedent of any implication. Finally, when writing formulas, we often use the following shorthands:

$$\top = \bot \rightarrow \bot \quad \text{and} \quad \neg F = F \rightarrow \bot.$$

A set of formulas is a *theory*. In the case of all semantics we discuss here, there is no essential difference between *finite* theories and formulas. The former can be represented as the conjunctions of their elements. We distinguish between formulas and theories as we want to address the case of infinite theories, too.

In the paper, we consider several special types of formulas and theories. A *rule* is a formula

$$A_1 \wedge \ldots \wedge A_m \wedge \neg B_1 \wedge \ldots \wedge \neg B_n \rightarrow C_1 \vee \ldots \vee C_r \vee \neg D_1 \vee \ldots \vee \neg D_s, \qquad (1)$$

where A_i's, B_i's C_i's and D_i's are atoms. We call $A_1 \wedge \ldots \wedge A_m \wedge \neg B_1 \wedge \ldots \wedge \neg B_n$ and $C_1 \vee \ldots \vee C_r \vee \neg D_1 \vee \ldots \vee \neg D_s$ the *body* and the *head* of the rule, respectively. If $m = n = 0$, we represent the rule by its head. If $r = s = 0$, we write $\bot$ for the head of the rule. A *program* is a set of rules.

The stable-model semantics was defined first for *normal programs* (rule heads have exactly one atom and no negated atoms). It was later extended to *disjunctive programs* (rule heads have no negated atoms) [12], programs as understood here (that is, collections of rules as defined above) [13], and to *programs with nested expressions* [14]. Finally, the case of arbitrary theories was addressed by Pearce [5] and, later and in a different way, by Ferraris and Lifschitz [8,9]. These two approaches are equivalent. We will now discuss each of them, starting with the latter one.

For a formula F and a set of atoms Y, we define the *GL-reduct* of F with respect to Y, written as F^Y, by induction:

R1. $$\bot^Y = \bot$$

R2. If A is an atom: $$A^Y = \begin{cases} A & \text{if } Y \models A \\ \bot & \text{otherwise} \end{cases}$$

R3. For $\circ = \wedge$ and $\vee$: $$(G \circ H)^Y = \begin{cases} G^Y \circ H^Y & \text{if } Y \models G \circ H \\ \bot & \text{otherwise} \end{cases}$$

R4. For $\rightarrow$: $$(G \rightarrow H)^Y = \begin{cases} G^Y \rightarrow H^Y & \text{if } Y \models G \rightarrow H \\ \bot & \text{otherwise.} \end{cases}$$

We could have folded case (R4) into the case (R3). However, all definitions of reduct we consider later in the paper differ only in the way the implication is handled and so, we show this case separately.

For a theory $\mathcal{F}$, we define the GL-reduct $\mathcal{F}^Y$ by setting $\mathcal{F}^Y = \{F^Y \mid F \in \mathcal{F}\}$. Next, we define $Y \subseteq At$ to be a *stable model* of $\mathcal{F}$ if Y is a minimal model of the theory $\mathcal{F}^Y$. One can show that stable models are models (hence, the term stable *model* is justified).

This notion of a stable model generalizes all earlier ones. It also coincides with the one proposed by Pearce [5]. The approach by Pearce is based on the logic HT [6], a logic located strictly between the intuitionistic and the propositional logics. Stable models are defined in terms of the satisfiability relation $\models_{ht}$ in the logic HT. A pair $\langle X, Y\rangle$, where $X, Y \subseteq At$, is an *HT-interpretation* if $X \subseteq Y$. The relation $\models_{ht}$, between HT-interpretations and formulas, is defined inductively as follows:

1. $\langle X, Y\rangle \not\models_{ht} \bot$
2. $\langle X, Y\rangle \models_{ht} A$ if $X \models A$ (applies only if $A \in At$)
3. $\langle X, Y\rangle \models_{ht} F \wedge G$ if $\langle X, Y\rangle \models_{ht} F$ and $\langle X, Y\rangle \models_{ht} G$
4. $\langle X, Y\rangle \models_{ht} F \vee G$ if $\langle X, Y\rangle \models_{ht} F$ or $\langle X, Y\rangle \models_{ht} G$
5. $\langle X, Y\rangle \models_{ht} F \rightarrow G$ if $Y \models F \rightarrow G$; and $\langle X, Y\rangle \not\models_{ht} F$, or $\langle X, Y\rangle \models_{ht} G$.

The relation extends in a standard way to theories. If for a theory $\mathcal{F}$, $\langle X, Y\rangle \models_{ht} \mathcal{F}$, then $\langle X, Y\rangle$ is an *HT-model* of $\mathcal{F}$.

Pearce [5] defined Y to be a stable model of a theory $\mathcal{F}$ if and only if $\langle Y, Y\rangle \models_{ht} \mathcal{F}$ and for every $X \subseteq Y$ if $\langle X, Y\rangle \models_{ht} \mathcal{F}$, then $X = Y$ (a form of *minimality*). Lifschitz and Ferraris [9] proved that the two approaches are equivalent by showing the following two key results.

Theorem 1. *Let $\mathcal{F}$ be a theory.*

1. *For every $Y \subseteq At$, $Y \models \mathcal{F}$ if and only if $Y \models \mathcal{F}^Y$*
2. *For every $X \subseteq Y \subseteq At$, $X \models \mathcal{F}^Y$ if and only if $\langle X, Y\rangle \models_{ht} \mathcal{F}$.*

3 FLP Semantics

Faber et al. [10] based their work on a notion of reduct that differs from the one proposed by Gelfond and Lifschitz. Using our notation, it can be defined as follows. Let R be a disjunctive rule

$$A_1 \wedge \ldots \wedge A_m \wedge \neg B_1 \wedge \ldots \wedge \neg B_n \rightarrow C_1 \vee \ldots \vee C_r,$$

where A_i, B_i and C_i are atoms, and let Y be a set of atoms. The *FLP-reduct* $R^{\underline{Y}}$ (the notation we use is meant to distinguish between the FLP- and the GL-reduct) is either R, if $Y \models A_1 \wedge \ldots \wedge A_m \wedge \neg B_1 \wedge \ldots \wedge \neg B_n$, or $\top$, otherwise. Given a disjunctive program $\mathcal{P}$, $\mathcal{P}^{\underline{Y}}$ is obtained by replacing each rule $R \in \mathcal{P}$ with $R^{\underline{Y}}$. Finally, Y is a *stable model* of $\mathcal{P}$ in the sense of Faber et al., if Y is a minimal model of $\mathcal{P}^{\underline{Y}}$. Faber et al. [10] proved that *their* stable models of disjunctive programs coincide with standard stable models. They also observed that the FLP-reduct does not depend on the syntactic form of the body of a rule. All that matters is whether the body is satisfied by Y. Thus, they extended the definition to more general formulas that are of the form

$$F \rightarrow C_1 \vee \ldots \vee C_r, \tag{2}$$

where C_i are atoms and F is a propositional formula.[1] That allowed them to extend the concept of a stable model to the class of theories that consist of such "generalized" disjunctive rules. Importantly, they proved that stable models, in their sense, of such theories are *minimal models*, while the stable-model semantics does not have that property (for instance, the program $\mathcal{P} = \{\neg\neg A \rightarrow A\}$ has only one "Faber et al." stable model, $\emptyset$, but two standard stable models, $\emptyset$ and $\{A\}$).

[1] They used conjunctions of literals and aggregate atoms as F, but that detail is immaterial here.

3.1 General FLP Semantics

To extend that approach to arbitrary propositional theories, we first generalize the notion of the FLP-reduct. To this end, we follow the inductive pattern of the definition of the GL-reduct. There is no change for $F = \bot$, $F = A$, where $A \in At$, and $F = G \circ H$, where $\circ = \vee$ and $\wedge$. Indeed, there does not seem to be any other way, in which these cases could be handled. Thus, the only case that requires a discussion is that of $F = G \rightarrow H$. Once that case is settled, we will define Y to be an *FLP-stable model* of a theory $\mathcal{F}$ if Y is a minimal model of the FLP-reduct $\mathcal{F}^{\underline{Y}}$.

So, let us discuss the case of the implication. A literal reading of the FLP-reduct for rules suggests the following inductive definition for the case $F = G \rightarrow H$:

$$(G \rightarrow H)^{\underline{Y}} = G \rightarrow H, \text{ if } Y \models G; \text{ otherwise, } (G \rightarrow H)^{\underline{Y}} = \top.$$

However, under that choice, all occurrences of $\rightarrow$ (and so, also all occurrences of $\neg$) in the consequent of another implication would be interpreted in the classical way. While not a problem for formulas that do not have any implications occurring in the consequent of any "top-level" implication (and so, working correctly for the class of formulas considered by Faber et al.), in general it leads to some counterintuitive behavior.

For instance, let $\mathcal{F} = \{\neg\neg p\}$ and $\mathcal{G} = \{\neg q \rightarrow \neg\neg p\}$. As q does not appear in the head of the rule of $\mathcal{G}$, it must be false in every reasonable generalization of the stable-model semantics. Consequently, $\mathcal{F}$ and $\mathcal{G}$ should have the same stable models. However, under the proposed definition it would not be so. Let $Y = \{p\}$. Since $\neg\neg p = (p \rightarrow \bot) \rightarrow \bot$ and $Y \not\models p \rightarrow \bot$, we would have $\mathcal{F}^{\underline{Y}} = \{\top\}$. Consequently, Y would not be a minimal model of $\mathcal{F}^{\underline{Y}} = \{\top\}$ (as $\emptyset$ is a model, too) and so, Y would not be a "stable" model of $\mathcal{F}$. On the other hand, as $Y \models \neg q$, $\mathcal{G}^{\underline{Y}} = \{\neg q \rightarrow \neg\neg p\}$. Thus, clearly, Y would be a minimal model of $\mathcal{G}^{\underline{Y}}$ and, consequently, a "stable" model of $\mathcal{G}$. A problem in itself, it brings up yet another one. In $\mathcal{G}$, p has no head occurrence (informally, there is no "defining clause" for p in $\mathcal{G}$), yet $\mathcal{G}$ would have $\{p\}$ as a "stable" model.

Thus, we need to handle the case of $\rightarrow$ differently, but in such a way that under the restriction to theories consisting of formulas (2) we obtain the same concept of a stable model as the one proposed by Faber et al. In particular, we must ensure that all occurrences of $\rightarrow$ in the consequent of another occurrence of $\rightarrow$ are treated consistently in the same non-classical way. In the remainder of this section we will argue that it can be accomplished by the following definition:

$$\text{FLP4. } (G \rightarrow H)^{\underline{Y}} = \begin{cases} G \rightarrow H^{\underline{Y}} & \text{if } Y \models G \text{ and } Y \models H \\ \top & \text{if } Y \not\models G \\ \bot & \text{otherwise (that is, when } Y \not\models G \rightarrow H). \end{cases}$$

While it looks different than the original definition by Faber et al. [10], it preserves its basic idea of keeping intact the bodies of rules that contribute to the reduct. Indeed, when the implication is "strongly" satisfied (both its antecedent and consequent are satisfied by Y), we keep the antecedent unchanged. However, to make sure the implications occurring in the antecedent are treated in a consistent way, we replace the consequent recursively with its reduct. The case when Y "weakly" satisfies the implication, that is, does not satisfy its antecedent, is dealt with as in the previous attempt (and as in the definition by Faber et al.), reflecting the principle that if the implication "does not fire," it is immaterial and can be replaced by $\top$ ("removed"). In the case when the implication is not satisfied by Y, it can be replaced by $\bot$. Faber et al. do not distinguish this case and, in fact, proceed differently. They keep the rule in the program. However, they could have replaced it with $\bot$, as we propose (following the pattern for GL-reduct), without affecting the resulting concept of a stable model. Indeed, if Y does not satisfy a rule in a program, Y cannot be a stable model of that program. Replacing a rule violated by Y with $\bot$ just makes that explicit.

To summarize, we define the *FLP-reduct* of the formula F with respect to Y, $F^{\underline{Y}}$, recursively, by using the clauses (R1) - (R3) of the definition of the GL-reduct (adjusted to the notation $F^{\underline{Y}}$), as well as the clause (FLP4) for the implication $\rightarrow$. We extend the definition to theories in the standard way. With this definition in hand, we define next the notion of an FLP-stable model of a propositional theory (as announced above).

Definition 1. *Let $\mathcal{F}$ be a theory. A set of atoms Y is an* FLP-stable model *of $\mathcal{F}$ if Y is a minimal model of $F^{\underline{Y}}$.*

3.2 Basic Properties

We start with a generalization of the well-known property of the standard GL-reduct of disjunctive programs (cf. Theorem 1).

Proposition 1. *For every theory $\mathcal{F}$ and for every set of atoms Y, $Y \models \mathcal{F}$ if and only if $Y \models \mathcal{F}^{\underline{Y}}$.*

Proof. It is enough to prove that for every formula F, we have $Y \models F$ if and only if $Y \models F^{\underline{Y}}$. We proceed by induction. The base cases of $F = \bot$ and $F = A$, where $A \in At$, are evident. Let $F = G \wedge H$. If $Y \not\models F$, then $F^{\underline{Y}} = \bot$. Thus, both sides of the equivalence are false, and the equivalence follows. If $Y \models F$ or $Y \models F^{\underline{Y}}$, then $F^{\underline{Y}} = G^{\underline{Y}} \wedge H^{\underline{Y}}$. By the definition:

1. $Y \models F$ if and only if $Y \models G$ and $Y \models H$, and
2. $Y \models F^{\underline{Y}}$ if and only if $Y \models G^{\underline{Y}}$ and $Y \models H^{\underline{Y}}$.

By the induction hypothesis, the equivalence of $Y \models F$ and $Y \models F^{\underline{Y}}$ follows. The argument for $\vee$ is similar. Thus, let $F = G \rightarrow H$. If $Y \not\models F$, then $F^{\underline{Y}} = \bot$ and the equivalence in the assertion holds. Similarly, if $Y \not\models G$, then $F^{\underline{Y}} = \top$, and both $Y \models F$ and $Y \models F^{\underline{Y}}$ hold. Finally, let $Y \models G$ and $Y \models H$. In that case, $F^{\underline{Y}} = G \rightarrow H^{\underline{Y}}$. By the inductive hypothesis, $Y \models H^{\underline{Y}}$ and so, $Y \models G \rightarrow H^{\underline{Y}}$. Thus, also in that case, both $Y \models F$ and $Y \models F^{\underline{Y}}$ hold. □

It follows that FLP-stable models are indeed models of formulas and theories.

Corollary 1. *Let $\mathcal{F}$ be a theory and Y a set of atoms. If Y is an FLP-stable model of $\mathcal{F}$, then Y is a model of $\mathcal{F}$.*

This result allows us to prove that on theories consisting of formulas of the form (2) FLP-stable models defined here and stable models of Faber et al. [10] coincide. Thus, our approach is a generalization of the one by Faber et al.

Theorem 2. *Let $\mathcal{P}$ be a theory consisting of formulas of type (2). Then Y is a stable model of $\mathcal{P}$ according to the definition by Faber et al. [10] if and only if Y is the FLP-stable model, according to Definition 1.*

Proof. Let $\mathcal{P}$ be a theory consisting of formulas (2), and let Y be a set of atoms. For a formula $R = F \rightarrow C_1 \vee \ldots \vee C_r$ from $\mathcal{P}$, we denote by R' and R'' the reducts of R with respect to Y according to Faber et al., and according to our definition, respectively. We also extend this notation to sets of such formulas.

Reasoning in either direction we can assume that Y is a model of $\mathcal{P}$ (it is known that stable models according to Faber et al., are models [10]; for FLP-stable models, it follows from Corollary 1). Thus, $\mathcal{P}'$ consists of those rules $R = F \rightarrow C_1 \vee \ldots \vee C_r$, for which $Y \models F$. In addition, it might possibly contain $\top$. The reduct $\mathcal{P}''$ differs only in that each formula $R = F \rightarrow C_1 \vee \ldots \vee C_r$ from $\mathcal{P}$ that is retained in $\mathcal{P}'$, contributes to $\mathcal{P}''$ its reduct $R'' = F \rightarrow C_1' \vee \ldots \vee C_t'$, where $C_1', \ldots C_t'$ are precisely those elements in $\{C_1, \ldots, C_r\}$ that hold in Y. In addition, as $\mathcal{P}'$, $\mathcal{P}''$ may also contain $\top$. It is evident, that for every $Z \subseteq Y$, $Z \models \mathcal{P}'$ if and only if $Z \models \mathcal{P}''$. Thus, Y is a minimal model of $\mathcal{P}'$ if and only if Y is a minimal model of $\mathcal{P}''$, and so, the result follows. □

One of problematic properties of the literal attempt to generalize the approach by Faber et al. was that stable models of some theories contained atoms without head occurrences. The next result shows that our generalization behaves properly.

Proposition 2. *Let $\mathcal{F}$ be a theory and Y an FLP-stable model of $\mathcal{F}$. Then every atom in Y has a head occurrence in $\mathcal{F}$.*

Finally, we note two properties that we use later in the paper.

Proposition 3. *For every formulas F and G, and for every set of atoms Y:*

1. $F^{\underline{Y}} \equiv \bot$ *if and only if* $Y \not\models F$
2. $(F \circ G)^{\underline{Y}} \equiv F^{\underline{Y}} \circ G^{\underline{Y}}$*, where* $\circ = \wedge$ *or* $\vee$.

3.3 Minimal-Model Property

The main objective of Faber et al. [10] was to generalize the stable-model semantics to the class of theories consisting of rules of the form (2) so that stable models would be minimal models. Faber et al. proved that their generalization indeed has that property.

The extended FLP semantics has the minimal-model property for a broad class of theories, including those consisting of rules (2), but not in general. Let $F = \neg A \vee A$ and $Y = \emptyset$. Since $Y \models A \rightarrow \bot$ and $Y \not\models A$, $(\neg A)^{\underline{Y}} = (A \rightarrow \bot)^{\underline{Y}} = \top$. Moreover, $A^{\underline{Y}} = \bot$. Thus, $F^{\underline{Y}} \equiv (\neg A)^{\underline{Y}} \vee A^{\underline{Y}} \equiv \top$. Clearly, Y is a minimal model of $F^{\underline{Y}}$ and so, an FLP-stable model of F. Next, let us consider $Z = \{A\}$. We now have $(\neg A)^{\underline{Z}} = (A \rightarrow \bot)^{\underline{Z}} = \bot$ and $A^{\underline{Z}} = A$. Thus, $F^{\underline{Z}} \equiv (\neg A)^{\underline{Z}} \vee A^{\underline{Z}} \equiv A$. Again, Z is a minimal model of $F^{\underline{Z}}$ and so, an FLP-stable model of F. Thus, FLP-stable models of F do not form an antichain and Z is not a minimal model of F.

To describe a broad class of theories for which FLP-stable models are minimal models, we introduce *disjunctive-monotone* formulas. A formula F is *monotone* if for every $X \subseteq Y \subseteq At$, $X \models F$ implies $Y \models F$. A formula F is *disjunctive-monotone* if every occurrence of $\vee$ in F operates on monotone formulas.

Proposition 4. *For every disjunctive-monotone formula F and every sets of atoms X and Y such that $X \subseteq Y$, if $X \models F$ and $Y \models F$ then $X \models F^{\underline{Y}}$.*

Proof. We proceed by induction. The case of $F = \bot$ is vacuously true. If $F = A$, where A is an atom, then $F^{\underline{Y}} = A = F$ (it follows from the assumption that $Y \models A$). Thus, $X \models F^{\underline{Y}}$. For the inductive step, there are three cases to consider.

Case 1. $F = G \wedge H$. Clearly, both formulas G and H are disjunctive-monotone, $X \models G$, $X \models H$, $Y \models G$ and $Y \models H$. By the induction hypothesis, $X \models G^{\underline{Y}}$ and $X \models H^{\underline{Y}}$. Consequently, $X \models G^{\underline{Y}} \wedge H^{\underline{Y}} = (G \wedge H)^{\underline{Y}} = F^{\underline{Y}}$.
Case 2. $F = G \vee H$. Since $X \models F$, $X \models G$ or $X \models H$. Wlog we may assume that $X \models G$. Since F is disjunctive-monotone, G is disjunctive-monotone. Moreover, G is monotone. Thus, $Y \models G$. By the induction hypothesis, $X \models G^{\underline{Y}}$. Since $F^{\underline{Y}} \equiv G^{\underline{Y}} \vee H^{\underline{Y}}$, $X \models F^{\underline{Y}}$.
Case 3. $F = G \rightarrow H$. Since $Y \models F$, $F^{\underline{Y}} \neq \bot$. If $F^{\underline{Y}} = \top$ then $X \models F^{\underline{Y}} = \top$. Thus, we may assume that $Y \models G$, $Y \models H$ and $F^{\underline{Y}} = G \rightarrow H^{\underline{Y}}$. If $X \not\models G$, then $X \models F^{\underline{Y}}$. If $X \models G$, then $X \models H$. Since H is disjunctive-monotone, by induction it holds that $X \models H^{\underline{Y}}$. Thus, $X \models F^{\underline{Y}}$ in that case, too. □

Corollary 2. *Let $\mathcal{F}$ be a theory such that every formula in $\mathcal{F}$ is of the form H or $G \rightarrow H$, where H is disjunctive-monotone. For every $X \subseteq Y \subseteq At$, if $X \models \mathcal{F}$ and $Y \models \mathcal{F}$, then $X \models \mathcal{F}^{\underline{Y}}$.*

Proof. To prove the result, it suffices to prove it for each formula F in $\mathcal{F}$. If F is monotone-disjunctive, then the result follows from Proposition 4. If $F = G \rightarrow H$, where H is monotone-disjunctive, we reason as follows. Since $Y \models F$, we have $F^{\underline{Y}} = \top$; or $Y \models G$, $Y \models H$ and $F^{\underline{Y}} = G \rightarrow H^{\underline{Y}}$. In the first case, $X \models F^{\underline{Y}}$ is evident. In the second case, if $X \not\models G$, the assertion follows. Otherwise, since $X \models F$, $X \models H$. By Proposition 4, $X \models H^{\underline{Y}}$ follows. Consequently, $X \models F^{\underline{Y}}$ follows, as well. □

Corollary 3. *Let $\mathcal{F}$ be a theory such that every formula in $\mathcal{F}$ is of the form H or $G \rightarrow H$, where H is disjunctive-monotone. If Y is an FLP-stable model of $\mathcal{F}$ then Y is a minimal model of $\mathcal{F}$.*

Proof. Since Y is a model of $\mathcal{F}^{\underline{Y}}$, Y is a model of $\mathcal{F}$ (Proposition 1). Let us assume that $X \models \mathcal{F}$ and $X \subseteq Y$. By Corollary 2, $X \models \mathcal{F}^{\underline{Y}}$. Since Y is a minimal model of $\mathcal{F}^{\underline{Y}}$, $X = Y$. Thus, Y is a minimal model of $\mathcal{F}$. □

Corollary 3 extends the result by Faber et al., as it applies to theories consisting of formulas of type (2). It can be generalized further to the case, where each formula in a theory is of the form $H_k \rightarrow (H_{k-1} \rightarrow (\ldots \rightarrow (H_1 \rightarrow H_0)\ldots))$, where $k \geq 0$ and H_0 is disjunctive monotone. The argument is essentially the same.

3.4 Computational Complexity for FLP Semantics

It is well known that the truth value of a formula in an interpretation can be found in polynomial time. It follows that given a formula and a set of atoms Y, one can compute $F^{\underline{Y}}$ in polynomial time by means of a simple recursive algorithm that directly follows the definition of the reduct. It follows also that the problem to decide whether a model of a formula is a minimal model is in the class coNP. Thus, the existence of the FLP-stable model is in the class Σ_2^P. Σ_2^P-hardness of the problem follows from the fact that on disjunctive programs FLP-stable models coincide with stable models [10], and the existence problem for stable models is Σ_2^P-complete [15]. Consequently, deciding the existence of an FLP-stable model is Σ_2^P-complete, too. We state that result below, together with two other related results that can be proved by similar arguments.

Theorem 3. *The problem of the existence of the FLP-stable model is Σ_2^P-complete. The skeptical reasoning with FLP-stable models (is a given atom a member of every FLP-stable model) is Π_2^P-complete. The brave reasoning with FLP-stable models (is a given atom a member of some FLP-stable model) is Σ_2^P-complete.*

3.5 HT-Interpretations and FLP Semantics — Strong Equivalence

We now describe FLP-stable models in terms of HT-interpretations, and apply that result to characterize strong equivalence with respect to the FLP semantics. First, we define a certain satisfiability relation $\models_{flp}$ between HT-interpretations and formulas. The definition is inductive and follows the same pattern as that for $\models_{ht}$. The cases $\langle X, Y\rangle \models_{flp} F$ for $F = \bot$, $F = A$, where $A \in At$, $F = G \wedge H$ and $G \vee H$, are handled as in the case of $\models_{ht}$. For the implication we have the following clause:

5′. $\langle X, Y\rangle \models_{flp} G \rightarrow H$ if $Y \models G \rightarrow H$; and $Y \not\models G$, or $X \not\models G$, or $\langle X, Y\rangle \models_{flp} H$.

The relation $\models_{flp}$ extends in a standard way to HT-interpretations and *sets* of formulas. If $\mathcal{F}$ is a theory and $\langle X, Y\rangle \models_{flp} \mathcal{F}$, we say that $\langle X, Y\rangle$ is an *FLP-model* of $\mathcal{F}$ (not to be confused with an FLP-*stable* model).

We have the following simple property of $\models_{flp}$, mirroring a similar one for $\models_{ht}$ [9].

Proposition 5. *Let $\mathcal{F}$ be a theory. For every sets $X \subseteq Y \subseteq At$, if $\langle X, Y\rangle \models_{flp} \mathcal{F}$, then $Y \models \mathcal{F}$.*

Proof. It suffices to prove that for every formula F, if $\langle X, Y\rangle \models_{flp} F$, then $Y \models F$ The case $F = \bot$ is evident. If $F = A$, where $A \in At$, and $\langle X, Y\rangle \models_{flp} F$, then $A \in X$. Thus, $A \in Y$ and $Y \models F$. The inductive step for $F = G \wedge H$ and $F = G \vee H$ is standard. If $F = G \rightarrow H$ and $\langle X, Y\rangle \models_{flp} F$ then, in particular, $Y \models F$ (by the definition of $\models_{flp}$ for the case of implication). Thus, the result follows. □

The $\models_{flp}$ relation and the FLP-reduct are closely connected (cf. Theorem 1).

Proposition 6. *For every formula F and for every two sets of atoms $X \subseteq Y$, $X \models F^{\underline{Y}}$ if and only if $\langle X, Y\rangle \models_{flp} F$.*

Proof. We proceed by induction. The case when $F = \bot$ is straightforward. Let $F = A$, where $A \in At$. If $X \models A^{\underline{Y}}$, then $A^{\underline{Y}} \neq \bot$. Thus, $A^{\underline{Y}} = A$. It follows that $X \models A$ and so, $\langle X, Y\rangle \models_{flp} A$. Conversely, if $\langle X, Y\rangle \models_{flp} A$, then $X \models A$. Since $X \subseteq Y$, $Y \models A$. Thus, $A^{\underline{Y}} = A$ and $X \models A^{\underline{Y}}$ as required.

Next, let $F = G \wedge H$. If $X \models (G \wedge H)^{\underline{Y}}$, then $(G \wedge H)^{\underline{Y}} = G^{\underline{Y}} \wedge H^{\underline{Y}}$. Thus, $X \models G^{\underline{Y}}$ and $X \models H^{\underline{Y}}$. By the inductive hypothesis, $\langle X, Y\rangle \models_{flp} G$ and $\langle X, Y\rangle \models_{flp} H$. Thus, $\langle X, Y\rangle \models_{flp} G \wedge H$, as needed. Conversely, let $\langle X, Y\rangle \models_{flp} G \wedge H$. Then $\langle X, Y\rangle \models_{flp} G$ and $\langle X, Y\rangle \models_{flp} H$ and, by the inductive hypothesis, $X \models G^{\underline{Y}}$ and $X \models H^{\underline{Y}}$. Thus, $X \models G^{\underline{Y}} \wedge H^{\underline{Y}}$. By Proposition 3, $G^{\underline{Y}} \wedge H^{\underline{Y}} \equiv (F \wedge G)^{\underline{Y}}$. Thus, $X \models (F \wedge G)^{\underline{Y}}$.

The argument for the case $F = G \vee H$ is similar. Thus, we move on to the case $F = G \to H$. We have the following equivalences:

1. $X \models (G \to H)^{\underline{Y}}$
2. $Y \not\models G$; or $Y \models G$ and $Y \models H$, and $X \models G \to H^{\underline{Y}}$
3. $Y \not\models G$; or $Y \models H$ and $X \models G \to H^{\underline{Y}}$
4. $Y \not\models G$ or $Y \models H$; and $Y \not\models G$ or $X \models G \to H^{\underline{Y}}$
5. $Y \models G \to H$; and $Y \not\models G$ or $X \not\models G$, or $X \models H^{\underline{Y}}$
6. $Y \models G \to H$; and $Y \not\models G$ or $X \not\models G$, or $\langle X, Y\rangle \models_{flp} H$

The last statement is equivalent to $\langle X, Y\rangle \models_{flp} F$ and the result follows. □

Corollary 4. *Let $\mathcal{F}$ be a theory and Y a set of atoms. Then Y is an FLP-stable model of $\mathcal{F}$ if and only if $\langle Y, Y\rangle \models_{flp} \mathcal{F}$ and for every $X \subset Y$, $\langle X, Y\rangle \not\models_{flp} \mathcal{F}$.*

Proof. By the definition, Y is an FLP-stable model of $\mathcal{F}$ if and only if $Y \models \mathcal{F}^{\underline{Y}}$ and, for every $X \subset Y$, $X \not\models \mathcal{F}^{\underline{Y}}$. We apply Proposition 6. The former condition is equivalent to $\langle Y, Y\rangle \models_{flp} \mathcal{F}$. The latter one is equivalent to $\langle X, Y\rangle \not\models_{flp} \mathcal{F}$. Thus, the assertion follows. □

We are now ready to discuss the notion of strong FLP-equivalence. Theories $\mathcal{F}$ and $\mathcal{G}$ and *strongly FLP-equivalent* if for every theory $\mathcal{H}$, the theories $\mathcal{F} \cup \mathcal{H}$ and $\mathcal{G} \cup \mathcal{H}$ have the same FLP-stable models. This is a literal adaptation of the standard definition of strong equivalence [7] to the case of FLP-stable models.

Theorem 4. *Let $\mathcal{F}$ and $\mathcal{G}$ be two formulas. Then, $\mathcal{F}$ and $\mathcal{G}$ are strongly FLP-equivalent if and only if $\mathcal{F}$ and $\mathcal{G}$ have the same FLP-models.*

Proof. ($\Leftarrow$) For every theory $\mathcal{H}$, $\langle X, Y\rangle \models_{flp} \mathcal{F} \cup \mathcal{H}$ if and only if $\langle X, Y\rangle \models_{flp} \mathcal{G} \cup \mathcal{H}$. By Corollary 4, $\mathcal{F} \cup \mathcal{H}$ and $\mathcal{G} \cup \mathcal{H}$ have the same FLP-stable models.
($\Rightarrow$) Let us assume that there are $X \subseteq Y \subseteq At$ such that $\langle X, Y\rangle$ satisfies one of $\mathcal{F}$ and $\mathcal{G}$ but not the other. Wlog, we may assume that $\langle X, Y\rangle \models_{flp} \mathcal{F}$ and $\langle X, Y\rangle \not\models_{flp} \mathcal{G}$. By Proposition 6, it follows that $X \models \mathcal{F}^{\underline{Y}}$ and $X \not\models \mathcal{G}^{\underline{Y}}$. The first property implies that $\mathcal{F}^{\underline{Y}} \not\equiv \bot$. Consequently, by Proposition 3, $Y \models \mathcal{F}$. By Proposition 1, $Y \models \mathcal{F}^{\underline{Y}}$.
Case 1. $Y \not\models \mathcal{G}^{\underline{Y}}$. It follows that $\langle Y, Y\rangle \not\models_{flp} \mathcal{G}$. Thus, for every $\mathcal{H}$, $Y \not\models \mathcal{G} \cup \mathcal{H}$ and so, Y is not an FLP-stable model of $\mathcal{G} \cup \mathcal{H}$. Let us now define $\mathcal{H} = Y$. We have $(\mathcal{F} \cup \mathcal{H})^{\underline{Y}} \equiv \mathcal{F}^{\underline{Y}} \cup \mathcal{H}^{\underline{Y}}$. Moreover, $\mathcal{H}^{\underline{Y}} = \mathcal{H}$. Thus, $(\mathcal{F} \cup \mathcal{H})^{\underline{Y}} \equiv \mathcal{F}^{\underline{Y}} \cup \mathcal{H}$. It follows that (a) $Y \models (\mathcal{F} \cup \mathcal{H})^{\underline{Y}}$, and (b) there is no $X \subset Y$ such that $X \models (\mathcal{F} \cup \mathcal{H})^{\underline{Y}}$. Thus, Y is an FLP-stable model of $\mathcal{F} \cup \mathcal{H}$. As we noted, Y is not an FLP-stable model of $\mathcal{G} \cup \mathcal{H}$. Thus, $\mathcal{F}$ and $\mathcal{G}$ are not strongly FLP-equivalent, a contradiction.
Case 2. $Y \models \mathcal{G}^{\underline{Y}}$. We recall that $X \not\models \mathcal{G}^{\underline{Y}}$. Thus, $X \subset Y$. We define

$$\mathcal{H} = X \cup \{A \to B \mid A, B \in Y \setminus X\}.$$

We have $(\mathcal{F} \cup \mathcal{H})^{\underline{Y}} \equiv \mathcal{F}^{\underline{Y}} \cup \mathcal{H}^{\underline{Y}}$, Moreover, it is easy to check that $\mathcal{H}^{\underline{Y}} = \mathcal{H}$. Thus, $(\mathcal{F} \cup \mathcal{H})^{\underline{Y}} \equiv \mathcal{F}^{\underline{Y}} \cup \mathcal{H}$. We recall that $X \models \mathcal{F}^{\underline{Y}}$. We also have $X \models \mathcal{H}$. Thus, $X \models (\mathcal{F} \cup \mathcal{H})^{\underline{Y}}$ and so, Y is not an FLP-stable model of $\mathcal{F} \cup \mathcal{H}$. It follows that Y is not an FLP-stable model of $\mathcal{G} \cup \mathcal{H}$ and so, there is $Z \subset Y$ such that $Z \models (\mathcal{G} \cup \mathcal{H})^{\underline{Y}}$. Since $(\mathcal{G} \cup \mathcal{H})^{\underline{Y}} \equiv \mathcal{G}^{\underline{Y}} \cup \mathcal{H}$, $Z \models \mathcal{H}$ and so, $Z = X$. However, $X \not\models \mathcal{G}^{\underline{Y}}$, a contradiction. □

4 Normal Forms and a Comparison with Stable-Model Semantics

The following result was obtained by Cabalar and Ferraris [16]. It concerns representing theories by programs — theories consisting of rules (formulas of the form (1)).

Theorem 5. *For every theory $\mathcal{F}$ there is a program $\mathcal{G}$ (in the same language) such that $\mathcal{F}$ and $\mathcal{G}$ have the same HT-models (are equivalent in the logic HT).*

In other words, every theory $\mathcal{F}$ is strongly equivalent to some program $\mathcal{G}$. A similar result holds for the FLP-models. In what follows we write $\neg Y$ for $\{\neg y \mid y \in Y\}$. We first state three auxiliary results (the proofs are simple and we omit them).

Proposition 7. *Let $X \subset Y \subseteq Z$ be finite. Then $\langle U, V\rangle$, where $U \subseteq V \subseteq Z$, is an FLP-countermodel of $\bigwedge X \wedge \bigwedge \neg\overline{Y} \rightarrow \bigvee \overline{X} \vee \bigvee \neg Y$ (where the set complements $\overline{X}$ and $\overline{Y}$ are defined with respect to Z) if and only if $U = X$ and $V = Y$.*

Proposition 8. *Let $Y \subseteq Z$ be finite. Then $\langle U, V\rangle$, where $U \subseteq V \subseteq Z$, is an FLP-countermodel to $\bigwedge Y \wedge \bigwedge \neg\overline{Y} \rightarrow \bot$ (where the set complement $\overline{Y}$ is defined with respect to Z) if and only if $V = Y$.*

Proposition 9. *Let F be a formula. If $\langle Y, Y\rangle$ is an FLP-countermodel of F, then for every $X \subseteq Y$, $\langle X, Y\rangle$ is an FLP-countermodel of F.*

Theorem 6. *Let $\mathcal{F}$ be a theory. There exists a program $\mathcal{G}$ such that $\mathcal{F}$ and $\mathcal{G}$ have the same FLP-models.*

Proof. For $F \in \mathcal{F}$, we consider FLP-countermodels $\langle X, Y\rangle$ of F such that $Y \subseteq At(F)$. For each FLP-countermodel $\langle X, Y\rangle$ with $X \subset Y$, we take the formula defined in Proposition 7 (with $Z = At(F)$). For each countermodel $\langle X, Y\rangle$ such that $X = Y$, we take the formula from Proposition 8. We take for $\mathcal{G}$ the set of all rules constructed in that way from countermodels of formulas in $\mathcal{F}$. By Proposition 9, $\mathcal{F}$ and $\mathcal{G}$ have the same FLP-countermodels consisting of atoms in $At(\mathcal{F})$ and so, the same FLP-models consisting of atoms in $At(\mathcal{F})$. Thus, they have the same FLP-models. □

We saw that not every stable model of a theory is an FLP-stable model of a theory. We also note that not every FLP-stable model is a stable model. For instance, let $F = (A \vee \neg A) \rightarrow A$, and $Y = \{A\}$. It is easy to check that $F^Y = A \rightarrow A$. Since $\emptyset \models F^Y$, Y is not a stable model of F^Y. However, $F^{\underline{Y}} = (A \vee \neg A) \rightarrow A \equiv A$. Thus, Y is an FLP-stable model of F. It follows that the two semantics are different and neither is stronger than the other one. However, each can be expressed in terms of the other one. To see that, we first observe that HT- and FLP-models of rules coincide.

Proposition 10. *Let R be a rule. Then, R has the same HT- and FLP-models.*

Theorems 5 and 6 yield now the following two corollaries relating the two semantics.

Corollary 5. *For every theory $\mathcal{F}$ there are programs $\mathcal{F}'$ and $\mathcal{F}''$ such that*

1. *$\langle X, Y\rangle$ is an HT-model of $\mathcal{F}$ if and only if $\langle X, Y\rangle$ is an FLP-model of $\mathcal{F}'$*
2. *$\langle X, Y\rangle$ is an FLP-model of $\mathcal{F}$ if and only if $\langle X, Y\rangle$ is an HT-model of $\mathcal{F}''$*

Corollary 6. *For every theory $\mathcal{F}$ there are programs $\mathcal{F}'$ and $\mathcal{F}''$ such that*

1. *Y is a stable model of $\mathcal{F}$ if and only if Y is an FLP-stable model of $\mathcal{F}'$*
2. *Y is an FLP-stable model of $\mathcal{F}$ if and only if Y is a stable model of $\mathcal{F}''$*

5 Supported Models

The approach that yielded generalizations of stable- and FLP-model semantics for arbitrary propositional theories can also be applied to the supported-model semantics.

For a formula F and a set of atoms X, we define the *SPP-reduct* of F with respect to Y, written as $F^{\underline{\underline{Y}}}$, by adapting to the new notation the inductive clauses (R1) - (R3), and using the following definition for the implication:

$$\text{SPP4.} \qquad (G \rightarrow H)^{\underline{\underline{Y}}} = \begin{cases} H^{\underline{\underline{Y}}} & \text{if } Y \models G \text{ and } Y \models H \\ \top & \text{if } Y \not\models G \\ \bot & \text{otherwise.} \end{cases}$$

This notion of reduct is motivated by the definition of supported models in the case of programs with disjunctive rules (no negation in the head) [17]. The reduct that is relevant there consists of the heads of rules with bodies satisfied by Y. In the first case, we define the reduct $(G \rightarrow H)^{\underline{\underline{Y}}}$ to be $H^{\underline{\underline{Y}}}$ rather than just H due to the same reasons we followed when generalizing the FLP-reduct.

Definition 2. *Let $\mathcal{F}$ be a theory. A set of atoms Y is a* supported model *of $\mathcal{F}$ if Y is a minimal model of $\mathcal{F}^{\underline{\underline{Y}}}$.*

The results and the proofs that worked in the case of stable-model and FLP semantics work, with only minor changes (and with one exception), in the case of supported models, too. Thus, we omit most of the proofs. We start by gathering in one result several basic properties of the SPP-reduct and supported models.

Proposition 11. *For every theory $\mathcal{F}$ and every set of atoms Y:*

1. *$Y \models \mathcal{F}$ if and only if $Y \models F^{\underline{\underline{Y}}}$*
2. *if Y is a supported model of $\mathcal{F}$, then Y is a model of $\mathcal{F}$*
3. *if Y is a supported model of $\mathcal{F}$, then every atom in Y has a head occurrence in $\mathcal{F}$.*

Next, we characterize supported models in terms of a certain satisfiability relation that connects HT-interpretations and formulas. It follows closely the definitions of $\models_{ht}$ and $\models_{flp}$ but is modified for the case of the implication.

$5''$. $\langle X, Y\rangle \models_{spp} F \rightarrow G$ if $Y \models F \rightarrow G$, and $Y \not\models F$ or $\langle X, Y\rangle \models_{spp} G$.

If $\langle X, Y\rangle \models_{spp} \mathcal{F}$, we say that $\langle X, Y\rangle$ is an *SPP-model* of $\mathcal{F}$.

The following result is analogous to similar results for $\models_{ht}$ and $\models_{flp}$, and ties the SPP-reduct and the relation $\models_{spp}$.

Proposition 12. *For every formula F and for every two sets of atoms $X \subseteq Y$, $X \models F^{\underline{Y}}$ if and only if $\langle X, Y\rangle \models_{spp} F$.*

The main consequence of Proposition 12 is a characterization of supported models in terms of the relation $\models_{spp}$.

Corollary 7. *Let $\mathcal{F}$ be a theory and Y a set of atoms. Then Y is a supported model of $\mathcal{F}$ if and only if $\langle Y, Y\rangle \models_{spp} \mathcal{F}$ and for every $X \subset Y$, $\langle X, Y\rangle \not\models_{spp} \mathcal{F}$.*

We use these results to study strong SPP-equivalence of theories. Two theories $\mathcal{F}$ and $\mathcal{G}$ are *strongly SPP-equivalent* if for every theory $\mathcal{H}$, $\mathcal{F} \cup \mathcal{H}$ and $\mathcal{G} \cup \mathcal{H}$ have the same supported models. Corollary 7 implies that if $\mathcal{F}$ and $\mathcal{G}$ have the the same SPP-models then they are strongly SPP-equivalent. Unlike in the other two cases, though, a weaker condition suffices to provide a characterization of strong SPP-equivalence. An SPP-model is *essential* if it is of the form $\langle Y, Y\rangle$ or $\langle Y \setminus \{A\}, Y\rangle$, where $A \in At$. We now have the following characterization of strong SPP-equivalence. Unlike the one developed for programs [18], where the general case is established through a certain reduction to normal programs, the present characterization is direct. We state first an auxiliary property, which can be demonstrated by simple induction.

Proposition 13. *For every formula F and for every interpretation Y, $F^{\underline{Y}}$ is monotone.*

Theorem 7. *Let $\mathcal{F}$ and $\mathcal{G}$ be two theories. Then, $\mathcal{F}$ and $\mathcal{G}$ are strongly SPP-equivalent if and only if $\mathcal{F}$ and $\mathcal{G}$ have the same essential SPP-models.*

Proof. ($\Rightarrow$) Let $\langle Y, Y\rangle$ be an SPP-model of $\mathcal{F}$. It follows that Y is a supported model of $\mathcal{F} \cup Y$. Thus, Y is a supported model of $\mathcal{G} \cup Y$. Consequently, Y is a model of $\mathcal{G}$ and so, $\langle Y, Y\rangle$ is an SPP-model of $\mathcal{G}$ (indeed, by Proposition 11, $Y \models G^{\underline{Y}}$, and the claim follows by Proposition 12). Next, let $\langle Y \setminus \{A\}, Y\rangle$ be an SPP-model of $\mathcal{F}$. It follows that $Y \setminus \{A\} \models \mathcal{F}^{\underline{Y}}$. Let us assume that $\langle Y \setminus \{A\}, Y\rangle$ is not an SPP-model of $\mathcal{G}$. Then, $Y \setminus \{A\} \not\models \mathcal{G}^{\underline{Y}}$. Let us consider $\mathcal{G} \cup (Y \setminus \{A\})$. Since $\langle Y, Y\rangle$ is an SPP-model of $\mathcal{F}$, $\langle Y, Y\rangle$ is an SPP-model of $\mathcal{G}$ (we proved that above). Since every model of $(\mathcal{G} \cup (Y \setminus \{A\}))^{\underline{Y}} \equiv \mathcal{G}^{\underline{Y}} \cup (Y \setminus \{A\})$ contains $Y \setminus \{A\}$, and $Y \setminus \{A\} \not\models \mathcal{G}^{\underline{Y}}$, it follows that Y is a minimal model of $(\mathcal{G} \cup (Y \setminus \{A\}))^{\underline{Y}}$. Thus, Y is a supported model of $\mathcal{G} \cup (Y \setminus \{A\})$. Consequently, it is a supported model of $\mathcal{F} \cup (Y \setminus \{A\})$. But we have $Y \setminus \{A\} \models (\mathcal{F} \cup (Y \setminus \{A\}))^{\underline{Y}}$, a contradiction. Thus, $\langle Y \setminus \{A\}, Y\rangle$ is an essential SPP-model of $\mathcal{G}$. By symmetry, essential SPP-models of $\mathcal{F}$ and $\mathcal{G}$ coincide.

($\Leftarrow$) Let $\mathcal{H}$ be any theory and let Y be a supported model of $\mathcal{F} \cup \mathcal{H}$. It follows that $\langle Y, Y\rangle$ is an SPP-model of $\mathcal{F}$ and of $\mathcal{H}$. By the assumption, $\langle Y, Y\rangle$ is an SPP-model of $\mathcal{G}$ and of $\mathcal{H}$. Thus, $\langle Y, Y\rangle \models_{spp} \mathcal{G} \cup \mathcal{H}$ and so, $Y \models (\mathcal{G} \cup \mathcal{H})^{\underline{Y}}$. Let $X \subset Y$ be such that $X \models (\mathcal{G} \cup \mathcal{H})^{\underline{Y}}$. It follows that $X \models \mathcal{G}^{\underline{Y}}$ and $X \models \mathcal{H}^{\underline{Y}}$. Let $A \in Y \setminus X$ (such an a exists). Then, by Proposition 13, $Y \setminus \{A\} \models \mathcal{G}^{\underline{Y}}$ and $Y \setminus \{A\} \models \mathcal{H}^{\underline{Y}}$. Thus, $\langle Y \setminus \{A\}, Y\rangle$

is an SPP-model of $\mathcal{G}$ and so, of $\mathcal{F}$. Moreover, $Y \setminus \{A\} \models \mathcal{F}^{\underline{\underline{Y}}} \cup \mathcal{H}^{\underline{\underline{Y}}} = (\mathcal{F} \cup \mathcal{H})^{\underline{\underline{Y}}}$. This contradicts Y being a supported model of $\mathcal{F} \cup \mathcal{H}$. Thus, no such X exists and Y is a supported model of $\mathcal{G} \cup \mathcal{H}$. The claim follows now by the symmetry argument. $\square$

We conclude with two properties of supported-model semantics. The first one relates (FLP-)stable and supported models. The second one is a normal form result.

Theorem 8. *For every theory $\mathcal{F}$ and every set of atoms Y, if Y is a stable model of $\mathcal{F}$ or Y is an FLP-stable model of $\mathcal{F}$, then Y is a supported model of $\mathcal{F}$.*

Theorem 9. *Let $\mathcal{F}$ be a theory. Then there is a normal program $\mathcal{G}$ such that $\mathcal{F}$ and $\mathcal{G}$ have the same essential SPP-models (and so, are strongly SPP-equivalent and have the same supported models).*

6 Discussion and Conclusions

Ferraris and Lifschitz [9] proved that the stable-model semantics can be extended to the language of propositional logic by means of an appropriate generalization of the notion of the GL-reduct. We showed that the approach by Ferraris and Lifschitz can be adapted to two other semantics of programs: the FLP and supported-model semantics. Moreover, the generalizations require only small changes in the definition of the reduct that concern how the implication is handled in the case both its antecedent and consequent are satisfied by the context. In the case of the FLP-reduct, we keep the antecedent of the implication unchanged, in the case of the SPP-reduct, we drop it.

Not only the definitions follow the same pattern. The theories of the three semantics are quite similar, too, both in the way the results are stated as well as proved. In particular, in each case, we have a corresponding characterization of the semantics in terms of a satisfiability relation between HT-interpretations and formulas. As before, what differentiates between the relations is the way the implication is handled.

The uniformity with which the three semantics can be defined and studied is striking. It suggests that considering the reduct-based approach in the general language of logic, may reveal new insights into the phenomenon of nonmonotonicity. A related question is whether any other semantics can be defined in this way, that is, whether there are any other notions of reduct that might lead to useful formalisms. As there seem to be no simple ways to modify the reduct left, the uniform approach presented here suggests that the realm of nonmonotonic semantics of programs and theories may essentially boil down to the three ones discussed in the paper.

The uniformity notwithstanding, there are also differences. We saw that the relation $\models_{spp}$ is, in some sense, weaker than the other two. Further, the relation $\models_{ht}$, which captures the stable-models semantics defines a logic, namely the logic HT. To the contrary, the relation $\models_{flp}$, which captures the FLP semantics does not: the set of formulas F such that for every $\langle X, Y\rangle$, $\langle X, Y\rangle \models_{flp} F$, while closed under modus ponens, is not closed under substitution. Also, while stable and FLP semantics are closely related, supported-model semantics is essentially different (cf. the characterization of strong SPP-equivalence, and the normal-form theorem). A detailed comparison of the semantics is beyond the scope of this paper. We leave it for future work.

References

1. Gelfond, M., Lifschitz, V.: The Stable Semantics for Logic Programs. In: Kowalski, R.A., Bowen, K.A. (eds.) Proceedings of the 5th International Conference and Symposium on Logic Programming, pp. 1070–1180. MIT Press, Cambridge (1988)
2. Marek, V., Truszczyński, M.: Stable Models and an Alternative Logic Programming Paradigm. In: Apt, K., Marek, W., Truszczyński, M., Warren, D. (eds.) The Logic Programming Paradigm: a 25-Year Perspective, pp. 375–398. Springer, Berlin (1999)
3. Niemelä, I.: Logic Programming with Stable Model Semantics as a Constraint Programming Paradigm. Annals of Mathematics and Artificial Intelligence 25, 241–273 (1999)
4. Gelfond, M., Leone, N.: Logic Programming and Knowledge Representation – the A-Prolog Perspective. Artificial Intelligence 138, 3–38 (2002)
5. Pearce, D.: A New Logical Characterisation of Stable Models and Answer Sets. In: Dix, J., Przymusinski, T.C., Moniz Pereira, L. (eds.) NMELP 1996. LNCS, vol. 1216, pp. 57–70. Springer, Heidelberg (1997)
6. Heyting, A.: Die Formalen Regeln der Intuitionistischen Logik. Sitzungsberichte der Preussischen Akademie von Wissenschaften. Physikalisch-mathematische Klasse, 42–56 (1930)
7. Lifschitz, V., Pearce, D., Valverde, A.: Strongly Equivalent Logic Programs. ACM Transactions on Computational Logic 2(4), 526–541 (2001)
8. Ferraris, P.: Answer Sets for Propositional Theories. In: Baral, C., Greco, G., Leone, N., Terracina, G. (eds.) LPNMR 2005. LNCS (LNAI), vol. 3662, pp. 119–131. Springer, Heidelberg (2005)
9. Ferraris, P., Lifschitz, V.: Mathematical Foundations of Answer Set Programming. In: Artëmov, S., Barringer, H., d'Avila Garcez, A., Lamb, L.C., Woods, J. (eds.) We Will Show Them! Essays in Honour of Dov Gabbay, pp. 615–664. College Publications (2005)
10. Faber, W., Leone, N., Pfeifer, G.: Recursive aggregates in disjunctive logic programs: Semantics and complexity. In: Alferes, J.J., Leite, J. (eds.) JELIA 2004. LNCS (LNAI), vol. 3229, pp. 200–212. Springer, Heidelberg (2004)
11. Niemelä, I., Simons, P.: Extending the Smodels System with Cardinality and Weight Constraints. In: Minker, J. (ed.) Logic-Based Artificial Intelligence, pp. 491–521. Kluwer Academic Publishers, Boston (2000)
12. Gelfond, M., Lifschitz, V.: Classical Negation in Logic Programs and Disjunctive Databases. New Generation Computing 9, 365–385 (1991)
13. Lifschitz, V., Woo, T.: Answer Sets in General Nonmonotonic Reasoning. In: Nebel, B., Rich, C., Swartout, W.R. (eds.) Proceedings of the 3rd International Conference on Principles of Knowledge Representation and Reasoning, pp. 603–614. Morgan Kaufmann, San Francisco (1992)
14. Lifschitz, V., Tang, L.R., Turner, H.: Nested Expressions in Logic Programs. Annals of Mathematics and Artificial Intelligence 25, 369–389 (1999)
15. Eiter, T., Gottlob, G.: On the Computational Cost of Disjunctive Logic Programming: Propositional Case. Annals of Mathematics and Artificial Intelligence 15, 289–323 (1995)
16. Cabalar, P., Ferraris, P.: Propositional Theories are Strongly Equivalent to Logic Programs. Theory and Practice of Logic Programming 7, 745–759 (2007)
17. Brass, S., Dix, J.: Characterizations of the Disjunctive Stable Semantics by Partial Evaluation. Journal of Logic Programming 32, 207–228 (1997)
18. Truszczyński, M., Woltran, S.: Hyperequivalence of Logic Programs with Respect to Supported Models. In: Fox, D., Gomes, C.P. (eds.) Proceedings of the 23rd National Conference on Artificial Intelligence, pp. 560–565. AAAI Press, Menlo Park (2008)

A Tabling Implementation Based on Variables with Multiple Bindings

Pablo Chico de Guzmán[1], Manuel Carro[1], and Manuel Hermenegildo[1,2]

[1] School of Computer Science, Univ. Politécnica de Madrid, Spain
[2] IMDEA Software, Spain
pchico@clip.dia.fi.upm.es, {mcarro,herme}@fi.upm.es

Abstract. Suspension-based tabling systems have to save and restore computation states belonging to OR branches. Stack freezing combined with (forward) trailing is among the better-known implementation approaches for this purpose. Resuming a goal using this technique reinstalls the bindings for all the variables in the environment where the goal was suspended. In this paper we explore an alternative approach where variables can keep track of *several* bindings, associated with suspensions. Resuming a goal boils down to determining which suspension has to be resumed, in order to select, when dereferencing, the bindings which were active at the moment of suspending. We present the ideas behind this approach, highlight several advantages over other suspension-based implementations, and perform an experimental evaluation. We also recall the similarity between OR-parallelism and suspension-based implementations of tabling, and discuss similarities with the *Version Vectors Method*, among others.

Keywords: Logic Programming, Tabling, Implementation, Performance, OR-Parallelism.

1 Introduction

Tabling [1,2,3] is a strategy for executing logic programs which *memoizes* already processed calls and their answers to improve several of the limitations of the SLD resolution strategy. It guarantees termination for programs with the bounded term size property, improves efficiency in programs which repeatedly perform some computation, and has been successfully applied to deductive databases [4], program analysis [5,6], semantic Web reasoning [7], model checking [8], etc.

There are two main approaches for the implementation of tabling: *suspension-based tabling* and *linear tabling*. In *suspension-based tabling* the computation state of suspended tabled subgoals has to be preserved to avoid backtracking over them. This is done either by *freezing* the stacks, as in XSB [9], by copying to another area, as in CAT [10], or by using an intermediate solution as in CHAT [11]. *Linear tabling* maintains instead a single execution tree without requiring suspension and resumption of sub-computations. The computation of the (local) fixpoint is performed by failing on branches which loop and reexecuting them when there is an answer (obtained from some other branch) for the looping goal, until no more solutions are found. Examples of this method are the linear tabling of B-Prolog [12,13] and the DRA scheme [14]. Suspension-based mechanisms achieve very good performance but, in general, they need more memory, used to freeze consumer states and to save answers to be reused. Linear mechanisms, on the other hand, do not use memory to freeze computations, but their efficiency is affected by subgoal recomputation.

The most successful suspension-based implementations of tabling are based on trail management. In these, suspension and resumption operations, which allow stopping the execution in a part of the search tree and restarting it in a different node, use the regular trail and/or a forward trail to record the bindings made in the execution path between two nodes (saved as choicepoints in the corresponding stack) and to remember which bindings have to be reinstalled. This technique can perform speculative work if the bindings which are reinstalled are not used.

In order not to incur in the possible overheads stemming from reinstalling bindings which are not going to be used, we propose an implementation based on using variables with multiple bindings (*Multi Value Binding* variables). A global flag indicating which consumer is active at each moment is used as a *key* to retrieve, from a MVB, the value corresponding to that consumer. Therefore, switching to a consumer is a constant-time operation, triggered by giving this global flag the appropriate value. In turn, and in our current implementation, variable access is not constant-time any more.[1] Herein, we present and evaluate an implementation of this idea.

2 Tabling and Variable Management

We start by providing a brief introduction to tabling. Due to space limitations several details of the implementation of tabling based on suspension are not discussed. For a more complete description, the reader is referred to [9,11,15].

Tabling Basics: Tabling changes the operational semantics for predicates marked with the *`:- table`* declaration. The compiler and runtime system distinguish the first occurrence of a goal marked as tabled (the *generator*) and subsequent variant calls (the *consumers*). The generator applies resolution using the program clauses to derive answers for the goal. When a call identical to a previous one is found,[2] the consumer *suspends* the current execution path (using implementation-dependent means) and starts execution on a different branch.

When an alternative branch finally succeeds, the answer generated for the initial query is inserted in a table associated with the original goal. This makes it possible to reactivate suspended calls and to continue execution at the point

[1] We are not taking into account the dereferencing cost here, assuming instead that it is a constant-time operation which was already present in the system.

[2] Which would enter an infinite loop in SLD resolution.

where they were stopped. Thus, consumers do not use SLD resolution, but obtain instead the answers from the table where they were previously inserted by the generator. Predicates not marked as tabled are executed according to SLD resolution, hopefully with minimal overhead due to the availability of tabling. The overall process can be seen graphically as the ability to suspend execution in a part of the tree which cannot progress (because it enters a loop) and continue it somewhere else, where a solution for the looping goal can be produced.

Tabling Implementations Based on Trail Management: We use the CHAT [11] approach in order to illustrate suspension-based techniques which try to fully reinstall consumer environments and show that that can incur in costs due to speculative work. We have chosen CHAT because of its simplicity and because we think that the improved version of CHAT presented in [16] is among the most efficient tabling implementations based on trail management.

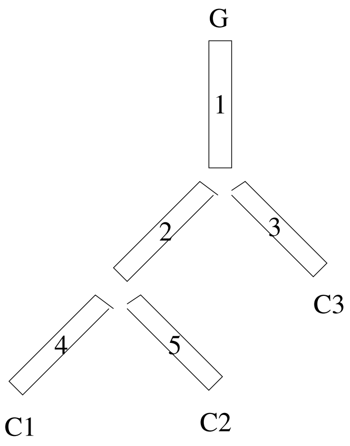


Fig. 1. Sharing trail in CHAT

CHAT implements suspension by freezing (i.e., protecting by updating pointers) the heap and the local stack and saving the consumer choice points to be reinstalled when the consumer is to restart. Consumers keep track of their conditional bindings (i.e., those bindings appearing in the part of the trail between the consumer and its *leader* generator[3]) to enable them to be reinstalled later on when resuming. Using a tree structure as conditional binding storage (Figure 1), each of these bindings is saved only once, although they are shared between several consumers: the trail of consumer `C1` is composed of segments 1, 2 and 4, but segment 2 is shared with consumer `C2` and segment 1 is shared also with consumer `C3`. The conclusion is that CHAT, just as SLG-WAM, performs very well with respect to memory usage because it shares all that can be shared and no bindings are saved twice (see [16] for a detailed explanation of CHAT trail management). There is, however, a speculative component of work in CHAT and in all tabling implementations based on trail management. When a consumer restarts, all of its conditional bindings are reinstalled in the stacks. However, in general not all of these bindings will be necessary in the rest of the execution (some of them might not even be visible at that point). Reinstalling them is, therefore, wasted work. As an extreme example, consider the

[3] A leader is the generator which marks the following completion point of a consumer. Originally, the leader of a consumer is its generator, but it can change to previous generators if the consumer generator cannot be directly completed because of dependencies.

case of a consumer that has a (large) number of conditional bindings, which are reinstalled when the consumer is restarted. If the consumer fails immediately after that, these bindings will never be accessed and will instead be immediately untrailed.

3 MVB Tabling

One of the aims of MVB-based tabling is to avoid the speculative work of systems based on trailing. To this end, we have defined a new kind of variable (assigning a new tag, MVB, to it), which can keep several bindings at once. Depending on a global flag, which we will name the `sid` (from *Suspension Identifier*), the correct binding of a variable is accessed. Therefore, dereferencing a variable can, under this approach, be seen as a function:

$$\mathit{Deref} : \mathit{Variable} \times \mathit{SuspensionId} \longrightarrow \mathit{Value}$$

which can retrieve the value a given variable had when a suspension was performed if we provide the identifier of such a suspension.

We now present more details on how this suspension identifier is managed, and how the multi-value binding variables are accessed and kept up to date.

The Management of the Suspension Identifier: the value of the `sid` global flag associated with a normal SLD execution is *zero*. Whenever a consumer appears, a new `sid` is associated with it by incrementing a global counter, `last sid`. When a generator completes, `last sid` and the `sid` global flag are reset to the value they had at the moment in which the generator was created —i.e., a sort of backtracking is performed on the `last sid` when completing generators.

With this scheme, resuming a consumer boils down to changing the value of `sid` to the identifier of the suspension associated with the consumer. Consequently, the bindings accessed through MVB variables will correspond to the bindings existing when the consumer was suspended.

Which Variables are MVB? The variables which have to maintain different bindings for each consumer are those appearing in the trail between a consumer and its *leader*, because these bindings would otherwise be lost on backtracking. These bindings are associated with the suspension identifier of the consumer which suspends. For the *zero* suspension identifier the corresponding variables behave as normal WAM variables, and they will be unbound on backtracking.

How is an MVB Variable Implemented? The implementation of MVB variables is orthogonal to the idea behind them. In our current implementation, MVB variables keep their bindings using a list relating `sid`s with variable values. Traversing this list is necessary in order to determine the value a variable has for some suspension, which makes accessing the value of a variable a non-constant time operation. The (faster) alternative of using an array indexed by suspension identifier may need reserving too much memory in advance. Other possibilities, such as hash tables, are under consideration (and are discussed further in Section 5).

It is important to remark that the suspension identifiers for bindings inserted in the list associated with an MVB are increasing and consecutive, because they correspond to consumers which have been created on or after the initial creation of the variable. Not all (conditional) variables will necessarily be bound to different values in different suspensions. Additionally, a binding of an MVB can be seen by several consumers, for the same reason that a segment of trail can be shared between different consumers in CHAT. Therefore, we actually compress somewhat the list representation. Instead of a single suspension identifier per node in the list, each binding is associated with a pair of `sid`s which represent a range of suspensions for which the variable had the same value. Thus, each of the elements of the list of an MVB variable is a tuple of the form `<bind,first,last>`, where `bind` is the binding for the variable for the `sid`s between `first` and `last`, both included.

To reduce the impact of non-constant variable access, an MVB variable is equipped with a *cache* where the last accessed value and the range of suspension identifiers for which it is valid are stored. If retrieving from the cache fails (because the `sid` looked up is not available), the binding is searched for in the list and the cache is updated. In addition, the list is kept sorted from the most recent suspensions, with the highest `sid`, to older suspensions, with a lower `sid`. Whenever a new binding is added to the list, its associated `sid` is necessarily greater than all the `sid`s associated with existing bindings, because consumers are generated in a sequential order, and therefore it just has to be added to the front of the list. Keeping the list sorted improves the efficiency of lookups (see later a description of how lookups are done).

If an MVB variable is accessed with a `sid` which does not appear in the MVB list, the cache is updated to point to a new free value, which allows considerable memory savings, since the elements of the list will always be bindings. If that free variable is bound and a suspension is performed later, those bindings will generate a new element of the MVB list.

This MVB representation is illustrated in Figure 6. For example, in heap stack number 5 variable B has created an MVB variable pointing to the cache (which is a free variable associated with `sid` *zero*) and then the value 3 associated with the $\langle 2, 2\rangle$ range of `sid`s and the value 2 associated with the $\langle 1, 1\rangle$ range.

Suspensions Nested Inside Resumptions: Assume that `consumer A` is restarted and `consumer B` appears in this restarted execution. There might be MVB variables of `consumer A` which are associated with the `sid` of `consumer A`. If `consumer B` is restarted, some of these variables could be accessed. The right binding for `consumer B` is the same as the binding for `consumer A`, but the `sid` associated with `consumer B` does not belong to that MVB variable and a free variable would be returned instead.

To solve this problem the *dependence* between `consumer A` and `consumer B` has to be remembered. Since a consumer can depend directly on at most another consumer, it is enough to have a single field per consumer to record this dependency. This dependency is registered when suspending, and the `sid` of the consumer which suspends is made to depend on the `sid` of the consumer within

```
Term accessMVB(Ref var) {
  int sidAux = sid;
  <value, first, last> = cache(var);
  if (sidAux in [first,last]) return value;
  <value,first,last> = firstElem(var);
  do {
     if (sidAux in [first,last]) {
        if (sid == sidAux) updateCacheMVB(var,<value,first,last>);
        else updateCacheMVB(var,<value,sid,sid>);
        return value;
     }
     if (sidAux > last) {
    sidAux = getDependence(sidAux)
    if (!sidAux) {
           updateCacheMVB(var,<free_var,sid,sid>);
           return free_var;
    }
     }
     else <value,first,last> = nextElem(var);
  } while (hasMoreElem(var));
}
```

Fig. 2. Pseudo-code for MVB access

whose resumption the new consumer has appeared. A `sid` can of course depend on other `sid`s transitively.

As `sid`s are generated in sequential order, a suspension can only depend on another previous suspension, which would have a smaller `sid`. Therefore when the MVB list is accessed, if the `sid S` we are looking for is greater than the `last sid` of the element of the MVB list under inspection (which means that it is not going to appear in that list) we can search for the `sid` which `S` depends on, starting at that point in the list. Thanks to the order of `sid`s in the list and the ordering between dependencies, the MVB variable is traversed at most once, even if several dependencies are followed. The code for MVB variable access is shown in Figure 2. Note that we do not advance to the next element if there is a `sid` dependency, because the same element should be inspected again.

When are MVB Variables Created? MVB variables could be created when suspending a goal, by examining which variables reachable from that goal are conditional and have been recorded in the trail. However, this in the end needs to "simulate" backtracking by traversing the choicepoints and the associated entries in the trail. Since this is going to be done anyway on backtracking, we have decided to actually create them when backtracking from the consumer choicepoints. This saves also work when a binding is shared between several consumers, because regular variables in the shared part are converted into MVB variables only by one of them. This is in some sense similar to trail sharing (see Section 2) in CHAT.

```
if (!TrailYounger(trailPointer, top(MVBstateStack)) {
    createOrUpdateMVB(trailPointer);
    trailPointer = trailPointer − 1;
    top(MVBstateStack) = trailPointer;
    if (top(MVBstateStack) == pre_top(MVBstateStack)) pop(MVBstateStack);
}
else Untrail(trailPointer);
```

Fig. 3. Pseudo-code for MVB untrail

A possible solution is to use a special type of backtracking for the choicepoints associated with tabled execution. When a consumer suspends, the choicepoints between that consumer and the generator have to be marked to reflect the initial suspension they belong to. This is done by scanning the choicepoint stack from the topmost choicepoint (i.e., the one corresponding to the suspending consumer) until an already marked choicepoint (which belongs to the execution of another suspended consumer) is found. On backtracking, MVB variables would have to be created and associated with a range of suspensions. The `sid` range associated with these variables is the one which goes from the mark associated with the choicepoint where that variable was trailed to that of the last consumer which suspended. This is, however, difficult to implement in systems which discard choicepoints just before the last alternative is taken (this includes our implementation platform, Ciao [17]): in such cases the bindings of such an alternative would be wrongly associated with the `sid` of the previous choicepoint `sid` mark. Non-trivial changes to the stack management have to be made to avoid this (i.e., not removing a choicepoint when the last alternative is taken), and a memory optimization will be lost.

Our solution is implemented in the untrail operation, and the idea is to use an additional stack (the `MVBstateStack`) which will implement a mechanism similar to the choicepoint-based one, but where the `MVBstateStack` keeps track only of the fields which would otherwise go in the choicepoints. This makes it possible to make these fields survive choicepoint removal without having to fiddle around with the choicepoint management and keeping its "last alternative optimization."

Each time a consumer suspends, the pointer to the top of the trail is pushed onto the `MVBstateStack`, and it is associated with the `sid` of the consumer which suspends. When performing untrailing, if the trail pointer is equal to the value of the topmost element of `MVBstateStack`, we create an MVB variable (or insert a new binding in the MVB list if it was already created) and the current binding is associated with the suspension range from the `sid` corresponding to the topmost element of `MVBstateStack` and the last consumer which suspended, `last sid`. The trail pointer stored at the top of the `MVBstateStack` is then decremented, and if it is equal to the previous element in `MVBstateStack`, it is popped out because the following value to be untrailed is also shared with previous consumers.

The code of the new untrail operation is shown in Figure 3. This can obviously be made more efficient, but we are showing a simple version for clarity.

When Are MVB Variables Removed? When a generator is completed, none of the MVB variables (or, more precisely, none of their bindings) created within its execution are needed any longer. As we want to interfere as little as possible with the existing backtracking / untrailing mechanism, each time an MVB variable is created (or updated) it is recorded in a list associated with the last generator. On generator completion, the bindings added under the subtree of that generator are removed. If all of the elements of the MVB list are removed, the MVB variable is made a regular and free variable again.

Freezing Stacks: The way choice points, the heap, and the local stack are managed in the MVB approach is just the same as in CHAT with the improvements in [16]. The main difference is the management of the trail. CHAT saves the values pointed to from the trail of each consumer to reinstall them when resuming, and MVB tabling uses MVB variables to do that.

Changes to the Prolog Virtual Machine: The changes to be made to a Prolog engine are a special untrail operation (Figure 3), an additional case for the dereferencing routine in order to make it understand MVB variables[4] (Figure 2), and the changes needed to freeze stacks *à la* CHAT. In the case of Ciao, a WAM instruction also has to be modified: `get_first_local_value` gives the first value to a local variable. As it just checks if the local variable was or not initialized to a stack variable, an MVB variable living there could be overwritten. The new `get_first_local_value` instruction checks if the previous value is an MVB variable to make the assignment without destroying the MVB information. In that case the assignment is stored in the cache of the MVB variable.

These changes are in our experience quite local and easy to do, which allows us to conclude that MVB tabling-based implementation is not hard.

4 MVB Tabling Execution

We will try to illustrate the MVB tabling approach presented previously using a simple tabled program (Figure 4). The tabled execution tree of this program (not specifically for MVB tabling) is shown in Figure 5, and Figure 6 shows the creation of MVB variables. Figure 7 shows the management of the `MVBstateStack` to create MVB variables to be shared between several consumers.

We start with the query `p(X)`. Execution starts with a global `sid` of *zero*. `A` and `B` are created as unbound variables in Figure 6 (1), and they are unified with `A = 1` and `B = 2` (Figure 6 (2)) before the execution is suspended because a consumer is found (step 3). The suspension identifier associated with this consumer is `sid` = 1, and `last sid` is updated to be 1.

Figure 7 (1) shows the entries in the trail for `A,B` and the record to be inserted in `MVBstateStack`. Each consumer inserts a pair `<trail_pointer, consumer_sid>` in the `MVBstateStack` —in this case, it is the pair `<2,1>`.

Execution fails then and backtracks over the last choice point (2). Since the `trail pointer` is equal to the value of the trail stored at the top of the `MVBstateStack`, an MVB variable is created and associated with the range from

[4] Which in our case are marked with a special tag.

```
:- table p/1.

p(X) :-
    A = 1,
   (B = 2; B = 3),
    p(X),
    A = B.

p(1).
```

Fig. 4. Tabled program

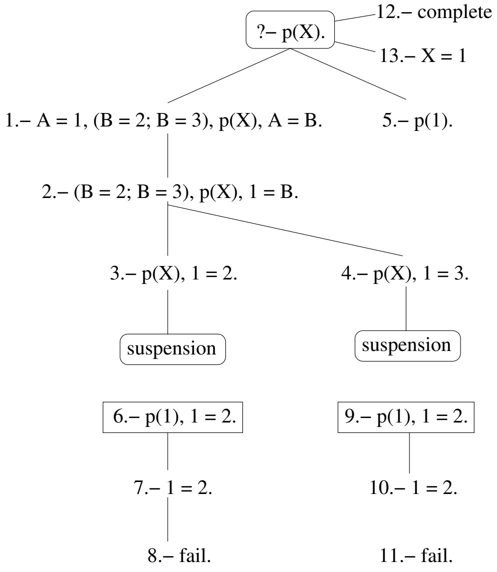


Fig. 5. MVB tabling execution

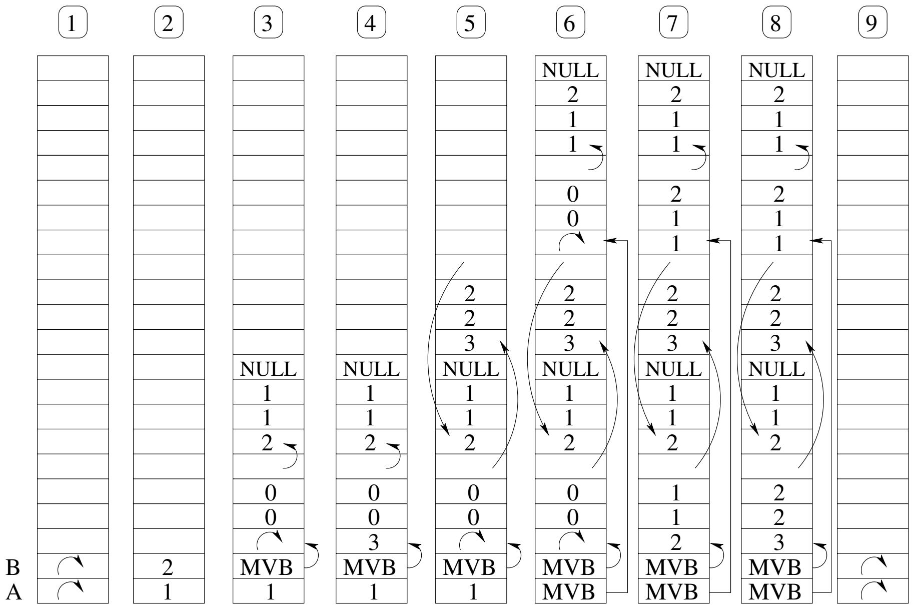


Fig. 6. MVB variables

the `sid` of the topmost element of `MVBstateStack` to the `last sid` variable, which is 1 (Figure 6 (3)). Besides, the value of the trail pointer in the topmost element of `MVBstateStack` is decremented (Figure 7 (2)). Recall that free

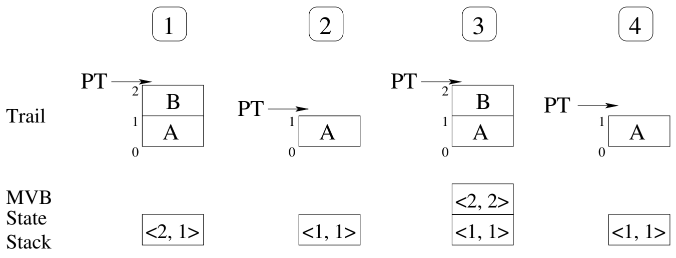


Fig. 7. MVB trail management

variables are created in the cache associated with `sid` *zero*. This corresponds to the (unbound) variable which would have been restored on backtracking. The previous binding is maintained as an MVB list element.

After backtracking, `B = 3` (step 4) is executed and the cache of the MVB variable is updated (Figure 6 (4)). A new consumer is found which is associated with the `sid` 2, and `last sid` is updated to be 2. A new element is inserted in `MVBstateStack`, `<2,2>`, which represents the current `trail pointer` and the new consumer `sid` (Figure 7 (3)). Execution fails and backtracks over the second clause of `p/1`. When `B` is untrailed, its MVB variable is updated to the value `B = 3` and it is associated with the `sids` (2,2) (Figure 6 (5)), inserted in descending order, as explained in Section 3.

To know the `sids` that **B** is associated with, the topmost item of `MVBstate-Stack` is used. When the trail value stored there is decremented, it reaches the same value as the previous element in `MVBstateStack`, and the top of the stack is popped out. (Figure 7 (4)). Now, variable `A` is untrailed, and a new MVB variable is created (Figure 6 (6)), but it is associated with `sids` (1,2) because `last sid` is 2 and the topmost element of `MVBstateStack` represents `sid` 1.

The second clause of `p/1` is executed and the first answer, `p(1)`, is found (step 5). It is inserted in an external table (as all tabling implementations do), and then consumers can be restarted with this answer. In order to do that, the `sid` variable is updated to be the `sid` associated with the consumer being restarted: for example, when consumer 1 is restarted (step 6) and variable `A` is accessed, the cache of its MVB variable is updated with the value associated with `sid` 1, and the same for variable `B` (Figure 6 (7)). When consumer 2 is restarted (step 9), variable `A` is accessed in the cache, but the cache of variable `B` has to be updated (Figure 6 (8)). Finally, the generator `p(X)` can be completed (step 12) and all of the MVB variables created under its execution tree are unbound (Figure 6 (9)). The `sid` and `last sid` variables are updated to the value just before the generator execution (*zero*, in this case) and the answers found and stored in the answer table are returned on backtracking (step 13).

5 Performance Evaluation

In the following two sections we will analyze the performance of our MVB scheme and CHAT, both from a theoretical and an experimental point of view.

5.1 CHAT and MVB — A Conceptual Comparison

In any realization of CHAT and MVB, some essential operations will remain largely untouched regardless of the implementation details. We will focus on these to compare CHAT and MVB at a high abstraction level.

MVB and CHAT freeze the heap and local stacks using the same techniques, and they store consumer choice points and answers using also a similar approach; therefore, their memory consumption should be similar as well. Even the cost of saving the trail of a consumer can be comparable with the cost of creating the MVB variables of such a consumer. Memory consumption should be in the same order too: for each trailed value, CHAT uses 2 slots (value and pointer), and MVB uses 4 slots (value, initial state, last state, and a pointer to the next list element). In our view, the main differences between both approaches are:

– CHAT reinstalls (speculatively) the conditional bindings of consumers.
– MVB variable access is affected by MVB variable indirection.

Although we are using an MVB cache, sometimes the MVB list has to be traversed to find the right MVB bindings for the current `sid`.[5] Consequently, MVB should outperform CHAT when the speculative work of reinstalling the conditional bindings of a consumer is larger than the cost of the overhead due to the MVB variable access, which we will experimentally measure in Section 5.2 for a number of common benchmarks.

This means that *artificial* code can be created where large parts of the trail are saved in a consumer to be later reinstalled and not really used. Therefore MVB could be arbitrarily better than CHAT for such an example. On the other hand, similarly artificial code can also be written where MVB variables have a large amount of bindings and they continuously suffer from cache misses. In this case, MVB could perform arbitrarily worse than CHAT. The question therefore remains: in practice, and assuming similarly involved implementation techniques for, e.g., CHAT and MVB, how much does MVB variable handling (including dereferencing), the special untrailing, and the additions to backtracking affect both tabled and regular (SLD) execution.

5.2 Experimental Evaluation

We have implemented the techniques proposed in this paper as an extension of the Ciao system [17]. All of the timings and measurements have been made with Ciao-1.13, using the standard, unoptimized bytecode-based compilation, and with the `MVB` extensions loaded. For XSB we have used XSB 3.1. All the executions were performed using local scheduling and disabling garbage collection. We used `gcc 4.1.1` to compile all the systems, and we executed them on a machine with Debian Linux 5.0, kernel 2.6.18, and an Intel Xeon DESCHUTES processor.

[5] A more efficient mechanism for accessing variable bindings is possible (Section 6), but practical experiments make us doubt about its real usefulness (Section 5.2).

Impact of MVB on SLD Execution: A question to ask is to what extent the changes we have introduced in the Prolog machinery (e.g., special trailing, extra cases in dereferencing, changes in one WAM instruction) impact the speed of non-tabled execution. We have measured this using the ECRC set of analytical benchmarks[6] which test different characteristics of Prolog execution using dedicated benchmarks. In our experiments, enabling or disabling the MVB extensions did not have any measurable impact on SLD execution speed.

Impact of MVB on Variable Access: We have measured also to what extent having to traverse a list of bindings (even with the improvement of a cache for the most recently accessed value —in the case of a cache hit the value is accessed in constant time) can impact accessing a given value for a consumer. This is difficult to predict as it depends, for each benchmark and variable, on how many conditional bindings for that variable are made by the different consumers.

To measure this, we have instrumented our implementation to count the number of value cache misses, the percentage of cache misses with respect to the total accesses, the average length of the chain of values, and the average number of items traversed in this list for each cache miss.[7] The statistics are shown in Table 1. The main conclusions we can draw are: even if the number of consumers is in principle unknown and can be very large, value chains are usually rather small, which suggests that an implementation with direct indexing may not in the end bring large advantages. Moreover, the benchmark with the longest value chains (**sgm**) has as well the best cache behavior: only 1% of the accessed values were not in the cache. The cache behavior is in general reasonable in the rest of the benchmarks as well.

Table 1. Some statistics on the dynamic behavior of MVB variables

Measure	**sgm**	**atr2**	**pg**	**kalah**	**gabriel**	**disj**	**cs_o**	**cs_r**	**peep**	**Avg.**
Cache misses	1504	2545	147	100	155	103	33	54	335	—
Cache misses (%)	1%	7%	16.6%	17.8%	9%	9.5%	6.5%	5.3%	5.7%	8.7%
Avg. MVB length	30.6	1	3.5	1.4	1.8	1.3	1.2	1.2	2.1	4.9
Avg. list trav.	15.8	1	11.5	1.3	3.8	2.5	1.8	1.7	5.4	5

General Performance Assessment: Table 2 aims at determining how the proposed implementation of tabling compares with other tabling implementations. To that end we have implemented CHAT tabling in Ciao, in order to have a system with a comparable base speed and a similar code maturity. We are also comparing with XSB, arguably the most successful tabling system based on trailing. We have used a set of benchmarks which appear in other performance evaluations of tabling approaches.

[6] Available as a Ciao Prolog 1.13 library and also at the URL `http://www.cs.cmu.edu/afs/cs.cmu.edu/project/ai-repository/ai/lang/prolog/code/library/`

[7] This can be larger than the average list length because value searches can concentrate on lists with lengths over the average.

In this table we provide, for several benchmarks, the raw time (in milliseconds) taken to execute them using tabling. Since these benchmarks do not create large memory structures, the differences between CHAT and MVB are not as large as could be using bigger benchmarks. Expanding this assessment is part of our future work. Besides, the code quality of both implementations (CHAT and MVB) can still be improved, as well as that of the consumer scheduler, MVB representation, etc. However, in general we believe that these two implementations are at a similar level of maturity and should be comparable in terms of speed. Of course any improvement in them would bring a competitive advantage with respect to XSB.

Table 2. MVB vs. CHAT vs. XSB

Program	MVB	CHAT	XSB
sgm	1806	1905	1649
atr2	339.2	353.4	351.0
pg	13.11	13.20	12.03
kalah	19.23	18.82	16.77
gabriel	19.83	19.39	18.42
disj	15.19	15.12	14.02
cs_o	29.18	29.28	25.30
cs_r	58.19	57.80	51.03
peep	60.01	59.20	52.10

The results are in general quite encouraging: speed results are very similar both to CHAT and to XSB, which arguably makes the technique competitive. This provides confidence that an improved implementation (for example, tabling primitives are not yet compiled into WAM code and still have to traverse the Prolog-C interface as in [15]), the internal representation for MVB can be improved, and goal scheduling is still simplistic and does not try to favor our technique by decreasing the probability of cache misses) can make MVB a viable technique for tabling which does not need very complicated stack management and which can compete with state-of-the-art systems.

6 Tabling and Implementation Techniques for Or-Parallelism

The basic problem of Or-parallel systems is “how to represent different bindings of the same variable corresponding to different branches of the search space” [18]. This is of course a concern shared with suspension-based systems for tabling, where suspending and resuming a goal basically has to resort to a representation which makes it possible to save and recover bindings existing at some other part of the search tree. This has been recognized in early work [19]. The similarity between Or-Parallel and tabling using complex trail and stack management (e.g., implementations of the SRI model [20] and SLG-WAM) and those relying on copying (e.g., CHAT and the MUSE [21]) have been mentioned elsewhere [11].

However, to the best of our knowledge, variable access has remained largely untouched in all tabling systems, when, from an abstract point of view, making a variable access different values depending on the environment (e.g., the `sid` global flag) which the variable is seeing is a fundamental operation. This has been tackled by installing as a “solid block” all the bindings a consumer has to see, instead of using a *switch* to change the viewpoint of the consumer. This is precisely what systems such as Aurora [22] did —in that case by maintaining variables as indexes on a binding array with different entries for each processor.

This is not radically different from out approach. In both cases the function in Section 3 is implemented. However, Aurora first discriminates on the second function argument (to select the worker) and then on the first (to select the variable). In our case we take first the variable to dereference, and then the consumer inside whose environment it has to be evaluated; that matches what the Version Vectors model [23] does. While, as mentioned before, the relation between Or-Parallelism and tabling has been studied before, we believe that it is still possible to establish further connections which can bring more implementation techniques and, e.g., scheduling algorithms developed in the realm Or-Parallelism to tabled systems, and thus also make new implementations possible which seamlessly exploit these relationships.

7 Conclusions

We have presented and evaluated an implementation technique for tabling based on keeping simultaneously several bindings for variables (MVB), corresponding to the environments of the consumers. From the experiments we can conclude that, while theoretically MVB can be arbitrarily better or worse than CHAT, in practice it is a viable way of avoiding some speculative work inherent to trailing-based implementations of tabling. Although our implementation can still be improved in several directions, the performance we obtain is already acceptable and comparable with state-of-the-art systems. Finally, we have revisited the similarity between OR-parallelism and suspension-based implementations of tabling, and conclude that some additional cross-fertilization is probably still possible.

Acknowledgments. This work was funded in part by EU projects FET IST-15905 *MOBIUS*, IST-215483 *SCUBE*, FET IST-231620 *HATS*, and 06042-ESPASS, MICINN projects TIN-2008-05624 *DOVES*, TIN2005-09207-C03 *MERIT-COMVERS*, MIN project FIT-340005-2007-14, and CM project S-0505/TIC/0407 *PROMESAS*. Pablo Chico is also funded by an FPU MICINN scholarship.

References

1. Warren, D.S.: Memoing for logic programs. Communications of the ACM 35(3), 93–111 (1992)
2. Chen, W., Warren, D.S.: Tabled Evaluation with Delaying for General Logic Programs. Journal of the ACM 43(1), 20–74 (1996)
3. Tamaki, H., Sato, M.: OLD resolution with tabulation. In: Wada, E. (ed.) Logic Programming 1986. LNCS, vol. 264, pp. 84–98. Springer, Heidelberg (1987)
4. Ramakrishnan, R., Ullman, J.D.: A survey of research on deductive database systems. Journal of Logic Programming 23(2), 125–149 (1993)
5. Warren, R., Hermenegildo, M., Debray, S.K.: On the Practicality of Global Flow Analysis of Logic Programs. In: Fifth International Conference and Symposium on Logic Programming, pp. 684–699. MIT Press, Cambridge (1988)

6. Dawson, S., Ramakrishnan, C.R., Warren, D.S.: Practical Program Analysis Using General Purpose Logic Programming Systems – A Case Study. In: Proceedings of PLDI 1996, pp. 117–126. ACM Press, New York (1996)
7. Zou, Y., Finin, T., Chen, H.: F-OWL: An Inference Engine for Semantic Web. In: Hinchey, M.G., Rash, J.L., Truszkowski, W.F., Rouff, C.A. (eds.) FAABS 2004. LNCS, vol. 3228, pp. 238–248. Springer, Heidelberg (2004)
8. Ramakrishna, Y., Ramakrishnan, C., Ramakrishnan, I., Smolka, S., Swift, T., Warren, D.: Efficient Model Checking Using Tabled Resolution. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 143–154. Springer, Heidelberg (1997)
9. Sagonas, K., Swift, T.: An Abstract Machine for Tabled Execution of Fixed-Order Stratified Logic Programs. ACM Transactions on Programming Languages and Systems 20(3), 586–634 (1998)
10. Demoen, B., Sagonas, K.: CAT: The Copying Approach to Tabling. In: Palamidessi, C., Meinke, K., Glaser, H. (eds.) ALP 1998 and PLILP 1998. LNCS, vol. 1490, pp. 21–35. Springer, Heidelberg (1998)
11. Demoen, B., Sagonas, K.: CHAT: the copy-hybrid approach to tabling. Future Generation Computer Systems 16, 809–830 (2000)
12. Zhou, N.F., Shen, Y.D., Yuan, L.Y., You, J.H.: Implementation of a linear tabling mechanism. J. of Functional and Logic Programming 2001(10) (October 2001)
13. Zhou, N.F., Sato, T., Shen, Y.D.: Linear Tabling Strategies and Optimizations. Theory and Practice of Logic Programming 8(1), 81–109 (2008)
14. Guo, H.F., Gupta, G.: A Simple Scheme for Implementing Tabled Logic Programming Systems Based on Dynamic Reordering of Alternatives. In: International Conference on Logic Programming, pp. 181–196 (2001)
15. de Guzmán, P.C., Carro, M., Hermenegildo, M.: Towards a Complete Scheme for Tabled Execution Based on Program Transformation. In: Gill, A. (ed.) PADL 2009. LNCS, vol. 5418, pp. 224–238. Springer, Heidelberg (2009)
16. Demoen, B., Sagonas, K.: CHAT is θ(SLG-WAM). In: Ganzinger, H., McAllester, D., Voronkov, A. (eds.) LPAR 1999. LNCS, vol. 1705, pp. 337–357. Springer, Heidelberg (1999)
17. Bueno, F., Cabeza, D., Carro, M., Hermenegildo, M., López-Garcí, P., Puebla, G.: The Ciao System. Ref. Manual (v1.13). Technical report, C. S. School, UPM (2006), http://www.ciaohome.org
18. Warren, D.H.D.: OR-Parallel Execution Models of Prolog. In: Ehrig, H., Levi, G., Montanari, U. (eds.) CAAP 1987 and TAPSOFT 1987. LNCS, vol. 249. Springer, Heidelberg (1987)
19. Warren, D.S.: Efficient Prolog Memory Management for Flexible Control Strategies. In: International Symposium on Logic Programming, Silver Spring, MD, Atlantic City, pp. 198–203. IEEE Computer Society, Los Alamitos (1984)
20. Warren, D.H.D.: The SRI Model for OR-Parallel Execution of Prolog—Abstract Design and Implementation. In: Furukawa, K., Fujisaki, T., Tanaka, H. (eds.) Logic Programming 1987. LNCS, vol. 315, pp. 92–102. Springer, Heidelberg (1988)
21. Ali, K., Karlsson, R.: The MUSE Approach to Or-Parallel Prolog. International Journal of Parallel Programming 19(2), 129–162 (1990)
22. Lusk, E., Butler, R., Disz, T., Olson, R., Stevens, R., Warren, D.H.D., Calderwood, A., Szeredi, P., Brand, P., Carlsson, M., Ciepielewski, A., Hausman, B., Haridi, S.: The Aurora Or-parallel Prolog System. New Generation Computing 7(2/3), 243–271 (1988)
23. Hausman, B., Ciepielewski, A., Haridi, S.: Or-Parallel Prolog Made Efficient on Shared Memory Multiprocessors. In: Symposium on Logic Programming, SICS, August 1987, pp. 69–79 (1987)

A Term-Based Global Trie for Tabled Logic Programs

Jorge Costa, João Raimundo, and Ricardo Rocha[⋆]

DCC-FC & CRACS
University of Porto, Portugal
{jcosta,jraimundo,ricroc}@dcc.fc.up.pt

Abstract. A critical component in the implementation of an efficient tabling system is the design of the data structures and algorithms to access and manipulate tabled data. Arguably, the most successful data structure for tabling is tries. However, when used in applications that pose many queries and/or have a large number of answers, tabling can build arbitrarily many and/or very large tables, quickly filling up memory. In this paper, we propose a new design for the table space organization where all terms in tabled subgoal calls and tabled answers are represented only once in a common global trie instead of being spread over several different trie data structures. Our initial experiments using the YapTab tabling system show significant reductions on memory usage without compromising running time.

Keywords: Tabling Logic Programming, Table Space, Implementation.

1 Introduction

Tabling is an implementation technique that overcomes some limitations of traditional Prolog systems in dealing with redundant sub-computations and recursion. Tabling has become a popular and successful technique thanks to the ground-breaking work in the XSB Prolog system and in particular in the SLG-WAM engine [1]. The success of SLG-WAM led to several alternative implementations that differ in the execution rule, in the data-structures used to implement tabling, and in the changes to the underlying Prolog engine. Implementations of tabling are now widely available in systems like Yap Prolog, B-Prolog, ALS-Prolog, Mercury and more recently Ciao Prolog.

A critical component in the implementation of an efficient tabling system is the design of the data structures and algorithms to access and manipulate tabled data. Arguably, the most successful data structure for tabling is *tries* [2]. Tries are trees in which common prefixes are represented only once. The trie data structure provides complete discrimination for terms and permits look up and possibly insertion to be performed in a single pass through a term, hence resulting in a very efficient and compact data structure for term representation.

[⋆] This work has been partially supported by the FCT research projects STAMPA (PTDC/EIA/67738/2006) and JEDI (PTDC/EIA/66924/2006).

Despite the good properties of tries, when used in applications that pose many queries and/or have a large number of answers, tabling can build arbitrarily many and/or very large tables, quickly filling up memory [3]. A possible solution for this problem is to dynamically abolish some of the tables. This can be done by using explicit tabling primitives or by using a memory management strategy that automatically recovers space among the least recently used tables when memory runs out [4]. An alternative approach is to store tables externally in a relational database system and then reload them back only when necessary [5].

A complementary approach to the previous problem is to study how less redundant and more compact data structures can be used to better represent the table space. In this paper, we propose a new design for the table space organization where all terms in tabled subgoal calls and tabled answers are represented only once in a *common global trie* instead of being spread over several different trie data structures. Our approach resembles the *hash-consing* technique [6], as it shares data that is structurally equal, thus saving memory usage by reducing redundancy in term representation. We will focus our discussion on a concrete implementation, the YapTab system [7], but our proposals can be easy generalized and applied to other tabling systems.

The remainder of the paper is organized as follows. First, we briefly introduce some background concepts about tries and the table space. Next, we introduce YapTab's new design for the table space organization using the common global trie and then, we describe how we have extended YapTab to provide engine support for the new design. At last, we present some experimental results and we end by outlining some conclusions.

2 Tabling Tries

The basic idea behind tabling is straightforward: programs are evaluated by storing answers for tabled subgoals in an appropriate data space, called the *table space*. Repeated calls to tabled subgoals[1] are not re-evaluated against the program clauses, instead they are resolved by consuming the answers already stored in their table entries. During this process, as further new answers are found, they are stored in their tables and later returned to all repeated calls.

Within this model, the table space may be accessed in a number of ways: **(i)** to find out if a subgoal is in the table and, if not, insert it; **(ii)** to verify whether a newly found answer is already in the table and, if not, insert it; and **(iii)** to load answers to repeated subgoals. With these requirements, a correct design of the table space is critical to achieve an efficient implementation. YapTab uses *tries* which is regarded as a very efficient way to implement the table space [2].

A trie is a tree structure where each different path through the trie data units, the *trie nodes*, corresponds to a term described by the tokens labelling the nodes traversed. Two terms with common prefixes will branch off from each other at the first distinguishing token. For example, the tokenized form of the term

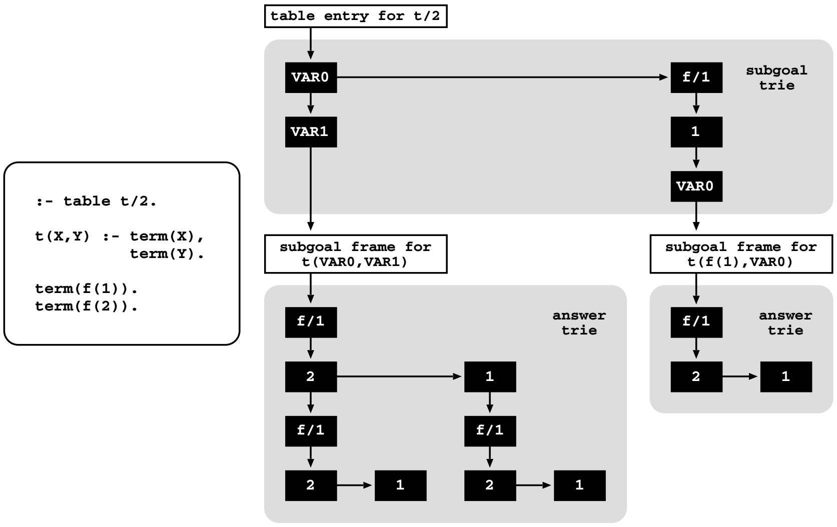
[1] A subgoal repeats a previous subgoal if they are the same up to variable renaming.

$f(X, g(Y, X), Z)$ is the sequence of 6 tokens: $f/3$, VAR_0, $g/2$, VAR_1, VAR_0 and VAR_2, where each variable is represented as a distinct VAR_i constant [8].

To increase performance, YapTab implements tables using two levels of tries: one for subgoal calls; the other for computed answers. More specifically:

– each tabled predicate has a *table entry* data structure assigned to it, acting as the entry point for the predicate's *subgoal trie*.
– each different subgoal call is represented as a unique path in the subgoal trie, starting at the predicate's table entry and ending in a *subgoal frame* data structure, with the argument terms being stored within the path's nodes. The subgoal frame data structure acts as an entry point to the *answer trie*.
– each different subgoal answer is represented as a unique path in the answer trie. Contrary to subgoal tries, answer trie paths hold just the substitution terms for the free variables which exist in the argument terms of the corresponding subgoal call [2]. Repeated calls to tabled subgoals load answers by traversing the answer trie nodes bottom-up.

An example for a tabled predicate `t/2` is shown in Fig. 1. Initially, the subgoal trie is empty. Then, the subgoal `t(f(1),Y)` is called and three trie nodes are inserted: one for functor `f/1`, a second for constant `1` and one last for variable `Y` (`VAR0`). The subgoal frame is inserted as a leaf, waiting for the answers. Next, the subgoal `t(X,Y)` is also called. The two calls differ in the first argument, so tries bring no benefit here. Two new trie nodes, for variables `X` (`VAR0`) and `Y` (`VAR1`), and a new subgoal frame are inserted. At the end, the answers for each subgoal are stored in the corresponding answer trie as their values are computed. Subgoal `t(f(1),Y)` has two answers, `Y=f(1)` and `Y=f(2)`, so we need three trie



```
:- table t/2.

t(X,Y) :- term(X),
          term(Y).

term(f(1)).
term(f(2)).
```

Fig. 1. Original table organization

nodes to represent both: a common node for functor `f/1` and two nodes for constants `1` and `2`. For subgoal `t(X,Y)` we have four answers, resulting from the combination of the answers `f(1)` and `f(2)` for variables `X` and `Y`, which requires nine trie nodes to represent them. Note that, for this particular example, the completed answer trie for `t(X,Y)` includes in its representation the completed answer trie for `t(f(1),Y)`.

3 Common Global Trie

In this section, we describe YapTab's new design for the table space organization. Our new design can be seen as an extension of a previous approach [9], where we first introduced the idea of using a common global trie. In what follows, we will refer to our previous approach as the *Global Trie for Calls and Answers* (GT-CA), and to our new design as the *Global Trie for Terms* (GT-T). Next, we start by briefly introducing the GT-CA approach and then we discuss in more detail how we have extended and optimized it to our new GT-T design.

3.1 Global Trie for Calls and Answers

In the GT-CA approach, all tabled subgoal calls and answers are stored in a common global trie instead of being spread over several different trie data structures. The GT-CA still is a tree structure where each different path through the trie nodes corresponds to a subgoal call and/or answer. However, here a path can end at any internal trie node and not necessarily at a leaf trie node.

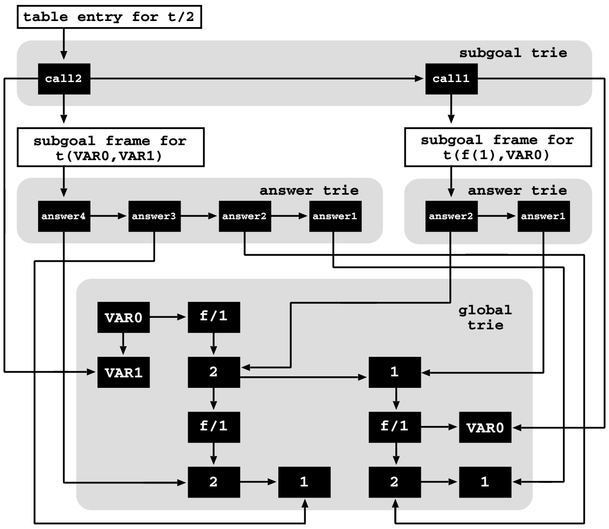


Fig. 2. GT-CA table organization

The original subgoal trie and answer trie data structures are now represented by a unique level of nodes that point to the corresponding paths in the GT-CA (see Fig. 2 for details). For the subgoal tries, each node now represents a different subgoal call where the node's token is the pointer to the unique path in the GT-CA that represents the argument terms for the subgoal call. For the answer tries, each node now represents a different subgoal answer where the node's token is the pointer to the unique path in the GT-CA that represents the substitution terms for the free variables which exist in the argument terms. With this organization, answers are now loaded by following the pointer in the node's token and then by traversing bottom-up the corresponding GT-CA's nodes.

Figure 2 uses again the example from Fig. 1 to illustrate how the GT-CA design works. Initially, the subgoal trie and the GT-CA are empty. Then, the first subgoal `t(f(1),Y)` is called and three nodes are inserted in the GT-CA: one to represent the functor `f/1`, another for the constant `1` and a last one for variable `Y` (`VAR0`). Next, a node representing the path inserted in the GT-CA is stored in the subgoal trie (node labeled `call1`). For the second subgoal call, `t(X,Y)`, we start again by inserting the call in the GT-CA and then we store a node in the subgoal trie (node labeled `call2`) to represent the path inserted in the GT-CA. Each answer is also inserted first in the GT-CA and then we store a node in the corresponding answer trie (nodes labeled `answer1`, `answer2`, `answer3` and `answer4`) to represent the path inserted in the GT-CA.

This example shows us that with the GT-CA we cannot share the representation of common terms appearing at different argument or substitution positions. For example, the terms `f(1)`, `f(2)` and `VAR0` appear more than once represented in the global trie. Moreover, with this example, we can see also that terms in the GT-CA can end at any internal trie node and not necessarily at a leaf trie node. This happens because tabled subgoals calls and answers are not always necessarily *pure* terms. A subgoal call is, in fact, represented by a sequence of argument terms and an answer is, in fact, represented by a sequence of substitution terms. Thus, when the number of argument or substitution terms is greater than one, then we may have situations where a subgoal call or answer can end at internal nodes of other subgoal calls and/or answers. This raises a problem when supporting table abolish operations because the nodes representing an individual subgoal call or answer may not be removed if they belong to other different paths. This problem can be solved by introducing an extra field in each trie node to count the number of paths it belongs to and only allow deletion when it reaches zero, but this solution is contradictory with our goal of saving memory.

Another problem with the GT-CA design is that, on completion of a subgoal, a strategy exists that avoids answer recovery using bottom-up unification and performs instead what is called a *completed table optimization* [2]. This optimization implements answer recovery by top-down traversing the completed answer trie and by executing specific WAM-like code from the answer trie nodes. With the GT-CA design, the nodes in the global trie can belong to several different subgoal/answer tries, and thus this optimization is no longer possible.

We next discuss how we have extended and optimized this table organization to the new GT-T design in order to solve these problems.

3.2 Global Trie for Terms

The GT-T was designed in order to maximize the sharing of tabled data that is structurally equal. In the GT-T design, all argument and substitution terms appearing in tabled subgoal calls and/or answers are represented *only once* in the common global trie. The GT-T still is a tree structure where each different path through the trie nodes represents a *unique* argument and/or substitution term, therefore always ending at a leaf trie node. Each path in a subgoal or answer trie is now composed of a fixed number of trie nodes representing the argument or substitution terms in the corresponding tabled subgoal call or answer. For the subgoal tries, each node now represents an argument term where the node's token is the pointer to the unique path in the GT-T representing the term. For the answer tries, each node now represents a substitution term where the node's token is the pointer to the unique path in the GT-T representing the term.

Figure 3 uses again the example from Fig. 1 to illustrate how the GT-T design works. Initially, the subgoal trie and the GT-T are empty. Next, the first subgoal `t(f(1),Y)` is called and the two argument terms, `f(1)` and `Y` (`VAR0`), are first inserted in the GT-T. Then, the argument terms are represented in the subgoal trie by two nodes (nodes labeled `arg1` and `arg2`), each one pointing to the leaf node of the corresponding term inserted in the GT-T. For the second subgoal

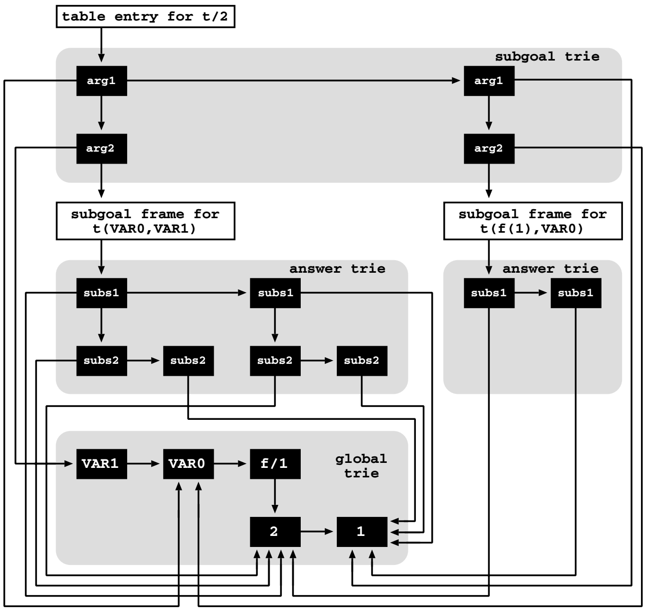


Fig. 3. GT-T table organization

call, `t(X,Y)`, the argument terms `VAR0` and `VAR1` are also inserted first in the GT-T and then we store also two nodes in the subgoal trie, each one pointing to the corresponding representation in the GT-T.

For the answers, each substitution term is also inserted first in the GT-T and then we store a node in the corresponding answer trie to represent its path in the GT-T (nodes labeled `subs1` and `subs2`). The substitution terms for the complete set of answers for the two subgoal calls only include the terms `f(1)` and `f(2)`. Thus, as `f(1)` is already stored in the global trie, we only need to insert `f(2)` in order to be able to represent the full set of answers. As we are maximizing the sharing of common terms appearing at different argument or substitution positions, for this particular example, this results in a very compact representation of the global trie.

Regarding space reclamation, as each different path in the GT-T always ends at a leaf node, we can use the child field (that is always NULL in a leaf node) to count the number of references to the path it represents and only allow deletion when it reaches zero. This solves the previous problem of supporting table abolish operations without introducing extra memory overheads.

Regarding compiled tries, the idea is to keep the global trie only with the term representation and store the WAM-like instructions in the answer tries, as in the original design [2]. The difference is that for the GT-T approach, the WAM-like instructions are more *high-level*, i.e., instead of working at the level of atoms/terms/functors/lists as in [2], each instruction works at the level of the substitution terms. For example, consider again the four answers for the call `t(X,Y)`. When loading these answers, we have two choices for `X` and, for each `X`, we have two choices for `Y`. In the GT-T design, the answer trie nodes representing the choices for `X` and for `Y` (nodes labeled respectively `subs1` and `subs2`) are compiled with a WAM-like sequence such as `try_subs_term` (for the first choices) and `trust_subs_term` (for the second/last choices). GT-T's compiled tries also include a `retry_subs_term` instruction (for intermediate choices) and a `do_subs_term` instruction (for single choices).

4 Implementation Details

We then describe in more detail the data structures and algorithms for YapTab's new table design. We start with Fig. 4 showing in more detail the table organization previously presented in Fig. 3 for the subgoal call `t(X,Y)`.

Internally, tries are represented by a top *root node*, acting as the entry point for the corresponding subgoal, answer or global trie data structure. For the subgoal tries, the root node is stored in the corresponding table entry's `subgoal_trie_root_node` data field. For the answer tries, the root node is stored in the corresponding subgoal frame's `answer_trie_root_node` data field. For the global trie, the root node is stored in the `GT_ROOT_NODE` global variable.

Regarding trie nodes, they are internally implemented as 4-field data structures. The first field (`token`) stores the token for the node and the second (`child`), third (`parent`) and fourth (`sibling`) fields store pointers, respectively,

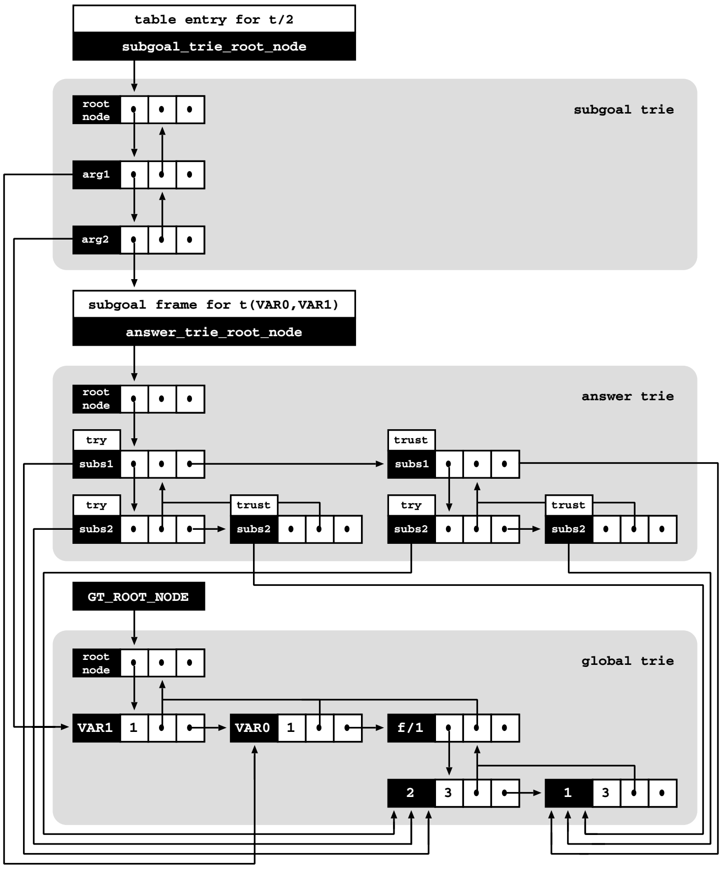


Fig. 4. Implementation details for the GT-T table organization

to the first child node, to the parent node, and to the next sibling node. Remember that for the global trie, the leaf node's `child` field is used to count the number of references to the path it represents. For the answer tries, an additional field (`code`) is used to support compiled tries. Traversing a trie to check/insert for new calls or for new answers is implemented by repeatedly invoking a `trie_node_check_insert()` procedure for each token that represents the call/answer being checked. Given a trie node `parent` and a token `t`, the `trie_node_check_insert()` procedure returns the child node of `parent` that represents the given token `t`. Figure 5 shows the pseudo-code for this procedure.

Initially, the procedure checks if the list of sibling nodes is empty. If this is the case, a new trie node representing the given token `t` is initialized and inserted as the first child of the given `parent` node. To initialize new trie nodes, we use a `new_trie_node()` procedure with four arguments, each one corresponding to the initial values to be stored respectively in the `token`, `child`, `parent` and `sibling` fields of the new trie node. For answer trie nodes, the `code` field is computed later when completion takes place.

```
trie_node_check_insert(TRIE_NODE parent, TOKEN t) {
  child = parent->child
  if (child == NULL) {              // the list of sibling nodes is empty
    child = new_trie_node(t, NULL, parent, NULL)
    parent->child = child
  } else if (not_a_hash_table(child)) {  // sibling nodes without hashing
    sibling_nodes = 0              // to count the number of sibling nodes
    do {          // check if token t is already in the list of siblings
      if (child->token == t) return child
      sibling_nodes++
      child = child->sibling
    } while (child)
    child = new_trie_node(t, NULL, parent, parent->child)
    if (sibling_nodes > MAX_SIBLING_NODES_PER_LEVEL) {  // alloc new hash
      hash = new_hash_table(child)
      parent->child = hash
    } else
      parent->child = child
  } else {                                   // sibling nodes with hashing
    hash = child
    bucket = hash_function(hash, t)    // get the hash bucket for token t
    child = bucket
    sibling_nodes = 0
    while (child) {     // check if token t is already in the hash bucket
      if (child->token == t) return child
      sibling_nodes++
      child = child->sibling
    }
    child = new_trie_node(t, NULL, parent, bucket)
    if (sibling_nodes > MAX_SIBLING_NODES_PER_BUCKET)      // expand hash
      expand_hash_table(hash)
  }
  return child
}
```

Fig. 5. Pseudo-code for the `trie_node_check_insert()` procedure

Otherwise, if the list of sibling nodes is not empty, the procedure checks if they are being indexed through a hash table. Searching through a list of sibling nodes is initially done sequentially. This could be too expensive if we have hundreds of siblings. A threshold value (`MAX_SIBLING_NODES_PER_LEVEL`) controls whether to dynamically index the nodes through a hash table, hence providing direct node access and optimizing search. Further hash collisions are reduced by dynamically expanding the hash tables when a second threshold value (`MAX_SIBLING_NODES_PER_BUCKET`) is reached for a particular hash bucket.

If not using hashing, the procedure then traverses sequentially the list of sibling nodes and checks for one representing the given token `t`. If such a node is found then execution is stopped and the node returned. Otherwise, a new trie node is initialized and inserted in the beginning of the list. If reaching the threshold value `MAX_SIBLING_NODES_PER_LEVEL`, a new hash table is initialized and inserted as the first child of the given `parent` node.

If using hashing, the procedure first calculates the hash bucket for the given token `t` and then, it traverses sequentially the list of sibling nodes in the bucket checking for one representing `t`. Again, if such a node is found then execution is stopped and the node returned. Otherwise, a new trie node is initialized and inserted in the beginning of the bucket list. If reaching the threshold value `MAX_SIBLING_NODES_PER_BUCKET`, the current hash table is expanded.

To manipulate tries we use two interface procedures:

```
trie_load(TRIE_NODE leaf)
trie_check_insert(TRIE_NODE root, TERM t)
```

The `trie_load()` is used to load a term from a trie back to the Prolog engine, where `leaf` is the reference to the leaf node of the term to be loaded.

The `trie_check_insert()` is used for traversing a trie to check/insert for new terms, where `root` is the root node of the trie to be used and `t` is the term to be inserted. It invokes repeatedly the previous `trie_node_check_insert()` procedure for each token that represents the given term `t` and returns the reference to the leaf node representing its path. Note that inserting a term requires in the worst case allocating as many nodes as necessary to represent its path. On the other hand, inserting repeated terms requires traversing the trie structure until reaching the corresponding leaf node, without allocating any new node.

When inserting terms in the table space we need to distinguish two situations: **(i)** inserting tabled calls in a subgoal trie structure; and **(ii)** inserting answers in a particular answer trie structure. The former situation is handled by the `subgoal_check_insert()` procedure as shown in Fig. 6 and the latter situation is handled by the `answer_check_insert()` procedure as shown in Fig. 7.

In the original table design, the `subgoal_check_insert()` procedure simply uses the `trie_check_insert()` procedure to check/insert the given `call` in the subgoal trie corresponding to the given table entry `te`. In the new GT-T design, for each argument term `t`, it first checks/inserts the term `t` in the GT-T and, then, it uses the reference to the leaf node representing `t` in the GT-T (`leaf_gt_node` in Fig. 6) as the token to be checked/inserted in the subgoal trie corresponding to the given table entry `te`. Note that this is done by calling the `trie_node_check_insert()` procedure, thus if the list of sibling nodes in the

```
subgoal_check_insert(TABLE_ENTRY te, SUBGOAL_CALL call, ARGS_ARITY a) {
  if (GT_ROOT_NODE) {                                   // GT-T table design
    st_node = te->subgoal_trie_root_node
    for (i = 1; i <= a; i++) {
      t = get_argument_term(call, i)
      leaf_gt_node = trie_check_insert(GT_ROOT_NODE, t)
      leaf_gt_node->child++      // increase number of paths it represents
      st_node = trie_node_check_insert(st_node, leaf_gt_node)
    }
    leaf_st_node = st_node
  } else                                           // original table design
    leaf_st_node = trie_check_insert(te->subgoal_trie_root_node, call)
  return leaf_st_node
}
```

Fig. 6. Pseudo-code for the `subgoal_check_insert()` procedure

```
answer_check_insert(SUBGOAL_FRAME sf, ANSWER answer, SUBS_ARITY a) {
  if (GT_ROOT_NODE) {                                // GT-T table design
    at_node = sf->answer_trie_root_node
    for (i = 1; i <= a; i++) {
      t = get_substitution_term(answer, i)
      leaf_gt_node = trie_check_insert(GT_ROOT_NODE, t)
      leaf_gt_node->child++     // increase number of paths it represents
      at_node = trie_node_check_insert(at_node, leaf_gt_node)
    }
    leaf_at_node = at_node
  } else                                         // original table design
    leaf_at_node = trie_check_insert(sf->answer_trie_root_node, answer)
  return leaf_at_node
}
```

Fig. 7. Pseudo-code for the `answer_check_insert()` procedure

```
answer_load(ANSWER_TRIE_NODE leaf_at_node, SUBS_ARITY a) {
  if (GT_ROOT_NODE) {                                // GT-T table design
    at_node = leaf_at_node
    for (i = a; i >= 1; i--) {
      leaf_gt_node = at_node->token
      t = trie_load(leaf_gt_node)
      put_substitution_term(t, answer)
      at_node = at_node->parent
    }
  } else                                         // original table design
    answer = trie_load(leaf_at_node)
  return answer
}
```

Fig. 8. Pseudo-code for the `answer_load()` procedure

subgoal trie exceeds the `MAX_SIBLING_NODES_PER_LEVEL` threshold value, then a new hash table is initialized as described before.

The `answer_check_insert()` procedure works similarly. In the original table design, it checks/inserts the given `answer` in the answer trie corresponding to the given subgoal frame `sf`. In the new GT-T design, for each substitution term `t`, it first checks/inserts the term `t` in the GT-T and, then, it uses the reference to the leaf node representing `t` in the GT-T (`leaf_gt_node` in Fig. 7) as the token to be checked/inserted in the answer trie corresponding to the given subgoal frame `sf`. Again, if the list of sibling nodes in the answer trie exceeds the `MAX_SIBLING_NODES_PER_LEVEL` threshold value, a new hash table is initialized.

Finally, the `answer_load()` procedure is used to consume answers. Figure 8 shows the pseudo-code for it. In the original table design, it simply uses the `trie_load()` procedure to load from the answer trie back to the Prolog engine the answer given by the trie node `leaf_at_node`. In the new GT-T design, for each answer trie node `at_node`, now it uses the `trie_load()` procedure to load from the GT-T back to the Prolog engine the substitution term given by the reference (`leaf_gt_node` in Fig. 8) stored in the corresponding `token` field.

5 Experimental Results

We next present some experimental results comparing YapTab with and without support for the common global trie data structure. The environment for our experiments was an Intel(R) Core(TM)2 Quad 2.66GHz with 2 GBytes of main memory and running the Linux kernel 2.6.24.23 with YapTab 5.1.4.

To put the performance results in perspective and have a well-defined starting point comparing the GT-CA and GT-T approaches, first we have defined a tabled predicate `t/5` that simply stores in the table space terms defined by `term/1` facts, and then we used a top query goal `test/0` to recursively call `t/5` with all combinations of one and two free variables in the arguments. We experimented the `test/0` predicate with 10 different kinds of 1000 `term/1` facts: integers, atoms, compound (with arities 1, 2, 4 and 6) and list (with lengths 1, 2, 4 and 6) terms. An example of such code for compound terms of arity 1 is shown next.

```
:- table t/5.
t(A,B,C,D,E) :- term(A), term(B), term(C), term(D), term(E).

test :- t(A,f(1),f(1),f(1),f(1)), fail.                 term(f(1)).
...                                                     term(f(2)).
test :- t(f(1),f(1),f(1),f(1),A), fail.                 term(f(3)).
test :- t(A,B,f(1),f(1),f(1)), fail.                    ...
...                                                     term(f(998)).
test :- t(f(1),f(1),f(1),A,B), fail.                    term(f(999)).
test.                                                   term(f(1000)).
```

Table 1 shows the table memory usage (columns ***Mem***), in MBytes, and the running times, in milliseconds, to store (columns ***Str***) the tables (first execution) and to load from the tables (second execution) the complete set of subgoals/answers without (columns ***Load***) and with (columns ***Cmp***) compiled tries for YapTab using the original table organization (column ***YapTab***), using the previous GT-CA approach (column ***GT-CA/YapTab***) and using the new GT-T design (column ***GT-T/YapTab***). For the GT-CA and GT-T approaches we only show the memory and running time ratios over YapTab's original table organization.

The results in Table 1 suggest that GT-T support is the best approach to reduce memory usage and that this reduction increases proportionally to the length and redundancy of the terms stored in the global trie. In particular, for compound and list terms, the results show an increasing and very significant reduction on memory usage, for both GT-CA and GT-T approaches. The results for integer and atoms terms are also very interesting as they show that the cost of representing only atomic terms in the global trie (around 8% for GT-CA and 0% for GT-T in these experiments) can be manageable when we increase redundancy. Note that integers and atoms terms are represented by a single node in the original YapTab design, and by an extra node (therefore requiring two nodes) if using a global trie.

Regarding running time, these results seem to indicate that memory reduction comes at a price in storing time (around 25% for GT-CA and 7% for GT-T in these experiments). Note that with GT-CA and GT-T support, we pay the cost of navigating in two tries when checking/storing/loading a term. Moreover, in

Table 1. Table memory usage (in MBytes) and store/load times (in milliseconds) for YapTab with and without support for the common global trie data structure

Terms	***YapTab***				***GT-CA/YapTab***				***GT-T/YapTab***			
	Mem	***Str***	***Load***	***Cmp***	***Mem***	***Str***	***Load***	***Cmp***	***Mem***	***Str***	***Load***	***Cmp***
1000 ints	191	1009	358	207	1.08	1.56	1.30	n.a.	1.00	1.32	1.18	1.69
1000 atoms	191	1040	337	231	1.08	1.54	1.41	n.a.	1.00	1.26	1.24	1.54
1000 f/1	191	1474	548	239	1.08	1.35	1.33	n.a.	1.00	1.28	1.11	1.88
1000 f/2	382	1840	632	353	**0.58**	1.25	1.37	n.a.	**0.50**	1.11	1.18	1.58
1000 f/4	764	2581	786	631	**0.33**	1.21	1.35	n.a.	**0.25**	1.07	1.16	1.14
1000 f/6	1146	3379	1032	765	**0.25**	1.12	1.29	n.a.	**0.17**	1.01	1.05	1.08
1000 [ ]/1	382	1727	466	365	**0.58**	1.32	1.44	n.a.	**0.50**	1.17	1.21	1.29
1000 [ ]/2	764	2663	648	459	**0.33**	1.06	1.55	n.a.	**0.25**	**0.93**	1.20	1.48
1000 [ ]/4	1528	4461	1064	720	**0.20**	1.10	1.57	n.a.	**0.13**	**0.81**	1.01	1.28
1000 [ ]/6	2293	6439	2386	1636	**0.16**	1.02	1.05	n.a.	**0.08**	**0.71**	**0.58**	**0.68**
Average					**0.57**	**1.25**	**1.37**	**n.a.**	**0.49**	**1.07**	**1.09**	**1.36**

some situations, the cost of storing a new term in an empty/small trie can be less than the cost of navigating in the global trie, even when the term is already stored in the global trie. However, our results seem to suggest that this cost decreases proportionally to the length and redundancy of the terms stored in the global trie. In particular, for list terms, GT-T support showed to outperform the original YapTab design and, in particular, the reduction seems to decrease also proportionally to the length of the list terms stored in the global trie.

The results obtained for loading terms also show a cost on running time (around 37% for GT-CA and 9% and 36% for GT-T without and with compiled tries in these experiments). We think that this cost is smaller for GT-T as a result of a cache behaviour effect. With GT-T, as we need to navigate in the global trie for each substitution term, we kept accessing the same global trie nodes, thus reducing eventual cache misses. This seems also to be the reason why for list terms of length 6, GT-T clearly outperforms the original YapTab design, both without and with compiled tries. Note that, for this particular case, the GT-T support only consumes 8% of the memory used in the original YapTab.

Next, we tested our approach with two well-known Inductive Logic Programming (ILP) benchmarks: the *carcinogenesis* (***Carc***) and the *mutagenesis* (***Muta***) data sets. We used these data sets in a Prolog program that simulates the test phase of an ILP system. For that, first we ran the April ILP system [10] for the two data sets, each with two different configurations, in order to collect the set of clauses generated for each configuration. The simulator program then uses the corresponding set of generated clauses to run the positive and negative examples defined for each data set against them. To evaluate clauses, we used two different strategies: ***Pred*** denotes the tabling of individual predicates and ***Conj*** denotes the tabling of literal conjunctions (as described in [3]). By tabling conjunctions, we only need to compute them once. The strategy is then recursively applied as the ILP system generates more specific clauses, but this can increase the table memory usage arbitrarily.

Table 2. Table memory usage (in MBytes) and store/load times (in seconds) for YapTab with and without support for the common global trie data structure

Data Sets	*YapTab*				*GT-CA/YapTab*				*GT-T/YapTab*			
	Mem	*Str*	*Load*	*Cmp*	*Mem*	*Str*	*Load*	*Cmp*	*Mem*	*Str*	*Load*	*Cmp*
Pred												
Carc_P1	1.6	70.72	71.26	72.95	**0.82**	1.35	1.34	n.a.	**0.62**	1.07	1.05	1.03
Carc_P2	2.1	51.19	50.44	55.97	**0.87**	1.42	1.44	n.a.	**0.51**	1.23	1.30	1.22
Muta_P1	0.6	98.93	5.57	5.86	**0.73**	1.20	1.19	n.a.	**0.63**	**0.91**	1.00	**0.94**
Muta_P2	0.6	93.01	2.01	2.40	**0.73**	1.26	1.47	n.a.	**0.63**	**0.96**	1.22	1.10
Average					**0.79**	**1.31**	**1.36**	**n.a.**	**0.60**	**1.04**	**1.14**	**1.07**
Conj												
Carc_C1	18.5	0.56	0.51	0.48	**0.53**	1.57	1.63	n.a.	**0.39**	1.20	1.22	1.08
Carc_C2	2802.8	93.85	70.16	36.44	**0.50**	1.50	1.50	n.a.	**0.14**	1.11	1.09	**0.82**
Muta_C1	84.7	97.02	7.36	6.14	**0.66**	1.30	1.65	n.a.	**0.53**	**0.99**	1.22	1.35
Muta_C2	675.6	92.76	1.36	1.53	**0.16**	1.25	1.42	n.a.	**0.16**	**0.98**	1.10	**0.78**
Average					**0.46**	**1.41**	**1.55**	**n.a.**	**0.31**	**1.07**	**1.16**	**1.01**

Table 2 shows the table memory usage (columns ***Mem***), in MBytes, and the running times, in seconds, to store (columns ***Str***) the tables (first execution) and to load from the tables (second execution) the complete set of subgoals/answers without (columns ***Load***) and with (columns ***Cmp***) compiled tries for YapTab using the original table organization (column ***YapTab***), using the previous GT-CA approach (column ***GT-CA/YapTab***) and using the new GT-T design (column ***GT-T/YapTab***). Again, for the GT-CA and GT-T approaches we only show the memory and running time ratios over YapTab's original table organization.

In general, the results in Table 2 confirm the results obtained in Table 1 for memory usage. GT-T support clearly outperforms the original and GT-CA designs for memory usage. In particular, for the ***Conj*** strategy, memory usage showed to be significantly less with GT-T support. This happens because after a certain time, the ***Conj*** strategy will not table new terms, but only answers that are combinations of previous terms, therefore making the GT-T approach more feasible as it can share the representation of common terms appearing at different argument or substitution positions.

Regarding running time, the results in Table 2 also confirm and reinforce the results obtained in Table 1. GT-T support clearly outperforms the GT-CA design for storing and loading times and, for some configurations, it also outperforms the original YapTab design. This is the case for configurations either without or with compiled tries. These results suggest that, at least for some class of applications, GT-T support has potential to achieve significant reductions in memory usage without compromising running time.

6 Conclusions and Further Work

We have presented a new design for the table space organization where all argument and substitution terms appearing in tabled subgoal calls and/or answers are represented only once in a common global trie instead of being spread over several different trie data structures. The goal is to reduce redundancy in term representation by maximizing the sharing of tabled data that is structurally equal. Our experiments using the YapTab tabling system showed that our approach has potential to achieve significant reductions on memory usage without compromising running time.

Further work will include exploring the impact of applying our proposal to other real-world applications that pose many subgoal queries, possibly with a large number of redundant answers, seeking real-world experimental results allowing us to improve and expand our current implementation. In particular, we intend to study how alternative/complementary designs for the table space organization can further reduce redundancy in term representation.

References

1. Sagonas, K., Swift, T.: An Abstract Machine for Tabled Execution of Fixed-Order Stratified Logic Programs. ACM Transactions on Programming Languages and Systems 20(3), 586–634 (1998)
2. Ramakrishnan, I.V., Rao, P., Sagonas, K., Swift, T., Warren, D.S.: Efficient Access Mechanisms for Tabled Logic Programs. Journal of Logic Programming 38(1), 31–54 (1999)
3. Rocha, R., Fonseca, N.A., Santos Costa, V.: On Applying Tabling to Inductive Logic Programming. In: Gama, J., Camacho, R., Brazdil, P.B., Jorge, A.M., Torgo, L. (eds.) ECML 2005. LNCS (LNAI), vol. 3720, pp. 707–714. Springer, Heidelberg (2005)
4. Rocha, R.: On Improving the Efficiency and Robustness of Table Storage Mechanisms for Tabled Evaluation. In: Hanus, M. (ed.) PADL 2007. LNCS, vol. 4354, pp. 155–169. Springer, Heidelberg (2006)
5. Costa, P., Rocha, R., Ferreira, M.: Relational Models for Tabling Logic Programs in a Database. In: Seipel, D., Hanus, M., Wolf, A. (eds.) INAP and WLP 2007. LNCS (LNAI), vol. 5437, pp. 99–116. Springer, Heidelberg (2009)
6. Goto, E.: Monocopy and Associative Algorithms in Extended Lisp. Technical Report TR 74-03, University of Tokyo (1974)
7. Rocha, R., Silva, F., Santos Costa, V.: On applying or-parallelism and tabling to logic programs. Theory and Practice of Logic Programming 5(1-2), 161–205 (2005)
8. Bachmair, L., Chen, T., Ramakrishnan, I.V.: Associative Commutative Discrimination Nets. In: Gaudel, M.-C., Jouannaud, J.-P. (eds.) CAAP 1993, FASE 1993, and TAPSOFT 1993. LNCS, vol. 668, pp. 61–74. Springer, Heidelberg (1993)
9. Costa, J., Rocha, R.: One Table Fits All. In: Gill, A. (ed.) PADL 2009. LNCS, vol. 5418, pp. 195–208. Springer, Heidelberg (2009)
10. Fonseca, N.A., Silva, F., Camacho, R.: April - An Inductive Logic Programming System. In: Fisher, M., van der Hoek, W., Konev, B., Lisitsa, A. (eds.) JELIA 2006. LNCS (LNAI), vol. 4160, pp. 481–484. Springer, Heidelberg (2006)

A New Approach to Non-termination Analysis of Logic Programs

Dean Voets* and Danny De Schreye

Department of Computer Science, K.U.Leuven, Belgium
Celestijnenlaan 200A, 3001 Heverlee
{Dean.Voets,Danny.DeSchreye}@cs.kuleuven.be

Abstract. In this paper, we present a new approach to non-termination analysis of logic programs, based on moded SLDNF-resolution. Moded SLDNF-resolution is a symbolic execution for moded goals, developed for termination prediction. To prove non-termination, we use a complete loop checker to create a finite symbolic derivation tree of a logic program for a moded query. Then, we check if this derivation tree contains an infinite loop, using a new non-termination condition. We implemented this approach and tested it on the benchmark from the Termination Competition of 2007. The results are very satisfactory: our tool is able to prove non-termination and construct non-terminating queries for all non-terminating benchmark programs, and thus, significantly improves on the results of the only other non-termination analyzer, *NTI*.

Keywords: non-termination analysis, program analysis.

1 Introduction

One of the central concerns of declarative programming, in particular of Logic Programming, is that the use of a declarative programming style in a declarative programming language leads to less error-prone, more understandable and better maintainable programs. However, it is well-known that a declarative programming style also results in less efficient computations, and in the extreme case, in non-terminating computations. The latter problem has received considerable attention within the community. Much research has been done on termination analysis, loop detection and more recently, non-termination analysis.

Among these areas, termination analysis has by far received most attention. Most of the more powerful approaches and techniques have been introduced in the last decade: the constraint-based approach to termination analysis [6], the local approaches [4], the use of types in termination analysis [2], powerful transformational approaches [11], termination inference [8], and the porting of TRS-techniques to the LP-context [9].

A rather recent concern in this research is the precision of the termination analysis. Since termination is undecidable in general, only sufficient conditions for termination are verified. It is important to have a good understanding of the precision of these techniques: do they actually capture most of the terminating computations?

* Supported by the Fund for Scientific Research - FWO-project G0561-08.

With respect to the other two approaches, loop detection and non-termination analysis, there is often confusion concerning their relation. Because both approaches use similar techniques, their distinguishing features and aims are not always well understood. Loop detection is a run-time technique. It aims to cut infinite derivations for a concrete query at run-time. For an extensive overview and comparison of different loop checking algorithms, we refer to [1]. Non-termination analysis is a compile-time approach. It aims to prove that a certain class of queries will result in non-terminating computations for at least some of the queries in the considered class. Non-termination analysis is performed for classes of queries described in terms of modes (or types). One of the key concerns of non-termination analysis is to address the important issue of precision analysis of termination analysis. A termination analysis can be shown to be precise by proving that the class of queries for which termination could not be proven is actually non-terminating. This has been one of the main goals and achievements of the only non-termination analyzer developed up till now, NTI[10].

Very recently, yet another, fourth approach to the problem has been introduced: termination and non-termination prediction [12]. In this approach, techniques developed in loop-detection are lifted to classes of moded queries to allow for a prediction of the termination behavior of these queries. Although the predictions do not take the form of formal proofs, experiments show that they can be extremely precise. Moreover, for non-termination prediction, it has been proven that by increasing a parameter in the analysis, the repetition number, in the limit, the prediction is always correct.

Our work has been inspired both by the work on termination/non-termination prediction and by NTI. We propose a new non-termination analysis. It reuses the analysis scheme proposed in [12] to produce a finite representation of the computation for a moded query, given some logic program. We introduce a new non-termination condition expressed in terms of this finite representation of the computation. We prove its correctness and extend it to increase its applicability.

It turns out that our characterization of non-terminating computations is more precise than that of NTI. We have implemented the technique and performed extensive experiments with it on the basis of the benchmark of the termination analysis competition of 2007[1]. The experiments show that our technique has a 100% success-rate on this benchmark, outperforming the only competing approach, NTI.

The paper is organized as follows. In the next section we introduce some preliminaries. In section 3, we present our conditions implying non-termination and show that we are able to derive classes of non-terminating queries. In Section 4, we present our experimental evaluation and we compare our analyzer with the non-termination inference tool NTI [10]. Finally, Section 5 concludes this paper.

2 Preliminaries

In this section, we introduce some preliminaries concerning the symbolic derivation trees used to prove non-termination. First, we introduce moded SLDNF-trees

[1] http://www.lri.fr/~marche/termination-competition/

as defined in [12]. These trees represent the derivation trees of all concrete queries corresponding to a moded query. Then, we introduce complete loop checks for these SLDNF-trees and introduce *LP-check* [12], a loop check for moded SLDNF-resolution introduced for termination prediction.

2.1 Moded Generalized SLDNF-Trees

We assume the reader is familiar with standard terminology of logic programs, in particular with SLDNF-resolution and substitutions, as described in [7]. Variables are denoted by character strings beginning with a capital letter. Predicates, functions and constant symbols are denoted by character strings beginning with a lower case letter. A term is a constant, a variable, or a function of the form $f(t_1, ..., t_m)$ where f is a function symbol and each t_i is a term. We denote the set of terms constructible from a program P, by $Term_P$. An atom is of the form $p(t_1, ..., t_m)$ where p is a predicate symbol. Two atoms are called *variants* if they are equal up to variable renaming. An atom A is *more general* than an atom B, if there exists a substitution θ, such that $A\theta = B$. A literal is an atom A or the negation $\neg A$ of A.

A general logic program P is a finite set of clauses of the form $A \leftarrow L_1, ..., L_n$, where A is an atom and each L_i is a literal. A goal G_i is a headless clause $\leftarrow L_1, ..., L_n$. A query, Q, is a conjunction of literals $L_1, \ldots, L_n$. Without loss of generality, we assume that Q consists only of one atom.

Let P be a logic program and G_0 a query. G_0 is evaluated by building a *generalized SLDNF-tree* GT_{G_0} as defined in [12], in which each node is represented by $N_i : G_i$ where N_i is the name of the node and G_i is a goal attached to the node. We do not reproduce the definition of a generalized SLDNF-tree. Roughly speaking, GT_{G_0} is the set of standard SLDNF-trees for $P \cup \{G_0\}$ augmented with an ancestor-descendant relation on their literals. Let L_i and L_j be the selected literals at two nodes N_i and N_j, respectively. L_i is an *ancestor* of L_j, denoted $L_i \prec_{anc} L_j$, if the proof of L_i goes via the proof of L_j. Throughout the paper, we choose to use the best-known *depth-first, left-most* control strategy, as is used in Prolog, to select goals and literals. So by the *selected literal* in each node $N_i :\leftarrow L_1, ..., L_n$, we refer to the left-most literal L_1. For any node $N_i : G_i$, we use L_i^1 to refer to the selected literal in G_i.

Recall that in SLDNF-resolution, let $L_i = \neg A$ be a ground negative literal selected at N_i, then, by the negation-as-failure rule [7], a subsidiary child SLDNF-tree will be built to solve A. In a generalized SLDNF-tree GT_{G_0}, such parent and child SLDNF-trees are connected from N_i to N_{i+1} via a dotted edge "$\cdots\triangleright$" ,called a *negation arc*, and A at N_{i+1} inherits all ancestors of L_i at N_i. Therefore, a path of a generalized SLDNF-tree may come across several SLDNF-trees through dotted edges. Any such path starting at the root node $N_0 : G_0$ of GT_{G_0} is called a *generalized SLDNF-derivation.*

We do not consider *floundering* queries; i.e., we assume that no non-ground negative literals are selected at any node of a generalized SLDNF-tree (see [12]).

A derivation step is denoted by $N_i : G_i \Rightarrow_C N_{i+1} : G_{i+1}$, meaning that applying a clause C to G_i produces $N_{i+1} : G_{i+1}$.

As stated in the introduction, we want to prove non-termination for classes of queries described using *modes*. An *input* mode stands for an arbitrary ground term, i.e. it can be any variable-free term of $Term_P$. An *output* mode stands for a free variable. A query Q is a *moded* query if some arguments of Q are input modes, otherwise, it is a *concrete* query. Because an input mode denotes an arbitrary ground term, we may approximate the effect of an input mode, by treating it as a special variable I, in such a way that in SLDNF-derivations I can be substituted by a constant or function, but cannot be substituted by an ordinary variable. Therefore, when unifying a special variable I and a variable X, we always substitute I for X. In the remainder of the paper, we denote a special variable by underlining the variable's name.

Definition 1. *Let P be a logic program and $Q = p(\underline{I_1}, ..., \underline{I_m}, T_1, ..., T_n)$ a moded query. The* ***moded generalized SLDNF-tree*** *of P for Q, is defined to be the generalized SLDNF-tree GT_{G_0} for $P \cup \{\leftarrow p(\underline{I_1}, ..., \underline{I_m}, T_1, ..., T_n)\}$, with each $\underline{I_i}$ being a distinct special variable not occurring in any T_j. The special variables $\underline{I_1}, ..., \underline{I_m}$ are called* ***input variables****.* □

In a moded generalized SLDNF-tree, an input variable $\underline{I}$ may be substituted by either a constant or a function $f(t_1, \ldots, t_n)$. If $\underline{I}$ is substituted by $f(t_1, \ldots, t_n)$, all variables in $t_1, \ldots, t_n$ are also called input variables and treated as special variables. We refer to Figure 1(a) for an illustration of (part of) a moded generalized SLDNF-tree. The figure also illustrates a loop check.

A moded atom A corresponds to a set of concrete atoms, called the *denotation* of A. Let $\underline{I_1}, ..., \underline{I_n}$ be all input variables occurring in A. Let $t_1, \ldots, t_n \in Term_P$. $A(t_1 \rightarrow \underline{I_1}, \ldots, t_n \rightarrow \underline{I_n})$ denotes the concrete atom obtained by replacing the input variables $\underline{I_1}, ..., \underline{I_n}$ by the terms $t_1, \ldots, t_n$.

Definition 2. *Let A be an atom with $\underline{I_1}, \ldots, \underline{I_n}$ as its input variables. The* ***denotation*** *of A is*

$$Den(A) = \left\{A(t_1 \rightarrow \underline{I_1}, \ldots, t_n \rightarrow \underline{I_n}) \mid t_i \in Term_P, t_i \textit{ is ground}\right\}. \qquad \square$$

This concept can be adapted to moded goals in a straightforward way. Note that the denotation of a concrete atom is a singleton containing the atom itself.

2.2 Loop Checking

A complete loop check for moded goals cuts all infinite branches in a moded generalized SLDNF-tree.

Definition 3. *A loop check L is* ***complete*** *w.r.t. moded SLDNF-resolution if for every logic program P and moded query Q, every infinite derivation of P for Q is cut by L.* □

Many simple complete loop checks can be constructed, for example a bound on the number of times a certain predicate occurs in a derivation. However, only one loop check for moded SLDNF-resolution is discussed in the literature, *LP-check* [12]. LP-check is a complete loop check developed for termination prediction.

In [12], it is proven that every infinite derivation contains an infinite chain of *loop goals*. These are goals satisfying some conditions on the selected literals. A clause is cut by LP-check if a prefix of such a chain is encountered.

Definition 4. *Let A be a moded atom, the* ***symbol string*** *of A, S_A, is the string obtained by reading all predicate symbols, function symbols, constants and variables in A, from left to right, with the variables replaced by $\mathcal{X}$.*

A symbol string S_{A_1} is a ***projection*** *of S_{A_2}, denoted $S_{A_1} \sqsubseteq_{proj} S_{A_2}$, if S_{A_1} is obtained from S_{A_2} by removing zero or more elements.* □

Example 1. Let $A_1 = a$ and $A_2 = f(X, g(X, f(a, \underline{I})))$. Then, $S_{A_1} = a$, $S_{A_2} = f\mathcal{X}g\mathcal{X}fa\mathcal{X}$ and $S_{A_1} \sqsubseteq_{proj} S_{A_2}$. □

Definition 5. *Let $N_i : G_i$ and $N_j : G_j$ be two nodes in a derivation with $L_i^1 \prec_{anc} L_j^1$ and $S_{L_i^1} \sqsubseteq_{proj} S_{L_j^1}$. Then, G_j is called a* ***loop goal*** *of G_i.* □

LP-check uses a *repetition number* defining how long the chain of loop goals can become before it is cut by LP-check.

Definition 6. *Given a repetition number $r \geq 2$,* ***LP-check*** *is defined as follows: Any derivation D in a generalized SLDNF-tree is cut at a node N_{g_r} if D has a prefix of the form*

$$N_0 : G_0 \Rightarrow_{C_0} ... \; N_{g_1} : G_{g_1} \Rightarrow_{C_k} ... \; N_{g_2} : G_{g_2} \Rightarrow_{C_k} ... \; N_{g_r} : G_{g_r} \Rightarrow_{C_k} \qquad (1)$$

such that (a) for any $j < r$, $G_{g_{j+1}}$ is a loop goal of G_{g_j}, and (b) for all $j \leq r$, the clause C_k applied to G_{g_j} is the same. The prefix is called an ***LP-cut****, the nodes $N_{g_1}, \ldots, N_{g_r}$ are called the nodes of the LP-cut.* □

Because LP-check is a rather expensive loop check, a variant on LP-check is defined in [12]: *LP-check with pruning.* This loop check reduces the amount of redundant branches by pruning clauses if they are already applied to an ancestor or descendant with a variant as a selected literal. We illustrate these loop checks with the *binary tree* program.

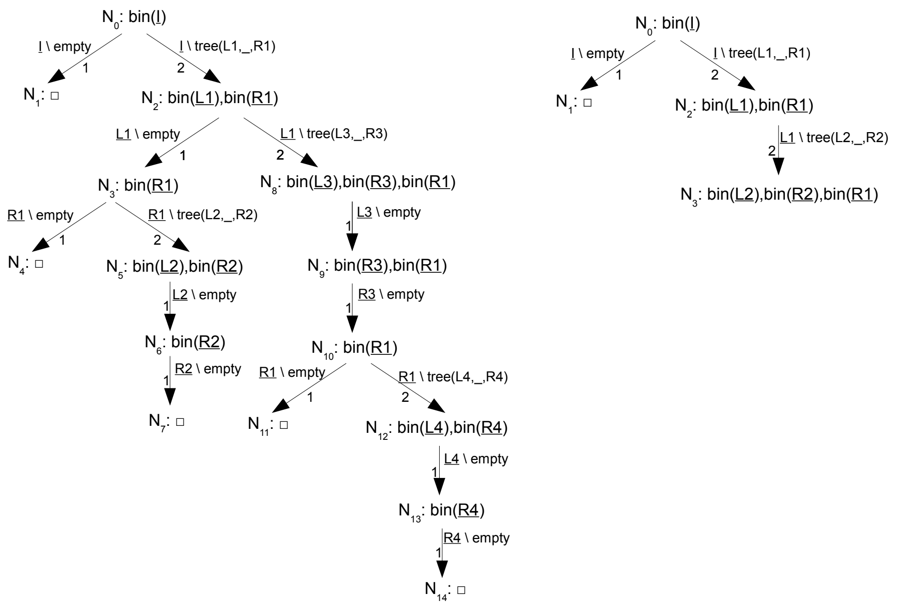
Example 2. The following program succeeds if the argument of the query represents a binary tree.

```
bin(empty).                    bin(tree(L,_,R)):- bin(L), bin(R).
```

Figures 1(a) and 1(b) show moded SLDNF-trees constructed using LP-check and LP-check with pruning, respectively, for the *binary tree* program with *bin*($\underline{I}$) as a query and 3 as a repetition number.

In the SLDNF-tree constructed by LP-check, as shown in Figure 1(a), clause 2 is cut at node N_5 because of LP-cut: $N_0 \Rightarrow_{C_2} \ldots N_3 \Rightarrow_{C_2} N_5 \Rightarrow_{C_2}$. Similarly, LP-check cuts clause 2 at nodes: N_6, N_8, N_9, N_{12} and N_{13}.

The SLDNF-tree constructed by LP-check with pruning, depicted in Figure 1(b), is much smaller. Clause 1 is cut at nodes N_2 and N_3, because that clause is applied to the selected literal of N_0, which is both an ancestor and a variant of the selected literals of nodes N_2 and N_3. At node N_3, clause 2 is cut by LP-check with pruning because of LP-cut: $N_0 \Rightarrow_{C_2} N_2 \Rightarrow_{C_2} N_3 \Rightarrow_{C_2}$. □



(a) LP-check (b) LP-check with pruning

Fig. 1. Two loop checks for moded SLDNF-resolution

3 A New Non-termination Condition

In this section, we present a new non-termination analysis technique for general logic programs with moded queries. We consider a program *non-terminating* w.r.t. a moded query, if the denotation of the query contains at least one concrete query that has an infinite branch in its generalized SLDNF-tree.

3.1 The Moded More General Relation

To prove non-termination, we prove that a path between two nodes N_b and N_e in a moded SLDNF-derivation can be repeated infinitely often. To find such a path, we check three properties. Because the rules in the path must be applicable independent of the values of the input variables, no substitutions on the input variables may occur in the path from N_b to N_e. Because this path should be a loop, L_b^1 must be an ancestor of L_e^1. Finally, a special more general relation for moded atoms must hold between L_b^1 and L_e^1. We will show that these three conditions imply non-termination.

A moded atom A is *moded more general* than a moded atom B, if any atom in the denotation of A is more general than some atom in the denotation of B.

Definition 7. *A moded atom A is* **moded more general** *than a moded atom B w.r.t. a program P, $A \rhd B$, iff:*

$\forall I \in Den(A), \exists I' \in Den(B) : I$ *is more general than* I' □

We illustrate this moded more general relation with some small examples.

Example 3. The following relations hold w.r.t. the *binary tree* program from Example 2:

– $bin(X) \rhd bin(\underline{I})$
  The denotation of $bin(X)$ only contains the atom itself, which is more general than any atom in the denotation of $bin(\underline{I})$, e.g. $bin(empty)$.
– $bin(tree(tree(\underline{In}, Xn), Y)) \rhd bin(tree(\underline{I}, tree(X, empty)))$
  For example, if $\underline{In} = empty$, then $\underline{I} = tree(empty, empty)$ yields an atom in the denotation that satisfies the more general relation. □

Because the denotation of a moded atom is in general infinite, we cannot check this property for every atom in the denotation. However, there is a syntactic sufficient condition to check if the moded more general relation holds between two given moded atoms A and B. The condition is based on a particular kind of unifiability of the atoms.

We introduce the following notations. Let $InVar_P$ be the set of input variables and Var_P the set of normal variables. To every $\underline{I} \in InVar_P$ we associate a fresh normal variable I. Let $Term_P^+$ denote the set of all terms constructible in the underlying language of P augmented with the variables $\{I \mid \underline{I} \in InVar_P\}$.

Proposition 1. *Let A and B be two moded atoms. Let A_1 and B_1 be renamings of these atoms such that they have no shared variables. Let A_2 and B_2 denote variants of A_1 and B_1 in which every input variable $\underline{I}$ is replaced by I. Let $N_1^a, \dots, N_n^a$ be a subset of the normal variables in A_1 and $I_1^b, \dots, I_m^b$ be the fresh variables associated to the input variables in B_2.*

If A_2 and B_2 are unifiable with a substitution $\gamma = \{N_1^a \setminus t_1, \dots, N_n^a \setminus t_n, I_1^b \setminus t_1^+, \dots, I_m^b \setminus t_m^+\}$ with $t_1, \dots, t_n \in Term_P$ and $t_1^+, \dots, t_m^+ \in Term_P^+$, then A is moded more general then B. □

Proof. Let $\alpha = \{N_1^a \setminus t_1, \dots, N_n^a \setminus t_n\}$ and $\beta = \left\{I_1^b \setminus t_1^+, \dots, I_m^b \setminus t_m^+\right\}$. Because $I_1^b, \dots, I_m^b$ can not occur in $t_1, \dots, t_n$, $\gamma = \beta \circ \alpha$, and by unifiability, $A_2\alpha\beta = B_2\alpha\beta$. Moreover, since B_2 does not contain $N_1^a, \dots, N_n^a$, $B_2\alpha\beta = B_2\beta$, and since $A_2\alpha$ does not contain $I_1^b, \dots, I_m^b$, $A_2\alpha\beta = A_2\alpha$. Thus, $A_2\alpha = B_2\beta$.

Let A_c be an element of $Den(A_1)$. Then, there exists a substitution $\psi = \{I_1^a \setminus s_1, \dots, I_k^a \setminus s_k\}$, where $\underline{I_1^a}, \dots, \underline{I_k^a}$ are all input variables of A_1, $s_1, \dots, s_k \in Term_P$ and $s_1, \dots, s_k$ are ground, such that $A_c = A_2\psi$.

Now consider the atom $B_c = B_2\beta\psi$. First, $B_c \in Den(B_1)$. This is because β replaces all I_j^b of B_2 by terms t_j^+. These terms t_j^+ may contain variables I_l^a of A_2, but these are all substituted to ordinary ground terms $s_l \in Term_P$ by ψ.

Finally, $A_c\alpha = A_2\psi\alpha = A_2\alpha\psi = B_2\beta\psi = B_c$. Note that $A_2\psi\alpha = A_2\alpha\psi$ because no s_i of ψ can contain a variable N_j^a of α, nor can any t_i of α contain a variable I_j^a of ψ. Thus A_c is more general than an element of $Den(B_1)$. □

Example 4. The moded atoms of the last example are already variable disjunct. To check if the moded more general relation holds, we have to check if the atoms are unifiable with a substitution of the correct forms.

– $bin(X) = bin(I)$ with substitution: $\{I \setminus X\}$
– $bin(tree(tree(In, Xn), Y)) = bin(tree(I, tree(X, empty)))$ with substitution: $\{I \setminus tree(In, Xn), Y \setminus tree(X, empty)\}$ □

3.2 Non-termination of Moded More General Loops

If a moded SLDNF-derivation contains a path without substitutions on input variables, such that the ancestor relation and the moded more general relation hold between the first and last selected literal in that path, we call this path a *moded more general loop*. We will show that a moded more general loop implies non-termination.

Definition 8. *In a moded SLDNF-derivation D, nodes $N_i : G_i$ and $N_j : G_j$ are a* ***moded more general loop****, $N_i : G_i \overset{mmg}{\rightarrow} N_j : G_j$, iff:*

– *No substitutions on input variables occur in the path from N_i to N_j.*
– *$L_i^1 \prec_{anc} L_j^1$.*
– *$L_j^1 \rhd L_i^1$* □

Note that when no confusion can occur, we may omit writing the goal in the moded more general loop.

A moded more general loop, $N_i : G_i \overset{mmg}{\rightarrow} N_j : G_j$, corresponds to an infinite loop for every concrete goal in the denotation of G_i.

Theorem 1 (Sufficiency of the moded more general loop). *Let $N_i : G_i \overset{mmg}{\rightarrow} N_j : G_j$ be a moded more general loop in a moded SLDNF-derivation D of a program P and a moded query I. The sequence of clauses from N_i to N_j, $\langle C_1, \ldots, C_n \rangle$, can be repeated infinitely often for any goal in $Den(G_i)$.* □

Proof. Because L_i^1 is an ancestor of L_j^1, only the selected literal of N_i influences if the sequence of clauses can be repeated infinitely often.

Because no substitutions on input variables occur in the path from N_i down to N_j, $\langle C_1, \ldots, C_n \rangle$ is applicable to any atom in $Den(L_i^1)$. Obviously, this path is also applicable to any atom A, which is more general than some atom B in $Den(L_i^1)$. Furthermore, after applying $\langle C_1, \ldots, C_n \rangle$ to A, the resulting selected literal is more general than the selected literal after applying $\langle C_1, \ldots, C_n \rangle$ to B.

As $L_j^1 \rhd L_i^1$, any atom in $Den(L_j^1)$ is more general than some atom in $Den(L_i^1)$.

Therefore, let S be the union of $Den(L_i^1)$ and all more general atoms. Then, $\langle C_1, \ldots, C_n \rangle$ is applicable to any atom of S, and after applying these clauses, the selected literal of the resulting goal is again an atom of S. Thus, this sequence of clauses is infinitely often applicable to elements of S. □

We illustrate this non-termination condition with our running example.

Example 5 (Non-termination proof of binary tree). Let us revisit Example 1 with a query $bin(X)$. The SLDNF-tree constructed by LP-check for this program and query is almost the same as in Figure 1(a). The only difference is that the input variables are replaced by ordinary variables.

The constructed SLDNF-tree satisfies the conditions of Definition 8, so $N_0 \overset{mmg}{\rightarrow} N_2$ is a moded more general loop. Therefore, non-termination of this example is proven by Theorem 1. □

Observe that Theorem 1 can straightforwardly be generalized to conclude non-termination for any goal that is more general than an element of $Den(G_i)$. In particular, the analysis is not restricted to goals with ground inputs: Theorem 1 also holds for an "extended" denotation of G_i, with non-ground inputs.

3.3 Input-Generalizations

Our experimental evaluation (see Section 4) shows that for many non-terminating programs, non-termination can be proven using the moded more general loop. But, the next example shows that there is room for further improvement.

Example 6 (Termination behavior of flat).

```
flat(niltree, nil).
flat(tree(X, niltree, XS), cons(X, YS)) :- flat(XS, YS).
flat(tree(X, tree(Y, YS1, YS2), XS), ZS) :-
        flat(tree(Y, YS1, tree(X, YS2, XS)), ZS).
```

This program, *flat*, flattens a binary tree into a list denoted with the *cons* notation. To flatten the tree, the program repeatedly moves one element from the left to the right subtree until the left subtree is empty. When the left subtree is empty, we proceed by processing the right subtree. If the first argument of the query is a variable, this program loops w.r.t. the third clause.

Figure 2 shows a part of the moded generalized SLDNF-tree constructed for moded query $flat(T, \underline{I})$ using LP-check with repetition number 3. No nodes in the derivations satisfy Definition 8. The reason for this is that we replace a variable by a compound term when applying the third clause. □

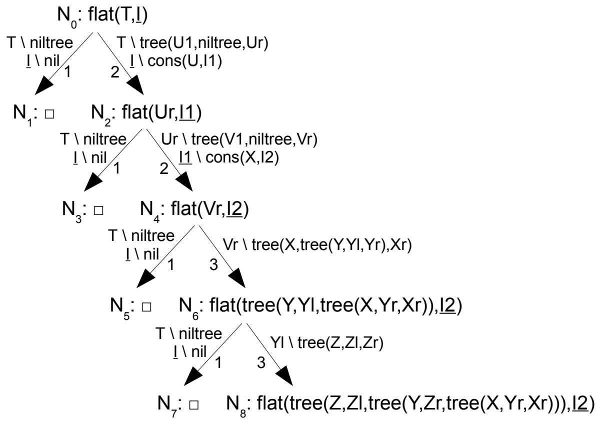


Fig. 2. Moded generalized SLDNF-tree with LP-check of *flat* (Example 6)

To prove non-termination for programs such as $flat$, we define an *input-generalization.* This input-generalization is such that proving non-termination of an input-generalized goal implies non-termination of the original goal.

Definition 9. *We say that A^α is an input-generalization of an atom A, if there exist terms $t_1, \ldots, t_n$ in A and fresh input variables $\underline{I_1}, \ldots, \underline{I_n}$ such that $A^\alpha = A(\underline{I_1} \rightarrow t_1, \ldots, \underline{I_n} \rightarrow t_n)$ and $Var(A^\alpha) \cap Var((t_1, \ldots, t_n)) = \emptyset$.* □

Example 7 (Input generalizations).
- $bin(tree(\underline{I}, \underline{I_1}))$ is an input-generalization of $bin(tree(\underline{I}, tree(X, empty)))$
- $bin(\underline{I_2})$ is an input-generalization of $bin(tree(\underline{I}, \underline{I_1}))$
- $bin(tree(\underline{I_3}, X))$ is not an input generalization of $bin(tree(tree(X, Y), X))$ This last example refers to the condition of the empty intersection of the variable sets. We return to this condition in Example 8. □

To check if a path is non-terminating w.r.t. an input-generalized goal, we define an *input-generalized derivation.* This derivation is constructed by applying a path in a given derivation to the input-generalized selected literal of the first node in the path.

Definition 10. *Let D be a moded SLDNF-derivation $N_i, \ldots, N_j$, such that $L_i^1 \prec_{anc} L_j^1$. Let $\langle C_1, \ldots, C_n \rangle$ be the sequence of clauses applied from N_i to N_j and let A^α be an input-generalization of L_i^1.*

The ***input-generalized derivation*** *D' for A^α, is constructed by applying the sequence of clauses $\langle C_1, \ldots, C_n \rangle$ to A^α. The* ***input-generalized nodes*** *N_i^α and N_j^α are the top and bottom nodes of D', respectively.* □

Next, we prove that non-termination of the input-generalized derivation implies non-termination of the original goal. First we introduce two lemmas.

Lemma 1. *Let A^α be an input generalization of A, then $A \rhd A^\alpha$.* □

Proof. Let $\underline{I_1}, \ldots, \underline{I_n}$ be the input variables of A and $\underline{I_{n+1}}, \ldots, \underline{I_m}$ be the new introduced input variables in A^α. For every concrete atom A_c in $Den(A)$, $\underline{I_1}, \ldots, \underline{I_n}$ are replaced by ground terms. To construct an atom A_c^α of $Den(A^\alpha)$, for which A_c is more general then A_c^α, one replaces $\underline{I_1}, \ldots, \underline{I_n}$ by the same values as in A_c and $\underline{I_{n+1}}, \ldots, \underline{I_m}$ by instances of the corresponding terms, $t_{n+1}, \ldots, t_m$, in A_c. Due to the condition that $Var(A^\alpha) \cap Var((t_{n+1}, \ldots, t_m)) = \emptyset$, A_c is more general than A_c^α. □

Example 8. To explain the condition on the intersection of the variables in Definition 9, consider the atom $A = a(X, f(X))$. If we omit the condition on the variables, we could consider $A^\alpha = a(X, \underline{I})$ as an input generalization. $Den(A) = \{a(X, f(X))\}$ and $a(X, f(X))$ is not more general than any element in $Den(a(X, \underline{I}))$. So, the property that $A \rhd A^\alpha$ would not hold. □

Lemma 2. *Let A and B be atoms such that $A \rhd B$ and let every atom in $Den(B)$ be non-terminating w.r.t. a program P, then, every atom in $Den(A)$ is non-terminating w.r.t. P.* □

Proof. Every atom of $Den(A)$ is more general than a non-terminating atom. □

Corollary 1 (Non-termination with input-generalization). *Let $N_i : G_i$ and $N_j : G_j$ be nodes in a derivation D of a program P for a moded query I, such that $L_i^1 \prec_{anc} L_j^1$, and let N_i^α and N_j^α be input-generalized nodes in an input-generalized derivation D' of N_i and N_j for A.*

If $N_i^\alpha \overset{mmg}{\rightarrow} N_j^\alpha$, then every concrete goal in the denotation of G_i is non-terminating w.r.t. program P. □

Proof. Follows from Theorem 1 and the two previous lemmas. □

We illustrate these input-generalizations by revisiting the *flat* example.

Example 9 (Non-termination of flat). To prove non-termination, we generalize node N_6 to $flat(tree(Y, Yl, \underline{In}), \underline{I2})$, by changing the subterm $tree(X, Yr, Xr)$ to a new input variable $\underline{In}$.

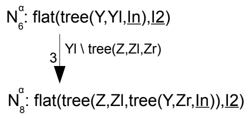

N_6^α: flat(tree(Y,Yl,In),I2)
3
Yl \ tree(Z,Zl,Zr)
N_8^α: flat(tree(Z,Zl,tree(Y,Zr,In)),I2)

Fig. 3. Input-generalized SLDNF-derivation of *flat*

Figure 3, shows the input-generalized moded SLDNF-derivation for $flat(tree(Y, Yl, \underline{In}), \underline{I2})$. This derivation is a moded more general loop: $N_6^\alpha \overset{mmg}{\rightarrow} N_8^\alpha$. Therefore, non-termination of the program *flat* w.r.t. the concrete goals in the denotation of the goal of N_6 is proven by Corollary 1. □

Note that a concrete query in the denotation of a moded query might not reach the moded more general loop. However, classes of non-terminating top level queries can be obtained by applying all substitutions on the input variables between the root and the first node of the moded more general loop. In the last example, this class of top level queries is $flat(T, cons(\underline{U}, cons(\underline{X}, \underline{I2})))$.

4 Experimental Evaluation

To evaluate our approach, we implemented a non-termination analyzer *P2P*, *from Prediction to Proof*, based on Corollary 1. We tested *P2P* on a benchmark of 48 non-terminating pure logic programs. First, we describe our analyzer and the benchmark. Then, we compare our tool with the non-termination inference tool *NTI* [10].

4.1 *P2P*: *From Prediction to Proof*

We implemented *P2P* in SWI-prolog[2]. *P2P* is freely available[3] and consists of two components. First, the implementation of the termination prediction

[2] Homepage of SWI-prolog: http://www.swi-prolog.org/

[3] Available at http://www.cs.kuleuven.be/˜dean/p2p.html

approach [12], $TPoLP$[4], constructs the moded SLDNF-derivation and predicts the termination behavior. If $TPoLP$ predicts a derivation to be non-terminating, the second component tries to prove non-termination in the derivation.

To prove non-termination, $P2P$ checks if the derivation contains a moded more general loop or it uses a backtracking search to attempt to construct an input generalized derivation that contains a moded more general loop. Although many input generalizations can be constructed, proving non-termination in a derivation can be done rather efficiently. This is because the LP-cuts made by $TPoLP$ correctly identify an infinite loop if the repetition number is sufficiently high. Therefore, instead of checking the conditions of the moded more general loop between all pairs of nodes in the derivation, it suffices to check these conditions for the pairs of nodes of the LP-cut.

4.2 Benchmark of Termination Problems

Our benchmark consists of the non-terminating pure logic programs from the termination competition of 2007. The benchmark and the results from the tools that participated in the competition are available[5]. The benchmark of the termination competition contains around 300 logic programs and moded queries representing different challenges in termination and non-termination analysis. A few programs from the competition are omitted because they contain non-logical operations such as arithmetics. The competition benchmark contains some doubles. These were also omitted. The benchmark contains 48 non-terminating programs. All programs contain between 2 and 15 clauses, except for binary4, which contains 41 clauses. The only other non-termination analyzer, NTI [10], proves non-termination for 45 benchmark programs.

Table 1 shows our experimental evaluation on this benchmark using LP-check with pruning, with 4 as a repetition number. The result of our tool is given in the column $P2P$, V denotes that non-termination is proven while X denotes that no non-termination proof was found. The result of NTI is given in the column NTI. The columns *Size* and *Time* show the size in the number of nodes of the SLDNF-tree and the analysis time in seconds, respectively.

The results are very satisfactory. For all programs in the benchmark, non-termination is proven and a class of non-terminating queries can be constructed. The analyzer is very fast. Any benchmark program is analyzed in less than a second and the memory use never exceeds a few megabytes.

As stated, these experiments have been performed using 4 as a repetition number. When we use 3 as a repetition number, our tool fails to prove non-termination of programs $pl7.6.2.a$ and $pl7.6.2.b$. These are two erroneous implementations of a path find algorithm. When using 2 as repetition number, proving non-termination fails for about 25% of the benchmark programs.

[4] Available at http://www.cs.kuleuven.be/~dean/termination_prediction.html

[5] Available at http://www.lri.fr/~marche/termination-competition/

Table 1. Benchmark of non-terminating pure logic programs

Name program	*P2P*	Size	Time	*NTI*	Name program	*P2P*	Size	Time	*NTI*
ackermann-ioi	V	9	0.33	V	permutation-fb	V	22	0.26	V
bad sublist	V	33	0.29	V	pl1.1	V	8	0.25	V
binary4	V	12	0.27	V	pl3.1.1	V	12	0.30	V
delete-bff	V	13	0.31	V	pl3.5.6	V	13	0.31	V
der-fb	V	22	0.29	V	pl4.0.1-oooi	V	33	0.27	V
doublehalfpred	V	38	0.28	V	pl4.5.2	V	481	0.36	V
example4-2	V	4	0.23	V	pl4.5.3a	V	10	0.29	V
flatlength-fbf	V	14	0.23	V	pl4.5.3b	V	10	0.24	V
flatlength-ffb	V	19	0.23	V	pl4.5.3c	V	11	0.27	V
flat-oi	V	9	0.26	X	pl5.2.2	V	59	0.27	V
frontier-fb	V	12	0.27	V	pl7.6.2.a	V	39	0.27	X
ifdiv	V	19	0.29	V	pl7.6.2.b	V	45	0.33	X
in-bf	V	18	0.29	V	quicksort-fb	V	72	0.26	V
inorder-fb	V	4	0.27	V	quicksort-oi	V	74	0.26	V
insert-bff	V	22	0.29	V	reverse-fb	V	9	0.32	V
log2a-oi	V	35	0.25	V	select-bff	V	8	0.32	V
log2b-oi	V	29	0.28	V	slowsort-fb	V	123	0.27	V
mapcolor	V	23	0.31	V	slowsort-oi	V	26	0.26	V
member-bf	V	8	0.27	V	sublist-bf	V	30	0.21	V
mergesort	V	171	0.28	V	subset-bf	V	21	0.23	V
mergesort-oi	V	54	0.28	V	subset-fb	V	14	0.26	V
mergesort_variant	V	15	0.23	V	suffix-bf	V	9	0.25	V
minimum-fb	V	8	0.29	V	transpose2	V	6	0.28	V
naive reverse-fb	V	8	0.37	V	tree_member-bf	V	12	0.28	V

4.3 Comparison with *NTI*

To infer non-terminating queries, *NTI* first transforms a given program into a binary program using *binary unfoldings*. Then, it compares the head and body of the clauses in the binary program with a special more general relation. If this relation holds, non-termination is proven.

The binary unfolding of a program represent the calls made during program execution. Thus, it corresponds to comparing the selected literals in our symbolic computation. The binary unfolding of a program can be computed using a fixpoint operator.

The special more general relation used by *NTI*, $\triangle$*-more general*, is based on the notion of *derivation neutral (DN) filters*. These filters are functions defining, for a clause and argument position, which terms have no influence on the applicability of the clause. Furthermore, if the head atom satisfies the filter, the body atom must satisfy the filter as well. We explain *NTI*'s non-termination condition and compare it with our approach using some small examples.

Example 10 (Recursive clause of reverse-fb).

```
rev([H|T],Temp,Res):- rev(T,[H|Temp],Res).
```

In this clause, the second argument is not replaced by a more general one. Therefore, *NTI* needs a DN filter to prove non-termination. The applicability of the clause does not depend on the value of *Temp*, so we can use the trivial filter, instance of X, for the second argument position. We can also use this filter for the last argument position. Therefore, *NTI* concludes that this clause is non-terminating for each goal where the first argument is more general than $[H|T]$ and the second and third argument are instances of X. □

These DN filters cannot depend on the names of the variables. Therefore, they cannot express that two argument must contain a common subterm.

Example 11 (Variable independent filters).

```
a(X,X):- a(s(X),s(X)).
```

Both arguments are replaced by more specific ones and the applicability does not depend on the value of X. However, since both arguments must be bound to the same term, NTI fails to prove non-termination of this example. □

Instead of comparing all the arguments independently, our approach compares the selected literals. Therefore, our condition does not have this restriction.

Because NTI requires that each argument is either replaced by a more general one or satisfies a DN filter, NTI fails to prove non-termination if in one argument, a subterm is replaced by a more general one while another subterm is replaced by a more specific one. This is because of the requirement that if the head atom satisfies the filter, the body atom needs to satisfy it as well.

Example 12 (Looping clause of $flat$-oi). In the third clause of flat-oi in Example 9, the first argument of this clause contains two such subterms. `XS` is replaced by `tree(X,YS2,XS)` and `tree(Y, YS1, YS2)` is replaced by `YS1`. □

Because we allow arguments to contain both input and ordinary variables, our condition does not have this restriction.

Table 1 shows that NTI fails to prove non-termination of 3 programs. These 3 programs are examples of the two classes of problems that are illustrated by Examples 11 and 12. The actual results on the termination competition were worse for NTI, as we have rewritten some programs that NTI could not parse.

5 Conclusion and Future Work

We introduced a new approach to non-termination analysis of logic programs based on a finite, symbolic derivation tree for a moded query. This symbolic tree represents the derivation trees of all concrete queries denoted by the moded query. To prove non-termination we look for a loop in this symbolic derivation tree. We implemented this approach and evaluated it on a benchmark of 48 non-terminating programs from the termination competition of 2007. Our tool, $P2P$, proves non-termination of all benchmark programs. We have shown that our technique improves on the results of the only other non-termination analyzer, NTI, and that we can handle 2 new classes of programs.

A class of programs that we currently cannot handle, are programs that require types to describe the looping goals. These are programs of the following form:

```
p(L):-list(L), p([a|L]).
list([]).                              list([H|T]):- list(T).
```

We cannot prove that a query $p(\underline{L})$ with $\underline{L}$ an input list is non-terminating, because substitutions on the input variables occur between all selected literals that satisfy the moded more general relation. To overcome this problem, we plan to extend our technique with type analysis [3] and non-failure analysis [5]. For this program and query, the type analysis would infer that all arguments are lists and the non-failure analysis would infer that $list(L)$ cannot fail if L is a list. If we combine this information, it is clear that the first clause is a loop.

We also plan to extend our technique for programs containing arithmetic expressions by using a finite domain solver to infer domains and initial values such that the arithmetic conditions in the loop will always succeed.

Acknowledgments. We thank the reviewers for their feedback and helpful comments.

References

1. Bol, R.N.: Loop checking in logic programming. CWI (Centre for Mathematics and Computer Science), Amsterdam, The Netherlands (1995)
2. Bruynooghe, M., Codish, M., Gallagher, J., Genaim, S., Vanhoof, W.: Termination analysis through combination of type based norms. TOPLAS 29(2), 10 (2007)
3. Bruynooghe, M., Janssens, G.: An instance of abstract interpretation integrating type and mode inferencing. In: ICLP/SLP, pp. 669–683 (1988)
4. Codish, M.: Proving termination with (boolean) satisfaction. In: King, A. (ed.) LOPSTR 2007. LNCS, vol. 4915, pp. 1–7. Springer, Heidelberg (2008)
5. Debray, S.K., López-García, P., Hermenegildo, M.V.: Non-failure analysis for logic programs. In: ICLP, pp. 48–62 (1997)
6. Decorte, S., De Schreye, D., Vandecasteele, H.: Constraint-based termination analysis of logic programs. TOPLAS 21(6), 1137–1195 (1999)
7. Lloyd, J.W.: Foundations of Logic Programming, 2nd edn. Springer, Heidelberg (1987)
8. Mesnard, F., Bagnara, R.: Cti: A constraint-based termination inference tool for iso-prolog. TPLP 5(1-2), 243–257 (2005)
9. Nguyen, M.T., De Schreye, D.: Polytool: Proving termination automatically based on polynomial interpretations. In: Puebla, G. (ed.) LOPSTR 2006. LNCS, vol. 4407, pp. 210–218. Springer, Heidelberg (2007)
10. Payet, É., Mesnard, F.: Nontermination inference of logic programs. ACM Transactions on Programming Languages and Systems 28(2), 256–289 (2006)
11. Schneider-Kamp, P., Giesl, J., Serebrenik, A., Thiemann, R.: Automated termination analysis for logic programs by term rewriting. In: Puebla, G. (ed.) LOPSTR 2006. LNCS, vol. 4407, pp. 177–193. Springer, Heidelberg (2007)
12. Shen, Y.-D., De Schreye, D., Voets, D.: Termination prediction for general logic programs. Report CW 2009 536, K.U.L. Accepted for TPLP

Constraint Answer Set Solving

Martin Gebser, Max Ostrowski, and Torsten Schaub*

Universität Potsdam, Institut für Informatik, August-Bebel-Str. 89, D-14482 Potsdam

Abstract. We present a new approach to integrating Constraint Processing (CP) techniques into Answer Set Programming (ASP). Based on an alternative semantic approach, we develop an algorithmic framework for conflict-driven ASP solving that exploits CP solving capacities. A significant technical issue concerns the combination of conflict information from different solver types. We have implemented our approach, combining ASP solver *clingo* with the generic CP solver *gecode*, and we empirically investigate its computational impact.

1 Introduction

Answer Set Programming (ASP;[1]) is a declarative problem solving approach, combining a rich yet simple modeling language with high-performance solving capacities. This has already resulted in various applications, among them decision support systems for NASA shuttle controllers [2,3] and various reasoning tools in systems biology [4,5,6]. However, certain aspects of such applications are more naturally modeled by additionally using non-Boolean constructs, accounting for resources, fine timings, or functions over finite domains. Moreover, a dedicated treatment of large domains avoids the grounding bottleneck inherent to all propositional solving approaches.

In Satisfiability checking (SAT;[7,8]), this led to the subarea of Satisfiability Modulo Theories (SMT;[9]), extending SAT solvers by theory-specific solvers. This allows SMT problems to incorporate predicates from specialized theories into propositional formulas. Solving an SMT problem then consists of finding a (hybrid) assignment to all Boolean and theory-specific variables satisfying a given formula along with its theory-specific constituents. Apart from a close solver integration, the key to efficient SMT solving lies in elaborated conflict-driven learning techniques that are capable of combining conflict information from different solver types (cf. [9]).

Groundbreaking work on enhancing ASP with Constraint Processing (CP;[10,11]) techniques was conducted in [12,13,14]. Based on firm semantic underpinnings, these approaches provide a family of ASP languages parametrized by different constraint classes. While [12] develops a high-level algorithm viewing both ASP and CP solvers as black boxes, [14] embeds a CP solver into a traditional DPLL-style backtracking algorithm, similar to the one underlying the ASP solver *smodels* [15]. Although [12,13,14] resulted in two consecutive extensions of *smodels* with CP capacities, they do not match the performance of state-of-the-art SMT solvers, simply because they do not support advanced backjumping and conflict-driven learning techniques.

* Affiliated with Simon Fraser University, Canada, and Griffith University, Australia.

We address this problem and propose an alternative way of combining ASP and CP solving. To begin with, we pursue an alternative semantic approach that is based on a propositional language rather than a multi-sorted, first-order language, as used in [12,13,14]. Our approach follows the so-called lazy approach of advanced SMT solvers by abstracting from the constraints in a specialized theory [9]. The idea is as follows. The ASP solver passes the portion of its (partial) Boolean assignment associated with constraints to a CP solver, which then checks these constraints against its theory via constraint propagation. As a result, it either signals unsatisfiability or, if possible, extends the Boolean assignment by further constraint atoms. For conflict-driven learning within the ASP solver, however, each assigned constraint atom must be justified by a set of (constraint) atoms providing a "reason" for the underlying inference. Yet, to the best of our knowledge, this is not supported by off-the-shelf CP solvers.[1] As a consequence, we develop an algorithmic framework for conflict-driven ASP solving that integrates CP solving capacities while overcoming the aforementioned difficulty. We have implemented our approach in the new system *clingcon* [16], combining ASP solver *clingo* [17] with the generic CP solver *gecode* [18], and provide an empirical analysis demonstrating its computational impact.

2 Background

A (*normal*) *logic program* over an alphabet $\mathcal{A}$ is a finite set of *rules* of the form

$$a_0 \leftarrow a_1, \dots, a_m, \mathit{not}\ a_{m+1}, \dots, \mathit{not}\ a_n\ , \tag{1}$$

where $a_i \in \mathcal{A}$ is an *atom* for $0 \le i \le n$.[2] A *literal* is an atom a or its (default) negation $\mathit{not}\ a$. For a rule r as in (1), let $\mathit{head}(r) = a_0$ be the *head* of r and $\mathit{body}(r) = \{a_1, \dots, a_m, \mathit{not}\ a_{m+1}, \dots, \mathit{not}\ a_n\}$ be the *body* of r. Given a set B of literals, let $B^+ = \{a \in \mathcal{A} \mid a \in B\}$ and $B^- = \{a \in \mathcal{A} \mid \mathit{not}\ a \in B\}$. Furthermore, given some set $\mathcal{B}$ of atoms, define $B|_{\mathcal{B}} = (B^+ \cap \mathcal{B}) \cup \{\mathit{not}\ a \mid a \in B^- \cap \mathcal{B}\}$. The set of atoms occurring in a logic program P is denoted by $\mathit{atom}(P)$. A set $X \subseteq \mathcal{A}$ is an *answer set* of a program P over $\mathcal{A}$, if X is the $\subseteq$-smallest model of the *reduct* $P^X = \{\mathit{head}(r) \leftarrow \mathit{body}(r)^+ \mid r \in P, \mathit{body}(r)^- \cap X = \emptyset\}$. An answer set can also be seen as a Boolean assignment satisfying all conditions induced by program P (cf. [19]).

A *constraint satisfaction problem* (CSP) is a triple (V, D, C), where V is a set of *variables* with respective *domains* D, and C is a set of *constraints*. Each variable $v \in V$ has an associated domain $\mathit{dom}(v) \in D$. Following [10], a constraint c is a pair (S, R) consisting of a k-ary *relation* R defined on a vector $S \subseteq V^k$ of variables, called the *scope* of R. That is, for $S = (v_1, \dots, v_k)$, we have $R \subseteq \mathit{dom}(v_1) \times \dots \times \mathit{dom}(v_k)$. We use $S(c) = S$ and $R(c) = R$ to access the scope and the relation of $c = (S, R)$. For an assignment $A : V \to \bigcup_{v \in V} \mathit{dom}(v)$ and a constraint (S, R) with $S = (v_1, \dots, v_k)$, define $A(S) = (A(v_1), \dots, A(v_k))$, and let $\mathit{sat}_C(A) = \{c \in C \mid A(S(c)) \in R(c)\}$.

[1] Advanced SMT solvers, like [9], address this through handcrafted theory solvers.

[2] The semantics of choice rules and integrity constraints is given through program transformations. For instance, $\{a\} \leftarrow$ is a shorthand for $a \leftarrow \mathit{not}\ a'$ plus $a' \leftarrow \mathit{not}\ a$ and similarly $\leftarrow a$ for $a' \leftarrow a, \mathit{not}\ a'$, for a new atom a'.

3 Constraint Logic Programs: Syntax and Semantics

For extending logic programs with constraint handling capacities, we consider an extended alphabet distinguishing regular and constraint atoms, denoted by $\mathcal{A}$ and $\mathcal{C}$, respectively. Then, *constraint logic programs* P are defined as regular logic programs over an extended alphabet $\mathcal{A} \cup \mathcal{C}$ such that $head(r) \in \mathcal{A}$ for each $r \in P$.

We identify constraint atoms with constraints via a function $\gamma : \mathcal{C} \rightarrow C$; furthermore, $\gamma(Y) = \{\gamma(c) \mid c \in Y\}$ for any $Y \subseteq \mathcal{C}$. The set of constraints comprised in a constraint logic program P is given by $C[P] = \gamma(atom(P) \cap \mathcal{C})$. While the associated variables $V[P]$ are obtained from the respective constraint scopes, we assume a default domain $D[P]$ for each variable (e.g., provided by a declaration within P).

For a constraint logic program P over $\mathcal{A} \cup \mathcal{C}$ and an assignment $A : V[P] \rightarrow D[P]$, we define the *constraint reduct* as

$$\begin{aligned} P^A = \{ & head(r) \leftarrow body(r)|_{\mathcal{A}} \mid r \in P, \\ & \gamma(body(r)|_{\mathcal{C}}{}^{+}) \subseteq sat_{C[P]}(A), \gamma(body(r)|_{\mathcal{C}}{}^{-}) \cap sat_{C[P]}(A) = \emptyset\} \ . \end{aligned}$$

Then, a set $X \subseteq \mathcal{A}$ is a *constraint answer set* of P wrt A, if X is an answer set of P^A.

Unlike with (standard) atoms in $\mathcal{A}$, the unique names assumption cannot be applied to constraint atoms in $\mathcal{C}$, intentionally representing relations, in a meaningful way. For instance, the same relation between integer variables x and y is described via syntactically different expressions $x < y$ and $((-y-1) \leq -(x+1)) \wedge (x \neq y)$. To reflect this, the definitions of the constraint reduct and constraint answer sets treat constraint literals over $\mathcal{C}$ similar to negative body literals, and truth values are determined outside the actual logic program. Hence, we also do not directly consider constraint atoms as heads but view a rule r with $head(r) \in \mathcal{C}$ as standing for $\leftarrow body(r), not\ head(r)$.

Although our semantics is propositional, the atoms in $\mathcal{A}$ and $\mathcal{C}$ are constructible from a multi-sorted, first-order signature given by:

– a set $\mathcal{P}_{\mathcal{A}} \cup \mathcal{P}_{\mathcal{C}}$ of predicate symbols such that $\mathcal{P}_{\mathcal{A}} \cap \mathcal{P}_{\mathcal{C}} = \emptyset$,
– a set $\mathcal{F}_{\mathcal{A}} \cup \mathcal{F}_{\mathcal{C}}$ of function symbols (including constant symbols),
– a set $\mathcal{V}_{\mathcal{A}}$ of regular variable symbols, and
– a set $\mathcal{V}_{\mathcal{C}} \subseteq \mathcal{T}(\mathcal{F}_{\mathcal{A}})$ of constraint variable symbols, where $\mathcal{T}(\mathcal{F}_{\mathcal{A}})$ denotes the set of all ground terms over $\mathcal{F}_{\mathcal{A}}$.

As common in ASP, the atoms in $\mathcal{A} \cup \mathcal{C}$ are obtained by a grounding process, systematically substituting all occurrences of regular variables in $\mathcal{V}_{\mathcal{A}}$ by (ground) terms from $\mathcal{T}(\mathcal{F}_{\mathcal{A}})$. Atoms in $\mathcal{A}$ are formed from predicate symbols in $\mathcal{P}_{\mathcal{A}}$ and terms in $\mathcal{T}(\mathcal{F}_{\mathcal{A}})$, while the ones in $\mathcal{C}$ are formed from predicate symbols in $\mathcal{P}_{\mathcal{C}}$ and terms over $\mathcal{F}_{\mathcal{C}}$ and $\mathcal{V}_{\mathcal{C}}$. This definition tolerates occurrences of similar ground terms in atoms of both $\mathcal{A}$ and $\mathcal{C}$.

Our approach follows the one taken by SMT solvers in letting the ASP solver deal with the atomic, that is, Boolean structure of the program, while a CP solver addresses the "sub-atomic level" by dealing with the constraints associated with constraint atoms. Whenever a constraint atom $c \in \mathcal{C}$ is assigned to true ($\mathbf{T}$) or false ($\mathbf{F}$) by the ASP solver, the CP solver enforces the satisfaction or violation of the associated constraint $\gamma(c)$.

For illustration, let us consider a constraint logic program consisting of the rules in (2)–(12). This is an authentic program, processable by our solver; its syntax extends the input language of *gringo* [20] and thus allows for using integral ranges, as in (2), and cardinality rules, as in (4). For simplicity, we omit domain atoms $bucket(B)$, $bucket(C)$, and $time(T)$, respectively, in rules (5)–(10):

$$\begin{align} time(0..t_{max}) & \tag{2}\\ bucket(a) \quad bucket(b) & \tag{3}\\ 1\ \{pour(B,T) : bucket(B)\}\ 1 &\leftarrow time(T), T < t_{max} \tag{4}\\ 1 \leq^{\$} amt(B,T) &\leftarrow pour(B,T), T < t_{max} \tag{5}\\ amt(B,T) \leq^{\$} 3 &\leftarrow pour(B,T), T < t_{max} \tag{6}\\ amt(B,T) =^{\$} 0 &\leftarrow not\ pour(B,T), T < t_{max} \tag{7}\\ vol(B,T{+}1) =^{\$} vol(B,T) + amt(B,T) &\leftarrow T < t_{max} \tag{8}\\ down(B,T) &\leftarrow vol(C,T) <^{\$} vol(B,T) \tag{9}\\ up(B,T) &\leftarrow not\ down(B,T) \tag{10}\\ vol(a,0) =^{\$} 0 \quad vol(b,0) =^{\$} 1 & \tag{11}\\ &\leftarrow up(a,t_{max})\ . \tag{12} \end{align}$$

This program describes a balance with two buckets, a and b, at each end. According to (4), we must pour a certain amount of water into exactly one of the buckets at each time point. The amount of added water may vary between 1 and 3. The balance is down at one bucket's side, if the bucket contains more water than the other; otherwise, it is up. Initially, bucket a is empty while b contains 1 unit. The goal is to find sequences of *pour* actions making the side of bucket a be down after t_{max} time steps (cf. (12)).

The above program has the following signature:

$$\begin{array}{rlrl} \{B,C,T\} \subseteq \mathcal{V}_{\mathcal{A}} & & & \\ \{0,\dots,t_{max},+,a,b,amt,vol\} \subseteq \mathcal{F}_{\mathcal{A}} & & \{0,1,3,+\} \subseteq \mathcal{F}_{\mathcal{C}} & \\ \{<,time,bucket,pour,up,down\} \subseteq \mathcal{P}_{\mathcal{A}} & & \{=^{\$},<^{\$},\leq^{\$}\} \subseteq \mathcal{P}_{\mathcal{C}}\ . & \end{array}$$

The contents of $\mathcal{V}_{\mathcal{C}}$ as well as of $\mathcal{A}$ and $\mathcal{C}$ becomes clear when looking at the ground program obtained by instantiating all variables in $\mathcal{V}_{\mathcal{A}}$ with terms from $\mathcal{T}(\mathcal{F}_{\mathcal{A}})$. To see this, let us look at the ground instantiation of rule (7) and (8) obtained from substitution $\{B \mapsto b, T \mapsto 1\}$ along with constant mapping $t_{max} \mapsto 2$, and evaluating $1 < 2$ as well as $1{+}1$ (as done by grounders like *gringo*):

$$\begin{align} amt(b,1) =^{\$} 0 &\leftarrow not\ pour(b,1)\\ vol(b,2) =^{\$} vol(b,1) + amt(b,1) &\leftarrow\ . \end{align}$$

These two ground rules encompass three constraint variables and three atoms:

$$\begin{array}{ll} & \{amt(b,1), vol(b,1), vol(b,2)\} \subseteq \mathcal{V}_{\mathcal{C}}\\ \{pour(b,1)\} \subseteq \mathcal{A} & \{amt(b,1) =^{\$} 0,\ vol(b,2) =^{\$} vol(b,1) + amt(b,1)\} \subseteq \mathcal{C}\ . \end{array}$$

While the actual ASP solver assigns a Boolean value to the constraint atom $vol(b,2) =^{\$} vol(b,1) + amt(b,1)$, the CP solver deals with the associated constraint $\gamma(vol(b,2) =^{\$} vol(b,1) + amt(b,1))$, eventually assigning (integral) values to constraint variables $vol(b,2)$, $vol(b,1)$, and $amt(b,1)$.

For $t_{max} \mapsto 2$, the above program has eleven constraint answer sets, namely, four different Boolean assignments associated with varying constraint assignments, summarized by the following Boolean and constraint variable assignments:

$up(a,0)$	$pour(a,0)$	$amt(a,0)$	$up(a,1)$	$pour(a,1)$	$amt(a,1)$	$up(a,2)$
$\mathbf{T}$	$\mathbf{T}$	1	$\mathbf{T}$	$\mathbf{T}$	$1,2,3$	$\mathbf{F}$
$\mathbf{T}$	$\mathbf{T}$	$2,3$	$\mathbf{F}$	$\mathbf{T}$	$1,2,3$	$\mathbf{F}$
$\mathbf{T}$	$\mathbf{T}$	3	$\mathbf{F}$	$\mathbf{F}$	0	$\mathbf{F}$
$\mathbf{T}$	$\mathbf{F}$	0	$\mathbf{T}$	$\mathbf{T}$	3	$\mathbf{F}$

While the first two groups of answer sets "pour into bucket a" twice, the last two also "pour into bucket b", namely, one unit at either time point 0 or 1.

As a general remark, note that replacing 3 in rule (6) by a significantly larger number (e.g., 30000) does neither affect the size of the ground program nor the number of different Boolean assignments. In fact, the size of the ground instantiation of a program is completely independent of the domain size of its constraint variables. Given the simplicity of the above example, larger domains do also not deteriorate the runtime of a CP solver like *gecode* too much, while they would drastically increase runtime and space required by ASP grounders and solvers.

4 Conflict-Driven Nogood Learning with Constraint Processing

We now develop an algorithm for computing constraint answer sets that extends a previous algorithm to compute standard answer sets [19] by a CP "oracle." The basic algorithm for finding standard answer sets is called Conflict-Driven Nogood Learning (CDNL); it includes conflict-driven learning and backjumping according to the First-UIP scheme [21,22,7]. That is, whenever a conflict happens, a conflict nogood containing a Unique Implication Point (UIP) is identified by iteratively resolving a violated nogood against a second nogood that is a reason for some literal in it. In view of the fact that CP solver *gecode* used in our implementation does not provide any reasons (it only reports whether a conflict has occurred), the extended algorithm works under the assumption that its CP oracle cannot be queried for reasons. Nonetheless, conflict resolution requires some reason when resolving out a literal, and the major difficulty we address is to identify sufficient yet non-trivial reasons outside the CP oracle.

As mentioned before, a standard answer set can be seen as an assignment satisfying certain conditions induced by a program P. A (Boolean) *assignment* $\mathbf{A}$ over domain $atom(P) \cup \{body(r) \mid r \in P\}$ is a sequence $(\sigma_1, \dots, \sigma_m)$ of (*signed*) *literals* σ_i of the form $\mathbf{T}v_i$ or $\mathbf{F}v_i$, where $v_i \in atom(P) \cup \{body(r) \mid r \in P\}$ for $1 \leq i \leq m$; $\mathbf{T}v_i$ expresses that v_i is *true* and $\mathbf{F}v_i$ that it is *false*. (We omit the attribute *signed* for literals whenever clear from the context.) The complement of a literal σ is denoted by $\overline{\sigma}$, that is, $\overline{\mathbf{T}v} = \mathbf{F}v$ and $\overline{\mathbf{F}v} = \mathbf{T}v$, and we let $var(\sigma) = v$. For $\mathbf{A} = (\sigma_1, \dots, \sigma_{i-1}, \sigma_i, \dots)$, $\mathbf{A}[\sigma_i] = (\sigma_1, \dots, \sigma_{i-1})$ is the prefix of $\mathbf{A}$ up to σ_i. We sometimes abuse notation and identify an assignment with the set of its contained literals. Given this, we access the true and false variables in $\mathbf{A}$ via $\mathbf{A}^{\mathbf{T}} = \{v \mid \mathbf{T}v \in \mathbf{A}\}$ and $\mathbf{A}^{\mathbf{F}} = \{v \mid \mathbf{F}v \in \mathbf{A}\}$. For a canonical representation of (Boolean) constraints, we make use of nogoods [10,11].

In our setting, a *nogood* is a finite set $\{\sigma_1,\dots,\sigma_k\}$ of literals, expressing a constraint violated by any assignment $\mathbf{A}$ containing $\sigma_1,\dots,\sigma_k$. The nogoods derived from the completion of P are denoted by Δ_P, and Λ_P contains the ones that are implicitly given by loop formulas (cf. [19]). An assignment $\mathbf{A}$ is a *solution* for P if $\mathbf{A}^{\mathbf{T}} \cap \mathbf{A}^{\mathbf{F}} = \emptyset$, $\mathbf{A}^{\mathbf{T}} \cup \mathbf{A}^{\mathbf{F}} = atom(P) \cup \{body(r) \mid r \in P\}$, and $\delta \not\subseteq \mathbf{A}$ for all $\delta \in \Delta_P \cup \Lambda_P$. As shown in [19], $\mathbf{A}^{\mathbf{T}} \cap atom(P)$ is an answer set of P iff $\mathbf{A}$ is a solution for $\Delta_P \cup \Lambda_P$. We skip further details on Δ_P and Λ_P, as they are not affected by adding a CP oracle.[3]

Switching back to constraint logic programs P over $\mathcal{A} \cup \mathcal{C}$, by $\mathbf{A}|_{\mathcal{C}} = \{\mathbf{T}c \in \mathbf{A} \mid c \in \mathcal{C}\} \cup \{\mathbf{F}c \in \mathbf{A} \mid c \in \mathcal{C}\}$, we denote the projection of a Boolean assignment $\mathbf{A}$ to literals over constraint atoms. Furthermore, we associate P with the CSP

$$CSP[P] = \big(V[P] \cup atom(P)|_{\mathcal{C}}, D, \big\{\big((S(\gamma(c)), c), c \equiv R(\gamma(c))\big) \mid c \in atom(P)|_{\mathcal{C}}\big\}\big)$$

where D contains $dom(v) = D[P]$ for every $v \in V[P]$ and $dom(c) = \{\mathbf{T}, \mathbf{F}\}$ for every $c \in atom(P)|_{\mathcal{C}}$.[4] A constraint relation of the form $c \equiv R(\gamma(c))$ is called *reified*: it associates the truth value of $c \in atom(P)|_{\mathcal{C}}$ with the valuation of the corresponding constraint $\gamma(c)$. We below slightly abuse notation by identifying the scope $S(\gamma(c)) = (v_1,\dots,v_k)$ of $\gamma(c)$ with the corresponding set $\{v_1,\dots,v_k\}$.

4.1 Main Algorithm

Our main algorithm for computing a constraint answer set of P is shown in Algorithm 1. It shares with the one in [19] the assignment $\mathbf{A}$, recorded nogoods ∇, and decision level dl but adds a flag $event$ (cf. Line 4 in Algorithm 1), whose admissible values are $assertion$ and $decision$. The purpose of this flag is to enable propagation, invoked in Line 6, to mark derived literals such that blocks can be distinguished: all literals in the same block are derived either by unit propagation on $\Delta_P \cup \Lambda_P \cup \nabla$ or by constraint propagation on $CSP[P]$. In order to retrieve such blocks in conflict analysis, invoked in Line 9 and 23, each literal $\sigma \in \mathbf{A}$ is associated with a reason flag $res(\sigma)$. A block of literals derived by unit propagation starts with a literal σ_{dc} where $res(\sigma_{dc}) = dc$, followed by arbitrarily many literals σ_{up} for which $res(\sigma_{up}) = up$. In turn, a block of literals derived by constraint propagation is given by consecutive literals σ_{cp} such that $res(\sigma_{cp}) = cp$.

After propagation in Line 6, we distinguish the cases of a *conflict* (Line 7–12), a *total assignment* (Line 13–27), or a *partial assignment* (Line 29–33). In the latter case, a heuristic decision needs to be made, and an undecided literal σ_d, whose reason is by decision, is selected (Line 29–30). Furthermore, the decision level is incremented, and σ_d is appended to $\mathbf{A}$ (Line 31–32). Finally, setting $event$ to $decision$ in Line 33 signals to the following propagation step that the last literal in $\mathbf{A}$ is a decision literal. The case of a *conflict* is signaled via a $status$ flag returned by propagation, if its value is either cUP (conflict in unit propagation) or cCP (conflict in constraint propagation). A conflict above decision level 0, i.e., at least one decision literal is involved in the conflict,

[3] The only difference is that atoms of $\mathcal{C}$ are not subject to completion in Δ_P and loop nogoods in Λ_P. That is, they can be assigned to $\mathbf{T}$ without requiring any justification from P.

[4] We assume that $\{\mathbf{T}, \mathbf{F}\} \cap D[P] = \emptyset$. Moreover, we write literal $\mathbf{T}c$ or $\mathbf{F}c$ for $c \in atom(P)|_{\mathcal{C}}$ assigned to either $\mathbf{T}$ or $\mathbf{F}$, respectively.

Algorithm 1. CDNL-ASPMCSP

Input : A constraint logic program P.
Output : A constraint answer set of P.

```
$\mathbf{A} \leftarrow \emptyset$                                   // (Boolean) assignment
$\nabla \leftarrow \emptyset$                                   // set of (dynamic) nogoods
$dl \leftarrow 0$                                   // decision level
$event \leftarrow assertion$                        // propagation mode
loop
    $(\mathbf{A}, \nabla, status) \leftarrow$ PROPAGATION($P, \nabla, \mathbf{A}, event$)
    if $status \in \{cUP, cCP\}$ then
        if $dl = 0$ then exit
        $(\delta, dl) \leftarrow$ CONFLICTANALYSIS($P, \nabla, \mathbf{A}, status$)
        $\nabla \leftarrow \nabla \cup \{\delta\}$
        $\mathbf{A} \leftarrow \mathbf{A} \setminus \{\sigma \in \mathbf{A} \mid dl(\sigma) > dl\}$
        $event \leftarrow assertion$
    else if $\mathbf{A}^{\mathbf{T}} \cup \mathbf{A}^{\mathbf{F}} = atom(P) \cup \{body(r) \mid r \in P\}$ then
        $(A, status) \leftarrow$ LABELING($CSP[P], \mathbf{A}|_{\mathcal{C}}$)
        if $status = conflict$ then
            $dl \leftarrow dl + 1$
            repeat
                $dl \leftarrow \max\{dl(\sigma) \mid \sigma \in \mathbf{A}|_{\mathcal{C}}, dl(\sigma) < dl\}$
                if $dl = 0$ then exit
                $(A, status) \leftarrow$ LABELING($CSP[P], \{\sigma \in \mathbf{A}|_{\mathcal{C}} \mid dl(\sigma) < dl\}$)
            until $status \neq conflict$
            $\mathbf{A} \leftarrow \mathbf{A} \setminus \{\sigma \in \mathbf{A} \mid dl(\sigma) > dl\}$
            $(\delta, dl) \leftarrow$ CONFLICTANALYSIS($P, \nabla, \mathbf{A}, cAS$)
            $\nabla \leftarrow \nabla \cup \{\delta\}$
            $\mathbf{A} \leftarrow \mathbf{A} \setminus \{\sigma \in \mathbf{A} \mid dl(\sigma) > dl\}$
            $event \leftarrow assertion$
        else return $(\mathbf{A}^{\mathbf{T}} \cap \mathcal{A}, \{v \mapsto A(v) \mid v \in V[P]\})$
    else
        $\sigma_d \leftarrow$ SELECT($P, \nabla, \mathbf{A}$)
        $res(\sigma_d) \leftarrow dc$
        $dl \leftarrow dl + 1$
        $\mathbf{A} \leftarrow \mathbf{A} \circ \sigma_d$
        $event \leftarrow decision$
```

is analyzed in Line 9. It results in a new nogood δ, recorded in Line 10, that implies the complement of a UIP by unit propagation at a decision level to which backjumping returns to in Line 11. (Note that $dl(\sigma)$ provides the decision level at which σ has been assigned.) Finally, by setting $event$ to $assertion$ in Line 12, we signal the following propagation step that δ will be asserting.

The treatment of a *total assignment* is the main difference to the algorithm in [19]. Before also solving $CSP[P]$, we cannot be sure whether Boolean assignment $\mathbf{A}$ belongs to a constraint answer set of P. Thus, the CP oracle is queried whether there is a solution A for $CSP[P]$ (Line 14), given the truth values assigned to atoms of $\mathcal{C}$ in $\mathbf{A}$. If such a solution A exists, we have found a constraint answer set that is returned in

Line 27. Note that the additional check is necessary because the CP oracle is not permitted to make choices during constraint propagation, which in general will not be able to assign all variables in $V[P]$ or to detect unsatisfiability, given only the truth values of Boolean variables shared with $atom(P)$. This is also the reason why, in case of unsatisfiability detected now, we do not know from which decision level on $CSP[P]$ had actually been unsatisfiable wrt the literals over $\mathcal{C}$ in $\mathbf{A}$. Hence, the loop in Line 17–21 successively backtracks through $\mathbf{A}$ until hitting a decision level dl such that $CSP[P]$ can be satisfied, given only the literals over $\mathcal{C}$ in $\mathbf{A}$ from decision levels smaller than dl. Then, conflict analysis is invoked in Line 23 with cAS signaling a conflict on a putative constraint answer set. Conflict-driven learning, backjumping, and the following assertion (Line 24–26) are similar to a conflict encountered by propagation.

4.2 Propagation Algorithm

The main idea of our propagation procedure, shown in Algorithm 2, is to iterate unit propagation on $\Delta_P \cup \nabla$, unfounded set detection accompanied by selective recording of nogoods from Λ_P, and finally constraint propagation on $CSP[P]$. Before this process starts, we set a flag cp to $true$ if constraint propagation should be performed initially (Line 1) or in reaction to a decision literal over $\mathcal{C}$ (Line 2). Otherwise, cp is made $false$ (Line 3), given that $\mathbf{A}$ has not been extended by literals over constraint atoms since the last constraint propagation step.

Conflicts during unit propagation are in Line 6 detected via some nogood from $\Delta_P \cup \nabla$ violated by $\mathbf{A}$, and they are signaled via return value cUP. If there is no conflict, we in Line 7 check whether there are nogoods δ that contain a single unassigned literal, while all other literals belong to $\mathbf{A}$. Then, unit propagation infers the complement $\overline{\sigma}$ of such a last unassigned literal σ in order to avoid the inclusion of δ in $\mathbf{A}$. As mentioned above, we use a flag $res(\overline{\sigma})$ to later on identify a block of literals derived by unit propagation. The value of $event$ now determines whether a new block begins (Line 10–11), which is marked by setting $res(\overline{\sigma})$ to dc, or an existing one is extended (Line 12). Finally, flag cp is set in Line 13 if $\overline{\sigma}$ is over an atom of $\mathcal{C}$. After reaching a fixpoint of unit propagation without any conflict, unfounded set handling (cf. [19]) is performed for non-tight [23] programs in Line 17–19. Note that an already identified nonempty unfounded set needs first to be falsified completely before a new (nonempty) unfounded set $U \subseteq atom(P)|_{\mathcal{A}} \setminus \mathbf{A}^{\mathbf{F}}$ is determined in Line 18 (if no such U exists, UNFOUNDEDSET returns $\emptyset$). Finally, atoms in a nonempty unfounded set U will be falsified by unit propagation after adding a loop nogood from Λ_P to ∇ in Line 19.

Finally, constraint propagation (Line 21–32) takes place only if unit propagation cannot infer any further literal, checked via $U = \emptyset$ in Line 20. Furthermore, we are sure that no new literals over $atom(P)|_{\mathcal{C}}$ will be derived if none was recently added to $\mathbf{A}$ (if $cp = false$), in which case the whole propagation terminates in Line 21. Otherwise, constraint propagation in Line 22 may result either in a conflict (Line 23), signaled via return value cCP, or in an assignment A over $V[P] \cup atom(P)|_{\mathcal{C}}$, whose possible additions to $\mathbf{A}$ on the common constraint atoms are provided by B (Line 24). If additions to $\mathbf{A}$ are possible ($B = \emptyset$ does not hold in Line 25), we do them in Line 26–30, and the reason flags of the derived literals are set to cp (Line 28). Afterwards, flag cp is reset to $false$ in Line 31, so that another constraint propagation step will be performed only

Algorithm 2. PROPAGATION

```
Input  : A constraint logic program $P$, a set $\nabla$ of nogoods, a (Boolean) assignment $\mathbf{A}$,
         and an $event \in \{decision, assertion\}$.
Output : A (Boolean) assignment, set of nogoods, and a $status \in \{cUP, cCP, fix\}$.
1  if $\mathbf{A} = \emptyset$ then $cp \leftarrow true$                          // do initial constraint propagation
2  else if $event = decision$ and $var(\sigma) \in \mathcal{C}$ where $\mathbf{A} = \mathbf{A}' \circ \sigma$ then $cp \leftarrow true$
3  else $cp \leftarrow false$
4  $U \leftarrow \emptyset$                                                        // unfounded set
5  loop
6      if $\delta \subseteq \mathbf{A}$ for some $\delta \in \Delta_P \cup \nabla$ then return $(\mathbf{A}, \nabla, cUP)$
7      $\Sigma \leftarrow \{\delta \in \Delta_P \cup \nabla \mid \delta \setminus \mathbf{A} = \{\sigma\}, \overline{\sigma} \notin \mathbf{A}\}$
8      if $\Sigma \neq \emptyset$ then let $\delta \setminus \mathbf{A} = \{\sigma\}$ for some $\delta \in \Sigma$ in
9          if $event = assertion$ then
10             $res(\overline{\sigma}) \leftarrow dc$
11             $event \leftarrow decision$
12         else $res(\overline{\sigma}) \leftarrow up$
13         if $var(\overline{\sigma}) \in \mathcal{C}$ then $cp \leftarrow true$           // redo constraint propagation
14         $\mathbf{A} \leftarrow \mathbf{A} \circ \overline{\sigma}$
15     else
16         if $P$ is non-tight then
17             $U \leftarrow U \setminus \mathbf{A}^{\mathbf{F}}$
18             if $U = \emptyset$ then $U \leftarrow$ UNFOUNDEDSET$(P, \mathbf{A})$
19             if $U \neq \emptyset$ then let $a \in U$ in $\nabla \leftarrow \nabla \cup \{\lambda(a, U)\}$
20         if $U = \emptyset$ then
21             if $cp = false$ then return $(\mathbf{A}, \nabla, fix)$
22             $(A, status) \leftarrow$ CONSTRAINTPROPAGATION$(CSP[P], \mathbf{A}|_{\mathcal{C}})$
23             if $status = conflict$ then return $(\mathbf{A}, \nabla, cCP)$
24             $B \leftarrow \{\mathbf{T}c \in A \mid \mathbf{T}c \notin \mathbf{A}\} \cup \{\mathbf{F}c \in A \mid \mathbf{F}c \notin \mathbf{A}\}$
25             if $B = \emptyset$ then return $(\mathbf{A}, \nabla, fix)$
26             repeat
27                 $B \leftarrow B \setminus \{\sigma\}$ for some $\sigma \in B$
28                 $res(\sigma) \leftarrow cp$
29                 $\mathbf{A} \leftarrow \mathbf{A} \circ \sigma$
30             until $B = \emptyset$
31             $cp \leftarrow false$
32             $event \leftarrow assertion$
```

after inferring further literals over $atom(P)|_{\mathcal{C}}$ by unit propagation. Finally, flag $event$ is set to $assertion$, which has the effect that the next literal $\overline{\sigma}$ inferred by unit propagation (if any is inferred) will be marked as the first one of a new block via dc for $res(\overline{\sigma})$. In view of the fact that constraint propagation may extend $\mathbf{A}$ with further literals, we note that our propagation technique matches "theory propagation" [9] in SMT solvers.

4.3 Conflict Analysis Algorithm

On every conflict beyond decision level 0, our conflict analysis procedure in Algorithm 3 identifies a new nogood according to the First-UIP scheme. While a literal

Algorithm 3. CONFLICTANALYSIS

```
Input  : A constraint logic program $P$, a set $\nabla$ of nogoods, a (Boolean) assignment $\mathbf{A}$,
         and a $status \in \{cUP, cCP, cAS\}$.
Output : A derived nogood and a decision level.

1  if $status = cAS$ then
2      $\delta \leftarrow \{\sigma \in \mathbf{A}|_{\mathcal{C}} \mid dl(\sigma) = \max\{dl(\sigma') \mid \sigma' \in \mathbf{A}\}\}$
3      repeat
4          $touched \leftarrow \delta$
5          $\delta \leftarrow \{\sigma \in \mathbf{A}|_{\mathcal{C}} \mid S(\gamma(var(\sigma))) \cap \bigcup_{\sigma' \in \delta} S(\gamma(var(\sigma'))) \neq \emptyset\}$
6      until $\delta = touched$
7  else if $status = cCP$ then
8      let $\sigma \in \mathbf{A}$ such that $res(\sigma) = dc$ and $\forall \sigma' \in \mathbf{A} \setminus (\mathbf{A}[\sigma] \cup \{\sigma\}) : res(\sigma') = up$ in
9          $\delta \leftarrow \mathbf{A}|_{\mathcal{C}} \setminus \mathbf{A}[\sigma]$
10     repeat
11         $touched \leftarrow \delta$
12         $\delta \leftarrow \{\sigma \in \mathbf{A}|_{\mathcal{C}} \mid S(\gamma(var(\sigma))) \cap \bigcup_{\sigma' \in \delta} S(\gamma(var(\sigma'))) \neq \emptyset\}$
13     until $\delta = touched$
14 else
15     $\delta \leftarrow \varepsilon$ for some $\varepsilon \in \Delta_P \cup \nabla$ such that $\varepsilon \subseteq \mathbf{A}$
16     $touched \leftarrow \emptyset$
17 loop
18     let $\sigma \in \delta$ such that $\delta \setminus \mathbf{A}[\sigma] = \{\sigma\}$ in
19         $k \leftarrow \max\{dl(\sigma') \mid \sigma' \in \delta \setminus \{\sigma\}\}$
20         if $k = dl(\sigma)$ then
21             if $res(\sigma) = cp$ then
                   $\mathbf{A} \leftarrow \mathbf{A} \setminus \{\sigma' \in \mathbf{A} \setminus \mathbf{A}[\sigma] \mid$
22                     $\exists \sigma'' \in (\mathbf{A}[\sigma'] \setminus \mathbf{A}[\sigma]) \cup \{\sigma'\} : res(\sigma'') \neq cp\}$
23                 $\varepsilon \leftarrow \{\sigma' \in \delta \cap \mathbf{A} \mid \forall \sigma'' \in \mathbf{A}[\sigma] \setminus \mathbf{A}[\sigma'] : res(\sigma'') = cp\} \setminus touched$
24                 while $\varepsilon \not\subseteq touched$ do
25                     $touched \leftarrow touched \cup \varepsilon$
26                     $\varepsilon \leftarrow \{\sigma'' \in \mathbf{A}|_{\mathcal{C}} \mid S(\gamma(var(\sigma''))) \cap \bigcup_{\sigma' \in \varepsilon} S(\gamma(var(\sigma'))) \neq \emptyset\}$
27                 $\mathbf{A} \leftarrow \mathbf{A} \setminus \{\sigma' \in \mathbf{A} \mid \forall \sigma'' \in \mathbf{A} \setminus \mathbf{A}[\sigma'] : res(\sigma'') = cp\}$
28                 $\delta \leftarrow (\delta \cup \varepsilon) \cap \mathbf{A}$
29             else let $\varepsilon \in \Delta_P \cup \nabla$ such that $\varepsilon \setminus \mathbf{A}[\sigma] = \{\overline{\sigma}\}$ in $\delta \leftarrow (\delta \setminus \{\sigma\}) \cup (\varepsilon \setminus \{\overline{\sigma}\})$
30         else return $(\delta, k)$
```

derived by unit propagation has at least one reason in $\Delta_P \cup \nabla$, no such reasons exist for literals derived by constraint propagation. Since we assume that the CP oracle does also not provide us with reasons, we can merely try to reconstruct a non-trivial reason (one that does not include all previously assigned literals over $\mathcal{C}$) from structural properties of $CSP[P]$. Our approach for this is inspired by graph-based backjumping/learning [10] where, for a variable in question, other variables it shares constraints with are considered as potential reasons. In fact, we identify a sufficient reason by considering all literals over $atom(P)|_{\mathcal{C}}$ assigned prior to a constraint atom c and connected to c via their scopes, starting from literals σ with $S(\gamma(var(\sigma))) \cap S(\gamma(c)) \neq \emptyset$.

The sketched strategy is applied when a conflict is due to a putative answer set (Line 1–6), where $CSP[P]$ is unsatisfiable under the current assignment $\mathbf{A}$. Furthermore, since the backtracking scheme in Algorithm 1 guarantees satisfiability when taking only the literals in $\mathbf{A}$ at smaller decision levels than the current one, we also know that literals over $\mathcal{C}$ at the maximum decision level are involved in the conflict. Hence, we take them as initial reason δ for the conflict (Line 2), and iteratively add all literals in $\mathbf{A}$ over $\mathcal{C}$ connected to some literal in δ via non-disjoint scopes (Line 3–6). The so obtained δ provides a sufficient reason for the unsatisfiability of $CSP[P]$; it is processed further using the standard First-UIP scheme (described below). If the conflict at hand has been encountered during constraint propagation (Line 7–13), we know that the literals over $\mathcal{C}$ in the last block derived by unit propagation (determined in Line 8–9) are involved. In Line 10–13, we then use the same technique as above to identify a sufficient reason δ for the conflict. Finally, if the conflict has been detected during unit propagation (Line 15–16), we can simply determine some violated nogood δ in $\Delta_P \cup \nabla$.

The loop in Line 17–30 eventually exploits the First-UIP scheme, eliminating literals from δ until it contains exactly one literal σ from the current decision level. If this is not yet the case (tested in Line 20), some σ in δ needs to be replaced with a reason why it was included in $\mathbf{A}$. Here, we distinguish the cases that σ has been derived by constraint propagation (Line 21–28) or by unit propagation (Line 29). In the latter case, we can simply resolve δ against a known nogood ε in $\Delta_P \cup \nabla$, as done in [19]. Otherwise, determining an appropriate reason is more sophisticated. In fact, in Line 22, we eliminate all successors of σ in $\mathbf{A}$ that do not belong to the same block as σ of literals derived by constraint propagation. This reflects that the removed literals cannot have contributed to the CP oracle deriving σ. In Line 23, we then determine in ε all literals (over $\mathcal{C}$) of δ that belong to the same block as σ in order to explore their constraint interdependencies, where an optimization consists of ignoring literals in $touched$, given that they have been explored already. In Line 24–26, we further extend ε with connected literals over $\mathcal{C}$, like in Line 3–6 and Line 10–13. Finally, we remove the whole block of σ from $\mathbf{A}$ and δ (Line 27–28), as possible contributions to the conflict have been explored exhaustively, while the remaining literals of ε are added to δ.

In comparison to [19], it is apparent that the accommodation of a CP oracle makes the required computations more sophisticated, as extra information is needed to distinguish literals derived by constraint propagation from those inferred by unit propagation. The identification of appropriate reasons is a major bottleneck in a conflict-driven learning ASP solver, in particular, if the CP oracle does not support it. In such a case, workarounds are needed to approximate sufficient reasons. Their impact regarding computational cost is empirically investigated in the next section.

5 Experiments

We implemented our approach to constraint answer set solving within the new solver *clingcon* (0.1.0;[16]), extending ASP system *clingo* (2.0.2;[17]) with the generic CP solver *gecode* (2.2.0;[18]). Our experiments consider *clingcon* using four different approaches to incorporate constraint propagation: (a) lazy reason calculation during conflict analysis exploiting constraint interdependencies, as shown in Algorithm 3; (b)

Table 1. Comparing *clingo*, *adsolver*, and *clingcon*

	clingo	*adsolver*				*clingcon*				
Benchmark	5	5	7	11	13	5	7	11	13	20
3-0/025	162.84	14.74	51.42	460.57	365.37	1.19	1.97	4.21	5.99	17.84
3-0/050	173.28	31.39	108.21	471.41	—	1.26	2.32	6.80	11.85	27.36
3-0/100	175.94	448.90	188.33	—	—	1.32	2.35	10.11	12.04	38.78
3-0/125	165.64	19.78	60.07	224.60	—	1.18	1.94	4.05	10.00	133.99
5-0/025	174.12	28.78	107.41	—	—	1.28	2.90	5.87	14.27	66.55
5-0/050	163.25	13.57	42.00	204.34	497.64	1.18	1.97	4.71	10.04	241.59
5-0/100	168.16	21.50	66.10	282.36	514.08	1.20	1.98	4.13	6.45	25.32
5-0/125	174.38	32.02	104.32	429.72	—	1.34	2.95	6.39	9.70	81.17
8-0/025	177.82	41.57	140.93	—	—	1.30	2.73	11.00	12.69	222.49
8-0/050	167.72	18.83	54.76	215.43	—	1.18	1.93	4.02	7.76	457.86
8-0/100	165.55	13.72	41.03	208.74	—	1.21	2.00	5.05	6.10	26.17
8-0/125	162.29	16.81	53.40	246.64	519.59	1.20	1.99	4.15	6.69	17.82
∅	169.25	58.47	84.83	378.65	558.06	1.24	2.25	5.87	9.47	113.08

immediate reason recording for literals derived by constraint propagation (discussed in the context of SMT in [9, Section 5.1]) exploiting constraint interdependencies; (c) lazy reason calculation during conflict analysis without using constraint interdependencies, rather, taking all assigned literals over $\mathcal{C}$ as trivial reason; (d) immediate reason recording for literals derived by constraint propagation without using constraint interdependencies. We also include *adsolver* (1.55;[14]), combining an (extended) ASP solver with a CP solver for difference constraints. Our experiments consider a benchmark suite stemming from decision support systems for NASA shuttle controllers [2,3][5], which involve mapping logical time steps on real-time. All experiments were run on a 3.4GHz PC under Linux. We report results in seconds, taking the average of three runs, each restricted to 600s time and 3GB RAM.[6]

Table 1 compares *clingo*, *adsolver*, and variant (c) of *clingcon*, which turned out to be the best choice on the considered benchmarks (see below), on 12 randomly picked sample instances and varying number of logical time steps. The instances stem from the *instances-monica* folder, “3-0/025” means subfolder *ins-3-0* instance *instance_025*. Average runtimes over all instances are provided in the last row of Table 1, taking timeouts as maximum time, viz., 600s. Using *clingo* (on direct ASP encodings), we can solve the instances for 5 time steps, where the major effort is made in grounding; in fact, we observed memory exhaustion on all instances for 7 time steps. With *adsolver*, such space problems do not occur, but it runs into timeouts (indicated by —) for 11 and 13 time steps. Up to these time steps, *clingcon* still scales well and is an order of magnitude faster than *adsolver*. The last column shows results for the greatest step number, 20, for which *clingcon* solved all instances within 600s.

Table 2 compares the four different settings of *clingcon* with each other. We observed that exploiting constraint interdependencies in variants (a) and (b) may decrease the

[5] `http://www.krlab.cs.ttu.edu/Software/Download/rcs`

[6] Much main memory is needed solely for grounding in *clingo*.

Table 2. Comparing different strategies within *clingcon*

	11				13				20			
Benchmark	(a)	(b)	(c)	(d)	(a)	(b)	(c)	(d)	(a)	(b)	(c)	(d)
3-0/025	12.83	4.90	4.21	4.21	36.71	8.09	5.99	5.90	—	64.89	17.84	18.78
3-0/050	49.21	13.25	6.80	8.42	219.01	35.38	11.85	12.41	—	370.71	27.36	30.46
3-0/100	36.44	9.54	10.11	5.30	34.82	53.96	12.04	7.16	—	—	38.78	33.67
3-0/125	6.31	4.25	4.05	4.07	163.47	21.09	10.00	8.52	—	377.06	133.99	31.46
5-0/025	69.36	15.38	5.87	10.52	176.81	23.39	14.27	15.06	—	—	66.55	118.65
5-0/050	24.40	6.58	4.71	4.67	311.11	22.93	10.04	7.19	—	542.27	241.59	—
5-0/100	6.39	4.30	4.13	4.11	72.95	6.36	6.45	5.70	—	83.85	25.32	49.03
5-0/125	61.72	17.66	6.39	6.96	111.92	41.84	9.70	10.87	—	—	81.17	73.68
8-0/025	76.73	7.69	11.00	9.25	248.29	37.33	12.69	7.81	—	—	222.49	—
8-0/050	6.33	4.19	4.02	4.05	189.04	15.75	7.76	8.05	—	52.14	457.86	32.29
8-0/100	7.71	4.49	5.05	4.16	220.12	11.71	6.10	5.94	—	159.37	26.17	23.72
8-0/125	5.11	4.31	4.15	4.14	38.91	15.31	6.69	7.06	—	69.64	17.82	18.80
∅	30.21	8.05	5.87	5.82	151.93	24.43	9.47	8.47	—	343.33	113.08	135.88

number of heuristic decisions made by *clingcon*. As regards runtime, it however turns out that the additional effort does not pay off. For one, this is because the calculation of constraint interdependencies is not yet fully optimized in *clingcon*, and the overhead could possibly be reduced. This also explains why variant (b), more eagerly recording reasons than (a), is faster: storing more reasons permits more inferences by unit propagation, and thus, it reduces calculations of constraint interdependencies. However, variants (c) and (d) using simple-to-compute trivial reasons still seem to be superior. Interestingly, the lazy approach of (c) to calculate reasons during conflict analysis performs better than (d) recording reasons during propagation, which is converse to the relationship between (a) and (b). This shift of behaviors can be explained by the overhead of reason calculation: while it is expensive with (a) so that recording more reasons with (b) helps, it is cheap with (c), and exhaustive reason recording in (d) slows down unit propagation more than additional inferences pay off.

6 Discussion

We introduced a novel approach to integrating CP capacities into modern ASP solvers based on conflict-driven learning and backjumping. Our semantic approach relies on a propositional language rather than a multi-sorted, first-order language, as used in [12,13,14]. Also, we follow the lazy approach of advanced SMT solvers by abstracting from the constraints in a specialized theory [9]. A major difficulty in this endeavor was the current lack of CP solvers providing an interface supporting conflict-driven learning. We addressed this problem by developing a new algorithmic framework for incorporating a CP "oracle" into the approach to conflict-driven ASP solving introduced in [19]. Apart from extending unit propagation through constraint propagation, the major extension dealt with conflict analysis and the elaboration of reasons for atoms derived by constraint propagation.

Our approach differs in several ways from the ones developed in [12,13,14]. As mentioned above, our semantic approach is propositional and abstracts from the constraints in a specialized theory. Unlike this, [12,13,14] start with a multi-sorted, first-order language leading to a propositional program through grounding. As well, they rely upon so-called mixed predicates for linking constants with constraint variables. Also, [12,13,14] use traditional ASP solving algorithms, based on DPLL-style backtracking. In fact, *adsolver*'s implementation relies on *lparse* and *smodels*. The implementation described in [13] allows the usage of difference constraints of the form $X - Y > k$ for variables X, Y and constant k; at most one such constraint is allowed within an integrity constraint. The underlying CP solver is handcrafted and thus supports incremental solving and backtracking. Unlike this, we use with *gecode* an off-the-shelf CP system. Although it is incremental, backtracking and reason generation must be dealt with externally. In [24], "functional oracles" allow for computing instantiations of so-called external predicates during grounding. Constraint atoms in our sense can also be viewed as being external to some extent, given that the associated constraints are evaluated by a CP engine. Importantly, the non-Boolean domains of variables in such constraint are still present in the solving phase, while a functional oracle would have to make the domains explicit for constructing a propositional program under standard answer set semantics.

We have empirically evaluated *adsolver* and *clingcon* on the benchmark suite used to appraise *adsolver*'s performance in [13,14]. First of all, we note that both systems escape the grounding bottleneck faced by traditional ASP systems like *clingo*. All in all, however, we observed that *clingcon* outperforms *adsolver* by up to two orders of magnitude. Also, we investigated the effect of different variants of reason generation on the performance of *clingcon*. As regards the current prototype, it turned out that additional efforts into the elaboration of constraint interdependencies do not pay off. However, this issue deserves further attention and is subject to future research.

Acknowledgments. We are grateful to Michael Gelfond and Yuanlin Zhang for useful discussions on the subject of this paper. This work was partially funded by DFG under Grant SCHA 550/8-1 and by the GoFORSYS[7] project under Grant 0313924.

References

1. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press, Cambridge (2003)
2. Nogueira, M., Balduccini, M., Gelfond, M., Watson, R., Barry, M.: An A-prolog decision support system for the space shuttle. In: Ramakrishnan, I.V. (ed.) PADL 2001. LNCS, vol. 1990, pp. 169–183. Springer, Heidelberg (2001)
3. Balduccini, M., Gelfond, M., Nogueira, M.: Answer set based design of knowledge systems. Annals of Mathematics and Artificial Intelligence 47(1-2), 183–219 (2006)
4. Baral, C., Chancellor, K., Tran, N., Tran, N., Joy, A., Berens, M.: A knowledge based approach for representing and reasoning about signaling networks. In: Proceedings of the Twelfth International Conference on Intelligent Systems for Molecular Biology/Third European Conference on Computational Biology (ISMB 2004/ECCB 2004), pp. 15–22 (2004)

[7] `http://www.goforsys.org`

5. Dworschak, S., Grote, T., König, A., Schaub, T., Veber, P.: Tools for representing and reasoning about biological models in action language $\mathcal{C}$. In: Pagnucco, M., Thielscher, M. (eds.) Proceedings of the Twelfth International Workshop on Nonmonotonic Reasoning (NMR 2008). The University of New South Wales, Technical Report Series, pp. 94–102 (2008)
6. Gebser, M., Schaub, T., Thiele, S., Usadel, B., Veber, P.: Detecting inconsistencies in large biological networks with answer set programming. In: Garcia de la Banda, M., Pontelli, E. (eds.) ICLP 2008. LNCS, vol. 5366, pp. 130–144. Springer, Heidelberg (2008)
7. Mitchell, D.: A SAT solver primer. Bulletin of the European Association for Theoretical Computer Science 85, 112–133 (2005)
8. Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.): Handbook of Satisfiability. IOS Press, Amsterdam (2009)
9. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT modulo theories: From an abstract Davis-Putnam-Logemann-Loveland procedure to DPLL(T). Journal of the ACM 53(6), 937–977 (2006)
10. Dechter, R.: Constraint Processing. Morgan Kaufmann Publishers, San Francisco (2003)
11. Rossi, F., van Beek, P., Walsh, T. (eds.): Handbook of Constraint Programming. Elsevier, Amsterdam (2006)
12. Baselice, S., Bonatti, P., Gelfond, M.: Towards an integration of answer set and constraint solving. In: Gabbrielli, M., Gupta, G. (eds.) ICLP 2005. LNCS, vol. 3668, pp. 52–66. Springer, Heidelberg (2005)
13. Mellarkod, V., Gelfond, M.: Integrating answer set reasoning with constraint solving techniques. In: Garrigue, J., Hermenegildo, M. (eds.) FLOPS 2008. LNCS, vol. 4989, pp. 15–31. Springer, Heidelberg (2008)
14. Mellarkod, V., Gelfond, M., Zhang, Y.: Integrating answer set programming and constraint logic programming. Annals of Mathematics and Artificial Intelligence (to appear, 2008)
15. Simons, P., Niemelä, I., Soininen, T.: Extending and implementing the stable model semantics. Artificial Intelligence 138(1-2), 181–234 (2002)
16. `http://www.cs.uni-potsdam.de/clingcon`
17. `http://potassco.sourceforge.net`
18. `http://www.gecode.org`
19. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Conflict-driven answer set solving. In: Veloso, M. (ed.) Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI 2007), pp. 386–392. AAAI Press/MIT Press, Menlo Park (2007)
20. Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., Thiele, S.: A user's guide to gringo, clasp, clingo, and iclingo. In: [17]
21. Marques-Silva, J., Sakallah, K.: GRASP: A search algorithm for propositional satisfiability. IEEE Transactions on Computers 48(5), 506–521 (1999)
22. Zhang, L., Madigan, C., Moskewicz, M., Malik, S.: Efficient conflict driven learning in a Boolean satisfiability solver. In: Proceedings of the International Conference on Computer-Aided Design (ICCAD 2001), pp. 279–285 (2001)
23. Fages, F.: Consistency of Clark's completion and the existence of stable models. Journal of Methods of Logic in Computer Science 1, 51–60 (1994)
24. Calimeri, F., Cozza, S., Ianni, G.: External sources of knowledge and value invention in logic programming. Annals of Mathematics and Artificial Intelligence 50(3-4), 333–361 (2007)

On the Implementation of Weight Constraint Rules in Conflict-Driven ASP Solvers

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

Universität Potsdam, Institut für Informatik, August-Bebel-Str. 89, D-14482 Potsdam

Abstract. We present the first comprehensive approach to integrating cardinality and weight rules into conflict-driven ASP solving. We begin with a uniform, constraint-based characterization of answer sets in terms of nogoods. This provides the semantic underpinnings of our approach in fixing all necessary inferences that must be supported by an appropriate implementation. We then provide key algorithms detailing the salient features needed for implementing weight constraint rules. This involves a sophisticated unfounded set checker as well as an extended propagation algorithm along with the underlying data structures. We implemented our techniques within the ASP solver *clasp* and demonstrate their effectiveness by an experimental evaluation.

1 Introduction

One of the most appealing features of Answer Set Programming (ASP; [1]) is its rich declarative modeling language. Among the most popular language constructs are cardinality and weight constraints [2] being particular forms of count and sum aggregates.

Existing techniques for implementing such aggregates fall into two categories. Traditional backtracking-oriented ASP solvers like *smodels* [2] use counter-based algorithms based on [3]. On the other hand, SAT-based ASP solvers like *cmodels* [4] eliminate such aggregates by transforming them into normal (or nested) logic programming rules. While the former approach has proven its versatility, it does not carry over to modern ASP solving technology based on backjumping and conflict-driven learning [5,6]. Although this is accomplishable by the transformational approach, it fails to scale due to a significant increase in space [7].

We address this problem and present the first comprehensive approach to integrating weight constraint rules into conflict-driven ASP solving. To this end, we begin with a uniform, constraint-based characterization of answer sets in terms of nogoods. This provides the semantic underpinnings of our approach in fixing all necessary inferences that must be supported by an appropriate implementation. We then provide key algorithms detailing the salient features needed for implementing weight constraint rules. This involves a sophisticated unfounded set checker as well as an extended propagation algorithm along with the underlying data structures. Our techniques are implemented within the ASP solver *clasp* [8]. We evaluate the performance of *clasp* relative to the two existing approaches and thus demonstrate its effectiveness.

* Affiliated with Simon Fraser University, Canada, and Griffith University, Australia.

2 Background

Following [2], we consider *weight constraint programs* over an alphabet $\mathcal{A}$, consisting of *weight rules* of the form

$$v\,\{a_0 = 1\} \leftarrow w\,\{a_1 = w_1, \dots, a_m = w_m, {\sim}a_{m+1} = w_{m+1}, \dots, {\sim}a_n = w_n\} \quad (1)$$

where $v \in \{0,1\}$, w is a non-negative integer, w_i are positive integers for $1 \leq i \leq n$, and a_i are *atoms* in $\mathcal{A}$ for $0 \leq i \leq n$. Furthermore, we assume $a_j \neq a_k$ for $0 < j < k \leq m$ and $m < j < k \leq n$, respectively. The set of atoms occurring in a weight constraint program Π is denoted by $A(\Pi)$. A *weight literal* is of the form $a = w$ or ${\sim}a = w$; a and ${\sim}a$ are *regular literals*, where ${\sim}$ stands for default negation. For a set A of atoms, we let ${\sim}A = \{{\sim}a \mid a \in A\}$; for a set L of regular literals, let $L^+ = \{a \mid a \in L \cap \mathcal{A}\}$ and $L^- = \{a \mid a \in L \cap {\sim}\mathcal{A}\}$. We define $A(l = w) = a$ for the atom in a weight literal $l = a$ or $l = {\sim}a$, respectively, and $W(l = w) = w$ for its weight. Accordingly, for a set $\mathcal{L}$ of weight literals, $A(\mathcal{L}) = \{A(\ell) \mid \ell \in \mathcal{L}\}$ and $\Sigma[\mathcal{L}] = \sum_{\ell \in \mathcal{L}} W(\ell)$.

For a rule r as in (1), let $H(r) = v\,\{a_0 = 1\}$ be the *head* of r, $B(r) = w\,\{a_1 = w_1, \dots, a_m = w_m, {\sim}a_{m+1} = w_{m+1}, \dots, {\sim}a_n = w_n\}$ the *body* of r, and $lb(B(r)) = w$ the *lower bound* of $B(r)$. Such a body constitutes a *weight constraint*. We extend the above projections to weight constraints as follows. Given $\mathcal{W} = B(r)$ for r as in (1), we define $\mathcal{W}^+ = \{a_1 = w_1, \dots, a_m = w_m\}$, $\mathcal{W}^- = \{{\sim}a_{m+1} = w_{m+1}, \dots, {\sim}a_n = w_n\}$, and $A(\mathcal{W}) = A(\mathcal{W}^+ \cup \mathcal{W}^-)$. A set X of atoms satisfies $\mathcal{W}$, written $X \models \mathcal{W}$, if

$$\Sigma[\{p \in \mathcal{W}^+ \mid A(p) \in X\} \cup \{n \in \mathcal{W}^- \mid A(n) \notin X\}] \;\geq\; lb(\mathcal{W})\,.$$

That is, a weight constraint is satisfied if the sum of the weights of its satisfied literals does not fall below the lower bound given by w. Accordingly, rule r is satisfied by X, written $X \models r$, if $X \models B(r)$ implies $X \models H(r)$; and $X \models \Pi$ if $X \models r$ for all $r \in \Pi$.

For a rule r as in (1) and a set X of atoms, the reduct of $B(r)$ wrt X is defined as

$$B(r)^X = w'\,B(r)^+ \text{ where } w' = \max\left\{0\,, lb(B(r)) - \Sigma[\{n \in B(r)^- \mid A(n) \notin X\}]\right\}.$$

Given this, the *reduct* of a weight constraint program Π wrt X is

$$\Pi^X = \left\{\, 1\,\{a_0 = 1\} \leftarrow B(r)^X \mid r \in \Pi, A(H(r)) \cap X = \{a_0\} \,\right\}.$$

Finally, X is an *answer set* of Π if $X \models \Pi$ and $Y \not\models \Pi^X$ for all $Y \subset X$.

As detailed in [2], weight constraint rules are expressive enough to (linearly) capture normal rules, integrity constraints, cardinality rules, and general weight constraint rules of the form $\mathcal{W}_0 \leftarrow \mathcal{W}_1, \dots, \mathcal{W}_n$, where $\mathcal{W}_i$ is a general weight constraint for $0 \leq i \leq n$.

3 Inferences from Weight Constraint Programs

This section provides the logical fundament of the computational techniques detailed in Section 4. To this end, we adapt the nogood-based characterization of answer sets from [8] to accommodate weight constraints. As a result, we obtain a clear semantic framework to specify (unit) propagation over weight rules.

An *assignment* $\mathbf{A}$ over a *domain*, $dom(\mathbf{A})$, is a sequence $(\sigma_1, \dots, \sigma_n)$ of (*signed*) *literals* σ_i of the form $\mathbf{T}v_i$ or $\mathbf{F}v_i$, where $v_i \in dom(\mathbf{A})$ for $1 \leq i \leq n$; $\mathbf{T}v_i$ expresses that v_i is *true* and $\mathbf{F}v_i$ that it is *false*. (We omit the attribute *signed* for literals whenever clear from the context.) The complement of a literal σ is denoted by $\overline{\sigma}$, that is, $\overline{\mathbf{T}v} = \mathbf{F}v$ and $\overline{\mathbf{F}v} = \mathbf{T}v$. We sometimes abuse notation and identify an assignment with the set of its contained literals. Given this, we access the true and false variables in $\mathbf{A}$ via $\mathbf{A}^{\mathbf{T}} = \{v \mid \mathbf{T}v \in \mathbf{A}\}$ and $\mathbf{A}^{\mathbf{F}} = \{v \mid \mathbf{F}v \in \mathbf{A}\}$. For a canonical representation of (Boolean) constraints, we make use of nogoods [9]. In our setting, a *nogood* is a finite set $\{\sigma_1, \dots, \sigma_m\}$ of literals, expressing a constraint violated by any assignment $\mathbf{A}$ containing $\sigma_1, \dots, \sigma_m$. For a set Δ of nogoods, an assignment $\mathbf{A}$ is a *solution* for Δ if $\mathbf{A}^{\mathbf{T}} \cap \mathbf{A}^{\mathbf{F}} = \emptyset$, $\mathbf{A}^{\mathbf{T}} \cup \mathbf{A}^{\mathbf{F}} = dom(\mathbf{A})$, and $\delta \not\subseteq \mathbf{A}$ for all $\delta \in \Delta$. Given a weight constraint program Π, we adopt the convention that $dom(\mathbf{A}) = A(\Pi) \cup \{H(r), B(r) \mid r \in \Pi\}$.

For a weight constraint $\mathcal{W}$, the following pair of sets of nogoods stipulates correspondence between the truth value of $\mathcal{W}$ and the sum of true literals' weights:

$$\begin{aligned}\omega(\mathcal{W}) = \big\{\{\mathbf{F}\mathcal{W}\} \cup \{\mathbf{T}A(p) \mid p \in P\} \cup \{\mathbf{F}A(n) \mid n \in N\} \mid \quad & \quad (2)\\ P \subseteq \mathcal{W}^+, N \subseteq \mathcal{W}^-, \Sigma[P \cup N] \geq lb(\mathcal{W})\big\} &\\ \varpi(\mathcal{W}) = \big\{\{\mathbf{T}\mathcal{W}\} \cup \{\mathbf{F}A(p) \mid p \in P\} \cup \{\mathbf{T}A(n) \mid n \in N\} \mid \quad & \quad (3)\\ P \subseteq \mathcal{W}^+, N \subseteq \mathcal{W}^-, \Sigma[(\mathcal{W}^+ \setminus P) \cup (\mathcal{W}^- \setminus N)] < lb(\mathcal{W})\big\}\ . &\end{aligned}$$

Observe that the nogoods in $\omega(\mathcal{W})$ and $\varpi(\mathcal{W})$, respectively, capture the weakest conditions under which $\mathcal{W}$ evaluates to true or false, respectively. In general, the number of such weakest conditions is exponential in the number of literals in $\mathcal{W}$. Hence, it is impractical to explicitly construct $\omega(\mathcal{W})$ and $\varpi(\mathcal{W})$, and we below develop implementation techniques for unit propagation that work on $\mathcal{W}$ directly.

The correspondence between the truth of a weight constraint and its elements can be formalized as follows.

Proposition 1. *Let $\mathcal{W}$ be a weight constraint and $\mathbf{A}$ be an assignment such that $\mathbf{A}^{\mathbf{T}} \cap \mathbf{A}^{\mathbf{F}} = \emptyset$ and $\mathbf{A}^{\mathbf{T}} \cup \mathbf{A}^{\mathbf{F}} = A(\mathcal{W})$. Then, the following statements hold:*

1. *$\delta \setminus \mathbf{A} = \{\mathbf{F}\mathcal{W}\}$ for some $\delta \in \omega(\mathcal{W})$ iff $\mathbf{A}^{\mathbf{T}} \models \mathcal{W}$;*
2. *$\delta \setminus \mathbf{A} = \{\mathbf{T}\mathcal{W}\}$ for some $\delta \in \varpi(\mathcal{W})$ iff $\mathbf{A}^{\mathbf{T}} \not\models \mathcal{W}$.*

Proposition 1 shows that the weight constraints in a weight constraint program are fully determined by their literals when collecting the nogoods for all heads and bodies:

$$\Omega(\Pi) = \textstyle\bigcup_{r\in\Pi}\big(\omega(H(r)) \cup \varpi(H(r)) \cup \omega(B(r)) \cup \varpi(B(r))\big)\ .$$

As an answer set X of a program Π is a minimal model of Π^X, we have that a corresponding total assignment $\mathbf{A}$, viz., $\mathbf{A}^{\mathbf{T}} \cap A(\Pi) = X$, must be a model of Π, and each atom in X needs to be supported by a rule r such that $B(r) \in \mathbf{A}^{\mathbf{T}}$. When combined with $\Omega(\Pi)$, the following set of nogoods formalizes these two requirements:

$$\begin{aligned}\Delta(\Pi) = \big\{&\{\mathbf{F}H(r), \mathbf{T}B(r)\} \mid r \in \Pi\big\} \cup\\ &\big\{\{\mathbf{T}a, \mathbf{F}B(r) \mid r \in \Pi, A(H(r)) = \{a\}\} \mid a \in A(\Pi)\big\}\ .\end{aligned}$$

Proposition 2. *Let Π be a weight constraint program and $X \subseteq A(\Pi)$. Then, $X \models \Pi$ such that, for every $a \in X$, there is some $r \in \Pi$ with $A(H(r)) = \{a\}$ and $X \models B(r)$ iff there is a (unique) solution $\mathbf{A}$ for $\Omega(\Pi) \cup \Delta(\Pi)$ such that $\mathbf{A^T} \cap A(\Pi) = X$.*

The nogoods associated with weight constraint programs allow us to identify propagation operations along with their reasons. We say that a nogood δ is *unit-resulting* wrt an assignment $\mathbf{A}$ if $\delta \setminus \mathbf{A} = \{\sigma\}$ and $\overline{\sigma} \notin \mathbf{A}$. In such a situation, $\overline{\sigma}$ is mandatory to avoid the inclusion of δ in $\mathbf{A}$; in other words, δ *implies* $\overline{\sigma}$ wrt $\mathbf{A}$. The process of iteratively adding implied literals to $\mathbf{A}$ until violating some nogood or reaching a fixpoint (without any further implied literals) is called *unit propagation*. The implementation within *clasp* of unit propagation on nogoods in $\Omega(\Pi)$ is detailed in Section 4. Note that $\Omega(\Pi)$ merely provides a logical specification, while *clasp* works on weight constraints directly and determines nogoods in $\Omega(\Pi)$ only if needed as reasons.

In order to also capture minimality of an answer set X as a model of Π^X, for a program Π and a (partial) assignment $\mathbf{A}$, we define a set $U \subseteq A(\Pi)$ as *unfounded* for Π wrt $\mathbf{A}$ if, for every rule $r \in \Pi$, some of the following conditions holds:

1. $A(H(r)) \cap U = \emptyset$,
2. $B(r) \in \mathbf{A^F}$, or
3. $\Sigma[\{p \in B(r)^+ \mid A(p) \notin \mathbf{A^F} \cup U\} \cup \{n \in B(r)^- \mid A(n) \notin \mathbf{A^T}\}] < lb(B(r))$.

If U is unfounded for Π wrt $\mathbf{A}$, it means that none of its atoms belongs to any answer set given by a total extension of $\mathbf{A}$. In fact, the first condition expresses that r cannot support U, while the second condition checks that r is not applicable under $\mathbf{A}$. Finally, the third condition detects cases where $lb(B(r))$ cannot be reached via (weight) literals not false under $\mathbf{A}$, thereby, disregarding positive literals that depend on U.

To describe unfounded set conditions in terms of nogoods, for a set U of atoms, we define the *external sets* of literals for U in a weight rule r, $ext_r(U)$, as:

$$\{A(P) \cup {\sim}A(N) \mid P \subseteq B(r)^+, N \subseteq B(r)^-, A(P) \cap U = \emptyset, \Sigma[P \cup N] \geq lb(B(r))\} .$$

Note that elements L of $ext_r(U)$ are exactly the sets of literals such that the third unfounded set condition does not apply to r as long as $(L^+ \cap \mathbf{A^F}) \cup (L^- \cap \mathbf{A^T}) = \emptyset$, that is, if no literal in L is falsified by $\mathbf{A}$. Furthermore, for $U \subseteq A(\Pi)$, we call a set $C \subseteq \bigcup_{r \in \Pi, L \in ext_r(U)} L$ of literals a *cover set* for U in Π, if $C \cap L \neq \emptyset$ for every $r \in \Pi$ and $L \in ext_r(U)$. Note that, for any $r \in \Pi$, a cover set C for U in Π satisfies $\Sigma[\{p \in B(r)^+ \mid A(p) \notin C \cup U\} \cup \{n \in B(r)^- \mid {\sim}A(n) \notin C\}] < lb(B(r))$; otherwise, we would have $L = \{l \mid (l = w) \in B(r), l \notin C \cup U\} \in ext_r(U)$ and $C \cap L = \emptyset$, so that C would not be a cover set for U in Π. Letting $cov_\Pi(U)$ denote the set of all cover sets for U in Π, for some $u \in U$, the *loop nogoods*, $\lambda(u, U)$, are:

$$\bigcup_{\Lambda \subseteq \{r \in \Pi \mid A(H(r)) \cap U \neq \emptyset, ext_r(U) \neq \emptyset, A(B(r)^+) \cap U \neq \emptyset\}} \{\{\mathbf{F}a \mid a \in C^+\} \cup \{\mathbf{T}b \mid b \in C^-\} \cup \{\mathbf{T}u\} \cup \{\mathbf{F}B(r) \mid r \in \Pi \setminus \Lambda, A(H(r)) \cap U \neq \emptyset, ext_r(U) \neq \emptyset\} \mid C \in cov_\Lambda(U)\} .$$

Note that, for all rules $r \in \Pi$ such that $A(H(r)) \cap U \neq \emptyset$ and $ext_r(U) \neq \emptyset$, nogoods in $\lambda(u, U)$ reflect the second ($B(r) \in \mathbf{A^F}$) and the third (via Λ) unfounded set condition. Given the correspondence of the truth value of $B(r)$ and those of its (weight) literals stipulated via $\Omega(\Pi)$, the third condition needs to be checked separately only if $A(B(r)^+) \cap U \neq \emptyset$, which explains the choice of Λ. As with $\omega(\mathcal{W})$ and $\varpi(\mathcal{W})$ in (2) and (3), the size of $\lambda(u, U)$ is exponential in the number of literals in rule bodies, and on the implementation side, selected loop nogoods are determined on demand (see below).

For illustration, consider a program containing the following weight rules:

$$0\,\{a{=}1\} \leftarrow 2\,\{c{=}1, e{=}1, {\sim}b{=}1\} \tag{4}$$
$$1\,\{a{=}1\} \leftarrow 3\,\{b{=}2, {\sim}c{=}1, {\sim}d{=}1\} \tag{5}$$
$$1\{b{=}1\} \leftarrow 4\,\{a{=}3, c{=}2, {\sim}d{=}1, {\sim}e{=}3\}\,. \tag{6}$$

Taking $U = \{a, b\}$, we observe that the body of the rule in (4) does not positively depend on U, while the external sets for U in (5) are empty. For the rule r in (6), we get $ext_r(U) = \big\{\{c, {\sim}e\}, \{{\sim}d, {\sim}e\}, \{c, {\sim}d, {\sim}e\}\big\}$ and $cov_{\{r\}}(U) = \big\{\{{\sim}e\}, \{c, {\sim}d\}, \{c, {\sim}e\}, \{{\sim}d, {\sim}e\}, \{c, {\sim}d, {\sim}e\}\big\}$. Observe that $\{{\sim}e\}$ and $\{c, {\sim}d\}$ are the minimal cover sets for U in $\{r\}$, while the other three are subsumed by $\{{\sim}e\}$. We thus obtain the following non-redundant loop nogoods in $\lambda(u, U)$, where $u = a$ or $u = b$:

$$\big\{\mathbf{T}u, \mathbf{F}\,2\,\{c{=}1, e{=}1, {\sim}b{=}1\}, \mathbf{F}\,4\,\{a{=}3, c{=}2, {\sim}d{=}1, {\sim}e{=}3\}\big\}$$
$$\big\{\mathbf{T}u, \mathbf{F}\,2\,\{c{=}1, e{=}1, {\sim}b{=}1\}, \mathbf{T}e\big\}$$
$$\big\{\mathbf{T}u, \mathbf{F}\,2\,\{c{=}1, e{=}1, {\sim}b{=}1\}, \mathbf{F}c, \mathbf{T}d\big\}\,.$$

For a weight constraint program Π, we can now simply collect all loop nogoods:

$$\Lambda(\Pi) = \textstyle\bigcup_{\emptyset \subset U \subseteq A(\Pi), u \in U} \lambda(u, U)\,.$$

These nogoods ultimately establish a one-to-one correspondence between answer sets and solutions.

Theorem 1. *Let Π be a weight constraint program and $X \subseteq A(\Pi)$. Then, X is an answer set of Π iff there is a (unique) solution $\mathbf{A}$ for $\Omega(\Pi) \cup \Delta(\Pi) \cup \Lambda(\Pi)$ such that $\mathbf{A}^{\mathbf{T}} \cap A(\Pi) = X$.*

The basic *clasp* algorithm, relying on conflict-driven learning [5,6], has been described in [8], and its global structure remains unaffected if the nogoods to work with are exchanged. However, the identification of unfounded sets, described in [10] for disjunctive offspring *claspD*, needs to be adapted to weight constraint programs. We thus provide the logics of a dedicated unfounded set checking algorithm in Algorithm 1; its implementation in *clasp* will be described in Section 4. Given a program Π and an assignment $\mathbf{A}$, we assume that there is a predefined set $Do \subseteq A(\Pi)$ of atoms to investigate. Furthermore, each atom $a \in \bigcup_{r \in \Pi} A(H(r))$ has a *source pointer* [2], denoted by $sp(a)$, to a weight constraint $B(r)$ such that $A(H(r)) = \{a\}$ for some $r \in \Pi$; a source pointer $sp(a)$ has an associated set $sp(a)^{\#}$ of atoms considered as not belonging to any unfounded set $U \subseteq A(\Pi) \setminus \mathbf{A}^{\mathbf{F}}$. Finally, for every $a \in A(\Pi)$, number $c(a)$ denotes a strongly connected component C of the positive atom dependency graph of Π, defined by $(A(\Pi), \{(a, b) \mid r \in \Pi, A(H(r)) = \{a\}, b \in A(B(r)^+)\})$; atoms a of trivial strongly connected components (without edges) are identified by $c(a) = 0$. As pointed out in [2], unfounded set checking can be localized to non-trivial strongly

Algorithm 1. UNFOUNDEDSET

```
Input  : A weight constraint program $\Pi$ and an assignment $\mathbf{A}$.
Output : An unfounded set for $\Pi$ wrt $\mathbf{A}$.
1  $Do \leftarrow Do \setminus \mathbf{A^F}$
2  $Add \leftarrow \{a \in A(\Pi) \setminus (\mathbf{A^F} \cup Do) \mid c(a) \neq 0, sp(a) \in \mathbf{A^F}\}$
3  repeat
4      $Do \leftarrow Do \cup Add$
5      foreach $a \in A(\Pi)$ such that $sp(a)^{\#} \cap Add \neq \emptyset$ do $sp(a)^{\#} \leftarrow sp(a)^{\#} \setminus Add$
6      $Add \leftarrow \{a \in A(\Pi) \setminus (\mathbf{A^F} \cup Do) \mid c(a) \neq 0, \Sigma[\{n \in sp(a)^- \mid A(n) \notin \mathbf{A^T}\} \cup$
          $\{p \in sp(a)^+ \mid A(p) \notin \mathbf{A^F}, c(A(p)) \neq c(a)$ or $A(p) \in sp(a)^{\#}\}] < lb(sp(a))\}$
7  until $Add = \emptyset$
8  while $Do \neq \emptyset$ do let $a \in Do$ in
9      $U \leftarrow \{a\}$
10     repeat
11         $B \leftarrow \{B(r) \mid r \in \Pi, A(H(r)) \cap U \neq \emptyset, \Sigma[\{n \in B(r)^- \mid A(n) \notin \mathbf{A^T}\} \cup$
                      $\{p \in B(r)^+ \mid A(p) \notin \mathbf{A^F} \cup U\}] \geq lb(B(r)), B(r) \notin \mathbf{A^F}\}$
12         if $B = \emptyset$ then return $U$
13         else let $\mathcal{W} \in B$ in
14             $S \leftarrow \{s \in A(\mathcal{W}^+) \cap Do \mid c(s) = c(a)\}$
15             if $\Sigma[\{n \in \mathcal{W}^- \mid A(n) \notin \mathbf{A^T}\} \cup \{p \in \mathcal{W}^+ \mid A(p) \notin \mathbf{A^F} \cup S\}] \geq lb(\mathcal{W})$
               then
16                 if $\{s \in A(\mathcal{W}^+) \mid c(s) = c(a)\} \neq \emptyset$ then
17                     $\mathcal{W}^{\#} \leftarrow \{s \in A(\mathcal{W}^+) \mid c(s) = c(a), s \notin \mathbf{A^F} \cup S\}$
18                 foreach $u \in U$ such that $\{r \in \Pi \mid A(H(r)) = \{u\}, B(r) = \mathcal{W}\} \neq \emptyset$ do
19                     $sp(u) \leftarrow \mathcal{W}$
20                     $U \leftarrow U \setminus \{u\}$
21                     $Do \leftarrow Do \setminus \{u\}$
22             else $U \leftarrow U \cup S$
23     until $U = \emptyset$
24 return $\emptyset$
```

connected components (SCCs) without sacrificing soundness. In turn, we require as an invariant that $(A(\Pi), \{(a,b) \mid a \in A(\Pi), c(a) \neq 0, b \in sp(a)^{\#}\})$ is an acyclic graph (viz., all of its SCCs must be trivial). We then skip unfounded set checks for a as long as $sp(a) \notin \mathbf{A^F}$ and $\Sigma[\{n \in sp(a)^- \mid A(n) \notin \mathbf{A^T}\} \cup \{p \in sp(a)^+ \mid A(p) \notin \mathbf{A^F}, c(A(p)) \neq c(a)$ or $A(p) \in sp(a)^{\#}\}] \geq lb(sp(a))$, which means that some acyclic justification exists for a so that it cannot be unfounded.

The main *clasp* algorithm [8] triggers propagation, which includes unfounded set checks, after every heuristic decision. We assume that Do is empty when a heuristic decision is made, and repeated calls to UNFOUNDEDSET may successively fill it with atoms to check for unfoundedness. As a matter of fact, atoms falsified by unit propagation can be excluded and are thus eliminated in Line 1 of Algorithm 1. Non-false atoms whose source pointers have been falsified are scheduled for an unfounded set check in

Line 2 and 4; note that such atoms a must belong to non-trivial SCCs of the positive atom dependency graph of Π ($c(a) \neq 0$). The purpose of Line 6 is to iteratively identify atoms a such that $sp(a) \notin \mathbf{A}^{\mathbf{F}}$, while the existence of an acyclic justification is still not guaranteed. Iteration is needed because, in Line 5, possible occurrences of atoms that got into the scope of unfounded checks are removed from $sp(a)^{\#}$, so that a previously known acyclic justification for a may be put into question.

Having collected all non-false atoms that possibly are unfounded, the loop in Line 8–23 tries to re-establish acyclic justifications for the atoms in Do, starting from one atom a at a time and filling a potential unfounded set U. In Line 11, we determine all non-false weight constraints whose lower bounds can be reached via non-false literals outside U and that thus may be usable to justify an atom in U. Conversely, if no such weight constraint exists, we have identified a nonempty unfounded set U and return it in Line 12. The propagation routine [8] of *clasp* will then take care of falsifying all atoms in U before the next call to UNFOUNDEDSET. If U is not (yet) unfounded, in Line 13, we pick an arbitrary weight constraint $\mathcal{W}$ whose bound can be reached without using U; and in Line 14, we determine all possibly unfounded atoms in $\mathcal{W}$ from the same (non-trivial) SCC as the initial atom a. The fact that only such atoms may be used to extend U in Line 22 exhibits the localization of unfounded set checks to SCCs. However, we only extend U if the addition makes the sum of non-false literals' weights from outside U drop below the lower bound of $\mathcal{W}$, as checked in Line 15. If the latter is not the case, we are sure that some atoms in U have an acyclic justification via $\mathcal{W}$, and such atoms cannot belong to an unfounded set. Furthermore, the atoms in $A(\mathcal{W}^{+})$ from the SCC of a that are already acyclicly justified can be memorized in $\mathcal{W}^{\#}$ (Line 16–17). As long as the justifications or the source pointers, respectively, of these atoms do not change, this helps to avoid further (unsuccessful) unfounded set checks (cf. the condition in Line 6) for the atoms in U justified via $\mathcal{W}$ in Line 19. Finally, the acyclicly justified atoms are removed from the unfounded set U to be computed as well as from the scope Do of unfounded set checks (Line 20–21).

Notably, source pointers enable lazy unfounded set checking, performed only in reaction to changes in assignment $\mathbf{A}$. Beyond that, a second major benefit is backtrack-freeness. In fact, source pointers are still valid after backtracking, even though they might be set differently than in the state when $\mathbf{A}$ has previously been extended. However, only the existence of some acyclic justification for every non-false atom is important, while it is unnecessary to pick or reconstruct a specific one. The implementation of source pointers in *clasp*, described in the next section, follows this principle and does not reset source pointers upon backtracking (or backjumping, respectively).

4 Implementation of Weight Constraints in *clasp*

This section is dedicated to the implementation of weight constraints within the conflict-driven ASP solver *clasp*. Note that in *clasp* normal rules are *not* handled as described above. Instead, unit propagation on normal rules is applied by means of the more efficient, backtrack-free *Two-Watched-Literals* algorithm [11]. However, the dedicated treatment of weight constraints enables *clasp* to handle them natively, without relying on any transformation (cf. the comparison in Section 5).

Unit Propagation. The number of nogoods for a weight constraint $\mathcal{W}$ is in general exponential in the size of $A(\mathcal{W})$. Hence, it is impractical to explicitly construct $\omega(\mathcal{W})$ and $\varpi(\mathcal{W})$. Rather, our idea is to take advantage of Proposition 1 and to capture $\omega(\mathcal{W})$ and $\varpi(\mathcal{W})$ by two corresponding linear Pseudo-Boolean (PB) constraints (cf. [12]) that must be satisfied by any solution. In terms of weight constraint notation, we have

$$\begin{array}{ll} (PB_\omega) & w' \ \{\mathcal{W} = w', \sim A(p) = W(p), \ A(n) = W(n) \mid p \in \mathcal{W}^+, n \in \mathcal{W}^-\} \\ (PB_\varpi) & lb(\mathcal{W}) \ \{\sim\mathcal{W} = lb(\mathcal{W}), \ \ell \mid \ell \in \mathcal{W}^+ \cup \mathcal{W}^-\} \end{array}$$

where $w' = (\Sigma[\mathcal{W}^+ \cup \mathcal{W}^-] - lb(\mathcal{W})) + 1$. The first PB constraint (PB_ω) is obtained from Proposition 1.1; it is satisfied by an assignment iff all nogoods in $\omega(\mathcal{W})$ are satisfied. The same holds for (PB_ϖ), obtained from Proposition 1.2, and nogoods in $\varpi(\mathcal{W})$. Note that $\mathcal{W}$ can be assigned to false, while (PB_ω) and (PB_ϖ) must always be satisfied.

For a PB constraint $\mathcal{W}$ (a true weight constraint) and an assignment $\mathbf{A}$, let $\mathcal{T}, \mathcal{U}, \mathcal{F}$ denote the literals of $\mathcal{W}^+ \cup \mathcal{W}^-$ being true, unassigned, and false in $\mathbf{A}$. Then, $\mathcal{W}$ is unit when $\Sigma[\mathcal{W}^+ \cup \mathcal{W}^-] - \Sigma[\mathcal{F}] < lb(\mathcal{W}) + W(\ell)$ for $W(\ell) = \max\{W(\ell') \mid \ell' \in \mathcal{U}\}$ and $\ell \in \mathcal{U}$. In this case, $\mathcal{W}$ implies ℓ, and the implying assignment is $\mathcal{F}$. Unit propagation for PB constraints can be implemented using the following procedure:

1. Initialize a counter $S_\mathcal{W}$ to $\Sigma[\mathcal{W}^+ \cup \mathcal{W}^-]$.
2. Whenever a literal ℓ in $\mathcal{W}^+ \cup \mathcal{W}^-$ becomes false, set $S_\mathcal{W}$ to $S_\mathcal{W} - W(\ell)$.
3. If $S_\mathcal{W} < lb(\mathcal{W}) + \max\{W(\ell') \mid \ell' \in \mathcal{U}\}$, set each literal $\ell \in \mathcal{U}$ to true whose weight $W(\ell)$ satisfies the condition $W(\ell) > S_\mathcal{W} - lb(\mathcal{W})$.

The *clasp* implementation allows for arbitrary *Boolean constraints* through an abstraction similar to the one in [13]. Each concrete constraint type must implement functions for propagation and calculation of reasons. Also, functions for simplifying the constraint and for updating the constraint on backtracking can be specified but are not mandatory. Another important abstraction used in *clasp* is that of a *watch list*. For each literal l, a list is maintained storing constraints that need to be updated when l becomes true. Each individual entry in a watch list stores (a reference to) a constraint and an integer. A constraint can use the integer, passed as an argument to its propagate function, to associate data with the watched literal, e.g., the literal's position in the constraint.

Based on these abstractions, we implemented a constraint type `WC`, combining unit propagation on (PB_ω) and (PB_ϖ) for a weight constraint $\mathcal{W}$. Observe that one of the two PB constraints is obsolete once $\mathcal{W}$ is assigned. Also, the literals of (PB_ω) and (PB_ϖ) differ only in their signs. Their weights are identical, as one can simply set the weight of $\mathcal{W}$ to $\max\{w', lb(\mathcal{W})\}$ in both constraints without affecting satisfiability.

In the following algorithms, we use symbols **true** and **false** to refer to assigned truth values. In addition to primitive types like `int`, we use the following abstract data types:

`Lit` The type of (signed Boolean) literals. A literal instance has three fields: a variable index, a sign flag, and a watched flag. The variable index stores the underlying variable of a literal. The sign flag indicates whether the variable is negated. The `operator ¬()` returns the complement of a literal l and, given an integer i, the expression $l{*}i$ returns $\neg l$ if $i < 0$ and l otherwise.

`Vec<T>` A dynamic array of type `T`. Given a `Vec<T> v` of size n, the element at position $1 \leq$ `i` $\leq n$ is accessed via `v[i]`.

Algorithm 2. `WC::propagate(Lit p, int wd, Solver s)`

```
Input : A watched literal that became true, the data
        associated with the watch, and a solver object.
1  int ac = sign(wd)                           /* get affected constraint and */
2  Lit W = lits[1]*ac                          /* associated constraint literal */
3  if s.isTrue(W) || active + ac == 0 then
4      return NO CONFLICT                      /* constraint is satisfied or other is active */
5  int idx = abs(wd)                           /* index of $\neg p$ */
6  C(ac) = C(ac) - weight(idx)
7  lits[idx].watched = false                   /* mark as processed */
8  trail.push(wd)                              /* remember for backtracking */
9  while umax $\leq$ lits.size() && weight(umax) > C(ac) do
10     if lits[umax].watched then
11         active = ac                         /* mark constraint as active */
12         trail.push(umax*ac)
13         Lit x = lits[umax]*ac
14         if not s.force(x,this) then
15             return CONFLICT
16     ++umax
17 return NO CONFLICT
```

The type `WC` has the following fields:

`lits` Stores the literals of (PB_{ϖ}) ordered by decreasing weight. The weight of $\neg\mathcal{W}$ is set to $\max\{w', lb(\mathcal{W})\}$, and hence `lits[1]` stores $\neg\mathcal{W}$.[1] The literals of (PB_{ω}) are accessed by multiplying the literals in `lits` with `-1`.
`active` An integer denoting whether both (PB_{ω}) and (PB_{ϖ}) are relevant (`0`), only (PB_{ω}) is relevant (`-1`), or only (PB_{ϖ}) is relevant (`1`) under the current assignment.
$\mathtt{C}_{\omega}$ A counter initialized to $\Sigma[\mathtt{lits}] - lb(PB_{\omega})$.
$\mathtt{C}_{\varpi}$ A counter initialized to $\Sigma[\mathtt{lits}] - lb(PB_{\varpi})$.
`umax` The index of the literal with the greatest weight not yet (known to be) assigned.
`trail` A queue of assigned literals used for backtracking and computing reasons.

Initially, `active` is `0`, `umax` is `1`, and the `trail` is empty. Also, we add watches $(\neg l_i, i)$ and $(l_i, -i)$ for all literals l_i in `lits` and set the watched flags of the literal instances to true. For example, consider $\mathcal{W} = 4\,\{a\!=\!3, c\!=\!2, {\sim}d\!=\!1, {\sim}e\!=\!3\}$. In this case, `lits` is $[\neg\mathcal{W}\!=\!6, a\!=\!3, \neg e\!=\!3, c\!=\!2, \neg d\!=\!1]$. Moreover, $\mathtt{C}_{\varpi}$ is 11, $\mathtt{C}_{\omega}$ is 9, and we add watches $(\mathcal{W}, 1), (\neg a, 2), \ldots, (d, 5)$ and $(\neg\mathcal{W}, -1), (a, -2), \ldots, (\neg d, -5)$.

Algorithm 2 shows the procedure for propagating a weight constraint, triggered when one of the watched literals becomes **true**. Staying with the example, assume that a is set to **true**. Then, Algorithm 2 is called with `p` $= a$ and `wd` $= -2$. From the sign of `wd`, we determine the affected PB constraint, i.e., (PB_{ϖ}) if `wd` > 0 and (PB_{ω}) if `wd` < 0. Since $\mathcal{W}$ is not yet assigned and `active` is `0`, (PB_{ω}) is relevant under

[1] We assume that for all literals ℓ in a weight constraint $\mathcal{W}$, we have $W(\ell) \leq lb(\mathcal{W})$. Weights greater than the lower bound are replaced with the bound in a preprocessing step.

Algorithm 3. WC::reason(Lit p, Vec<Lit> out)

```
Input : A literal propagated by this constraint.
Output: A set of true literals implying p.
1 foreach int d ∈ trail do
2     if sign(d) == active then
3         int idx = abs(d)
4         Lit x = lits[idx]*active
5         if not lits[idx].watched then out.push(¬x)
6         else if x == p then break
```

the current assignment, and so we decrease $\mathtt{C}_\omega$ by 3 (the weight of a) in Line 6. We then set the watched flag of the literal instance to false to indicate that the respective counter was updated. Also, we push wd to the `trail` so that we can suitably increase $\mathtt{C}_\omega$ again on backtracking and to compute reasons for assignments. Finally, given that `lits[umax]`$=\neg\mathcal{W}$ and `weight(umax)`$=6=\mathtt{C}_\omega$, the while loop in Line 9–16 is skipped. Next, assume that c becomes **false**. Hence, $\mathtt{wd} = 4$, and the affected constraint is (PB_ϖ). Since (PB_ϖ) is also not yet unit, no new assignments are derived. Finally, assume that d is assigned to **false**. From $\mathtt{wd} = -5$, we again extract (PB_ω) as the affected PB constraint, and after decreasing $\mathtt{C}_\omega$ to 5, we have `weight(umax)`$>\mathtt{C}_\omega$. That is, the constraint is now unit so that the while loop is entered in Line 9. The loop considers only literal instances whose watched flags are true, while other literals were already processed. Since `lits[1]` has its watched flag set, the condition in Line 10 is satisfied, (PB_ω) is marked as active, and $\mathcal{W}$ is forced to **true**. Note that `lits[1]` is $\neg\mathcal{W}$, but after multiplying with `-1` (the active constraint), we correctly get $\mathcal{W}$. As mentioned before, `lits` is ordered by decreasing weight. Thus, after `umax` is increased, it points to a (the literal with the next greatest weight to consider), and as $3 \leq 5 = \mathtt{C}_\omega$, propagation stops. When Algorithm 2 is then called for $\mathcal{W}$ and $\mathtt{wd} = 1$, the condition in Line 3 is true (`active` is `-1` and `ac` is `1`), i.e., nothing needs to be done.

Conflict-Driven Nogood Learning. The CDNL algorithm [8] of *clasp* applies the common First-UIP scheme [5,6] for resolving conflicts. The procedure starts with a violated nogood δ and resolves literals out of δ until only one literal assigned at the current decision level remains. For this to work, each concrete constraint type must implement a procedure, which, given a literal `p` implied by a constraint of that type, returns a set of (**true**) literals implying `p`. Algorithm 3 shows this procedure for weight constraints.

The idea is to dynamically extract a nogood from either $\omega(\mathcal{W})$ or $\varpi(\mathcal{W})$, depending on the currently active PB constraint. Reconsider the previous example and assume that Algorithm 3 is called with $\mathtt{p} = \mathcal{W}$. The constraint's `trail` is $[-2, 4, -5, -1]$, and the active constraint (`-1`) is (PB_ω). Then, element 4 is skipped because it was not added by the active PB constraint (cf. Line 2). For the other elements, we check whether the corresponding literal instances still have their watched flags set. If not, the element corresponds to a literal that is **false** in the active constraint and thus belongs to the implying assignment. Otherwise, the literal is **true** and was forced by the active constraint. We also push such implied literals to the `trail` (cf. Line 12 in Algorithm 2) because a weight constraint can become unit multiple times, and in that case only the **false** literals assigned earlier are part of the implying assignment. For the example, we add a and $\neg d$ to `out`, but not $\mathcal{W}$, because the watched flag of `lits[1]` is still set. Furthermore, since $\texttt{p} = \mathcal{W}$, the condition in Line 6 is true, and the extracted nogood is $\{\mathbf{F}\mathcal{W}, \mathbf{T}a, \mathbf{F}d\} \in \omega(\mathcal{W})$. Accordingly, the fact that a is **true** and d is **false** provides a reason for $\mathcal{W}$ being **true**.

When a conflict is resolved and one or more decision levels are removed, constraint types implementing an undo function are notified. The corresponding procedure for weight constraints pops entries corresponding to unassigned literals from the `trail`, again using the sign of a stored integer to determine the affected PB constraint and the watched flag to distinguish a processed from an implied literal. Counters are only increased for the former, and the watched flag is then set back to true to indicate that the corresponding literal contributes again to the respective counter value. If the literal with the greatest weight, viz., $\neg\mathcal{W}$, is unassigned, the constraint can no longer be unit, and hence `active` is set back to `0`. Otherwise, `active` is left unchanged, meaning that the previously active PB constraint stays in effect. Finally, `umax` is set back to the index of the unassigned literal with the greatest weight.

Unfounded Set Checking. A second set of data structures is used for representing the atoms and rule bodies that need to be considered during unfounded set checking (and extraction of loop nogoods). This is motivated by the fact that only the non-trivial SCCs of a program's positive atom dependency graph are relevant during unfounded set checks. For a program Π, *clasp*'s unfounded set checker stores the set $\{a \in A(\Pi) \mid c(a) \neq 0\}$ as `Atom` instances. For an atom a in that set, the corresponding `Atom` instance contains:

`scc` the atom's component number $c(a)$,
$\texttt{p}_s$ its set of possible sources $\{B(r) \mid r \in \Pi, A(H(r)) = \{a\}\}$,
`pos` the set of rule bodies $\{B(r) \mid r \in \Pi, a \in B(r)^+, A(H(r)) = \{a'\}, c(a') = c(a)\}$,
`source` (a pointer to) its current source $sp(a) \in \texttt{p}_s$, and
`vs` a flag indicating whether `source` is currently valid. Initially, `vs` is set to false and a is added to *Do* (cf. Algorithm 1).

The set of (distinct) weight constraints $\{B(r) \mid r \in \Pi, A(H(r)) = \{a\}, c(a) \neq 0\}$ is represented by instances of type `Body`. For a weight constraint $\mathcal{W}$ in that set, the corresponding `Body` instance stores:

`scc` the body's component number, $c(\mathcal{W})$, that is set to $c(a)$ if there is some $r \in \Pi$ such that $A(H(r)) = \{a\}$, $B(r) = \mathcal{W}$, and $\{b \in A(\mathcal{W}^+) \mid c(b) = c(a) \neq 0\} \neq \emptyset$, or to 0 otherwise.[2]
`extern` its "external" literals $\{p \in \mathcal{W}^+, n \in \mathcal{W}^- \mid c(A(p)) = 0 \text{ or } c(A(p)) \neq c(\mathcal{W})\}$,
`intern` its "internal" literals $\{p \in \mathcal{W}^+ \mid c(A(p)) = c(\mathcal{W}) \neq 0\}$,
`heads` its "heads" $\{a \mid r \in \Pi, A(H(r)) = \{a\}, B(r) = \mathcal{W}, c(a) \neq 0\}$, and
`C` a counter initialized to $lb(\mathcal{W}) - \Sigma[\texttt{extern}]$.

[2] Note that $c(\mathcal{W})$ is unique. If $r_1, r_2 \in \Pi$ with $B(r_1) = B(r_2) = \mathcal{W}$, $A(H(r_1)) = \{a_1\}$, $A(H(r_2)) = \{a_2\}$, $c(a_1) = c(b_1) \neq 0$, $c(a_2) = c(b_2) \neq 0$ for $b_1, b_2 \in \mathcal{W}^+$, then $c(a_1) = c(a_2)$.

Algorithm 4. findSource(Atom a)

```
1  Set<Atom> T = {a}, U = $\emptyset$
2  while T\U != $\emptyset$ do let Atom a $\in$ T\U in
3      U = U $\cup$ {a}
4      foreach Body B $\in$ a.p$_s$ do
5          if B $\in$ $\mathbf{A^F}$ then continue
6          else if B.scc != a.scc || B.C $\leq$ 0 || B.update() then
7              a.source = B
8              Set<Atom> S = {a}
9              while S != $\emptyset$ do let Atom a $\in$ S in
10                 S = S\{a}, T = T\{a}, U = U\{a}, $Do$ = $Do$\{a}
11                 foreach Body B $\in$ a.pos do B.atomSourced(a,S)
12             break
13         else B.addUnsourced(T)
14 return U
```

Again, we use the watched flags of the literal instances in B.extern and B.intern, for an instance B of Body, to distinguish the literals that currently contribute to the value of B.C from the rest. That is, initially all literals in B.extern have their watched flags set to true, while those in B.intern have them set to false. Logically, the literals in B.intern whose watched flags are true correspond to the atoms in $B^{\#}$. Furthermore, B is a valid source for an atom a in B.heads if B is not **false** and $B.C \leq 0$ or $B.scc \neq a.scc$. In order to efficiently detect when one of the first two conditions is violated, we use watches for B as well as literals in B.extern and B.intern. During unit propagation, if a literal l in B.extern or B.intern whose watched flag is set becomes **false**, B.C is increased by weight(l), and l's watched flag is set to false. In addition, invalidated sources are accumulated. Note that B.C is not updated during backtracking, but only during unfounded set propagation (see below).

Once unfounded set propagation begins, invalidated sources are used to initialize *Add* (cf. Algorithm 1). That is, we add all (non-**false**) atoms to *Add* whose sources are invalid. If a.vs is true for an atom a included in *Add*, we set it to false and propagate the removal of the source pointer by notifying all bodies in a.pos. Each affected body B then checks whether a currently belongs to $B^{\#}$, i.e., whether a in B.intern has its watched flag set. In this case, the watched flag is set to false, and B.C is increased accordingly. Since this may invalidate B, the whole process is repeated until no more atoms are added to *Add* (and *Do*).

Following the idea of Algorithm 1, we then try to re-establish acyclic justifications for the atoms in *Do*, where Line 9–23 of Algorithm 1 are implemented as in Algorithm 4. The abstract data type Set<Atom> refers to the mathematical concept of a set (of Atom instances) along with operations on them. The atoms in T are considered in turn. That is, in each iteration of the loop starting in Line 2, one atom a is selected and added to a set U. Then, all non-**false** bodies B in a.p$_s$ are inspected. At this point, B is updated only if it is currently not a valid source for a. That is, if $B.scc = a.scc$ and $B.C > 0$, B.update() checks for literals in B.extern and B.intern that are neither **false** nor have their watched flags set. If such a literal is found in B.extern, its watch flag is set and B.C is decreased by the literal's weight. For such a literal in B.intern, the same is done only if the corresponding atom currently has a valid source pointer. If even after updating B is not a valid source, T is extended with non-**false** atoms in B.intern lacking a valid source (Line 13). This is similar to Line 22 of Algorithm 1. In particular, since at this point B.scc is always equal to a.scc, the same localization to SCCs is achieved. On the other hand, if B is a valid source, it is used as new source for a, and the new source pointer is propagated by notifying all bodies in a.pos. Each affected body B that is currently not a valid source checks whether adding a to $B^{\#}$ turns it into a valid source. If so, non-**false** atoms in B.heads currently lacking a source are added to S. Thus, source pointer propagation is iterated until S is finally empty. Furthermore, atoms for which a new source pointer has been set are removed from T, U, and *Do*. Note that both during updates and source pointer propagation, $B^{\#}$ is extended only if B is not (yet) a valid source. This way, it is guaranteed that atoms added to $B^{\#}$ are acyclicly justified independently of B. Finally, once $T \setminus U$ is empty, all potential sources were inspected. Any remaining atoms in U are unfounded wrt the current assignment and are returned in Line 14.

Note that, for bodies of normal rules and weight constraints $\mathcal{W}$ with $c(\mathcal{W}) = 0$, the set of "external" literals is not relevant during unfounded set checking, and only the truth values of such bodies or weight constraints, respectively, are considered. Also, for bodies of normal rules, no (additional) watches are needed for body literals because unit propagation already falsifies such a body whenever one of its literals becomes **false**.

5 Experiments

We implemented our approach within the ASP solver *clasp* (1.2.0). Our experiments consider *clasp* (in its default configuration) using three different ways of treating weight constraints: (a) standard setting, using the described approach; (b) with (quadratic) transformation of weight constraints (cf. [7]); (c) with selective transformation of weight constraints. Variant (c) applies strategy (b) to weight constraints with lower bound 1 and whenever the number of resulting nogoods is smaller than 16, otherwise it applies strategy (a). We also consider *smodels* (2.33 with option -restart[3]) and *cmodels* (3.78) because of their distinct treatment of weight constraints. The full experiments, additionally including *pbmodels*, *smodelscc*, as well as *smodels* without lookahead, are given at [14] (see also below). We conducted experiments on a variety of benchmarks taken from the *SLparse* category of the first ASP system competition.[4] Among them, *BlockedNQueens*, *BoundedSpanningTree*, and *SocialGolfer* comprise choice and cardinality rules, while *TravelingSalesperson*, *WeightedLatinSquare*, and *WeightedSpanningTree* contain also weight rules. In addition, we consider a handcrafted benchmark, *ExtHamPath*, possessing non-trivial unfounded sets due to recursive cardinality constraints. Each of the benchmark sets consists of five instances.[5]

[3] This variant of *smodels* performed best on our benchmarks.

[4] http://asparagus.cs.uni-potsdam.de/contest

[5] All benchmarks are available at [14].

Table 1. Benchmark results on a 3.4GHz PC under Linux, each run restricted to 600s and 1GB

Benchmark	*clasp* (a)	*clasp* (b)	*clasp* (c)	*cmodels*	*smodels*
BlockedNQueens	34.01(0)	70.11(0)	49.70(0)	570.20(0)	363.42(1)
BoundedSpanningTree	8.12(0)	9.94(0)	8.16(0)	19.53(0)	1381.98(4)
SocialGolfer	601.79(3)	606.13(3)	604.23(3)	1035.77(5)	1802.65(9)
TravelingSalesperson	2.17(0)	140.02(0)	1.40(0)	2195.75(6)	21.63(0)
WeightedLatinSquare	0.06(0)	0.59(0)	0.10(0)	1.19(0)	1700.90(2)
WeightedSpanningTree	5.08(0)	5.93(0)	5.67(0)	11.40(0)	953.97(2)
ExtHamPath	10.64(0)	84.72(0)	18.25(0)	28.67(0)	2223.28(6)
$\Sigma(\Sigma)$	661.87(3)	917.44(3)	687.51(3)	3862.51(11)	8649.47(24)

Table 1 summarizes our results by giving the sum of runtimes obtained on the five instances in each benchmark set; each instance is measured by taking the average over three shuffles obtained with ASP tools from TU Helsinki.[6] A timeout is accounted for by the maximum time of 600s, and timeouts are also indicated in parentheses. We mention that *pbmodels* with *minisat+* and *satzoo* yields 6925.55(32) and 9954.02(36) in total, respectively; *smodels* without lookahead takes 13141.25(61).

Looking at *BlockedNQueens* and *TravelingSalesperson*, we observe a drastic effect through a dedicated treatment of cardinality and weight constraints. While instances of the former contain many relatively large cardinality constraints, instances of the latter contain a single weight constraint with 600 literals. In *clasp* (b), this leads to an extension of programs by over 600000 normal rules and more than 300000 auxiliary atoms. As a consequence, both transformation-based approaches, *clasp* (b) and *cmodels*, are outperformed by orders of magnitude by *clasp* (a/c) and *smodels*. Unlike this, *BoundedSpanningTree* and *SocialGolfer* include only a few small to midsize cardinality rules and so produce almost no overhead on transformation-based approaches. The same applies to *WeightedLatinSquare* and *WeightedSpanningTree* as regards the transformation of weight constraints. In contrast to the benchmarks from the *SLparse* category, *ExtHamPath* contains many small yet recursive cardinality rules inducing a large positive dependency graph and many non-trivial unfounded sets. We attribute *smodels*' poor performance on this benchmark to exhaustive lookahead operations. Given that the small size of cardinality constraints puts the remaining approaches on equal footing, the customized unfounded set algorithm in *clasp* (a) shows a decent performance.

Our experiments demonstrate that the combination of conflict-driven learning with a dedicated treatment of weight constraints has an edge over either singular approach. Although the overhead of a dedicated treatment seems disadvantageous on small weight constraints, the hybrid approach of *clasp* (c) does not improve on the overall performance of the fully dedicated one, viz., *clasp* (a).

6 Discussion

We presented a comprehensive approach to integrating weight (and cardinality) rules into conflict-driven ASP solving, utilizing a nogood-based characterization of answer

[6] `http://www.tcs.hut.fi/Software/asptools`

sets to specify (unit) propagation over weight rules. To be precise, we established a one-to-one correspondence between the answer sets of a weight constraint program and the solutions for the nogoods induced by the program. In view of the exponential number of loop nogoods, we developed a dedicated, source-pointer-based unfounded set checking algorithm that computes loop nogoods only on demand, while aiming at lazy unfounded set checking and backtrack-freeness. Similarly, we are faced with an exponential number of nogoods stemming from weight constraints, although language-extending, quadratic representations exist. Unlike this, we advocate a dedicated treatment of weight constraints, akin to the one used in *smodels* yet extended to conflict-driven learning and backjumping. We developed our computational approach from the semantic foundations laid in Section 3. Our design is guided by two Pseudo-Boolean constraints that must be satisfied by any solution. In view of this, Section 4 provided a rather detailed account of the key features of the weight constraint implementation in *clasp*. Our experiments show that our dedicated approach to handling weight constraints is competitive and does not seem to produce significant overhead on benchmarks with only small constraints, putatively favoring transformation techniques.

Acknowledgments. This work was supported by DFG under grant SCHA 550/8-1.

References

1. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press, Cambridge (2003)
2. Simons, P., Niemelä, I., Soininen, T.: Extending and implementing the stable model semantics. Artificial Intelligence 138(1-2), 181–234 (2002)
3. Dowling, W., Gallier, J.: Linear-time algorithms for testing the satisfiability of propositional Horn formulae. Journal of Logic Programming 1, 267–284 (1984)
4. Giunchiglia, E., Lierler, Y., Maratea, M.: Answer set programming based on propositional satisfiability. Journal of Automated Reasoning 36(4), 345–377 (2006)
5. Marques-Silva, J., Sakallah, K.: GRASP: A search algorithm for propositional satisfiability. IEEE Transactions on Computers 48(5), 506–521 (1999)
6. Zhang, L., Madigan, C., Moskewicz, M., Malik, S.: Efficient conflict driven learning in a Boolean satisfiability solver. In: ICCAD 2001, pp. 279–285 (2001)
7. Ferraris, P., Lifschitz, V.: Weight constraints as nested expressions. Theory and Practice of Logic Programming 5(1-2), 45–74 (2005)
8. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Conflict-driven answer set solving. In: IJCAI 2007, pp. 386–392. AAAI Press, Menlo Park (2007)
9. Dechter, R.: Constraint Processing. Morgan Kaufmann, San Francisco (2003)
10. Drescher, C., Gebser, M., Grote, T., Kaufmann, B., König, A., Ostrowski, M., Schaub, T.: Conflict-driven disjunctive answer set solving. In: KR 2008, pp. 422–432. AAAI Press, Menlo Park (2008)
11. Moskewicz, M., Madigan, C., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: DAC 2001, pp. 530–535. ACM Press, New York (2001)
12. Sheini, H., Sakallah, K.: Pueblo: A hybrid Pseudo-Boolean SAT solver. Journal on Satisfiability, Boolean Modeling and Computation 2, 165–189 (2006)
13. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004)
14. http://www.cs.uni-potsdam.de/clasp

A Language for Large Ensembles of Independently Executing Nodes

Michael P. Ashley-Rollman[1], Peter Lee[1], Seth Copen Goldstein[1], Padmanabhan Pillai[2], and Jason D. Campbell[2]

[1] Carnegie Mellon University, Pittsburgh, PA 15213
{mpa,petel,seth}@cs.cmu.edu
[2] Intel Research Pittsburgh, Pittsburgh, PA 15213
{padmanabhan.s.pillai,jason.d.campbell}@intel.com

Abstract. We address how to write programs for distributed computing systems in which the network topology can change dynamically. Examples of such systems, which we call *ensembles*, include programmable sensor networks (where the network topology can change due to failures in the nodes or links) and modular robotics systems (whose physical configuration can be rearranged under program control). We extend Meld [1], a logic programming language that allows an ensemble to be viewed as a single computing system. In addition to proving some key properties of the language, we have also implemented a complete compiler for Meld. It generates code for TinyOS [14] and for a Claytronics simulator [12]. We have successfully written correct, efficient, and complex programs for ensembles containing over one million nodes.

1 Introduction

Several types of distributed systems have the property that the network topology can change dynamically. For example, in *ad hoc* sensor networks [6], it is common for the network to change due to failures in both the nodes and network links. Modular robotic systems [21] are another common example. Under software control, a modular robotic system can rearrange how its modules are connected, which means that its network topology changes, too. A third example is Claytronics [11], which can be viewed as a type of modular robotic system containing, potentially, millions of unreliable modules. We use the term *ensemble* to refer to any such network-varying distributed system.

How shall we write programs for ensembles? Writing software is hard; writing software for distributed systems is even harder. Add to that the possibility of a dynamic network topology, and it is easy to see that we are faced with a daunting programming problem. Furthermore, real-world ensembles such as robots and sensor nets pose additional challenges such as unreliable communications, imperfect or failing actuators, and soft and hard compute errors. The complexity of writing code to recover from such failures is magnified by the number of potential interactions within an ensemble.

Given these considerations, we have been extending the language Meld [1], that allows ensembles to be programmed as a unified whole and then compiled automatically into fully distributed code. This approach frees programmers from the need to understand how or where in the system the program state is to be maintained or messages sent.

Furthermore, Meld programs are written in the logic programming paradigm, leading to clear and concise programs. And as our early experiments have thus far demonstrated, Meld programs are inherently robust to changes in network topology and provide for simple fault handling. The initial design of Meld was influenced heavily by Datalog [4] and, like Datalog, programs in Meld lend themselves to proofs of correctness.

We have made substantial progress on the design, implementation, and application of Meld [1]. We have adopted X-Y stratification from LDL++ [22] and adapted it to work in a distributed environment. We have addressed problems with the deletion algorithm used in prior work, such as P2 [15]. We have achieved what we believe to be a fully practical language for a range of modular robotics and sensor network applications, and have completed a thorough definition of the operational semantics, much of which has been implemented in the Twelf [19] proof system. A complete compiler for Meld has also been implemented, which generates code for TinyOS [14] and a Claytronics simulator [12]. Using the compiler and simulator, we have written correct, efficient programs for ensembles containing over one million nodes.

2 Related Work

The P2 [15] language and SNLog [5] language, which were designed for programming overlay networks and sensor networks respectively, showed that a logic programming approach could be used to allow an ensemble to be programmed as a unified whole. Meld adds, among other things, support for robot actuation and sensing, and is based on a well-defined formal semantics. In principle, even very large ensembles can be reasoned about formally.

Several languages have been proposed for sensor nets. Hood [20], Tinydb [16], and Regiment [18] provide excellent support for applications such as data collection and aggregation, but do not directly address more dynamic scenarios involving physical reconfiguration. Pleiades [13], also designed for sensor networks, can be used in situations with dynamic network topologies. It adopts a programming style similar to OpenMP, for example allowing one to write parallel loops that run across multiple modules. Of course, this requires the programmer to specify whether a variable is to be stored locally or propagated about the ensemble as the program runs. Thus, the programmer's focus is on the individual modules instead of the whole ensemble.

Origami Shape Language [17] and Proto [3] are effective for programming distributed systems as a whole. They were originally targeted towards stationary wireless modules, rather than ensembles. Recently, Proto has been extended to mobile robots [2]. LDP [7] was derived from a method for distributed debugging. Originally designed for modular robotics, LDP sends condition-matchers around the ensemble. It is based on a tick model, generating a new set of matchers throughout the ensemble on each tick. While this works well in highly dynamic systems, it can lead to excessive messaging in more static environments.

3 Meld: Language and Meaning

A running Meld program consists of a database of *facts* and a set of production rules for operating on and generating new facts. A Meld fact is a predicate and a tuple of

Known Facts	$\Gamma ::= \cdot \mid \Gamma, f(\hat{t})$	Facts	$F ::= f(\hat{x})$
Accumulated Actions	$\Psi ::= \cdot \mid \Psi, a(\hat{t})$	Constraints	$C ::= c(\hat{x})$
Set of Rules	$\Sigma ::= \cdot \mid \Sigma, R$	Expression	$E ::= E \wedge E \mid F \mid \forall F.E \mid C$
Actions	$A ::= a(\hat{x})$	Rule	$R ::= E \Rightarrow F \mid E \Rightarrow A$
			$\mid agg(F, g, y) \Rightarrow F$

Fig. 1. Abstract syntax for Meld programs

values; the predicate denotes a particular relation for which the tuple is an element. Facts represent the state of the world based on observations and inputs (e.g., sensor readings, connectivity or topology information, runtime parameters, etc.), or they reflect the internal state of the program. Starting from an initial set of axioms, new facts are derived and added to the database as the program runs. In addition to facts, *actions* are also generated. They are syntactically similar to facts but cause side effects that change the state of the world rather than the derivations of new facts. In a robotics application, for example, actions are used to initiate motion or control devices. Meld rules can use a variety of arithmetic, logical, and set-based expressions, as well as aggregation operations.

3.1 Structure of a Meld Program

Fig. 1 shows the abstract syntax for states, rules, expressions, and constraints in Meld. A Meld program consists of a set of production rules. A rule may contain variables, the scope of which is the entire rule. Each rule has a head that specifies a fact to be generated and a body of prerequisites to be satisfied. If all prerequisites are satisfied, then the new fact is added to the database. Each prerequisite expression in the body of the rule can either match a fact or specify a constraint. Matching is achieved by finding a consistent substitution for the rule's variables such that one or more facts in the database are matched. A constraint is a boolean expression evaluated on facts in the database; these can use arithmetic, logical, and set-based subexpressions. Finally, `forall` statements iterate over all matching facts in the database and ensure that for each one, a specified expression is satisfied.

Meld rules may also derive actions, rather than facts. Action rules have the same syntax as rules, but have a different effect. When the body of this rule is proved true, its head is not inserted into the database. Instead, it causes an action to be carried out in the physical world. The action, much like a fact, has a predicate and a tuple, which can be assigned values by the expressions in the rule.

An important concept in Meld is the *aggregate*. The purpose of an aggregate is to define a type of predicate that combines the values in a collection of facts. As a simple example, consider the program shown in Fig. 2, for computing the maximum temperature across all the nodes in an ensemble. The `parent` rules, (c) and (d), build a spanning tree across the ensemble, and then the `maxTemp` rules, (e) and (f), use this tree to compute the maximum temperature. Considering first the rules for calculating the maximum, rule (e) sets the base case; rule (f) then propagates the maximum temperature (T) of the subtree rooted at one node (A) to its parent (B). Applied across the

```
(a) type logical neighbor parent(module, first module).
(b) type maxTemp(module, max float).

(c) parent(A, A) :- root(A).
(d) parent(A, B) :- neighbor(A, B), parent(B, _).
(e) maxTemp(A, T) :- temperature(A, T).
(f) maxTemp(B, T) :- parent(A, B), maxTemp(A, T).

(g) type globalMax(module, float).
(h) globalMax(A, T) :- maxTemp(A, T), root(A).
(i) globalMax(B, T) :- neighbor(A, B), globalMax(A, T).

(j) type localMax(module).
(k) localMax(A) :- temperature(A, T),
                   forall neighbor(A, B) temperature(B, T') T > T'.
```

Fig. 2. A data aggregation example coded in Meld. A spanning tree is built across the ensemble and used to aggregate the max temperatures of all nodes to the root. The result is flood broadcast back to all nodes. Each node also determines whether it is a local maximum.

ensemble, this has the effect of producing the maximum temperature at the root of the tree. To accomplish this, the rule prototype given in (b) specifies that when `maxTemp` is matched, the `max` function should be used to aggregate all values in the second field for those cases where the first field matches. In the case of the `parent` rules, the prototype given in (a) indicates the use of the `first` aggregate, limiting each node to a single parent. The `first` aggregate keeps only the first value encountered in any match on the rest of the tuple. The meaning of `logical neighbor` is explained in §4.1.

In general, aggregates may use arbitrary functions to calculate the aggregate value. In the abstract syntax, this is given as a function g that calculates the value of the aggregate given all of the individual values. The result of applying g is then substituted for y in F. In practice, as described by LDL++[22], the programmer implements this as three functions: two to create an aggregate and one to retrieve the final value. The first two functions consist of one to create an aggregate from a single value and a second to update the value of an existing aggregate given another value. The third function, which produces the final value of the aggregate, permits the aggregate to keep additional state necessary to compute the aggregate. For example, an aggregate to compute the average would keep around the sum of all values and the number of values seen. When the final value of the aggregate is requested, the current value of sum is divided by the total number of values seen to produce the requested average.

3.2 Meaning of a Meld Program

The state of an ensemble running a Meld program consists of two parts: derived facts and derived actions. Γ is the set of facts that have been derived in the current world. Γ is a list of facts that are known to be true. Initially, Γ is populated with observations that modules make about the world. Ψ, is the set of actions derived in the current world. These are much like the derived facts that make up Γ, except that they are intended to have an effect upon the ensemble rather than be used to derive further facts.

As a Meld program runs, new facts and actions are derived from existing facts which are then added to Γ and Ψ. Once one or more actions have been derived, they can be applied to bring about a change in the physical world. When the actions have been applied to the world, all derived facts are discarded and replaced with a set of new observations about the world. The program then restarts execution in the new world.

The evaluation rules for Meld allow for significant uncertainty with respect to actions and their effects. This underspecification has two purposes. First, it does not make assumptions about the type of ensemble nor the kinds of actions which can be triggered by a Meld program. Second, it admits the possibility of noisy sensors and faulty actuators. In the case of modular robotics, for instance, a derived action may request that a robot move to a particular location. External constraints, however, may prevent the robot from moving to the desired location. It is, therefore, important that Γ end up containing the actual position of the robot rather than the location it desired to reach.

This result is achieved by discarding Γ when an action is applied and starting fresh. By doing this, we erase all history from the system, removing any dependencies on the intended effect of the action. This interpretation also accounts for the fact that sensors may fail, be noisy, and even in the best case that observations of the real world that are known to the ensemble are only a subset of those that are available in the real world. To account for changes that arise due to external forces we also allow the program to restart even when Ψ is empty.

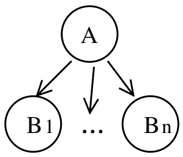
This interpretation permits us to give Meld programs a well-defined meaning even when actuators fail, external forces change the ensemble, or sensors are noisy. In turn, this imbues Meld with an inherent form of fault tolerance. The greatest limitation of this approach, however, is in our ability to reason about programs. By allowing the ensemble to enter a state other than the one intended by the action, we eliminate the ability to directly reason about what a program does. To circumvent this, it is necessary to make assumptions about how likely an action is to go awry and in what ways it's possible for it to go awry. This is discussed further in §5.2.

4 Distributed Implementation of Meld programs

In this section we describe how Meld programs can be run as forward-chaining logic programs across an ensemble. We first explain the basic ideas that make this possible. We then describe the additional techniques of deletion and X-Y stratification that are required to make this feasible and efficient.

4.1 Basic Distribution/Implementation Approach

Meld is an ensemble programming language; efficient and scalable execution requires Meld programs to be distributed across the nodes of the ensemble. To facilitate this, we require the first variable of a fact, called the *home variable*, to refer to the node where the fact will be stored. This convention permits the compiler to distribute the facts in a Meld program to the correct nodes in the ensemble. It also permits the runtime to introduce facts representing the state of the world at the right nodes, i.e., facts that result from local observations are available at the corresponding module, e.g., `A` in the `temperature` predicate of Fig. 2 refers to the temperature observed at node `A`.



original rule from the temperature example:

```
localMax(A) :- temperature(A, T),
               forall neighbor(A, B)
                   [temperature(B, T'),
                    T > T'].
```

send rule after splitting:

```
__remote_LM(A, B, T') :- neighbor(B, A),
                         temperature(B, T').
```

local rule after splitting:

```
localMax(A) :- temperature(A, T),
               forall neighbor(A, B)
                   [__remote_LM(A, B, T'),
                    T > T'].
```

Fig. 3. Example of splitting a rule into its local and send parts. On the left, the spanning tree for home nodes is shown. On the right is a rule from the program in Fig. 2 along with the two rules that result from localizing it.

Just as the data is distributed to nodes in the ensemble, the rules need to be transformed to run on individual modules. Extending a technique from the P2 compiler, the rules of a program are *localized* — split into rules with local bodies — such that two kinds of rules exist. The first of these are *local rules* in which every fact in the body and head of the rule share the same home node. The second kind of rule is a *send rule* for which the entire body of the rule resides on one module while the head of the rule is instantiated on another module.

To support communication for the send rules, the compiler requires a means of determining what routes will be available at runtime. This is facilitated by special facts, called *logical neighbor facts*, which indicate runtime connectivity between pairs of modules, and potentially multi-hop routes between them. Among the axioms introduced by the runtime system are logical neighbor facts called `neighbor` facts, which indicate a node's direct communication partners. Beyond an ability to communicate (assumed to be symmetric), any meaning attributed to these facts are implementation-dependent (e.g. for Claytronics, these indicate physically neighboring modules; for sensor networks, these indicate motes within wireless range). Additional logical neighbor facts (e.g. `parent`) can be derived transitively from existing ones (e.g. two `neighbor` facts) with the route automatically generated by concatenation. Symmetry is preserved automatically by the creation of a new predicate to support the inverted version of the fact (which contains the reverse route at runtime).

Using the connectivity relations guaranteed by logical neighbor facts, the compiler is able to localize the rules and ensure that routes will be known for all send rules. The compiler considers the graph of the home nodes for all facts involved in the a rule, using the connectivity relations supplied by logical neighbor facts as edges. A spanning tree, rooted at the home node of the head of the rule, is generated (as shown in Fig. 3).

For each leaf in the tree, the compiler generates a new predicate (e.g. `__remote_LM`), which will reside on the parent node, and creates a send rule for deriving this predicate based on all of the relevant facts that reside on the leaf node. The new predicate is added

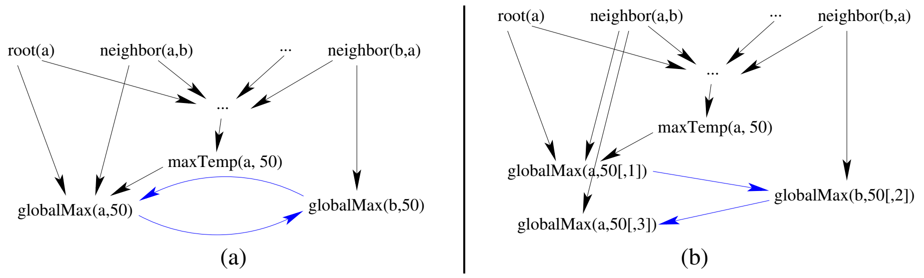


Fig. 4. Partial derivation graph for the program in Fig. 2. The graph on the left shows the derivation graph for this program using the simple reference counting approach. Note the cycle in the graph which prevents this approach from working correctly. The graph on the right shows how the cycle is eliminated through the usage of the derivation counting approach.

as a requirement in the parent, replacing the facts from the leaf node, and the leaf node is removed from the graph. This is repeated until only the root node remains at which point we are left with a local rule. Note that this process may add dependencies on symmetric versions of logical neighbor facts, such as `neighbor(B, A)` in Fig. 3.

Constraints from the original rule can be placed in the local rule's body to produce a correct implementation of the program. A better, more efficient alternative, however, places the constraints in the send rules. This way, if a constraint does not hold, then a message is not sent, effectively short-circuiting the original rule's evaluation. To this end, constraints are pushed as far down the spanning tree as possible to short-circut the process as early as possible.

The techniques of assigning home nodes, generating logical neighbors for multi-hop communications, and automaticly tranforming rules into local and send parts allow Meld to execute a program on a distributed set of communicating nodes.

4.2 Triggered Derivations

A Meld program, as a bottom-up logic, executes by deriving new facts from existing facts via application of rules. Efficient execution requires applying rules that are likely to find new derivations. Meld accomplishes this by ensuring that a new fact is used in every attempt at finding a derivation. Meld maintains a *message queue* which contains all the new facts. As a Meld program executes, a fact is pulled out of the queue. Then, all the rules that use the fact in their body are selected as candidates rules. For each candidate, the rest of its rule body is matched against the database and, if the candidate can be proven, the head of the rule is instantiated and added to the message queue. This triggered activation of rules by newly derived facts is essential to make Meld efficient.

4.3 Deletion

One of the largest hurdles to efficiently implementing Meld is that whenever the world changes we must discard all known facts and start the program over from the beginning, as described in §3.2. Fortunately, we can more selectively handle such changes by borrowing the notion of *deletion* from P2. P2 was designed for programming network

(a) Initial facts with ref counts:

```
neighbor(a,b) (×1)  root(a) (×1)
neighbor(b,a) (×1)  maxTemp(a,50) (×1)
```

(b) Facts after application of rules with reference counts:

```
neighbor(a,b) (×1)  root(a) (×1)          globalMax(b,50) (×1)
neighbor(b,a) (×1)  maxTemp(a,50) (×1)    globalMax(a,50) (×2)
```

(c) Facts after deletion of maxTemp(a,50) using basic reference counts:

```
neighbor(a,b) (×1)  root(a) (×1)          globalMax(b,50) (×1)
neighbor(b,a) (×1)  globalMax(a,50) (×1)
```

(d) Facts after application of rules with reference counts with depths:

```
neighbor(a,b) (×1)  root(a) (×1)          globalMax(b,50)(×1@2)
neighbor(b,a) (×1)  globalMax(a,50) (×1@1; ×1@3)
```

(e) Facts after deletion of maxTemp(a,50) using reference counts with depths:

```
neighbor(a,b) (×1)  neighbor(b,a) (×1)    root(a) (×1)
```

Fig. 5. Example of deletion with reference counts, and derivation counts with depth (counts and depths shown in parentheses after each fact). Based on the program from Fig. 2, the `globalMax(a,50)` fact can be cyclically derived from itself through `globalMax(b,50)`. Derivation counts that consider depth allow deletions to occur correctly, while simple reference counts fail. Facts leading up to `maxTemp(a,50)` are omitted for brevity and clarity.

overlays and uses deletion to correctly handle occasional link failures. Although the ensembles we consider may experience more frequent changes in their world, these can be handled effectively with a local, efficient implementation of deletion.

Deletion avoids the problem of simultaneously discarding every fact at every node and restarting the program by carefully removing only those facts from the system which can no longer be derived. Deletion works by considering a deleted fact and matching the rules in exactly the same way as derivations are done to determine which other facts depend on the deleted one. Each of these facts is then, in turn, deleted. Strictly following this approach will result in a "conservative" approach that deletes too many facts, e.g., ones with alternative derivations that do not depend on the previously deleted facts. This approach would be correct if at each step all possible derivations were tried again, but produces a problem given our triggered application of rules. In other words, a derivable fact that is "conservatively" deleted may never be re-derived, even though an alternate derivation may exist. Therefore, it is necessary to have an exact solution to deletion in order to use our triggered approach to derivation.

P2 addresses this issue by using reference counting techniques similar to those used in garbage collection. Instead of keeping track of the number of objects that point to an object, it keeps track of the number of derivations that can prove a particular fact. When a fact is deleted, this count is decremented. If the count reaches zero, then the fact is removed from the database and facts derived from it are recursively deleted. This approach works for simple cases, but suffers from the cyclic "pointer" problem. In Meld a fact is often used to derive another instance of itself, leading to cyclic derivation graphs (shown in Fig. 4(a)). In this case, simple reference counting fails to properly delete the fact, as illustrated in parts a–c of Fig. 5.

In the case of Meld, and unlike a reference counting garbage collector, we can resolve this problem by tracking the depth of each derivation. For facts that can be involved in a cyclic derivation, we keep a reference count for each existing derivation depth. When a fact with a simple reference count is deleted, we proceed as before. When a fact with reference counts for each derivation depth is deleted, we decrement the reference count for that derivation depth. If the smallest derivation depth is decremented to zero, then we delete the fact and everything derived from it. If one or more derivations still exist after this process completes, then we reinstantiate the fact with the new derivation depth. This process serves to delete any other derivations of the fact that depended upon the fact and eliminates the possibility of producing an infinite cyclic derivation with no start. This solution is illustrated in Fig. 4(b) and parts d–e of Fig. 5.

4.4 Concerning Deletion and Actions

Since the message queue contains both newly derived facts and the deletion of facts, an opportunity for optimization presents itself. If a new fact (F) and the deletion of that fact ($\cancel{F}$) both exist in the message queue, one might think that both F and $\cancel{F}$ can be silently removed from the queue as they cancel one another out. This would be true if all derived rules had no side-effects. However, the possibility of deriving an action requires caution.

The key difference between facts and actions is that for facts we need to know only whether it is true or not, while for an action we must act each time it is derived. The semantics of Meld require that deletions be completed "instantly," taking priority over any derivations of new facts. Thus, when F comes before $\cancel{F}$, then silently removing both from the queue is safe since $\cancel{F}$ undoes the derivation of any fact that might be derived from F.

If, however, $\cancel{F}$ comes before F, then canceling them is not safe. In this case, processing them in the order required by the semantics could result in deleting and rederiving an action, causing it to be correctly performed. Had we silently deleted both F and $\cancel{F}$, the action would not occur. Thus, this optimization breaks correctness when $\cancel{F}$ occurs before F in the queue. As a result, we only cancel out facts in the queue when the fact occurs before the deletion of the fact.

4.5 X-Y Stratification

A naïve way to implement aggregates (and `forall` statements which require similar considerations) is to assume that all values for the predicate are known, and calculate the aggregate accordingly. If a new value arrives, one can delete the old value, recompute, and instantiate the new one. At first glance, this appears to be a perfectly valid approach, though somewhat inefficient due to the additional work to clean up and update aggregate values that were based on partial data. Unfortunately, however, this is not the case, as the additional work may be arbitrarily expensive. For example, an aggregate computed with partial data early in the program may cause the entire program to execute with the wrong value; an update to the aggregate effectively entails discarding and deleting all facts produced, and rerunning the program. As this can happen multiple, times, this is clearly neither efficient nor scalable, particularly for aggregates that depend on other aggregates. Finally, there is a potential for incorrect behavior—any actions based on the wrong aggregate values may be incorrect and cannot be undone.

Rather than relying on deletion, we ensure the correctness and efficiency of aggregates by using *X-Y stratification*. X-Y stratification, used by LDL++[22], is a method for ensuring that all of the contributing values are known before calculating the value of an aggregate. This is done by imposing a global ordering on the processing of facts to ensure that all possible derivations for the relevant facts have been explored before applying an aggregate. This guarantees that the correct value of an aggregate will be calculated and eliminates the need for expensive or impossible corrections via deletion.

Unfortunately, ensuring a global ordering on facts for X-Y Stratification as described for LDL++ requires global synchronization, an expensive, inefficient process for an ensemble. We propose a safe relaxation of X-Y Stratification that requires only local synchronization and leverages an understanding of the communications paths in Meld programs. Because Meld has a notion of local rules and send rules (described in §4.1), the compiler can determine whether a fact derivation depends on facts from only the local module, the neighboring modules, or some module far away in the ensemble. Aggregation of facts that originate locally can safely proceed once all such facts have been derived locally. If a fact can come only from a neighboring module, then it is sufficient to know that all of the neighboring modules have derived all such facts and will produce no more. In these two cases, only local synchronization between a module and its immediate neighbors is necessary to ensure stratification.

Therefore, locally on each node, we impose an ordering on fact derivations. This is precisely the ordering that is provided via X-Y stratification, but it is only enforced within a node's neighborhood, i.e., between a single node and its direct neighbors. An aggregation of facts that can only be derived locally on a single node is handled in the usual way. Aggregation of facts that might come from a direct neighbor is deferred until each neighbor has promised not to send any additional facts of that type. Thus, to ensure that all the facts contributing to an aggregate are derived beforehand, some nodes are allowed to idle, even though they may be able to produce new facts based on aggregates of partial sets of facts. For the rare program that aggregates facts which can originate from an arbitrary module in the ensemble, it may be necessary to synchronize the entire ensemble. The compiler, therefore, disallows aggregates that depend upon such facts. To date we have not needed such an aggregate, but intend to investigate this further in the future.

5 Analysis and Discussion

In this section we discuss some of the advantages and disadvantages of writing programs in Meld. To facilitate this, we consider two real programs for modular robots that have been implemented in Meld in addition to the temperature averaging program for sensor networks shown in Fig. 2. These programs implement a shape change algorithm as provided by Dewey et. al. [8] (a simplified version is shown in Fig. 6) and a localization algorithm provided by Funiak et. al. [10]. The localization algorithm generates a coordinate system for an ensemble by estimating node positions from local sensor data and then iteratively refining the estimation.

```
// Choose only best state:
// FINAL=0, PATH=1, NEUTRAL=2
type state(module, min int).
type parent(module, first module).
type notChild(module, module).

// generate PATH state next to FINAL
state(B, PATH) :-
   neighbor(A, B),
   state(A, FINAL),
   position(B, Spot),
   0 = inTargetShape(Spot).

// propagate PATH/FINAL state
state(B, PATH) :-
   neighbor(A, B),
   state(A, PATH).

state(B, FINAL) :-
   neighbor(A, B),
   state(A, FINAL),
   position(B, Spot),
   1 = inTargetShape(Spot).

// construct deletion tree from FINAL
parent(B, A) :-
   neighbor(A, B),
   state(B, PATH),
   state(A, FINAL).

// extend deletion tree along PATH
parent(B, A) :-
   neighbor(A, B),
   state(B, PATH),
   parent(A, _).
```

```
// B is not a child of A
notChild(A, B) :-
   neighbor(A, B),
   parent(B, C), A != C.

notChild(A, B) :-
   neighbor(A, B),
   state(B, FINAL).


// action to destroy A, give resources to B
// can apply if A is a leaf in deletion tree
destroy(A, B) :-
   state(A, PATH),
   neighbor(A, B),
   resources(A, DESTROY),
   resources(B, DESTROY),
   forall neighbor(A, N)
      notChild(A, N).


// action to transfer resource from A to B
give(A, B) :-
   neighbor(A, B),
   resources(A, CREATE),
   resources(B, DESTROY),
   parent(A, B).


// action to create new module
create(A, Spot) :-
   state(A, FINAL),
   vacant(A, Spot),
   1 = inTargetShape(Spot),
   resources(A, CREATE).
```

Fig. 6. A metamodule-based shape planner based on [8] implemented in Meld. It uses an abstraction that provides metamodule creation, destruction, and resource transfer as basic operations. The code ensures the ensemble stays connected by forming trees and deleting only leaf nodes. This code has been tested in simulations with up to 1 million metamodules, demonstrating the scalability of the distributed Meld implementation.

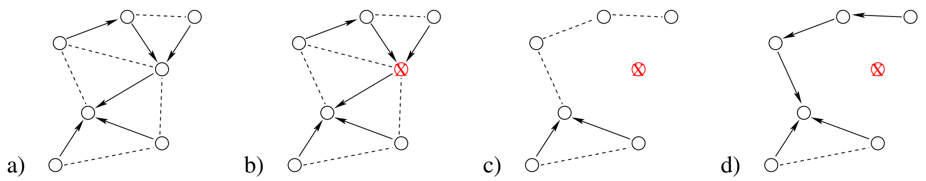


Fig. 7. The max temperature program (in Fig. 2) (a) creates a tree. When (b) a node fails, the Meld runtime is able to (c) destroy the subtree rooted at the failed node via deletion and (d) reconnect the tree.

The shape change algorithm is a motion planner for modular robots. Planning for individual modules is plagued by non-holomonic constraints, however planning can be done for groups, called *metamodules*, with only holonomic constraints. Dewey's algorithm runs on this metamodule abstraction rather than on individual modules. These metamodules are not capable of motion themselves. Instead they can be absorbed into (destroyed by) or extruded out of (created by) an adjacent metamodule. An absorbed metamodule can be transfered from one metamodule to an adjacent one, allowing it to travel throughout the ensemble as a resource. The planner makes local decisions on where to create new metamodules, destroy existing ones, and how to move resources.

5.1 Fault Tolerance

As evident from the discussion in §4, Meld inherently provides a degree of fault tolerance to programs. The operational semantics of Meld allows for arbitrary changes in the physical world; any visible change causes removal of facts that are no longer supported by the derivation rules. In the event that a module ceases to function (fail-dead), every fact that is derived from axioms about that module is deleted. New axioms, representing the new state of the world, are introduced and affected portions of the algorithm are rerun. This allows the program to run as though the failed module had never been present, modulo actions that have already occurred. As long as the program has no special dependence on this module, it continues to run and tolerates the failure. Other failures can also be tolerated as long as the program can proceed without the lost functionality.

For the temperature averaging program, this feature of Meld is very effective. If, for instance, a module fails then a break occurs in the constructed tree. In a naïve implementation in another language, this could result in a failure to complete execution or a failure to include observations from the subtree rooted at the failed node. An implementation that can tolerate such a fault and reconstruct the tree (assuming the ensemble is still connected) requires significant additional code, foresight, and effort from the programmer. The Meld implementation, however, requires nothing additional. When a module fails, Meld automatically deletes the subtree rooted at the failed node and, if the network is still connected, adds these modules back into the tree, as shown in Fig. 7.

5.2 Provability

As Meld is a logic programming language, Meld programs are generally well-suited for use in correctness proofs. In particular, the structure and semantics let one directly reason about and apply proof methods to Meld program implementations, rather than on just the specifications or translated pseudo-code representations. Furthermore, Meld code is amenable to mechanized analysis via theorem checkers such as Twelf [19]. Twelf is designed for analyzing program logics, but can be used for analyzing logic program implementations as well.

Proofs of correctness for programs involving actions, however, may need to make assumptions about what happens when an action is attempted. For the planner example, a proof of correctness has been carried out with the assumption that actions are always performed exactly as specified. The planner has been proven to achieve a correct target shape in finite time while maintaining the connectivity of the ensemble.[1] These simplifying assumptions, however, prevent any formal reasoning about fault tolerance, as discussed in §5.1. Although empirical evidence shows that the Meld implementation is indeed tolerant to some faults, a good fault model will be required to formally analyze this aspect of the program.

5.3 Messaging Efficiency

The distributed implementation of Meld is effective at propagating just the information needed for making forward progress on the program. As a result, a Meld program's message complexity can be competitive with hand-crafted messaging written in other languages. This was demonstrated in [1] for small programs and our enhancements carry this through for more complex programs that use aggregates. In particular, the use of aggregates can cause high message complexity. Before adding X-Y stratification, aggregates that depend on data received from neighbors, such as those used in the iterative refinement steps of the localization algorithm, could cause multiple re-evaluations of the aggregate as data trickled in. In the worst case, this could cause an avalanche of facts with intermediate values to be sent throughout the ensemble, each of which is then deleted and replaced with another partial result. For localization, this resulted in a lack of progress due to an explosion of messages on all but trivially small examples in the original implementation of Meld. Our addition of X-Y-stratification to Meld alleviates this issue: the result of an aggregate is not generated or propagated until all supporting facts have been seen, limiting both messaging and computation overheads. With X-Y stratification, localization has been demonstrated on ensembles of up to 10,000 nodes, with a message complexity logarithmic in the number of modules, exactly as one would expect from a high level description of the algorithm.

5.4 Memory Efficiency

Although the Meld compiler is not fully optimized for memory, many Meld programs have small memory footprints and can, therefore, fit into the limited memory available on sensor network motes and on modular robots. To test this, we measure the maximum memory used among all the modules in an ensemble executing the example Meld programs. Both the temperature aggregation program and the shape change algorithm

[1] A sketch of the proofs is available in [8] and the full proofs on the Meld source code are available in [9].

prove to have very small memory footprints, requiring at most only 488 and 932 bytes per module, respectively. The aggregation program is sensitive to neighborhood size; this was assumed to be 6, and the memory required grows by 38 bytes for each additional neighbor. Furthermore, these numbers assume 32-bit module identifiers and temperature readings; 16-bit module identifiers and data would halve the maximum memory footprint. Both of these programs fit comfortably into the memory available on a mote or a modular robot.

The localization algorithm, on the other hand, requires tens to hundreds of kilobytes of memory depending on the ensemble size. This is due to the lack of support within Meld for dynamic state. Because of this limitation, the localization algorithm is written such that it produces a new (static) estimated position fact for each step of iterative refinement. Furthermore, as the old estimates are used in the derivation of the new ones, these are not discarded and they quickly accumulate. As the ensemble grows, more steps of iterative refinement are required, generating even larger quantities of outdated facts that only serve to establish a long chain of derivation from the axioms. Thus, programs that require dynamic state (such as algorithms involving iterative refinement) can not currently be efficiently run in Meld.

6 Conclusions and Future Work

Meld has grown into a substantially more effective language for programming ensembles of independently executing nodes. Our early experiments have shown that concise and efficient programs involving very large numbers of nodes can be developed successfully. Both of the example programs in this paper (for calculating max temperature in a sensor network and for achieving a desired 3D shape in a modular robotics system) were shown to be concise and efficient in our extended version of Meld.

The Meld programs we have written thus far are, to a surprising degree, tolerant of node failure. Such robust behavior in the face of individual node failures is, we believe, an important property, especially as ensemble size grows. We also showed that Meld programs are amenable to formal analysis and proof. In particular, because of Meld's logic-programming roots, programs written in Meld can be used directly in proofs of correctness, e.g., the shape-change planner has been proven correct in this manner.

We have extended Meld in ways that enable better efficiency on larger ensembles, and believe that large ensembles are precisely where the advantages of Meld become most valuable. We described results from simulations of Meld programs running on up to 1 million nodes. For systems of this scale, we found Meld's ability to generate all of the needed messaging and distribution of state across the nodes to be a great aid in helping the programmer to understand, control, and reason about the program.

Despite all of this progress, Meld is still not an ideal language language for certain problem domains. For instance, problems requiring the maintenance of dynamic state, as demonstrated via the iterative gradient decent in the localization algorithm, are not efficiently executable in Meld. While such state can be encoded in Meld, the lack of direct support leads to suboptimal behavior. In particular, such encodings can require unbounded quantities of memory and may fall apart in the event of a fault. This issue of dynamic state will need to be addressed for Meld to become an ideal language for writing a more general class of ensemble programs. In the meantime, Meld offers distinct advantages for implementing many classes of distributed algorithms for execution on a variety of ensemble types.

Acknowledgements. We thank Robert J. Simmons and Frank Pfenning for providing feedback on and insights into Meld and this paper. We thank the anonymous reviewers for their helpful comments. This work was supported by Intel Corporation, Microsoft Corporation, and the National Science Foundation under Grant #CNS-0428738.

References

1. Ashley-Rollman, M.P., Goldstein, S.C., Lee, P., Mowry, T.C., Pillai, P.: Meld: A declarative approach to programming ensembles. In: Proc. of the IEEE Int'l Conf. on Intelligent Robots and Systems (October 2007)
2. Bachrach, J., McLurkin, J., Grue, A.: Protoswarm: A language for programming multi-robot systems using the amorphous medium abstraction. In: Int'l Conf. in Autonomous Agents and Multiagent Systems (AAMAS) (May 2008)
3. Beal, J., Bachrach, J.: Infrastructure for engineered emergence on sensor/actuator networks. IEEE Intelligent Systems 21(2), 10–19 (2006)
4. Ceri, S., Gottlob, G., Tanca, L.: What you always wanted to know about Datalog (and never dared to ask). IEEE Transactions on Knowledge and Data Engineering 1(1), 146–166 (1989)
5. Chu, D., Tavakoli, A., Popa, L., Hellerstein, J.: Entirely declarative sensor network systems (2006)
6. Culler, D., Estrin, D., Srivastava, M.: Guest editors' introduction: Overview of sensor networks. Computer 37(8), 41–49 (2004)
7. De Rosa, M., Goldstein, S.C., Lee, P., Campbell, J.D., Pillai, P.: Programming modular robots with locally distributed predicates. In: Proc. of the IEEE Int'l Conf. on Robotics and Automation (2008)
8. Dewey, D., Srinivasa, S., Ashley-Rollman, M.P., De Rosa, M., Pillai, P., Mowry, T.C., Campbell, J.D., Goldstein, S.C.: Generalizing metamodules to simplify planning in modular robotic systems. In: Proc. of Int'l Conf. on Intelligent Robots and Systems, Nice, France (September 2008)
9. Dewey, D., Srinivasa, S., Ashley-Rollman, M.P., De Rosa, M., Pillai, P., Mowry, T., Campbell, J.D., Goldstein, S.C.: Generalizing metamodules to simplify planning in modular robotic systems. Technical Report CMU-CS-08-139, Carnegie Mellon University (2008)
10. Funiak, S., Ashley-Rollman, M.P., Pillai, P., Campbell, J.D., Goldstein, S.C.: Distributed localization of modular robot ensembles. In: Proc. of the 3rd Robotics Science and Systems (2008)
11. Goldstein, S., Campbell, J., Mowry, T.: Programmable matter. IEEE Computer (June 2005)
12. Intel Corporation and Carnegie Mellon University. Dprsim: The dynamic physical rendering simulator (2006), http://www.pittsburgh.intel-research.net/dprweb/
13. Kothari, N., Gummadi, R., Millstein, T., Govindan, R.: Reliable and efficient programming abstractions for wireless sensor networks. In: PLDI 2007: Proc. of the 2007 ACM SIGPLAN conference on Programming language design and implementation, pp. 200–210. ACM, New York (2007)
14. Levis, P.: TinyOS Programming. UC - Berkeley (2006)
15. Loo, B.T., Condie, T., Garofalakis, M., Gay, D.E., Hellerstein, J.M., Maniatis, P., Ramakrishnan, R., Roscoe, T., Stoica, I.: Declarative networking: language, execution and optimization. In: Proc. of the 2006 ACM SIGMOD int'l conf. on Management of data, pp. 97–108. ACM Press, New York (2006)

16. Madden, S.R., Franklin, M.J., Hellerstein, J.M., Hong, W.: Tinydb: an acquisitional query processing system for sensor networks. ACM Trans. Database Syst. 30(1), 122–173 (2005)
17. Nagpal, R.: Programmable Self-Assembly: Constructing Global Shape Using Biologically-Inspired Local Interactions and Origami Mathematics. Ph.D thesis, MIT, MIT AI Lab. Technical Memo 2001-008 (2001)
18. Newton, R., Morrisett, G., Welsh, M.: The Regiment macroprogramming system. In: Proc. of the Int'l conf. on Information Processing in Sensor Networks (IPSN 2007) (April 2007)
19. Pfenning, F., Schürmann, C.: System description: Twelf - a meta-logical framework for deductive systems. In: Proc. of Int'l Conf. on Automated Deduction, pp. 202–206 (1999)
20. Whitehouse, K., Sharp, C., Brewer, E., Culler, D.: Hood: a neighborhood abstraction for sensor networks. In: Proc. of the 2nd int'l conf. on Mobile systems, applications, and services, pp. 99–110. ACM Press, New York (2004)
21. Yim, M., Shen, W.-M., Salemi, B., Rus, D., Moll, M., Lipson, H., Klavins, E., Chirikjian, G.S.: Modular self-reconfigurable robot systems [grand challenges of robotics]. IEEE Robotics and Automation Magazine 14(1), 43–52 (2007)
22. Zaniolo, C., Arni, N., Ong, K.: Negation and aggregates in recursive rules: the LDL++ approach. In: Ceri, S., Tsur, S., Tanaka, K. (eds.) DOOD 1993. LNCS, vol. 760, pp. 204–221. Springer, Heidelberg (1993)

Integrating Software Testing and Run-Time Checking in an Assertion Verification Framework

Edison Mera[1], Pedro Lopez-García[2,3], and Manuel Hermenegildo[2,4]

[1] Complutense University of Madrid (UCM), Spain
[2] IMDEA Software, Spain
[3] Spanish Research Council (CSIC), Spain
[4] School of Computer Science, Technical University of Madrid (UPM), Spain
edison@fdi.ucm.es, pedro.lopez@imdea.org, herme@fi.upm.es

Abstract. We present a framework that unifies unit testing and run-time verification (as well as static verification and static debugging). A key contribution of our overall approach is that we preserve the use of a unified assertion language for all of these tasks. We first describe a method for compiling run-time checks for (parts of) assertions which cannot be verified at compile-time via program transformation. This transformation allows checking preconditions and postconditions, including conditional postconditions, properties at arbitrary program points, and certain computational properties. Most importantly, we propose a minimal addition to the assertion language which allows defining unit tests to be run in order to detect possible violations of the (partial) specifications expressed by the assertions. We have implemented the framework within the Ciao/CiaoPP system and effectively applied it to the verification of ISO Prolog compliance and to the detection of different types of bugs in the Ciao system source code. Experimental results are presented that illustrate different trade-offs among program size, running time, or levels of verbosity of the messages shown to the user.

Keywords: run-time verification, unit testing, static/dynamic debugging, assertions, program verification.

1 Introduction

We present an approach that unifies *unit testing* with *run-time verification* within an overall framework that also comprises *static verification* and *static debugging* [3,7,8,11,12]. This novel framework for program development is aimed at finding bugs in programs or validating them with respect to (partial) specifications given in terms of *assertions* (using the concept of *abstractions* as over-/under-approximations of program semantics). A novel and expressive language of assertions allows describing quite general program properties [2,4,10,13].

The previous work in this context cited above has concentrated mostly on the static (i.e., compile-time) checking of such assertions as well as on techniques for reducing at compile-time the number of checks that have to be performed dynamically (i.e., at run time): any assertions present in the program are verified (or falsified) to the extent possible during the compilation phase, since compile-time checking is always preferable to run-time checking –always incomplete as a means of verification. However the existence in all practical programs of data only known at run-time and the rich nature of the properties considered make a certain degree of run-time checking inevitable –a reasonable price to pay in return for property expressiveness.

In this paper we concentrate instead on the run-time portion of the model. Our aim is to a) develop effective implementation techniques for run-time checking that integrate seamlessly into our combined compile-time/run-time framework and b), based on this, to also develop *well-integrated* facilities for unit testing. To this end, we have first developed an implementation of run-time checks, as an evolution of the approach sketched in [12], based on transforming the program into a new one which preserves the semantics of the original program and at the same time checks during its execution the assertions. Such transformation allows checking preconditions and postconditions, including conditional postconditions, i.e., postconditions that must hold only when certain preconditions hold. It also allows checking properties at arbitrary program points (i.e., in literal positions in clause bodies) as well as certain computational properties (properties that are not specific to a program point but rather to whole computations, such as, for example, determinism, non-failure, or use of resources –steps, time, memory, etc.).

Our transformation also addresses to some extent one of the main drawbacks of run-time checking (in addition to incompleteness): the overhead introduced during execution of the program. The proposed transformation reduces run-time overhead by avoiding meta-interpretation whenever possible and by using special features of the low-level language when appropriate. Also, run-time checks can be compiled inline as opposed to calling a library, saving (meta-)call overhead. Another relevant issue addressed by our transformation is being able to provide messages to the user which are as informative as possible when a violation of the safety policy is found, i.e., when a run-time check fails. To this end, the transformation saves appropriate information at source code level in the transformed file. Depending on the level of code instrumentation selected, increasingly more accurate information about the assertions is saved, and, thus, presented, offering different trade-offs between information level and program size.

With respect to *testing*, we propose a minimal extension to the assertion language in order to be able to define *unit tests* [5]. The resulting language can express for example the input data for performing such unit tests, the expected output, the number of times that the unit tests should be repeated, etc. In contrast to previous work in this area (e.g., [1], [17], or the unit test framework recently included in SWI-Prolog [16]), a key contribution of our approach is that these unit tests blend in with the assertion language and reuse the overall framework. In particular, only *test drivers* need to be added because the assertions and their run-time tests act as the checkers for the cases defined by the unit tests. An advantage of our approach is that the unit test specifications can be encapsulated in the same module that contains the predicates being tested, or placed in a separate file containing the tests for the module or modules of the application. This contrasts with, e.g., the `plunit` unit testing of SWI-Prolog, where unit test specifications are written in the source code of the module or in a dedicated file with the same name as the module being tested.

Both the run-time check generation and the unit testing approaches proposed have been implemented within the CiaoPP/Ciao system. We provide some experimental results which illustrate the implementation trade-offs involved. As mentioned before, thanks to the CiaoPP/Ciao machinery only the (parts of) assertions which cannot be verified at compile-time are converted into run-time checks. Since in our approach unit tests are also assertions, static analysis can also eliminate parts of or whole unit tests. At the same time, the tight integration also allows using the unit test drivers to exercise run-time checks corresponding to those parts of assertions that could not be checked at compile-time, even if they were not conceived as tests.

2 The Ciao Assertion Language

Assertions are linguistic constructions which allow expressing properties of programs. They allow talking about preconditions, (conditional) postconditions, whole executions, program points, etc. For space considerations, we will focus on a subset of the Ciao assertion language: assertions referring to *execution states* and *computations* (see [13,2] for a detailed description of the full language). Also, although the assertion language incorporates significant syntactic sugar, we will use only the (unfortunately more verbose) raw forms. An execution state $\langle G \,|\, \theta \rangle$ consists of the current goal G and the current constraint store θ which contains information on the values of variables. By *computation* we mean the (sorted) execution tree containing all possible sequences of reductions between execution states of a goal from a calling state.

Predicate Assertions: They refer to properties of a particular predicate. In the schemas below a concrete assertion will include concrete values in place of *Pred*, *Precond* and *Postcond*. In all schemas *Pred* is a *predicate descriptor*, i.e., a predicate symbol applied to distinct free variables, and *Precond* and *Postcond* are logic formulas about execution states, that we call *state-formulae*. An atomic *state-formula* is constructed with a *state property predicate* (e.g., `list(X)` or `X > 3`) which expresses properties about (the values) of variables. A *state-formula* can also be a conjunction or disjunction of *state-formulae*. Standard (C)LP syntax is used, with comma representing conjunction (e.g., "`( list(X), list(Y) )`") and semicolon disjunction (e.g., "`( list(X) ; int(X) )`").

– *Describing success states:* `:- success` *Pred* [`:` *Precond*] `=>` *Postcond*.

  Interpretation: in any call to *Pred*, if *Precond* succeeds in the calling state and the computation of the call succeeds, then *Postcond* should also succeed in the success state.

  Example 1. The following assertion expresses that for any call to predicate `qsort/2` with the first argument bound to a list of numbers, if the call succeeds, then the second argument should also be bound to a list of numbers:
  `:- success qsort(A,B) : list(A,num) => list(B,num).`

If *Precond* is omitted, the assertion is equivalent to:
`:- success` *Pred* `: true =>` *Postcond*`.`
and it is interpreted as "for any call to *Pred* which succeeds, *Postcond* should succeed in the success state."

– *Describing admissible calls:* $\boxed{\texttt{:- calls } \textit{Pred} \texttt{ : } \textit{Precond}\texttt{.}}$
Interpretation: in all calls to *Pred*, the formula *Precond* should succeed in the calling state.

Example 2. The following assertion expresses that in all calls to predicate `qsort/2`, the first argument should be bound to a list of numbers:
`:- calls qsort(L,R) : list(L,num).`

The set of all `call` assertions is considered *closed* in the sense that they must cover all valid calls.

– *Describing properties of the computation:*
$\boxed{\texttt{:- comp } \textit{Pred}\ [\texttt{:}\ \textit{Precond}\ ]\ \texttt{+}\ \textit{comp-formula}\texttt{.}}$
Interpretation: for any call to *Pred*, if *Precond* succeeds in the calling state, then *comp-formula* should also succeed for the computation of *Pred*.

Example 3. `:- comp qsort(L,R) :(list(L,num), var(R))+ not_fails.`
where the atom `not_fails` is implicitly interpreted as `not_fails (qsort(L,R))`, i.e., it is as if it executed $\langle qsort(L, R) \,|\, \theta\rangle$ and checked that it does not fail.

In addition, other assertion schemas such as `entry` and `exit` assertions can be used to refer to external calls to the module.[1]

Program-point assertions: The program points considered are the places in a program in which a new literal may be added, i.e., before the first literal (if any) of a clause, between two literals, and after the last literal (if any) of a clause. Program-point assertions are literals appearing at the corresponding program point and which are of the form: $\boxed{\texttt{check(}\textit{state-formula}\ \texttt{).}}$ The resulting assertion should be interpreted as "whenever computation reaches a state originated at the program point in which the assertion is, *state-formula* should succeed."

Status: Independently of the schema used, each assertion has a flag (`check`, `trust`, `true`, etc.), the assertion "status," which determines whether the assertion is to be checked, to be trusted, has already been proved correct by analysis, etc. Again for simplicity we use only the `check` status herein (which is assumed by default when no flag is present).

3 Run-Time Checking of Assertions

In this section we first focus on run-time checking of predicate assertions, and then we comment on the approach for program-point assertions. Our run-time

[1] Note that in CiaoPP the `pred` assertions of exported predicates can be used optionally instead of `entry` and `exit` assertions to define the module interface.

step one	step two
p :- *entry-checks*, *exit-preconditions-checks*, *ext-comp-checks*(**p1**), *exit-postconditions-checks*. `%    p renamed to p1 within module`	**p1** :- *calls-checks*, *success-preconditions-checks*, *comp-checks*(*call_stack*(**p2**, *locator*)), *success-postconditions-checks*. **p2** :- body$_0$. ... **p2** :- body$_n$.

Fig. 1. The *transforming procedure definitions* scheme for run-time checking

checking system is composed of a set of transformations, to be performed by the preprocessor, and a library containing a number of primitives that the transformed programs will call.

Applying the transformation that we call *transforming procedure definitions*,[2] the original predicate is rewritten so that it performs the run-time checks itself, each time it is called, and calls to it are left unchanged. Figure 1 illustrates this approach for a predicate `p`. In this transformation the original predicate `p` is renamed to `p2` and a new definition of `p`, which performs the run-time checks, is added by following two steps. "Step one" (first column of the figure) is used to add any run-time checks corresponding to, e.g., `entry` and `exit` assertions before and after a call to a new predicate `p1`. The objective of this first transformation is to separate external calls from internal ones. Then `p1` is defined so that it calls predicate `p2` and performs all run-time checks corresponding to each type of (kernel) predicate-level assertions, i.e., `calls`, `success`, or `comp` in the right place. In this kind of transformation, calls to `p` are left unchanged.

Transforming Single Predicate Assertions: We first consider the case where there is only one predicate assertion for a given predicate. We show schemes for transforming assertions into run-time checks for each type of (kernel) predicate assertion, i.e., `calls`, `success`, or `comp`. Other, higher-level assertions (such as `pred` assertions) and all additional syntactic sugar (such as modes or "star notation") are translated by the compiler into the kernel assertions before applying the transformation. These schemes express what run-time library predicates are called and where such calls are placed. Figure 2 shows the schemes, whereas the run-time library predicates are described below.[3]

`checkc(`C`,`F`)`: checks condition C and sets F to `true` or `false` depending on whether it succeeds or not. Defined as: `(\+` C `->` F `= false ;` F `= true)`.

`rtcheck(`C`)`: checks if condition C succeeds or not. If C fails, an exception is raised. This can be understood simply as `\+\+` C (so that bindings/constraints produced by the condition succeeding are removed –an *entailment* check).

[2] We refer the reader to [9] for a discussion of the trade-offs between the transformation described and an alternative one where the run-time checks are placed before and after any call to predicates affected by assertions.

[3] The schemas for `entry`/`exit` assertions are the same as the corresponding to `calls`/`success` assertions, and thus are not shown in the Figure.

Assertion:	The definition of *Pred* is transformed into:
`:- calls` *Pred* `:` *Cond*.	*Pred* `:-` `rtcheck(`*Cond*`)`, *Pred'*. *Pred'* `:-` ... .
`:- success` *Pred* `:` *Precond* `=>` *Postcond*.	*Pred* `:-` `checkc(`*Precond*`,F)`, *Pred'*, `checkif(F,`*Postcond*`)`. *Pred'* `:-` ... .
`:- comp` *Pred* `+` *Comp*.	*Pred* `:-` `check_comp(`*Comp*`(G),G,`*Pred'*`)`. *Pred'* `:-` ... .
`:- comp` *Pred* `:` *Precond* `+` *Comp*.	*Pred* `:-` `checkc(`*Precond*`,F)`, `checkif_comp(F,`*Comp*`(G),G,`*Pred'*`)`. *Pred'* `:-` ... .

Fig. 2. Translation schemes for different kinds of predicate assertions

`checkif(`F`,`P`)`: postcondition P is checked if F is `true`. If P fails, an exception is raised. This can be defined as: `(`F `== true -> rtcheck(`P`) ; true)`.

`checkif_comp(`F`,`*Comp*`(`G`),`G`,`*Pred'*`)`: checks a computational property if F is `true`, for a given computational property *Comp*(G), and a predicate *Pred'* to be checked. For example, if the property is `not_fails/1` and the predicate `qsort(A,B)`, then we call `checkif_comp(`F`,not_fails(`G`),`G`,qsort2(A,B))`. In turn, *Pred'* is used to pass the direct call to the predicate (i.e., `qsort2(A,B)` in the example). If F is `false` then *Pred'* is called, executing the procedure directly. If F is `true` then G is unified with *Pred'* and *Comp*(*Pred'*) is called. This relies on the fact that `comp` properties are written assuming that the goal to be called is passed as an argument and that they take care of both running the procedure and checking whether the computational property holds. Again, if the (in this case, computational) property does not hold, an exception is raised. The predicate `checkif_comp/4` can be defined as:
`checkif_comp(fail, _, _,` *Pred*`):- call(`*Pred*`)`.
`checkif_comp(true,` *CompCall*`,` *Pred*`,` *Pred*`):- call(`*CompCall*`)`.

`check_comp(`*Comp*`(`G`),`G`,`*Pred'*`)`: a specialized version of `checkif_comp(true,` *Comp*`(`G`),` G`,` *Pred'*`)`, where the first parameter is assumed to be `true`.

`call_stack(`C`,` L`)`: adds the current source code locator L to the locator stack S allowing to show the call stack on run-time errors. This can be understood as: `intercept(`C`, rtc_error(`S`,_`T`), throw(rtc_error([`L`|`S`],_`T`)))`.

The previous library predicates are implemented in such a way that they perform the checks without modifying the program state, introducing side effects, errors, etc. In other words, if all run-time errors are intercepted, the semantics of the program must be preserved.

Combining Several Predicate Assertions: We now consider the case where there are several assertions for a given predicate. Translating several `calls` or `success` assertions is relatively straightforward: the corresponding `rtcheck/1` and `checkc/2` are placed before the call to *Pred'*, and any calls to `checkif/2` are gathered after it. In the case of `calls` assertions run-time check exceptions for the unsatisfied assertions are thrown only if *all* such checks fail.

Combining computational properties is somewhat more involved. First we consider the case of a single `comp` assertion with several properties, such as, e.g.:

```
:- comp qsort(A,B) : (list(A, int), var(B)) + ( is_det, not_fails ).
```

In this case the properties will simply be nested in the *Comp* field as follows: *prop1*`(`*prop2*`(` *... propN*`(`*Pred'*`)` *...*`))` (the *Pred'* field stays obviously the same). For example, for the assertion above the *Comp* field will be `not_fails(is_det(qsort2(A,B)))`. If the `comp` property has a precondition, it will be checked only once and then either the *Comp* field or *Pred′* will be called.

The situation is more complex when several `comp` assertions have to be combined. Consider for example the following two `comp` assertions:

```
:- comp qsort(A,B) : (ground(A), var(B)) + is_det.
:- comp qsort(A,B) : (list(A,int), var(B)) + not_fails.
```

Assuming that `F1` and `F2` are the flags resulting from checking the conditions `ground(A), var(B)` and `list(A,int), var(B)` respectively, the composition of the two assertions above would be:

```
checkif_comp(F2, not_fails(G2), G2,
    checkif_comp(F1,is_det(G1), G1, qsort2(A,B))).
```

After all the transformations explained above have been made, an invocation of `call_stack/2` is instrumented in order to save the locator in the stack.

Program-Point Assertions: This is a comparatively simpler task than transforming predicate-level assertions: only one program point needs to be transformed for each assertion; only the `rtcheck/1` and `check_comp/1` primitives are required; and in the case of computational properties; their definitions are called directly. Clauses are transformed as follows:

Program-point assertion:	The clause is transformed into:
Pred `:-` ..., `check(`*Cond*`)`, ...	*Pred* `:-` ..., `rtcheck(`*Cond*`)`, ...
Pred `:-` ..., `check(`*CompProp*`(`*Goal*`))`, ...	*Pred* `:-` ..., `check_comp(`*CompProp*`(`*Goal*`))`, ...

4 Defining Unit Tests

In order to define a unit test we have to express on one hand *what to execute* and on the other hand *what to check* (at run-time). A key characteristic of our approach is that we use the assertion language described in Section 2 for expressing what to check. This way, the same properties that can be expressed for static or run-time checking can also be checked in unit testing. However, we have added a minimal number of elements to the assertion language for expressing *what to execute*. In particular, we have added a new assertion schema:

`:- texec` *Pred* [`:` *Precond*] [`+`*Exec-Formula*]`.`

which states that we want to execute (as a test) a call to *Pred* with its variables instantiated to values that satisfy *Precond*. *Exec-Formula* is a conjunction of properties describing how to drive this execution. In our approach many of the properties usable in *Precond* (e.g., types) can be run as value generators for these variables, so that input data can be automatically generated for the unit tests (see the technique described in [6]). However, we have defined some specific properties, such as random value generators.

Example 4. The assertion:

```
:- texec append(A, B, C) : (A=[1,2],B=[3],var(C)).
```

expresses that a call to `append/3` with the first and second arguments bound to `[1,2]` and `[3]` respectively and the third one unbound should be executed.

Example 5. We can define a unit test using the assertion in Example 4 together with the following two assertions expressing *what to check at run-time*:

```
:- check success append(A,B,C):(A=[1,2],B=[3],var(C)) => C=[1,2,3].
:- check comp    append(A,B,C):(A=[1,2],B=[3],var(C)) + not_fails.
```

The success assertion states that if a call to `append/3` with the first and second arguments bound to `[1,2]` and `[3]` respectively and the third one unbound terminates with success, then the third argument should be bound to `[1,2,3]`. The `comp` assertion says that such a call will not fail. □

The advantage of the integrated framework that we propose is that the execution expressed by a `texec` assertion for unit testing can also be used for checking parts of other assertions that could not have been checked at compile-time and thus remain as run-time checks. This way, a single set of run-time checking machinery is used for both run-time checks and unit testing. In addition, static checking of assertions can safely avoid (possibly parts of) unit test execution.

We now introduce another predicate assertion schema, the `test` schema, which can be seen as syntactic sugar for a set of predicate assertions:

`:- test` *Pred* [`:` *Precond*] [`=>` *Postcond*] [`+` *Comp-Exec-Props*]`.`

This assertion is interpreted as the combination of three assertions:[4]

`:- texec` *Pred* [`:` *Precond*] [`+` *Exec-Props*]`.`
`:- check success` *Pred* [`:` *Precond*] [`=>` *Postcond*]`.`
`:- check comp` *Pred* [`:` *Precond*] [`+`*Comp-Props*]`.`

For example, the assertion:

```
:- test append(A,B,C) : (A=[1,2],B=[3],var(C)) => C=[1,2,3]
                      +  not_fails.
```

is conceptually equivalent to that in Example 4, plus the two in Example 5.

These are examples of predefined properties that can be used in *Exec-Formula*:

`try_sols(N)`: Expresses an upper bound `N` on the number of solutions to be checked. For example, the assertion:

```
:- texec append(A, B, C): (A=X, B=Y, C=Z) + try_sols(7).
```

expresses that the call to `append(X, Y, Z)` should be executed to get at most the first 7 solutions through backtracking.

`times(N)`: Expresses that the execution should be repeated `N` times. For example, while checking ISO prolog compliance, a test for the `retract`/1 predicate failed rarely, so that the test was modified adding the primitive `times`/1:

[4] In fact, a completeness assertion –using "`<=`", see [13]– could also be generated.

```
:- test  retract_test7(A) + times(50).
retract_test7(A) :- retract((foo(A) :- A,call(A))).
```

in order to repeat the test fifty times to increase the chances of test failure.

`exception(Excep)`: Expresses that a test execution should throw the exception `Excep`. For example, consider the predicate `p/1` defined as follows:

```
p(a).
p(b) :- fail.
p(c) :- throw(error(c, "error c")).
```

The following tests succeed:

```
:- test p(A) : (A = a) + not_fails.
:- test p(A) : (A = b) + fails.
:- test p(A) : (A = c) + exception(error(c,_)).
```

The first one states that the call `p(a)` should not fail, the second one that `p(b)` should fail, and the third one that `p(c)` should raise an exception.

`user_output(String)`: Expresses that a predicate should write the string `String` into the current output stream. For example, the following test involving the library predicate `display`/ 1 succeeds:

```
:- test display(A) : (A = hello) + user_output("hello").
```

However, the following tests report an error:

```
:- test display(A) : (A = hello) + user_output("bye").
:- test display(A) : (A = hello) + user_output("hello!").
```

Other properties are provided for example to express that a predicate should write the string `Str` into the current error stream (`user_error(Str)`), to express a time-out `T` for a test execution (`resource(ub, time, T)`), or to generate random input data with a given probability distribution (e.g., for floating point numbers, including special cases like *infinite*, *not-a-number* or *zero* with sign).

5 Generating User-Friendly Messages

Whenever a run-time check fails, an exception is raised. An exception handler will then catch the exception and report the error. However, with the transformations presented so far little information can be provided to the user beyond the precondition or postcondition that is producing the violation, since this is the only parameter passed to most of the checking predicates. In contrast, during compile-time checking, when an assertion is proved not to hold, both the assertion and the program point where the assertion was violated are reported, in a format designed so that the graphical program development environment can locate these points in the source code and highlight them automatically.

In order to also provide precise information when reporting violated assertions when performing run-time checks, we have added an extra argument to the checking predicates through which certain information is passed, such as the location of the corresponding assertion(s) and the calling program point in the source code. This information can then be passed to the exception handler when the exception occurs, which prints it in a format that is compatible with that used when reporting compile-time checking errors. Thus, run-time errors can also be easily traced back to the sources automatically by the development environment. The transformation instruments the transformed code to include the necessary information.

There is a clear trade-off between the size of and the overhead introduced in the instrumented program and the quality of the messages issued. Different levels of information may be appropriate for different contexts. The current implementation of the run-time check transformations offers several optional levels of instrumentation. For brevity we report on two levels in our experiments, explained below:

Low: information is saved to report the actual assertion being violated and the property or properties that caused such violation.

High: in addition, predicates with assertions are further instrumented so that when a run-time check fails a call stack dump is also shown up to the exact program point where the violation occurs, showing for each predicate the literal in its body that caused such violation.[5]

To illustrate these levels, consider the following assertion and property definitions, in addition to a definition of `qsort/2` such as that of Figure 3:

```
:- success qsort(A,B)  => (ground(B),sorted_num_list(B)).
:- prop sorted_num_list/1.
sorted_num_list([]).
sorted_num_list([X]):- num(X).
sorted_num_list([X,Y|Z]):- num(X),num(Y),X=<Y,sorted_num_list([Y|Z]).
```

which ensures that `qsort/2` always returns a ground, *sorted* list. Assume also that the program has been written in a buggy way (to be discovered later). With *low* instrumentation level the output during execution would be similar to:

```
?- qsort([1,2],X).
{In /tmp/qsort.pl
ERROR: (lns 8-9) Run-time check failure in assertion for:
       qsort:qsort([1,2],[2,1]).
In *success*, unsatisfied property:
       sorted_num_list([2,1]).
ERROR: (lns 16-21) Check failed in qsort:qsort/2.}
```

Two errors are reported for a single run-time check failure: the first error shows the actual assertion being violated and the second marks the first clause of the predicate which violates the assertion. However, not enough information is provided to determine which literal made the erroneous call. For the *high* instrumentation level transformation the output is:

```
?- call_rtc(qsort([3,1,2],B)).
{In /tmp/qsort.pl
```

[5] This can also be done at a lower level, via engine primitives, but we are interested herein in measuring only the cost of source level transformations.

```
:- calls   qsort(A,B) :  list(A,num).
:- success qsort(A,B) :  list(A,num) => list(B,num).
:- comp    qsort(A,B) : (list(A,num), var(B)) + not_fails.

qsort([X|L],R) :- partition(L,X,L1,L2), qsort(L2,R2), qsort(L1,R1),
                  append(R2,[X|R1],R).
qsort([],[]).

:- calls   partition(A,B,C,D) : (list(A), num(B)).
:- success partition(A,B,C,D) : (list(A), num(B)) => (list(C), list(D)).
:- comp    partition(A,B,C,D) : (list(A), num(B)) +  (not_fails,is_det).

partition([],B,[],[]).
partition([E|R],C,[E|Left1],Right):- E < C, !, partition(R,C,Left1,Right).
partition([E|R],C,Left,[E|Right1]):- partition(R,C,Left,Right1).
```

Fig. 3. A quick-sort program with assertions

```
ERROR: (lns 8-9) Run-time check failure in assertion for:
       qsort:qsort([1,2],[2,1]).
In *success*, unsatisfied property:
       sorted_num_list([2,1]).
ERROR: (lns 16-21) Check failed in qosrt:qsort/2.
ERROR: (lns 16-21) Check failed when invocation of
       qsort:qsort([3,1,2],_1)
       called qsort:qsort([1,2],_2) in its body.}
{In /tmp/qsort.pl
ERROR: (lns 8-9) Run-time check failure in assertion for:
       qsort:qsort([3,1,2],[3,2,1]).
In *success*, unsatisfied property:
       sorted_num_list([3,2,1]).
ERROR: (lns 16-21) Check failed in qsort:qsort/2.}
```

This example uses the `call_rtc/1` meta-predicate to intercept the run-time error, show the related message, and continue execution as if the program where not being checked. The output makes it easier to locate the error since the call stack dump provides the list of calling predicates being checked.

Note that the first part of the assertion is not violated, since `B` is ground. However, on success the output of `qsort/2` is a sorted list but in reverse order, which gives us a hint: the arguments in the call to `append/3` are mistakenly swapped.

6 Experimental Results

We now report on some experimental results from our implementation within the Ciao/CiaoPP system of the testing and run-time checking approach proposed. Both have been integrated fully into the development environment allowing easy execution of tests and run-time checking of assertions present in modules. The system is available in the latest Ciao betas (1.13.x) at `http://www.ciaohome.org`.

Table 1. Qsort size increment with several configurations of run-time checks

Qsort	Low						High					
Obj Size:	Inline			Library			Inline			Library		
7467 (bytes)	M	T	M+T	M	T	M+T	M	T	M+T	M	T	M+T
Entry	1.41	1.69	1.77	1.34	1.38	1.44	1.66	1.94	2.02	1.57	1.61	1.68
Exit	1.55	1.82	1.97	1.28	1.33	1.44	1.78	2.06	2.21	1.50	1.55	1.65
Comp*	1.67	1.89	1.93	5.46	5.49	5.54	2.05	2.28	2.31	5.64	5.68	5.73
E/E/C	2.32	2.67	2.88	5.88	5.95	6.11	2.88	3.23	3.44	6.25	6.31	6.48
Calls	1.42	1.64	1.75	1.32	1.33	1.43	1.62	1.84	1.95	1.50	1.51	1.61
Success	1.55	1.77	1.92	1.26	1.29	1.39	1.74	1.97	2.12	1.42	1.44	1.55
Comp	1.63	1.85	1.88	5.38	5.41	5.46	2.01	2.24	2.28	5.57	5.60	5.65
C/S/C	2.10	2.46	2.65	5.66	5.73	5.88	2.63	3.00	3.20	5.98	6.11	6.26

Table 2. Slowdown of `qsort/2` with several configurations of run-time checks

Qsort	Low						High					
exec time:	Inline			Library			Inline			Library		
675 (us)	M	T	M+T	M	T	M+T	M	T	M+T	M	T	M+T
Entry	1.00	1.86	1.87	1.05	1.89	1.90	1.01	1.89	1.87	1.03	1.91	1.91
Exit	1.02	2.73	2.73	1.03	2.76	2.78	1.02	2.74	2.75	1.03	2.79	2.80
Comp*	1.01	1.87	1.87	1.02	1.93	1.92	1.02	1.88	1.90	1.05	1.91	1.92
E/E/C	1.01	3.60	3.60	1.04	3.67	3.68	1.02	3.62	3.65	1.05	3.69	3.69
Calls	3.52	165	162	76	243	321	42	207	205	135	301	382
Success	5.62	329	333	164	515	667	42	380	383	229	595	746
Comp	6.39	166	167	106	272	343	82	254	254	264	447	512
C/S/C	9.77	352	353	194	578	761	91	450	453	379	776	948

The experiments measure both program size and time overhead due to run-time checks. We first used the `qsort` program in Figure 3, with an input list of size 600 to run several experiments for different settings:

– **Library or inlined run-time checks:** we have implemented the transformation first as described in the previous sections, where the `check` predicates are assumed to be in a library (columns labeled **Library**). Ratios shown are w.r.t. the execution time of the program with no run-time checks. In addition, an alternative approach has been implemented in which the definitions of the run-time check library predicates are actually *inlined* in the calling program. This often achieves better performance but sometimes at the cost of increased code size. Note, however, that code size does not increase in all cases because such inlining is, in fact, a restricted kind of partial evaluation that tries to solve as many unifications as possible at compilation time, and sometimes terms become smaller after such optimization.

– **Use of types or modes properties:** since checking complex types, such as in the `list(int)` check, which needs to traverse lists of integers over and

over again,[6] is more expensive than checking modes (which in our case is handled through a call to the `var/1` ISO Prolog builtin) we have separated these cases in the experiments. In columns labeled T and M only types or modes are checked respectively, whereas in columns labeled $M+T$ both types and modes are checked.
– **Low or high instrumentation:** as defined in Section 5.
– **Using several kinds of assertions:** several combinations of different kinds of assertions have been tested (first column).

Tables 1 and 2 present the overhead, in size and time respectively, for the experiments expressed as a ratio w.r.t. the execution of the program with run-time checks disabled. Execution was on a MacBook Pro, Intel Core 2 Duo at 2.4Ghz, 2GB of RAM, Ubuntu Linux 8.10 and Ciao version 1.13. The columns in the tables present combinations of the configurations explained above. The rows show results for different kinds of assertions. For `comp` assertions we have that in **Comp*** the check is performed only at the entry point of the module, but not for the internal calls that occur inside.

The results show that the *high* level of instrumentation is quite expensive while the overhead implied by the *low* level is better, specially in the case of inlining. This confirms our expectations. The high overhead implied by the *high* level of instrumentation is due in part to the simplistic way in which this type of instrumentation is implemented for these experiments. Note also that the values of the `Library` column are quite large when compared with the ones of the `Inline` column because the inline transformation avoids metacalls.

Table 3 shows experimental results for larger programs, namely, the Ciao, CiaoPP, and LPdoc systems (including the libraries they use), all of which contain numerous assertions in their code. It shows the size (in kilobytes) of binary and object files using several instrumentation levels of run-time checks. The binary refers to the statically-linked executable of the main module of such systems which corresponds to the command-line executable. The object files include all the libraries used by such systems. Note that in all cases the sizes of the files depend on the number of assertions instrumented for run-time checking. Interestingly, the impact of run-time tests on execution time in these much larger benchmarks is much smaller than for `qsort`. For example, the overhead introduced in the execution of LPdoc, which includes a good number of assertions in its source, is in practice below the measurement noise level.

Regarding unit tests, we have added at the time of writing 220 unit tests to the Ciao/CiaoPP system (in addition to the other traditional system tests which did not use the unit test framework). These tests have been effective in detecting some errors introduced in those modules during later code changes. The execution time of such tests is approximately 90 seconds in the computer described before. We also have applied the implemented framework to the verification of ISO Prolog compliance of Ciao. We have coded 976 unit tests for this

[6] This overhead can be significantly reduced via multiple specialization [15,14]. However, that optimization has not been applied in this case in order to measure the overhead of fully checking the assertion.

Table 3. Size (in kilobytes) of binary and object files using several instrumentation levels of run-time checks, for large benchmarks

App	Source Metrics				Compiled		Run-Time Checked (ratio)			
Name	Size		Assertions		Binary		Low		High	
	Lines		Modules		Object		Inline	Library	Inline	Library
Ciao	S	4340	A	3230	B	2970	1.22	1.22	1.25	1.24
	L	131392	M	634	O	16312	1.21	1.19	1.24	1.22
CiaoPP	S	4831	A	1199	B	13026	1.21	1.21	1.23	1.23
	L	152365	M	517	O	14562	1.21	1.20	1.23	1.22
LPdoc	S	438	A	153	B	1929	1.10	1.07	1.11	1.07
	L	12750	M	21	O	1167	1.13	1.08	1.14	1.09

purpose. These allowed the detection of a large number of previously unknown limitations and errors: 262 issues related to non-compliance with the standard, 90 related to missing predicates or functionality, and 39 related to bugs in the functionality. While a large number of these were repetitions of a few individual errors they have been nevertheless very useful. These tests currently run in under 15 seconds. This time is much less than the other tests for Ciao because they are concentrated in only one file and the driver does not need to scan all the source code. Note that in these experiments we are not doing any compile-time checking, which would in fact eliminate many of the unit tests.

7 Conclusions

We have described our design and implementation of a framework that unifies unit testing and run-time verification (as well as static verification and static debugging). A key contribution of our approach is that a unified assertion language is used for all of these tasks. We have proposed methods for compiling run-time checks for (parts of) assertions which cannot be verified at compile-time via program transformation. We have also proposed a minimal addition to the assertion language which allows defining unit tests to be run in order to detect possible violations of the (partial) specifications expressed by the assertions. We have implemented the framework within the Ciao/CiaoPP system and presented some experimental results to illustrate different trade-offs among program size, running time, or levels of verbosity of the messages shown to the user. The experimental results confirm our expectations regarding these trade-offs: run-time checks do not pose an excessive amount of overhead when low levels of instrumentation are introduced and the calls to library predicates are inlined. The tests and run-time checks are proving quite useful in practice for detecting bugs.

Acknowledgments. This work was funded in part by EU projects FET IST-15905 *MOBIUS*, IST-215483 *SCUBE*, FET IST-231620 *HATS*, and 06042-ESPASS, Ministry of Science projects TIN-2008-05624 *DOVES*, Ministry of Industry project FIT-340005-2007-14, and CAM project S-0505/TIC/0407 *PROMESAS*.

References

1. Belli, F., Jack, O.: Implementation-based Analysis and Testing of Prolog Programs. In: ISSTA 1993: Proc. of the ACM SIGSOFT Int'l. Symp. on Software Testing and Analysis, pp. 70–80. ACM, New York (1993)
2. Bueno, F., Cabeza, D., Carro, M., Hermenegildo, M., López-García, P., Puebla, G. (eds.): The Ciao System. Ref. Manual (v1.13). Technical report, C. S. School, UPM (2006), http://www.ciaohome.org
3. Bueno, F., Deransart, P., Drabent, W., Ferrand, G., Hermenegildo, M., Maluszynski, J., Puebla, G.: On the Role of Semantic Approximations in Validation and Diagnosis of Constraint Logic Programs. In: Proc. of the 3rd. Int'l WS on Automated Debugging–AADEBUG, May 1997, pp. 155–170. U. Linköping Press (1997)
4. The CLIP Group. Program Assertions. The Ciao System Documentation Series – TR CLIP4/97.1, Facultad de Informática, UPM (August 1997)
5. Eickelmann, N.S., Richardson, D.J.: An Evaluation of Software Test Environment Architectures. In: ICSE 1996: Proc. of the Int'l. Conf. on Software Engineering, pp. 353–364. IEEE Computer Society, Los Alamitos (1996)
6. Gómez-Zamalloa, M., Albert, E., Puebla, G.: On the Generation of Test Data for Prolog by Partial Evaluation. In: Workshop on Logic-based methods in Programming Environments (WLPE 2008), vol. WLPE/2008/06, pp. 26–43 (2008)
7. Hermenegildo, M., Puebla, G., Bueno, F.: Using Global Analysis, Partial Specifications, and an Extensible Assertion Language for Program Validation and Debugging. In: The Logic Programming Paradigm: a 25–Year Perspective, pp. 161–192. Springer, Heidelberg (1999)
8. Hermenegildo, M., Puebla, G., Bueno, F., López García, P.: Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). Science of Comp. Progr. 58(1–2) (2005)
9. Mera, E., López-García, P., Hermenegildo, M.: Towards Integrating Run-Time Checking and Software Testing in a Verification Framework. Technical Report CLIP1/2009.0, T. U. Madrid (UPM) (March 2009)
10. Puebla, G., Bueno, F., Hermenegildo, M.: An Assertion Language for Debugging of Constraint Logic Programs. In: ILPS 1997 WS on Tools and Environments for (C)LP (October 1997), ftp://clip.dia.fi.upm.es/pub/papers-/assert_lang_tr_discipldeliv.ps.gz
11. Puebla, G., Bueno, F., Hermenegildo, M.: A Framework for Assertion-based Debugging in Constraint Logic Programming. In: Bossi, A. (ed.) LOPSTR 1999. LNCS, vol. 1817. Springer, Heidelberg (2000)
12. Puebla, G., Bueno, F., Hermenegildo, M.: A Generic Preprocessor for Program Validation and Debugging. In: Deransart, P., Małuszyński, J. (eds.) DiSCiPl 1999. LNCS, vol. 1870, pp. 63–107. Springer, Heidelberg (2000)
13. Puebla, G., Bueno, F., Hermenegildo, M.: An Assertion Language for Constraint Logic Programs. In: Deransart, P., Małuszyński, J. (eds.) DiSCiPl 1999. LNCS, vol. 1870, pp. 23–61. Springer, Heidelberg (2000)
14. Puebla, G., Hermenegildo, M.: Implementation of Multiple Specialization in Logic Programs. In: Proc. ACM SIGPLAN Symposium on Partial Evaluation and Semantics Based Program Manipulation, June 1995, pp. 77–87. ACM Press, New York (1995)
15. Puebla, G., Hermenegildo, M.: Abstract Multiple Specialization and its Application to Program Parallelization. JLP 41(2&3), 279–316 (1999)
16. Wielemaker, J.: SWI Prolog Unit Tests, http://www.swi-prolog.org/pldoc/package/plunit.html
17. Zhao, L., Gu, T., Qian, J., Cai, G.: Test Frame Updating in CPM Testing of Prolog Programs. Software Quality Control 16(2), 277–298 (2008)

Debugging for Model Expansion

Johan Wittocx[⋆], Hanne Vlaeminck, and Marc Denecker

Department of Computer Science, K.U. Leuven, Belgium

Abstract. Due to the development of efficient solvers, declarative problem solving frameworks based on model generation are becoming more and more applicable in practice. However, there are almost no tools to support debugging in these frameworks. For several reasons, current solvers are not suitable for debugging by tracing. In this paper, we propose a new solver algorithm for one of these frameworks, namely Model Expansion, that allows for debugging by tracing. We explain how to explore the trace of this solver in order to quickly locate a bug and we compare our debugging method with existing ones for Answer Set Programming and the Alloy system.

1 Introduction

In many real-life problems, one searches for objects of complex nature such as plans, schedules and assignments. Such objects are often naturally represented as finite structures satisfying a theory in a formal logic. This observation led to the development of several declarative problem solving frameworks based on the computational task of *finite model generation*. Prominent examples of such frameworks are Answer Set Programming (ASP), Propositional Satisfiability (SAT) and Constraint Programming (CP). Although much progress is being made in the development of efficient solving algorithms, debugging methods for these frameworks are still in their infancy.

In this paper, we propose a novel debugging method for *model expansion* (MX), a convenient extension of model generation. An MX problem for a logic $\mathcal{L}$ is the problem of finding models of a given $\mathcal{L}$-theory T that expand a given finite interpretation I_σ for a subset σ of the symbols in T. Every problem in **NP** can be cast as an MX problem for (extensions of) classical first-order logic (FO). E.g., to cast the well-known graph colouring problem as an MX problem for sorted FO, let T be the following theory:

$$\forall v\ \exists c\ Col(v,c). \tag{1}$$

$$\forall v, c_1, c_2\ (Col(v,c_1) \land Col(v,c_2) \supset c_1 = c_2). \tag{2}$$

$$\forall v_1, v_2, c_1, c_2\ (Col(v_1,c_1) \land Col(v_2,c_2) \land Edge(v_1,v_2) \supset c_1 \neq c_2) \tag{3}$$

Here, $Col(v,c)$ means that vertex v has colour c, and $Edge(v_1,v_2)$ means that there is an edge between vertices v_1 and v_2. The given structure I_σ specifies the

[⋆] Johan Wittocx is research assistant of the Fund for Scientific Research Flanders (FWO-Vlaanderen).

available colours, the vertices of the input graph and the edges of that graph, i.e., an interpretation for the predicate $Edge/2$. A solution to this MX problem is an expansion M of I_σ with an interpretation of the predicate Col such that $M \models T$. Such an interpretation describes a proper colouring of the input graph, using the colours listed in I_σ.

Typically, the theory of an MX problem is more compact and readable than a program to solve the same problem in a standard programming language. Nevertheless, bugs are made when writing MX specifications. Such bugs manifest themselves in two different ways, which require a different sort of debugging support. A bug causes a solver to either produce an unintended model, or to omit an intended one. To illustrate the former type of bug, assume that a user makes a typo in the above graph colouring example and writes the tautology

$$\forall v, c_1, c_2\ (Col(v, c_1) \land Col(v, c_2) \supset \underline{c_1 = c_1}) \tag{4}$$

instead of sentence (2). This will cause a solver to produce models where some nodes have more than one colour. By inspecting these models, a user can deduce that the bug is located in sentence (4), since this is the constraint that should express that a vertex has at most one colour. The second type of bug is often more difficult to locate. E.g., if a user makes the typical mistake of assuming that variables with different names take different values [19], and therefore writes

$$\forall v, c_1, c_2\ \neg(Col(v, c_1) \land Col(v, c_2)). \tag{5}$$

instead of (2), then a solver will answer that the problem has no solution. Indeed, (5) forces that $Col(v, c)$ is false for every v and c, which contradicts (1). Observe that a user has no clue of where to search for a bug now. In this paper, we focus on debugging support for the second type of bugs.

The most used approach to debug programs in a standard programming language is by analyzing the *trace*, i.e., the sequence of steps performed while running the program. Also, debugging by analyzing a trace has proven to be useful in many declarative programming contexts such as Prolog [18,4], Haskell [16], ILP [21], constraint programming [14] and deductive databases [10]. This suggests to debug MX theories by analyzing the trace of an MX solver. However, to make such an approach work, the solver should satisfy the following two requirements:

1. All reasoning steps should be as simple as possible. At least, they should be clear for someone who knows the informal semantics of the used logic.
2. All reasoning should be shown on the original theory as provided by the user. If the solver relies on a (preprocessing) phase where the theory is brought into some normal form, it should be possible to translate its reasoning back into reasoning on the original theory. Indeed, reasoning on a transformed theory is not transparent for a user.

Current MX solvers satisfy neither the first nor the second requirement.

In this paper, we propose a new MX solver algorithm for FO that satisfies the two requirements above. Hence, it allows for debugging by analyzing the trace.

The solver is based on formal proof system for MX, which is called the *MX-calculus* and is introduced in Section 3. If an MX problem has no solutions, the trace of the solver corresponds to an MX-calculus proof for the inconsistency. If the solver finds a model, this model can easily be extracted from the trace. In Section 4, we present two techniques to further facilitate debugging. The first one allows a user to describe (part of) expected models that were omitted by a solver. This yields smaller, and hence more comprehensive, proofs. The other technique consists of an interactive sessions that guides the user to relevant parts of a proof. To show that our debugging approach can be used for richer logics than FO, we extend in Section 5 the MX-calculus to FO(ID), an extension of FO with inductive definitions. Finally, we compare our debugging approach with the (very different) approaches proposed for ASP and the Alloy language.

2 Preliminaries

2.1 First-Order Logic

We assume the reader is familiar with first-order logic (FO). We introduce the conventions and notations used throughout this paper.

A vocabulary Σ consists of variables, predicate and function symbols. Variables are denoted by lowercase letters, predicate and function symbols by uppercase letters. Tuples of variables are denoted by $\overline{x}$, $\overline{y}$, etc. Abusing notation, we also use $\overline{x}$ to denote the set of variables occurring in the tuple $\overline{x}$. For a formula φ, we often write $\varphi[\overline{x}]$ to indicate that $\overline{x}$ are the free variables of φ.

The truth values *true* and *false* are denoted by $\mathbf{t}$ and $\mathbf{f}$. A Σ-interpretation I consists of a domain D and an assignment of appropriate values to each of the symbols in Σ, i.e.:

– an element $x^I \in D$ to every variable $x \in \Sigma$;
– a relation $P^I \subseteq D^n$ to every n-ary predicate symbol $P \in \Sigma$;
– a function $F^I : D^n \rightarrow D$ to every n-ary function symbol $F \in \Sigma$.

An interpretation for only the predicate and function symbols of Σ is called a *Σ-structure*. For a variable x and an element $d \in D$, $I[x/d]$ is the interpretation that assigns d to x and corresponds to I on all other symbols. This notation is extended to tuples of variables and domain elements of the same length. The truth value $I(\varphi)$ of a formula φ in I and the satisfaction relation $\models$ are defined as usual. The restriction of I to a vocabulary $\sigma \subseteq \Sigma$ is denoted by $I|_\sigma$. If $I|_\sigma = J$, then I is called an *expansion of J to Σ*.

The result of replacing all free occurrences of a variable x in a formula $\varphi[x, \overline{y}]$ by a domain element d is denoted by $\varphi[d, \overline{y}]$. The result $\varphi[\overline{d}]$ of replacing all free variables $\overline{x}$ of a formula $\varphi[\overline{x}]$ by domain elements $\overline{d}$ is called an *instance of* $\varphi[\overline{x}]$. If φ is an atom, then an instance of φ is also called a *domain atom*. The truth value of $\varphi[\overline{d}]$ in an interpretation I is defined by $I(\varphi[\overline{d}]) := I[\overline{x}/\overline{d}](\varphi[\overline{x}])$. If $I(\varphi[\overline{d}]) = \mathbf{t}$, then we also write $I \models \varphi[\overline{d}]$.

2.2 Model Expansion

We now formally define model expansion for FO.

Definition 1. *Let T be an FO theory over a vocabulary Σ, σ a subvocabulary of Σ and I_σ a finite σ-structure. The* model expansion (MX(FO)) search problem for input *$\langle \Sigma, T, \sigma, I_\sigma \rangle$ is the problem of finding models M of T that expand I_σ. The corresponding* MX(FO) decision problem *is the problem of deciding whether such a model exists.*

The MX(FO) decision problem is in **NP**. As shown in [15], it follows from Fagin's theorem on $\exists$SO that MX(FO) *captures* **NP**: for each **NP** decision problem P over finite σ-structures, there exists a vocabulary $\Sigma \supseteq \sigma$ and a theory T over Σ such that for any σ-structure I_σ, the answer to P is the answer to the MX(FO) decision problem with input $\langle \Sigma, T, \sigma, I_\sigma \rangle$. Moreover, it is often the case that T is a natural modelling of the problem P.

2.3 Three-Valued Interpretations

Besides the normal (two-valued) interpretations, we will also use three-valued ones. We denote the truth value *unknown* by $\mathbf{u}$. A three-valued Σ-interpretation $\tilde{I}$ consists of a domain D and an assignment of

– an element $x^{\tilde{I}} \in D$ to every variable $x \in \Sigma$;
– a function $P^{\tilde{I}} : D^n \rightarrow \{\mathbf{t}, \mathbf{f}, \mathbf{u}\}$ to every n-ary predicate symbol $P \in \Sigma$;
– a function $F^{\tilde{I}} : D^n \rightarrow (\mathcal{P}(D) \setminus \emptyset)$ to every n-ary function symbol $P \in \Sigma$.

If for every tuple $\overline{d}$ of domain elements, predicate P and function F, it holds that $P^{\tilde{I}}(\overline{d}) \neq \mathbf{u}$ and $F^{\tilde{I}}(\overline{d})$ is a singleton, then $\tilde{I}$ is two-valued: it corresponds to the interpretation I defined by $\overline{d} \in P^I$ iff $P^{\tilde{I}} = \mathbf{t}$ and $F^I(\overline{d}) = d$ iff $F^{\tilde{I}} = \{d\}$.

The *precision order* $<_p$ on the set of truth values is induced by $\mathbf{u} <_p \mathbf{f}$ and $\mathbf{u} <_p \mathbf{t}$. This order is extended to three-valued Σ-structures: if $\tilde{I}$ and $\tilde{J}$ have the same domain D, then we define $\tilde{I} \leq_p \tilde{J}$ iff $P^{\tilde{I}}(\overline{d}) \leq_p P^{\tilde{J}}$ and $F^{\tilde{I}}(\overline{d}) \supseteq F^{\tilde{J}}(\overline{d})$ for every $\overline{d}$, P and F. Observe that two-valued structures are maximally precise three-valued structures. On the other hand, the least precise three-valued structure assigns $P^{\tilde{I}}(\overline{d}) = \mathbf{u}$ and $F^{\tilde{I}}(\overline{d}) = D$ for every $\overline{d}$, P and F.

Using the above concepts, we define a generalization of the MX problem.

Definition 2. *Let Σ be a vocabulary, T an FO theory over Σ and $\tilde{I}$ a three-valued Σ-structure with finite domain D. A two-valued Σ-structure M is a solution to the* MX(FO) search problem *with input $\langle T, \tilde{I} \rangle$ if $M \models T$ and $M \geq_p \tilde{I}$.*

The MX(FO) problem with input $\langle \Sigma, T, \sigma, I_\sigma \rangle$ as defined in definition 1 corresponds to the MX(FO) problem with input $\langle T, \tilde{I} \rangle$ where $\tilde{I}|_\sigma = I_\sigma$ and $\tilde{I}|_{\Sigma \setminus \sigma}$ is the least precise $(\Sigma \setminus \sigma)$-structure. In the rest of this paper, we use the generalized definition. We denote the set of solutions of the MX search problem with input $\langle T, \tilde{I} \rangle$ by $\mathrm{MX}_T(\tilde{I})$. If $\mathrm{MX}_T(\tilde{I}) = \emptyset$, we say that the MX problem is *unsatisfiable.*

3 The Model Expansion Calculus

As mentioned in the introduction, the debugging method we propose in this paper relies on an MX-solver that outputs a *proof* of inconsistency in case its input is an unsatisfiable MX problem. In this section, we present a formal proof system, called the *MX-calculus* to represent such a proof.

3.1 MX-Trees

For the rest of this paper, fix a theory T and a finite three-valued structure $\tilde{I}$. Proofs for $\langle T, \tilde{I} \rangle$ in the MX-calculus are built using rules of the form

$$\frac{\mathcal{I}_1, \dots, \mathcal{I}_n}{\mathcal{J}_1 \mid \dots \mid \mathcal{J}_m} \tag{6}$$

where $\mathcal{I}_1, \dots, \mathcal{I}_n, \mathcal{J}_1, \dots, \mathcal{J}_m$ are *signed instances*: pairs of an instance $\varphi[\overline{d}]$ and a positive ($\oplus$) or negative ($\ominus$) *sign*. Signed instances are denoted by $\varphi[\overline{d}]^\oplus$ or $\varphi[\overline{d}]^\ominus$. We call $\mathcal{I}_1, \dots, \mathcal{I}_n$ the *premises* of the rule, and $\mathcal{J}_1, \dots, \mathcal{J}_m$ its *consequences*. Intuitively, the rule means that if all its positive, respectively negative, premises are true, respectively false, then at least one of its consequences is positive and true or negative and false. A rule is *sound* if its intuitive meaning is indeed a sound reasoning. More precisely:

Definition 3. *If $\mathcal{I}$ is the signed instance $\varphi[\overline{d}]^\oplus$, respectively $\varphi[\overline{d}]^\ominus$, then denote by $\mathfrak{S}(\mathcal{I})$ the instance $\varphi[\overline{d}]$, respectively $\neg\varphi[\overline{d}]$. A rule of the form (6) is* sound with respect to $\langle T, \tilde{I} \rangle$ *if for every $M \in \mathrm{MX}_T(\tilde{I})$ such that $M \models \bigwedge_{1 \le i \le n} \mathfrak{S}(\mathcal{I}_i)$, it holds that $M \models \bigvee_{1 \le i \le m} \mathfrak{S}(\mathcal{J}_i)$.*

We distinguish between three types of rules in the MX-calculus: initialization, propagation and cut rules. All of them are sound with respect to $\langle T, \tilde{I} \rangle$.

Initialization Rules. The following are the seven initialization rules for $\langle T, \tilde{I} \rangle$. None of them has premises.

$$\text{(I+}\downarrow\text{)}\ \frac{}{\varphi^\oplus} \qquad \text{(I-}\uparrow\text{)}\ \frac{}{d = d'^\ominus} \qquad \text{(I-}\uparrow\text{)}\ \frac{}{Q(\overline{d})^\ominus} \qquad \text{(I-}\uparrow\text{)}\ \frac{}{G(\overline{d}) = d^\ominus}$$

$$\text{(I+}\uparrow\text{)}\ \frac{}{d = d^\oplus} \qquad \text{(I+}\uparrow\text{)}\ \frac{}{P(\overline{d})^\oplus} \qquad \text{(I+}\uparrow\text{)}\ \frac{}{F(\overline{d}) = d^\oplus}$$

Here, φ is a sentence of T, d and d' two different domain elements, $P(\overline{d})$ an atom such that $\tilde{I}(P(\overline{d})) = \mathbf{t}$, $Q(\overline{d})$ an atom such that $\tilde{I}(Q(\overline{d})) = \mathbf{f}$, F a function such that $F^{\tilde{I}}(\overline{d}) = \{d\}$ and G a function such that $d \notin G^{\tilde{I}}(\overline{d})$. Intuitively, rule (I+$\downarrow$) expresses that a sentence of T is necessarily true. The other rules assert the truth value in $\tilde{I}$ of an atom that is not unknown in $\tilde{I}$.

Propagation Rules. If the domain D of $\tilde{I}$ is given by $D = \{d_1, \dots, d_n\}$, the following are all propagation rules for $\langle T, \tilde{I} \rangle$.

– negation rules:

$$(\neg\text{-}\downarrow)\ \frac{\neg\varphi^{\oplus}}{\varphi^{\ominus}} \qquad (\neg\text{+}\downarrow)\ \frac{\neg\varphi^{\ominus}}{\varphi^{\oplus}} \qquad (\neg\text{-}\uparrow)\ \frac{\varphi^{\oplus}}{\neg\varphi^{\ominus}} \qquad (\neg\text{+}\uparrow)\ \frac{\varphi^{\ominus}}{\neg\varphi^{\oplus}}$$

– conjunction rules (where $j \in [1, m]$):

$$(\wedge\text{+}\uparrow)\ \frac{{\varphi_1}^{\oplus}, \ldots, {\varphi_m}^{\oplus}}{\bigwedge_{i\in[1,m]} {\varphi_i}^{\oplus}} \qquad (\wedge\text{-}\uparrow)\ \frac{{\varphi_j}^{\ominus}}{\bigwedge_{i\in[1,m]} {\varphi_i}^{\ominus}} \qquad (\wedge\text{+}\downarrow)\ \frac{\bigwedge_{i\in[1,m]} {\varphi_i}^{\oplus}}{{\varphi_j}^{\oplus}}$$

$$(\wedge\text{-}\downarrow)\ \frac{\bigwedge_{i\in[1,m]} {\varphi_i}^{\ominus}, {\varphi_1}^{\oplus}, \ldots, {\varphi_{j-1}}^{\oplus}, {\varphi_{j+1}}^{\oplus}, \ldots, {\varphi_j}^{\oplus}}{{\varphi_j}^{\ominus}}$$

– disjunction rules (where $j \in [1, m]$):

$$(\vee\text{-}\uparrow)\ \frac{{\varphi_1}^{\ominus}, \ldots, {\varphi_m}^{\ominus}}{\bigvee_{i\in[1,m]} {\varphi_i}^{\ominus}} \qquad (\vee\text{+}\uparrow)\ \frac{{\varphi_j}^{\oplus}}{\bigvee_{i\in[1,m]} {\varphi_i}^{\oplus}} \qquad (\vee\text{-}\downarrow)\ \frac{\bigvee_{i\in[1,m]} {\varphi_i}^{\ominus}}{{\varphi_j}^{\ominus}}$$

$$(\vee\text{+}\downarrow)\ \frac{\bigvee_{i\in[1,m]} {\varphi_i}^{\oplus}, {\varphi_1}^{\ominus}, \ldots, {\varphi_{j-1}}^{\ominus}, {\varphi_{j+1}}^{\ominus}, \ldots, {\varphi_j}^{\ominus}}{{\varphi_j}^{\oplus}}$$

– universal rules:

$$(\forall\text{+}\uparrow)\ \frac{\varphi[d_1]^{\oplus}, \ldots, \varphi[d_n]^{\oplus}}{\forall x\ \varphi[x]^{\oplus}} \qquad (\forall\text{-}\uparrow)\ \frac{\varphi[d_i]^{\ominus}}{\forall x\ \varphi[x]^{\ominus}} \qquad (\forall\text{+}\downarrow)\ \frac{\forall x\ \varphi[x]^{\oplus}}{\varphi[d_i]^{\oplus}}$$

$$(\forall\text{-}\downarrow)\ \frac{\forall x\ \varphi[x]^{\ominus}, \varphi[d_1]^{\oplus}, \ldots, \varphi[d_{i-1}]^{\oplus}, \varphi[d_{i+1}]^{\oplus}, \ldots, \varphi[d_n]^{\oplus}}{\varphi[d_i]^{\ominus}}$$

– existential rules:

$$(\exists\text{-}\uparrow)\ \frac{\varphi[d_1]^{\ominus}, \ldots, \varphi[d_n]^{\ominus}}{\exists x\ \varphi[x]^{\ominus}} \qquad (\exists\text{+}\uparrow)\ \frac{\varphi[d_i]^{\oplus}}{\exists x\ \varphi[x]^{\oplus}} \qquad (\exists\text{-}\downarrow)\ \frac{\exists x\ \varphi[x]^{\ominus}}{\varphi[d_i]^{\ominus}}$$

$$(\exists\text{+}\downarrow)\ \frac{\exists x\ \varphi[x]^{\oplus}, \varphi[d_1]^{\ominus}, \ldots, \varphi[d_{i-1}]^{\ominus}, \varphi[d_{i+1}]^{\ominus}, \ldots, \varphi[d_n]^{\ominus}}{\varphi[d_i]^{\oplus}}$$

– equality rules:

$$(=\pm\updownarrow)\ \frac{\mathcal{I}, t_1 = {t_2}^{\oplus}}{\mathcal{I}'}$$

where $\mathcal{I}'$ is the result of replacing in $\mathcal{I}$ an occurrence of t_1 by t_2, or an occurrence of t_2 by t_1.

– function rules (where $d_j \neq d_k$):

$$\text{(F-}\updownarrow\text{)}\quad \frac{F(\overline{d}) = d_j{}^{\oplus}}{F(\overline{d}) = d_k{}^{\ominus}}$$

$$\text{(F+}\updownarrow\text{)}\quad \frac{F(\overline{d}) = d_1{}^{\ominus}, \ldots, F(\overline{d}) = d_{i-1}{}^{\ominus}, F(\overline{d}) = d_{i+1}{}^{\ominus}, \ldots, F(\overline{d}) = d_n{}^{\ominus}}{F(\overline{d}) = d_i{}^{\oplus}}$$

We stress that each of these rules is easy to understand. E.g., disjunction rule ($\vee$+$\uparrow$) says that a disjunction is true if one of its disjuncts is true. Universal rule ($\forall$-$\downarrow$) says that if a formula $\forall x\ \varphi[x]$ is false, but for all domain elements d except d_i, the instance $\varphi[d]$ is true, then $\varphi[d_i]$ is false. Indeed, if $\varphi[d_i]$ would be true, then also $\forall x\ \varphi[x]$ would be true.

Cut Rule. The cut rule for $\langle T, \tilde{I}\rangle$ is given by $\dfrac{}{\varphi[\overline{d}]^{\oplus} \quad | \quad \varphi[\overline{d}]^{\ominus}}$

3.2 Soundness and Completeness

An MX-calculus proof for the inconsistency of $\mathrm{MX}_T(\tilde{I})$ is a tree, built using the rules defined above, such that each of its branches contains a contradiction. Formally, it is defined as follows.

Definition 4. *An* MX-rule *is an initialization, propagation or cut rule. An* MX-tree for $\langle T, \tilde{I}\rangle$ *is inductively defined by*

– *the empty tree is an MX-tree for $\langle T, \tilde{I}\rangle$;*
– *if $\mathcal{T}$ is an MX-tree for $\langle T, \tilde{I}\rangle$, B a branch of $\mathcal{T}$ and $\dfrac{\mathcal{I}_1, \ldots, \mathcal{I}_n}{\mathcal{J}_1 \mid \ldots \mid \mathcal{J}_m}$ an MX-rule for $\langle T, \tilde{I}\rangle$ such that all $\mathcal{I}_i$ occur in B, then the result of adding in $\mathcal{T}$ all $\mathcal{J}_1 \ldots \mathcal{J}_m$ as children to the leaf of B is an MX-tree for $\langle T, \tilde{I}\rangle$.*

Example 1. Let T_1 be the theory consisting of sentence (1), (3) and (5) of the introduction, and let $\tilde{I}_1$ be a three-valued structure with domain D_1 containing precisely the two colours **red** and **blue**, and at least one node **d**. Assume $Col^{\tilde{I}_1}(\mathbf{d}, \mathbf{blue}) = \mathbf{f}$. Figure 1 shows an MX-tree for $\langle T_1, \tilde{I}_1\rangle$. The used MX-rules and premises are indicated next to each node.

We say that a branch of an MX-tree is *closed* if for some instance $\varphi[\overline{d}]$, it contains both $\varphi[\overline{d}]^{\oplus}$ and $\varphi[\overline{d}]^{\ominus}$. We call $\varphi[\overline{d}]^{\oplus}$ and $\varphi[\overline{d}]^{\ominus}$ *conflicting instances* of that branch. E.g., the left branch of the tree in Figure 1 is closed because it contains the conflicting instances $Col(\mathbf{red}, \mathbf{d})^{\oplus}$ and $Col(\mathbf{red}, \mathbf{d})^{\ominus}$. We indicate a closed branch with the symbol $\times$. An MX-tree is closed if all its branches are closed. An *MX-proof for* $\langle T, \tilde{I}\rangle$ is a closed MX-tree for $\langle T, \tilde{I}\rangle$. The next theorem states the soundness and completeness of the MX-calculus.

Theorem 1. *There exists an MX-proof for $\langle T, \tilde{I}\rangle$ iff $\mathrm{MX}_T(\tilde{I})$ is unsatisfiable.*

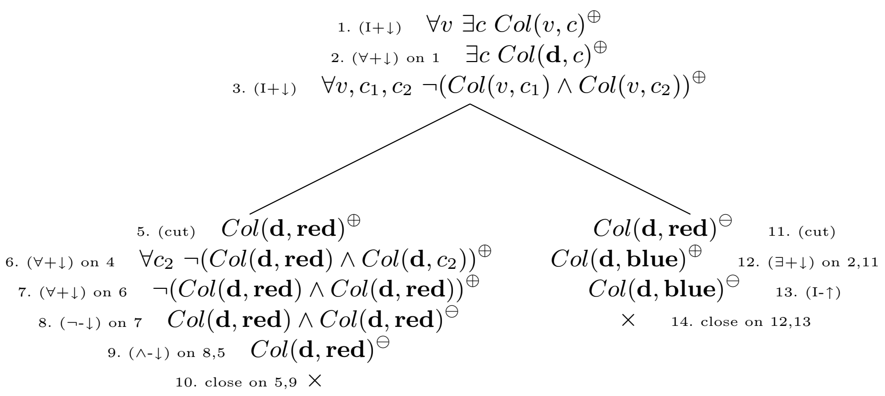

1. (I+↓) $\forall v\ \exists c\ Col(v,c)^{\oplus}$
2. (∀+↓) on 1 $\exists c\ Col(\mathbf{d},c)^{\oplus}$
3. (I+↓) $\forall v, c_1, c_2\ \neg(Col(v,c_1) \wedge Col(v,c_2))^{\oplus}$
5. (cut) $Col(\mathbf{d},\mathbf{red})^{\oplus}$
6. (∀+↓) on 4 $\forall c_2\ \neg(Col(\mathbf{d},\mathbf{red}) \wedge Col(\mathbf{d},c_2))^{\oplus}$
7. (∀+↓) on 6 $\neg(Col(\mathbf{d},\mathbf{red}) \wedge Col(\mathbf{d},\mathbf{red}))^{\oplus}$
8. (¬-↓) on 7 $Col(\mathbf{d},\mathbf{red}) \wedge Col(\mathbf{d},\mathbf{red})^{\ominus}$
9. (∧-↓) on 8,5 $Col(\mathbf{d},\mathbf{red})^{\ominus}$
10. close on 5,9 ×
$Col(\mathbf{d},\mathbf{red})^{\ominus}$ 11. (cut)
$Col(\mathbf{d},\mathbf{blue})^{\oplus}$ 12. (∃+↓) on 2,11
$Col(\mathbf{d},\mathbf{blue})^{\ominus}$ 13. (I-↑)
× 14. close on 12,13

Fig. 1. An MX-tree for Example 1

3.3 Saturated Branches

Although Theorem 1 guarantees the existence of an MX-proof for $\langle T, \tilde{I}\rangle$ if $\text{MX}_T(\tilde{I}) = \emptyset$, a naive implementation of the MX-calculus could never find such a proof. This is even the case if a same formula is never added twice to a branch. An implementation could, e.g., endlessly apply the cut rule for different instances. However, as we will show below, we can restrict the instances that may occur in an MX-tree in such a way that termination is guaranteed. Moreover, if such a restricted MX-tree $\mathcal{T}$ is not closed and cannot be extended, a model of $\text{MX}_T(\tilde{I})$ can easily be extracted from $\mathcal{T}$.

For a theory T over Σ, we call an instance a *T-instance* if it is either an instance of a subformula of T, or an atom of the form $P(\overline{d})$, $F(\overline{d}) = d$ or $d = d$. A rule of the form (6) is *T-restricted* if all $\mathcal{J}_i$ are signed T-instances. We call a branch B of an MX-tree *T-saturated* if for every T-restricted MX-rule of the form (6) that can be applied to B, at least one of the $\mathcal{J}_i$ already occurs in B.

Definition 5. *Let B be a non-closed branch of an MX-tree for $\langle T, \tilde{I}\rangle$. The* implicit structure of B *is the three-valued structure $\tilde{B}$ with the same domain as $\tilde{I}$ and defined by*

$$\tilde{B}(P(\overline{d})) = \begin{cases} \mathbf{t} & \textit{if } P(\overline{d})^{\oplus} \in B \\ \mathbf{f} & \textit{if } P(\overline{d})^{\ominus} \in B \\ \mathbf{u} & \textit{otherwise} \end{cases}$$

and $\tilde{B}(F(\overline{d})) = \{d' \mid F(\overline{d}) = d'^{\ominus} \notin B\}$ for every predicate P, function F and domain elements $\overline{d}, d'$.

Lemma 1. *Let B be a non-closed T-saturated branch, and $\tilde{B}$ its implicit structure. Then for any two-valued structure $M \geq_p \tilde{B}$, it holds that $M \models T$.*

According to this lemma, theorem 1 can be refined as follows.

Theorem 2. *There exists an MX-proof for $\langle T, \tilde{I} \rangle$ containing only signed T-instances iff* $\mathrm{MX}_T(\tilde{I})$ *is unsatisfiable.*

Since there are only finitely many signed T-instances, this theorem guarantees termination of an algorithm that constructs restricted MX-proofs. Moreover, if such an algorithm terminates with a non-closed tree $\mathcal{T}$ for $\langle T, \tilde{I} \rangle$, then the implicit structure of each of $\mathcal{T}$'s non-closed branches is more precise than $\tilde{I}$. According to Lemma 1, we can easily extract solutions for $\mathrm{MX}_T(\tilde{I})$ from such a branch. This result suggests the following procedure to solve the MX-search problem for input $\langle T, \tilde{I} \rangle$.

1. Let $\mathcal{T}$ be the empty tree.
2. As long as $\mathcal{T}$ contains a non-closed, non-saturated branch B, enlarge $\mathcal{T}$ by applying a T-restricted MX-rule to B.
3. If $\mathcal{T}$ is closed, return "unsatisfiable". Else, return a two-valued structure that is more precise than the implicit structure of one of the T-saturated branches of $\mathcal{T}$.

Observe that this solver algorithm meets the requirements mentioned in the introduction. Indeed, all MX-rules are simple, and since only T-restricted rules are used, all reasoning is performed on the original theory T.

We made a prototype native implementation of the solver algorithm where binary decision diagrams are used to represent large sets of instances in a compact way. Since no heuristics were implemented to guide the search, the implementation turns out to be a lot slower than other MX-solvers. On problems with bugs however, it is efficient enough to be useful in practice.

A non-native approach to construct MX-proofs consists of translating the trace of an efficient MX-solver $\mathcal{M}$ into an MX-proof. This approach would yield a system that is close in efficiency to $\mathcal{M}$.

4 Debugging

As mentioned in the introduction, we propose a debugging system for locating and explaining bugs that cause a solver to omit a number of expected models. In the extreme case, such a bug makes $\mathrm{MX}_T(\tilde{I})$ unsatisfiable. Basically, the approach to locate and explain a bug in an input $\langle T, \tilde{I} \rangle$ is as follows:

1. The user specifies a structure $\tilde{J} \geq_p \tilde{I}$ describing a class of expected models and such that $\mathrm{MX}_T(\tilde{J})$ is unsatisfiable.
2. Construct a T-restricted MX-proof for $\langle T, \tilde{J} \rangle$.
3. Explore the proof to find the reason for $\mathrm{MX}_T(\tilde{J})$ being unsatisfiable.

The first step serves three different purposes. From a technical point of view, it ensures that a proof can be constructed in the second step, since $\mathrm{MX}_T(\tilde{J}) = \emptyset$. If $\mathrm{MX}_T(\tilde{I})$ itself is unsatisfiable, one could take $\tilde{J}$ equal to $\tilde{I}$. From a user point of view, $\tilde{J}$ can describe a specific class of models that is missing. E.g., in the graph colouring problem, a user can specify that he expects a solution where node $\mathbf{d}$ is red by assigning $Col^{\tilde{J}}(\mathbf{d}, \mathbf{red}) = \mathbf{t}$. Observe that if the user supplies a structure $\tilde{J}$ such that $\mathrm{MX}_T(\tilde{J}) \neq \emptyset$, then instead of generating a proof, a model $M \in \mathrm{MX}_T(\tilde{J})$ can be returned to indicate that there exists a solution that is in the class of structures described by $\tilde{J}$. Finally, the more precise $\tilde{J}$ is, the more concise proofs can be constructed for $\mathrm{MX}_T(\tilde{J})$. Evidently, more concise proofs are more comprehensive.

We now elaborate on the third step of the debugging method, which is based on earlier work by Shapiro [18]. Since the proofs can be quite large, even for simple problems, it is necessary that only parts that are relevant for the user are shown. In particular, if a user understands that a certain instance $\varphi[\overline{d}]$ is necessarily true in all models in $\mathrm{MX}_T(\tilde{J})$, then it is not needed to show him how the truth $\varphi[\overline{d}]$ was actually derived. Also, all shown proof steps must be stated in natural language, so that they are comprehensible for a user who is not familiar with the MX-calculus.

Concretely, the method we propose is an interactive session where the system provides a set S of signed instances that were derived (i.e., occur in the proof) to the user. Initially, the user is told that his input yields a conflict, and $S = \{\varphi^{\oplus}, \varphi^{\ominus}\}$, where $\varphi^{\oplus}$ and $\varphi^{\ominus}$ are conflicting instances of one of the branches of the proof. The user can then ask for an instance $\mathcal{I} \in S$ the reason why it was derived. Depending on $\mathcal{I}$, there are three possibilities:

- If a propagation rule was used to add $\mathcal{I}$ to the proof, the system's reply consists of (in natural language) that rule and its premises. The premises are added to the set S.
- If an initialization rule was used, the system replies that $\mathcal{I}$ was provided as input.
- If the cut rule was used and $\mathcal{I}$ is of the form $\varphi[\overline{d}]^{\oplus}$, the system replies that there is a conflict if $\varphi[\overline{d}]$ is assumed to be false. Conflicting instances of a branch below $\varphi[\overline{d}]^{\ominus}$ are added to the set S. For instances $\mathcal{J}$ in S whose derivation depends on $\varphi[\overline{d}]^{\ominus}$, the system indicates that $\mathcal{J}$ is derived under the assumption that $\varphi[\overline{d}]^{\ominus}$ is false. Similarly if $\mathcal{I}$ is of the form $\varphi[\overline{d}]^{\ominus}$.

We illustrate the method on the graph colouring example.

Example 1 (Continued). Assume we want to detect the bug in theory T_1 and therefore explore the proof shown in Figure 1 for $\langle T_1, \tilde{I}_1\rangle$. Figure 2 shows a possible exploration this proof. By the time the user asks question (3), he has most probably detected the bug. Since the derivation of (12) only holds when (11) is assumed, this is explicitly stated. Observe that (13), although it occurs below (11), does not use (11) in its derivation.

Choices and Proof Size. If the cut rule is often applied to construct a proof, or a proof is very large, it can be the case that — even with the exploration method outlined above — a user loses the overview. However, for several reasons we expect that this can often be avoided in practice:

(**S** is the debugging system, **U** the user, the numbers refer to the nodes in the proof)

S: Your input implies that $Col(\mathbf{d}, \mathbf{red})$ must be true (5) and false (9) at the same time. $S = \{(5), (9)\}$.
U: (9) (i.e., the question: "Why must $Col(\mathbf{d}, \mathbf{red})$ be false?")
S: Because $Col(\mathbf{d}, \mathbf{red})$ must be true and $Col(\mathbf{d}, \mathbf{red}) \wedge Col(\mathbf{d}, \mathbf{red})$ must be false (8). $S = \{(5), (8)\}$.
U: (8)
S: Because $\neg(Col(\mathbf{d}, \mathbf{red}) \wedge Col(\mathbf{d}, \mathbf{red}))$ must be true (7). $S = \{(5), (7)\}$.
U: (7)
S: Because $\forall c_2 \; \neg(Col(\mathbf{d}, \mathbf{red}) \wedge Col(\mathbf{d}, c_2))$ must be true, and we can take $c_2 = \mathbf{red}$.
⋮
U: (3)
S: Because this is a sentence of the input theory. $S = \{(5)\}$.
U: (5)
S: If $Col(\mathbf{d}, \mathbf{red})$ is assumed to be false, then there is a conflict: $Col(\mathbf{d}, \mathbf{blue})$ must be true (12) and false (13). $S = \{(12)$ if (11) is assumed$, (13)\}$.
⋮

Fig. 2. Interactively exploring a proof

- The *small scope hypothesis* [7] claims that a bug typically occurs already in the context of structures $\tilde{I}$ with a small domain. For such a structure, a proof for $\langle T, \tilde{I} \rangle$ will be small too.
- A good solver based on the MX-calculus should minimize the use of the cut rule by only applying it when no other rule can be applied.
- If the description $\tilde{J}$ of the expected models is sufficiently precise, it is not needed to often apply the cut rule. In the extreme case where $\tilde{J}$ is two-valued, using the cut rule can be avoided altogether.

Nevertheless, in a worst case scenario, a bug is due to a combination of partially correct formulas and only shows up in large instances. Such bugs may be very hard to find and correct. This problem is inherent to debugging in all declarative languages.

5 Inductive Definitions

Although all NP problems can be cast as MX(FO) problems, the modelling is not always easy. In a finite context, inductive definitions such as the definition of *reachability* in a graph, can be encoded in FO, but this is far from straightforward. The logic FO(ID) extends FO with a native construct to represent definitions. Hence, this logic simplifies the modelling task. Also, MX(FO(ID)) solvers that natively support FO(ID) tend to be faster than systems that rely on a transformation to FO for problems involving recursion [13]. Modelling methodology in MX(FO(ID)) and ASP are very similar [12]. There exists a simple transformation from MX(FO(ID)) to ASP specifications [11] and vice versa [5].

5.1 Preliminaries

To facilitate the rest of the presentation, we assume from now on that a vocabulary contains no function symbols. We define the truth value of a formula φ in a three-valued interpretation $\tilde{I}$ by the standard Kleene semantics and denote it by $\tilde{I}(\varphi)$. For a truth value $\mathbf{v}$ and atom $P(\overline{d})$, we denote by $\tilde{I}[P(\overline{d})/\mathbf{v}]$ the interpretation that assigns $\mathbf{v}$ to $P(\overline{d})$ and corresponds to $\tilde{I}$ on all other symbols. This notation is extended to sets of atoms.

In FO(ID), a *definition* Δ is a finite set of rules of the form

$$\forall \overline{x}\ (P(\overline{x}) \leftarrow \varphi),$$

where P is a predicate and φ an FO formula. The free variables of φ should be among $\overline{x}$. $P(\overline{t})$ is called the *head* of the rule, φ the *body*. Predicates that occur in the head of a rule of Δ are called *defined* predicates of Δ. All other symbols are called *open* with respect to Δ. The set of open symbols of Δ is denoted by Open(Δ). The semantics of definitions is given by their well-founded model [22]. As argued in [2], the well-founded semantics correctly formalizes the semantics of monotone and non-monotone inductive definitions. We present this semantics according to [3]. Let Δ be a definition and $\tilde{I}$ a three-valued structure. A *well-founded induction for Δ above $\tilde{I}$* is a sequence $\langle \tilde{J}_\xi \rangle_{0 \leq \xi \leq \alpha}$ such that

1. $\tilde{J}_0$ assigns $P^{\tilde{J}_0}(\overline{d}) = \mathbf{u}$, if P is a defined predicate and corresponds to $\tilde{I}$ on the open symbols;
2. For each limit ordinal $\lambda \leq \alpha$, $\tilde{J}^\lambda = \text{lub}_{\leq_p}(\{\tilde{J}_\xi \mid \xi \leq \lambda\})$
3. For every ordinal ξ, $\tilde{J}_{\xi+1}$ relates to $\tilde{J}_\xi$ in one of the following ways:
   (a) $\tilde{J}_{\xi+1} = \tilde{J}_\xi[P(\overline{d})/\mathbf{t}]$ for some domain atom $P(\overline{d})$ such that $\tilde{J}_\xi(P(\overline{d})) = \mathbf{u}$ and for some rule $\forall \overline{x}\ (P(\overline{x}) \leftarrow \varphi)$ in Δ, $\tilde{J}_\xi[\overline{x}/\overline{d}](\varphi) = \mathbf{t}$.
   (b) $\tilde{J}_{\xi+1} = \tilde{J}_\xi[P(\overline{d})/\mathbf{f}]$ for some domain atom $P(\overline{d})$ such that $\tilde{J}_\xi(P(\overline{d})) = \mathbf{u}$ and for every rule $\forall \overline{x}\ (P(\overline{x}) \leftarrow \varphi)$ in Δ, $\tilde{J}_\xi[\overline{x}/\overline{d}](\varphi) = \mathbf{f}$.
   (c) $\tilde{J}_{\xi+1} = \tilde{J}_\xi[U/\mathbf{f}]$, where U is a set of domain atoms, such that for each $P(\overline{d}) \in U$, $\tilde{J}_\xi(P(\overline{d})) = \mathbf{u}$ and all $\forall \overline{x}\ (P(\overline{x}) \leftarrow \varphi)$ in Δ, $\tilde{J}_{\xi+1}[\overline{x}/\overline{d}](\varphi) = \mathbf{f}$.

Intuitively, (a) says that a domain atom $P(\overline{d})$ can be made true if there is a rule with $P(\overline{x})$ as head and body $\varphi[\overline{x}]$ such that $\varphi[\overline{d}]$ is already true. If, on the other hand, for all rules $P(\overline{x}) \leftarrow \varphi[\overline{d}]$, $\varphi[\overline{d}]$ is false, then (b) expresses that $P(\overline{d})$ can be made false. Finally, (c) explains that $P(\overline{d})$ can be made false if there is no possibility to make a corresponding body true, except by circular reasoning. The set U, commonly called an *unfounded set*, is a witness to this: making all atoms in U false also makes all corresponding bodies false.

A well-founded induction is called *terminal* if it cannot be extended anymore. The limit of a terminal well-founded induction is its last element. In [3], it is shown that each terminal well-founded induction for Δ above $\tilde{I}$ has the same limit, which corresponds to the well-founded model of Δ extending $\tilde{I}|_{\text{Open}(\Delta)}$, and is denoted by $\text{wfm}_\Delta(\tilde{I})$. The well-founded model is three-valued in general.

A two-valued interpretation I satisfies a definition Δ if $I = \text{wfm}_\Delta(I)$. An FO(ID) theory T is a set of FO sentences and definitions. I satisfies T if it satisfies all definitions and sentences in T.

5.2 Extending the MX-calculus to FO(ID)

To handle definitions in the MX-calculus, we extend it with two new classes of rules: *completion* and *unfounded set* rules. Rules of the former class handle case (a) and (b) of the well-founded inductions, the rule in the latter class handles case (c).

Completion Rules. If Δ is a definition in the FO(ID) theory T, P a defined predicate of Δ and $\forall \overline{x}\ P(\overline{x}) \leftarrow \varphi_1[\overline{x}], \ldots, \forall \overline{x}\ P(\overline{x}) \leftarrow \varphi_n[\overline{x}]$ all rules in Δ with head P, then the following are the four completion rules for $\langle T, \tilde{I} \rangle$.

$$\frac{\varphi_i[\overline{d}]^{\oplus}}{P(\overline{d})^{\oplus}} \qquad \frac{\varphi_1[\overline{d}]^{\ominus}, \ldots, \varphi_n[\overline{d}]^{\ominus}}{P(\overline{d})^{\ominus}} \qquad \frac{P(\overline{d})^{\ominus}}{\varphi_i[\overline{d}]^{\ominus}}$$

$$\frac{P(\overline{d})^{\oplus}, \varphi_1[\overline{d}]^{\ominus}, \ldots, \varphi_{i-1}[\overline{d}]^{\ominus}, \varphi_{i+1}[\overline{d}]^{\ominus}, \ldots, \varphi_n[\overline{d}]^{\ominus}}{\varphi_i[\overline{d}]^{\oplus}}$$

Unfounded Set Rule. Let $\tilde{K}$ be a three-valued structure with the same domain as $\tilde{I}$, $\{Q_1(\overline{d}_1), \ldots, Q_n(\overline{d}_n)\}$ the set of all domain atoms that are true in $\tilde{K}$ and $\{Q_{n+1}(\overline{d}_{n+1}), \ldots, Q_m(\overline{d}_m)\}$ the set of all domain atoms that are false in $\tilde{K}$. Let Δ be a definition of T and U a set of domain atoms, defined with respect to Δ and unknown in $\tilde{K}$. If for $R(\overline{d}) \in U$ and any rule $\forall \overline{x}\ R(\overline{x}) \leftarrow \varphi[\overline{x}]$ in Δ, $\tilde{K}[U/\mathbf{f}](\varphi[\overline{d}]) = \mathbf{f}$, then

$$\frac{Q_1(\overline{d}_1)^{\oplus}, \ldots, Q_n(\overline{d}_n)^{\oplus}, Q_{n+1}(\overline{d}_{n+1})^{\ominus}, \ldots, Q_m(\overline{d}_m)^{\ominus}}{P(\overline{d})^{\ominus}}$$

where $P(\overline{d}) \in U$, is an unfounded set rule for $\langle T, \tilde{I} \rangle$.

Soundness and Completeness. The MX$^{\mathrm{ID}}$-calculus for FO(ID) is the MX-calculus for FO, extended with the completion and unfounded set rules. The results of sections 3.2 and 3.3 carry over to this extension of the MX-calculus. In particular, we have the following theorem.

Theorem 3. *There exists an MX^{ID}-proof for $\langle T, \tilde{I} \rangle$ containing only signed T-instances iff* $\mathrm{MX}_T(\tilde{I})$ *is unsatisfiable.*

5.3 Debugging for FO(ID)

The debugging method for MX(FO) can be extended to a debugging method MX(FO(ID)), by constructing MX$^{\mathrm{ID}}$-proofs instead of MX-proofs. There is no problem communicating the use of a completion rule to the user. For the unfounded set rule however, this is not entirely straightforward. A possible explanation to a user who asks why $P(\overline{d})$ is false, where this atom was derived by an application of the unfounded set rule is (using the same notations as in the section where we introduced the unfounded set rule)

Since $Q_1(\overline{d}_1), \ldots, Q_n(\overline{d}_n)$ must be true and $Q_{n+1}(\overline{d}_{n+1}), \ldots, Q_m(\overline{d}_m)$ must be false, $P(\overline{d})$ belongs to a set U of defined atoms of Δ that can only become true because of circular reasoning.

If the user asks why the atoms in U can only become true because of circular reasoning, the system could let him explore a proof for the inconsistency of $\left\langle \bigvee_{\varphi[\overline{d}] \in B} \varphi[\overline{d}], \tilde{K}[U/\mathbf{f}] \right\rangle$, where the $B = \{\varphi[\overline{d}] \mid \forall \overline{x}\ R(\overline{x}) \leftarrow \varphi[\overline{x}] \in \Delta \text{ and } R[\overline{d}] \in U\}$. I.e., a proof for the fact that $\tilde{K}[U/\mathbf{f}](\varphi[\overline{d}])$ is false for any $\varphi[\overline{d}] \in B$.

6 Related Work

Much work in debugging for declarative programming systems focusses on a specific procedural semantics (e.g., on a particular execution model for Prolog). See, e.g., [8,9]. The trace of a solver is then a sequence of steps according to the procedural semantics, rather than a formal proof. Details are given on how to obtain and store the trace efficiently. [21] addresses these issues in a context where the small scope hypothesis does not hold.

Existing approaches for debugging an input for a model generator can be divided into two classes: the approaches that aim at *locating* a bug, and those that aim at *explaining* derivations made by a model generator. Clearly, these classes are complementary. A system of the first class can extract a part of the theory where the bug is located. Then, a system of the second class can explain why this part contains a bug. As far as we know, our debugging method is the first one for (extended) FO model generation that belongs to the second class.

The ALLOY system [1] is a model expansion system for a syntactic variant of FO. In [19], a debugging method for overconstrained (hence unsatisfiable) instances was presented. It consists of extracting an *unsatisfiable core*, i.e., a small inconsistent subset, from the theory and presenting it to the user. Hence, it belongs to the first class. If the unsatisfiable core is small, the user can quickly locate the bug. If it is somewhat larger, it can still be difficult to detect the bug. In this case, our system could be used with the unsatisfiable core as input to further guide the search for a bug. This has the side benefit of speeding up our approach: a proof of inconsistency for the small unsatisfiable core is smaller and can be constructed faster than a proof for the inconsistency of the whole theory.

In the context of ASP, several approaches to debugging have been presented. A recent overview can be found in [6]. Most ASP debugging methods belong to the first class mentioned above. E.g., the method described in [6] returns for an input $\langle T, \tilde{I} \rangle$ a two-valued interpretation $M \geq_p \tilde{I}$ and a number of constraints, rules and/or unfounded sets that are violated by M. The method of [20] returns a minimal set of rules such that the theory without these rules is satisfiable. An advantage of these two methods is that they can be implemented in ASP itself.

A debugging method of the second class for ASP was presented in [17]. It allows a user to interrupt the computation of an ASP solver and to ask an explanation for any atom that is not unknown at that moment. Explanations are given in the form of graphs, called *justifications*.

7 Conclusions and Future Work

In this paper, we presented the MX-calculus as a proof system for FO(ID). We showed how to explore proofs in the MX-calculus to debug input for MX(FO(ID)) model generators. Future work consists of extending the MX-calculus to support other extensions of FO, such as aggregates, and building a debugging system that can handle the full input language of model generators such as IDP and MXG.

References

1. Chang, F.: Alloy analyzer 4.0 (2007), `http://alloy.mit.edu/alloy4/`
2. Denecker, M., Ternovska, E.: A logic of nonmonotone inductive definitions. ACM Trans. Comput. Log. 9(2) (2008)
3. Denecker, M., Vennekens, J.: Well-founded semantics and the algebraic theory of non-monotone inductive definitions. In: Baral, C., Brewka, G., Schlipf, J. (eds.) LPNMR 2007. LNCS, vol. 4483, pp. 84–96. Springer, Heidelberg (2007)
4. Ducassé, M.: Opium: An extendable trace analyzer for prolog. J. Log. Program. 39(1-3), 177–223 (1999)
5. East, D., Truszczynski, M.: Predicate-calculus-based logics for modeling and solving search problems. ACM Trans. Comput. Log. 7(1), 38–83 (2006)
6. Gebser, M., Pührer, J., Schaub, T., Tompits, H.: A meta-programming technique for debugging answer-set programs. In: Fox, D., Gomes, C.P. (eds.) AAAI, pp. 448–453. AAAI Press, Menlo Park (2008)
7. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. The MIT Press, Cambridge (2006)
8. Jahier, E., Ducass, M., Ridoux, O.: Specifying prolog trace models with a continuation semantics. In: Lau, K.-K. (ed.) LOPSTR 2000. LNCS, vol. 2042, p. 165. Springer, Heidelberg (2001)
9. Langevine, L., Ducassé, M., Deransart, P.: A propagation tracer for gnu-prolog: From formal definition to efficient implementation. In: Palamidessi, C. (ed.) ICLP 2003. LNCS, vol. 2916, pp. 269–283. Springer, Heidelberg (2003)
10. Mallet, S., Ducassé, M.: Generating deductive database explanations. In: ICLP, pp. 154–168 (1999)
11. Mariën, M., Gilis, D., Denecker, M.: On the relation between ID-Logic and Answer Set Programming. In: Alferes, J.J., Leite, J.A. (eds.) JELIA 2004. LNCS, vol. 3229, pp. 108–120. Springer, Heidelberg (2004)
12. Mariën, M., Wittocx, J., Denecker, M.: The IDP framework for declarative problem solving. In: LaSh 2006, pp. 19–34 (2006)
13. Mariën, M., Wittocx, J., Denecker, M., Maurice, B.: SAT(ID): Satisfiability of propositional logic extended with inductive definitions. In: Kleine Büning, H., Zhao, X. (eds.) SAT 2008. LNCS, vol. 4996, pp. 211–224. Springer, Heidelberg (2008)
14. Meier, M.: Debugging constraint programs. In: Montanari, U., Rossi, F. (eds.) CP 1995. LNCS, vol. 976, pp. 204–221. Springer, Heidelberg (1995)
15. Mitchell, D., Ternovska, E.: A framework for representing and solving NP search problems. In: AAAI 2005, pp. 430–435. AAAI Press/MIT Press (2005)
16. Nilsson, H.: Tracing piece by piece: Affordable debugging for lazy functional languages. In: ICFP, pp. 36–47 (1999)
17. Pontelli, E., Son, T.C.: Justifications for logic programs under answer set semantics. In: Etalle, S., Truszczyński, M. (eds.) ICLP 2006. LNCS, vol. 4079, pp. 196–210. Springer, Heidelberg (2006)

18. Shapiro, E.Y.: Algorithmic Program Debugging. MIT Press, Cambridge (1983)
19. Shlyakhter, I., Seater, R., Jackson, D., Sridharan, M., Taghdiri, M.: Debugging overconstrained declarative models using unsatisfiable cores. In: ASE-18, pp. 94–105 (2003)
20. Syrjänen, T.: Debugging inconsistent answer set programs. In: NMR 2006, Lake District, UK, May 2006, pp. 77–84 (2006)
21. Tronçon, R., Janssens, G.: A delta debugger for ilp query execution. In: CoRR, abs/cs/0701105 (2007)
22. Van Gelder, A., Ross, K.A., Schlipf, J.S.: The well-founded semantics for general logic programs. Journal of the ACM 38(3), 620–650 (1991)

Metabolic Network Expansion with Answer Set Programming

Torsten Schaub* and Sven Thiele

Universität Potsdam, Institut für Informatik, August-Bebel-Str. 89, D-14482 Potsdam, Germany

Abstract. We propose a qualitative approach to elaborating the biosynthetic capacities of metabolic networks. In fact, large-scale metabolic networks as well as measured datasets suffer from substantial incompleteness. Moreover, traditional formal approaches to biosynthesis require kinetic information, which is rarely available. Our approach builds upon a formal method for analyzing large-scale metabolic networks. Mapping its principles into Answer Set Programming (ASP) allows us to address various biologically relevant problems. In particular, our approach benefits from the intrinsic incompleteness-tolerating capacities of ASP. Our approach is endorsed by recent complexity results, showing that the reconstruction of metabolic networks and related problems are NP-hard.

1 Introduction

The availability of high-throughput methods in molecular biology has resulted in a rapid growth of biological knowledge, gathered in web databases such as KEGG (http://www.genome.jp/kegg) or MetaCyc (http://metacyc.org). Of particular interest are biosynthetic capacities of metabolic networks in view of the design of bioprocesses. However, large-scale metabolic networks as well as measured datasets suffer from substantial incompleteness. Many networks are only partially defined and only few metabolites can be identified without ambiguity. Moreover, traditional formal approaches to biosynthesis (cf. [1,2,3,4,5]) require kinetic information, which is rarely available.

We address this problem and propose a qualitative approach based on Answer Set Programming (ASP;[6]). This approach benefits from the intrinsic incompleteness-tolerating capacities of ASP and allows us to take advantage of its rich modelling language and highly efficient implementations. Our approach is endorsed by recent complexity results, showing that the reconstruction of metabolic networks and related problems are NP-hard [7,8].

Our approach builds upon a formal method for analyzing large-scale metabolic networks developed in [9,10]. The basic idea is that a reaction operates only if its reactants are either available as nutrients or can be provided by other metabolic reactions. Starting from some nutrients, referred to as *seeds*, this allows for expanding a metabolic network by successively adding operable reactions and their products. The set of metabolites in the resulting network is called the *scope* of the seeds and represents all metabolites that can principally be synthesized from the seeds by the analyzed metabolic network.

* Affiliated with Simon Fraser University, Canada, and Griffith University, Australia.

Mapping the principles of this approach into ASP allows us to address various biologically relevant problems. A primary problem deals with the completion of genome-scale metabolic networks. When building a metabolic network, as for the recently sequenced green alga *Chlamydomonas reinhardtii* (`http://www.goforsys.de`), the initial core draft is done by appeal to genomic information. Then, experimental data, in particular, measured metabolites, are taken to define the functionality of the overall network. The above methodology can then be used to check whether a drafted network provides the synthesis routes to comply with the required functionality. If this fails, the draft network can be completed by importing reactions from metabolic reference network stemming from other organisms until the obtained network provides the measured functionality (cf. [11]). Another important problem concerns the determination of seed compounds needed for the synthesis of certain other compounds. As demonstrated in [12], solving this problem is important for indicating (minimal) nutritional requirements for sustaining maintenance or growth of an organism.

Both problems have a combinatorial nature and thus give rise to a multitude of solutions. We address this problem by taking advantage of the various reasoning modes provided by ASP. On the one hand, we use ASP's optimization techniques for finding cardinality or subset minimal solutions, respectively. On the other hand, we exploit consequence aggregation for finding metabolites common to all (optimal) solutions or at least one of them, respectively. Moreover, the aforementioned problems are often subject to additional constraint, aiming at the avoidance of side products or producing target products by staying clear from certain seeds, respectively. Finally, the elaboration tolerance of ASP greatly supports the process of drafting metabolic networks involving continuous validation and increasing functionalities stemming from measured data.

2 Background

Biological problem definition. Following [8], a *metabolic network* is commonly represented as a directed bipartite graph $G = (R \cup M, E)$, where R and M are sets of nodes standing for *reactions* and *metabolites*, respectively. Given a such metabolic network G, we sometimes refer to its components by $R(G)$, $M(G)$, and $E(G)$. Whenever $(m, r) \in E$ for $m \in M$ and $r \in R$, the metabolite m is called a *reactant* of reaction *r*; for $(r, m) \in E$, metabolite m is called a *product* of r. More formally, for $(R \cup M, E)$ and $r \in R$ define $\mathit{reac}(r) = \{m \in M \mid (m, r) \in E\}$ and $\mathit{prod}(r) = \{m \in M \mid (r, m) \in E\}$.

The aforementioned biological concept of a *scope* can be expressed in terms of reachability. Given a metabolic network $(R \cup M, E)$ and a set $M' \subseteq M$ of *seed* metabolites, a reaction $r \in R$ is *reachable* from M', if $\mathit{reac}(r) \subseteq M'$, that is, if all its reactants are reachable. Moreover, a metabolite $m \in M$ is *reachable* from M', either if $m \in M'$ or if $m \in \mathit{prod}(r)$ for some reaction $r \in R$ being *reachable* from M'. Finally, the *scope* of M', written $\Sigma_{(R \cup M, E)}(M')$ or simply $\Sigma(M')$, is the closure of M' under reachability from M'. Note that the scope of a set of metabolites can be computed in polynomial time.

Now, we can make precise the aforementioned biological problems. In the *metabolic network completion*, we are given a metabolic network $(R \cup M, E)$ along with two sets $S, T \subseteq M$ of (seed and target) metabolites, and a reference network $(R' \cup M', E')$. The goal is to find a set of reactions $R'' \subseteq R' \setminus R$ such that $T \subseteq \Sigma_G(S)$ where

$$\begin{aligned} G &= ((R \cup R'') \cup (M \cup M''), E \cup E'')\ , \\ M'' &= \{m \in M' \mid r \in R'', m \in \mathit{reac}(r) \cup \mathit{prod}(r)\}\ ,\ \text{and} \\ E'' &= \{(m,r) \in E' \mid r \in R'', m \in \mathit{reac}(r)\} \cup \{(r,m) \in E' \mid r \in R'', m \in \mathit{prod}(r)\}\ . \end{aligned}$$

We call R'' the *completion* of $(R \cup M, E)$ from $(R' \cup M', E')$ wrt (S, T). Two optimization variants of this problem are obtained by finding a cardinality or subset minimal set of reactions. Further refinements may also optimize on the distance between seeds and targets or minimize forbidden side products.

Three variants of the *inverse scope problem* can be distinguished [8]. In the basic one, we are given a metabolic network $(R \cup M, E)$ and a set $T \subseteq M$ of (target) metabolites. The goal is to find a set of (seed) metabolites $S \subseteq M$ such that $T \subseteq \Sigma(S)$. The two optimization variants of this problem aim at finding a cardinality or subset minimal solution. The second problem restricts the domain of the available seed metabolites. In addition to $(R \cup M, E)$ and $T \subseteq M$, we are given a set of (forbidden) metabolites $F \subseteq M$. Then, the goal is to find a set of (seed) metabolites $S \subseteq (M \setminus F)$ such that $T \subseteq \Sigma(S)$. Apart from optimizing the required seed metabolites, one may also minimize undesired metabolites rather then excluding them. The third problem adds an additional constraint on the avoidance of side products. In addition to $(R \cup M, E)$ and $T, F \subseteq M$, we are given another set of (forbidden) metabolites $E \subseteq M$. Then, the goal is to find a set of (seed) metabolites $S \subseteq (M \setminus F)$ such that $T \subseteq \Sigma(S)$ and $\Sigma(S) \cap E = \emptyset$. As above, the optimization variants can also take side products into account.

Answer Set Programming. We refer the reader to [6] for a formal introduction to ASP and concentrate in what follows on aspects relevant to our application. A *logic program* is a finite set of *rules* of the form

$$a \leftarrow b_1, \ldots, b_m, \mathit{not}\ c_{m+1}, \ldots, \mathit{not}\ c_n\ , \tag{1}$$

where a, b_i, c_j are *atoms* for $0 < i \leq m < j \leq n$. A *literal* is an atom a or its (default) negation $\mathit{not}\ a$. A rule r as in (1) is called a *fact*, if $l = n = 1$, and an *integrity constraint*, if $l = 0$. We denote predicate and constant symbols by lowercase letters and variables by uppercase letters. A logic program with variables is regarded as the set of all its ground-instantiated rules. Moreover, we take advantage of *choice rules* and *conditional literals* [13]; both of which can be regarded as macros. In a choice rule, the head a in (1) is replaced by a set $\{a_1, \ldots, a_l\}$; it allows us to derive any subset provided the rule's body is satisfied. A conditional literal is of form $a : b$ where a and b are literals (containing common variables); informally, it stands for the sequence of all instantiations of a obtained by restricting the substitution of variables common to a and b to those of b (cf. [13] for details). For instance, given $m(1)$, $m(2)$, and $r(a)$, the choice rule $\{p(R, M) : m(M)\} \leftarrow r(R)$ stands for $\{p(a,1), p(a,2)\} \leftarrow r(a)$.

The answer sets of a program P are models of P satisfying a certain stability criterion (cf. [6] for details). An answer set is represented by the set of atoms that are true in it. Apart from testing the existence of an answer set of a program or enumerating all its answers, the following reasoning modes are supported. For this, define $AS(P)$ as the set of all answer sets of Program P. Then, the cautious and brave consequence of P are defined as $\bigcap_{X \in AS(P)} X$ and $\bigcup_{X \in AS(P)} X$. Notably, both sets are computable through linear many computations of one answer set, rather than computing possibly exponential number of answer sets in $AS(P)$. Another mode of interest is solution projection [14], which computes only the projections of all answer sets on a set P of atoms, that is, $\{X \cap P \mid X \in AS(P)\}$, thereby greatly reducing computational efforts.

Cardinality based optimization is provided in ASP through minimize (or maximize) statements of the form

$$\mathit{minimize}\{b_1 = w_1, \ldots, b_m = w_m, \mathit{not}\ c_{m+1} = w_{m+1}, \ldots, \mathit{not}\ c_n = w_n\}$$

enforcing that only answer sets with minimum value of $\sum_{b_i \in X, 1 \leq i \leq m} w_i + \sum_{c_j \notin X, m+1 \leq j \leq n} w_j$ are computed, where $w_1, \ldots, w_n$ are integers. There can be several minimize and/or maximize statements which order the stable models lexicographically. Subset based minimization and/or maximization is more complex and left for future work.

3 Logic Program Representations

3.1 Metabolic Network Completion

We start by representing a metabolic network G_n as a set of facts.

$$\begin{aligned} \mathcal{G}(G_n) = \quad & \{\mathit{reaction}(r, n) \mid r \in R(G_n)\} \\ & \cup \{\mathit{reactant}(m, r) \mid r \in R(G_n), m \in \mathit{reac}(r)\} \\ & \cup \{\mathit{product}(m, r) \mid r \in R(G_n), m \in \mathit{prod}(r)\} \end{aligned}$$

While our draft network provides an incomplete biological model, the seed and target metabolites are obtained from experimental data. The seed metabolites are provided as nutrients in an experiment, the target metabolites are measured as its final outcome. A metabolic draft network G_d along with two sets $S, T \subseteq M$ of seed and target metabolites, and a reference network G_r results in the following set of facts, $\mathcal{C}(G_d, G_r, S, T)$.

$$\mathcal{G}(G_d) \cup \mathcal{G}(G_r) \cup \{\mathit{draft}(d)\} \cup \{\mathit{seed}(s) \mid s \in S\} \cup \{\mathit{target}(t) \mid t \in T\} \qquad (2)$$

The current draft network is identified by the fact $\mathit{draft}(d)$.

Draft Scope. The scope of the seed metabolites in the draft network G_d can be determined by the following rules.

$$\begin{aligned} \mathit{dscope}(M) & \leftarrow \mathit{seed}(M) \\ \mathit{dscope}(M) & \leftarrow \mathit{product}(M, R), \mathit{reaction}(R, N), \mathit{draft}(N), \\ & \qquad \mathit{dscope}(M') : \mathit{reactant}(M', R) \end{aligned} \qquad (3)$$

The first rule declares all seed metabolites $M \in S$ as producible. The second rule defines recursively that a product M of a reaction R is producible, whenever all reactants M' of R are available. Together with the encoding of G_d and S in (2), the set of rules in (3) results in a single answer set X such that $dscope(m) \in X$ iff $m \in \Sigma_{G_d}(S)$ for $m \in M(G_d)$.

Potential Scope. While drafting a metabolic network of an organism biologists are regularly confronted with experiments that show that a certain metabolite can be measured, although it is not producible by the current draft network. To this end, they incorporate metabolic reactions known from metabolic networks of other organisms.

In analogy to the rules in (3), the (potential) scope of the seed metabolites in the draft network G_d augmented by the reference network G_r can be determined as follows.

$$\begin{aligned} pscope(M) &\leftarrow seed(M) \\ pscope(M) &\leftarrow product(M,R), reaction(R,N), \\ &\qquad pscope(M') : reactant(M',R) \end{aligned} \tag{4}$$

Note that dropping the qualification $draft(N)$ from (3) makes us use all available reactions. As before, given the encoding in (2), the set of rules in (4) induces a single answer set X such that $pscope(m) \in X$ iff $m \in \Sigma_{G_d \cup G_r}(S)$ for $m \in M$ where $G_d \cup G_r$ stands for the pairwise union of G_d and G_r.

While the scope of the draft network in (3) gives a lower limit on the metabolites producible from the seeds by the draft network, the potential scope obtained from the augmented network in (4) constitutes an upper limit. Note that targets outside the potential scope cannot be explained.

Metabolic Network Completion. The goal of metabolic network completion is to extend the draft network with reactions from the reference network, so that the target metabolites can be synthesized by the augmented network from the seeds. The reactions of interest belong to the reference network but not the draft network. The following choice rule captures all candidate reactions.

$$\{xreaction(R) : not\ reaction(R,N) : draft(N)\} \tag{5}$$

The condition $not\ reaction(R,N) : draft(N)$ guarantees that all chosen reactions belong to $R(G_r) \setminus R(G_d)$. In fact, the encoding in (2) and the choice rule in (5) give a set of answer sets being in a one-to-one correspondence to the subsets of $R(G_r) \setminus R(G_d)$.

The (extended) scope of the seed metabolites in the draft network G_d extended by reactions from G_r is defined as follows.

$$\begin{aligned} xscope(M) &\leftarrow seed(M) \\ xscope(M) &\leftarrow product(M,R), reaction(R,N), draft(N), \\ &\qquad xscope(M') : reactant(M',R) \\ xscope(M) &\leftarrow product(M,R), xreaction(R), \\ &\qquad xscope(M') : reactant(M',R) \end{aligned} \tag{6}$$

Finally, we have to make sure that an extended scope is able to produce all target metabolites. This is addressed by the following integrity constraint.[1]

$$\leftarrow target(M), not\ xscope(M) \tag{7}$$

[1] In practice, this constraint is extended by $pscope(M)$ to ignore non-producible targets.

Given the above rules, each of its answer set corresponds to a completion of the draft network and vice versa.

Proposition 1. *Let G_d and G_r be metabolic networks and let S and T be sets of metabolites.*

If X is an answer set of logic program[2] $\mathcal{C}(G_d, G_r, S, T) \cup \{(5), (6), (7)\}$*, then* $\{r \mid xreaction(r) \in X\}$ *is a completion of G_d from G_r wrt (S, T) and vice versa.*

Refined Metabolic Network Completion. Although the above encoding is formally adequate, it suffers from too many uninteresting completions that makes it fail to scale on large metabolic networks comprising several thousand metabolites. We address this problem by some refinements reducing the set of candidate reactions.

At first, we restrict the choice in (5) to "interesting" reactions.

$$\leftarrow xreaction(R), not\ ireaction(R) \tag{8}$$

The qualification expressed by $ireaction(R)$ requires that a reaction of interest must lead to some target metabolites.

$$\begin{aligned} ireaction(R) &\leftarrow interesting(M), \\ &\qquad product(M, R), reaction(R, N) \\ interesting(M) &\leftarrow target(M), not\ dscope(M) \\ interesting(M) &\leftarrow reactant(M, R), ireaction(R), \\ &\qquad not\ dscope(M) \end{aligned} \tag{9}$$

With the first rule we declare a reaction as interesting if it produces interesting metabolites. The second rule defines all target metabolites that cannot be produced by the draft network as interesting, and the third rule states that metabolites needed by interesting reactions and not producible by the draft network are interesting. This concept provides a significant reduction of the set of candidate reactions in view of the given target metabolites.

Second, we further restrict the choice in (5) to "operable" reactions.

$$\begin{aligned} &\leftarrow xreaction(R), not\ oreaction(R) \\ oreaction(R) &\leftarrow xscope(M) : reactant(M', R), \\ &\qquad reaction(R, N), not\ draft(N) \end{aligned} \tag{10}$$

The integrity constraint enforces that each extending reaction is operable, that is, satisfies $oreaction(R)$. The following rule defines a (candidate) reaction as operable, if all its reactants are producible by the current network extension.

The next result shows that the above refinements preserve soundness.

Proposition 2. *Let G_d and G_r be metabolic networks and let S and T be sets of metabolites.*

If X is an answer set of $\mathcal{C}(G_d, G_r, S, T) \cup \{(5), (6), (7), (8), (9), (10)\}$*, then* $\{r \mid xreaction(r) \in X\}$ *is a completion of G_d from G_r wrt (S, T).*

[2] Recall that a rule with variables stands for the set of all its ground instantiations.

Optimal Completions. A further natural way to reduce the number of solutions is to concentrate on network completions containing the fewest number of reactions. In ASP, this can be accomplished by the following minimize statement.

$$minimize\ \{xreaction(R) : ireaction(R) : not\ reaction(R, N)\} \tag{11}$$

Interestingly, our refinements are satisfied by such minimal completions, so that we get a soundness and completeness result under optimization.

Proposition 3. *Let G_d and G_r be metabolic networks and let S and T be sets of metabolites.*

If X is an answer set of $\mathcal{C}(G_d, G_r, S, T) \cup \{(5), (6), (7), (8), (9), (10)\} \cup \{(11)\}$, then $\{r \mid xreaction(r) \in X\}$ is a minimum completion of G_d from G_r wrt (S, T) and vice versa.

Sometimes reactions can be associated with confidence levels, for instance, obtained from the proximity of their host organism to the organism addressed by the draft network. This allows us to prefer among the minimum completions those composed of reactions with higher confidence levels; this is accomplished by adding the following statement.

$$maximize\{xreaction(R) = L : ireaction(R) : not\ reaction(R, N) : confidence(R, L)\}$$

Reasoning Modes. Given the above ensemble of rules, the reasoning modes of ASP solvers allow us to answer a variety of additional biologically relevant questions. What target metabolites are producible by the draft network? What new metabolites can be produced by adding reactions from other pathways? What is the minimal number of reactions that must be added to explain a target metabolite? What are the minimum or minimal extended scopes? Which reactions belong to all extended scopes, or even all minimum extended scopes? The latter are accomplished by a combination of optimization and cautious reasoning. We return to these question in Section 4 and show how they are realized. The next section also shows how certain seeds or side-products can either be avoided or minimized.

3.2 Inverse Scope Problem

Given a metabolic network and a set of target metabolites, we are interested in sets of seed metabolites that allow for producing the target metabolites from the network.

Basic Setting. Reactions, targets, and seeds are represented as in Section 3. That is, given a network G_n and sets S, T of metabolites, the inverse scope problem is based on the following set of facts.

$$\mathcal{I}(G_n, S, T) = \mathcal{G}(G_n) \cup \{seed(s) \mid s \in S\} \cup \{target(t) \mid t \in T\} \tag{12}$$

By appeal to the encoding of the basic scope in (3), we can then express our task similar to the completion problem by exchanging the roles of reactions and seed metabolites.

$$\{seed(M) : not\ target(M)\} \tag{13}$$

$$\leftarrow target(M), not\ dscope(M) \tag{14}$$

Similar to (5), the choice construct in (13) captures the seed candidates, while the integrity constraint in (14) makes sure that all target metabolites can by synthesized from the seeds chosen in (13).

The next proposition shows that our encoding is sound and complete.

Proposition 4. *Let G_n be a metabolic network and let S and T be sets of metabolites.*

If X is an answer set of logic program $\mathcal{I}(G_n, S, T) \cup \{(3), (13), (14)\} \cup \{draft(n)\}$, then $T \subseteq \Sigma(\{m \mid seed(m) \in X\})$ and vice versa.

The fact $draft(n)$ is merely added for compatibility with (3).

Refined Setting. As above, some refinements lend themselves for reducing the putative seed metabolites.

$$\leftarrow seed(M), not\ imetabolite(M) \tag{15}$$

A metabolite of interest, viz $imetabolite(M)$, must lead to at least one target metabolite.

$$\begin{aligned} imetabolite(M) &\leftarrow target(M) \\ imetabolite(M) &\leftarrow reactant(M,R), ireaction(R) \\ ireaction(R) &\leftarrow imetabolite(M), product(M,R), reaction(R,N) \end{aligned} \tag{16}$$

The first rule defines target metabolites as interesting. The second one extends this to metabolites being reactants of interesting reactions. Similar to (9), the last rule states that interesting reactions are those that produce interesting metabolites.

Although the last refinement eliminates (uninteresting) solutions, it preserves minimum ones. Hence, cardinality minimum solutions to the inverse seed problem are obtained by simply adding the following optimization statement.

$$minimize\{seed(M) : not\ target(M)\}$$

Avoiding Side or Seed Metabolites. The elaboration biosynthetic capacities of often subject to further restrictions, for instance, avoiding seed metabolites or certain side products. This has led to the definition of the two variants of the inverse scope problem defined in Section 2.

Both problems are easily addressed in ASP, once a metabolite, m, is declared as being forbidden, viz. $forbidden(m)$:

$$\begin{aligned} &\leftarrow seed(M), forbidden(M) \\ &\leftarrow dscope(M), forbidden(M) \end{aligned}$$

While the first constraint eliminates forbidden metabolites from the seeds, the second rules out unwanted side products.

The complete exclusion of certain metabolites is sometimes to restrictive. To this end, one may replace one or both of the previous integrity constraint by appropriate minimization statements:

$$\begin{aligned} minimize\{seed(M) : forbidden(M)\} \\ minimize\{dscope(M) : forbidden(M)\} \end{aligned}$$

Recall that the order of the two statements determines their precedence.

Reasoning Modes. The inverse scope problem usually leads to numerous solutions. Cautious reasoning allows us to compute the ultimately essential seeds belonging to all solutions. Also, brave reasoning is of interest because often solutions are similar, so that the union of all seeds in a solution form a pool of potentially relevant nutrients. Finally, in view of the numerous, often unrelated combinatorial sources, an important role is played by projective solution enumeration for eliminating redundant solutions.

4 Experiments

For validating our approach, we investigate the metabolic network of *Escherichia coli* (E.coli). This choice is motivated by the fact that E.coli is a well studied organism, whose metabolic network is of moderate size, consisting of 3645 reactions and 1556 metabolites. Our experiments consider furthermore 94 seed metabolites and 28 target metabolites. The targets and seeds were chosen by our biological partners in view of the fact that E.coli is able to grow when glucose is the only carbon source. Hence, its metabolic network must be able to synthesize all necessary precursors for high-level processes, from glucose and inorganic material [15]. That is why the targets contain all 20 amino acids, the nucleotide phosphates ATP, CTP, GTP and UTP as well as the deoxy forms dATP, dGTP, dUTP and dTTP; and the seeds are only glucose and inorganic metabolites. In fact, all considered targets could be produced by the original E.coli network. This setup allows us to control and vary our experiments by producing draft networks through eliminating reactions from E.coli's original network.

All experiments[3] were run with ASP grounder *gringo* (2.0.2) and ASP solver *clasp* (1.2.0) on a Linux PC with a Core2DuoE6400 processor and 2GB memory. The computation time was limited to 600 seconds, timeouts are shown throughout as "-".

4.1 Metabolic Network Completion

For our experiments on network completion, biologist provided us with draft networks. The draft networks have been created with biological background knowledge, by removing 50, 100 and 200 reactions from the original E.coli network. Also, derived reactions have been removed by the biologists. This means, for example, that for reversible reactions also the inverse reactions were removed, and for reactions that are generalizations, all subsumed special cases were removed as well. The resulting networks failed to produce 7, 10, and 20 targets, respectively. As reference network, we have chosen the entire MetaCyc database (`http://metacyc.org`) containing 13882 reactions. This set of reactions spans the search space specified in ([5]) for metabolic network completion.

In the first set of experiments, we proceed in two steps. First, we compute for each draft network and each target, the minimum number of reactions that need to be added to complete the network. Then, we compute all solutions satisfying this optimality criterion. In fact, in view of the large set of candidate reactions in the reference network, this approach turned out to be superior to a single step approach, enumerating all optimal solution through *clasp*'s branch and bound algorithm. Rather, we invoke *clasp* with

[3] Instances and encodings are available at: `http://www.cs.unipotsdam.de/bioasp`

Table 1. computing optimal completions for E. coli networks

	E.coli-50				E.coli-100				E.coli-200			
T	t_{opt}	*opt*	t_{all}	*#opt*	t_{opt}	*opt*	t_{all}	*#opt*	t_{opt}	*opt*	t_{all}	*#opt*
1	0.14	0	0.17	1	0.19	0	0.16	1	**368.72**	**1**	**2.17**	**7**
2	0.18	0	0.17	1	0.18	0	0.16	1	**368.52**	**1**	**2.22**	**7**
3	0.16	0	0.16	1	0.17	0	0.18	1	**195.34**	**7**	**304.35**	**135**
4	0.20	0	0.17	1	0.21	0	0.18	1	**42.89**	**3**	**20.40**	**35**
5	0.18	0	0.16	1	0.18	0	0.14	1	0.15	0	0.15	1
6	0.16	0	0.16	1	**159.07**	**2**	**2.53**	**6**	**226.41**	**7**	-	-
7	0.16	0	0.18	1	0.21	0	0.16	1	-	-	-	-
8	0.15	0	0.14	1	0.17	0	0.15	1	**46.39**	**3**	**29.59**	**35**
9	0.16	0	0.19	1	0.14	0	0.15	1	0.15	0	0.14	1
10	0.18	0	0.17	1	0.14	0	0.16	1	0.14	0	0.17	1
11	0.18	0	0.18	1	0.15	0	0.15	1	**26.58**	1	**2.18**	**7**
12	0.14	0	0.16	1	0.15	0	0.16	1	-	-	-	-
13	-	-	-	-	**105.15**	**4**	**12.35**	**1**	-	-	-	-
14	0.17	0	0.18	1	0.15	0	0.17	1	0.16	0	0.14	1
15	0.13	0	0.19	1	0.16	0	0.17	1	0.18	0	0.16	1
16	0.15	0	0.16	1	0.16	0	0.18	1	**367.10**	**1**	**2.20**	**7**
17	0.20	0	0.16	1	-	-	-	-	-	-	-	-
18	-	-	-	-	-	-	-	-	-	-	-	-
19	-	-	-	-	**80.63**	**2**	**5.18**	**3**	-	-	-	-
20	-	-	-	-	-	-	-	-	-	-	-	-
21	0.18	0	0.17	1	0.15	0	0.15	1	0.16	0	0.15	1
22	0.16	0	0.17	1	0.19	0	0.16	1	0.14	0	0.15	1
23	0.17	0	0.14	1	0.16	0	0.15	1	**353.70**	**1**	**2.17**	**1**
24	**37.87**	**3**	**21.28**	**4**	**3.92**	**6**	**29.78**	**5**	-	-	-	-
25	-	-	-	-	-	-	-	-	-	-	-	-
26	-	-	-	-	-	-	-	-	-	-	-	-
27	0.15	0	0.17	1	0.14	0	0.18	1	**46.07**	**3**	**37.08**	**35**
28	0.16	0	0.19	1	-	-	-	-	0.16	0	0.13	1

the option `--restart-on-model` that restarts after each minimum solution. This makes *clasp* converge much faster to an optimum solution. Once this is found, *clasp* is invoked again for enumerating all solutions satisfying the optimality criterion.

Table 1 summarizes our first set of experiments. The columns headed by E.coli-50, E.coli-100, and E.coli-200, respectively, provide results obtained on the aforementioned draft networks obtained by removing 50, 100 and 200, respectively, reactions from the original E.coli network. The first column identifies the chosen target metabolite. Then, for each draft network, the columns labeled t_{opt} show the time in seconds for computing the minimum number of reactions that need to be added to produce the target. The columns labeled *opt* provide the minimum number of reactions. The columns t_{all} show the time in seconds for computing all optimal solutions and the column *#opt* gives the number of optimal solutions.

For targets that could not be produced by the draft network, the results are either shown in boldface or are timeouts. For target metabolites whose production pathways are not disturbed, the computation time is insignificant. We observe six timeouts, while searching for an optimal completion on the E.coli-50 network. These six target metabolites could not be produced by the draft network in general. Interestingly, those metabolites cannot be produced by all three draft networks, giving us the hint that the pathways for this metabolites are very fragile. Comparing the results for E.coli-50 and E.coli-100, we see that for two targets, the experiments on E.coli-50 timeout, while they could be solved in time on E.coli-100. For E.coli-200, we see 10 experiments timeout, 10 computing the optimal value in time, and for 9 experiments *clasp* finishes computing all optimal solutions in time. This suggests that pathways, which can be disturbed by removing few reactions, are very fragile and hard to reconstruct, while more robust pathways, which are only disturbed when removing lots of reactions, are more easily repaired. For target metabolites whose production pathways are not disturbed, the computation time is insignificant.

In our second experiment, we investigate the scalability of our approach in view of the size of the reference network, taking into account the entire set of target metabolites. We created subsets of the MetaCyc network, choosing 10 random samples of 5000, 6000, 7000, 8000, and 9000 reactions. We fixed the draft network by removing 200 reactions from the E.coli network and tried to complete its completion relative to the differently large reference networks. Note that the joint explanation of all 28 targets is much more difficult than just explaining a single target. This is because the restrictions to interesting reactions introduced in Section 3 become less effective when aiming at multiple targets. On the other hand, the identification of a minimum completion producing a maximum set of target is a highly significant question in synthetic biology.

As above, our experiments use a multi-step process. In a first step, we use *clasp* to compute for each reference network the minimum number of reactions needed to complete the network. Once we have computed the optimal value, we continue by computing the reactions essential to all 28 targets, that is, the reactions contained in every answer set satisfying the optimality criterion. This is accomplished by computing the cautious consequences using the option `--cautious` of *clasp*. These reactions are essential for the joint production of all target metabolites. Finally, we use *clasp* as before to enumerate all optimal solutions.

The first line gives the size of the investigated reference network. The columns labeled with i identify the instance of the reference network. The column t_{opt} gives the computation time for computing the minimum number of reactions needed for a completion. Column t_c shows the time needed to compute the essential reactions, that is, all reactions that are in all minimum completion. The columns labeled with t_{all} show the time needed to compute all optimal solutions. The columns labeled with *#opt* show how many optimal solution have been found. All times are given in seconds.

We observe that the problem is easily handled up to a size of 6000 reactions; all such problems can be solved under a second. Starting with 7000 reactions, we start to obtain computational more demanding problems, and finally a lot of timeouts at size 9000. Notably, our experiments are restricted by a timeout of 10 minutes; existing approaches to network completion usually run simulations over the period of a day. Of course, we have to extend the timeout in a production mode as well. Interestingly, the successful runs show that finding the optimal number of solutions takes most of the computation time; an issue we want to address in the future by biological domain-specific heuristics.

Table 2. Completion with 5000,. . . ,9000 reactions

5000					
i	t_{opt}	*opt*	t_c	t_{all}	*#opt*
1	0.25	5	0.19	0.21	72
2	0.23	8	0.16	0.19	36
3	0.20	5	0.16	0.20	3
4	0.14	11	0.20	0.17	12
5	0.18	3	0.16	0.14	24
6	0.39	11	0.26	0.26	24
7	0.18	4	0.18	0.15	2
8	0.18	9	0.22	0.23	60
9	0.27	15	0.19	0.20	16
10	0.15	3	0.16	0.14	6

6000					
i	t_{opt}	*opt*	t_c	t_{all}	*#opt*
1	0.24	4	0.20	0.20	6
2	0.28	4	0.23	0.19	3
3	3.46	16	0.46	0.43	50
4	0.17	7	0.21	0.22	6
5	0.19	2	0.21	0.23	4
6	0.54	17	0.26	0.28	28
7	0.23	5	0.21	0.21	8
8	0.29	19	0.18	0.26	55
9	0.43	9	0.23	0.27	24
10	0.18	3	0.16	0.18	30

7000					
i	t_{opt}	*opt*	t_c	t_{all}	*#opt*
1	105.16	27	3.24	3.33	160
2	-				
3	10.82	19	2.15	1.86	48
4	0.38	5	0.36	0.38	168
5	0.83	14	0.46	0.44	27
6	0.58	7	0.42	1.15	10
7	0.30	2	0.27	0.23	3
8	16.12	14	0.38	0.54	88
9	58.00	17	1.39	0.89	300
10	11.20	18	9.40	8.28	80

8000					
i	t_{opt}	*opt*	t_c	t_{all}	*#opt*
1	265.14	15	274.56	251.34	672
2	1.07	7	0.25	0.30	5
3	5.16	13	1.23	1.13	4
4	1.50	8	0.36	0.35	10
5	0.68	13	0.87	0.98	12
6	78.49	20	48.91	49.67	288
7	195.66	8	1.77	1.58	40
8	5.98	15	3.44	3.63	24
9	9.08	11	0.53	0.59	8
10	0.89	11	0.48	0.42	12

9000					
i	t_{opt}	*opt*	t_c	t_{all}	*#opt*
1	12.34	17	7.45	8.95	18
2	-				
3	28.05	12	11.32	13.99	88
4	-				
5	-				
6	410.76	30	3.88	3.79	14
7	271.02	16	11.13	28.61	2976
8	-				
9	-				
10	-				

4.2 Inverse Scope Problem

Last but not least, let us evaluate our approach to the inverse scope problem. As above, we consider the complete E.coli network and try to compute for every target the minimum number of seeds needed to produce it. Once accomplished, we enumerate all minimum sets of seeds.

Again, we first solve the optimality problem and use *clasp* to compute the minimum number of seeds needed to produce the target metabolite; and in a second step we relaunch *clasp* to compute all optimal solutions.

The first column denotes the target metabolite for whose production the seeds are computed. The second column shows the time in seconds for computing the minimum number of seeds. The third one gives the minimum number of seeds. The fourth column shows the time in seconds for computing all optimal solutions, and the fifth one shows the number of optimal solutions.

The results show that most of the targets can be produced by providing one or two seeds only. Interestingly, we found that only groups of three seeds are needed to produce all 28 targets. We also checked with the cautious reasoning mode for essential seeds, belonging to all minimum solution but none were found. We further used *clasp* with option `--brave` to compute the union of all reactions occurring in optimal solutions and found a set of 136 different metabolites, from which all minimum sets of seeds are

Table 3. Computing minimal seeds for E.coli targets

T	t_{opt}	*opt*	t_{all}	*#opt*
1	2.40	1	14.82	6
2	0.45	1	35.76	12
3	0.38	1	16.02	6
4	28.21	1	25.42	4
5	19.41	2	-	-
6	4.30	2	187.06	50
7	1.29	2	166.73	63
8	15.79	1	17.24	4
9	13.45	1	13.98	4
10	0.89	1	17.00	5
11	0.53	1	25.92	9
12	7.28	1	14.78	4
13	4.78	1	9.88	4
14	13.23	1	7.67	4
15	0.32	1	29.58	11
16	0.32	1	31.16	11
17	14.05	1	24.46	1
18	0.28	1	19.66	3
19	10.44	2	-	-
20	23.33	1	27.58	5
21	14.23	1	6.90	4
22	0.37	1	49.79	11
23	-	-	-	-
24	-	-	-	-
25	17.19	1	21.36	4
26	0.55	1	33.24	5
27	19.85	1	15.22	4
28	-	-	-	-

taken. Since we are only discriminating the targets among the seeds in (13), we were surprised to find many seeds among the reactants of the reactions producing the targets. However, for more meaningful results, we need more biological knowledge, to exclude more metabolites as seeds.

5 Discussion

The easy characterization of reachability is one of the key features of ASP. We have exploited this to provide a simple yet powerful account of metabolic network synthesis, a crucial application in the elaboration and design of bioprocesses. The distinguishing feature of our ASP-based approach lies in the unique combination of ease of modelling and powerful reasoning modes, supported by efficient solver technology. In fact, existing qualitative approaches to network synthesis are based on stochastic simulations based on hidden Markov models (cf. [11]), taking several hours to obtain results from the relative frequencies of compounds in the simulations. Unlike this, our approach is complete and thus allows for proving rather than estimating the production of metabolites. Moreover, the various reasoning modes, including the enumeration of optimal solutions as well as cautious and brave reasoning with respect to all or optimal solutions only, respectively, are indispensable in a biological application due to the large number of possible solutions. For instance, cautious reasoning relative to optimal solutions makes us discover the essential nutritions for producing a target metabolite. These reasoning modes together with the high-level specification of metabolic networks make our approach attractive to biologists, given that they can easily elaborate and explore their model "in silico" by means of ASP.

From the perspective of ASP, our application fostered the development of new reasoning modes that were implemented within the ASP solver *clasp* (1.2.0).[4] For one

[4] `http://potassco.sourceforge.net`

thing, *clasp* allows for optimization techniques not available in any other ASP solver. Of particular interest is the `--restart-on-model` option that restarts after finding a solution (instead of backtracking). This led to a significant increase in converging to an optimal solution. To a turn, we then exploit the options `--opt-all` and `--opt-value` for enumerating all optimal models. Even though the latter can also be addressed by adding an appropriate constraint to the underlying ASP program, the options allow us to leave the underlying program untouched. For another thing, *clasp* allows for computing all brave and cautious consequences[5] by means of a linear[6] number of calls to a solver (internally computing one answer set) rather then enumerating the entire set of answer sets. This is accomplished by consecutive refinements of an internal constraint by appeal to the incremental solving techniques introduced in [16]. This feature is also unique to *clasp*, although a-priori given brave and cautious queries can be decided by other ASP solvers, like *dlv* [17], as well.

Although, to the best of our knowledge, our application is novel in the field of ASP in particular and declarative programming in general, there has been an increasing interest in using ASP and/or LP technology for addressing biological problems over the last years. Among them, we find [18,19,20,21] as well as [22] building upon abductive logic programming.

Future work will mainly deal with the elaboration of biological domain knowledge for a better narrowing of the solution space and the application of our methodology in the construction of the metabolic network of the recently sequenced green alga *Chlamydomonas reinhardtii* (`http://www.goforsys.de`). To hint biologists to missing reactions, we currently explore only the space of known chemical reactions, but in future one could also envisage abducing yet unknown chemical reactions, similar to [22].

Acknowledgements. This work was funded by the Federal Ministry of Education and Research within the GoFORSYS project (http://www.goforsys.org/; grant 0313924).

References

1. Savageau, M.: Biochemical system analysis: a study of function and design in molecular biology. Addison-Wesley, Reading (1976)
2. Kompala, D., Ramkrishna, D., Jansen, N., Tsao, G.: Investigation of bacterial-growth on mixed substrates. Biotechnology and Bioengineering 28(7), 1044–1055 (1986)
3. Bonarius, H., Schmid, G., Tramper, J.: Flux analysis of underdetermined metabolic networks: The quest for the missing constraints. Trends Biotechnology 15, 308–314 (1997)
4. Schilling, C., Schuster, S., Palsson, B., Heinrich, R.: Metabolic pathway analysis: Basic concepts and scientific applications in the post-genomic era. Biotechnology progress 15, 296–303 (1999)
5. Wildermuth, M.: Metabolic control analysis: biological applications and insights. Genome Biology 1(6), 1031.1–1031.5 (2000)
6. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press, Cambridge (2003)

[5] This is accomplished with options `--brave` and `--cautious`.

[6] That is, linear in the number of atoms.

7. Nikoloski, Z., Grimbs, S., May, P., Selbig, J.: Metabolic networks are np-hard to reconstruct. Journal of Theoretical Biology 254, 807–816 (2008)
8. Nikoloski, Z., Grimbs, S., Selbig, J., Ebenhöh, O.: Hardness and approximability of the inverse scope problem. In: Crandall, K.A., Lagergren, J. (eds.) WABI 2008. LNCS (LNBI), vol. 5251, pp. 99–112. Springer, Heidelberg (2008)
9. Ebenhöh, O., Handorf, T., Heinrich, R.: Structural analysis of expanding metabolic networks. Genome Informatics 15(1), 35–45 (2004)
10. Handorf, T., Ebenhöh, O., Heinrich, R.: Expanding metabolic networks: Scopes of compounds, robustness, and evolution. Journal of Molecular Evolution 61(4), 498–512 (2005)
11. Christian, N., May, P., Kempa, S., Handorf, T., Ebenhöh, O.: An integrative approach towards completing genome-scale metabolic networks (2008) (submitted for publication)
12. Handorf, T., Ebenhöh, O., Heinrich, R.: An environmental perspective on metabolism. Journal of Theoretical Biology 252(3), 498–512 (2008)
13. Simons, P., Niemelä, I., Soininen, T.: Extending and implementing the stable model semantics. Artificial Intelligence 138(1-2), 181–234 (2002)
14. Gebser, M., Kaufmann, B., Schaub, T.: Solution enumeration for projected boolean search problems. In: van Hoeve, W.-J., Hooker, J.N. (eds.) CPAIOR 2009. LNCS, vol. 5547, pp. 71–86. Springer, Heidelberg (2009)
15. Christian, N., May, P., Kempa, S., Handorf, T., Ebenhöh, O.: Personal communication (2008)
16. Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., Thiele, S.: Engineering an incremental ASP solver. In: [23], pp. 190–205
17. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV system for knowledge representation and reasoning. ACM TOCL 7(3), 499–562 (2006)
18. Baral, C., Chancellor, K., Tran, N., Tran, N., Joy, A., Berens, M.: A knowledge based approach for representing and reasoning about signaling networks. In: Proceedings ISMB 2004/ECCB 2004, pp. 15–22 (2004)
19. Dworschak, S., Grell, S., Nikiforova, V., Schaub, T., Selbig, J.: Modeling biological networks by action languages via answer set programming. Constraints 13(1-2), 21–65 (2008)
20. Gebser, M., Schaub, T., Thiele, S., Usadel, B., Veber, P.: Detecting inconsistencies in large biological networks with answer set programming. In: [23], pp. 130–144
21. Erdem, E., Türe, F.: Efficient haplotype inference with answer set programming. In: Proceedings AAAI 2008, pp. 436–441. AAAI Press, Menlo Park (2008)
22. Ray, O., Whelan, K., King, R.: A nonmonotonic logical approach for modelling and revising metabolic networks. In: Proceedings CISIS 2009. IEEE Press, Los Alamitos (to appear, 2009)
23. Garcia de la Banda, M., Pontelli, E. (eds.): ICLP 2008. LNCS, vol. 5366. Springer, Heidelberg (2008)

Answer Set Programming for Single-Player Games in General Game Playing

Michael Thielscher

Department of Computer Science
Dresden University of Technology
mit@inf.tu-dresden.de

Abstract. As a novel, grand AI challenge, General Game Playing is concerned with the development of systems that understand the rules of unknown games and play these games well without human intervention. In this paper, we show how Answer Set Programming can assist a general game player with the special class of single-player games. To this end, we present a translation from the general *Game Description Language* (GDL) into answer set programs (ASP). Correctness of this mapping is established by proving that the stable models of the resulting ASP coincide with the possible developments of the original GDL game. We report on experiments with single-player games from past AAAI General Game Playing Competitions which substantiate the claim that Answer Set Programming can provide valuable support to general game playing systems for this type of games.

1 Introduction

General Game Playing is concerned with the development of systems that understand the rules of previously unknown games and play these games well without human intervention. Identified as a new grand AI challenge, this endeavor requires to combine methods from a variety of a sub-disciplines, such as Knowledge Representation and Reasoning, Search, Game Playing, Planning, and Learning [1,2,3,4,5]. The annual AAAI General Game Playing contest has been established in 2005 to foster research in this area and to evaluate general game playing systems in a competitive setting [6]. During the competition, participating systems receive the rules of hitherto unknown games. The contestants get some time to "contemplate" about the game (typically 5 to 20 minutes) and then start playing against each other with a further time limit for each move (typically 20 to 60 seconds). All this takes place without human interference.

General game playing requires to formalize the rules of arbitrary games in such a way that they can be processed by machines. The *Game Description Language* (GDL) [7] serves this purpose by allowing one to describe any finite and information-symmetric n-player game. GDL uses the syntax of normal logic programs, and its semantics is given by a formal game model [8] on the basis of the *standard model* of stratified programs as defined in [9]. Due to the closeness of GDL rules to Answer Set Programs, in both syntax and semantics, the question naturally arises whether this programming paradigm can provide valuable support to a general game playing system.

In this paper, we give a positive answer to this question by showing that Answer Set Programming can assist general game players with the special class of single-player games. This type of games provides for an indirect competition: independent of the others, each player tries to achieve the best possible outcome according to the rules. Examples from previous AAAI competitions are the well-known game of Peg Jumping, where the goal is to end up with as few pegs as possible on a given board, or Knight's Tour, where the goal is to visit as many squares as possible on a checkerboard of a given size.

Successful general game playing systems, such as [2,3,4], use automatically generated heuristics in combination with search. A well-known technique in game playing is *endgame search* (see, e.g., [10]), which means to perform a depth-restricted, complete forward search from the current position in order to see whether a winning position has been reached. In this paper, we show how Answer Set Programming can be used for this purpose in General Game Playing. Inspired by existing approaches of using satisfiability techniques for planning problems [11,12,13], we first map any (single- or multi-player) GDL specification onto an ASP in such a way that the stable models coincide with the possible developments of the original game. We then show how Answer Set Programming can be used to perform a complete, depth-restricted forward search during game play in case of single-player games. Experiments with a variety of single-player games from past AAAI General Game Playing Competitions [6] show that for most games, Answer Set Programming clearly outperforms the techniques for complete forward search that are built into the currently best general game players and thus provides a valuable addition to any such system.

The rest of the paper is organized as follows. In the next section, we recapitulate the basic syntax and semantics of GDL. In the section that follows, we map GDL descriptions onto "temporally extended" answer set programs and prove that the stable models for the resulting program coincides with the possible developments of the original game. In Section 4, we present a provably correct method of applying this result to perform endgame search in single-player games using Answer Set Programming. In Section 5, we give an overview of successful experiments with an off-the-shelf ASP system [14] for a variety of single-player games taken from the past AAAI General Game Playing Competitions. We conclude in Section 6. For the rest of the paper, we assume that the reader is familiar with the basic concepts of answer sets, as can be found, e.g., in [15].

2 Game Description Language

The Game Description Language (GDL) has been developed to formalize the rules of any finite game with complete information in such a way that the description can be automatically processed by a general game player. Due to lack of space, we can give just a very brief introduction to GDL and have to refer to [7] for details.

Table 1. The GDL keywords

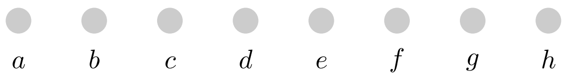
`role(R)`	`R` is a player
`init(F)`	`F` holds in the initial position
`true(F)`	`F` holds in the current position
`legal(R,M)`	player `R` has legal move `M`
`does(R,M)`	player `R` does move `M`
`next(F)`	`F` holds in the next position
`terminal`	the current position is terminal
`goal(R,N)`	player `R` gets goal value `N`

GDL is based on the standard syntax of normal logic programs. We adopt the Prolog convention according to which variables are denoted by uppercase letters and predicate and function symbols start with a lowercase letter. As a tailor-made specification language, GDL uses a few pre-defined predicate symbols shown in Table 1. A further standard predicate is `distinct(X,Y)`, which means syntactic inequality of the two arguments.

GDL imposes the following restrictions on the use of these keywords in a set of clauses describing a game.

– `role` only appears in facts;
– `init` and `next` only appear as head of clauses, and `init` is not connected (in the dependency graph for the set of clauses) to any of `true`, `legal`, `does`, `next`, `terminal`, or `goal`;
– `true` and `does` only appear in clause bodies, and `does` is not connected to any of `legal`, `terminal`, or `goal`.

As an example, Figure 1 shows a complete set of GDL rules for the following, simple single-player game. Starting with eight coins in a row,



jump with any coin forming a singleton stack over two coins onto another single coin. Repeat until you end up with as few as possible (ideally, zero) single coins.[1]

GDL imposes some further, general restrictions on a set of clauses with the intention to ensure finiteness of the set of derivable predicate instances. Specifically, the program must be *stratified* [9,16] and *allowed* [17]. Stratified logic programs are known to admit a specific *standard model* as defined in [9]. Based on this concept of a standard model, a set of GDL rules can be understood as a description of a formal game model—a state transition system—as follows [8].

To begin with, any valid game description G in GDL contains a finite set of function symbols, including constants, which implicitly determines a set of

[1] For instance, you may first jump with the coin in a over the coins in b and c onto the coin in d. Next, you can take the single coin in c and jump over the two coins which are now in position d, landing on the coin in e. But then no further move will be possible, which according to the rules in Figure 1 means goal value 0.

```
role(player).
succ(a,b).
...
succ(g,h).
init(cell(a,single)).
init(cell(Y,single)) :- succ(X,Y).
legal(P,jump(X,Y))   :- true(cell(X,single)), true(cell(Y,single)),
                        twobetween(X,Y).
legal(P,jump(X,Y))   :- true(cell(X,single)), true(cell(Y,single)),
                        twobetween(Y,X).
next(cell(X,nocoin)) :- does(player,jump(X,Y)).
next(cell(X,double)) :- does(player,jump(Y,X)).
next(cell(X,Number)) :- true(cell(X,Number)), does(player,jump(Y,Z)),
                        distinct(X,Y), distinct(X,Z).

terminal          :- not continuable
continuable       :- legal(player,Move).
goal(player,100) :- not lonelycoin.
goal(player, 50) :- lonelycoin, not threelonelycoins.
goal(player,  0) :- threelonelycoins.
lonelycoin        :- true(cell(X,single)).
threelonelycoins :- true(cell(X,single)), true(cell(Y,single)),
                     true(cell(Z,single)), distinct(X,Y),
                     distinct(X,Z), distinct(Y,Z).
twobetween(X,Y)   :- succ(X,Z), true(cell(Z,nocoin)), twobetween(Z,Y).
twobetween(X,Y)   :- succ(X,Z), true(cell(Z,single)), onebetween(Z,Y).
twobetween(X,Y)   :- succ(X,Z), true(cell(Z,double)), nilbetween(Z,Y).
onebetween(X,Y)   :- succ(X,Z), true(cell(Z,nocoin)), onebetween(Z,Y).
onebetween(X,Y)   :- succ(X,Z), true(cell(Z,single)), nilbetween(Z,Y).
nilbetween(X,Y)   :- succ(X,Z), true(cell(Z,nocoin)), nilbetween(Z,Y).
nilbetween(X,Y)   :- succ(X,Y).
```

Fig. 1. A complete GDL description for the coin game. The positions are encoded using the feature $\mathit{cell}(X,Y)$, where $X \in \{a,\dots,h\}$ and $Y \in \{\mathit{nocoin}, \mathit{single}, \mathit{double}\}$.

ground terms Σ. This set constitutes the symbol base Σ in the formal semantics for G. The players and the initial position of a game can be directly determined from the clauses for, respectively, `role` and `init` in G. In order to determine the legal moves, update, termination, and goalhood for any given position, this position has to be encoded first, using the keyword `true`. To this end, for any *finite* subset $S = \{f_1, \dots, f_n\} \subseteq \Sigma$ of a set of ground terms, the following set of logic program facts encodes S as the current position.

$$S^{\mathtt{true}} \stackrel{\text{def}}{=} \{\mathtt{true}(f_1)\mathtt{.},\ \dots,\ \mathtt{true}(f_n)\mathtt{.}\}$$

The legal moves for each player r in position S can then be determined as the derivable instances of $\mathtt{legal}(r,\mathtt{M})$. Similarly, the fact whether S is terminal is determined by whether `terminal` is derivable, in which case the derivable instances of $\mathtt{goal}(r,\mathtt{N})$ determine the goal values for the individual players.

Finally, for any function $A : \{r_1, \ldots, r_n\} \mapsto \Sigma$ that assigns a move to each player $r_1 \in \Sigma, \ldots, r_n \in \Sigma$, let the following set of facts encode A as joint move.

$$A^{\texttt{does}} \stackrel{\text{def}}{=} \{\texttt{does}(r_1, A(r_1))\texttt{.},\ \ldots,\ \texttt{does}(r_n, A(r_n))\texttt{.}\}$$

The derivable instances of `next(F)` then determine the position resulting from joint move A in the current position encoded by $S^{\texttt{true}}$. All this is summarized in the following definition.

Definition 1. [8] *Let G be a GDL specification whose signature determines the set of ground terms Σ. The* semantics *of G is the state transition system (R, S_1, T, l, u, g) where*[2]

- $R = \{r \in \Sigma : G \models \texttt{role}(r)\}$ *(the players);*
- $S_1 = \{f \in \Sigma : G \models \texttt{init}(f)\}$ *(the initial position);*
- $T = \{S \in 2^{\Sigma} : G \cup S^{\texttt{true}} \models \texttt{terminal}\}$ *(the terminal positions);*
- $l = \{(r, a, S) : G \cup S^{\texttt{true}} \models \texttt{legal}(r, a)\}$*, where* $r \in R$*,* $a \in \Sigma$*, and* $S \in 2^{\Sigma}$ *(the legality relation);*
- $u(A, S) = \{f \in \Sigma : G \cup S^{\texttt{true}} \cup A^{\texttt{does}} \models \texttt{next}(f)\}$*, for all* $A : (R \mapsto \Sigma)$ *and* $S \in 2^{\Sigma}$ *(the update function);*
- $g = \{(r, n, S) : G \cup S^{\texttt{true}} \models \texttt{goal}(r, n)\}$*, where* $r \in R$*,* $n \in \mathbb{N}$*, and* $S \in 2^{\Sigma}$ *(the goal relation).*

For example, given the game rules in Figure 1 it is easy to see that the initial state is

$$S_1 = \{\mathit{cell}(a, \mathit{single}), \ldots, \mathit{cell}(d, \mathit{single}), \ldots, \mathit{cell}(h, \mathit{single})\}$$

The addition of $S_1^{\texttt{true}}$ to the game rules shows that $(\mathit{player}, \mathit{jump}(a, d), S_1) \in l$, and the resulting position (with $A = \{\mathit{player} \mapsto \mathit{jump}(a, d)\}$) is

$$u(A, S_1) = \{\mathit{cell}(a, \mathit{nocoin}), \ldots, \mathit{cell}(d, \mathit{double}), \ldots, \mathit{cell}(h, \mathit{single})\}$$

Definition 1 provides a formal semantics by which a GDL description is interpreted as an abstract n-player game: in every position S, starting with S_1, each player r chooses a move a that satisfies $l(r, a, S)$. As a consequence the game state changes to $u(A, S)$, where A is the joint move. We introduce the following notation for possible developments in a game. Consider two finite sequences of, respectively, joint moves $A_1, \ldots, A_k$ and states $S_2, \ldots, S_{k+1}$ $(k \geq 0)$. Then

$$S_1 \xrightarrow{A_1} S_2 \xrightarrow{A_2} \ldots \xrightarrow{A_{k-1}} S_k \xrightarrow{A_k} S_{k+1} \tag{1}$$

if for each $i = 1, \ldots, k$ we have

- $S_i \notin T$,
- $(r, A_i(r), S_i) \in l$ for each $r \in R$, and
- $S_{i+1} = u(A_i, S_i)$.

[2] Below, entailment $\models$ is via the aforementioned standard mode as defined in [9], and 2^{Σ} denotes the *finite* subsets of Σ.

The game ends when a position in T is reached, and then g determines the outcome for each player. The syntactic restrictions in GDL (see [7] for details) ensure that entailment wrt. the standard model is decidable and that only finitely many instances of each predicate are entailed. This guarantees that the definition of the semantics is effective.

3 Mapping GDL to a Logic Program with Time

The semantics for GDL according to Definition 1 shows that the plain game rules need to be repeatedly applied when determining the legal moves and their effects for different positions. Therefore, in order to be able apply logic programming techniques to directly reason about the evolution of a game position, the rules need to be "temporalized" in a way that is common for encodings of temporal domains as logic programs (see, e.g., [11,12,18]). In the following, we use the new predicate `holds(F,T)` to denote that feature `F` holds in the game position *at time* `T`. Time shall be encoded by natural numbers starting with `1`.

Definition 2. *Let G be a set of GDL rules, then the* temporal extension *of G, written $ext(G)$, is the set of logic program clauses obtained from G as follows.*

1. *Each occurrence of* `init`(φ) *is replaced by* `holds(`φ`,1)`*, and each atom $p(t_1, \ldots, t_n)$ in the body of a clause for* `init` *is replaced by $p(t_1, \ldots, t_n, 1)$, provided that $p \neq$* `distinct`*.*
2. *Each occurrence of* `true`(φ) *is replaced by* `holds(`φ`,T)`*, and each* `next`(φ) *by* `holds(`φ`,T+1)`*.*
3. *Each occurrence of each atom $p(t_1, \ldots, t_n)$ is replaced by $p(t_1, \ldots, t_n, \mathtt{T})$, provided that $p \notin \{$*`init`, `true`, `next`, `role`, `distinct`*$\}$, and* `distinct`(t_1, t_2) *is replaced by* `not` t_1`=`t_2*.*

As an example, consider the temporal extension of the game rules in Figure 1.

```
role(player).
succ(a,b,T).
...
succ(g,h,T).
holds(cell(a,single),1).
holds(cell(Y,single),1) :- succ(X,Y,1).

legal(P,jump(X,Y),T) :- holds(cell(X,single),T),
                        holds(cell(Y,single),T),
                        twobetween(X,Y,T).
legal(P,jump(X,Y),T) :- holds(cell(X,single),T),
                        holds(cell(Y,single),T),
                        twobetween(Y,X,T).

holds(cell(X,nocoin),T+1) :- does(player,jump(X,Y),T).
holds(cell(X,double),T+1) :- does(player,jump(Y,X),T).
holds(cell(X,Number),T+1) :- holds(cell(X,Number),T),
                             does(player,jump(Y,Z),T),
                             not X=Y, not X=Z.

terminal(T)          :- not continuable(T).
continuable(T)       :- legal(player,Move,T).
goal(player,100,T)   :- not lonelycoin(T).
goal(player, 50,T)   :- lonelycoin(T), not threelonelycoins(T).
goal(player,  0,T)   :- threelonelycoins(T).
lonelycoin(T)        :- holds(cell(X,single),T).
threelonelycoins(T) :- holds(cell(X,single),T),
                        holds(cell(Y,single),T),
                        holds(cell(Z,single),T),
                        not X=Y, not X=Z, not Y=Z.

twobetween(X,Y,T) :- succ(X,Z,T), true(cell(Z,nocoin),T),
                     twobetween(Z,Y,T).
twobetween(X,Y,T) :- succ(X,Z,T), true(cell(Z,single),T),
                     onebetween(Z,Y,T).
twobetween(X,Y,T) :- succ(X,Z,T), true(cell(Z,double),T),
                     nilbetween(Z,Y,T).
onebetween(X,Y,T) :- succ(X,Z,T), true(cell(Z,nocoin),T),
                     onebetween(Z,Y,T).
onebetween(X,Y,T) :- succ(X,Z,T), true(cell(Z,single),T),
                     nilbetween(Z,Y,T).
nilbetween(X,Y,T) :- succ(X,Z,T), true(cell(Z,nocoin),T),
                     nilbetween(Z,Y,T).
nilbetween(X,Y,T) :- succ(X,Y,T).
```

It is easy to see that the resulting program can be made more efficient by omitting the time argument in any predicate that

1. is not among the GDL keywords, and
2. does not depend on `true` in the original game description.

Thus, for instance, predicate $succ(x, y)$ in our example need not carry the time argument because its extension does not depend on the current game position. This independence can be easily computed from the dependency graph for a set of GDL rules.

The ease with which GDL descriptions can be mapped onto a logic program with explicit time is the major reason for the expectation that Answer Set Programming can be a valuable addition to a general game playing system. The temporalized GDL rules allow us to encode the fact that *at time* t the players choose a joint legal move $A : R \mapsto \Sigma$ (where $R = \{r_1, \ldots, r_n\}$ are the roles in the game and Σ the symbol base) by the following additional facts.

$$A^{\texttt{does}}(t) \overset{\text{def}}{=} \{\texttt{does}(r_1, A(r_1), t)\texttt{.},\ \ldots,\ \texttt{does}(r_n, A(r_n), t)\texttt{.}\}$$

With this, the mapping of a GDL description to a logic program with time can be proved correct with regard to the semantics of GDL according to Definition 1.

Theorem 1. *Let G be a valid GDL description and (R, S_1, T, l, u, g) its semantics. For any finite sequence of legal joint moves and states, we have that*

$$S_1 \xrightarrow{A_1} S_2 \xrightarrow{A_2} \ldots S_k \xrightarrow{A_k} S_{k+1}$$

if and only if the standard model $\mathcal{M}$ of $ext(G) \cup A_1^{\texttt{does}}(1) \cup \ldots \cup A_k^{\texttt{does}}(k)$ satisfies the following.

- $S_i = \{f \in \Sigma : \mathcal{M} \models \texttt{holds}(f, i)\}$ *for each* $1 \le i \le k+1$ *and*
- $\mathcal{M} \models \texttt{legal}(r, A_i(r), i)$ *for each* $r \in R$ *and each* $1 \le i \le k$*.*

Proof. By induction. For $k = 1$ the claim follows from

- the replacement of $\texttt{init}(\varphi)$ by $\texttt{holds}(\varphi, 1)$ in $ext(G)$, and by the construction of S_1 in Definition 1;
- the replacement of $\texttt{legal}(\varrho, \alpha)$ and $\texttt{true}(\varphi)$ by, respectively, $\texttt{legal}(\varrho, \alpha, \texttt{T})$ and $\texttt{holds}(\varphi, \texttt{T})$ in $ext(G)$, and by the construction of l in Definition 1.

The induction step follows from

- the replacement of $\texttt{next}(\varphi)$, $\texttt{true}(\varphi)$, and $\texttt{does}(\varrho, \alpha)$ by $\texttt{holds}(\varphi, \texttt{T}+1)$, $\texttt{holds}(\varphi, \texttt{T})$, and $\texttt{does}(\varrho, \alpha, \texttt{T})$, respectively, in $ext(G)$, and by the construction of u in Definition 1;
- the replacement of $\texttt{legal}(\varrho, \alpha)$ and $\texttt{true}(\varphi)$ by, respectively, $\texttt{legal}(\varrho, \alpha, \texttt{T})$ and $\texttt{holds}(\varphi, \texttt{T})$ in $ext(G)$, and by the construction of l in Definition 1.

This result shows that the temporally extended logic program can be used to infer the evolution of the game position given a sequence of joint moves. As an example, consider the addition of the sequence of moves (cf. Footnote 1)

```
does(player,jump(a,d),1).
does(player,jump(c,e),2).
```

to the temporalized extension of the game description in Figure 1. It is easy to verify that the standard model for the resulting logic program includes each of the following.

```
legal(player,jump(a,d),1)       legal(player,jump(c,e),2)
holds(cell(a,nocoin),3)         holds(cell(b,single),3)
holds(cell(c,nocoin),3)         holds(cell(d,double),3)
holds(cell(e,double),3)         holds(cell(f,single),3)
holds(cell(g,single),3)         holds(cell(h,single),3)
terminal(3)                     goal(player,0,3)
```

4 Using ASP for Single-Player Games

Theorem 1 lays the foundation for the use of Answer Set Programming to perform endgame search for single-player games, that is, a complete, depth-restricted search starting in the current game position with the aim to find a winning sequence of moves within the given horizon. Because the standard model of a stratified program coincides with its only answer set (see, e.g., [15]), ASP can be used directly to determine the legality of a sequence of moves and the result from any current position. The basic idea, then, is to take the temporal extension of the GDL rules for a single-player game and to search for a sequence of moves that satisfies the following conditions.

1. Exactly one move is made at every point in time unless a terminal position has been reached.
2. Each move is legal when being played.
3. A terminal position is eventually reached.
4. The terminal position determines the intended goal value.

Any answer set that satisfies these constraints provides a solution to the game, that is, a sequence of moves that leads from the current position to a terminal position with the intended goal value.

In order to implement this search in a general game player, we need two common additions that have been defined for ASP [19]: a *weight atom*

$$m \; \{ \; p \; : \; d \; \} \; n$$

means that an answer set contains at least m and at most n different instances of atom p for which condition d holds (in the answer set). A *constraint* is a rule of the form `:-` $b_1, \ldots, b_k$ and excludes any answer set that satisfies all literals $b_1, \ldots, b_k$.

With the help of weight atoms and constraints, endgame search is performed by augmenting a temporally extended GDL specification for a single-player game by the clauses

```
1:  1{does(r,M,T) : move_domain(M)}1 :- not terminated(T).
    terminated(T)   :- terminal(T).
    terminated(T+1) :- terminated(T).
2:  :- does(r,M,T), not legal(r,M,T).                              (2)
3:  :- 0{terminated(T) : time_domain(T)}0.
4:  :- terminated(T), not terminated(T-1), not goal(r,g_max,T).
    :- terminated(1), not goal(r,g_max,1).
```

Here, r is assumed to be the (only) constant for which the game rules include the clause `role`(r), and natural number g_{max} stands for the goal value the game player is aiming for. These additional clauses provide a formal encoding of the four aforementioned conditions on the answer sets to provide a solution to the single-player game.

1: Predicate *terminated*(t) is used to indicate that a terminal position has been reached at (or before) time t. This auxiliary predicate is used to restrict the requirement for a legal move to every position before reaching a terminal one.
2: No answer set can stipulate a move that is not legal.
3: No answer set can have zero instances of *terminated*(t).
4: The state at the exact time of termination must have the desired goal value. (The last clause deals with the very special case that the game is already terminated at time 1.)

The ASP clauses in (2) require the definition of the domain of moves (using the predicate `move_domain`) according to the underlying game description. A suitable definition can be easily computed on the basis of the dependency graph for the given game description; see Section 5 for details. A similar definition is required for the domain of time, which is assumed to be given as $\{1, \ldots, n+1\}$ where n is the intended horizon for the endgame search.

If clauses (2) are added to the temporal extension of a GDL game according to Definition 2, then this amounts to a complete, depth-restricted search right from the initial position. Aiming instead at endgame search from the current position during game play, this can be easily achieved by substituting the collection of `holds`$(f, 1)$ facts, which result from the given `init`(f) clauses, by a collection of similar facts using the features that constitute the current position.

As an example, recall from the preceding section the temporal extension of the GDL rules of Figure 1. Let these be augmented by

```
coordinate(a).
...
coordinate(h).
move_domain(jump(X,Y)) :- coordinate(X), coordinate(Y).

1 { does(player,M,T) : move_domain(M) } 1 :- not terminated(T).
terminated(T)   :- terminal(T).
terminated(T+1) :- terminated(T).
:- does(player,M,T), not legal(player,M,T).
:- 0 { terminated(T) : time_domain(T) } 0.
:- terminated(T), not terminated(T-1), not goal(player,100,T).
:- terminated(1), not goal(player,100,1).
```

The answer sets for this program coincide with the solutions to the original game; an example is the answer that includes the following atoms.

```
does(player,jump(d,g),1)        does(player,jump(f,b),2)
does(player,jump(c,a),3)        does(player,jump(e,h),4)
terminal(5)                     goal(player,100,5)
```

The correctness of the method to solve single-player games with the help of Answer Set Programming is given by the following two theorems.

Theorem 2. *Consider a GDL description G with semantics (R, S_1, T, l, u, g) such that $R = \{r\}$ for some r. Let $\alpha_1, \ldots, \alpha_n$ and $S_2, \ldots, S_{n+1}$ be two sequences ($n \geq 0$) such that*

– $S_1 \xrightarrow{\{\alpha_1\}} \ldots \xrightarrow{\{\alpha_n\}} S_{n+1}$,
– $S_{n+1} \in T$, *and*
– $(r, g_{max}, S_{n+1}) \in g$.

Then $ext(G) \cup$ (2) admits an answer set in which

$$\texttt{does}(r, \alpha_1, 1) \quad \ldots \quad \texttt{does}(r, \alpha_n, n) \tag{3}$$

are exactly the positive instances of predicate `does`.

Proof. From Theorem 1 and the fact that the only answer set for $ext(G)$ coincides with its standard model, and given that none of $S_1, \ldots, S_n$ is terminal,[3] it follows that there is an answer set for $ext(G)$ augmented by the first three clauses in (2) that includes (3) as the only instances of predicate `does`. This is also an answer set for the entire program $ext(G) \cup$ (2) since

– $(r, \alpha_i, S_i) \in l$ for each $i = 1, \ldots, n$,
– $S_{n+1} \in T$, and
– $(r, g_{max}, S_{n+1}) \in g$ and either $S_n \notin T$ or $n = 0$.

Theorem 3. *Consider a GDL description G with semantics (R, S_1, T, l, u, g) such that $R = \{r\}$ for some r. If $\mathcal{A}$ is an answer set for $ext(G) \cup$ (2), then there exists $n \geq 0$ such that*

$$\texttt{does}(r, \alpha_1, 1) \quad \ldots \quad \texttt{does}(r, \alpha_n, n) \tag{4}$$

are exactly the positive instances of predicate `does` *in $\mathcal{A}$, and there are states $S_2, \ldots, S_{n+1}$ such that*

– $S_1 \xrightarrow{\{\alpha_1\}} \ldots \xrightarrow{\{\alpha_n\}} S_{n+1}$,
– $S_{n+1} \in T$, *and*
– $(r, g_{max}, S_{n+1}) \in g$.

Proof. Since $ext(G) \cup$ (2) contains only one clause with `does` in the head, the clauses labeled '1:' and '3:' in (2) ensure the existence of a finite sequence $\alpha_1, \ldots, \alpha_n$ (where $n \geq 0$) such that (4) are the only positive instances of `does` in $\mathcal{A}$. From Theorem 1 and the clauses labeled '2:' and '3:' in (2) it follows that there are states $S_2, \ldots, S_{n+1}$ such that

$$S_1 \xrightarrow{\{\alpha_1\}} \ldots \xrightarrow{\{\alpha_n\}} S_{n+1} \quad \text{and} \quad S_{n+1} \in T$$

Finally, the constraints labeled '4:' in (2) ensure that $(r, g_{max}, S_{n+1}) \in g$.

[3] Which follows from $S_1 \xrightarrow{\{\alpha_1\}} \ldots S_n \xrightarrow{\{\alpha_n\}} S_{n+1}$; cf. the conditions stated below (1).

5 Experimental Results

We have implemented the ASP-based endgame search for single-player games using CLINGO [14] as an off-the-shelf answer set solver in combination with the general game playing system described in [4]. The answer set programs are automatically generated from the game description. This requires to first compute the domain of moves as used in predicate `move_domain(M)` in clauses (2). The domains, or more precisely supersets thereof, of predicates and functions in a given game description can, in general, be computed by generating a dependency graph from the rules. The graph contains one node for every argument position of every function and predicate symbol, and one node for each function symbol itself (including each constant). An edge is added between an argument node and a function symbol node if the latter appears in the respective argument of a function or predicate in a rule of the game. An edge between two argument position nodes is added if there is a rule in the game in which the same variable appears in both arguments. Argument positions in each connected component of the graph share a domain, and the constants and function symbols in the connected components are the domain elements. Specifically, we take as the overall domain of the moves that of the second argument of `legal`.

Figure 2 shows a small excerpt of the dependency graph for our running example game. The first argument of *jump* and the first argument of *cell* are connected because they share variable X in the game rule

```
next(cell(X,nocoin)) :- does(player,jump(X,Y)).
```

Similarly, the second argument of *jump* and the first argument of *cell* are connected because they share variable X in the game rule

```
next(cell(X,double)) :- does(player,jump(Y,X)).
```

The `init` rules along with the *succ* facts then imply that the first and second argument of *jump* and the first argument of *cell* all share the domain $\{a, \ldots, h\}$. The dependency graph also contains a link from the second argument of `legal` to function *jump*. This is a consequence of the clause

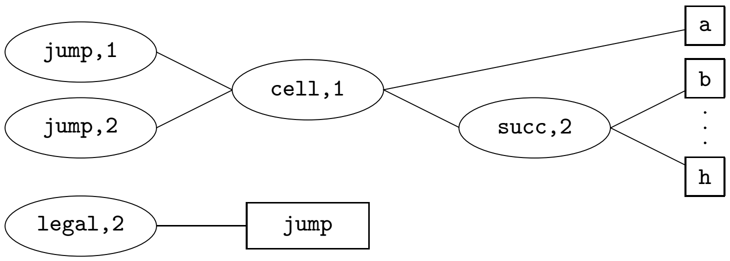


Fig. 2. An excerpt of a dependency graph for calculating domains of functions and predicates. Ellipses denote argument *positions* of functions or predicates, respectively, while rectangles denote function symbols themselves (including constants).

Table 2. Results of a complete search for a variety of single-player games, with times given in seconds. Experiments were run on a 1.66GHz processor. Symbol $*$ indicates that the ASP system was aborted because it used more than 1 GB RAM, while Fluxplayer was aborted after not finding a complete solution within 30 minutes. (We enforced these rather strict limits in view of practical play, as endgame search is only one of several tasks during the contemplation phase or when deciding on the next move.)

Game	ASP search time	Fluxplayer	Solution length
asteroids	0.11	2.60	10
asteroidsparallel	0.72	97.19	10
blocksworldparallel	9.78	1.80	3
duplicatestatelarge	0.16	17.54	14
eightpuzzle	77.11	$*$	30
factoringaperturescience	0.28	3.86	4
factoringdeterminate	0.05	2.14	5
knightmove (8×8-board)	$*$	$*$	(64)
knightstour (6×5-board)	7.33	117.70	30
peg	$*$	9.53	31
ruledepthlinear	0.19	9.60	49
ruledepthquadratic	$*$	12.70	44
statespacelarge	0.27	343.38	14
wargame01	$*$	39.53	48

```
legal(P,jump(X,Y)) :- true(cell(X,single)), ... .
```

Hence, the domain of moves in this game contains every possible instance of $jump(X, Y)$ with $X, Y \in \{a, \ldots, h\}$.

In addition to the domain of moves, the intended maximal depth for an endgame search defines the domain for the additional time variable which is used in the temporally extended program as well as in the clauses (2). Given domain restrictions for all variables, any existing answer set programming system can be employed to carry out the endgame search for single-player games.

We conducted experiments with all single-player games that were available at the time of publication through the online repository `games.stanford.edu`. Most of these games were used in past AAAI General Game Playing Competitions [6]. For the sake of reproducibility, we only report on the experiments where ASP search was applied straight away to the initial position. The results are shown in Table 2. It turns out that most of the games can actually be solved completely in reasonable, often very short, time that would have allowed a general game player to pre-compute a winning strategy during the "contemplation" phase. The results also show that in most cases the ASP search clearly outperforms the previously used forward search in our Fluxplayer—which over the past competitions proved to be the overall best performing system on single-player games [4].

6 Conclusion

We have shown how any game specified in the general Game Description Language can be mapped onto a normal logic program with time, and we have proved the correctness of this mapping against the formal game semantics for GDL. On this basis, we have illustrated how Answer Set Programming can be successfully deployed as a search method to assist general game players with tackling single-player games. Due to the closeness between GDL and ASP—in both syntax and semantics—the latter is ideally suited for performing blind search as part of a general strategy to solve single-player games.

We have substantiated this claim by reporting on experimenting with single-player games from past AAAI General Game Playing Competitions. The results show that actually most of these games could have been solved right from the start by an off-the-shelf ASP system. As games become more complex, they cannot be expected to be tackled by blind search alone, but still an ASP-based component constitutes a valuable addition to any general game playing system when it comes to performing depth-limited endgame search during game play.

The main limitation for the deployment of current ASP systems is the required grounding of the program, which easily becomes too large to be of practical use. Fortunately, a general game playing system can use the sizes of the domains for each variable to give an estimate of the size of the fully grounded, temporally extended game rules. On this basis, the system can easily decide on the fly whether or not it should execute the ASP-based endgame search in the current position.

For future work, we intend to investigate ways to use ASP for endgame search in multi-player games on the basis of Theorem 1. This will not be a straightforward extension of the method presented in this paper, because a single answer set determines a winning joint strategy for all players rather than a winning strategy against one or more opponents. Another direction of future work consists in investigating whether the very recently developed method of first-order Answer Set Programming [20] can be used to help a general game playing system perform endgame search in cases where grounding is too expensive.

Acknowledgment

This research was partially supported by *Deutsche Forschungsgemeinschaft* under Contract TH 541/16-1.

References

1. Pell, B.: Strategy Generation and Evaluation for Meta-Game Playing. Ph.D thesis, Trinity College, University of Cambridge (1993)
2. Kuhlmann, G., Dresner, K., Stone, P.: Automatic heuristic construction in a complete general game player. In: Proceedings of the AAAI National Conference on Artificial Intelligence, Boston, pp. 1457–1462. AAAI Press, Menlo Park (2006)
3. Clune, J.: Heuristic evaluation functions for general game playing. In: Proceedings of the AAAI National Conference on Artificial Intelligence, Vancouver, pp. 1134–1139. AAAI Press, Menlo Park (2007)
4. Schiffel, S., Thielscher, M.: Fluxplayer: A successful general game player. In: Proceedings of the AAAI National Conference on Artificial Intelligence, Vancouver, pp. 1191–1196. AAAI Press, Menlo Park (2007)

5. Finnsson, H., Björnsson, Y.: Simulation-based approach to general game playing. In: Proceedings of the AAAI National Conference on Artificial Intelligence, Chicago, pp. 259–264. AAAI Press, Menlo Park (2008)
6. Genesereth, M., Love, N., Pell, B.: General game playing: overview of the AAAI competition. AI Magazine 26, 62–72 (2005)
7. Love, N., Hinrichs, T., Haley, D., Schkufza, E., Genesereth, M.: General Game Playing: Game Description Language Specification. Technical Report LG–2006–01, Stanford Logic Group, Computer Science Department, Stanford University (2006), `games.stanford.edu`
8. Schiffel, S., Thielscher, M.: A multiagent semantics for the game description language. In: Filipe, J., Fred, A., Sharp, B. (eds.) Proceedings of the International Conference on Agents and Artificial Intelligence (ICAART), Porto. Springer, Heidelberg (2009)
9. Apt, K., Blair, H.A., Walker, A.: Towards a theory of declarative knowledge. In: Minker, J. (ed.) Foundations of Deductive Databases and Logic Programming, pp. 89–148. Morgan Kaufmann, San Francisco (1987)
10. Müller, M., Gasser, R.: Experiments in computer Go endgames. In: Nowakowski, R. (ed.) Games of No Chance, pp. 273–284. Cambridge University Press, Cambridge (1996)
11. Kautz, H., Selman, B.: Planning as Satisfiability. In: Neumann, B. (ed.) Proceedings of the European Conference on Artificial Intelligence (ECAI), pp. 359–363. John Wiley & Sons, Ltd., Chichester (1992)
12. Ernst, M.D., Millstein, T.D., Weld, D.S.: Automatic SAT-compilation of planning problems. In: Pollack, M.E. (ed.) Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI), Nagoya, Japan, pp. 1169–1176. Morgan Kaufmann, San Francisco (1997)
13. Lifschitz, V.: Answer set programming and plan generation. Artificial Intelligence 138, 39–54 (2002)
14. Potsdam Answer Set Solving Collection (2008), `potassco.sourceforge.net`
15. Gelfond, M.: Answer sets. In: van Harmelen, F., Lifschitz, V., Porter, B. (eds.) Handbook of Knowledge Representation, pp. 285–316. Elsevier, Amsterdam (2008)
16. van Gelder, A.: The alternating fixpoint of logic programs with negation. In: Proceedings of the 8th Symposium on Principles of Database Systems, ACM SIGACT-SIGMOD, pp. 1–10 (1989)
17. Lloyd, J., Topor, R.: A basis for deductive database systems II. Journal of Logic Programming 3, 55–67 (1986)
18. Giunchiglia, E., Lee, J., Lifschitz, V., McCain, N., Turner, H.: Nonmonotonic causal theories. Artificial Intelligence 153, 49–104 (2004)
19. Niemelä, I., Simons, P., Soininen, T.: Stable model semantics of weight constraint rules. In: Gelfond, M., Leone, N., Pfeifer, G. (eds.) LPNMR 1999. LNCS, vol. 1730, pp. 317–331. Springer, Heidelberg (1999)
20. Lee, J., Meng, Y.: On loop formulas with variables. In: Brewka, G., Doherty, P., Lang, J. (eds.) Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning (KR), Sydney, pp. 444–453 (2008)

Finding Similar or Diverse Solutions in Answer Set Programming*

Thomas Eiter[1], Esra Erdem[2], Halit Erdoğan[2], and Michael Fink[1]

[1] Institute of Information Systems, Vienna University of Technology, Vienna, Austria
{eiter,fink}@kr.tuwien.ac.at

[2] Faculty of Engineering and Natural Sciences, Sabancı University, Istanbul, Turkey
esraerdem@sabanciuniv.edu, halit@su.sabanciuniv.edu

Abstract. We study finding similar or diverse solutions of a given computational problem, in answer set programming, and introduce offline methods and online methods to compute them using answer set solvers. We analyze the computational complexity of some problems that are related to finding similar or diverse solutions, and show the applicability and effectiveness of our methods in phylogeny reconstruction.

Keywords: similar/diverse solutions, answer set programming, phylogenies.

1 Introduction

Although, for many computational problems, the main concern is to find a best solution (e.g., a most preferred product configuration, a shortest plan, a most parsimonious phylogeny), for some problems, computing a subset of good solutions that are diverse or similar may be desirable. For instance, in product configuration, one could be interested in obtaining several diverse configurations of a product instead of checking all possible configurations, to pick one. In planning, it may be desirable to compute a set of plans that are similar to each other, so that, when the plan that is being executed fails, one can switch to a most similar one. Motivated by such applications, we study the problem of computing similar or diverse solutions in answer set programming (ASP), and then show the applicability of our approach to another interesting problem: phylogeny reconstruction (i.e., computing leaf-labeled trees, called phylogenies, to model the evolutionary history of a set of species).

Problems related to computing similar or diverse solutions have been studied in the context of propositional logic [2], and constraint programming [12,13]. On the other hand, although there are many appealing ASP applications (e.g., product configuration [22], planning [15], phylogeny reconstruction [4]), for which finding similar/diverse solutions could be useful, such problems have not been studied in ASP. The methods we develop in this paper fulfill this need in ASP.

Phylogeny reconstruction is important for research areas as disparate as genetics, historical linguistics, zoology, anthropology, archaeology. For example, a phylogeny

* This work has been supported by TUBITAK Grant 107E229 and the Wolfgang Pauli Institute, Vienna.

of parasites may help zoologists to understand the evolution of human diseases [6]; a phylogeny of languages may help scientists to better understand human migrations [23]. For a given set of taxonomic units, some existing phylogenetic systems, like that of [5,4], generate more than one phylogeny that explains the evolutionary relationships between the given taxonomic units. There are phylogenetic systems that compute a summary of these phylogenies (a consensus tree [1] or a supertree [21]). However, in such cases, especially when there are too many phylogenies computed by a system, an expert needs to compare these phylogenies in detail, by analyzing the similar/diverse ones with respect to some distance measure, to pick the most plausible ones. Although there are precisely defined measures to find the distance between them [17,3,20,14], there is no phylogenetic system that helps experts to analyze phylogenies by comparing them. The methods we develop in this paper fulfill this need in phylogenetics.

In particular, the main contributions of this paper are as follows.

- We describe two kinds of computational problems related to finding similar/diverse solutions of a given problem, in the context of ASP (Section 2). Both kinds of problems take as input an ASP program P that describes a problem, a distance measure Δ that maps a set of solutions of the problem to a nonnegative integer, and two nonnegative integers n and k. One problem asks for a set S of size n that contains k-similar (resp. k-diverse) solutions, i.e., $\Delta(S) \leq k$ (resp. $\Delta(S) \geq k$); the other problem asks, given a set S of n solutions, for a k-close (k-distant) solution s ($s \notin S$), i.e., $\Delta(S \cup \{s\}) \leq k$ (resp. $\Delta(S \cup \{s\}) \geq k$).
- We study the computational complexity of these problems establishing completeness results under reasonable assumptions for the problem parameters (Section 3).
- We introduce an offline method to compute a set of n k-similar (or k-diverse) solutions to a given problem, by computing all solutions in advance and then using some clustering methods to find the similar (diverse) solutions (Section 4).
- We introduce three online methods to compute a set of n k-similar (or k-diverse) solutions to a given problem (Section 5). Online Method 1 reformulates the given program to compute n-distinct solutions and formulates the distance function as an ASP program, so that all n k-similar (k-diverse) solutions can be extracted from an answer set for the union of these ASP programs. Online Method 2 does not modify the given ASP program, but formulates the distance function as an ASP program, so that a k-close (or k-distant) solution can be extracted from an answer set for the union of these ASP programs and a previously computed solution; by iteratively computing k-close (k-distant) solutions, we can compute online a set of n k-similar (or k-diverse) solutions. Online Method 3 does not modify the given program, and does not formulate the distance function as an ASP program, but it modifies the ASP solver, in our case CLASP [10], to compute all n k-similar (or k-diverse) solutions at once.
- We illustrate the applicability of these approaches on the phylogeny reconstruction problem, by defining new distance measures for a set of phylogenies (Section 6), by describing how the offline method and the online methods are applied to find similar/diverse phylogenies (Section 7). After that, we compare the efficiency and effectiveness of these methods on the family of Indo-European languages studied in [4] (Section 8).

ASP programs mentioned below are presented in an extended version `http://people.sabanciuniv.edu/esraerdem/papers/iclp09-extended.pdf`.

2 Computational Problems

We are interested in the following two sorts of problems related to computation of a diverse/similar collection of solutions:

n k-SIMILAR SOLUTIONS (resp.n k-DIVERSE SOLUTIONS)
Given an ASP program $\mathcal{P}$ that formulates a computational problem P, a distance measure Δ that maps a set of solutions for P to a nonnegative integer, and two nonnegative integers n and k, decide whether a set S of n solutions for P exists such that $\Delta(S) \leq k$ (resp. $\Delta(S) \geq k$).

k-CLOSE SOLUTION (resp.k-DISTANT SOLUTION)
Given an ASP program $\mathcal{P}$ that formulates a computational problem P, a distance measure Δ that maps a set of solutions for P to a nonnegative integer, a set S of solutions for P, and a nonnegative integer k, decide whether a solution s ($s \notin S$) for P exists such that $\Delta(S \cup \{s\}) \leq k$ (resp. $\Delta(S \cup \{s\}) \geq k$).

For instance, suppose that $\mathcal{P}$ describes the phylogeny reconstruction problem for Indo-European languages. Then finding three diverse phylogenies is an instance of the former problem. On the other hand, if we already have picked two phylogenies, then finding another phylogeny that differs from these two is an instance of the latter.

The first kind of problems above has two parameters, n and k, so we can fix one and try to minimize (resp. maximize) the distance between solutions to find the most similar (resp. diverse) solutions.

MAXIMAL k-SIMILAR SOLUTIONS (resp.MAXIMAL k-DIVERSE SOLUTIONS)
Given an ASP program $\mathcal{P}$ that formulates a computational problem P, a distance measure Δ that maps a set of solutions for P to a nonnegative integer, and a nonnegative integer k, find a maximal set S of solutions for P such that $\Delta(S) \leq k$ (resp. $\Delta(S) \geq k$) exists.

n MOST SIMILAR SOLUTIONS (resp.n MOST DIVERSE SOLUTIONS)
Given an ASP program $\mathcal{P}$ that formulates a computational problem P, a distance measure Δ that maps a set of solutions for P to a nonnegative integer, and a nonnegative integer n, find a set S of n solutions for P with the minimum (resp. maximum) distance $\Delta(S)$.

Similarly, in the second class of problems, we can try to minimize (resp. maximize) the distance k between a solution and a set of solutions, to find the most close (resp. distant) solution.

MOST CLOSE SOLUTION (resp.MOST DISTANT SOLUTION)
Given an ASP program $\mathcal{P}$ that formulates a computational problem P, a distance measure Δ that maps a set of solutions for P to a nonnegative integer, and a set S of solutions for P, find a solution s ($s \notin S$) for P with the minimum (resp. maximum) distance $\Delta(S \cup \{s\})$.

Table 1. Complexity results for computing similar solutions.

#	Problem	Complexity
1	n k-SIMILAR SOLUTIONS	NP
2	k-CLOSE SOLUTION	NP
3	MAXIMAL k-SIMILAR SOLUTIONS	FNP//log
4	n MOST SIMILAR SOLUTIONS	$\mathrm{FP}^{\mathrm{NP}}$ (FNP//log)
5	MOST CLOSE SOLUTION	$\mathrm{FP}^{\mathrm{NP}}$ (FNP//log)
6	k-CLOSE SET	NP

We can generalize k-CLOSE SOLUTION (resp.k-DISTANT SOLUTION) problems to sets of solutions:

k-CLOSE SET (resp.k-DISTANT SET)
Given an ASP program $\mathcal{P}$ that formulates a computational problem P, a distance measure Δ that maps a set of solutions for P to a nonnegative integer, a set S of solutions for P, and a nonnegative integer k, decide whether a set S' of solutions for P exists such that $|\Delta(S) - \Delta(S')| \leq k$ (resp. $|\Delta(S) - \Delta(S')| \geq k$).

3 Complexity Results

In this section, we turn our attention to the computational complexity of the problems presented in Section 2. In order to do so, we first have to make some reasonable assumptions on some of the problem parameters.

For the remainder of this section, let the ASP program $\mathcal{P}$ that formulates a computational problem P, be a propositional normal logic program. We assume that the given number n of different solutions to consider (respectively the size of the set S) in instances of the problems n k-SIMILAR SOLUTIONS and n MOST SIMILAR SOLUTIONS is polynomial in the size of the input.

Furthermore, we consider distance measures Δ that map a set of solutions for P to a nonnegative integer (which is usually implicitly done when real values have to be represented for computation). As for computing $\Delta(S)$ for a set of solutions S, in general we assume that deciding whether $\Delta(S) \leq k$ for a given k is in NP. Moreover, we assume that the value of $\Delta(S)$ is bounded by an exponential in the size of S (and thus has polynomially many bits in the size of S).

Under these assumptions, the computational complexity (cf. [18] for a background on the subject) of the problems concerning the computation of similar or diverse solutions we are interested in, is given in Table 1. All entries are completeness results (under usual reductions) and hardness holds even if $\Delta(S)$ is computable in polynomial time. Moreover, the results are the same for the 'symmetric' problems, i.e., when SIMILAR is replaced with DIVERSE, and CLOSE is replaced with DISTANT, respectively.

Membership for problem n k-SIMILAR SOLUTIONS (resp. n k-DIVERSE SOLUTIONS) follows from the fact that we can guess not only a candidate set S (since S is polynomially bounded) but also a witness for $\Delta(S) \leq k$ (resp. $\Delta(S) \geq k$), and check in polynomial time whether every $s \in S$ is a solution and that $\Delta(S) \leq k$ (resp. $\Delta(S) \geq k$). For hardness, one simply reduces answer-set existence for normal, propositional programs to this problem, which is an NP-complete problem. However, hardness holds also for nodal distance of trees (a distance measure introduced in Section 6 for comparing phylogenies) encoded in a program (which naturally uses auxiliary atoms).

Theorem 1. *Problem n k-SIMILAR SOLUTIONS (resp. n k-DIVERSE SOLUTIONS) is* NP*-complete. Hardness holds even if $\Delta(S)$ is computable in polynomial time.*

For a hardness result resorting to partial Hamming distance confer [2]. By similar arguments we obtain NP-completeness for problem k-CLOSE SOLUTION (resp. k-DISTANT SOLUTION).

Theorem 2. *Problem k-CLOSE SOLUTION (resp. k-DISTANT SOLUTION) is* NP*-complete. Hardness holds even if $\Delta(S)$ is computable in polynomial time.*

When looking for maximal (wrt. subset inclusion) solutions, we face a function problem; here we assume that any S of size larger than n is clipped to any subset S' of S of size n. In particular, MAXIMAL k-SIMILAR SOLUTIONS (resp. MAXIMAL k-DIVERSE SOLUTIONS) is solvable in $\mathrm{FNP}//\log$. Intuitively, $\mathrm{FNP}//\log$ is the class of function problems solvable in polynomial time using a nondeterministic Turing Machine with output tape that may consult once an oracle that computes the optimal value of an optimization problem solvable in NP. A requirement is that this value has logarithmically many bits in the size of the input (see, e.g., [7,9] for more information on $\mathrm{FNP}//\log$ and other function classes used in this section).

Membership can be shown by computing the cardinality of a maximal set of solutions S using the oracle. Note that since $|S|$ is polynomially bounded in the size of the input, it has logarithmically many bits as required. Then, one can nondeterministically compute a set S of respective size together with a witness for $\Delta(S) \leq k$, and check in polynomial time that this is indeed the case.

Hardness can be shown, e.g., for $\Delta(S)$ that takes the maximal (resp. minimal) Hamming distance between answer sets in S on a subset of the atoms; note that such a *partial Hamming distance* is a natural measure for problem encodings, where the disagreement on output atoms is measured. This measure is not unrelated to the ones introduced for comparing phylogenies in Section 6; one can polynomially reduce nodal distance to partial Hamming distance, and vice versa also partial Hamming distance to nodal distance of trees (allowing auxiliary atoms in the LP encoding).

Theorem 3. *Problem* MAXIMAL k-SIMILAR SOLUTIONS *(resp.* MAXIMAL k-DIVERSE SOLUTIONS*) is* $\mathrm{FNP}//\log$*-complete. Hardness holds even if $\Delta(S)$ is computable in polynomial time.*

$\mathrm{FP}^{\mathrm{NP}}$-membership of n MOST SIMILAR SOLUTIONS (resp. n MOST DIVERSE SOLUTIONS) is obtained by first using the NP-oracle to compute the minimum distance using binary search (deciding polynomially many n k-SIMILAR SOLUTIONS problems). Then, the oracle is used to compute S in polynomial time. Hardness follows from a reduction of the Traveling Salesman Problem (TSP). Notably, if the distances are polynomial in the size of the input, i.e., if the value of $\Delta(S)$ is polynomially bounded in the size of S, then the problem is $\mathrm{FNP}//\log$-complete.

Theorem 4. *Problem n MOST SIMILAR SOLUTIONS (resp. n MOST DIVERSE SOLUTIONS) is* $\mathrm{FP}^{\mathrm{NP}}$*-complete, and* $\mathrm{FNP}//\log$*-complete if the value of $\Delta(S)$ is polynomial in the size of S. Hardness holds even if $\Delta(S)$ is computable in polynomial time.*

Proceeding similarly as before, completeness for $\mathrm{FP}^{\mathrm{NP}}$(resp. $\mathrm{FNP}//\log$ if $\Delta(S)$ is small) is obtained for MOST CLOSE SOLUTION (and for MOST DISTANT SOLUTION).

Theorem 5. *Problem* MOST CLOSE SOLUTION *(resp.* MOST DISTANT SOLUTION*) is* $\mathrm{FP}^{\mathrm{NP}}$*-complete, and* $\mathrm{FNP}//\log$*-complete if the value of* $\Delta(S)$ *is polynomial in the size of* S*. Hardness holds even if* $\Delta(S)$ *is computable in polynomial time.*

For the generalization of k-CLOSE SOLUTION (resp. of k-DISTANT SOLUTION) to sets, namely k-CLOSE SET (resp. k-DISTANT SET), NP-completeness holds by similar arguments as for the former problem(s).

Theorem 6. *Problem* k-CLOSE SET *(resp.* k-DISTANT SET*) is* NP*-complete. Hardness holds even if* $\Delta(S)$ *is computable in polynomial time.*

4 Offline Methods

We introduce an offline method to compute a set of n k-similar (resp. k-diverse) solutions to a given problem, by computing all solutions in advance and then using some clustering methods to find the similar (diverse) solutions. The idea is to make clusters of n solutions, measure the distance of the set of solutions in each cluster, and pick the cluster whose distance is less (resp. greater) than k.

We can solve this problem by means of a graph problem: build a complete graph G whose nodes correspond to solutions and edges are labeled by distances between the corresponding solutions; and decide whether there is a clique C of size n in G whose weight (i.e., the distance of the set of solutions denoted by the clique) is less than k (resp. greater than k). The set of vertices in the clique represents n k-similar phylogenies. Such a clique can be computed using ASP, or one of the existing exact/approximate algorithms.

5 Online Methods

We introduce three online methods to compute a set of n k-similar (or k-diverse) solutions to a given problem P, given an ASP program $\mathcal{P}$ that represents P and a distance function Δ that maps a set of solutions of P to a nonnegative integer.

Online Method 1 (Fig. 1) reformulates the given program $\mathcal{P}$ to compute n-distinct solutions, formulates the distance function Δ as an ASP program $\mathcal{D}$, and formulates constraints on the distance function as an ASP program $\mathcal{C}$, so that all n k-similar (k-diverse) solutions can be extracted from an answer set for the union of these ASP programs, $\mathcal{P} \cup \mathcal{D} \cup \mathcal{C}$. Such a reformulation of $\mathcal{P}$ can be obtained in two stages. First, we copy every rule of the program n times: the i'th copy of the rule is obtained from r by replacing every atom $p(t_1, t_2, ..., t_m)$ in r with $p(i, t_1, t_2, ..., t_m)$. Now we have a program that computes n solutions to the problem P. Then, we add a constraint to ensure that no two solutions are same.

Online Method 2 (Fig. 2) does not modify the given ASP program $\mathcal{P}$, but formulates the distance $\Delta(S)$ of a given set S of solutions as an ASP program $\mathcal{D}$, and constraints

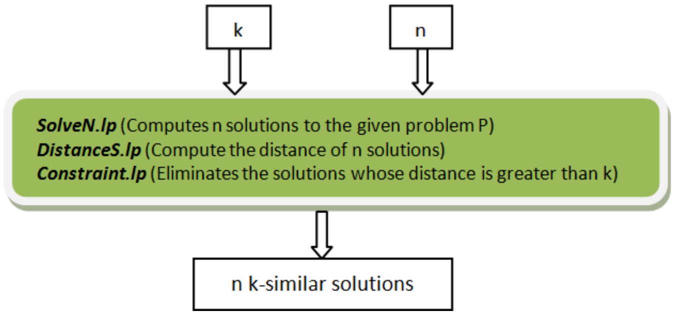

k
n
SolveN.lp (Computes n solutions to the given problem P)
DistanceS.lp (Compute the distance of n solutions)
Constraint.lp (Eliminates the solutions whose distance is greater than k)
n k-similar solutions

Fig. 1. Computing n k-similar solutions, with Online Method 1

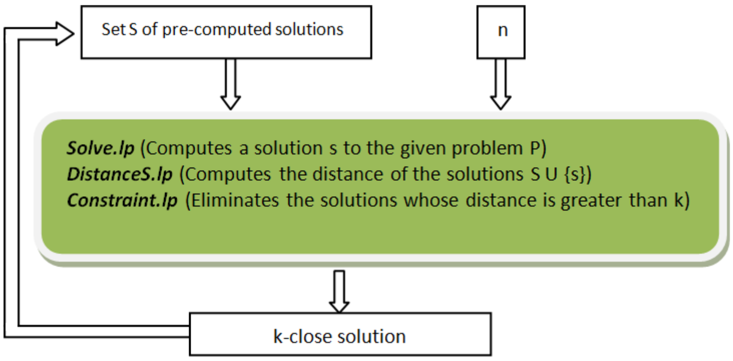

Set S of pre-computed solutions
n
Solve.lp (Computes a solution s to the given problem P)
DistanceS.lp (Computes the distance of the solutions S U {s})
Constraint.lp (Eliminates the solutions whose distance is greater than k)
k-close solution

Fig. 2. Computing n k-similar solutions, with Online Method 2. Initially $S = \emptyset$. In each run, a solution is computed and added to S, until $|S| = n$. The distance function and the constraints in the program ensures that when we add the computed solution to S, the set stays k-similar.

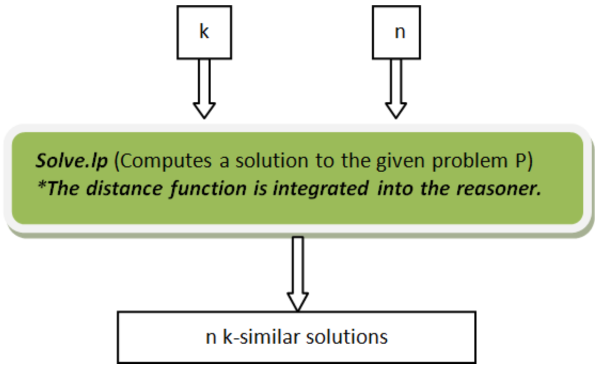

k
n
Solve.lp (Computes a solution to the given problem P)
*The distance function is integrated into the reasoner.
n k-similar solutions

Fig. 3. Computing n k-similar solutions, with Online Method 3. We implement the distance function into the ASP reasoner, so that the ASP reasoner becomes biased to compute similar solutions. Each computed solution is stored by the reasoner until a set of n k-similar solutions is computed.

on the distance function as an ASP program $\mathcal{C}$, so that a k-close (or k-distant) solution can be extracted from an answer set for $\mathcal{P} \cup \mathcal{D} \cup \mathcal{C}$. By iteratively computing a k-close (k-distant) solution, we can compute online a set of n k-similar (or k-diverse) solutions.

Online Method 3 (Fig. 3) does not modify the given program, and does not formulate the distance function as an ASP program, but it modifies the ASP solver CLASP to compute all n k-similar (or k-diverse) solutions at once.

6 Distance Measures for Similar or Diverse Phylogenies

A *phylogenetic tree (phylogeny)* for a set of taxonomic units is a finite rooted leaf-labeled binary tree. To compare a set of phylogenies, and analyze the similar or diverse ones in this set, we can measure the distance of a set of phylogenies by some function Δ. In the following, we introduce a distance function to measure the similarity/diversity of a set of phylogenies, in terms of a distance function for two phylogenies. We present the trees in the Newick format, where the sister subtrees are enclosed by parentheses.

Two distance functions for two phylogenies Among the existing functions for measuring the distance between two trees [17,3,20,14], we consider the distance function of [3] based on the nodal distances in trees. The *nodal distance* $ND_T(x, y)$ between two leaves x and y in a tree T is the number of edges contained in the shortest path from one leaf to the other. For example, consider the tree $(a, (b, c))$; the nodal distance between a and b is 3, whereas the nodal distance between b and c is 2. Intuitively, the nodal distance between two leaves in a tree represents the degree of their relationship in that tree. After defining the nodal distance, [3] measures the distance $D_n(T, T')$ between two leaf-labeled trees T and T', both with the same set L of leaves, as follows:

$$D_n(T, T') = \sum_{(x,y) \in L} |ND_T(x, y) - ND_{T'}(x, y)|.$$

The difference of the nodal distances of two leaves in two trees represents the contribution of these leaves to the distance between the trees. Let $T_1 = (a, (b, c))$ and $T_2 = (c, (a, b))$ be two trees. In order to compute the distance between T_1 and T_2, we compute the nodal distances of the pairs $\{a, b\}$, $\{a, c\}$ and $\{b, c\}$ for both trees and take the sum of the differences. In this case the distance between T_1 and T_2 is 2.

The second distance function we consider is introduced specifically for languages, based on our discussions with the historical linguist Don Ringe. For each vertex x of a tree $\langle V, E\rangle$, let $desc(x)$ denote its descendants and $depth(x)$ its depth. To define the similarity of two phylogenies $\langle V, E\rangle$ and $\langle V', E'\rangle$, let us first define the similarity of two vertices $v \in V$ and $v' \in V'$: $f(v, v') = 1$ if $desc(v) \neq desc(v')$; and $f(v, v') = 0$ otherwise. Let *weight* be a function mapping every depth to a nonnegative integer. Then we can define the similarity of two trees $T = \langle V, E\rangle$ and $T' = \langle V', E'\rangle$, with the roots R and R' respectively, at depth i ($0 \leq i \leq \min\{\max_{v \in V} depth(v), \max_{v' \in V'} depth(v')\}$), by the following measure:

$$g(0, T, T') = weight(0) \times f(R, R')$$
$$g(i+1, T, T') = g(i, T, T') + weight(i+1) \times \textstyle\sum_{x \in V, y \in V', depth(x) = depth(y) = i+1} f(x, y)$$

and the similarity of two trees as follows:

$$D_l(T,T') = g(\min\{\max_{v\in V} \mathit{depth}(v), \max_{v'\in V'} \mathit{depth}(v')\}, T, T').$$

For instance, for $T_1 = (a,(b,(c,(d,e))))$ and $T_2 = (a,(d,(c,(b,e))))$, considering that $\mathit{weight}(i) = 4-i$, $D_l(T_1,T_2) = 3\times2+2\times4+1\times3 = 17$. The idea is to assign bigger weights to smaller depths so that two phylogenies are more similar if the diversifications closer to the root are more similar. This is motivated by that reconstructing the evolution of languages closer to the root is more important for historical linguists.

A distance function for a set of phylogenies We define a distance function for measuring the distance of a set S of phylogenies, based on a distance function D for two phylogenies: for similarity (resp. diversity) we take the maximum (resp. minimum) of the distances between pairs of phylogenies in S

$$\Delta_D(S) = \max\{D(T_1,T_2) \mid T_1,T_2 \in S\}$$

In the following, we show the applicability of the offline methods and online methods, with the distance functions Δ_{D_n} and Δ_{D_l}.

7 Computation of Similar or Diverse Phylogenies

We can find n k-similar (resp. k-diverse) phylogenies for a set of taxonomic units, with an offline method as described in Section 4. Consider, for instance, a family of languages as the taxonomic units. With the approach of [4], we can compute all the phylogenies for a given set of languages. Then we build a complete graph G whose nodes denote these phylogenies, and the edges are labeled by the distances between phylogenies. Then we can find a clique of size n in G, such that the distance of the set of phylogenies denoted by this clique is less than or equal to k, as follows: remove each edge in G whose label is greater than k; and, ignoring the weights of the edges in the resulting graph, find a clique of size n. The set of vertices in the clique represents n k-similar phylogenies for the given set of taxonomic units.

In the online methods, we consider the ASP program `phylogeny-improved.lp` described in [4], to reconstruct phylogenies.

Online Method 1 suggests finding n k-similar (resp. k-diverse) phylogenies, by reformulating the given ASP program for phylogeny reconstruction, and using an answer set solver to compute all these solutions. A reformulation of `phylogeny-improved.lp`, as suggested by the first online method, can be obtained as follows:

1. We specify the number of solutions: `solution(1..n).`
2. In each rule of the program, we replace each atom `p(T1,T2,...,Tm)` (except the ones specifying the input, like atoms describing the leaves, the labels of the leaves, characters, and states of characters) with `p(N,T1,T2...,Tm)`, and add to the body `solution(N).`
3. Now we have a program that computes n phylogenies. To ensure that they are distinct, for each atom specifying a solution, in this case atoms describing the edges of a phylogeny, we add the rules

Algorithm 1. CLASP

```
Input: An ASP program $\Pi$
Output: An answer set $A$ for $\Pi$
  $A \leftarrow \emptyset$   // current assignment of literals
  $\triangledown \leftarrow \emptyset$   // set of conflicts
  while no answer set found do
    UNIT-PROPAGATION($\Pi, A, \triangledown$)   // propagate according to the current assignment and conflicts, and update the current assignment
    if there is a conflict in the current assignment then
      RESOLVE-CONFLICT($\Pi, A, \triangledown$)   // learn and update the conflict set and do backtracking
    else
      if current assignment does not yield an answer set then
        SELECT($\Pi, A, \triangledown$)   // select a literal to continue search
      else
        return A
      end if
    end if
  end while
```

```
different(S1,S2) :- edge(S1,X1,Y), edge(S2,X2,Y),
   vertex(X2;X1;Y), solution(S1;S2), S1 != S2, X1 != X2.
:- not different(S1,S2), solution(S1;S2), S1 != S2.
```

Online Method 2 suggests finding n k-similar (resp. k-diverse) phylogenies, by iteratively computing a k-close (resp. k-distant) phylogeny. Here we implement a perl script that calls the ASP solver repeatedly, with the phylogeny reconstruction program `phylogeny-improved.lp` and a distance function program, until we compute all n k-similar solutions.

Online Method 3 suggests finding n k-similar (resp. k-diverse) phylogenies, by modifying the ASP solver. Consider for instance the answer set solver CLASP [10]. CLASP does a conflict-driven DPLL-like [8,16] Branch & Bound search to find an answer set (solution) of the program: at each level, it does propagation followed by backtracking or selection of new literals according to the current conflicts. A rough working principle of CLASP is shown in Algorithm 1. As can be seen, CLASP goes through three main steps to find an answer set. In the UNIT-PROPAGATION step, it decides the literals that have to be included in the answer set due to the current assignment and conflicts. In the RESOLVE-CONFLICT step, it tries to resolve the conflicts encountered in the previous step. If there is a conflict, then CLASP learns it and does backtracking to an appropriate level. If there is no conflict and the currently selected literals do not represent an answer set, then, in SELECT, CLASP selects a new literal (based on BERKMIN's heuristic [11]) to continue search.

We can modify CLASP as in Algorithm 2, to compute n k-similar phylogenies. The modified solver, CLASP-NK, has some additional procedures: DISTANCE-ANALYZE identifies the partial phylogeny formed by the currently selected literals, and then computes a lower bound for the distance between a phylogeny that contains this partial phylogeny and the previously computed full phylogenies. Computing an exact lower bound requires enumerating all possible completions of the partial phylogeny, so we

Algorithm 2. CLASP-NK

```
Input: An ASP program $\Pi$, nonnegative integers $n$, and $k$, and a set $C$ of atoms considered in
    computation of the distance function
Output: A set $X$ of $n$ phylogenies that are $k$ similar ($n$ $k$-similar phylogenies)
  $X \leftarrow \emptyset$   // computed phylogenies
  $A \leftarrow \emptyset$   // current assignment of literals
  $\bigtriangledown \leftarrow \emptyset$   // set of conflicts
  while $|X| < n$ do
    PartialSolution $\leftarrow$ CurSelCon$(A, C)$   // the atoms that are marked as considered and that
    are currently selected constitute a partial solution
    $d \leftarrow$ DISTANCE-ANALYZE$(X,$PartialSolution$)$   // compute a lower bound for the distance
    between partial solution and previously computed phylogenies
    if $d > k$ then
      RESOLVE-CONFLICT$(\Pi, A, \bigtriangledown)$
    end if
    UNIT-PROPAGATION$(\Pi, A, \bigtriangledown)$
    if there is a conflict in the current assignment then
      RESOLVE-CONFLICT$(\Pi, A, \bigtriangledown)$
    else
      if current assignment does not yield an answer set then
        SELECT$(\Pi, A, \bigtriangledown)$
      else
        $X \leftarrow X \cup A$
        $A \leftarrow \emptyset$   // start searching for a new solution
      end if
    end if
  end while
  return X
```

compute an approximate lower bound by a heuristic function $LB(T, T')$ that estimates the distance (from below) between a complete phylogeny T and a complete phylogeny that contains a partial phylogeny T' with leaves L':

$$LB(T,T') = \sum_{(x,y)\in L'} |ND_T(x,y) - ND_{T'}(x,y)|.$$

Since this heuristic function is admissible (i.e., its value is always less than or equal to the exact lower bound), CLASP-NK does not miss a solution (n k-similar phylogenies) if one exists. This function is also monotonic in the number of leaves in partial phylogeny: if the partial phylogeny grows, then the distance increases also. If the lower bound $LB(T, T')$ is greater than k, then there is no need for CLASP-NK to search for a solution. In such a case, CLASP-NK marks the currently selected literals as a conflict, learns this conflict, and does the necessary backtracking. The rest of the algorithm is the same as that of CLASP except that CLASP-NK searches until it finds n solutions.

We can use other distance functions for CLASP-NK or we can compute similar/diverse solutions to other problems (e.g., planning, product configuration). For that, we need to modify CLASP-NK: we need to implement a suitable admissible distance measure, and change the DISTANCE-ANALYZE function of CLASP-NK.

8 Experimental Results

We applied the computational methods described above (i.e., the offline method, and the three online methods) to reconstruct similar/diverse phylogenies for Indo-European languages. We used the dataset assembled by Don Ringe and Ann Taylor [19]. As in [4], to compute such phylogenies, we considered the language groups Balto-Slavic (BS), Italo-Celtic (IC), Greco-Armenian (GA), Anatolian (AN), Tocharian (TO), Indo-Iranian (IIR), Germanic (GE), and the language Albanian (AL). While computing phylogenies, we also took into account some domain-specific information about these languages.

Let us first examine the results of experiments, considering the distance measures Δ_{D_n}, based on the nodal distance (Table 2). We present the results for the following computations: 2 most similar solutions, 2 most diverse solutions, 3 most similar solutions, 3 most diverse solutions, 6 most similar solutions. We solve these optimization problems by iteratively solving the corresponding decision problems (n k-SIMILAR/DIVERSE SOLUTION). For each method, we present the computation time,[1] the size of the memory used in computation, and the optimal value of k.

Let us first compare the online methods. In terms of both computation time and memory size, Online Method 3 performs the best, and Online Method 2 performs better than Online Method 1. These results conforms with our expectations: Online Method 1 requires an ASP representation of computing n k-similar/diverse phylogenies, and such a program may be too large for an answer set solver to compute an answer set for. Online Method 2 relaxes this requirement a little bit so that the answer set solver can compute the solutions more efficiently: it requires an ASP representation of phylogeny reconstruction, and an ASP representation of the distance measure, and then computes similar/diverse solutions one at a time. However, since the answer set solver needs to compute the distances between every two solutions, the computation time and the size of memory do not decrease much, compared to those for Online Method 1. Online Method 3 deals with the time consuming computation of distances between solutions, not at the representation level but at the search level; so it does not require an ASP representation of the distance function but requires a modification of the solver.

The offline method takes into account the previously computed 8 phylogenies for Indo-European languages (with at most 17 incompatible characters), and computes similar/diverse solutions using ASP as explained in Section 7. The offline method is more efficient, in terms of both computation time and memory, than Online Methods 1 and 2 since it does not compute phylogenies. On the other hand, the offline method is less efficient, in terms of both computation time and memory, than Online Method 3, since it requires both representation and computation of distances between solutions.

Here both the offline method and Online Method 1 guarantee to find an optimal solution, by iteratively solving the corresponding decision problems. On the other hand, Online Methods 2 and 3 compute similar/diverse solutions with respect to the first computed solution, and thus may not find the optimal value for k, as observed in the computation of 3 most similar phylogenies.

[1] All CPU times are in seconds, for a workstation with a 1.5GHz Xeon processor and 4x512MB RAM, running Red Hat Enterprise Linux (Version 4.3).

Table 2. Results of experiments, using the distance Δ_{D_n} based on the nodal distance

Problem	Offline Method	Online Method 1	Online Method 2	Online Method 3
2 most similar	12.39 sec.	26.23 sec.	19.00 sec.	1.46 sec.
	32MB	430MB	410MB	12MB
	$k = 12$	$k = 12$	$k = 12$	$k = 12$
2 most diverse	11.81 sec.	21.75 sec.	18.41 sec.	1.01 sec.
	32MB	430MB	410MB	15MB
	$k = 32$	$k = 32$	$k = 24$	$k = 32$
3 most similar	11.59 sec.	60.20 sec.	43.56 sec.	1.56 sec.
	32MB	730MB	626MB	15MB
	$k = 15$	$k = 15$	$k = 15$	$k = 16$
3 most diverse	11.91sec.	46.32 sec.	44.67 sec.	0.96 sec.
	32MB	730MB	626MB	15MB
	$k = 26$	$k = 26$	$k = 21$	$k = 26$
6 most similar	11.66sec.	327.28 sec.	178.96 sec.	1.96 sec.
	32MB	1.8GB	1.2GB	15MB
	$k = 25$	$k = 25$	$k = 29$	$k = 25$

Table 3. Results of experiments, using the distance Δ_{D_l} based on preferred diversifications

Problem	Offline Method	Online Method 1	Online Method 2
2 most similar	365.16 sec. (4.2GB)	16.11 sec. (236MB)	16.23 sec. (212MB)
3 most diverse	368.59 sec. (4.2GB)	46.11 sec. (659MB)	44.21 sec. (430MB)
6 most similar	368.45 sec. (4.2GB)	137.31 sec. (1.8GB)	212.59 sec. (1.1GB)

Now, let us consider the distance measures Δ_{D_l}, based on preference over diversifications (Table 3): two phylogenies are more similar if the diversifications closer to the root are more similar. Here we consider the similarities of diversifications until depth 3 (inclusive). We present the results for the following computations: 2 most similar solutions, 3 most diverse solutions, 6 most similar solutions. In Table 3, for each method, we present the computation time, the size of the memory used in computation, and the optimal value of k. Unlike what we have observed in Table 2, the offline method takes more time/space to compute similar/diverse solutions; this is due to the computation of distances with respect to Δ_{D_l} which requires summations, and representing summations in the language of LPARSE is not trivial. Other results are similar to the ones presented in Table 2.

In [4], after computing all 34 plausible phylogenies, the authors examine them manually and come up with three forms of tree structures, and then "filter" the phylogenies with respect to these tree structures. For instance, in Group 1, the trees are of the form (AN, (TO, (AL, (IC, (a tree formed for GE, GA, BS, IIR))))); in Group 2, the trees are of the form (AN, (TO, (IC, (a tree formed for GE, GA, BS, IIR, AL)))); in Group 3, the trees are of the form (AN, (TO, ((AL, IC), (a tree formed for GE, GA, BS, IIR)))). The results of our experiments with the distance measure Δ_{D_l} comply with these groupings. For instance, the 2 most similar phylogenies computed by Online Method 1 are in Group 1; the 3 most diverse phylogenies computed by Online Method 2 are in different groups. Likewise, the 6 similar phylogenies computed by our methods fall into Group 2.

9 Related Work

Finding similar or diverse solutions has been studied in propositional logic [2], and in constraint programming [13,12].

In [2], the authors propose two algorithms, $DP_{distance}$ and $DP_{distance+lasso}$, to solve DISTANCE-SAT—determining that a propositional CNF formula has a model that disagrees with a given partial interpretation on at most d variables. Our modification of CLASP's algorithm is similar to the first algorithm in that both algorithms check whether a partial interpretation computed in the DPLL-like search obeys the given distance constraints. On the other hand, unlike $DP_{distance}$, CLASP also uses conflict-driven learning: when it learns a conflicting set of literals, it will never try to select them in the later stages of the search. $DP_{distance+lasso}$ offers manipulations while selecting a new variable: it creates a set of candidate variables with respect to the distance function, computes weights of these variables relative to the distance function, and selects one with the maximum weight. On the other hand, in SELECT, CLASP creates a set of candidate variables, and selects one of the candidates to continue the search. Using the idea of $DP_{distance+lasso}$, we can modify CLASP further to manipulate the selection of variables with respect to the distance function. However, in the phylogeny reconstruction problem, since the domain of the distance function consists of the edge atoms which are far outnumbered by many auxiliary atoms, in SELECT the set of candidate variables generally consists of only auxiliary variables; due to these cases, the manipulation of the selection of variables is not expected to improve the computational efficiency.

[13,12] study various computational problems related to finding similar/diverse solutions, considering Hamming distance as in [2]. They present an offline method (similar to our method) that applies clustering methods, and two online methods: one based on reformulation (similar to Online Method 1), the other based on a greedy algorithm (similar to Online Method 2) that iteratively computes a solution that maximizes similarity to previous solutions. The computation of a k-close solution is due to a Branch & Bound algorithm (similar to the idea behind Online Method 3) that propagates some similarity/diversity constraints specific to the given distance function. Our offline/online methods are inspired by these methods of [13,12], but are not confined to only polynomial-time distance functions with polynomial range.

10 Conclusion

We have studied two kinds of computational problems related to finding similar/diverse solutions of a given problem, in the context of ASP: one problem asks for a set of n solutions that are k-similar (resp. k-diverse); the other one asks for a solution that is k-close (k-distant) to a given set of solutions. We have analyzed the computational complexity of these problems, and introduced offline/online methods to solve them. We have applied these methods to the phylogeny reconstruction problem, and observed their effectiveness in comparing many phylogenies for Indo-European languages.

There are many appealing ASP applications (e.g., product configuration, planning) for which finding similar/diverse solutions could be useful; on the other hand, no existing phylogenetic system can analyze phylogenies by comparing them. In this sense, our methods are useful both for ASP and for phylogenetics.

Acknowledgments. Thanks to Martin Gebser and Benjamin Kaufmann for their help with CLASP.

References

1. Adams, E.N.: Consensus techniques and the comparison of taxonomic trees. Syst. Zool 21, 390–397 (1972)
2. Bailleux, O., Marquis, P.: DISTANCE-SAT: complexity and algorithms. In: Proc. of AAAI, pp. 642–647 (1999)
3. Bluis, J., Shin, D.-G.: Nodal distance algorithm: Calculating a phylogenetic tree comparison metric. In: Proc. of BIBE, p. 87 (2003)
4. Brooks, D.R., Erdem, E., Erdoğan, S.T., Minett, J.W., Ringe, D.: Inferring phylogenetic trees using answer set programming. JAR 39(4), 471–511 (2007)
5. Brooks, D.R., Erdem, E., Minett, J.W., Ringe, D.: Character-based cladistics and answer set programming. In: Proc. of PADL, pp. 37–51 (2005)
6. Brooks, D.R., McLennan, D.A.: Phylogeny, Ecology, and Behavior: A Research Program in Comparative Biology. University of Chicago Press, Chicago (1991)
7. Chen, Z.-Z., Toda, S.: The Complexity of Selecting Maximal Solutions. Information and Computation 119, 231–239 (1995)
8. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem-proving. Communications of the ACM 5, 394–397 (1962)
9. Eiter, T., Subrahmanian, V.S.: Heterogeneous active agents, ii: Algorithms and complexity. Artif. Intell. 108(1-2), 257–307 (1999)
10. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Conflict-driven answer set solving. In: Proc. of IJCAI, pp. 386–392 (2007)
11. Goldberg, E., Novikov, Y.: Berkmin: A fast and robust sat-solver. Discrete Appl. Math. 155(12), 1549–1561 (2007)
12. Hebrard, E., Hnich, B., O'Sullivan, B., Walsh, T.: Finding diverse and similar solutions in constraint programming. In: Proc. of AAAI, pp. 372–377 (2005)
13. Hebrard, E., O'Sullivan, B., Walsh, T.: Distance constraints in constraint satisfaction. In: Proc. of IJCAI, pp. 106–111 (2007)
14. Hon, W.-K., Kao, M.-Y., Lam, T.-W.: Improved Phylogeny Comparisons: Non-shared Edges, Nearest Neighbor Interchanges, and Subtree Transfers. In: Hon, W.-K., Kao, M.-Y., Lam, T.-W. (eds.) Algorithms and Computation, pp. 369–382. Springer, Heidelberg (2000)
15. Lifschitz, V.: Action languages, answer sets and planning. In: The Logic Programming Paradigm: a 25-Year Perspective, pp. 357–373. Springer, Heidelberg (1999)
16. Marques-Silva, J., Sakallah, K.: A search algorithm for propositional satisfiability. IEEE Trans. Computers 5, 506–521 (1999)
17. Nye, T.M., Lio, P., Gilks, W.R.: A novel algorithm and web-based tool for comparing two alternative phylogenetic trees. Bioinformatics 22(1), 117–119 (2006)
18. Papadimitriou, C.: Computational Complexity. Addison-Wesley, Reading (1994)
19. Ringe, D., Warnow, T., Taylor, A.: Indo-European and computational cladistics. Transactions of the Philological Society 100(1), 59–129 (2002)
20. Robinson, D.F., Foulds, L.R.: Comparison of phylogenetic trees. Mathematical Biosciences 53(1-2), 131–147 (1981)
21. Semple, C., Steel, M.: A supertree method for rooted trees. Discrete Applied Mathematics 105, 147–158 (2000)
22. Soininen, T., Niemelä, I.: Developing a declarative rule language for applications in product configuration. In: Proc. of PADL, pp. 305–319 (1998)
23. White, J.P., O'Connell, J.F.: A Prehistory of Australia, New Guinea, and Sahul. Academic, San Diego (1982)

Attributed Data for CHR Indexing

Beata Sarna-Starosta[1] and Tom Schrijvers[2,*]

[1] LogicBlox Inc., Atlanta, Georgia, USA
bss@logicblox.com
[2] Department of Computer Science, K.U.Leuven, Belgium
tom.schrijvers@cs.kuleuven.be

Abstract. The overhead of matching CHR rules is alleviated by constraint store indexing. Attributed variables provide an efficient means of indexing on logical variables. Existing indexing strategies for ground terms, based on hash tables, incur considerable performance overhead, especially when frequently computing hash values for large terms.

In this paper we (1) propose *attributed data*, a new data representation for ground terms inspired by attributed variables, that avoids the overhead of hash-table indexing, (2) describe program analysis and transformation techniques that make attributed data more effective, and (3) provide experimental results that establish the usefulness of our approach.

Keywords: Constraint Handling Rules, indexing, program transformation, term representation, attributed variables.

1 Introduction

Constraint Handling Rules (CHR) [3] is a high-level rule-based declarative programming language, usually embedded in a host language such as Prolog or Haskell. CHR features *multi-headed rules*, i.e., rules with multiple predicates on the left-hand side (the *head*), which sets it apart from conventional declarative languages, where a rule's head admits only one predicate or function.

Multi-headed rules afford much of CHR's expressive power by allowing to easily combine information from distinct constraints via matching. However, as the matching procedure significantly affects the complexity of rule evaluation [13], this source of expressiveness often leads to performance bottlenecks. Aware of this problem, CHR developers have built data structures to support efficient indexing on variables (attributed variables [6]) and ground data (search trees [7]). With [11] came the realization that $\mathcal{O}(1)$ indexing is essential to implement CHR algorithms with optimal complexity, leading to the use of hash tables for indexing ground data, and the general result that the complexity of CHR systems equals that of RAM machines [13].

In this paper we advance the research on CHR indexing with the following contributions. We present *attributed data*, an alternative to hash tables for indexing ground data that does not suffer from as much overhead (Section 3); we

[*] Post-Doctoral Researcher of the Fund for Scientific Research - Flanders (Belgium) (F.W.O. - Vlaanderen).

describe a sequence of program post-processing steps that reduce the overhead incurred by the indexing transformations (Section 4); we propose an analysis to decide when to use the attributed data (Section 5); and we provide the experimental measurements that demonstrate the performance gain and the practical usefulness of our approach in K.U.Leuven CHR (Section 6).

Parts of this work described in Sections 3 and 4 have previously appeared at the CICLOPS 2008 symposium [8]. The implementation of the presented transformation is available at `http://www.cs.kuleuven.be/~toms/CHR/Indexing/`.

2 Motivation

A CHR rule is applicable when there exists a constraint substitution that matches the rule's head. Our experience has shown that the efficiency of CHR evaluation is significantly affected by the procedure of selecting such matching head substitutions for multi-headed rules. Indeed, in Frühwirth's analysis [13] the number of heads appears in the exponent of the worst-case time complexity formula.

CHR's on-demand approach builds head substitutions incrementally, by first matching the active constraint, and then adding stored constraints one at a time. The purpose of *indexing* is to bring relief to the matching bottleneck. While the naive approach considers all stored constraints as candidates for the substitution, indexing aims to considerably narrow down the number of candidates to consider.

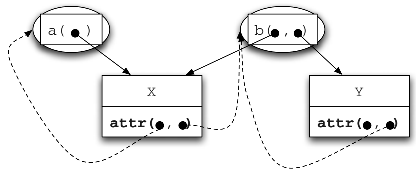
2.1 Attributed Variables

Efficient (constant-time) constraint store indexing has been traditionally implemented by means of *attributed variables* [5]. Attributed variables [4] provide a way to associate Prolog variables with mutable data represented as arbitrary terms. In the context of CHR, a variable's attribute corresponds to those stored constraints, in which the variable is involved. The attribute term has the form: `attr(`$Index_1$`,...,`$Index_n$`)`, where each $Index_i$ is a data structure, typically a list, that contains all constraints on the variable with a particular constraint symbol. The presence of all variable's constraints in its attribute expedites matching when the variable is shared among the constraints in the heads of the rules.

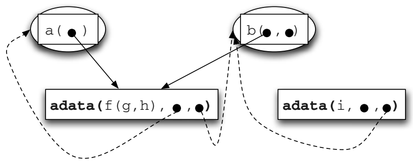
Example 1. Consider the rule:

```
a(X), b(X,Y) ==> write(Y).
```
(2.1)

Assuming that `a/1` and `b/2` are the only declared constraints, the attribute term of a constrained variable `X` has the form `attr(`$Index_a$`,`$Index_b$`)`, where $Index_a$ represents all stored constraints `a(X)` and $Index_b$ represents all stored constraints `b(X,Y)`. Figure 1(a) depicts a constraint store containing the constraints `a(X)` and `b(X,Y)`. The single-compartment boxes denote constraints, whereas the double-compartment boxes denote variables with attributes. The dashed arrows and ovals represent the index lists $Index_a$ and $Index_b$. Using such representation of the constraint store, given the constraint `a(X)` we can quickly find the matching constraint `b(X,Y)` by consulting the $Index_b$ list of variable `X`.



(a) Attributed Variables



(b) Attributed Data

Fig. 1. Constraints `a(X)` and `b(X,Y)` with two types of indexing

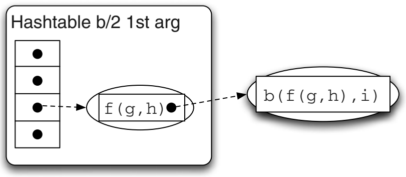


Fig. 2. Constraint `b(f(g,h),i)` with hash-table indexing

2.2 Ground Term Pattern Matching

Clearly, attributed variables are useful only when constraints involve Prolog variables. They cannot represent ground constraints, i.e. constraints in which all arguments are ground terms.

Example 2. The constraints `a(f(g,h))` and `b(f(g,h),i)` match the head of rule (2.1) under the substitution {`f(g,h)`/`X`, `i`/`Y`}. However, as these two constraints do not share any variables, attributed variable indexing cannot be exploited to extend the partial match `a(f(g,h))`. Note that even if the atom `i` was a variable, attributed variable indexing could not be used.

To account for cases such as that described in Example 2, early implementations of CHR accumulated constraints in global, unordered lists. This representation supported $\mathcal{O}(1)$-time insertion of the constraints, however, constraint lookup and deletion were—in the worst case—linear in the size of the store. The introduction of hash tables [11] facilitated indexing on ground data with amortized constant-time complexity for all operations.

Example 3. Figure 2 depicts the hash-table index on the first argument of the constraint `b/2`. When an active constraint `a(f(g,h))` is looking for a partner constraint to apply rule (2.1), it consults this hash table; A hash value computed for the term `f(g,h)` yields (modulo the array size) a position in the array; The bucket list at this position is traversed until detection of the bucket for `f(g,h)`, which contains a linked list of all `b/2` constraints with the first argument having the form `f(g,h)` (i.e., `b(f(g,h),i)` in our example).

The hash table is initialized to a small size, and dynamically expanded whenever the number of constraints exceeds a given threshold. The expansion involves replacing the current array with an array of doubled size, and re-evaluating the hash function for all elements. Frequent evaluation of the hash function, the traversal of the bucket lists, and the resizing operation incur constant, but potentially large, overhead on processing the hash tables, which makes them, as the means for constraint indexing, considerably slower than attributed variables: for a benchmark with tight loops involving no more than two constraints, we have measured a relative slow-down of about 50%.

Solution: Attributed Data In order to facilitate ground-term indexing with performance characteristics of attributed variables, we propose a representation which associates ground terms directly with their constraint store indexes. We call this term representation *attributed data*. Our approach considers only variable patterns; we have addressed indexing structure patterns in prior work [9]. Also, in the rest of the presentation we assume that our approach applies to only one constraint argument at a time.

Example 4. Figure 1(b) shows attributed data indexing applied to Example 2. Note how little it differs from the attributed variable indexing of Figure 1(a).

Even though based on the same idea as attributed variables, attributed data cannot be implemented by a simple adaptation of the attributed variable infrastructure to the domain of ground terms because of the different ways ground terms and variables are represented by Prolog systems. Every logic variable is created exactly once, and systems, such as WAM [1], maintain its single physical representation and update it in place. As a consequence, all occurrences of the same variable observe the effects of any updates—in particular, changing it into an attributed variable—through the shared representation. On the other hand, a ground term may have multiple physical representations, created at different times, and hence changing such a term into its attributed data representation in place has no effect on its copies. This difference imposes the need to implement a new way of supporting attributed data updates. Our answer to this need—a conversion function turning any copy of a ground term into the canonical, shared attributed data representation—is described in Section 3.2.

3 Attributed Data

The key insight underlying our new approach to pattern matching ground terms is that the externally provided ground terms can be transformed into the internal, attributed-variable–like representation by the CHR run time.

3.1 Attributed Data Representation

The internal representation $\mathcal{I}$ of a ground term $\mathcal{E}$ resembles an attributed variable in that it contains the ground term itself and its associated data:

$$\mathcal{I} = \texttt{adata}(\mathcal{E}, \mathit{Index}_1, \ldots, \mathit{Index}_n)$$

where each $Index_i$ is a constraint store index on an argument position of the term $\mathcal{E}$ in a head constraint of some program rule.

The number and form of attributed data indexes is orthogonal to the use of attributed data, and is determined by the CHR compiler based on the form of the rule heads and the set of constraints available when looking for a matching partner. A detailed discussion of this issue can be found in Section 3.2 of [7].

In this paper, we assume that each $Index_i$ is a flat list of constraint suspensions, with predefined operations for constraint addition and removal. The list can be updated (e.g., to replace an old index with a new one) by the destructive argument update predicate `setarg/3` implemented in most Prolog systems.

3.2 Conversion Functions

As discussed in Section 2.2, in order to support transforming ground terms into their attributed-data representations, we require an operation that is more involved than the built-in predicate `put_attr/3` used to turn ordinary logic variables into attributed variables. Hence, we use a conversion function ϕ which turns any copy of a ground term into the canonical, shared attributed-data representation.

Definition 1 (Conversion Functions). *The injective* conversion function ϕ *maps a ground term $t_\mathcal{E}$ of t onto its attributed-data representation $t_\mathcal{I}$:*

$$\phi(t_\mathcal{E}) = \begin{cases} h[t_\mathcal{E}] & \textit{if } h[t_\mathcal{E}] \textit{ is defined} \\ t_\mathcal{I} & \textit{otherwise} \\ & \quad \textit{such that } t_\mathcal{I} = \texttt{adata}(t_\mathcal{E},\emptyset_1,\ldots,\emptyset_n) \\ & \qquad\quad \textit{and } h := h[t_\mathcal{E} \rightarrow t_\mathcal{I}] \end{cases}$$

where h is a global hash table relating the ground terms to their known attributed-data representations. Each $\emptyset_i$ is an empty set of constraints, one for each argument position j of each constraint symbol c that is represented by attributed data. The injective conversion function $\psi = \phi^{-1}$ *maps the attributed-data representations $t_\mathcal{I}$ back to a copy of the ground term $t_\mathcal{E}$:*

$$\psi(\texttt{adata}(t_\mathcal{E}, Index_1, \ldots, Index_n)) = t_\mathcal{E}$$

Note that ϕ has an impure implementation, but a pure interface.

3.3 Source-to-Source Transformation

Apart from the performance aspect, the use of attributed data should be fully transparent to the programmer. Hence, to relieve the programmers from the need to explicitly call the conversion functions of Section 3.2, we provide a fully-automatable program transformation that introduces the conversions at *well-chosen* points in the program. The transformation serves two purposes: it (1) makes the programmers oblivious of the attributed-data representation, and (2) makes the attributed-data representation available for indexing to the CHR compiler. The first purpose implies that a CHR constraint c/n should be callable with ground terms as arguments, e.g. from the interactive Prolog shell. However, this conflicts with the second purpose, which requires the arguments of c/n to have the attributed-data form for indexing.

Our solution is to split the constraint c/n into two forms. The first form, c/n, is used externally by the programmers, and its arguments are ground terms. The second form, c'/n, is used internally when applying CHR rules, and its arguments are attributed data. The external form is defined in terms of the internal form by means of the *conversion CHR rule*, that applies the conversion function ϕ:

Definition 2 (Conversion Rule). *The* conversion rule Φ *replaces ground term argument* t_i *in constraint term* c/n *with attributed-data representation* $t'_i = \phi(t_i)$*:*

$$\texttt{c(}t_1\texttt{,}\ldots\texttt{,}t_i\texttt{,}\ldots\texttt{,}t_n\texttt{)}\ \texttt{<=>}\ t'_i\ \texttt{=}\ \phi(t_i)\texttt{,}\ \texttt{c}'\texttt{(}t_1\texttt{,}\ldots\texttt{,}t'_i\texttt{,}\ldots\texttt{,}t_n\texttt{).}$$

Example 5. Consider the constraint `arrow/2`, which in Thom Frühwirth's merge-sort program represents the numbers subject to the sort. The second argument of `arrow/2` is always ground. Thus, the conversion rule for this constraint has the form:

```
arrow(X,Ne) <=> Ni = φ(Ne), arrow'(X,Ni).
```

The original CHR rules should operate on the internal constraint form c'/n rather than c/n. For this purpose, we transform each rule into a *converted* rule.

Example 6. Consider the following rule on the `arrow/2` constraint:

```
arrow(X,A) \ arrow(X,B) <=> A < B | arrow(A,B).        (3.2)
```

In order to benefit from attributed data indexing, the rule head should be expressed in terms of the internal constraint form `arrow'/2`:

```
arrow'(X,A) \ arrow'(X,B) <=> A < B | arrow(A,B).
```

However, the rule as formulated above does not work: both the guard `A < B` and the body `arrow(A,B)` expect `A` and `B` to be ground terms rather than attributed data. Hence, we need to introduce the conversion functions:

```
arrow'(XI,AI) \ arrow'(XI,BI) <=> A=ψ(AI), B=ψ(BI),
                                  A < B | arrow(A,B).
```

More systematically:

Definition 3 (Converted Rule). *The* converted CHR rule *is defined as:*

$$\phi(H\ \{\substack{\texttt{<=>}\\\texttt{==>}}\}\ G \mid B)\ =\ H'\ \{\substack{\texttt{<=>}\\\texttt{==>}}\}\ G', G \mid B$$

where

– H' *differs from* H *in that any constraint* $c(t_1, \ldots, t_i, \ldots, t_n)$ *is replaced by its converted form* $c'(t_1, \ldots, x_i, \ldots, t_n)$*, where* x_i *is a fresh variable.*

$$\begin{tikzcd}
\underline{\text{original}} & \underline{\text{converted}} \\
c(X) \arrow[dr, "\phi"] & \\
& c'(Y) \arrow[dl, "\psi"'] \\
c(Z) \arrow[dr, "\phi"] & \\
& c'(U) \arrow[dl, "\psi"'] \\
\dots &
\end{tikzcd}$$

(a) Actual Situation

$$\begin{tikzcd}
\underline{\text{original}} & \underline{\text{converted}} \\
c(X) \arrow[dr, "\phi"] & \\
& c'(Y) \arrow[d] \\
& c'(U) \arrow[d] \\
& \dots
\end{tikzcd}$$

(b) Ideal Situation

Fig. 3. Transitions between the original and converted constraints

– *the new guard G' relates the original arguments of each constraint to the new ones: G' contains one $t_i = \psi(x_i)$ for each converted argument.*

Putting everything together:

Definition 4 (Converted Program). *The* converted CHR program $\phi(P)$ *is defined as the set of converted rules $\overline{R}$ comprising the original program, the functions ϕ and ψ, and the encoding of Φ:*

$$\phi(P) = \phi(\overline{R}) \cup \phi \cup \psi \cup \Phi$$

4 Post-processing

A converted program involves repeated application of the conversion functions to alternate between the external and internal representations of the constraints, which may be a major source of overhead. We now describe a four-step rewriting procedure, which statically eliminates most of this overhead. The procedure is based on the approach taken in our prior work on partial structure indexing [9].

In a typical execution scenario (Figure 3(a)), an external value is converted into the internal representation and matched in a head of a rule, then converted back in the rule's body for calling a new constraint, converted again to match another rule, and so on. Ideally (Figure 3(b)), converted rules should operate solely on the internal representation of the arguments, whereas the external values should be used only by the queries from outside the programs. Our rewriting procedure aims to trigger this ideal scenario. Rewritten programs execute in two phases: (1) conversion of arguments' external value to the internal representations, and (2) processing of the internal representations. For all but the most trivial programs, we expect the run-time cost of (1) to be marginal with respect to the cost of (2). This conjecture is born out by the benchmarks in Section 6.

We outline the rewriting steps by applying them to an example rule.

Example 7. Consider normalized version of Rule (3.2), which after conversion has the form:

```
arrow'(I₁,AI) \ arrow'(I₂,BI) <=> X = ψ(I₁), X = ψ(I₂),
                                   A = ψ(AI), B = ψ(BI),
                                   A < B | arrow(A,B).
```

Step 1: Make conversion explicit unfolds constraint calls according to the conversion rules:

```
arrow'(I₁,AI) \ arrow'(I₂,BI) <=> X = ψ(I₁), X = ψ(I₂),
                                   A = ψ(AI), B = ψ(BI),
                                   A < B | arrow'(φ(A),φ(B)).
```

Step 2: Eliminate identity conversion applies, from left to right, the equivalence $\forall t : \phi \circ \psi(t) = t$:

```
arrow'(I₁,AI) \ arrow'(I₂,BI) <=> X = ψ(I₁), X = ψ(I₂),
                                   A = ψ(AI), B = ψ(BI),
                                   A < B | arrow'(AI,BI).
```

Step 3: Convert external matching values to the internal representations applies from left to right the equivalence $\forall t_1, t_2 : \psi(t_1) = \psi(t_2) \Leftrightarrow t_1 = t_2$:

```
arrow'(I,AI) \ arrow'(I,BI) <=> X = ψ(I), A = ψ(AI), B = ψ(BI),
                                 A < B | arrow'(AI,BI).
```

Step 4: Clean up drops unused conversion guards:

```
arrow'(I,AI) \ arrow'(I,BI) <=> A = ψ(AI), B = ψ(BI),
                                 A < B | arrow'(AI,BI).
```

The proposed rewriting steps are not sufficient to enforce the ideal scenario of Figure 3(b). However, as shown in Section 6, they have good practical effects.

5 Analysis

The attributed data framework offers an attractive alternative to hash tables. Should our approach replace hash-table indexing for ground programs? It turns out that the overhead of setting up attributed data—with the help of a hash table—may be larger than the resulting run-time gain. Our experimental evaluation[1] indicates that, for overall performance improvement, each attributed data index should be used more than once. In this section we consider two strategies for deciding when to represent constraint arguments as attributed data.

[1] See the `fib` and `fib2` benchmarks in Section 6.

5.1 Manual Attributed Data Declaration

For some programs, the best decision as to when to use attributed data can be made by the programmer. Thus, we introduce the `attr_data` modifier to annotate individual arguments of a constraint in the constraint's declaration. The modifier is used in combination with a ground mode declaration `+`, and, optionally, a type declaration such as `int`.

Example 8. We indicate that attributed data shall be used for both integer-typed arguments of the merge-sort constraint `arrow/2` by means of the declaration:
`:- chr_constraint arrow(+int attr_data,+int attr_data).`

5.2 Automatic Attributed Data Index Selection

Although easy to implement, selecting the attributed data indices by hand may be challenging, and hence is prone to performance errors. Preferably, this task should be delegated to the CHR system, which has the advantage over the programmer in that it determines on which constraint arguments to index during matching. We propose an analysis that facilitates automatic selection of indexing arguments. The analysis, based on the abstract interpretation framework for CHR [12], approximates the number of times an argument is used for indexing. Based on our experimentally determined heuristic, we then select the arguments found to be used for indexing more than once.

In order to capture argument lookup information, we need to extend CHR's operational semantics. We assume that the built-in store is of the form $\bigwedge X_i = t_i/l_i$ and the constraints in the queries are of the form $c(X_1, \ldots, X_n)$. Here X_i are possibly identical variables, t_i are ground terms, and l_i are *lookup counts.* Informally, a lookup count for a term t is a natural number denoting how often t has been used to look up partner constraints. We omit the formal definition due to lack of space. Additionally, we assume that all stored constraints are ground.

Our analysis framework comprises an abstract domain of execution states, a function defining the conversion between the concrete and abstract execution states, and the abstraction of CHR's operational semantics.

Abstract Domain $\mathbf{\Sigma_a}$*.* An abstract execution state has two components: (1) the program point information present in the goal in order to determine applicable abstract semantic step, and (2) the abstraction of the lookup counts l_i. A concrete state is reduced to the corresponding abstract state by the abstraction function α_{ad}:

Definition 5 (Abstraction Function)

$$\alpha_{ad}(\langle A, S, B, T\rangle_n) = \langle \alpha_{ad-c}(A), \alpha_{ad-b}(B)\rangle$$

where the auxiliary functions α_{ad-c} *and* α_{ad-b} *respectively determine the abstract goal and abstract indexing count components.*

The abstraction functions for the two components are defined as:

$$\begin{aligned}\alpha_{ad-c}(c(X_i,\ldots,X_n)) &= c(X_i,\ldots,X_n)\\ \alpha_{ad-c}(c\#i\!:\!o) &= \alpha_{ad-c}(c)\!:\!o\\ \alpha_{ad-c}(c) &= \texttt{builtin} && (c \textit{ built-in})\\ \alpha_{ad-c}([c_1,\ldots,c_n]) &= [\alpha_{ad-c}(c_1),\ldots,\alpha_{ad-c}(c_n)]\end{aligned}$$

and $\alpha_{ad-b}(\wedge_i X_i = t_i/l_i) = \{X_i : \alpha_l(l_i)\}$ *where the abstraction of lookups is defined as* $\alpha_l(n) = \begin{cases} n & \textit{if } n \leq 1 \\ * & \textit{if } n > 1 \end{cases}$

That is, we reduce the lookup counts to 0, 1 or *many* (denoted by $*$).

The partial order $\prec$ and least upper bound operator $\sqcup$ for abstract states both assume that the program point component is identical. They are defined in terms of the point-wise application of the natural abstractions of $<$ and max over the lookup counts.

Definition 6 (Partial Ordering)

$$\langle A_1, B_1\rangle \preceq \langle A_2, B_2\rangle = A_1 \equiv A_2 \wedge B_1 \preceq B_2$$

where $B_1 \preceq B_2 = \forall (X\!:\!l_1) \in B_1 \exists (X : l_2) \in B_2 : l_1 \preceq l_2$, *and* $0 \prec 1 \prec *$.

Definition 7 (Least Upper Bound)

$$\langle A, B_1\rangle \sqcup \langle A, B_2\rangle = \langle A, B_1 \sqcup B_2\rangle$$

where $B_1 \sqcup B_2 = \{(X : l_1 \sqcup l_2) \mid (X : l_1) \in B_1, (X : l_2) \in B_2\}$,
and $l_1 \sqcup l_2 = \begin{cases} l_2 & \textit{if } l_1 \prec l_2 \\ l_1 & \textit{otherwise} \end{cases}$

Abstract Semantic Function. The abstract semantic function $\mathcal{AS}[\![\mathcal{P}]\!]$ is the abstraction of the operational semantics of CHR. The abstract semantic function exploits two pieces of information derived during the program analysis phase by the CHR compiler: which arguments of a constraint are used for indexing, and which occurrences of a constraint are passive.

The latter datum—passiveness of an occurrence o of a constraint c/n—means that the occurrence o does not fire a rule. We denote the conservative approximation derived by the CHR compiler as *passive*$(c/n, o)$. Since the CHR compiler does not generate code for passive occurrences, it is not necessary to increase lookup counts in this case, as no lookups are performed.

The former datum—which constraint arguments are used for indexing—is determined based on the rule head patterns and active occurrences of the constraints. We capture this information as $B' = \textit{indexing}(c, B, j, r)$ meaning that, after the attempt to fire rule r with active abstract constraint c at occurrence j, the abstract lookup counts change from B to B'.

Definition 8 (Abstract Semantic Function)

1. AbstractSolve

$$\mathcal{AS}[\![\mathcal{P}]\!](\langle \texttt{builtin}, B\rangle) = \langle \Box, B\rangle$$

As we have assumed that all constraints are ground, this transition does not reactivate any CHR constraints, and does not affect lookup counts.

2/3. AbstractActivate *Let c be a CHR constraint.*

$$\mathcal{AS}[\![\mathcal{P}]\!](\langle c, B\rangle) = \mathcal{AS}[\![\mathcal{P}]\!](\langle c\!:\!1, B\rangle)$$

This transition stores the new constraint. It does not affect lookup counts.

4. AbstractDrop *Let c be a CHR constraint with no occurrence c:j*

$$\mathcal{AS}[\![\mathcal{P}]\!](\langle c\!:\!j, B\rangle) = \langle \Box, B\rangle$$

This transition deactivates the active constraint. It does not affect lookup counts.

5a. AbstractSimplify *Let d be the j^{th} occurrence of c in a (renamed apart) rule $r \in \mathcal{P}$ with $\neg passive(c, j)$:*

$$r \;@\; H_1' \setminus H_2', d_{[j]}, H_3' \iff g \mid C$$

then $$\mathcal{AS}[\![\mathcal{P}]\!](\langle c\!:\!j, B\rangle) = \mathcal{AS}[\![\mathcal{P}]\!](s_1) \sqcup \mathcal{AS}[\![\mathcal{P}]\!](s_2)$$

where $$\begin{cases} B' = indexing(c, B, j, r) \\ s_1 = \langle \alpha_{ad-c}(C), B'\rangle \\ s_2 = \langle c\!:\!j+1, B'\rangle \end{cases}$$

6a. AbstractPropagate *Let d be the j^{th} occurrence of c in a (renamed apart) rule $r \in \mathcal{P}$ with $\neg passive(c, j)$:*

$$r \;@\; H_1', d_{[j]}, H_2' \setminus H_3' \iff g \mid C$$

then $$\mathcal{AS}[\![\mathcal{P}]\!](\langle c\!:\!j, B\rangle) = \mathcal{AS}[\![\mathcal{P}]\!](\langle c\!:\!j+1, B'\rangle)$$

where $B' = lfp(f, \langle B\rangle)$.

The auxiliary function f is defined as:

$$f(B_0) = B_2$$

where $$\begin{cases} B_1 = indexing(c, B_0, j, r) \\ \langle \Box, B_2\rangle = \mathcal{AS}[\![\mathcal{P}]\!](\langle \alpha_{ad-c}(C), B_1\rangle) \end{cases}$$

5b/6b. AbstractPassive *Let d be the j^{th} occurrence of c in a (renamed apart) rule $r \in \mathcal{P}$ with $passive(c, j)$, then*

$$\mathcal{AS}[\![\mathcal{P}]\!](\langle c\!:\!j, B\rangle) = \mathcal{AS}[\![\mathcal{P}]\!](\langle c\!:\!j+1, B\rangle)$$

```
f(N) \ f(N) <=> true.              f(N), f(N) <=> f(N).
f(0) <=> true.                     f(0) <=> true.
f(N) ==> M is N - 1, f(M).         f(N) ==> M is N - 1, f(M).

            (a)                                (b)
```

Fig. 4. Programs showing worse (a) and better (b) performance with attributed data

7. AbstractGoal

$$\mathcal{AS}[\![\mathcal{P}]\!](\langle [c_1,\ldots,c_n], B_0\rangle) = \langle \Box, B_n\rangle$$

where for $i \in 1..n$

$$\mathcal{AS}[\![\mathcal{P}]\!](\langle c_i, \mathit{proj}(B_{i-1}, c_i)\rangle) = \langle \Box, B'_i\rangle$$

and

$$\begin{aligned} \mathit{proj}(B, c(X_1,\ldots,X_n)) &= \{(X_i : l_i) \in B \mid i \in 1..n\} \\ \mathit{ext}(B_x, B_y) &= B_x \cup \{(X : l) \in B_y \mid \neg\exists l' : (X \rightarrowtail l') \in B_x\} \\ B_i &= \mathit{ext}(B'_i, B_{i-1}) \quad (i \in 1..n) \end{aligned}$$

This transition sequences a list (conjunction) of goals.

Example Consider the two programs in Fig. 4. The use of attributed data slows down the program in Fig. 4(a), but improves the performance of the program in Fig 4(b). The reason for this difference is that in the program in Fig. 4(a) each argument is used for indexed lookup only once, whereas in the program in Fig. 4(b) some arguments are used multiple times.

Consider now the analysis for the program in Fig. 4(a):

$$\begin{aligned} &\mathcal{AS}[\![\mathcal{P}]\!](\langle f(N), \{N:0\}\rangle) \\ &\quad= \mathcal{AS}[\![\mathcal{P}]\!](\langle f(N):1, \{N:0\}\rangle) \\ &\quad= \mathcal{AS}[\![\mathcal{P}]\!](\langle f(N):2, \{N:1\}\rangle) \sqcup \mathcal{AS}[\![\mathcal{P}]\!](\langle \texttt{builtin}, \{N:1\}\rangle) \\ &\quad= \langle \Box, \{N:1\}\rangle \sqcup \langle \Box, \{N:1\}\rangle \\ &\quad= \langle \Box, \{N:1\} \end{aligned}$$

where (since the second occurrence is passive)

$$\begin{aligned} &\mathcal{AS}[\![\mathcal{P}]\!](\langle f(N):2, \{N:1\}\rangle) \\ &\quad= \mathcal{AS}[\![\mathcal{P}]\!](\langle f(N):3, \{N:1\}\rangle) \\ &\quad= \mathcal{AS}[\![\mathcal{P}]\!](\langle f(N):4, \{N:1\}\rangle) \sqcup \mathcal{AS}[\![\mathcal{P}]\!](\langle \texttt{builtin}, \{N:1\}\rangle) \\ &\quad= \langle \Box, \{N:1\}\rangle \sqcup \langle \Box, \{N:1\}\rangle \\ &\quad= \langle \Box, \{N:1\} \end{aligned}$$

and

$$\begin{aligned} &\mathcal{AS}[\![\mathcal{P}]\!](\langle f(N):4, \{N:1\}\rangle) \\ &\quad= \mathcal{AS}[\![\mathcal{P}]\!](\langle f(N):5, \{N:1\}\rangle) \sqcup \mathcal{AS}[\![\mathcal{P}]\!](\langle [\texttt{builtin}, f(M)], \{N:1\}\rangle) \\ &\quad= \langle \Box, \{N:1\}\rangle \sqcup \langle \Box, \{N:1\}\rangle \\ &\quad= \langle \Box, \{N:1\} \end{aligned}$$

Hence, the lookup count never exceeds 1, and thus our heuristic indicates that attributed data should not be used. If, instead, we consider the program in Fig. 4(b), the main derivation becomes:

$$\begin{aligned}
&\mathcal{AS}[\![\mathcal{P}]\!](\langle f(N), \{N:0\}\rangle)\\
&\quad= \mathcal{AS}[\![\mathcal{P}]\!](\langle f(N):1, \{N:0\}\rangle)\\
&\quad= \mathcal{AS}[\![\mathcal{P}]\!](\langle f(N):2, \{N:1\}\rangle) \sqcup \mathcal{AS}[\![\mathcal{P}]\!](\langle \mathbf{f}(\mathbf{N}), \{N:1\}\rangle)\\
&\quad= \langle \square, \{N:1\}\rangle \sqcup \langle \square, \{N:*\}\rangle\\
&\quad= \langle \square, \{N:*\}
\end{aligned}$$

because the first rule now involves a recursive call. Now the lookup count is $*$ and using attributed data is recommended.

6 Evaluation

We implemented our approach in K.U.Leuven CHR [10] on SWI-Prolog [14], and tested it on several benchmarks[2]. All run times, given in seconds, as well as relative to the original for the transformed versions, were measured on an Intel Pentium 4, 2.00 GHz, with 512 MB RAM.

Our implementation of attributed data consists of two components: (1) a pre-processor, which transforms a CHR program with key annotations into its converted form, and (2) the actual code generator of the CHR compiler, which generates attributed data indexing instructions and emits definitions for the conversion functions. The function ϕ is implemented in terms of the hash tables used for hash-table indexing. The function ψ is always called as $B = \psi(A)$; we inline it as A `= adata(`B`,_,...,_)` at each call site.

Table 1 lists the run-time results of exploiting attributed data in K.U.Leuven CHR, measured for plain hash tables, plain attributed data instead of the hash table indexes, and attributed data with post-processed rule bodies.

Table 1. K.U.Leuven CHR run times (in sec.) for attributed data benchmarks

	index representation				
benchmark	hash table	attr. data	relative	post-processed	relative
`chrg`	2.17	2.10	96.8%	1.58	72.8 %
`flat_ram`	4.69	4.31	91.9%	2.50	53.3%
`mergesort`	3.33	4.89	146.8%	1.85	55.6 %
`reverse`	2.55	3.25	127.4%	1.92	75.3%
`uf_opt`	0.34	0.38	111.8%	0.25	73.5%
`turing`	1.50	1.31	87.3%	1.19	79.3%
`wfs`	1.32	0.88	66.7%	0.85	64.4%
`fib`	1.24	1.53	123.4%	1.52	122.6%
`fib2`	1.61	1.30	80.7%	1.05	65.2%
`dijkstra`	2.26	4.52	200.0%	3.53	156.2%
`dijkstra2`	1.54	2.20	142.9%	1.25	81.2 %

[2] Available at `http://www.cs.kuleuven.ac.be/~toms/CHR/Indexing/`

The block of the first seven benchmarks clearly demonstrates the positive effects of our approach. Although, the attributed data used on its own causes a slow-down (up to almost 50% for `mergesort`), when augmented with post-processing, it improves the run time by 20% to 50%.

The block of the last four benchmarks shows two programs for which the use of attributed data impairs the performance. The first program, `fib`, involves one hash-table lookup per new constraint. Hence, the attributed data manipulation is pure overhead (25%). For this reason, our analysis from Section 5 flags the program as unsuitable for attributed data; it does not flag any of the other benchmarks.

The second benchmark, `fib2`, replaces the simpagation rule of `fib`:

```
        fib(N,F1) \ fib(N,F2) <=> F1 = F2.
```

with the simplification rule:

```
        fib(N,F1), fib(N,F2) <=> F1 = F2, fib(N,F1).
```

This modification causes the parameter `N` to be reused in the new call in the rule's body[3]. As a consequence, attributed data requires only one hash-table lookup for every two new constraints, which results in a noticable speed-up.

The second slow-down, in `dijkstra`, results form a limitation of our current implementation, which does not allow multi-argument indices involving attributed data arguments. Thus, we were required to replace a two-argument hash table index by a single-argument attributed data index. For this benchmark, the two-argument index turns out to be more selective and more efficient. However, the use of *symbol specialization* [9] allowed to entirely eliminate the second matching argument from this benchmark, and thus obtain a speed-up in the resulting program `dijkstra2`.

7 Discussion and Related Work

We have presented attributed data—a new term representation that improves the efficiency of CHR indexing at a high level—and a complementary post-processing procedure that reduces the overhead of conversions between the new representation and the standard representation of Prolog terms. We have implemented our approach for the K.U.Leuven CHR system on SWI-Prolog. The evaluation on a set of benchmarks shows that attributed data enables performance improvement, and that post-processing is critical to fully realize this potential.

Several programming languages define features that resemble the concept of attributed data. The conversion function ϕ relates to *hash consing*—a technique, originated in Lisp, for mapping terms to, and representing them by, unique (hash) values. Although the main aim of hash consing is to reduce memory consumption by increased sharing, it is also used to speed up equality tests.

[3] Note that fib and fib2 implement different algorithms for computing fibonacci numbers, and should be only compared w.r.t the relative impact of our transformation.

Like attributed data, Mercury's *solver types* [2] impose a dual view of constraint arguments. The internal representation type is defined by the library programmer, rather than generated automatically. Externally, the solver type is abstract, but coercion functions should be provided for external representations. Finally, a folklore[4] optimization technique in C/C++ adds (pointer) fields to structures to concisely represent lists (and other data types) that contain them.

Acknowledgements. The authors thank the ICLP reviewers, Leslie De Koninck, Bart Demoen and Jon Sneyers for their helpful comments on the paper.

References

1. Aït-Kaci, H.: Warren's Abstract Machine: A Tutorial Reconstruction. MIT Press, Cambridge (1991)
2. Becket, R., et al.: Adding constraint solving to Mercury. In: Van Hentenryck, P. (ed.) PADL 2006. LNCS, vol. 3819, pp. 118–133. Springer, Heidelberg (2005)
3. Frühwirth, T.: Theory and practice of Constraint Handling Rules. Journal of Logic Programming 37(1–3), 95–138 (1998)
4. Holzbaur, C.: Metastructures vs. Attributed Variables in the Context of Extensible Unification. Technical Report TR-92-23, Austrian Research Institute for Artificial Intelligence, Vienna, Austria (1992)
5. Holzbaur, C., Frühwirth, T.: Compiling Constraint Handling Rules into Prolog with attributed variables. In: Nadathur, G. (ed.) PPDP 1999. LNCS, vol. 1702, pp. 117–133. Springer, Heidelberg (1999)
6. Holzbaur, C., Frühwirth, T.: A Prolog Constraint Handling Rules Compiler and Runtime System. Special Issue Journal of Applied Artificial Intelligence on Constraint Handling Rules 14(4) (April 2000)
7. Holzbaur, C., de la Banda, M.G., Stuckey, P.J., Duck, G.J.: Optimizing Compilation of Constraint Handling Rules in HAL. Theory and Practice of Logic Programming 5(4&5), 503–531 (2005)
8. Sarna-Starosta, B., Schrijvers, T.: An efficient term representation for CHR indexing. In: Carro, M., Demoen, B. (eds.) Proceedings of CICLOPS 2008, pp. 172–186 (2008)
9. Sarna-Starosta, B., Schrijvers, T.: Transformation-based indexing techniques for constraint handling rules. In: Schrijvers, T., Raiser, F., Frühwirth, T. (eds.) CHR 2008, RISC Report Series 08-10, University of Linz, Austria, Hagenberg, Austria, July 2008, pp. 3–18 (2008)
10. Schrijvers, T., Demoen, B.: The K.U.Leuven CHR system: Implementation and application. In: 1st Workshop on Constraint Handling Rules: Selected Contributions, pp. 1–5 (2004)
11. Schrijvers, T., Frühwirth, T.: Optimal Union-Find in Constraint Handling Rules. Theory and Practice of Logic Programming 6(1&2) (2006)
12. Schrijvers, T., Stuckey, P.J., Duck, G.J.: Abstract interpretation for Constraint Handling Rules. In: Barahona, P., Felty, A.P. (eds.) PPDP 2005, Lisbon, Portugal, July 2005, pp. 218–229. ACM Press, New York (2005)
13. Sneyers, J., Schrijvers, T., Demoen, B.: The computational power and complexity of Constraint Handling Rules. ACM Transactions on Programming Languages and Systems (TOPLAS) 31(12) (2009)
14. Wielemaker, J.: SWI-Prolog release 5.6.0 (2006), `http://www.swi-prolog.org/`

[4] e.g. the Linux kernel linked list: `http://isis.poly.edu/kulesh/stuff/src/klist/`

User Defined Indexing

David Vaz[1], Vítor Santos Costa[2], and Michel Ferreira[3]

[1] LIACC - DCC/FCUP, University of Porto, Portugal
[2] CRACS - DCC/FCUP, University of Porto, Portugal
[3] Instituto de Telecomunicações - DCC/FCUP, University of Porto, Portugal

Abstract. Logic programming provides an ideal framework for tackling complex data, such as the multi-dimensional vector-based data used to represent spatial databases. Unfortunately, the usefulness of logic programming systems if often hampered by the fact that most of these systems have to rely on a single unification-based mechanism as the only way to search in the database. While unification can usually take effective advantage of hash-based indexing, it is often the case that queries over more complex and structured data, such as the vectorial terms stored in spatial databases, cannot.

We propose a new extension to Prolog indexing: User Defined Indexing (UDI). In this mechanism, the programmer may add extra information to Prolog indices so that only interesting fragments of the database will be selected. UDI provides a general extension of indexing, and can be used for both instantiated and constrained variables. As a test case, we demonstrate how UDI can be combined with a constraint system to provide an elegant and efficient mechanism to generate and execute range queries and spatial queries. Experimental evaluation shows that this mechanism can achieve orders of magnitude speedups on non-trivial datasets.

1 Introduction

Logic programming provides an ideal framework for tackling complex data, using a single and universal representation of pieces of such data as *logic terms*. A logic term can represent things such as an unbound variable, a constant or an integer, or more complex and structured entities such an interval over reals or a vectorial polygon. The universality of this representation is a key feature for the declarative flavor of logic programs. In particular, it is the basis for the generic handling of data through the single mechanism of *unification*. While this unification and the term-based representation of the world are fundamental flagships of the logic programming paradigm, they can also entangle the usefulness and effectiveness of logic programming for data-intensive applications. Performing search through unification can be terribly ineffective because of the match-based process associated with it. Indexing tries to overcome this inefficiency, and has been coupled to the earliest Prolog implementations [1], in order to narrow the number of clauses to try. Indexing is however based on the representation of data, which is universally term-based, and is very much designed in Prolog systems around the lexical or syntactic form of such terms, rather than its *semantic*. A conspicuous example of that is the position-based indexing proposed in the WAM [2]. The divergence from semantic is contrary to the Prolog's focus on the *what*, instead of the *how*, but is rooted on the fact that indexing is an implementation issue, rather than a programmer's concern.

Lexical indexing is usually well performed through hashing techniques, which cope naturally with the match-based mechanism of unification. Efficient execution of Prolog queries requires the programmer to be aware of this close relationship between hash-based indexing and unification. The following two queries:

```
?- p(A), q(B), A=B.
?- p(A), q(A).
```

are semantically equivalent, but have very different performances, as the indexing over goal `q/1` is only effective when unification is pulled to the argument of the call. This is an important difference between Prolog and relational databases querying, where the execution of the later is preceded by an optimizer that is able to look globally to the query and define an execution plan that maximizes efficiency. Systems incorporating global analysis in the compilation of Prolog, such as Ciao [3], are also able to perform some goal reordering that would use the constraint over variable `B` on `q/1` goal [4]. However, hash-based indexing of Prolog predicates is not able to take advantage of constraints over arguments that are not based on unification. The following query:

```
?- p(A), q(B), A>B.
```

has no possible rewriting in Prolog that would make it efficient, even if we could pull the constraint to the call of goal `q/1`. This is due to the fact that the lexical, hash-based, indexing of predicate `q/1` is not able to take advantage of the *semantic* constraint of order, either defined in a term-based representation domain of numbers or strings, between variables `A` and `B`. In the same way, if the argument of predicates `p/1` and `q/1` are terms representing a vectorial polygon, then a query such as:

```
?- p(A), q(B), overlaps(A,B).
```

is also unable to improve efficiency based on a hash-based index over the argument of predicate `q/1`. Efficient indexing over spatial terms is particularly important, not only because of the usual mammoth size of such predicates, but also because of the computationally expensive execution of spatial operators.

In this paper we propose and implement a *semantic-oriented* indexing of Prolog predicates, where the programmer is able to define the indexing mechanism based on *what* the terms in the arguments of a predicate are meant to represent. This User Defined Indexing (UDI) allows users to provide an indexing function that selects a subset of the clauses of a predicate, given a set of constrained variables or bound Prolog terms. This function implements the type of indexing the user deems appropriate for the predicate, from specialized hash-based functions to multi-dimensional indexing suited for spatial terms. We propose a constraint based syntax over the logic variables that can affect the efficiency of indexing, retaining the declarative style of Prolog querying.

The remainder of this paper is organized as follows: Section 2 presents the current state-of-the-art of Prolog indexing; Section 3 explains the indexing mechanisms used to efficiently access data structured in ranges and multi-dimensional objects; Section 4 presents our proposal of User Defined Indexing and addresses the engine modifications to implement it in Yap; Section 5 gives some examples of user defined indexers and Section 6 performs a complete evaluation over very large datasets; Section 7 concludes the paper.

2 Indexing Prolog Programs

Indexing is a key feature in Prolog implementations and has been supported since Warren's DEC-10 Prolog system [1]. Both DEC-10 Prolog and Warren's WAM [2] implement indexing on the first argument, and this has become standard in Prolog systems. Figure 1(a) shows the WAM code for a small database shown next:

```
has_property(d1, salmonella, p).
has_property(d1, salmonella_n, p).
has_property(d2, salmonella, p).
has_property(d2, cytogen_ca, n).
has_property(d3, cytogen_ca, p).
```

Figure 1(a) shows WAM indexing code as a tree, with *switch* nodes, that implement clause selection, and *clause chain* nodes, that either jump to clauses or support backtracking through a set of clauses. There are two different switch nodes in the WAM. The first, *switch_on_type*, selects according whether the first argument, A_1, is unbound, constant, pair or structure. The second type, say *switch_on_constant* selects clauses that match a *value*. Given a large enough number of different values, the WAM will implement this operation as a lookup in the hash table. Looking up an hash table takes constant time in average, hence indexing can in the best case improve query execution from linear in the number of clauses to constant-time, with only a small overhead.

A natural step from the WAM is to index on multiple arguments. Systems such as Prolog by BIM [5] and SWI-Prolog [6] do so in an user-specified fashion.

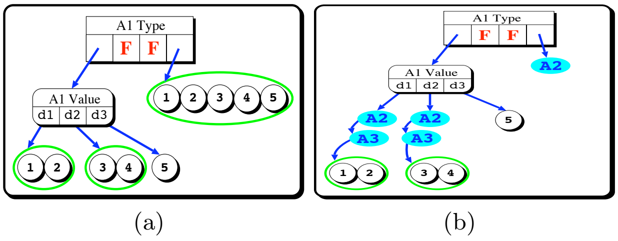


Fig. 1. WAM and JITI Code for `has_property/3`

This approach requires prior knowledge on which modes of usage are going to be used in the program, though. Just In Time Indexing (JITI) [7] addresses this problem by generating indexing code only when needed. For example:

```
?- has_property(d1, _, _).
```

would result on the JITI code shown in Figure 1(b): notice that the code is very similar to the original WAM code, but it includes "wait nodes", shown as filled ovals. Imagine next the queries:

```
?- has_property(d1, salmonella, _).
?- has_property(_, salmonella_, _).
```

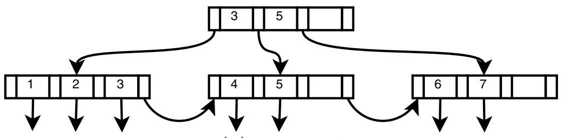
The JITI has the ability to expand the tree in Figure 1(b) with hashes on the first *two* arguments first, and then later building an alternative index that hashes on the *second* argument. This is implemented by generating new trees rooted at the *wait* nodes. The JITI seems to address well the cases where one wants to lookup a value in a database. This is an important application of Prolog, but not the only one.

3 Indexing Ranges and Spatial Data in RDMs

Indices based on exact matching of values are useful when searching for the value that matches some constraints, and they are usually implemented with hash tables. On the other hand, quite often users are interested in different styles of queries. One typical example is finding all values that are larger than some X; another are "ranges queries", that is, finding all values that are between two predefined boundaries. Such queries can be naturally written as logic programs, but are difficult to implement effectively with hash tables. More recently, there has been wide interest in storing and manipulating geographical data. These problems have motivated a large body of research in the Relational Database Management Systems (RDMs) community, which has proposed a number of indexing structures, such as B+-Trees [8] and R-Trees [9]. We briefly review these data structures next.

A B+-Tree is a self-balanced tree based on a B-Tree: it is often used in RDMs because it allows for logarithmic time selections, insertions and deletions. In B+-Trees data is stored in leaf nodes and only keys are stored in inner nodes (index nodes). Leaves in B+-Trees are linked to one another in a linked list. The main advantage of B+-Trees versus Hash Tables is that data is kept in order, making range queries (inequalities) possible and efficient. Figure 2(a) shows an example of a B+-Tree. The inner node separates the tree into three ranges $X < 3$, $3 < X \leq 5$ and $X > 5$. Given a key, search executes by going down from root of the tree and taking the branch covering the key, as shown in Figure 2(b).

B+-Trees are useful when addressing ordered values, but are not sufficient to index complex multi-dimensional data, such as spatial data. In this case an important operation is to compute whether two geographical objects *intersect*. Quite often, an object's Minimum Bounding Rectangle (MBR) or Bounding Box,

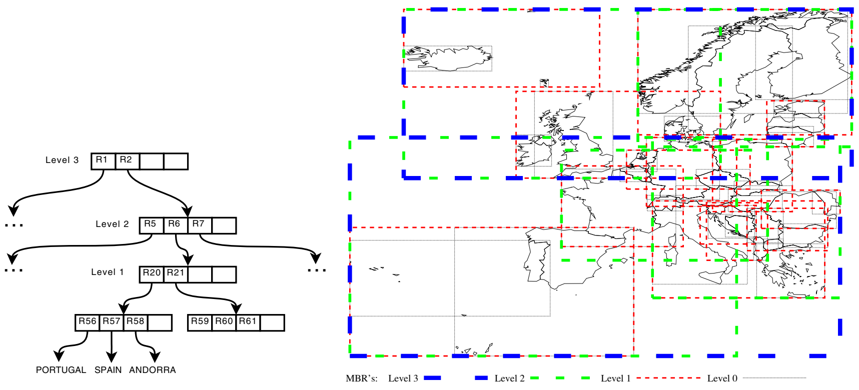


(a) Example

```
search (node, key)
  if (node is leaf)
    find key in node
  else
    find branch in node
    search (branch, key)
```

(b) Search

Fig. 2. B+Tree



(a) R-Tree Structure (b) MBR Containment

Fig. 3. R-Tree of European Countries

is used towards this goal. Namely, MBRs are used to implement the *R-Tree*, a major datastructure used in databases such as PostGIS [10] to quickly find all objects in a given area, e.g., "find all lakes in Switzerland".

R-Trees are inspired on B+-Trees. The key idea is that R-Trees use MBRs to index data. Each leaf nodes stores an object, and is keyed by the object's MBR. Inner nodes are keyed by an MBR that is the union of all MBRs below. Notice, that in contrast to B+-Trees, keys cannot be sorted as there is no order. On the other hand, searches in R-Tree are similar to searches in B+-Tree, except that *several MBRs in the same node may overlap with the searched MBR.* As a result we can have several valid branches at each node, and it is not possible to guarantee good worst-case performance. Indeed, in the worst case scenario, a query MBR can contain the whole dataset; in this case the complete indexing structure will need to be searched. Nevertheless, on most datasets the tree will maintain a shape that allows the search algorithm to quickly discard irrelevant regions.

Figure 3 shows an example R-Tree designed to store the boundaries of European countries. Figure 3(a) details part of the index structure, and Figure 3(b) graphically depicts the actual boundaries and MBRs that define the R-Tree. Notice that although European countries do not overlap, their MBRs do. The tree has three levels. The root node (Level 3) contains two MBRs, R_1 and R_2, shown as the wider lines. Notice that there is some overlap, as we cannot find a disjoint balanced union of MBRs that covers the whole of Europe. The overlap is even more evident on Level 2, Also observe that whereas Iceland, Greece and Portugal belong to a single box *for each level*, the central Alps region in Europe is covered by a large number of overlapping MBRs *at all levels*.

4 User Defined Indexing

In this work we are motivated by our desire to query complex databases declaratively and *efficiently*. Say, in Prolog in order to find all cities with more than 5 million people, we would state the query:

```
?- city(X,Pop), Pop > 5000000.
```

Execution of this query visits every city, returning only the ones with population over 5 million. This is arguably the most inefficient execution one can follow, especially if only a few cities have population above 5 million. In order to constrain the search we need to know that `Pop` must be over 5 million people first. This is not possible in Prolog, but it is possible in the framework of constraints:

```
?- Pop #> 5000000, city(X,Pop).
```

Notice that stating this constraint is not sufficient to improve performance: we must *use it to narrow search* over `city/2`. In this work, we propose to do so through indexing. This requires addressing two challenges:

1. We must be able to index on this constraint.
2. Because constraints provide a powerful and flexible language, it is not possible beforehand to implement an abstract machine that will address all possible constraints: we need a generic framework for indexing on unknown terms.

We address the latter problem first through our UDI mechanism, that allows programmers to define a function that selects a subset of clauses, given a class of attributed variables or Prolog terms. Next, we discuss the UDI in more detail.

4.1 Principles

Given a program P and a procedure Q defined as a set of clauses $\{c_1, \ldots, c_n\}$, Q's indexing code I^P is a function defined as follows: given a goal $G\pi$ and a matching procedure Q, where π is a set of constraints, then $I^P : (Q, G\pi) \to Q'$ selects a set of clauses $Q' \subset Q$ such that if reducing $G\pi$ against $c \in Q$ succeeds then $c \in Q'$.

Clearly, the most trivial indexing function is the identity function: $Q' = Q$. In general, as I^P incurs an overhead, one has to make sure that the benefits of computing I^P outweigh this overhead. One way to do so is to restrict how indexers are constructed. Typically, Prolog systems restrict I^p as follows:

1. I^P is known at compile-time; that is, the function I^P must be explicit before querying the program.

2. $I^P(Q, G\pi)$ is local, that is it depends on Q only and not on $P - Q$.
3. Indexing uses Herbrand constraints, that is, $I^P(Q, G\pi) = I^P(Q, G\sigma)$, where $\sigma \subset \pi$ are the Herbrand constraints in π.

Work on indexing has proceed by relaxing these constraints. YAP's JITI, for example, relaxes the first constraint: essentially the JITI implements an interesting subset of an "ideal" indexing function by only coding the cases shown to be useful. The second constraint is relaxed in systems such as Ciao [3] that can look at the whole program P to understand the possible queries and improve the quality of indexing code.

To the best of our knowledge, there is little work on indexing non-Herbrand constraints in the context of logic programming. We are interested in doing so through an user defined indexer, U^P. Our first observation is that we would not expect U^P to be the only indexer in the system: in general, it must be able to work with a default, system indexer. A simple approach would be to choose one indexing per procedure or query. But, ideally, we would want to have different indexers working together *over the same query.*

We can now state the properties of the user indexer U_i^P: **(i)** it must be correct; **(ii)** it must not perform arbitrarily worse than the default indexer; and **(iii)** it must work together with other indexers.

4.2 Pragmatics

User indexer are constructed and used as a three step process:

1. The programmer declares a predicate Q that will benefit from UDI, at this point the engine will initialize the respective UDI through `udi_init(Q)`;
2. The engine consults a new clause C for an user indexed procedure, the engine will call `udi_extend(C)`;
3. A call to an user-indexed goal, the engine will call `udi_exec(G)`.

Next, we use the `pop/2` example to show our implementation. We shall assume there are two U^P: one for B+-Trees and one for R-Trees.

Declarations. We use *declarations* to inform which U^P will be used by a procedure Q. Each declaration specifies which program-dependent interpretations we will give to the arguments of a procedure. A declaration is as follows:

```
:- udi pop(-,btree(int)).
```

First, the Prolog engine tags the procedure as user indexed. Next, for both U^Ps the engine calls `udi_init(pop(-,btree(int)))`. In the example, the btree indexer will **(i)** initialize a new, empty, B+-Tree of integers for `pop/2`; **(ii)** declare that the key to the tree will be the second argument; and **(iii)** store a pointer to the new tree in a record. This B+-Tree record will be returned to the engine as an opaque *handle*. The engine stores the handle in a table towards fast lookup of all UDI indexers for a procedure.

The rtree indexer will also be called. It will simply consult the declaration, and return `NULL`.

Asserting. Every time a clause for a user indexed predicate has been asserted, and *after* it has been compiled, the Prolog engine searches for UDI records. For each record, it calls `udi_extend()` with a pointer to the term describing the clause, a pointer to the compiled code, and the *handle*. In our example, the btree code would **(i)** recover the tree from the handle; **(ii)** fetch the key from the second argument, given the clause's source; **(iii)** insert a new record new key in the B+-Tree; **(iv)** associate the new record with the clause code. The UDI code then returns control to the engine.

The engine is not informed of what the indexing code does but it does assume the clause code will be respected.

Execution. Currently, we assume that each predicate has at most one user indexer. UDI code is supported by the following YAP instruction:

```
yamop *new = Yap_udi_search(P->u.lp.p);
if (!new) P = PREG->u.lp.l;
else P = new;
JMPNext();
```

The function `Yap_udi_search` receives a pointer to the procedure descriptor and fetches the handle matching the procedure table. It then calls `call_udi` using the handle as argument. If `call_udi` returns a `NULL` pointer YAP will fall back the default indexing code. Otherwise, YAP will execute the code returned by the UDI, which can be:

- a pointer to code, usually clause code;
- a pointer to a set of clauses;
- `NULL`, as explained above;
- `FAIL`: execution just fails.

From the engine point of view, the non-trivial case is when the engine must process a set of clauses. It must construct instructions that can enumerate every clause. In our case, we decided that the constructed object should be discardable on backtracking (or we will risk filling up memory). The current YAP implementation relies on "blobs", or opaque terms, to implement this functionality. Essentially, YAP creates an "opaque term" which just contains WAM code of the form `try-retry-trust`. These objects can be easily stack shifted, but choice-points may point to instructions within the blob, making garbage collection difficult.

5 User Defined Indexers

Next, we propose two constraint systems that rely on the UDI for efficient execution. We will use the following methodology:

- Data will be represented as a standard Prolog database.

- Constraints will be used to represent our queries. Thus we shall follow Datalog with constraints style [11]. Other styles such as HiLog would be possible [12], but we chose Datalog because it has a natural application to our indexing algorithms.
- We shall use UDI code to associate semantics to special procedures.

As explained above a typical query will be as follow:

```
?- Pop #> 5000000, city(X,Pop).
```

set queries can be written in Prolog style:

```
?- setof(Pop, (Pop #> 5000000, city(X,Pop)), Cities).
```

Notice that In this work we are interested in reasonably simple queries *executed on large databases*: indexing, and not constraint propagation, will be fundamental.

In the above example, the first step is to implement the constraint `#>`. YAP supports Demoen's implementation of attributed variables [13]. In this case, the constraint could be set by the following (simplified) code:

```
A '#>' B :-
        attributes:put_att_term(A,range(_,gt(B))),
        attributes:put_att_term(B,range(_,lt(A))).
```

The calls to the `put_att_term` built-in associate variable A and B with the constraint. Following Demoen, the constraint is represented as a compound term. The functor represents the constraint module, or package: in this case, B+-Trees are used to to verify satisfiability of `range` constraints. The first argument chains constraints for different modules on the same variable. The $N - 1$ remaining arguments correspond to the constraints for the same module: in this case, and towards readability, we represent them explicitly.

Notice that the code is simplified: the definition should be symmetric and cumulative and it should handle the cases where either A or B are instantiated.

The second step of execution is `city(X,Pop)`. The UDI code for `udi_exec` is then called and access the arguments of the predicate through the C-interface. The indexer executes as follows:

- Fetch the second argument A_2
- Verify whether A_2 is an attributed variable: if not, return `NULL`.
- Verify whether A_2 contains a term with main functor `range`: if not, return NULL.
- Translate the constraint(s) into a query on B+-trees.
- If the query returns no matching clauses, return `FAIL`
- If the query returns a matching clauses C , return C
- If the query returns several matching clauses, call the C-interface to construct a "blob" that will allow backtracking through the code.

Next we will discuss how to use UDI in our two examples: ranges and vectorial data. They both use trees as indexing structures, and therefore most of their definitions are similar. In both cases, `udi_init` and `udi_extend` are very similar: `udi_init` stores which arguments are indexed and initializes the tree; `udi_extend` will insert the indexed arguments in the trees, saving the clause pointer to use as return value in searches. Each UDI example works in a specific domain so the tree structure and set of constraints will be shown next.

5.1 Ranges

We propose two UDIs for range data: one for integers and one for floating point numbers. We will discuss them together, given the obvious similarity. We support seven constraints, two unary and five binary constraints:

```
 max A    min A    A #> B    A #>= B    A #< B    A #=< B    A #= B
```

In our simplified implementation each constraint term has 6 arguments. One represents a constraint `max`, `min`, and the other four the maximum and minimum limits, and whether we can match that limit. For example, a range query of the form:

```
?- Pop #> 100, Pop #< 1000, city(X,Pop).
```

will result in setting the following range constraint on `Pop`:

```
range(_,false,100,false,1000,false)
```

The UDI code searches for this `range` structure and translates it into a range query returning all values in the database such that their second argument is between 100 and 1000 (if any). If the second argument was set to `max` it would return the maximum value in this range: other queries such as average or mode can easily be implemented.

5.2 Vectorial Terms

The original motivation to this work was our interest in using vectorial terms or spatial terms as defined in previous work [14]. These are simple geometry types based on 2D points. Notice that the simplicity of the primitives does not mean that the terms themselves are simple. For example, the European countries boundaries in Figure 3(b) are represented in Prolog as multipolygons with several hundred points each (and this is a low resolution sample).

Here we use R-Trees as the indexing structure, following the ideas in Section 3. In this work, we will use `overlaps` binary constraint `&&`, the key operator on the Postgis spatial RMS [10]: `A && B` constraint is satisfied if `A`'s bounding box overlaps `B`'s bounding box. A query is shown next:

```
?- country(spain,P1), P2 && P1, country(Country,P2).
```

Here as `P1` is instantiated by the time `P2 && P1` is reached `P2` will be attributed by `overlap(_,P1)`, thus second call of `country` will search the tree succeeding only with Countries that have overlapping boundaries MBR with *spain*.

Notice that the `&&` only approximates overlapping the actual intersection must be performed after. For example:

```
?- country(spain,P1), P2 && P1, country(C,P2),
   intersection(P1,P2,P3).
```

The query searches for countries that intersect with Spain. The overlapping constraint prunes the results to Portugal, France and Andorra, but only the latter will eventually succeed. Notice that the same result would be achieved without the use of UDI, but with a high penalization in time:

```
?- country(spain,P1), country(C,P2), overlap(P1,P2),
   intersection(P1,P2,P3).
```

6 Experiments

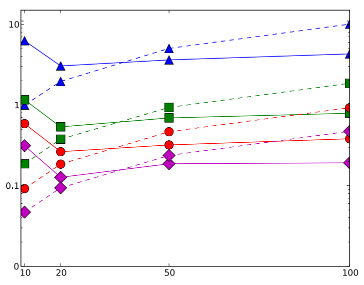
In this section we discuss the performance of our two UDI indexers. We compare against the default JITI indexing in YAP.

6.1 Experimental Setup

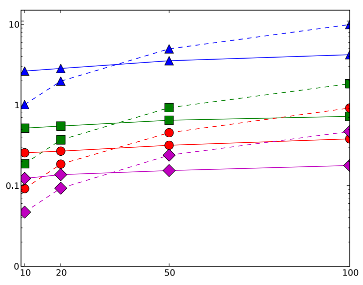
We performed our experiments on a Core 2 Duo P9500 @ 2.53GHz machine with 4GB of memory running Linux 2.6.27 in 64 bit mode.

Our goal in evaluating range queries was to compare Prolog and UDI as we vary selectivity and database size. As a first step, we created four datasets, with sizes between 512K and 10M tuples. Each dataset was filled in with a uniform distribution of random integers in the interval $[1, 100000000]$. Next, we experimented with simple queries, as shown in Figure 4. The first two queries select all values above or below a certain threshold, the third query selects a range in the database. We control how many tuples should be selected: values are 10%, 20%, 50% and 100% of all tuples. Figure 4 shows the execution time in all cases. Notice that we only show the constraint queries, the Prolog queries must be written with the tests after the database call.

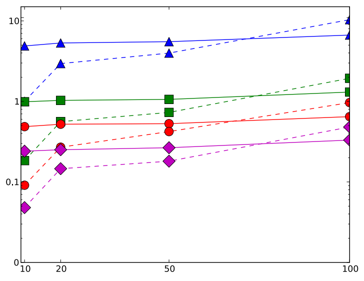
Regarding the evaluation of vectorial data, we are interested in performance on common spatial queries such as: "Find road intersections", "Find railway crosses", "Find road bridges over streams"; in all cases the queries are about overlapping objects. Our methodology was as follows: **(i)** we select two sources of geographical objects S_1 and S_2; **(ii)** for every object O in S_1, we query how many objects in S_2 O overlaps. Figures 5(a) and 5(b) displays the actual Prolog code used in both cases. Notice that we only compare if the object's MBRs overlap in these tests. We use datasets of Germany and California geographical data in these experiments, obtained at http://www.rtreeportal.org. Figure 5(c) shows the results; in Germany we overlap roads with utilities and level lines, and in California we compare roads and streams. Notice that the California roads dataset is over 2.1 million objects whereas Germany largest dataset only have 36k objects: we include the size of each dataset in brackets.



(a) `A #> 100000000*(1-P), a(A,X).`



(b) `A #< 100000000*P, a(A,X).`



(c) `A #> 100000000*(0.5-P/2), A #< 100000000*(0.5+P/2), a(A,X).`

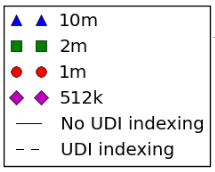


Fig. 4. B+-Tree UDI testing. Times in seconds (Y axis) of UDI versus no UDI, varying the result percentage over dataset size (X axis). Times are given in log scale.

6.2 Results and Discussion

Figure 4 shows that, as expected, in all cases performance of the UDI code varies linearly with the size of the output and with database size. Prolog does not benefit from tuple selection: as a result, performance tends to be independent of output size, although it still varies linearly with dataset size.

If the output size is close to the database size, there is no benefit in using the UDI. In this case, pure Prolog is faster than the UDI code, as it can use static data structures; the UDI has been designed to construct answer lists dynamically. The UDI starts to performs better as the output size decreases, and is up to 10 times faster for 10% output size. Notice that 10% of, say, 10MB is still quite large, so the UDI is doing very well although it is constructing very large data structures, and results will be even better for queries that have very small output sizes.

Figure 5(c) shows even better results for R-Trees. Notice that we just compute overlap, we do not execute spatial operators. In this case, Prolog uses an $O(n \times m)$ algorithm versus the RTrees average $O(n \times (log(m)))$, easily justifying two orders of magnitude speedups. The performance of the R-Tree UDI results is of same order of the magnitude as the results obtained by Postgis, an extension to the well known RDMs PostgreSQL, and the main Open Source RDMs solution in Geographical Information Systems.

```
:- [data1].
:- [data2].

overlap([(X1,Y1),(X2,Y2)],[(X3,Y3),(X4,Y4)]):-
       X1 =< X4, X3 =< X2, Y1 =< Y4, Y3 =< Y4.

:- time((data1(A,B),data2(C,D),
         overlap(B,D),fail)).
```

(a) Native Indexing.

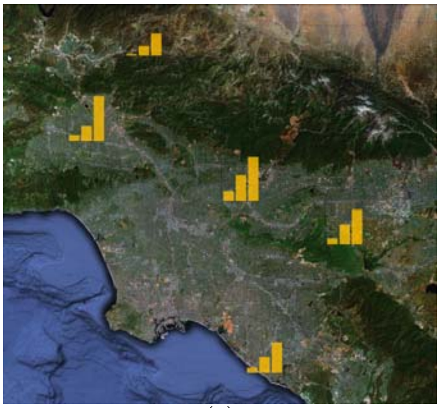
```
:- udi data1(-,rtree).
:- [data1].
:- udi data2(-,rtree).
:- [data2].

:- time((data1(A,B), D && B,
         data2(C,D),fail)).
```

(b) UDI Indexing.

data1	data2	native (a)	UDI (b)	ratio
Germany				
road (30k)	road (30k)	334.165s	0.170s	1965x
utility (17k)	road (30k)	219.259s	0.067s	3272x
road (30k)	utility (17k)	194.000s	0.090s	2155x
rrline (36k)	road (30k)	402.150s	0.106s	3793x
road (30k)	rrline (36k)	416.943s	0.095s	4388x
California				
streams (96k)	roads (2.1m)	81665.457s	13.543s	6030x
roads (2.1m)	roads (2.1m)	n.a.	80.989s	

(c) Results

Fig. 5. R-Tree UDI testing



(a)

```
:- udi roads(-,rtree).
:- [roads].
:- [earthquakes].

in_danger(ID,Count) :-
        earthquakes(ID,Epicenter),
        e_area(Epicenter,D),
        findall(ID2,
             (R && D, roads(R,ID2)),
                L),
        length(L,Count).
```

Earthquake	20km	40km	60km
1933	41.038	123.956	236.430
1971	12.799	75.223	173.749
1987	75.797	202.980	343.914
1994	45.816	117.734	206.294
2008	46.604	154.739	268.228

(b)

Fig. 6. Los Angeles five major earthquakes

6.3 Example Application

Our work in the UDI has been motivated by previous work [14] in the geographical viewer `simplegraphics`, where as the user zooms-in we need to display fewer objects or might display the objects in view in greater detail. In general, the ability to quickly prune spatial objects is fundamental for performance in these applications. Next, we show an example of how a straightforward logic program can be at the heart of a geographical querying system.

Los Angeles is subject to earthquakes on a daily basis, due to its location in the Pacific Ring of Fire. We have gathered the location of the last five major earthquakes, shown in Figure 6. By using a roads database and different ranges of action we can estimate how many roads could be affected by a new earthquake. The predicate `in_danger` specifies the problem and we use a simple graphical interface based on the site `http://maps.google.com` to depict the results, as shown in Figure 6. The UDI execution mechanism takes less than a second to run these queries, hence allowing us to quickly experiment with different epicenters and with different ranges. Moreover, this small program can be easily adapted as other sources of information, such as population counts, become available.

7 Conclusions

The use of Prolog as a general-purpose language, solving a wide variety of different problems, is clearly not limited by the expressiveness of logic terms. Declarative and intuitive representations of entities such as range intervals over reals or vectorial representations of spatial objects are easily expressed as logic terms. Such conceptual efficiency is naturally expected in a language built around the motto of the *"what"*. The *"how"*'s efficiency, however, is almost completely left, in Prolog, to sophisticated compilation techniques, where indexing fits. We argued, in this paper, that the current state-of-the-art of this Prolog indexing, disconnected from *what* a term represents, can entangle the general use of the language with data-intensive problems from novel domains, such as vectorial spatial databases. Our results provide unequivocal evidence of the advantages of UDI, showing real-world queries that take hours to execute based on hash-based indexing, and are completed within tenths of a second when suitable indexing is used.

Our proposal of UDI, coupled to a declarative constraining of logic variables, is able to allow the user to redefine *how* indexing is to be done for a particular predicate, based on *what* the arguments of the predicate represent, from scalar values to multi-dimensional objects. It is clear that UDI provides the user some explicit control over the procedural execution of Prolog code, which is very much justifiable when such explicit control is able to improve the efficiency of Prolog programs by several orders of magnitude, allowing an efficient handling of data in novel areas of application.

Acknowledgments

This work has been partially supported by projects JEDI (PTDC/EIA/66924/2006) and STAMPA (PTDC/EIA/67738/2006) and funds granted to LIACC, CRACS and IT through the Programa de Financiamento Plurianual, Fundação para a Ciência e Tecnologia and Programa POSC. David Vaz is funded by FCT PhD grant SFRH/BD/29648/2006.

References

1. Warren, D.H.D.: Implementing Prolog - Compiling Predicate Logic Programs. Technical Report 39 and 40, Department of Artificial Intelligence, University of Edinburgh (1977)

2. Warren, D.H.D.: An Abstract Prolog Instruction Set. Technical Note 309, SRI International (1983)
3. Hermenegildo, M., Bueno, F., Puebla, G.: The CIAO multi-dialect compiler and system: An experimentation workbench for future (C) LP systems. In: Parallelism and Implementation of Logic and Constraint Logic Programming (1999)
4. Puebla, G., Stuckey, P.: Optimization of logic programs with dynamic scheduling. In: Logic Programming: Proceedings of the Fourteenth International Conference on Logic Programming, MIT Press, Cambridge (1997)
5. Demoen, B., Mariën, A., Callebaut, A.: Indexing prolog clauses. In: NACLP, pp. 1001–1012 (1989)
6. Wielemaker, J.: SWI-Prolog 5.5: Reference Manual. SWI, University of Amsterdam, Roetersstraat 15, 1018 WB Amsterdam, The Netherlands (2008)
7. Santos Costa, V., Sagonas, K., Lopes, R.: Demand-driven indexing of prolog clauses. In: Dahl, V., Niemelä, I. (eds.) ICLP 2007. LNCS, vol. 4670, pp. 395–409. Springer, Heidelberg (2007)
8. Comer, D.: The ubiquitous b-tree. ACM Comput. Surv. 11(2), 121–137 (1979)
9. Guttman, A.: R-trees: A dynamic index structure for spatial searching. In: Yormark, B. (ed.) SIGMOD 1984, Proceedings of Annual Meeting, Boston, Massachusetts, June 18-21, pp. 47–57. ACM Press, New York (1984)
10. The Postgis Development Team: Postgis adds support for geographic objects to the postgresql object-relational database, `http://postgis.refractions.net/`
11. Revesz, P.: Introduction to constraint databases. Springer, New York (2002)
12. Chen, W., Kifer, M., Warren, D.S.: Hilog: A foundation for higher-order logic programming. J. Log. Program. 15(3), 187–230 (1993)
13. Demoen, B.: Dynamic attributes, their hprolog implementation, and a first evaluation. Technical report, Department of Computer Science, K.U.Leuven, Leuven, Belgium (2002)
14. Vaz, D., Ferreira, M., Lopes, R.: Spatial-yap: A logic-based geographic information system. In: Dahl, V., Niemelä, I. (eds.) ICLP 2007. LNCS, vol. 4670, pp. 195–208. Springer, Heidelberg (2007)

Integration of Abductive Reasoning and Constraint Optimization in SCIFF

Marco Gavanelli[1], Marco Alberti[2], and Evelina Lamma[1]

[1] ENDIF, University of Ferrara
[2] CENTRIA, Universidade Nova de Lisboa
http://www.ing.unife.it/docenti/MarcoGavanelli
http://pessoa.fct.unl.pt/m.alberti
http://www.ing.unife.it/docenti/elamma.htm

Abstract. Abductive Logic Programming (ALP) and Constraint Logic Programming (CLP) share the feature to constrain the set of possible solutions to a program via integrity or CLP constraints. These two frameworks have been merged in works by various authors, who developed efficient abductive proof-procedures empowered with constraint satisfaction techniques. However, while almost all CLP languages provide algorithms for finding an optimal solution with respect to some objective function (and not just *any* solution), the issue has received little attention in ALP.

In this paper we show how optimisation meta-predicates can be included in abductive proof-procedures, achieving in this way a significant improvement to research and practical applications of abductive reasoning.

In the paper, we give the declarative and operational semantics of an abductive proof-procedure that encloses constraint optimization meta-predicates, and we prove soundness in the three-valued completion semantics. In the proof-procedure, the abductive logic program can invoke optimisation meta-predicates, which can invoke abductive predicates, in a recursive way.

1 Introduction

Abductive Logic Programming (ALP) [1] is a set of programming languages deriving from Logic Programming. In an abductive logic program, a distinguished set of predicates, called *abducibles*, do not have a definition, but their truth value can be *assumed*. A set of formulae, called *Integrity Constraints* (IC, often in the form of implications) restrict the set of hypotheses that can be made, in order to avoid unrealistic hypotheses.

ALP is interesting as it supports hypothetical reasoning, and in the context of logic programming it supports a simple, sound implementation of negation by failure [2] (also called, in the context of ALP, negation by default), that is useful in many applications, such as planning [3]. Operationally, various abductive proof-procedures have been proposed in the past, and they have recently gained significant efficiency [4,5,6,7,8,9,10].

Many abductive proof-procedures are integrated with Constraint Logic Programming (CLP) [6,8,10]. However, while most CLP languages have optimisation meta-predicates, the issue has received little attention in the ALP community. Notably, van Nuffelen and Denecker [11] performed interesting experiments with ALP and aggregates (optimization is, in fact, an instance of aggregate meta-predicate), but, to the best of our knowledge, there is no proof of correctness for their procedure with aggregates. On the other hand the importance of aggregates in ALP is well motivated in [11], as the ALP solution is compared with a pure CLP solution: *"the CLP solution is a long program (400 lines) developed in some weeks of time [. . .], where as the above representation is a simple declarative representation of 11 logical formulae, written down after some hours of discussion."* The possibility of reducing the development time from weeks to hours is very attractive. On the other hand, a logic programming language without a convincing declarative semantics and a proof of soundness is incomplete.

In this paper, we show some subtleties of a naive declarative semantics for abduction with optimization, and propose a new declarative semantics, that overcomes those problems. Some of the subtleties have already been identified in CLP [12,13], others are particular of the ALP framework. We propose an operational semantics, that extends the $\mathcal{S}$CIFF proof-procedure [14], and that recalls the branch-and-bound procedure used by CLP solvers. Thanks to the new declarative semantics, we are able to show that the new proof-procedure, called $\mathcal{S}\text{CIFF}^{opt}$, is sound with respect to the three-valued completion semantics, in practical cases (those without floundering). In particular, we are able to deal with recursion through the optimization predicates: the abductive proof-procedure can invoke optimization predicates, which in turn can perform abductive reasoning. Recursion through optimization lets us deal with problems of games, that are typically in PSPACE, as we showed in a short version of this paper [15].

Finally, we show that the implementation of $\mathcal{S}$CIFF does not need to be extended to deal with optimization. This result comes from the choice of implementing $\mathcal{S}$CIFF in Constraint Handling Rules (CHR) [16], and we believe that the results proved here could be directly applicable to other CHR-based abductive proof-procedures [17,18].

2 The $\mathcal{S}$CIFF Proof-Procedure

$\mathcal{S}$CIFF is an abductive proof-procedure that follows the classical semantics of ALP with constraints. An ALP with constraints [6] is formally defined as a triple $\mathcal{P} \equiv \langle KB, \mathcal{A}, IC \rangle$, where KB is the knowledge base (a logic program), $\mathcal{A}$ is a distinguished set of predicates, called *abducibles*, and IC is a set of implications, called Integrity Constraints. With abuse of notation, we will use $\mathcal{A}$ also for the set of ground atoms built on the abducible predicates. Given a goal $\mathcal{G}$, the aim of abduction is to find an abductive answer, i.e., a pair $(\varDelta, \theta)$, where $\varDelta$ is a set $\varDelta \subseteq \mathcal{A}$ and θ is a substitution, such that

$$KB \cup \varDelta \cup \mathcal{T} \models \mathcal{G}\theta \wedge IC \tag{1}$$

where $\mathcal{T}$ is the theory of constraints [19], and will be omitted for simplicity in the following. Although most of the results are general, in the examples we will use a constraint sort on finite domains (CLP(FD)). Abducibles are in **bold**.

3 Syntax and Preliminaries

The optimization meta-predicates are the main objective of this work. The following syntax will represent an atom with three arguments:

$$min(X : G) = V$$

meaning that we are looking for the solution to the goal G that gives the minimal value to variable X; such value is V. When the value of the optimal solution is not of interest, we adopt the simplified syntax $min(X : G)$ (one can think of this simplified syntax as if adding an unnamed variable, as in Prolog: $min(X : G) = \_$). Of course, we have a symmetric meta-predicate max. We will use upper-case letters for variables and lower-case for predicates and constants (as in Prolog).

4 A Naive Declarative Semantics

The first intuition of a declarative semantics for abduction with optimization is to start from the declarative semantics of abduction itself (Eq 1) which states that, given an abductive program $\mathcal{P} \equiv \langle KB, \mathcal{A}, IC \rangle$, one's goal is to find a set $\Delta \subseteq \mathcal{A}$ of abducibles that (together with the knowledge base KB) entails both the goal G and the integrity constraints IC. I.e., we ask ourselves if there exists such a set Δ.

In the simplest possible situation, the optimization meta-predicate occurs only in the goal, e.g., $G \equiv min(X : p(X)) = V$, meaning that we want to find the minimal value V for variable X such that predicate $p(X)$ is true. The temptation is to adopt the same idea used to give semantics to optimization predicates in CLP, namely to rewrite $min(X : p(X)) = V$ as $p(V) \wedge not(\exists Y p(Y) \wedge Y < V)$, i.e., V is indeed the minimum if p is true and there is no smaller value Y that makes p true.

Now, combining abduction and optimization we would obtain:

$$KB \cup \Delta \models p(V) \wedge not(p(Y) \wedge Y < V) \wedge IC \tag{2}$$

This formalisation is very intuitive, but it does not provide a semantics usable in practical situations. The meaning of equation 2 is *"Do there exist a set Δ and a value V such that $p(V)$ is true and no other value Y smaller than V makes p true?"* Let us now consider a very simple abductive program:

$$p(X) \leftarrow \mathbf{a}(X) \wedge 1 \leq X \leq 2. \tag{3}$$

without integrity constraints. In this case, the declarative semantics in Eq. 2 would provide the following answers to the goal $min(X : p(X)) = V$:

$$\begin{array}{ll} \Delta_1 = \{\mathbf{a}(1)\} & V = 1 \\ \Delta_2 = \{\mathbf{a}(2)\} & V = 2 \\ \Delta_3 = \{\mathbf{a}(1), \mathbf{a}(2)\} & V = 1. \end{array}$$

We find such an answer counter intuitive, in particular Δ_2, and not in the direction of real-life applications. In the semantics of Eq. 2, for each set Δ that supports $p(X)$ we have a positive answer; the value V is simply the minimum value amongst the abduced literals. Classical applications of ALP are diagnosis, and planning; by combining abductive reasoning with optimization, one would expect to be able to answer to questions like *"What is the plan of minimal cost?"* or *"What is the explanation of maximal likelihood?"*, which means that the user wants to find *the optimal set* Δ, not that she wants to find any explanation Δ, and then take the minimal value that makes true a predicate with such assumptions. More formally, the intended meaning is not

$$(\exists \Delta, V) \qquad KB \cup \Delta \models p(V) \wedge not(\exists Y.p(Y) \wedge Y < V) \wedge IC$$

which means that given a set Δ that satisfies p and IC, V is the optimal value in that particular Δ, but that the set Δ should be in the scope of optimization, as in:

$$\begin{aligned}(\exists V, \Delta^*)\ [\quad & KB \cup \Delta^* \models p(V) \wedge IC \\ & \wedge\ not(\exists Y, \Delta'. Y < V \wedge KB \cup \Delta' \models p(Y) \wedge IC)]\end{aligned} \tag{4}$$

Of course, Eq 4 is meaningful only for optimization atoms that occur in the goal, but it does not give a semantics to general ALPs that contain optimization atoms in the KB or in the IC.

Starting from the practical need to combine abduction and constraint optimization, we propose a new declarative semantics for ALP with optimization. We ground our semantics on the $\mathcal{S}$CIFF language and proof-procedure [14], but we believe that our results could be easily extended to other proof-procedures.

5 Declarative Semantics

The declarative semantics is given, as usual, with respect to the ground program. When there are optimization literals, defining the grounding of a program is not immediate; we adopt the same definitions by Faber et al. [20], adapted to the $\mathcal{S}$CIFF syntax.

Definition 1. *A* Set Term *is either a symbolic or a ground set. A* Symbolic Set *is a pair* $\{V : Conj\}$*, where* V *is a variable and* $Conj$ *is a conjunction of atoms. A* Ground Set *is a set of pairs* $\langle t : Conj\rangle$*, where* t *is a numeric constant and* $Conj$ *is a ground conjunction of atoms.*

Definition 2. *An* Optimization Atom *is either of the form* $min(S) = V$ *or* $max(S) = V$*, where* S *is a set term.*

We will suppose for simplicity that optimization atoms occur only in the body of clauses, and not in Integrity Constraints.

Definition 3. *Given a clause, a* local variable *is a variable that occurs only in an optimization atom. All other variables are* global.

Definition 4. *Given a symbolic set without global variables* $S = \{V : Conj\}$*, the* instantiation of S *is a ground set of pairs* $\{\langle\gamma(V) : \gamma(Conj)\rangle | \gamma$ *is a substitution for the local variables in* $S\}$.

A Ground Instance *of a clause* r *is obtained in two steps:*

1. *all global variables are grounded*
2. *every symbolic set is replaced by its instantiation*

After defining the grounding of a program, we can give it semantics. We will restrict ourselves to *locally stratified programs*. Note that local stratification does not prevent the user to use recursion through optimization.

Definition 5. *A ground program is* locally stratified with respect to optimization *if there exists a level mapping* $||\cdot|| : \mathcal{H} \mapsto \mathbb{N}$ *(where* $\mathcal{H}$ *is the Herbrand Base) such that for each pair of ground atoms* h *and* b *occurring, respectively, in the head and in the body of a clause:*

– *if* b *occurs in the clause in an optimization atom, then* $||b|| < ||h||$
– *otherwise,* $||b|| \leq ||h||$.

If such a mapping exists, then the set of ground atoms in the Herbrand base is partitioned into levels. Suppose for simplicity that the levels take the values of the first natural numbers (so the first level takes value 0).

5.1 3-Valued Completion for Non-abductive Programs

We extend the 3-valued completion semantics [21,22] to the case with optimization meta-predicates. A (partial) interpretation I is a set of literals considered true. A literal p is false in I iff $\neg p \in I$. If $\{p, \neg p\} \cap I = \emptyset$, then p's truth value is *unknown* ($\perp$).

We report the extension of the T_P operator to the three-valued case:

Definition 6. *Consider an atom* p *of a defined predicate*

– $p \in T_P(I)$ *iff there is some instantiated clause* $R \in P$ *such that* R *has head* p*, and each subgoal literal in the body of* R *is true in* I.
– $p \in U_P(I)$ *iff for all clauses* $R \in P$ *that have head* p*, the body of the clause is false in* I.
– $W_P(I) = T_P(I) \cup \neg \cdot U_P(I)$*, where* $\neg \cdot U_P(I)$ *means the negation of all atoms in* $U_P(I)$ *(i.e., if atom* $a \in U_P(I)$*, then* $\neg a \in W_P(I)$*).*

Notice that Definition 6 gives a truth value only to literals defined in KB, other literals (abducibles, optimization atoms) have still unknown $\perp$ truth value.

T_P, U_P and W_P are monotonic transformations (i.e., $T_P(I) \subseteq T_P(J)$ whenever $I \subseteq J$), so considering the limit makes sense. For a transformation Φ, let [21]

– $\Phi \uparrow^0 (S) = S$,
– $\Phi \uparrow^{\alpha+1} (S) = \Phi(\Phi \uparrow^{\alpha} (S))$
– for limit ordinals λ, $\Phi \uparrow^{\lambda} (S) = \bigcup_{\alpha<\lambda} \Phi \uparrow^{\alpha} (S)$

Let $I_0 = \emptyset$, $I_\alpha = W_P \uparrow^{\alpha} (\emptyset)$, $I_\infty = W_P \uparrow^{\omega} (\emptyset)$, where ω is the first limit ordinal.

5.2 3-Valued Completion Semantics for Abductive Programs

Since $\mathcal{P}$ is an ALP, the truth of an atom depends on the assumed hypotheses. We consider, in the declarative semantics, all the possible groundings of abducible literals that satisfy the integrity constraints. Let $I_0(\Delta)$ the 3-valued interpretation corresponding to the set of abduced atoms Δ, i.e., $\forall a \in \Delta$, $a \in I_0(\Delta)$ and $\forall a \in \mathcal{A} \setminus \Delta$, $\neg a \in I_0(\Delta)$.

Let $I_\infty(\Delta) = W_P \uparrow^{\omega} (I_0(\Delta))$. As stated earlier, $I_\infty(\Delta)$ assigns value $\perp$ to all optimization atoms. Let us suppose that the program is stratified also with respect to negation [23]; in such a case, the three-valued completion semantics gives values true-false to each atom (never unknown), so the only unknown atoms are the optimization atoms and the atoms that depend on them.

If the program is locally stratified with respect to optimization, there will be an optimization atom $min(S) = V_m$ such that $\forall \langle V : C \rangle \in S$, C is not $\perp$ (i.e., there will be an optimization atom of minimum level).

We now define a new operator that gives semantics to the optimization atoms.

5.3 Extension of 3-Valued T_P for Optimization Atoms

Before defining the extension of the T_P operator to the optimization atoms, consider the following example, in CLP (without abduction):

$$min(X : max(Y : p(X, Y))) = V$$

Intuitively, this problem can be thought as a two player game: given that the possible solutions are those that satisfy predicate $p(X, Y)$, the first player tries to minimize X while the second maximizes Y. The point here is that the first player has control over the variable X, while the second instantiates variable Y: player 2 cannot choose the value of variable X. This means that we need to exclude some variables from the maximization; for this reason, various authors [12,13] proposed to extend the syntax, in order to let the user choose which variables are subject to optimization and which are not, by providing *protected variables* [13] to the optimization meta-predicate.

In ALP, we have the same issue, and a further one: which of the two nested atoms is responsible for grounding the set Δ? We would like both invocations to be able to abduce literals, otherwise the expressivity of our language would be strongly compromised: in fact, we would boil down to a two step procedure, and lose the possibility of recursion through optimization predicates. We decided to explicitly communicate to the optimization atom those *literals* it is responsible to abduce. We show in the following of this section that there is a precise declarative semantics.

We extend the syntax of the optimization meta predicate:

$$min_{\mathcal{A}_m}(X : p(X)) = V \tag{5}$$

the intuitive meaning is that we are looking for the minimum value for X such that $p(X)$ is true, knowing that in such minimization we are entitled to abduce only the literals occurring in the set $\mathcal{A}_m \subseteq \mathcal{A}$. Since the set $\mathcal{A}_m$ could be infinite, we sometimes represent its content with non-ground atoms, meaning that all possible groundings belong to $\mathcal{A}_m$.

We can now define precisely the declarative semantics of the optimization meta-predicate in the 3-valued completion semantics, by extending the T_P operator:

Definition 7. *A non-optimization literal $l \in M_P(I)$ iff $l \in W_P(I)$.*

Otherwise, consider the set of all the possible groundings of the abducible literals that satisfy the integrity constraints: $Cons = \{\Delta \in 2^{\mathcal{A}} | KB \cup \Delta \models IC\}$. Consider one specific set Δ^* (intuitively, the candidate set of abduced literals).

The atom $min_{\mathcal{A}_m}(S) = V_m$ is true, i.e., $min_{\mathcal{A}_m}(S) = V_m \in M_P(I(\Delta^*))$, iff all the following conditions hold:

1. *$\forall \Delta \in Cons$, $\forall \langle V : C\rangle \in S$, either $I(\Delta) \models C$ or $I(\Delta) \models \neg C$ (intuitively, C is not $\perp$ in all the possible Δ)*
2. *there exists some $\langle V : C\rangle \in S$ such that $V = V_m$ and*
   (a) *$I(\Delta^*) \models C$*
   (b) *$\nexists \langle V' : C'\rangle \in S$ s.t. $I(\Delta^*) \models C'$ and $V' < V$*
   (c) *$\nexists(\langle V'' : C''\rangle \in S$ and $\Delta'' \in Cons)$ such that*
      i. *$\Delta'' \cap (\mathcal{A} \setminus \Delta^*) \subseteq \mathcal{A}_m$*
      ii. *$\Delta'' \cap (\mathcal{A} \setminus \mathcal{A}_m) = \Delta^* \cap (\mathcal{A} \setminus \mathcal{A}_m)$*
      iii. *$I(\Delta'') \models C''$ and $V'' < V$*

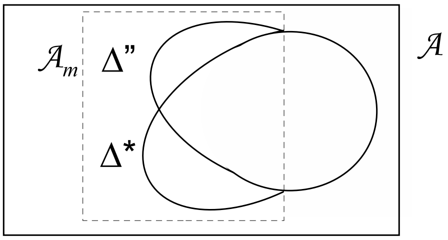


Fig. 1. Relation of sets in Definition 7

The atom $min_{\mathcal{A}_m}(S) = V_m$ *is false, i.e.,* $\neg(min_{\mathcal{A}_m}(S) = V_m) \in M_P(I(\Delta^*))$ *iff condition 1 holds, but at least one of the other conditions does not hold.*

Intuitively, condition 2b imposes that no better solution exists in the same Δ^*, while condition 2c requires that a better solution does not exist in a different set Δ''. Such set Δ'' cannot be completely different from Δ^*, as in this specific optimization we are supposed to optimize only with respect to $\mathcal{A}_m$, and not with respect to the whole set of abducibles $\mathcal{A}$. In the current optimization, we will only abduce literals in $\mathcal{A}_m$, while the other literals (in $\mathcal{A} \setminus \mathcal{A}_m$) can be abduced externally. For this reason, Δ'' coincides with Δ^* for the part in $\mathcal{A} \setminus \mathcal{A}_m$, and differs only for the part in $\mathcal{A}_m$ (Figure 1).

The operator M_P is monotonic, and gives a truth value to optimization predicates that contain only conditions whose truth value is known. In other words, if one knows the truth value of the argument of min, he can define the truth of min through the M_P operator. Otherwise, the min literal remains unknown.[1] Note that we require such knowledge for all the possible Δ that satisfy the integrity constraints (set $Cons$).

Given a set Δ, we can apply the operator M_P up to its fix-point; if a ground literal $a \in M_P \uparrow^\omega (I_0(\Delta))$, we write

$$\mathcal{P} \models^{opt}_{\Delta} a.$$

6 $\mathcal{S}$CIFF^{opt} Operational Semantics

The $\mathcal{S}$CIFF proof-procedure consists of a set of transitions that rewrite a node into one or more children nodes. It encloses the transitions of the IFF proof-procedure [7], and extends it in various directions. We recall the basics of $\mathcal{S}$CIFF; a complete description is in [14], with proofs of soundness, completeness, and termination.

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

Unfolding replaces a (non abducible) atom with its definitions;
Propagation if an abduced atom $a(X)$ occurs in the condition of an IC (e.g., $a(Y) \rightarrow p$), the atom is removed from the condition (generating $X = Y \rightarrow p$);

[1] This condition could be relaxed; e.g., $min(\langle 1, p \rangle) = 2$ is obviously false even if we do not know the truth of atom p.

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

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

Logic simplifications other simplifications like $true \rightarrow A \Leftrightarrow A$, etc.

$\mathcal{S}$CIFF includes also the transitions of CLP [19] for constraint solving.

We introduce a new transition to process optimisation meta-predicates.

In order to simplify the exposition, we assume that the goal argument of min constrains the value of the objective function to become ground in each leaf node, as in many practical implementations of CLP (see, e.g., SICStus manual). We plan to remove such assumption in future work, by considering *bounds* as in the operational semantics of optimization in CLP [13].

Definition 8 (Transition Optimize). *Given a node*

$$T' \equiv \langle R', CS', PSIC', \Delta' \rangle,$$

such that the resolvent R' contains exactly an optimization literal, i.e.,

$$R' = \{min_{\mathcal{A}_m}(F : G) = V\},$$

transition Optimize *opens a new $\mathcal{S}$CIFF derivation tree with starting node*

$$T^{opt} \equiv \langle G, CS', PSIC', \Delta' \rangle.$$

We call the derivations spawning from node T^{opt} sub-derivations. *If all sub-derivations finitely fail, then the successor of T' is the special node $false$.*

Otherwise, let S the set of leaf nodes of the sub-derivations starting from node T^{opt}. For each leaf node $N_j \in S$ (Figure 2) one can compute the value of F for the node. As in CLP, we can call F the objective function *and write, with an abuse of notation, $F(N_j)$ to indicate its value in a node.*

If there exists a node $N_k \in S$ in which the set of abduced literals Δ_k contains new literals not included in $\mathcal{A}_m$ (i.e., $\Delta_k \not\subseteq \mathcal{A}_m \cup \Delta'$), the derivation flounders.

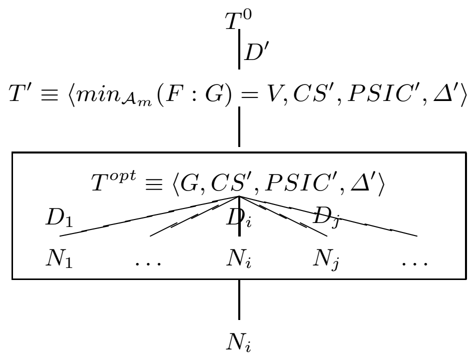

T^0
D'
$T' \equiv \langle min_{\mathcal{A}_m}(F : G) = V, CS', PSIC', \Delta' \rangle$
$T^{opt} \equiv \langle G, CS', PSIC', \Delta' \rangle$
D_1
D_i
D_j
N_1 ... N_i N_j ...
N_i

Fig. 2. $\mathcal{S}$CIFF^{opt} derivation

Note also that transition Optimize *is not applicable if R' contains more than one literal. In case no transitions are applicable and in the last node the resolvent is not empty, the derivation flounders (this can happen, e.g., in case the goal contains a conjunction of minimization literals).*

If there is a successful sub-derivation D_i starting from node T^{opt} with final node $N_i \in S$, and value $F^* \equiv F(N_i)$, then to the constraint store of each node $N_j \in S$ a new constraint $F(N_j) \leq F^*$ is added. To such nodes, constraint propagation is applied, and it can possibly fail. In case of success in some node N_k, again the new value $F_2^* \equiv F(N_k)$ is computed, and the new constraint $F(N_l) \leq F_2^*$ is added to the store of all the remaining nodes N_l. This process continues until the fix point. The final nodes of the successful sub-derivations are then generated as children of the node T'.

If the $\mathcal{S}$CIFF^{opt} operational semantics (including transition *Optimize*) has a successful (non floundering) derivation with abductive answer (Δ, σ) for a goal G and a program $\mathcal{P}$ we write

$$\mathcal{P} \vdash^{opt}_{\Delta\sigma} G\sigma.$$

Example 1. Consider the program in Eq. 3, with the goal $min_{\{\mathbf{a}(\_)\}}(X : p(X)) = V$. The only applicable transition is *Optimize*, which opens a new derivation for the goal $p(X)$. The $\mathcal{S}$CIFF proof-procedure has two possible derivations, with nodes (we report for simplicity only the sets of abduced literals): $\Delta_1 = \{\mathbf{a}(1)\}$, $\Delta_2 = \{\mathbf{a}(2)\}$. *Optimize* chooses one of them: suppose Δ_2. $F(\Delta_2) = 2$, so the new constraint $X \leq 2$ is added to all nodes; constraint propagation does not exclude any value nor causes failure. Another node is selected: Δ_1. $F(\Delta_1) = 1$, and the constraint $X \leq 1$ is added to all nodes. This causes a failure in Δ_2, since $2 \not\leq 1$, so the only viable solution is the node containing Δ_1. Such node is reported as child of the initial goal, with $V = 1$. No other transition is applicable, so we have success with $V = 1$

The optimization predicates are postponed after the others, because they can be safely applied at the end of a derivation. This is not an issue in many practical applications: for example, when solving constrained optimization problems in CLP, one first imposes constraints and then performs a search. Of course, in some cases postponing the optimization goal can lower efficiency. Another way to deal with such problem is using protected variables [13]; in this paper we do not use them to simplify the exposition, and leave for future work the integration of protected variables into our syntax and semantics.

Note that this limitation does not prevent recursion through optimization: when transition *Optimize* is applied, a new derivation starts, and optimisation can be applied inside it (it is a different derivation).

Example 2. Consider the goal $G_1 \equiv min_{\{\mathbf{a}(\_)\}}\left(X : min_{\{\mathbf{a}(\_)\}}(Y : p(X,Y))\right)$, with:

$$p(X,Y) \leftarrow 0 < X < Y, \mathbf{a}(X,Y).$$

Start D^1**:** Transition *Optimize* is applicable, and a new derivation tree is generated, rooted in $G_2 \equiv min_{\{\mathbf{a}(\_)\}}(Y : p(X,Y))$.

Start D^2**:** *Optimize* is applicable in the new derivation, and generates a new tree rooted in $G_3 \equiv p(X,Y)$.

Start D^3**:** G_3 is solved with the $\mathcal{S}$CIFF transitions, that in particular abduce $\mathbf{a}(X, Y)$. The best solution in this subtree is node $Y = 1$, $X = 0$, $\mathbf{a}(0, 1)$.
End D^3**:** Derivation D^3 terminates, so there is no floundering in it.
End D^2**:** No transitions after *Optimize*, so no floundering.
End D^1**:** no floundering.

We are not currently dealing with the case in which there is an optimisation sub-goal in the condition of an integrity constraint. We leave this issue for future research; if at the end of a derivation there is an implication with a minimization sub-goal in the condition, the derivation *flounders*.

7 Soundness

We will rely on the following theorems, proved in [14]:

Theorem 1 (Soundness of $\mathcal{S}$CIFF). *Given an abductive logic program $\mathcal{P}$, if $\mathcal{P} \vdash_\Delta \mathcal{G}$ with abductive answer (Δ, σ), then $\mathcal{P} \models_{\Delta\sigma} \mathcal{G}\sigma$*

Theorem 2 (Completeness of $\mathcal{S}$CIFF). *Given an abductive logic program $\mathcal{P}$, a (ground) goal G, for any ground set Δ such that $\mathcal{P} \models_\Delta \mathcal{G}$ then $\exists \Delta'$ such that $\mathcal{P} \vdash_{\Delta'} G$ with an abductive answer (Δ', σ) such that $\Delta'\sigma \subseteq \Delta$.*

We can now give the main result:

Theorem 3 (Soundness of $\mathcal{S}$CIFF^{opt}). *Given an abductive logic program $\mathcal{P}$ with optimization predicates, that is locally stratified both with respect to negation and to optimization, the following results hold:*

1. **(Soundness of success)** *if*
$$\mathcal{P} \vdash^{opt}_\Delta \mathcal{G}$$
with abductive answer (Δ, σ), then
$$\mathcal{P} \models^{opt}_{\Delta\sigma} \mathcal{G}\sigma$$
2. **(Soundness of failure)** *if the $\mathcal{S}$CIFF^{opt} derivation for a goal G finitely fails, then*
$$\mathcal{P} \not\models^{opt} \mathcal{G}$$

Proof. Suppose that the derivation does not contain applications of the *Optimize* transition. In this case, the thesis follows immediately from the soundness and completeness of the $\mathcal{S}$CIFF proof-procedure (Theorems 1 and 2).

Note that a non-floundering $\mathcal{S}$CIFF^{opt} derivation contains at most one application of the *Optimize* transition, otherwise the second application would make the derivation flounder. However, each $\mathcal{S}$CIFF^{opt} derivation can generate new sub-derivations (each possibly containing an application of *Optimize*). We call *full derivation* the forest of derivations triggered by a goal, including all sub-derivations for optimisation sub-goals.

By induction, suppose that all $\mathcal{S}$CIFF^{opt} full derivations of depth n (i.e., we have n levels of application of the *Optimize* transition) are sound (satisfy conditions 1 and

2). Consider a $\mathcal{S}$CIFF^{opt} derivation D that generates sub-derivations of depth up to n. The sub-derivations are sound by inductive hypothesis. The derivation D consists of a $\mathcal{S}$CIFF derivation D' followed by the application of the *Optimize* transition (Figure 2). Derivation D' is a $\mathcal{S}$CIFF derivation, that is sound and complete [14]. The *Optimize* transition is applied only to a node with a min goal, that must be the only element in the resolvent (otherwise other transitions would be applicable after *Optimize*). Let $T' \equiv \langle min_{\mathcal{A}_m}(F : G) = V, CS', PSIC', \Delta' \rangle$ the final node of D' (Figure 2).

Consider a sub-derivation D_i (with goal G); let N_i the final node of D_i. D_i is sound by inductive hypothesis.

Suppose that transition *Optimize* generates node N_i as successor of T'; we prove that all the conditions in Definition 7 hold.

Condition 1 holds because the program is stratified with respect to both negation and optimization. Since the program is stratified with respect to negation, the tree-valued completion semantics has a unique model, and no literal has unknown truth value.

Condition 2a requires the derivation D_i to be sound: this holds because of the inductive hypothesis.

Condition 2b requires that there is no other substitution θ' that, together with the same set of hypotheses Δ^*, provides a better value for V. In fact, if there existed a substitution θ' supporting a value $V' < V$, then (Δ^*, θ') would be an abductive answer to the goal $G \wedge F < V$ (i.e., $\mathcal{P} \models_{\Delta^* \theta'} (G \wedge F < V)\theta'$). But the goal $G \wedge F \leq V$ is the initial node of another sub-derivation, call it D_j. If D_j succeeded with a value $V' < V$, then transition *Optimize* would have added the constraint $F(N_i) \leq V'$ to the node N_i, which would have failed (contradicting the hypothesis that N_i is the successor of T'). If D_j failed, then, since for inductive hypothesis the soundness of failure holds for the sub-derivations, there is no abductive answer that supports the goal $G \wedge F \leq V$. Otherwise, D' may succeed with $V' = V$, but this does not contradict the assumption that V is one of the optima.

Condition 2c holds again due to the soundness of failure. It requires that there is no other substitution θ'' that, together with a different set of hypotheses Δ'' provides a better value V''. Moreover, the candidate set Δ'' can differ from Δ^* only for the literals in $\mathcal{A}_m$ (see Definition 7).

Suppose, by contradiction, that there is a set Δ'' satisfying the conditions 2c $(i - iii)$. Since $V'' < V$, (Δ'', θ'') is an abductive answer to the goal $G \wedge F \leq V$, that is the initial node of another derivation D^{over}. If D^{over} succeeds with a value $V'' < V$, then transition *Optimize* would not return the value V: it would impose the constraint $F(N_i) \leq V''$ to the node N_i, which would obviously fail (contradicting a previous hypothesis). If D^{over} fails, this failure would be unsound, contradicting the inductive hypothesis. If D^{over} succeeds with $V'' = V$, it contradicts the assumption that $V'' < V$.

8 Implementation

Constraint Handling Rules (CHR) [16] is a rule-based language useful to define new constraint solvers; here we cannot go into details for space reasons.

In the $\mathcal{S}$CIFF proof-procedure, abducible literals are mapped to CHR constraints; a general abducible $\mathtt{a}(X, Y)$ is represented as the constraint `abd(a(X,Y))`. Differently from other proof-procedures implemented in CHR [24,25,17,26], we do not map integrity constraints to CHR rules, but to other CHR constraints. For example, the IC

$$\mathbf{a}(X,Y), p(Y) \rightarrow r(X) \wedge q(Y) \vee q(X)$$

is mapped to the CHR constraint: `ic([abd(a(X, Y)), p(Y)], [[r(X), q(Y)], [q(X)]])`.

The operational semantics is defined by a set of transitions, some inherited from the IFF [7], some devoted to constraint processing, and others specific for $\mathcal{S}$CIFF. The transitions are then easily implemented into CHR rules; for example, transition *propagation* (with *case analysis*) [7] propagates an abducible with an implication[2]:

```
abd(P), ic([P1|Rest], Head) ==>
rename(ic([P1|Rest], Head), ic([RenP1|RenRest], RenHead)),
reif_unify(RenP1, P, B), (B = 1, ic(Rest, Head); B = 0)
```

`rename` computes a renaming that also considers the quantification of the variables, and `reif_unify` is our CHR implementation of *reified unification*: it is a ternary constraint relating two terms and a Boolean variable. Declaratively, if the two first arguments unify, then $B = 1$, otherwise, the two arguments do not unify and $B = 0$.

Another example is logical equivalence $[(true \rightarrow D_1 \vee \ldots D_n) \Leftrightarrow (D_1 \vee \cdots \vee D_n)]$:

```
ic([], Head) <=> member(Disjunct, Head), call(Disjunct)
```

that, given an IC with empty body, imposes that at least one of the disjuncts in the head holds. Notice that the chosen disjunct is executed as a Prolog goal: one of the features of the CHR implementation is that the abductive program written by the user is directly executed by the Prolog engine, and the resolvent of the proof-procedure coincides with the Prolog resolvent. Besides the efficiency gain of avoiding meta-interpretation, this means that every Prolog predicate can be invoked. In particular, we can invoke optimisation meta-predicates: in some cases, it is not enough to find *one* abductive solution, but the *best* solution with respect to some criteria is requested. In $\mathcal{S}$CIFF^{opt}, we use the same optimisation meta-predicates provided by the CLP solver, that efficiently implements a variant of the branch-and-bound algorithm.

9 Example

Consider a two player game, where each of the players A and B can play one move. The result of the two moves is a configuration with an associated value: one player's aim is to maximize the value, the other player's is to minimize it. Player A's move is represented by the abducible $\mathbf{a}(M_a, X)$, where M_a is the possible move and X is the obtained value. Analogously, player B abduces $\mathbf{b}(M_b, X)$. The obtained value is defined with a predicate $f(M_a, M_b, X)$ that gives the obtained value X corresponding to moves M_a and M_b. It can be defined (Figure 3) as a set of facts $f(0,0,5)$, $f(0,1,10)$, $f(1,0,4)$, $f(1,1,3)$. To compute the obtained value, we can define a predicate or an IC as the following:

[2] This is a sketch of the actual implementation, which is more optimized and, in particular, has a better exploitation of CHR indexing capabilities.

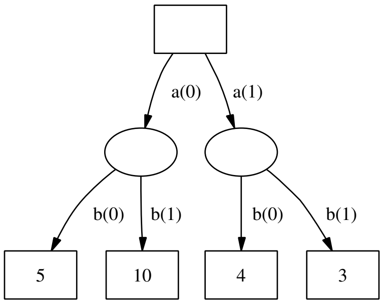


Fig. 3. min-max

$$\mathbf{a}(M_a, X_a), \mathbf{b}(M_b, X_b) \rightarrow X_a = X_b, f(M_a, M_b, X_a). \tag{6}$$

As player A moves first, and wants to maximize the value X, while player B moves next and his goal is to minimize X, the $\mathcal{S}$CIFF^{opt} goal will be

$$max_{\{\mathbf{a}(\_,\_),\mathbf{b}(\_,\_)\}}(V_b : \mathbf{a}(M_a, X_a) \wedge (M_a = 0 \vee M_a = 1) \wedge \\ min_{\{\mathbf{b}(\_,\_)\}}(X_b : \mathbf{b}(M_b, X_b) \wedge (M_b = 0 \vee M_b = 1)) = V_b)$$

We have four possible sets Δ satisfying the integrity constraint: Δ_0^0=$\{\mathbf{a}(0,5), \mathbf{b}(0,5)\}$, $\Delta_1^0 = \{\mathbf{a}(0,10), \mathbf{b}(1,10)\}$, $\Delta_0^1 = \{\mathbf{a}(1,4), \mathbf{b}(0,4)\}$, and $\Delta_1^1 = \{\mathbf{a}(1,3), \mathbf{b}(1,3)\}$. Declaratively, the internal goal $min_{\{\mathbf{b}(\_,\_)\}}$ is true in Δ_0^0 and Δ_1^1: Δ_0^1 is ruled out by Δ_0^0 (see condition 2c in Definition 7) and Δ_1^0 by Δ_1^1. The external $max_{\{\mathbf{a}(\_,\_),\mathbf{b}(\_,\_)\}}$ goal, thus, chooses from these two sets the Δ with maximum value of X_b, namely Δ_0^0; this output is the same as a min-max algorithm.

From an operational viewpoint, transition *Optimize* generates two nodes: one in which abduces $\mathbf{a}(0, X_a)$, and one with $\mathbf{a}(1, X_a)$.

$M_a = 0$ Transition *Optimize* is applied to min: it opens two nodes, one abducing $\mathbf{b}(0, X_b)$, the other with $\mathbf{b}(1, X_b)$.

$M_b = 0$ In the first, propagation of the integrity constraint (Eq. 6) imposes $X_a = X_b = 5$. Now transition *Optimize* of the internal min imposes the constraint $X_b \leq 5$ to all the open nodes in its scope, i.e., the node with value 10 in Figure 3

$M_b = 1$ In the second node, the propagation of the IC imposes $X_a = X_b = 10$, which conflicts with the constraint $X_b \leq 5$; CLP propagation results in a failure.

Now *Optimize* applied to min provides value 5 as optimum, and generates the node with Δ_0^0 as successor. The external *Optimize* (applied to max) adds the new constraint $X_a \geq 5$ to all open nodes, in particular to the open choice point.

$M_a = 1$ In this node, the external *Optimize* has imposed $V_b \geq 5$. Again, transition *Optimize* is applied to the min literal, and it opens two nodes. The minimum value computed for V_b is 3, and it does not satisfy $V_b \geq 5$, so the result is indeed $X_a = X_b = 5$.

This example shows how min-max problems can be easily encoded in $\mathcal{S}$CIFF^{opt}. In this simple example, we impose the optimization directly in the goal for ease of presentation, but it can be simply extended to other examples with recursion through minimization, to solve problems in PSPACE. In [15] we showed one such example.

10 Conclusions

Integration of abductive reasoning and constraint satisfaction has been vastly investigated in the recent years, and efficient proof-procedures have been developed [6,8,10]. Surprisingly, constraint optimization, one of the main topics in Constraint Programming, has been often left out of abductive proof-procedures, except for the interesting experiments reported (without proofs) in [11].

We extended the declarative and operational semantics of the $\mathcal{S}$CIFF proof-procedure to support this type of reasoning, resulting in the $\mathcal{S}$CIFF^{opt} framework. For $\mathcal{S}$CIFF^{opt} we proved a soundness result, that, to the best of our knowledge, is the first in the literature on abductive logic programming with constraint optimization. The soundness result holds when there is no floundering, a common issue in many logic programming languages. In future work, we plan to study the floundering issue in more detail, and extend the applicability of $\mathcal{S}$CIFF^{opt} to other problems, like those in which a conjunction of optimization atoms is required.

References

1. Kakas, A.C., Kowalski, R.A., Toni, F.: The role of abduction in logic programming. In: Gabbay, D.M., Hogger, C.J., Robinson, J.A. (eds.) Handbook of Logic in Artificial Intelligence and Logic Programming, vol. 5, pp. 235–324. Oxford University Press, Oxford (1998)
2. Eshghi, K., Kowalski, R.A.: Abduction compared with negation by failure. In: Levi, G., Martelli, M. (eds.) ICLP 1989, pp. 234–255 (1989)
3. Eshghi, K.: Abductive planning with the event calculus. In: ICLP 1988 (1988)
4. Kakas, A.C., Mancarella, P.: On the relation between Truth Maintenance and Abduction. In: Fukumura, T. (ed.) Proc. 1st Pacific Rim Int. Conf. on Artificial Intelligence, PRICAI (1990)
5. Denecker, M., De Schreye, D.: SLDNFA: an abductive procedure for abductive logic programs. Journal of Logic Programming 34, 111–167 (1998)
6. Kakas, A.C., Michael, A., Mourlas, C.: ACLP: Abductive Constraint Logic Programming. Journal of Logic Programming 44, 129–177 (2000)
7. Fung, T.H., Kowalski, R.A.: The IFF proof procedure for abductive logic programming. Journal of Logic Programming 33, 151–165 (1997)
8. Kakas, A.C., van Nuffelen, B., Denecker, M.: $\mathcal{A}$-System: Problem solving through abduction. In: Nebel, B. (ed.) Proc. of IJCAI 2001, pp. 591–596 (2001)
9. Alferes, J., Pereira, L., Swift, T.: Abduction in well-founded semantics and generalized stable models via tabled dual programs. Theory and Practice of Logic Programming 4 (2004)
10. Endriss, U., Mancarella, P., Sadri, F., Terreni, G., Toni, F.: The CIFF proof procedure for abductive logic programming with constraints. In: Alferes, J.J., Leite, J. (eds.) JELIA 2004. LNCS, vol. 3229, pp. 31–43. Springer, Heidelberg (2004)
11. van Nuffelen, B., Denecker, M.: Problem solving in ID-logic with aggregates. In: Proc. 8th Int. Workshop on Non-Monotonic Reasoning, NMR 2000, pp. 1–9 (2000)
12. Fages, F.: From constraint minimization to goal optimization in CLP languages. In: Freuder, E. (ed.) CP 1996. LNCS, vol. 1118. Springer, Heidelberg (1996)
13. Marriott, K., Stuckey, P.: Semantics of constraint logic programs with optimization. In: Aït-Kaci, H., Hanus, M., Moreno-Navarro, J. (eds.) ICLP Workshop: Integration of Declarative Paradigms, pp. 23–35 (1994)
14. Alberti, M., Chesani, F., Gavanelli, M., Lamma, E., Mello, P., Torroni, P.: Verifiable agent interaction in abductive logic programming: The SCIFF framework. ACM Transactions on Computational Logic 9 (2008)

15. Gavanelli, M., Alberti, M., Lamma, E.: Integrating abduction and constraint optimization in constraint handling rules. In: Ghallab, M., Spyropoulos, C.D., Fakotakis, N., Avouris, N. (eds.) ECAI 2008: 18th European Conference on Artificial Intelligence, pp. 903–904 (2008)
16. Frühwirth, T.: Theory and practice of constraint handling rules. Journal of Logic Programming 37, 95–138 (1998)
17. Christiansen, H., Dahl, V.: HYPROLOG: A new logic programming language with assumptions and abduction. In: Gabbrielli, M., Gupta, G. (eds.) ICLP 2005. LNCS, vol. 3668, pp. 159–173. Springer, Heidelberg (2005)
18. Badea, L., Tilivea, D.: Abductive partial order planning with dependent fluents. In: Baader, F., Brewka, G., Eiter, T. (eds.) KI 2001. LNCS, vol. 2174, pp. 63–77. Springer, Heidelberg (2001)
19. Jaffar, J., Maher, M.: Constraint logic programming: a survey. Journal of Logic Programming 19-20, 503–582 (1994)
20. Faber, W., Leone, N., Pfeifer, G.: Recursive aggregates in disjunctive logic programs: Semantics and complexity. In: Alferes, J.J., Leite, J. (eds.) JELIA 2004. LNCS, vol. 3229, pp. 200–212. Springer, Heidelberg (2004)
21. Fitting, M.: A Kripke-Kleene semantics for logic programs. Journal of Logic Programming 2, 295–312 (1985)
22. Kunen, K.: Negation in logic programming. Journal of Logic Programming 4, 289–308 (1987)
23. Apt, K.R., Bol, R.N.: Logic programming and negation: a survey. Journal of Logic Programming 19/20, 9–71 (1994)
24. Abdennadher, S., Christiansen, H.: An experimental CLP platform for integrity constraints and abduction. In: Larsen, H., Kacprzyk, J., Zadrozny, S., Andreasen, T., Christiansen, H. (eds.) FQAS, Flexible Query Answering Systems. LNCS, pp. 141–152. Springer, Heidelberg (2000)
25. Gavanelli, M., Lamma, E., Mello, P., Milano, M., Torroni, P.: Interpreting abduction in CLP. In: Buccafurri, F. (ed.) APPIA-GULP-PRODE Joint Conference on Declarative Programming, Reggio Calabria, Italy, Università Mediterranea di Reggio Calabria, pp. 25–35 (2003)
26. Alberti, M., Chesani, F., Daolio, D., Gavanelli, M., Lamma, E., Mello, P., Torroni, P.: Specification and verification of agent interaction protocols in a logic-based system. Scalable Computing: Practice and Experience 8, 1–13 (2007)

Encoding Table Constraints in CLP(FD) Based on Pair-Wise AC

Neng-Fa Zhou

CUNY Brooklyn College & Graduate Center
zhou@sci.brooklyn.cuny.edu

Abstract. We present an implementation of table constraints in CLP(FD). For binary constraints, the supports of each value are represented as a finite-domain variable, and action rules are used to propagate value exclusions. The bit-vector representation of finite domains facilitates constant-time removal of unsupported values. For n-ary constraints, we propose pair-wise arc consistency (AC), which ensures that each value has a support in the domain of every related variable. Pair-wise AC does not require introducing new problem variables as done in binarization methods and allows for compact representation of constraints. Nevertheless, pair-wise AC is weaker than general arc consistency (GAC) in terms of pruning power and requires a final check when a constraint becomes ground. To remedy this weakness, we propose adopting early checks when constraints are sufficiently instantiated. Our experimentation shows that pair-wise AC with early checking is as effective as GAC for positive constraints.

1 Introduction

A table constraint, or extensional constraint, over a tuple of variables specifies a set of tuples that are allowed (called *positive*) or disallowed (called *negative*) for the variables. Recently there has been a growing interest in this format of constraints. This format is well suited to problems where relations are more easily given in extension than in intension such as configuration problems involving datasets (e.g., crossword puzzles). Another reason for the popularity of this format is that certain intensional constraints, especially nonlinear and global constraints, can be more cheaply maintained when tabulated. The table format has been used in the CSP solver competitions and a good collection of problem instances are available. Arc consistency has been generalized for table constraints [5,12] (called *GAC*) and several data structures have been proposed for maintaining GAC for table constraints [2,3,6,7,9,10].

No previous work has been reported on introducing table constraints into CLP(FD). Because of the lack of sophisticated data structures such as multi-dimensional arrays and the necessity of manipulation of tagged data in CLP(FD), an efficient data structure designed for a low-level language may not be suited to CLP(FD). In this paper, we propose an encoding for table constraints in B-Prolog, a CLP(FD) system.

For a binary table constraint, the supports of each value are represented as a finite-domain variable. When either variable in the constraint is bound to a value, the other variable is unified with the finite-domain variable that represents the supports of the value. Whenever a value is excluded from the domain of a variable in the constraint, the supports of the value in the domain of the other variable are examined and those values that are no longer supported are excluded. As bit vectors are used to represent finite domains, the basic operations required in propagation can be performed efficiently [11].

For an n-ary table constraint, we propose pair-wise arc consistency (AC), which ensures that each value has a support in the domain of every related variable. As for binary constraints, supports of each value are also represented as a finite-domain variable. One of the advantages of pair-wise AC is that it, unlike binarization methods [1], does not introduce new problem variables. The newly introduced finite-domain variables are solely used as bit vectors to represent supports of values. Since supports are not updated during search, no events can occur in these new domain variables. This representation fits CLP(FD) since bits are not tagged individually. Another advantage of pair-wise AC is that constraints can be represented very compactly. Let n be the arity of an n-ary contraint and d be the size of the maximum domain. Supports of values can be represented with $O(n^2 \times d^2)$ space.

Nevertheless, pair-wise AC is weaker than GAC in terms of pruning power because, understandably, it is impossible to use $O(n^2 \times d^2)$ space to represent as many as d^n tuples. To remedy this weakness, we propose adopting early checks to enforce GAC when constraints are sufficiently instantiated. Early checking extends forward checking [8] because the number of variables contained in a constraint can be more than one when the constraint is checked.

For each variable X in a table constraint, a propagator is used to watch the `ins(`X`)` event which is posted when X is instantiated, and another propagator is used to respond to the `dom_any(`X`,`E`)` event which is posted whenever any element E is excluded from the domain of X. Propagators are described using action rules [16]. Our implementation propagates values like the AC-4 algorithm [13], and hence can be classified as fine-grained.

The contribution of this paper is twofold. First, this paper presents an encoding for table constraints which is suited to any CLP(FD) system that represents finite-domains as bit vectors and handles domain value exclusions as events. Second, this paper proposes pair-wise AC, which is a natural extension of AC but has never received much attention before, and proposes to remedy the weakness of pair-wise AC with early checking. We experimented with two different settings for early checking and our experimental results showed that pair-wise AC with early checking is as effective as GAC.

This paper is organized as follows: Section 2 overviews table constraints, consistency algorithms, CLP(FD), and action rules; Section 3 describes an encoding for binary constraints and gives the propagators as action rules; Section 4 gives the propagators for maintaining pair-wise AC; Section 5 proposes several improvements on the propagators; Section 6 describes early checking; Section 7 presents the experimental results; Section 8 discusses the related and future work.

2 Preliminaries

2.1 Table Constraints and Consistency

A table constraint is either positive or negative. A positive constraint takes the form X `in` R and a negative constraint takes the form X `notin` R where X is a tuple of variables $(X_1, \ldots, X_n)$ and R is a table defined as a list of tuples of integers where each tuple takes the form $(a_1, \ldots, a_n)$. In order to allow multiple constraints to share a table, we allow X to be a list of tuples of variables. In theory, a negative constraint can always be represented as a positive constraint by complementing the table, but in practice this is not always viable since the resulting table can be prohibitively large.

A table constraint is said to be *binary* if each tuple has only two components, and *n-ary* if each tuple has more than two components. A table constraint degenerates into a domain constraint in CLP(FD) if each tuple has only one component.

Let (X_1, X_2) `in` R be a binary table constraint. A value x_1 in the domain of X_1 is said to be *supported* in the constraint if there exists a value x_2 in the domain of X_2 such that (x_1, x_2) is included in R. The constraint is said to be *AC (arc consistent)* on X_1 if every value in the domain of X_1 is supported. The constraint is said to be *AC* if it is AC on both X_1 and X_2.

Let $(X_1, \ldots, X_n)$ `in` R be an n-ary table constraint. Let R_{ij} denote the projection of the table R over the ith and jth columns $(i < j)$. The binary projection of the constraint over X_i and X_j $(i < j)$ is the binary constraint (X_i, X_j) `in` R_{ij}. The n-ary constraint is said to be *pair-wise AC* if all of its binary projections are AC.

Consider the n-ary constraint $(X_1, \ldots, X_n)$ `in` R again. A value x_i in the domain of variable X_i is *gac-supported* in the constraint if there exists a tuple in R whose ith component is equal to x_i. The constraint is said to be *GAC* if every value in the domain of every variable is gac-supported. This condition can be given more formally as:

$$\forall_{i \in \{1..n\}} \forall_{x_i \in X_i} \exists_{x_1 \in X_1, \ldots, x_{i-1} \in X_{i-1}, x_{i+1} \in X_{i+1}, \ldots, x_n \in X_n} (x_1, x_2, \ldots, x_n) \in R$$

where variables are used to denote their domains.

In general, pair-wise AC is a weaker condition than GAC. For example, consider the following constraint:

```
(X,Y,Z) in [(0,1,1),
            (1,0,1),
            (1,1,0)]
```

After the assignment `X=1`, `Y=1`, and `Z=1`, the constraint is not GAC but it is still pair-wise AC.

When a table constraint is generated, tuples of variables and values in the form $(X_1, \ldots, X_n)$ are all transformed into the form $\mathtt{t}(X_1, \ldots, X_n)$ that takes less memory to store and is easier to manipulate.

2.2 CLP(FD)

CLP(FD) [8] is a constraint language that enhances Prolog with built-ins for specifying domain variables, constraints, and strategies for assigning values to variables (called *labeling*). The unification operator is enhanced to deal with domain variables. For two domain variables X and Y, after unification $X = Y$ the elements that are not in both domains are removed and the two variables become aliases.

The following built-ins are used in the implementation of table constraints:

– X `in` D: restricts X to take on a value from D, where D is a set of integers.
– X `notin` D: forbids X to take on any value from D.
– `fd_dom(`X`,`D`)`: D is the list of integers in the domain of X.
– `fd_disjoint(`X`,`Y`)`: The domains of X and Y are disjoint.
– `fd_set_false(`X`,`E`)`: excludes integer `E` from the domain of X. It is equivalent to X`#\=`E but more efficient.

These built-ins are available in B-Prolog. Similar built-ins are also available in other CLP(FD) systems or can be implemented using other primitives.

2.3 Action Rules and Events

The AR (*Action Rules*) language is designed to facilitate the specification of event-driven functionality needed by applications such as constraint propagators and graphical user interfaces where interactions of multiple entities are essential [16]. It was originally implemented in B-Prolog and now has been introduced into other Prolog systems [4].

An action rule takes the following form:

$$Agent, Condition, \{Event\} => Action$$

where *Agent* is an atomic formula that represents a pattern for agents, *Condition* is a conjunction of conditions on the agents, *Event* is a non-empty disjunction of patterns for events that can activate the agents, and *Action* is a sequence of arbitrary subgoals. An action rule degenerates into a *commitment rule* if *Event* together with the enclosing braces are missing. In general, a predicate can be defined with multiple action rules. For the sake of simplicity, we assume in this paper that each predicate is defined with only one action rule possibly followed by a sequence of commitment rules.

A subgoal is called an *agent* if it can be suspended and activated by events. For an agent α, a rule "H, C, $\{E\}$ => B" is *applicable* to the agent if there exists a matching substitution θ such that $H\theta = \alpha$ and the condition $C\theta$ is satisfied. The reader is referred to [16] for a detailed description of the language and its operational semantics.

The following event patterns are supported for programming constraint propagators:

- `generated`: After an agent is generated but before it is suspended for the first time. The sole purpose of this pattern is to make it possible to specify preprocessing and constraint propagation actions in one rule.
- `ins`(X): when the variable X is instantiated.
- `bound`(X): when a bound of the domain of X is updated. There is no distinction between lower and upper bounds changes.
- `dom`(X,E): when an *inner* value E is excluded from the domain of X. Since E is used to reference the excluded value, it must be the first occurrence of the variable in the rule.
- `dom`(X): same as `dom`(X,E) but the excluded value is ignored.
- `dom_any`(X,E): when an arbitrary value E is excluded from the domain of X. Unlike in `dom`(X,E), the excluded value E here can be a bound of the domain of X.
- `dom_any`(X): equivalent to the disjunction of `dom`(X) and `bound`(X).

Note that when a variable is instantiated, no `bound` or `dom` event is posted. Consider the following example:

```
p(X),{dom(X,E)} => write(dom(E)).
q(X),{dom_any(X,E)} => write(dom_any(E)).
r(X),{bound(X)} => write(bound).
go:-X :: 1..4, p(X), q(X), r(X), X #\= 2, X #\= 4, X #\= 1.
```

The query `go` gives the following outputs: `dom(2)`, `dom_any(2)`, `dom_any(4)` and `bound`.[1] The outputs `dom(2)` and `dom_any(2)` are caused by `X #\= 2`, and the outputs `dom_any(4)` and `bound` are caused by `X #\= 4`. After the constraint `X #\= 1` is posted, `X` is instantiated to `3`, which posts an `ins(X)` event but not a `bound` or `dom` event.

[1] In the current implementation of AR, when more than one agent is activated the one that was generated first is executed first. This explains why `dom(2)` occurs before `dom_any(2)` and also why `dom_any(4)` occurs before `bound`.

3 Binary Constraints

Given a binary table constraint `(X,Y) in R`, we build a hashtable `Hxy` for supports of values of `X`.[2] For each value `Ex` in the domain of `X`, there exists an entry `(Ex,Sx)` in `Hxy` where `Sx` is a finite-domain variable that represents `Ex`'s set of supports in the domain of `Y`. If `Ex` has only one support, then `Sx` is the support itself. Similarly, we also build a hashtable `Hyx` for supports of values of `Y`. After `(X,Y) in R` is posted, it is made AC by excluding all unsupported values from the domains of `X` and `Y`. In the following, `Hxy` and `Hyx` are called *support tables.*

Consider, for example, the following constraint:

```
(X,Y) in [(1,2),(2,1),(3,4),(3,5),(4,4)]
```

The hashtable `Hxy` contains

```
(1,2),(2,1),(3,S3:[4,5]),(4,4)
```

and the hashtable `Hyx` contains

```
(1,2),(2,1),(4,S4:[3,4]),(5,3)
```

where `S3:[4,5]` and `S4:[3,4]` are two finite-domain variables. After the constraint is posted, `X`'s domain becomes `[1,2,3,4]` and `Y`'s domain becomes `[1,2,4,5]`.

The two hashtables `Hxy` and `Hyx` are essentially two tries [6]. Each trie requires, in the worst case, O(`|X|`×`|Y|`) space. This representation is compact because of the indexing effect and the use of bit vectors for domain variables. As will be shown below, supports are never updated during search. Therefore, the domain variables used to represent supports never post any event.

For a binary constraint over `(X,Y)`, we generate propagators to watch `ins` and `dom_any` events on `X` and `Y`. The propagation is very straightforward. When `X` is bound to an integer, `Y`'s domain is reduced to retain only those elements that are supported by `X`. Whenever a value is excluded from the domain of `X`, the supports of the value in the domain of `Y` are examined and those values that are no longer supported by `X` are excluded from the domain of `Y`.

The propagator `watch_ins(X,Y,Hxy)`, defined below, watches `ins(X)` events.

```
watch_ins(X,Y,Hxy),var(X),{ins(X)} => true.
watch_ins(X,Y,Hxy) =>
    hashtable_get(Hxy,X,Sx),
    Y=Sx
```

The propagator is suspended as long as `X` is a variable. The second rule is applied after `X` becomes ground, which unifies `Y` with the set of supports of `X`.

[2] Hashtables are not available in ISO-Prolog. The built-in `hashtable_get(H,K,Val)` in B-Prolog retrieves from the table `H` the value `Val` with the key `K`. In the real implementation, a hashtable talored to tuples is used when the table is sparse or contain negative integers, and a structure is used otherwise.

```
watch_dom(X,Y,Hxy,Hyx),var(X),{dom_any(X,Ex)} =>
    hashtable_get(Hxy,Ex,Sx), % Sx supports Ex
    fd_dom(Sx,Eys),
    find_support(X,Eys,Y,Hyx).
watch_dom(X,Y,Hxy,Hyx) => true.

find_support(X,[],Y,Hyx).
find_support(X,[Ey|Eys],Y,Hyx):-
    hashtable_get(Hyx,Ey,Sy), % Sy supports Ey
    (fd_disjoint(X,Sy)->fd_set_false(Y,Ey);true),
    find_support(X,Eys,Y,Hyx).
```

Fig. 1. The propagator that watches `dom_any` events

The propagator `watch_dom(X,Y,Hxy,Hyx)`, defined in Figure 1, watches `dom_any` events on `X`.

Whenever a value `Ex` is excluded from the domain of `X`, `Ex`'s set of supports `Sx` is retrieved from `Hxy`. The predicate `find_support` examines every element in `Sx`, and excludes it from the domain of `Y` if it is no longer supported by `X`. In the real implementation, `find_support` is encoded in C which uses bit-wise operations to iterate through the elements of `Sx`.

4 Pair-Wise AC for n-Ary Constraints

Given an n-ary table constraint `Vars in R`, we build two hashtables `Hxy` and `Hyx` for each pair of variables `X` and `Y` in `Vars`, and generate propagators to watch `ins` and `dom_any` events. In this way, pair-wise AC is maintained.

Since pair-wise AC does not guarantee GAC, an n-ary constraint needs to be checked after it becomes ground. Let `HashR` be the hashtable representation of the table `R`. This *final check* is described as follows:

```
final_check(HashR,Vars),n_vars_gt(1,0),{ins(Vars)} => true.
final_check(HashR,Vars) => hashtable_get(HashR,Vars,_).
```

The condition `n_vars_gt(1,0)` means that the last argument (namely `Vars`) has more than 0 variables.[3] The subgoal is suspended while at least one of the variables in `Vars` is free. After all the variables are instantiated, `hashtable_get` checks if the tuple `Vars` is included in `HashR`.

[3] In general the built-in `n_vars_gt(m,n)` in B-Prolog means that the number of variables in the last `m` arguments of the head is greater than `n`, where both `m` and `n` are integer constants. Notice that the arguments are not passed to the built-in. The system always fetches those arguments from the current frame. This built-in is well used in constraint propagators to change the action when the number of variables in the constraint reaches a certain threshold.

```
register_pair(X,Term):-        % Term=pair(Y,Hxy,Hyx)
     get_attr(X,pairs,L),!,
     attach(Term,L).           % attach Term to the end of L
register_pair(X,Term):-
     L=[Term|_],               % create an open-ended list
     put_attr_no_hook(X,pairs,L), % no default hook on X
     watch_ins(X,L),
     watch_dom(X,L).

watch_ins(X,L),var(X),{ins(X)} => true.
watch_ins(X,L) =>
     ...   % for each term bin(Y,Hxy,Hyx) in L, enforce AC on Y
     ...   % after X is instantiated.

watch_ins(X,L),var(X),{dom_any(X,Ex)} =>
     ...   % for each term bin(Y,Hxy,Hyx) in L, enforece AC on Y
           % after Ex is excluded from the domain of X.
watch_ins(X,L) => true.
```

Fig. 2. The registration procedure

5 Improvements

In the encoding described above, two propagators are used for each variable in a pair of variables, one watching `ins` and the other watching `dom_any` events on the variable. When a variable is involved in n pairs, $2 \times n$ propagators are generated. One implementation technique for speeding-up propagation in CLP(FD) is to combine the propagators that watch the same event and take similar actions.[4] This technique can be used to speed-up propagation for table constraints too.

For a pair of variables `(X,Y)`, let `Hxy` and `Hyx` be the two support tables. The term `pair(Y,Hxy,Hyx)` is created and registered onto `X` under the attribute name `pairs`. If the attribute `pairs` does not exist yet, the attribute is created and two propagators are generated; if the attribute already exists, then the term is attached to the end of the attribute value, which is an incomplete list with an open end. Figure 2 gives part of the registration procedure. Similarly, the term `bin(X,Hyx,Hxy)` needs to be registered onto `Y`.

The registration procedure is further improved as follows. For a pair `(X,Y)`, if the support tables `Hxy` and `Hyx` represent the Cartesian product of the domains of the variables, it is unnecessary to do the registration at all because every value is guaranteed a support no matter how the variables are instantiated.

Moreover, if a pair has been registered already, we merge the old support tables with the new ones. Let `pair(Y,Hxy,Hyx)` be the term to be registered onto `X`, and `pair(Y,OldHxy,OldHyx)` be a term that has been already registered

[4] This technique is implemented in B-Prolog for constraints such as disequality constraints over two variables.

on X. We construct two new support tables NewHxy and NewHyx where NewHxy is the intersection of OldHxy and Hxy, and NewHyx is the intersection of OldHyx and Hyx. Then we use the built-in setarg/3 to replace OldHxy with NewHxy and OldHyx with NewHyx. This improvement allows inter-constraints sharing.

6 Early Checking

As shown above, pair-wise AC is weaker than GAC in terms of pruning power. The propagators for maintaining pair-wise AC resort to a final check to ensure that a constraint is indeed supported when it becomes ground. To remedy the weak pruning power of pair-wise AC, we can advance this final check to a point when the constraint still contain variables. An early check ensures that every value in the domain of every variable has a supporting tuple. We consider early checking for positive constraints, and similar ideas can be applied to negative constraints too.

There are two possible approaches to checking a constraint: One is to iterate through the values of the domains of the remaining variables in the constraint and, for each combination, we check if it is included in the table; the other is to iterate through the tuples in the table. Since the number of tuples in a given table is normally significantly smaller than the possible combinations of domain values when the number of variables is large, we follow the later approach.

To make it fast to iterate through the tuples in a table, we convert the table to a trie such that common prefixes of the tuples need not be examined more than once for each traversal. We only use one trie per table. The tuples are indexed on the first argument first, then second, and so on. The following defines a propagator that maintains GAC when variables are instantiated.

```
early_check(Trie,Vars),
    {generated,ins(Vars)}
=>
    enforce_gac(Trie,Vars).
```

The predicate enforce_gac(Trie,Vars) is also called when the propagator is first created. It first walks through the trie to record all the values that are supported, and then it examines each value in the domain of each variable in Vars and excludes it from the domain if it has no supporting tuple.

Since enforce_gac does not respond to domain value exclusions, pair-wise AC is still weaker than GAC even with this early checking. To always enforce GAC, we could call ensure_gac whenever a change occurs to the domain of any variable.

```
early_check_gac(Trie,Vars),
    {generated,ins(Vars),bound(Vars),dom(Vars)}
=>
    enforce_gac(Trie,Vars).
```

In addition to the `generated` and `ins(Vars)` events, this rule also watches `bound(Vars)` and `dom(Vars)` events. Recall that `bound(Vars)` is posted whenever a bound of the domain of any variable in `Vars` is changed and `dom(Vars)` is posted whenever an inner value is excluded from the domain of any variable. The two events `bound(Vars)` and `dom(Vars)` can be equivalently encoded as `dom_any(Vars)`.

Since `ensure_gac` is expensive, calling it on every change may not pay off. One compromise is to enforce GAC on domain value exclusions only when the constraint contains a certain number of variables. The following gives the refined propagator for early checking.

```
early_check_compromise(Trie,Vars),
    n_vars_gt(1,2),
    {generated,ins(Vars)}
=>
    enforce_gac(Trie,Vars).
early_check_compromise(Trie,Vars) =>
    early_check_bin(Trie,Vars).

early_check_bin(Trie,Vars),
    {generated,ins(Vars),bound(Vars),dom(Vars)}
=>
    enforce_gac(Trie,Vars).
```

Once the number of variables contained in the constraint is 2 or less (the condition `n_vars_gt(1,2)` fails), the propagator is replaced with `early_check_bin` which watches all changes to the domains of the variables.

Recall that the propagators for maintaining pair-wise AC already watch `dom_any` events. One may ask why we need to create a propagator to watch `dom` events here when the constraint becomes binary. The answer is that the support tables used in maintaining pair-wise AC are binary projections and they are never reduced while variables are instantiated. Consider the following example:

```
(X,Y,Z) in [(0,1,1),
            (0,2,2),
            (0,3,3),
            (1,1,2),
            (1,2,3),
            (1,3,1)]
```

The support table `Hyz` from `Y` to `Z` contains the following entries: `(1,S1:[1,2])`, (2,S2:[2,3]), and `(3,S3:[1,3])`. When `X` is bound to `0`, the support table should be reduced to contain `(1,1)`, `(2,2)`, and `(3,3)`. After that, when a value, say `2`, is excluded from the domain of `Y`, the support `2` should be excluded from the domain of `Z`. Nevertheless, because our solver does not reduce support tables, this effect couldn't be achieved without calling `enforce_gac(Trie,Vars)`.

Table 1. Comparison on CPU time (seconds)

Problem instance	MAX-ARITY	PAC	PAC+ET1	PAC+ET2	GAC
bdd_21_133_18_78_10	18	1.50	2.70	2.82	2.67
bdd_21_133_18_78_11	18	9.25	31.00	31.00	31.00
bdd_21_133_18_78_12	18	7.85	24.00	24.00	23.00
bdd_21_133_18_78_13	18	9.32	69.00	69.00	69.00
bdd_21_133_18_78_14	18	11.00	33.00	33.00	33.00
crossword_m1c_lex_vg10_11	11	>500	2.96	2.96	12.00
crossword_m1c_lex_vg10_12	12	>500	0.61	0.59	2.61
crossword_m1c_lex_vg11_12	12	>500	0.32	0.32	0.89
crossword_m1c_lex_vg11_13	13	>500	0.12	0.12	0.25
crossword_m1c_lex_vg11_15	15	0.31	0.47	0.47	0.12
jnh01	14	0.15	0.15	0.15	0.15
jnh02	10	0.16	0.16	0.16	0.15
jnh04	11	0.20	0.20	0.20	0.20
jnh05	11	0.15	0.15	0.15	0.15
jnh06	11	0.11	0.11	0.11	0.94
rand_10_20_10_5_10000_0	10	>500	1.42	1.42	1.43
rand_10_20_10_5_10000_10	10	>500	1.29	1.28	1.29
rand_10_20_10_5_10000_11	10	>500	1.87	1.82	2.46
rand_10_20_10_5_10000_12	10	>500	1.31	1.31	1.87
rand_10_20_10_5_10000_13	10	>500	1.37	1.37	1.92
renault_mgd	10	2.50	2.57	2.54	2.57
renault_mod_0	10	>500	>500	>500	>500
renault_mod_10	10	>500	>500	>500	>500
renault_mod_11	10	>500	>500	>500	>500
renault_mod_12	10	>500	>500	>500	>500
ssa_0432_003	5	1.21	0.14	0.14	0.15
ssa_2670_130	5	>500	>500	>500	>500
ssa_2670_141	4	0.000	0.000	0.000	0.000
ssa_6288_047	6	0.40	0.40	0.40	0.40
ssa_7552_038	6	0.47	0.47	0.31	0.31
tsp_20_142	3	>500	>500	81.00	81.00
tsp_20_190	3	>500	286.00	11.00	11.00
tsp_20_193	3	>500	>500	36.00	36.00
tsp_20_1	3	>500	57.00	0.95	0.95
tsp_20_29	3	>500	2.31	0.29	0.29

7 Experimental Results

The two built-ins, `in/2` and `notin/2`, in B-Prolog have been extended to allow positive and negative table constraints.[5] For positive constraints, pair-wise AC is used with early checking, which maintains GAC when constraints become ternary. For negative constraints, pair-wise AC is used with forward checking, which maintains GAC only when constraints become unary. For negative constraints, support tables are constructed without complementing given relations.

Thanks to the availability of action rules, the extension was implemented with relative ease. The extension contains about 300 lines of code in Prolog (and action rules) and 1000 lines of code in C, most of which are for preprocessing tables.

We compared pair-wise AC with and without early checking on a selected set of benchmarks used for the N-ARY-EXT category in the CSP solver competitions.[6] The problem instances were translated from XML into Prolog format.

[5] Table constraints are supported in version 7.3 and up.

[6] http://www.cril.univ-artois.fr/~lecoutre/research/benchmarks/benchmarks.html

Table 2. Comparison on backtracks

Problem instance	PAC	PAC+ET1	PAC+ET2	GAC
bdd_21_133_18_78_10	0	0	0	0
bdd_21_133_18_78_11	532563	9734	9734	9734
bdd_21_133_18_78_12	321883	13438	13438	13438
bdd_21_133_18_78_13	535481	11154	11154	11154
bdd_21_133_18_78_14	552313	10235	10235	10235
crossword_m1c_lex_vg10_11	91408432	675	675	193
crossword_m1c_lex_vg10_12	11401605	140	140	57
crossword_m1c_lex_vg11_12	10616917	61	61	22
crossword_m1c_lex_vg11_13	2100191	19	19	12
crossword_m1c_lex_vg11_15	0	0	0	0
jnh01	50	50	50	50
jnh02	8	8	8	8
jnh04	1611	1611	1611	1611
jnh05	41	41	41	41
jnh06	826	826	826	826
rand_10_20_10_5_10000_0	>268435455	1010	1010	999
rand_10_20_10_5_10000_10	>268435455	1000	1000	999
rand_10_20_10_5_10000_11	263869300	1003	1003	999
rand_10_20_10_5_10000_12	>268435455	1002	1002	997
rand_10_20_10_5_10000_13	>268435455	1882	1882	998
renault_mgd	13	0	0	0
renault_mod_0	>268435455	17614672	13761572	20704205
renault_mod_10	80724776	9859177	4931019	2200586
renault_mod_11	>268435455	8099239	3215349	1818967
renault_mod_12	251771657	9082875	6301406	2128999
ssa_0432_003	318288	10126	10126	10126
ssa_2670_130	72698460	23508987	23994708	24034014
ssa_2670_141	7	0	0	0
ssa_6288_047	23	23	23	23
ssa_7552_038	476	38	38	38
tsp_20_142	17326326	207966	15036	15036
tsp_20_190	25459227	366646	5750	5750
tsp_20_193	17622012	207826	2814	2814
tsp_20_1	21302231	65937	229	229
tsp_20_29	29304024	2723	31	31

The first 5 instances in each of seven selected problem classes were chosen. Each of the problem instances contains at least one positive constraint. Each instance was given a time limit of 500 seconds and no memory limit was imposed. The labeling strategy `ffc` (first-fail, breaking tie by selecting a most constrained variable) was used in all the runs. The machine used was a Pentium 3.0GHz with 1GB of RAM running Windows XP.

Table 1 shows the CPU times. In each row, the first column gives the name of a problem instance, the second column gives the maximum arity of the constraints in the instance, and each of the remaining columns gives the CPU time taken by each of the four different settings: PAC maintains pair-wise AC without early checking; PAC+ET1 triggers early checking after constraints become binary; PAC+ET2 triggers early checking after constraints become ternary; and GAC maintains GAC all the time as is done in `early_check_gac` shown above. Both PAC+ET1 and PAC+ET2 trigger early checking on `ins` events. PAC solved only 16 instances, PAC+ET1 solved 28 instances, and PAC+ET2 and GAC each solved 30 instances. In general, PAC alone is too weak, but it turned out to be the fastest on the bdd benchmarks. There is no remarkable difference between

PAC+ET2 and GAC for most of the instances, and PAC+ET2 is faster than GAC on some for the instances such as `crossword_m1c_lex_vg10_11`. Four of the five instances of `renault` were not solved. Profiling the runs indicated that these instances were very memory demanding and most of the execution time was spent on garbage collection.

Table 2 shows the number of backtracks in each run. For those runs that were terminated by time-out events, the numbers were also recorded. For instances that contain only Boolean variables (`bdd` and `jnh`), there is no difference among different settings for early checking since no `dom` or `dom_any` event can occur on Boolean variables. For instances that contain no constraint with more than 3 variables (`tsp`), there is no difference between PAC+ET2 and GAC.

We didn't directly compare our solver with other solvers for table constraints. The top ranked solvers, such as mddc, MDG, Mistral, and Abscon solved all the selected instances under a time limit of 1800 seconds.[7] We have to mention that the Windows-XP PC we used is probably slower than the Linux server used in the competition and our solver does not employ any restart strategy. Even under the same condition, it would be unfair to compare a CLP(FD) solver with a solver implemented directly in C or C++ because operations such dereferencing, tagging, and untagging incur measurable overhead in CLP(FD).

8 Related and Further Work

The key operation used in GAC algorithms is to find a support tuple for a value y in the domain of a variable Y after a value x has been excluded from the domain of a related variable X ($X \neq Y$) [2]. Significant efforts have been made to speed-up this operation by skipping irrelevant tuples that can never been supports for a value [3,6,10]. Indexing is an effective technique. The trie data structure [6] indexes tuples such that tuples that have the same prefix share nodes in the trie. In order to facilitate propagating changes originated at every variable in an n-ary constraint, the solver reported in [6] needs to build n tries, one for each variable. An MDD (multi-valued decision diagram) [3] is more effective than a trie in the sense that tuples that have the same suffix also share nodes. The solver `mddc-solv` based on MDD was ranked top in the N-ARY-EXT category in the third CSP solver competition.

Our encoding of binary constraints is similar to the trie encoding. The difference is that the children (leaves) of each interior node are represented as a finite-domain variable rather than a list or an array. This representation fits CLP(FD) since bit-vectors are used in the representation of finite domains and bits in bit vectors are not tagged individually. Any data structure that requires tagging and untagging would incur considerable overhead.

It is well known that any n-ary constraint can be binarized by using a dual representation (i.e., treating each constraint as a variable) or introducing hidden

[7] http://www.cril.univ-artois.fr/CPAI08/results/results.php?idev=15

variables for constraints [1]. Experiments have been done to compare various binarization schemes [14]. Our previous solver [17], like the early version of the Mistral solver [7]), introduces a new finite-domain variable for each n-ary constraint and encodes each tuple in the table as an integer. The main problem with that solver was that newly introduced variables could have very bigger domains and the solver could be flooded with events from these domains.

No previous work has been reported on introducing table constraints into CLP(FD). The case constraint in SICStus Prolog is used to implement the built-in `table/2`. Similar built-ins such as `fd_relation/2` in GNU-Prolog and `tuples_in/2` in SWI-Prolog have been implemented, but no detail of the implementation is published. None of these CLP(FD) systems directly supports negative table constraints.

In our solver, supports of values are not updated during search. This makes it possible for constraints to share tables and also renders it unnecessary to trail or copy supports of values. The drawback is that the support tables created for maintaining pair-wise AC for an n-ary constraint cannot be used to enforce AC when the constraint becomes binary. The early-checking propagators in our solver need to use the trie from the original table to enforce AC. Also the operation `fd_disjoint` does not become as cheap as it is supposed to be because the domain that represents supports of a value never shrinks. Recently, a new approach has been proposed that solves n-ary CSPs by reducing tables [10,15]. It is worthwhile to investigate if this approach can be integrated into our approach.

Further work needs to be done to investigate when and how early checking should be performed. Our solver does not do any early checking on negative constraints. Further investigation should cover negative constraints as well.

9 Conclusion

We have presented an encoding for table constraints in CLP(FD) based on pair-wise AC. In the encoding, the supports of each value are represented as a finite-domain variable, and action rules are used to propagate value exclusions. The encoding is compact and requires no new problem variables. To remedy the weak pruning power of pair-wise AC, we proposed integrating pair-wise AC with early checking. Our experimental results showed that such an integration is effective. Our approach differs from the major GAC algorithms in that it is based on pair-wise AC and is fine grained. More work remains to be done concerning when and how early checking should be performed.

References

1. Bacchus, F., Chen, X., van Beek, P., Walsh, T.: Binary vs. non-binary constraints. Artif. Intell. 140(1/2), 1–37 (2002)
2. Bessière, C., Régin, J.-C.: Arc consistency for general constraint networks: Preliminary results. In: IJCAI (1), pp. 398–404 (1997)

3. Cheng, K.C.K., Yap, R.H.C.: Maintaining generalized arc consistency on ad hoc r-ary constraints. In: Stuckey, P.J. (ed.) CP 2008. LNCS, vol. 5202, pp. 509–523. Springer, Heidelberg (2008)
4. Demoen, B., Nguyen, P.-L.: Two WAM implementations of action rules. In: Garcia de la Banda, M., Pontelli, E. (eds.) ICLP 2008. LNCS, vol. 5366, pp. 621–635. Springer, Heidelberg (2008)
5. Freuder, E.C.: Synthesizing constraint expressions. Commun. ACM 21(11), 958–966 (1978)
6. Gent, I.P., Jefferson, C., Miguel, I., Nightingale, P.: Data structures for generalised arc consistency for extensional constraints. In: AAAI, pp. 191–197 (2007)
7. Hebrard, E.: Mistral, a constraint satisfaction library. In: van Dongen, M.R.C., Lecoutre, C., Roussel, O. (eds.) Proceedings of the Second International CSP Solver Competition, pp. 35–42 (2008)
8. Van Hentenryck, P.: Constraint Satisfaction in Logic Programming. MIT Press, Cambridge (1989)
9. Katsirelos, G., Walsh, T.: A compression algorithm for large arity extensional constraints. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 379–393. Springer, Heidelberg (2007)
10. Lecoutre, C.: Optimization of simple tabular reduction for table constraints. In: Stuckey, P.J. (ed.) CP 2008. LNCS, vol. 5202, pp. 128–143. Springer, Heidelberg (2008)
11. Lecoutre, C., Julien, V.: Enforcing arc consistency using bitwise operations. Constraint Programming Letters 2, 21–35 (2008)
12. Mackworth, A.K.: On reading sketch maps. In: IJCAI, pp. 598–606 (1977)
13. Mohr, R., Henderson, T.C.: Arc and path consistency revisited. Artificial Intelligence 28, 225–233 (1986)
14. Stergiou, K., Samaras, N.: Binary encodings of non-binary constraint satisfaction problems: Algorithms and experimental results. J. Artif. Intell. Res (JAIR) 24, 641–684 (2005)
15. Ullmann, J.R.: Partition search for non-binary constraint satisfaction. Inf. Sci. 177(18), 3639–3678 (2007)
16. Zhou, N.-F.: Programming finite-domain constraint propagators in action rules. Theory and Practice of Logic Programming (TPLP) 6(5), 483–508 (2006)
17. Zhou, N.-F.: BPSOLVER 2008. In: van Dongen, M.R.C., Lecoutre, C., Roussel, O. (eds.) Proceedings of the Third International CSP Solver Competition, pp. 83–90 (2008)

Types for Secure Pattern Matching with Local Knowledge in Universal Concurrent Constraint Programming

Thomas Hildebrandt and Hugo A. López

IT University of Copenhagen
Programming, Logic and Semantics Group
Rued Langgaards Vej 7, DK-2300 Copenhagen S, Denmark
{hilde,lopez}@itu.dk

Abstract. The fundamental primitives of Concurrent Constraint Programming (CCP), *tell* and *ask*, respectively adds knowledge to and infers knowledge from a shared constraint store. These features, and the elegant use of the constraint system to represent the abilities of attackers, make concurrent constraint programming and timed CCP (`tcc`) interesting candidates for modeling and reasoning about security protocols. However, they lack primitives for the communication of secrets (or local names as in the π -calculus) between agents. The recently proposed *universal* `tcc` (`utcc`) introduces a universally quantified ask operation that makes it possible to infer knowledge which is local to other agents. However, it allows agents to guess knowledge even if it is encrypted or communicated on secret channels, simply by quantifying over both the encryption key (or channel) and the message simultaneously. We present a secure `utcc` ($\mathtt{utcc}_s$) based on: (i) a simple type system for constraints allowing to distinguish between restricted (secure) and non-restricted (universally quantifiable) variables in constraints, and (ii) a generalization of the universally quantified ask operation to allow the assumption of local knowledge. We illustrate the use of the $\mathtt{utcc}_s$ calculus with examples on communication of local names (as in the π -calculus) and for giving semantics to secure pattern matching in a prototypical security language.

Keywords: Concurrent Constraint Programming, Process Calculi, Type systems, Mobility, Security.

1 Introduction

A number of variants of process calculi and logical approaches have been proposed for the analysis of security protocols, including [2,6,5,8,3,11,4,14]. The approaches have generally two features in common: The first is the use of some kind of logical inference/pattern matching/unification to represent the ability of attackers and principals to infer what has been communicated, and from that knowledge construct new messages. The second is a way of representing and communicating local knowledge (such as keys or nonces in security protocols).

The combination of these two features calls for some means to control the ability to infer knowledge which is supposed to be inaccessible, e.g. a message encrypted by a key unknown to the attacker or the key itself. Typically, this takes the form of a restriction on the rules for inference of knowledge/pattern matching, designed particularly for the considered setting of security protocols. Sometimes the restriction is enforced by the language, as e.g. in [4], however in many cases the restriction must be maintained in the specification of the attacker and the protocol under analysis.

In the present paper we propose a more general solution to representing this kind of restriction. Even though we believe that the solution is broadly applicable, in this paper we focus on the setting of concurrent constraint programming (CCP). This is due to the fact that our work was directly triggered by the interesting recent proposal of the calculus of *universal* timed concurrent constraint programming (`utcc`)[14], which extends timed concurrent constraint programming [16] to include a universally quantified abstraction (ask) operation. Intuitively, the new operation added in `utcc`, written $(\lambda\ \vec{x}; \mathsf{c})\ P$, spans a copy of the residual process $P[\vec{t}/\vec{x}]$ for all possible inferences of $\mathsf{c}[\vec{t}/\vec{x}]$. This adds the ability to extend the scope of local knowledge which is not possible in CCP [9]. In particular it was illustrated in [14] how to model a notion of *link mobility* as found in the pi-calculus and to use the universal abstraction operator for communication of messages in security protocols.

However, the universal quantification in `utcc` is completely unrestricted. This means that in the proposed representations of link mobility and security protocols in `utcc`, every agent may guess channel names and encrypted values by universal quantification. It is thus necessary to enforce a restriction on the allowed processes to make sure that this is not possible.

As a general solution for making exactly such restrictions, we propose a simple type system for constraints used as patterns in abstractions, which essentially allow to distinguish between universally abstractable and secure variables in predicates. We also propose a novel notion of *abstraction under local knowledge*, which gives a general way to model that a process (principal) knows a key and can use it to decrypt a message encrypted with this key without revealing the key.

We exemplify the type system on π calculus-like mobility of local names and for giving semantics to a novel security protocol language called Security Protocol Concurrent Constraint Programming language (SPCCP), combining the best features of the Security CCP (SCCP) language proposed by Olarte and Valencia [14] and the Security Protocol Language (SPL) by Crazzolara and Winskel [6].

The foregoing document is divided as follows: Section 2 provides preliminary information and necessary definitions about constraint systems and the concurrent constraint family of programming languages. Section 3 introduces the type system for `utcc` the new abstraction rule over local knowledge, as well as termination and subject-reduction results over the type system proposed. In Section 4 we give more details on the use of the `utcc` with secure patterns. Finally, concluding remarks and future work are described in Section 5.

2 Preliminaries

This section provides the interested reader the main concepts of Temporal Concurrent Constraint Programming (`tcc`) and its universal extension (`utcc`), following the presentation of [14].

In CCP-based calculi all the (partial) information is *monotonically* accumulated in a so-called *store*. The store keeps the knowledge about the system in terms of *constraints*, or statements defining the possible values a variable can take (e.g., $x + y \geq 42$). Concurrent agents (i.e., processes) that are part of the system interact with each other using the store as a shared communication medium. They have two basic capabilities over the store, represented by *tell* and *ask* operations. While the former *adds* a piece of information about the system, the latter *queries* the store to determine if some piece of information can be inferred from its current content. Tell operations can act concurrently refining the information in the store while asks can serve as a general synchronization mechanism, that will be blocked if there is not enough information into the store to answer its query.

A fundamental notion in CCP-based calculi is that of a *constraint system*. Basically, a constraint system provides a signature from which syntactically denotable objects in the language called *constraints* can be constructed, and an entailment relation ($\Vdash$) specifying interdependencies among such constraints. More precisely,

Definition 1 (Constraint System). *A constraint system is a pair $CS = (\Sigma, \Delta)$ where Σ is a signature of function (F) and predicate (P) symbols, and Δ is a decidable theory over Σ (i.e., a decidable set of sentences over Σ with at least one model). The underlying language $\mathcal{L}$ of (Σ, Δ) contains the symbols $\neg, \wedge, \Rightarrow, \exists$ denoting logical negation, conjunction, implication, existential quantification.* Constants*, such as* `true` *and* `false` *denote the usual always true and always false values, respectively.* Constraints*, denoted by $\mathsf{c}, \mathsf{d}, \ldots$ are first-order formulae over $\mathcal{L}$. We say that* c entails d *in Δ, written $\mathsf{c} \Vdash_{\Delta} \mathsf{d}$ (or just $\mathsf{c} \Vdash \mathsf{d}$ when no confusion arises), if $\mathsf{c} \Rightarrow \mathsf{d}$ is true in all models of Δ. For operational reasons we shall require $\Vdash$ to be decidable.*

`tcc` arises as the extension of CCP for timed-systems: Including the notion of discrete time intervals (time units), a computation can be described as the interaction of a `tcc` process with the environment: At the instant i a `tcc` process P receives the store c as an initial stimulus, and when it reaches a quiescent point, it outputs d as the resulting constraint store with a residual process Q that will be executed in the instant $i + 1$. Here it is where one of the most important differences between ccp and `tcc` resides, as whilst the refinement of c during the execution of P at i is monotonic, d is not necessarily a refinement of c (that is, constraints can be forgotten).

Definition 2 (`tcc` process syntax). *Processes $P, Q, \ldots \in Proc$ are built from constraints $c \in \mathcal{C}$ and variables $x \in \mathcal{V}$ in the underlying constraint system by the following syntax.*

$$\begin{aligned} P, Q \ldots ::= \ & \mathbf{skip} \mid \mathbf{tell}\,(\mathsf{c}) \mid \mathbf{when}\, \mathsf{c}\, \mathbf{do}\, P \mid P \parallel Q \mid (\mathbf{local}\, \vec{x}; c)P \mid \\ & \mathbf{next}\,(P) \mid \mathbf{unless}\, c\, \mathbf{next}\,(P) \mid\, !P \end{aligned}$$

Intuitively, the process $\mathbf{skip}$ does nothing, $\mathbf{tell}\,(\mathsf{c})$ adds a new constraint c into the store, while $\mathbf{when}\ \mathsf{c}\,\mathbf{do}\,P$ asks if c is present into the store in order to execute P. $(\mathbf{local}\,\vec{x}; c)P$ binds a set of variables $\vec{x}$ in P by defining their existence under the constraint c. The operators associated with time allow the process to go one time unit in the future ($\mathbf{next}\,(P)$) or to define time-outs: if at the current time unit it is not possible to

$$\mathsf{R_T}\ \frac{}{\langle \mathbf{tell}\,(\mathsf{d}), \mathsf{c}\rangle \longrightarrow \langle \mathbf{skip}, \mathsf{c} \wedge \mathsf{d}\rangle} \qquad \mathsf{R_S}\ \frac{\gamma_1' \longrightarrow \gamma_2'}{\gamma_1 \longrightarrow \gamma_2}\ \text{if}\gamma_1 \equiv \gamma_1' \text{ and } \gamma_2 \equiv \gamma_2'$$

$$\mathsf{R_P}\ \frac{\langle P, \mathsf{c}\rangle \longrightarrow \langle P', \mathsf{c}'\rangle}{\langle P \parallel Q, \mathsf{c}\rangle \longrightarrow \langle P' \parallel Q, \mathsf{c}'\rangle} \qquad \mathsf{R_U}\ \frac{d \Vdash c}{\langle \mathbf{unless}\, \mathsf{c}\, \mathbf{next}\, P, \mathsf{d}\rangle \longrightarrow \langle \mathbf{skip}, \mathsf{d}\rangle}$$

$$\mathsf{R_R}\ \frac{}{\langle !P, \mathsf{c}\rangle \longrightarrow \langle P \parallel \mathbf{next}\,(!P), \mathsf{c}\rangle} \qquad \mathsf{R_L}\ \frac{\langle P, (\exists \tilde{\mathsf{x}}\mathsf{d}) \wedge \mathsf{c}\rangle \longrightarrow \langle P', (\exists \tilde{\mathsf{x}}\mathsf{d}) \wedge \mathsf{c}'\rangle}{\langle (\mathbf{local}\, \vec{x}; \mathsf{c})\ P, \mathsf{d}\rangle \longrightarrow \langle (\mathbf{local}\, \vec{x}; \mathsf{c}')\ P', (\exists \tilde{\mathsf{x}}\mathsf{c}') \wedge \mathsf{d}\rangle}$$

$$\mathsf{R_A}\ \frac{d \Vdash c[\vec{t}/\vec{x}] \qquad |\vec{t}| = |\vec{x}|}{\langle (\lambda\, \vec{x}; \mathsf{c})\ P, \mathsf{d}\rangle \longrightarrow \langle P[\vec{t}/\vec{x}] \parallel (\lambda\, \vec{x}; \mathsf{c} \wedge (\tilde{\mathsf{x}} \neq \tilde{\mathsf{t}}))\ P, \mathsf{d}\rangle}$$

$$\mathsf{R_O}\ \frac{\langle P, \mathsf{c}\rangle \longrightarrow^* \langle Q, \mathsf{d}\rangle \not\longrightarrow}{P \xRightarrow{(c,d)} F(Q)} \qquad \text{Where } F(Q) = \begin{cases} \mathbf{skip} & \text{if } Q = \mathbf{skip} \\ F(Q_1) \parallel F(Q_2) & \text{if } Q = Q_1 \parallel Q_2 \\ R & \text{if } Q = \mathbf{next}\,(R) \\ \mathbf{skip} & \text{if } Q = (\lambda\, \vec{x}; \mathsf{c})\ R \\ (\mathbf{local}\, \vec{x})\ F(R) & \text{if } Q = (\mathbf{local}\, \vec{x}; \mathsf{c})\ R \\ R & \text{if } Q = \mathbf{unless}\, \mathsf{c}\, \mathbf{next}\, R \end{cases}$$

Fig. 1. Transition System for `utcc`: Internal and Observable transitions

entail the constraint c then the process $\mathbf{unless}\, \mathsf{c}\, \mathbf{next}\, P$ will execute P at the next time unit. We will often use $\mathbf{next}^n\,(P)$ as a shorter version of $\mathbf{next}\,(\mathbf{next}\,(\ldots \mathbf{next}\,(P)))$ n-times. Finally, $P \parallel Q$ denotes the usual parallel execution and $!P$ denotes timed replication; that is, $!P = P \parallel \mathbf{next}\,(!P)$ executes P at the current time and replicates its behaviour over the next time period.

`utcc` [14] is an extension of the `tcc` calculus with a general *ask* defining a model of synchronization. While in `tcc` an ask $\mathbf{when}\, \mathsf{c}\, \mathbf{do}\, P$ is blocked if there is not enough information to entail c from the store, `utcc` inspires its synchronization mechanism on the notion of abstraction in functional programming languages. $(\lambda\, \vec{x}; \mathsf{c})\ P$ can be seen as the dual version of $(\mathbf{local}\, \vec{x}; \mathsf{c})\ P$ in which the variables are *abstracted* with respect to the constraint c and the process P. The operational semantics provides the intuitions on how `utcc` processes interact. In principle, a configuration is represented by the tuple $\langle P, \mathsf{c}\rangle$ where P denotes a set of processes and c a constraint store. P can evolve to a further process P' during an *internal transition* $(\rightarrow)$ where the constraint store c is monotonically refined, or can execute an *observable transition* $(\Longrightarrow)$, producing the result of the future function of P and the constraint store d. The set of operational rules is presented in Figure 1, where $\langle P, \mathsf{c}\rangle$ denotes a configuration, and $F(P)$ denotes the *future function of* P.

Definition 3 (Structural Congruence). *Structural congruence (denoted by $\equiv$) is defined for* `utcc` *by the axioms: (i) $P \equiv Q$ if they are α-equivalent. (ii) $P \parallel \mathbf{skip} \equiv P$. (iii) $P \parallel Q \equiv Q \parallel P$. (iv) $P \parallel (Q \parallel R) \equiv (P \parallel Q) \parallel R$. (v) $(\mathbf{local}\, \vec{x}; \mathsf{c})\ \mathbf{skip} \equiv \mathbf{skip}$. (vi) $P \parallel (\mathbf{local}\, \vec{x}; \mathsf{c})\ Q \equiv (\mathbf{local}\, \vec{x}; \mathsf{c})\ (P \parallel Q)$ if $\vec{x} \notin fv(P)$. (vii) $\langle P, c\rangle \equiv \langle Q, c\rangle$ iff $P \equiv Q$.*

Intuitively, the operational rules of `utcc` behaves almost in the same way as its counterpart in `tcc`, excepting by the general treatment of asks in `utcc`. Here we will describe the operational consequence of this change, we refer to [14] for further details on the operational semantics. Rule $\mathsf{R_A}$ describes the behavior of the abstraction $(\lambda\ \vec{x}; \mathsf{c})\ P$: a configuration here considers two stores, being c and d *local* and *global* stores respectively. If d entails $\mathsf{c}[\tilde{\mathsf{t}}/\tilde{\mathsf{x}}]$ then $P[\vec{t}/\vec{x}]$ is executed. Moreover, the abstraction persists in time, allowing any other process to match with $\vec{x}$ in P while no other replacements of $\vec{x}$ with $\vec{t}$ will occur, as d is augmented with a constraint disallowing this. The notion of *local information* can be evidenced in $\mathsf{R_L}$, considering a process $P = (\mathbf{local}\ \vec{x}; \mathsf{c})\ Q$, we have to consider: (i) that the information about $\vec{x}$ locally for P subsumes any other information present for the same set of variables in the global store; therefore, $\vec{x}$ is hidden by the use of an existential quantifier over $\tilde{\mathsf{x}}$ in d. (ii) that the information about $\vec{x}$ that P can produce after the reduction is still local, so we hide it by existentially quantifying $\vec{x}$ in c' before publishing it to the global store. After the reduction, c' will be the new local store of the evolution of internal processes. Finally, observable behaviour is described by $\mathsf{R_o}$: after having used the internal transitions in a process P to evolve to a process Q with a quiescent-point (in which no more information can be added/inferred), the reduction will continue by executing the future function of Q with the resulting constraint store.

3 `utcc` and Secure Pattern Matching

As described in Sec. 2, one of the main advantages of `utcc` with respect to `tcc` is that the universal abstraction operator allows for substitution of constraints for variables in processes. The extension has been proposed for the treatment of *mobile links* as present in the π -calculus [12] and pattern matching in modeling of security protocols. Below we will give two motivating examples for why a more refined abstraction operator is needed for modeling mobile *local* links and secret keys.

3.1 Motivating a Refined Universal Abstraction in `utcc`

Our first example refers to the π calculus-like mobility of local links. Consider the common scenario where a process P sends a request to a service offered by a process Q and includes in the request a local link on which it expects the reply. This can be modeled in `utcc` using a constraint system $CS = (\Sigma, \Delta)$ where Σ includes the predicates req, rep, and res, and the constant $\mathsf{0}$. The processes P and Q are defined as

$$P = (\mathbf{local}\ z)\ \big(\mathbf{tell}\,(\mathsf{req(z)})\ \parallel\ (\lambda\ y; \mathsf{rep(z,y)})\ \mathbf{next}\ (\mathbf{tell}\,(\mathsf{res(y)}))\big)$$

and

$$Q = (\lambda\ x; \mathsf{req(x)})\ \mathbf{tell}\,(\mathsf{rep(x,0)})$$

The predicates req and rep are used for the request and reply respectively, and the predicate res is used to report the result (and successful termination of P). The local operator is used to create a local variable z representing the local link.

The intention is that only the processes P and Q can synchronize via the local link z. However, the generality of abstraction in utcc makes it possible to violate this intention: Another process $E = (\lambda\, x, y; \mathsf{rep(x, y)})$ **skip** in parallel with the processes P and Q given above would be able to guess the link z (as well as the result) from the reply.

It is instructive to see how this could be avoided using the π-calculus, where the two processes could be modeled by

$$P = (\nu z)\big(\overline{\mathsf{req}}\langle z\rangle \parallel z\langle y\rangle.\overline{\mathsf{res}}\langle y\rangle\big) \quad \text{and} \quad Q = \mathsf{req}(x).\overline{x}\langle 0\rangle$$

In this case, the z and y are used differently in receiving the reply: The z is used as the communication channel and y is the binder for the received name. Another process in parallel would not be able to guess the channel z. As we will see below, our proposed type system for patterns allows to introduce this kind of distinction between the uses of variables in predicates.

Our second motivating example is from modeling of security protocols, where as pointed out in [4] it it should be impossible for an agent to abstract variables if a one-way function has been applied to it. Consider a unary predicate o (used for output of messages to the network) and an encryption function $\mathsf{enc(m, k)}$ which represents the encryption of the variable m with the key k. A process P that sends out a local message n encrypted by a local key k can be represented by $P = (\textbf{local}\ k, n)\ \textbf{tell}\,(\mathsf{o(enc(n, k))})$. However, in utcc a spy process defined as $S = (\lambda\, x, z; \mathsf{o(enc(x, z))})\ \textbf{tell}\,(\mathsf{o(x)} \wedge \mathsf{o(z)})$, will succeed in retrieving and publishing both the key and the encrypted message.

As for the π-like channels, our proposed type system for patterns will allow us to rule out universal abstraction of variables to which a one-way function has been applied. Further, to be able to allow abstraction of the message when the key is locally known, we propose a novel kind of abstraction assuming local knowledge, which generalizes the universal abstraction of utcc.

3.2 Types for Secure Abstraction Patterns in utcc

Based on the two motivating examples above, we argue that there are basically two sorts of arguments in functions and predicates: the ones that can be universally quantifiable, which means that one would be able to use the abstraction operator for a variable in that argument in order to find a possible matching, and the ones that are not.

We will thus divide the arguments of predicates and functions in two sorts and write $\mathsf{P}(\tilde{\mathsf{t}}; \tilde{\mathsf{t}}')$ and $\mathsf{f}(\tilde{\mathsf{t}}; \tilde{\mathsf{t}}')$ for respectively the predicate P and function f where both $\vec{t}$ and $\vec{t'}$ are tuples of terms over the function signature F, and $\vec{t}$ denotes the restricted arguments and $\vec{t'}$ the unrestricted ones. We assume that both arguments of the equality predicate are restricted. If a predicate or function has either only restricted or unrestricted parameters and the sort is clear from the context, we will simply write $\mathsf{P}(\tilde{\mathsf{t}})$ and $\mathsf{f}(\tilde{\mathsf{t}})$.

The sorted predicates allow us to use a binary predicate $\mathsf{piout(x; y)}$ representing the π-like communication of y (the object) on the channel x (the subject). By defining that the subject is a restricted argument and the object an unrestricted argument we obtain the required asymmetry in the roles of the variables. The type rules for patterns should then forbid the abstraction $(\lambda\ x; \mathsf{piout(x, y)})\ P$, as it would allow us to identify all channels (also channels not known to us) containing a particular message y. However, they should *allow* the abstraction $(\lambda\ y; \mathsf{piout}(\mathsf{x},\mathsf{y}))\ P$, reflecting that we can compute the possible messages on a channel x known to us. That is, we want to capture that if we *know* the values of the restricted variables, then we may abstract (i.e. compute all possible matches for) the unrestricted variables.

Similarly, sorted functions allow us to represent semantically that some functions are *one-way* functions such as the function $\mathsf{enc}(\mathsf{k},\mathsf{m})$ described above for encrypting the message m by the key k. Sorting both arguments as restricted will ensure that e.g. the abstractions $(\lambda\ \vec{x}; \mathsf{o}(\mathsf{enc}(\mathsf{k},\mathsf{m})))\ P$ will be forbidden for any non-empty $\vec{x} \subseteq \{\mathsf{k},\mathsf{m}\}$ Thus, even if the single argument of the o predicate is unrestricted (i.e. we can abstract all messages available on the network) then we can not compute the inverse of the encryption function. We may have functions for which an inverse is assumed to exist, such as a function $\mathsf{tup}_2(\mathsf{x},\mathsf{y})$ for making a pair of x and y. In that case it makes sense to allow abstractions over the two arguments by sorting them as unrestricted.

In general, patterns may be a conjunction of several predicates and thus variables may occur both restricted and unrestricted in the same pattern. An example of this is the abstraction $(\lambda\ y,z; \mathsf{c})\ P$, where $\mathsf{c} = \mathsf{piout}(\mathsf{y},\mathsf{z}) \wedge \mathsf{piout}(\mathsf{x},\mathsf{y})$. We argue that this pattern should be *allowed*, since it is possible first to match the unrestricted y in $\mathsf{piout}(\mathsf{x},\mathsf{y})$ and then subsequently, for the given y, match the unrestricted z in $\mathsf{piout}(\mathsf{y},\mathsf{z})$. Note that it is not enough simply to require the abstracted variables to occur unrestricted: Both variables x and y appear unrestricted in the abstraction $(\lambda\ x,y; \mathsf{piout}(\mathsf{x},\mathsf{y}) \wedge \mathsf{piout}(\mathsf{y},\mathsf{x}))\ P$, but neither of the two basic constraints can be matched without abstracting a restricted variable. As solution we define a set of type rules for constraints used as patterns in abstractions which capture that there exists an order of the basic constraints in which the first occurrence of each variable is unrestricted.

To allow abstractions in cases where the inverse key of the encryption is known we add a new rule $\mathsf{R}_{\mathsf{A}\rightarrow}$ given in Equation 1 in addition to the SOS rules pictured in Figure 1. $\mathsf{R}_{\mathsf{A}\rightarrow}$ allows for abstractions using constraints of the form $\mathsf{c} \Rightarrow \mathsf{c}'$, that is, assuming local knowledge c and a global store d, one can infer c'. The idea is to infer c' using c but without publishing it permanently to the store, as captured by the following operational rule:

$$\mathsf{R}_{\mathsf{A}\rightarrow} \quad \frac{\mathsf{d} \wedge \mathsf{c} \Vdash \mathsf{c}'[\tilde{\mathsf{t}}/\tilde{\mathsf{x}}] \quad |\tilde{\mathsf{t}}| = |\tilde{\mathsf{x}}| \quad \mathsf{d} \wedge \mathsf{c} \Vdash \mathtt{false} \Rightarrow \mathsf{d} \Vdash \mathtt{false}}{\langle(\lambda\ \vec{x}; \mathsf{c} \Rightarrow \mathsf{c}')\ P, \mathsf{d}\rangle \longrightarrow \langle P[\vec{t}/\vec{x}] \parallel (\lambda\ \vec{x}; \mathsf{c} \Rightarrow (\mathsf{c}' \wedge (\tilde{\mathsf{x}} \neq \tilde{\mathsf{t}}))\ P, \mathsf{d}\rangle} \tag{1}$$

The condition $\mathsf{d} \wedge \mathsf{c} \Vdash \mathtt{false} \Rightarrow \mathsf{d} \Vdash \mathtt{false}$ ensures that local assumptions do not make the store inconsistent when combining with the constraint store.

The typing rules for secure patterns and processes are defined in Figure 2. For simplicity we assume patterns are simply conjunction of predicates applied to terms over the function signature. The typing rules use an environment $\Gamma = \Gamma^R; \Gamma^U$, where Γ^R is the set of names used restricted and Γ^U is the set of names used unrestricted. When the distinction does not matter we simply write Γ. We employ three inductively defined functions on terms over the function signature: $unr(t)$, $res(t)$, and $var(t)$ yielding respectively the variables appearing unrestricted in t according to the sorting, the variables appearing restricted in t, and all variables appearing in t. We extend the functions to vectors of terms by $unr(\vec{t}) = \cup_{1 \leq i \leq |\vec{t}|} unr(t_i)$ (and similarly for res and var). Formally, the functions are given by $unr(x) = res(x) = var(x) = \{x\}$ for any variable x, and

$$\mathsf{T}_{\mathsf{pred}}\ \frac{}{\Gamma^R;\Gamma^U \vdash \mathsf{P}(\vec{t};\vec{t'}) : pat}\ \Gamma^R = var(\vec{t}) \cup res(\vec{t'}) \text{ and } \Gamma^U = unr(\vec{t'})\backslash\Gamma^R$$

$$\mathsf{T}_{\mathsf{assoc}}\ \frac{\Gamma \vdash \mathsf{c}_1 \wedge (\mathsf{c}_2 \wedge \mathsf{c}_3) : pat}{\Gamma \vdash (\mathsf{c}_1 \wedge \mathsf{c}_2) \wedge \mathsf{c}_3 : pat} \qquad \mathsf{T}_{\mathsf{commute}}\ \frac{\Gamma \vdash \mathsf{c}_1 \wedge \mathsf{c}_2 : pat}{\Gamma \vdash \mathsf{c}_2 \wedge \mathsf{c}_1 : pat}$$

$$\mathsf{T}_{\mathsf{comb}}\ \frac{\Gamma_1^R;\Gamma_1^U \vdash \mathsf{c}_1 : pat \quad \Gamma_2^R;\Gamma_2^U \vdash \mathsf{c}_2 : pat}{\Gamma^R;\Gamma^U \vdash \mathsf{c}_1 \wedge \mathsf{c}_2 : pat}\ \Gamma^R = (\Gamma_1^R \cup \Gamma_2^R)\backslash\Gamma_1^U \text{ and } \Gamma^U = (\Gamma_1^U \cup \Gamma_2^U)\backslash\Gamma_1^R$$

$$\mathsf{T}_{\mathsf{skip}}\ \frac{}{\vdash \mathbf{skip} : sec} \qquad \mathsf{T}_{\mathsf{tell}}\ \frac{}{\vdash \mathbf{tell}\,(\mathsf{c}) : sec}$$

$$\mathsf{T}_{\mathsf{par}}\ \frac{\vdash P : sec \quad \vdash Q : sec}{\vdash P \parallel Q : sec} \qquad \mathsf{T}_{\mathsf{next}}\ \frac{\vdash P : sec}{\vdash \mathbf{next}\,(P) : sec}$$

$$\mathsf{T}_{\mathsf{bang}}\ \frac{\vdash P : sec}{\vdash !P : sec} \qquad \mathsf{T}_{\mathsf{unls}}\ \frac{\vdash P : sec}{\vdash \mathbf{unless}\,\mathsf{c}\,\mathbf{next}\,P : sec}$$

$$\mathsf{T}_{\mathsf{abs}}\ \frac{\vdash P : sec \quad \Gamma^R;\Gamma^U \vdash \mathsf{c} : pat}{\vdash (\lambda\,\vec{x};\mathsf{d} \implies \mathsf{c})\,P : sec}\ \vec{x} \subseteq dom(\Gamma^U)\backslash fv(d) \qquad \mathsf{T}_{\mathsf{loc}}\ \frac{\vdash P : sec}{\vdash (\mathbf{local}\,\vec{x};\mathsf{c})\,P : sec}$$

Fig. 2. Typing rules for secure patterns and processes

$unr(f(\vec{t};\vec{t'})) = unr(\vec{t'})$, $res(f(\vec{t};\vec{t'})) = res(\vec{t})$, and $var(f(\vec{t};\vec{t'})) = var(\vec{t}) \cup var(\vec{t'})$. Note that obviously $var(t) = res(t) \cup unr(t)$ but also that $res(t) \cap unr(t)$ may be non-empty, i.e. a variable may appear both restricted and non-restricted.

The rule $\mathsf{T}_{\mathsf{Pred}}$ captures that all variables in $\vec{t}$ as well as the variables occurring restricted in $\vec{t'}$ in the predicate $\mathsf{P}(\vec{t};\vec{t'})$ are restricted. The rest of the variables are unrestricted. The rules $\mathsf{T}_{\mathsf{asoc}}$ and $\mathsf{T}_{\mathsf{commute}}$ allow us to change the ordering of the basic constraints. Finally, the rule $\mathsf{T}_{\mathsf{comb}}$ identifies the restricted and unrestricted variables in the joint pattern $\mathsf{c}_1 \wedge \mathsf{c}_2$ assuming that c_1 is matched first. That is, a variable is restricted if it appears restricted in either of the sub patterns c_1 and c_2 and not unrestricted in c_1. (If it appears unrestricted in c_1 it will be instantiated if c_1 is matched first, and thus it is allowed to appear restricted in c_2). Dually, the unrestricted variables in the joint pattern $\mathsf{c}_1 \wedge \mathsf{c}_2$ are the variables that appear unrestricted in either of the sub patterns c_1 and c_2, and do not appear restricted in c_1.

The objective of the type system is to determine the secure patterns, therefore typing rules over processes are rather simple. The only non-trivial rule is the rule $\mathsf{T}_{\mathsf{abs}}$ for abstractions, which ensure that c is a valid pattern such that the abstracted variables are unrestricted, and no variables in the local d are abstracted.

Theorem 1 (Termination of type checking). *For any process P the type-checking process terminates.*

Proof. (Sketch) Follows from the fact that there are only finitely many permutations of basic constraints (predicates) in a pattern.

The following lemmas are used to prove subject reduction.

Lemma 1 (Constraint substitution does not affect pattern typing). *Given* $\Gamma^R; \Gamma^U \vdash \mathsf{c} : pat$ *and* t *and* x*, then* $\Gamma^{R'}; \Gamma^{U'} \vdash \mathsf{c}[t/x] : pat$ *and* $\Gamma^U \setminus (fv(t) \cup \{x\}) \subseteq \Gamma^{U'} \setminus (fv(t) \cup \{x\})$.

Proof. (Outline) The proof proceeds by induction on the type inference of $\Gamma^R; \Gamma^U \vdash \mathsf{c} : pat$.

Lemma 2 (Constraint substitution does not affect process typing). *Given a typing judgment* $\vdash P' : sec$ *then* $\vdash P'[t/x] : sec$.

Proof. (Outline) The proof proceeds by induction on the type inference of $\vdash P' : sec$

Lemma 3 (Structural equivalence preserves typing). *Given* P, Q *processes, if* $P \equiv Q$ *and* $\vdash P : sec$ *, then* $\vdash Q : sec$.

Proof. The proof proceeds by trivial case analysis over the structural congruence rules in Definition 3.

Next we check that secure processes can not be made insecure during an internal transition step.

Lemma 4. *If* $\langle P, \mathsf{c}\rangle \longrightarrow \langle Q, \mathsf{d}\rangle$ *and* $\vdash P : sec$ *, then* $\vdash Q : sec$.

Proof. (Outline) The proof proceeds by induction on the depth of the inference $\langle P, \mathsf{c}\rangle \longrightarrow \langle Q, \mathsf{d}\rangle$ and using the definition of $\vdash P : sec$.

Finally, we show that if a process P is well-typed, it can not perform any internal steps, and its future is defined then the future of P is also well-typed.

Lemma 5. *For all* $\vdash P : sec$*, if* $F(P)$ *is defined and* $\exists \mathsf{d}. \langle P, \mathsf{d}\rangle \not\longrightarrow$ *then* $\vdash F(P) : sec$.

Proof. (Outline) The proof proceed by induction in the definition of $F(P)$.

We now have all the ingredients to prove subject reduction.

Theorem 2 (Subject-reduction). *If* $P \overset{(c,d)}{\Longrightarrow} Q$ *and* $\vdash P : sec$ *, then* $\vdash Q : sec$.

Proof. Assume $P \overset{(c,d)}{\Longrightarrow} Q$ and $\vdash P : sec$, then by rule $\mathsf{R_o}$ we get that $\langle P, \mathsf{c}\rangle \longrightarrow^n \langle Q', \mathsf{d}\rangle \not\longrightarrow$ and $Q = F(Q')$. We proceed by induction in n.

In the base case where $n = 0$, we have that $Q' = P$ and $\mathsf{c} = \mathsf{d}$. It follows from lemma 5 that $\vdash F(Q') : sec$.

For the induction step, assume $\langle P, \mathsf{c}\rangle \longrightarrow^1 \langle P', \mathsf{c}'\rangle \longrightarrow^n \langle Q', \mathsf{d}\rangle \not\longrightarrow$. Then $\vdash P' : sec$ by lemma 4 and thus we get by induction that $\vdash F(Q') : sec$.

4 Applications

This section illustrates the use of the type system with some examples in mobility and security. First, let us return to the π calculus example.

We assume the syntactic sugar $x\langle y\rangle$ stands for the binary predicate $\mathsf{piout}(x;y)$ and represents the use of the (restricted) channel x with the (unrestricted) message y. The following type inference show that we can quantify over either x or y for the pattern $y\langle x\rangle \wedge x\langle y\rangle$:

$$\dfrac{\dfrac{}{x;y \vdash x\langle y\rangle : pat}\,\mathsf{T_{pred}} \quad \dfrac{}{y;x \vdash y\langle x\rangle : pat}\,\mathsf{T_{pred}}}{x;y \vdash x\langle y\rangle \wedge y\langle x\rangle : pat}\,\mathsf{T_{comb}}$$

The way to read the first inference is that we can abstract y if we know x. Conversely, a second inference from the same pattern can lead to a typing of the form $y;x \vdash y\langle x\rangle \wedge x\langle y\rangle : pat$, capturing the fact that one can abstract x if we know y. However, note that we can not infer $\epsilon; x, y \vdash x\langle y\rangle \wedge y\langle x\rangle : pat$, and thus we are not allowed to simultaneously quantify over x and y.

To illustrate the application of $\mathtt{utcc}_s$ in the security domain, we follow the lines of the Security Protocol Language (SPL) [6] and SCCP [13] to define a specification language for security protocols that we have called the Security Protocol Concurrent Constraint Programming (SPCCP) language. The SPCCP embeds $\mathtt{utcc}_s$ in a syntax suitable for defining security protocols, capturing process specifications with respect to input and output events over a global network. The SPCCP language combines the best ideas from SPL and SCCP by having a simple notion of pattern matching as in SPL and using the constraint system to model the attackers ability to combine and split messages as in SCCP. Hereto we add the new concept of *pattern matching under local knowledge*, which allow us to syntactically guarantee that only message parts inferable from the available keys are extracted, which can not be guaranteed in SPL nor in SCCP.

Definition 4 (SPCCP). *The Secure Concurrent Constraint Programming language* SCCP *[13] is redefined by the following grammar:*

$$\begin{array}{lrl}
\textit{Values} & v,v' = & x \mid k \\
\textit{Keys} & k = & pub(x) \mid priv(x) \mid sym(x) \\
\textit{Messages and patterns} & M,N = & v \mid (M_1,\ldots,M_n) \mid \{M\}_k \\
\textit{Processes} & R = & \mathbf{nil} \mid \mathbf{local}(x)\,\mathbf{in}\,R \mid \mathbf{out}(M)\,.R \\
 & & \mid \mathbf{in}_{\forall}\vec{x}[N]_{\vec{k}}.R \mid !R \mid R \parallel R
\end{array},$$

where x range over a set of variables and the subscript $\vec{k}$ in $\mathbf{in}_{\forall}\vec{x}[N]_{\vec{k}}.R$ is a set of keys.

We define the semantics of SPCCP by giving a translation into $\mathtt{utcc}_s$ with a security constraint system given by the signature Σ with a single (unrestricted) unary predicate $\mathsf{o(t)}$ used for message output, and function symbols $\mathsf{F} = \{\mathsf{enc}, \mathsf{pub}, \mathsf{priv}, \mathsf{sym}, \mathsf{tup_n}\}$, and entailment relation given in Fig. 3 inspired on the requirements stated by Dolev and Yao in [7].

The binary function enc takes two unrestricted arguments: a key and a message. The key is intended to be either a symmetric, private, or public key generated by the (restricted) unary functions $sym(x)$, $priv(x)$, or $pub(x)$ respectively. Letting $\mathsf{k} \in \{pub, priv, sym\}$ and defining $sym^{-1} = sym$, and $pub^{-1} = priv$, the entailment rule scheme $\mathsf{E_{k-dec}}$ for decryption expresses how enc acts as symmetric or asymmetric encryption. The n-ary (unrestricted) tupling functions tup_n allow to create n-ary tuples,

$$\mathsf{E_{k-dec}}\quad \frac{\mathsf{c} \Vdash \mathsf{o(k^{-1}(x))} \quad \mathsf{c} \Vdash \mathsf{o(enc(k(x),m))}}{\mathsf{c} \Vdash \mathsf{o(m)}}, \text{ for } \mathsf{k} \in \{sym, pub\},\ sym^{-1} = sym, \text{ and } pub^{-1} = priv$$

$$\mathsf{E_{enc}}\quad \frac{\mathsf{c} \Vdash \mathsf{o(x)} \qquad \mathsf{c} \Vdash \mathsf{o(y)}}{\mathsf{c} \Vdash \mathsf{o(enc(x,y))}} \qquad \mathsf{E_{k-key}}\quad \frac{\mathsf{c} \Vdash \mathsf{o(x)}}{\mathsf{c} \Vdash \mathsf{o(k(x))}}, \text{ for } \mathsf{k} \in \{sym, pub, priv\}$$

$$\mathsf{E_{tup_n}}\quad \frac{\mathsf{c} \Vdash \mathsf{o(i_1)} \quad \dots \quad \mathsf{c} \Vdash \mathsf{o(i_n)}}{\mathsf{c} \Vdash \mathsf{o(tup_n(i_1,\dots,i_n))}} \qquad \mathsf{E_{proj}}\quad \frac{\mathsf{c} \Vdash \mathsf{o(tup_n(i_1,\dots,i_n))}}{\mathsf{c} \Vdash \mathsf{o(i_j)}} \quad j \in \{1,\dots,n\}$$

Fig. 3. Entailment relation for a security constraint system

from which the individual elements can be projected as expressed by the entailment rule $\mathsf{E_{proj}}$. As usual, the rules $\mathsf{E_{enc}}$, $\mathsf{E_{k-key}}$, and $\mathsf{E_{tup_n}}$ express that the output of any function of known output values can be inferred.

The messages/patterns of SPCCP are mapped to the terms generated by the corresponding function symbols and variables in the security constraint system, using the usual notation $(M_1,\dots,M_n)$ for n-tuples and $\{M\}_k$ for $\mathsf{enc}(\mathsf{k}, M)$. For a message M of SPCCP let $v(M)$ denote the set of variables in M. For a set of values $\vec{v} = \{v_1, v_2, \dots, v_i\}$ let $o(\vec{v})$ be short for $o(v_1) \wedge o(v_2) \wedge \dots \wedge o(v_i)$, and in particular $o(\emptyset) = \mathtt{true}$.

We are now ready to define the encoding of SPCCP in $\mathtt{utcc}_s$.

Definition 5 (SPCCP **encoding)**

$$[\![R]\!] : \begin{cases} \textbf{skip} & \textit{if } R = \textbf{nil} \\ (\textbf{local}\, x)\ [\![R']\!]_{\mathtt{utcc}} & \textit{if } R = \textbf{local}(x)\ \textbf{in}\ R' \\ \textbf{tell}\,(\mathsf{o(M)})\ \parallel\ \textbf{next}\ ([\![R']\!]_{\mathtt{utcc}}) & \textit{if } R = \textbf{out}(M)\,.R' \\ (\lambda\,\vec{x}; \mathsf{o}(\tilde{\mathsf{k}}) \Rightarrow \mathsf{o(N)} \wedge \mathsf{o}(\tilde{\mathsf{x}}))\ \textbf{next}\,([\![R']\!]_{\mathtt{utcc}}) & \textit{if } R = \textbf{in}_\forall \vec{x}[N]_{\vec{k}}.R' \\ ![\![R']\!]_{\mathtt{utcc}} & \textit{if } R = !R' \\ [\![R']\!]_{\mathtt{utcc}} \parallel [\![R'']\!]_{\mathtt{utcc}} & \textit{if } R = R' \parallel R'' \end{cases}$$

We will focus on outlining process constructions for pattern matching and network output. The remaining process constructions are mapped directly to the corresponding construct in $\mathtt{utcc}_s$.

$\textbf{out}(M).\,R$ adds the constraint $\mathsf{o(M)}$ to the constraint store and subsequently in the next time period behaves as (the encoding of) R.

SPCCP differs from SCCP in the treatment of keys and the input operation: priv(x), pub(x), and sym(x) yields respectively the private, public and symmetric key from generator x. The input operator written as $\textbf{in}_\forall \vec{x}[N]_{\vec{k}}.P$ should be read as "for all possible messages $\vec{m}$ (available under the assumption of knowing the keys $\vec{k}$) such that $N[\vec{m}/\vec{x}]$ is available as message at the network evolve into $P[\vec{m}/\vec{x}]$". Intuitively, the idea is to check if $\vec{m}$ is available as knowledge assuming locally that the keys in $\vec{k}$ are available as knowledge, and if so, bind the variables in P occurring in the pattern N with the corresponding values in $\vec{m}$. The pattern matching resembles the pattern matching construct in SPL. The key difference is that it proceed for all possible matches, and that we employ the new rule for for universal abstraction under local knowledge introduced in the previous section to allow the use of private keys as local information to perform the decryption of messages. Note that we also require that all the abstracted values can be inferred as output. This guarantees that secret values are not abstracted, and result in well-typedness of the encoding.

Proposition 1 **(SPCCP maps to well-typed $\mathtt{utcc}_s$ processes).** *For any* SPCCP *process* P, $\vdash [\![P]\!] : sec$.

4.1 Protocols

In Fig. 4 below we recall the protocol steps of the Needham-Schröeder-Lowe protocol [10] (herewith referred as NSL) used as example in [6].

$$\begin{array}{l}
(1)\ A \rightarrow B\ :\ \{m, A\}_{pub(B)} \\
(2)\ B \rightarrow A\ :\ \{m, n, B\}_{pub(A)} \\
(3)\ A \rightarrow B\ :\ \{n\}_{pub(B)}
\end{array}$$

Fig. 4. Needham-Schröeder-Lowe protocol with public-key encryption

The NSL protocol describes the interaction between agents A and B. First A sends to B a nonce along its agent name, encrypted with B's public key. Then B decrypts the message with his own private key extracting A's nonce. Next, B sends a message to A containing the proof of reception along with a fresh name encrypted under A's public key. Finally, A decrypts B's message and sends to B the name challenge received in the previous message encrypted with B's public key. The SPCCP version of the protocol is given in Fig. 5.

SPCCP share some similarities with the approaches in LYSA^{NS} [4], SCCP , and the SPL calculus. Particularly, observe that there is no need to explicitly define the communication channels in which agents are transmitting messages. The underlying model acts as an open network in which every actor can access all the messages posted provided that he has the proper keys to decrypt its the message. We assume a disclosure of public keys for every agent, while the private keys are kept secret for each principal. The key difference between the approach in SPCCP to the approaches in SPL and SCCP is that the abstraction of the contents of a message encrypted with a key is only

$$\begin{array}{lll}
Init(A, B, k_A, p_B) & = & \mathbf{new}(m)\,\mathbf{out}(\{m, A\}_{p_B}). \\
& & \quad \mathbf{in}_{\forall x}[\{m, x, B\}_{pub(k_A)}]_{priv(k_A)}. \\
& & \quad \mathbf{out}(\{x\}_{p_B}).\,\mathbf{nil} \\[1em]
Resp(A, B, k_B, p_A) & = & \mathbf{in}_{\forall y}[\{y, A\}_{pub(k_B)}]_{priv(k_B)}. \\
& & \quad \mathbf{new}(n)\,\mathbf{out}(\{y, n, B\}_{p_A}). \\
& & \quad \mathbf{in}_{\forall}[\{n\}_{pub(k_B)}]_{priv(k_B)}.\mathbf{nil} \\[1em]
System(A, B) & = & \mathbf{new}(k_A)\,\mathbf{new}(k_B)\,(Init(A, B, k_A, pub(k_B)) \\
& & \quad \parallel Resp(A, B, k_B, pub(k_A)))
\end{array}$$

Fig. 5. NSL protocol in SPCCP

allowed if one possesses the corresponding key for decryption. This is similar to the approach in the LYSA^{NS} calculus [4], except that we employ the constraint system and local knowledge instead of tailoring the pattern matching with a notion of key pairs.

The following specification exemplifies the translation into $\mathtt{utcc}_s$:

$$\begin{aligned}
Init(A,B,k_A,p_B) &= (\mathbf{local}\, m)\, \mathbf{tell}\, (\mathsf{o}(\{\langle \mathsf{m},\mathsf{A}\rangle\}_{\mathsf{p_B}})\\
&\quad \parallel \mathbf{next}\, (((\lambda\, x; \mathsf{o}(\mathsf{priv}(\mathsf{k_A})) \Rightarrow (\mathsf{o}(\{\mathsf{m},\mathsf{x},\mathsf{B}\}_{\mathsf{pub}(\mathsf{k_A})}) \wedge \mathsf{o}(\mathsf{x})))\ )\\
&\quad \parallel \mathbf{next}\, (\mathbf{tell}\, (\mathsf{o}(\{\mathsf{x}\}_{\mathsf{pub_B}})) \parallel \mathbf{next}\, (\mathbf{skip})))
\end{aligned}$$

$$\begin{aligned}
Resp(A,B,k_B,p_A) &= (\lambda\, y; \mathsf{o}(\mathsf{priv}(\mathsf{k_B})) \Rightarrow (\mathsf{o}(\{\mathsf{y},\mathsf{A}\}_{\mathsf{pub}(\mathsf{k_B})}) \wedge \mathsf{o}(\mathsf{y})))\\
&\quad \parallel \mathbf{next}\, (((\mathbf{local}\, n)\, \mathbf{tell}\, (\mathsf{o}(\{\mathsf{y},\mathsf{n},\mathsf{B}\}_{\mathsf{p_A}})))\\
&\quad \parallel \mathbf{next}\, ((\lambda\, \emptyset; \mathsf{o}(\mathsf{priv}(\mathsf{k_B})) \Rightarrow (\mathsf{o}(\{\mathsf{n}\}_{\mathsf{pub}(\mathsf{k_B})})))\ \parallel \mathbf{next}\, (\mathbf{skip})))
\end{aligned}$$

$$\begin{aligned}
System(A,B) &= (\mathbf{local}\, k_A)\, (\mathbf{local}\, k_B)\, Init(A,B,k_A,pub(k_B))\\
&\qquad \parallel Resp(A,B,k_B,pub(k_A))
\end{aligned}$$

5 Conclusions and Future Work

We have illustrated that the introduction of universal quantification to CCP for modeling mobile communication and security protocols introduce the problem that information which should be local can be obtained by universal quantification. As a way to remedy the problems we have proposed a simple type system for constraints used as patterns in abstractions which allows us to guarantee semantically that e.g. channel names and encrypted values are only extracted by agents that are able to infer the channel or non-encrypted value from the store. Furthermore, we proposed a novel kind of abstraction allowing abstraction under the assumption of local knowledge. The latter can be applied to infer the plain text of encrypted messages under the assumption of knowledge of the key, without adding the key permanently to the global store. We exemplified the type system by examples of mobility of local links (in the context of the π -calculus) and provided a new language for security protocols combining the key features of the Security CCP (SCCP) language and the SPL calculus, but adding the ability to syntactically constraining the ability to decrypting secret values inspired by the LYSA^{NS} calculus.

The present work is only in its first stage. However, we believe that the proposed distinction between variables that can be universally quantified and variables that can not is an elegant way to remedy the problems we have illustrated connected to the universal quantification to CCP. A next step will be to perform a detailed investigation of the proposed new variant of the SCCP calculus and applications to model security protocols. In particular, we plan to investigate the application of the analysis techniques for SCCP , SPL and LYSA^{NS} to the SPCCP language.

It is important to remark the importance of the current proposal with respect to other analysis techniques for security protocols. In [3], a framework for the analysis of secrecy properties is proposed with logic programming as its underlying mechanism. The specification language follows the line of the equational theory presented in the Applied π -calculus [1], encoding constructor and destructor functions by means of deduction rules in the framework. Here, pattern-matching is being used to encode the abilities of an attacker to abstract away information from the facts present in the store. Given that the attacker can apply the set of rules in a given specification, the correctness of the analysis relies on the power we give on the inference system. For instance, a rule $\mathsf{attaker(sign(m, sk))} \rightarrow \mathsf{attacker(sk)}$ could be specified and the attacker would be able to extract away the secret key from a signature. We believe that a type system similar to the one proposed in this paper can be applied here to limit the extra expressive power of the rule-based approach by allowing only to abstract only variables over unrestricted parts of the predicates, ruling out the example given above by declaring sk a restricted variable over $\mathsf{sign(m, sk)}$. Similar considerations can be applied to other systems that base their analysis on pattern-matching techniques, like the extended strand-space approach in [5] and Miller's linear logic approach for security protocols [11].

As also pointed out in the text the local operator of `utcc` does not really correspond to the generation of new names in nominal calculi. This has already been noticed by Palamidessi et al. [15], where a logical characterization of name restriction using the existential quantifier does not ensure uniqueness in the fragment of the π -calculus with mismatch. The same occurs in `utcc`: a process $(\mathbf{local}\ x)\ (\mathbf{local}\ y)\ P$ can hide both x and y from the store, but the current logical formulation does not ensure the uniqueness of x and y, as one may wish when dealing with nonces for security protocols. We leave for future work to study variants of the local operator ensuring uniqueness.

Acknowlegments. The authors would like to thank the anonymous reviewers for their suggestions for improvement of this paper. We would also like to thank Jorge A. Pérez and Frank Valencia for their comments on earlier versions of this document. This research has been partially supported by the Trustworthy Pervasive Healthcare Services (TrustCare) project. Danish Research Agency, Grant # 2106-07-0019 (www.TrustCare.eu)

References

1. Abadi, M., Fournet, C.: Mobile values, new names, and secure communication. In: POPL 2001 Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 104–115. ACM Press, New York (2001)
2. Abadi, M., Gordon, A.D.: A Calculus for Cryptographic Protocols: The SPi Calculus. Inf. Comput. 148(1), 1–70 (1999)
3. Blanchet, B.: An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In: 14th IEEE Computer Security Foundations Workshop (CSFW-14), Cape Breton, Nova Scotia, Canada, June 2001, pp. 82–96. IEEE Computer Society, Los Alamitos (2001)
4. Buchholtz, M., Riis Nielson, H., Nielson, F.: A calculus for control flow analysis of security protocols. International Journal of Information Security 2(3), 145–167 (2004)
5. Corin, R., Etalle, S.: An improved constraint-based system for the verification of security protocols. In: Hermenegildo, M.V., Puebla, G. (eds.) SAS 2002. LNCS, vol. 2477, pp. 326–341. Springer, Heidelberg (2002)
6. Crazzolara, F., Winskel, G.: Events in security protocols. In: ACM Conference on Computer and Communications Security, pp. 96–105 (2001)
7. Dolev, D., Yao, A.C.: On the security of public key protocols. Technical report, Dept. of Computer Science, Stanford University, Stanford, CA, USA (1981)
8. Fiore, M., Abadi, M.: Computing symbolic models for verifying cryptographic protocols. In: Proc. 14th IEEE Computer Security Foundations Workshop, pp. 160–173 (2001)

9. López, H.A., Pérez, J.A., Palamidessi, C., Rueda, C., Valencia, F.D.: A declarative framework for security: Secure concurrent constraint programming. In: Etalle, S., Truszczyński, M. (eds.) ICLP 2006. LNCS, vol. 4079, pp. 449–450. Springer, Heidelberg (2006)
10. Lowe, G.: An attack on the needham-schroeder public-key authentication protocol. Inf. Process. Lett. 56(3), 131–133 (1995)
11. Miller, D.: Encryption as an abstract data type: An Extended Abstract. In: Foundations of Computer Security (FCS). Electronic Notes in Theoretical Computer Science, vol. 84, pp. 3–15. Springer, Heidelberg (2003)
12. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes, parts I and II. Journal of Information and Computation 100, 1–77 (1992)
13. Olarte, C.A., Valencia, F.D.: The expressivity of universal timed ccp. In: 10th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming, Valencia, Spain, July 2008. ACM Press, New York (2008)
14. Olarte, C.A., Valencia, F.D.: Universal concurrent constraint programming: Symbolic semantics and applications to security. In: 23rd Annual ACM Symposium on Applied Computing (2008)
15. Palamidessi, C., Saraswat, V., Valencia, F., Victor, B.: On the Expressiveness of Linearity vs Persistence in the Asychronous Pi-Calculus. In: Proceedings of the 21st Annual IEEE Symposium on Logic in Computer Science, pp. 59–68. IEEE Computer Society, Washington (2006)
16. Saraswat, V., Jagadeesan, R., Gupta, V.: Foundations of timed concurrent constraint programming. In: Proceedings, Symposium on Logic in Computer Science, 1994. LICS 1994, pp. 71–80 (1994)

Logic Programming with Defaults and Argumentation Theories*

Hui Wan[1], Benjamin Grosof[2], Michael Kifer[1], Paul Fodor[1], and Senlin Liang[1]

[1] State University of New York at Stony Brook, USA
[2] Vulcan, Inc., USA

Abstract. We define logic programs with defaults and argumentation theories, a new framework that unifies most of the earlier proposals for defeasible reasoning in logic programming. We present a model-theoretic semantics and study its reducibility and well-behavior properties. We use the framework as an elegant and flexible foundation to extend and improve upon Generalized Courteous Logic Programs (GCLP) [19]—one of the popular forms of defeasible reasoning. The extensions include higher-order and object-oriented features of Hilog and F-Logic [7,21]. The improvements include much simpler, incremental reasoning algorithms and more intuitive behavior. The framework and its Courteous family instantiation were implemented as an extension to the FLORA-2 system.

Keywords: Defeasible reasoning, argumentation theory, well-founded models.

1 Introduction

Common-sense reasoning is one of the most important applications of logic programming. Surprisingly, the ability to do such reasoning within logic programming comes from a single, relatively simple device: default negation [8,15,16]. While this parsimony is convenient for theoretical studies, it is a major stumbling block for practical use of logic programming in common-sense reasoning: default negation is too low-level a concept to be safely entrusted to a knowledge engineer who is not a trained logician. Anecdotal evidence suggests that logicians are also not doing much better when it comes to modeling concrete application domains using default negation as a sole tool. These problems have been stimulating search for higher-abstraction forms of logic programing, which equip the knowledge engineer with frameworks and tools that are closer to the way humans tend to think of and describe the various application domains. These frameworks include object-oriented and higher-order concepts [21,7], inheritance and exceptions [32,22,34], and defeasible reasoning [3,5,6,10,11,12,17,19,25,27,31,33,35].

Defeasible reasoning in logic programming (LP) has been successfully used to model a broad range of application domains and tasks, including: policies, regulations, and law; actions, change, and process causality; Web services; aspects

* This work is part of the SILK (Semantic Inference on Large Knowledge) project sponsored by Vulcan, Inc.

of inductive/scientific learning and natural language understanding. However, there has been a bewildering multitude of formal approaches to defeasibility based on a wide variety of intuitions about desired behavior and conceptualization. The difficulties in agreeing on what is the "right" intuition are discussed in [17,6] among others. On top of this, the formal machinery employed by the different approaches is so diverse that there is little hope that more than a tiny subset of the approaches could be directly integrated in a practical, scalable reasoning system. It is also often unclear how to extend most of these approaches to include other popular LP frameworks, such as HiLog [7] and F-logic [21].

The present paper addresses these issues. First, we introduce a new general framework for defeasible reasoning that abstracts the intuitions about defeasibility into what we call *argumentation theories.* Then we develop a simple semantics for this framework and study its properties. The semantics is based on well-founded models [15]; due to space limitations, stable models [16] will be defined in an extended version of this paper. The relationship of this framework to other proposals is discussed in Section 5. The short story is that, based on our analysis, almost all approaches to defeasible reasoning in LP that we are aware of can be simulated in our framework with a suitable choice of an argumentation theory. This makes it possible to use different such theories in one reasoning system. Second, we develop a family of useful argumentation theories one of which captures the essence of *Generalized Courteous Logic Programs* (GCLP) [19]. This formulation provides a foundation to straightforwardly extend GCLP from predicate calculus-based LP to HiLog [7] and F-Logic [21], and also to improve upon GCLP's behavior and algorithms in several other significant ways, as detailed in Section 4. The usefulness of the HiLog and F-Logic features is well recognized in the literature and industry, e.g., for meta-reasoning; knowledge base translation and integration; modeling of agent's beliefs, context, and modality; knowledge provenance and navigational meta-data; and Semantic Web data models. Third, we have implemented our framework and several GCLP-style argumentation theories as an extension to FLORA-2.[1] Adding other such theories is straightforward.[2]

The rest of this paper is organized as follows. Section 2 illustrates GCLP-style defeasible reasoning with an example. Section 3 introduces our framework for defeasible reasoning with argumentation theories. Section 4 presents a family of argumentation theories, which extend and improve GCLP in several significant ways. Section 5 discusses related work, and Section 6 concludes the paper.

2 Motivating Example

The following example illustrates the basic ideas of defeasible reasoning. It uses a stripped-down syntax of FLORA-2 and models part of a game, complete with frame axioms, where blocks are moved from square to square on a board.

[1] `http://flora.sourceforge.net`

[2] FLORA-2 supports only argumentation theories with the well-founded semantics.

```
// moving blk  from->to, if to is free; from becomes free
@move loc(?s+1,?blk,?to) ∧ neg loc(?s+1,?blk,?from) :-
            move(?s,?blk,?from,?to), loc(?s,?blk,?from), not loc(?s,?,?to).
@frax1 loc(?s+1,?blk,?pos) :- loc(?s,?blk,?pos).         // frame axiom 1
@frax2 neg loc(?s+1,?blk,?pos) :- neg loc(?s,?blk,?pos). // frame axiom 2
@dloc  neg loc(?s,?blk,?pos).      // each location is free, by default
@state loc(0,block4,square7).      // initial state
// no block can be in two places at once
opposes(loc(?s,?blk,?y),loc(?s,?blk,?z)) :- posn(?y), posn(?z), ?y != ?z.
// move-action beats frame axioms; move & init state beats default location
overrides(move,frax1).   overrides(move,dloc).
overrides(move,frax2).   overrides(frax1,dloc).  overrides(state,dloc).
// facts
move(2,block4,square7,square3). // State 2: block4 moves square7->square3
posn(square1). posn(square2). ... ... ...  posn(square16).
```

The example illustrates the Courteous flavor [19] of defeasible reasoning. Here, some rules are labeled (e.g., `@move`, `@frax1`), and the predicate **overrides** specifies that some rules (e.g., the ones labeled `@move`) defeat others (e.g., the ones labeled `@frax1` and `@frax2`). We distinguish between the classical-logic-like *explicit negation*, `neg`, and default negation `not`. Literals L and `neg` L are incompatible and cannot both appear in a consistent model. The predicate **opposes** specifies additional incompatibilities. In our example, **opposes** says that no block can be present in two different positions on the board in the same state.

We can now see how defeasible reasoning works. The rule labeled `@dloc` is a "catch-case" low-priority default that says that all locations on the board are free. Contrary to this default, the fact labeled `@state` says that `block4` is at `square7` in state 0. This "defeats" the `@dloc` rule (due to **overrides**(`state,dloc`)), so `block4` is indeed at `square7`. Other squares are free unless the "catch-all" default `@dloc` is overridden. Such overriding can occur due to the `@move` rule, which specifies the effects of the move action. The `@move` rule also defeats the frame persistence axioms, `@frax1` and `@frax2`, which simply state that block locations persist from one state to another. Thus, in states 1 and 2 our block is still at `square7` and other squares are free. However, at state 2 a move occurs, and the `@move` rule derives that `block4` must be at `square3` in state 3. Due to the priorities specified via the predicate **overrides**, this latter derivation defeats the derivations produced by the frame axioms and the default location fact `@dloc`.

3 Defeasible Reasoning with Argumentation Theories

Let $\mathcal{L}$ be a logic language with the usual connectives $\wedge$ for conjunction and `:-` for rule implication; and two negation operators: `neg`, for explicit negation, and `not` for default negation. The alphabet of the language consists of: an infinite set of variables, which are shown in the examples as alphanumeric symbols prefixed with the question mark ?; and a set of constant symbols, which can appear as individuals, function symbols, and predicates. Constants will be shown as alphanumeric symbols that are not prefixed with a "?".

We use the standard notion of *terms* in logic programming. ***Atomic formulas***, also called ***atoms***, are quite general in form: they can be the atoms used in ordinary logic programming; or the higher-order expressions of HiLog [7]; or the frames of F-logic [21]. A ***literal*** has one of the following forms:

- An atomic formula.
- $\mathtt{neg}\,A$, where A is an atomic formula.
- $\mathtt{not}\,A$, where A is an atom.
- $\mathtt{not}\,\mathtt{neg}\,A$, where A is an atom.
- $\mathtt{not}\,\mathtt{not}\,L$ and $\mathtt{neg}\,\mathtt{neg}\,L$, where L is a literal; these are identified with L.

Let A denote an atom. Literals of the form A or $\mathtt{neg}\,A$ (or literals that reduce to these forms after elimination of double negation) are called **`not`-free literals**; literals that reduce to the form $\mathtt{not}\,A$ are called **`not`-literals**.

Definition 1. *A **plain** rule in a logic language $\mathcal{L}$ is an expression of the form*

$$L :\!-\, Body \qquad (1)$$

*where L, called the **head** of the rule, is a `not`-free literal in $\mathcal{L}$, and $Body$, called the **body** of the rule, is a conjunction of literals in $\mathcal{L}$.*[3] *As is common in logic programming, we will often write A, B to represent the conjunction $A \wedge B$.*
*A **labeled rule** in $\mathcal{L}$ is an expression of the form $@r\ \rho$, where ρ is a plain rule and r is a term, called the **label** of the rule. Thus, labeled rules have the form*

$$@r\ L :\!-\, Body \qquad (2)$$

*A rule label is not a rule identifier: several rules can have the same label. A **formula** is a literal, a conjunction of literals, or a rule. Given a rule of the form (2), the term*

$$\mathtt{handle}(r, L) \qquad (3)$$

*is called the **handle** for that rule. Here `handle` is a binary function symbol specifically reserved for representing rule handles. However, we do not make further assumptions about this symbol.* □

A **logic program with defaults and argumentation theories** (an `lpda`, for short) in a logic language $\mathcal{L}$ is a set of labeled and plain rules in $\mathcal{L}$.

In our theory, plain rules are considered to be definite statements about the real world. In contrast, labeled rules are *defeasible defaults*: some (or even all) instances of such rules can be "defeated" by other labeled rules in which case inferences produced by the defeated rules might be "overridden" or "canceled."

We will be often using variable-free expressions, which we call **ground**. Thus, a **ground term** is a term that contains no variables, a **ground literal** is a variable-free literal, and a **ground rule** is a rule that has no variables.

[3] This is easy to generalize to allow Lloyd-Topor extensions [23].

Lpdas are used in conjunction with *argumentation theories*. An argumentation theory is a set of rules that defines conditions under which some rule instances may be defeated or canceled by other rules.

Definition 2. *Let $\mathcal{L}$ be a logic language. An* ***argumentation theory*** *is a set, AT, of* plain *rules in $\mathcal{L}$ of the form (1). We also assume that the language $\mathcal{L}$ includes a unary predicate,* $\mathbf{defeated}_{AT}$*, which may appear in the heads of some rules in AT.*[4] *When confusion does not arise, we will omit the subscript AT.*

An `lpda` *$\mathcal{P}$ is said to be* ***compatible*** *with AT if* $\$\mathbf{defeated}_{AT}$ *does not appear in the rule heads in $\mathcal{P}$. It is often useful to consider stronger compatibility requirements, which impose additional syntactic restrictions on $\mathcal{P}$. If C_{AT} is such a compatibility requirement then we will speak of* ***C_{AT}-compatible*** `lpda`*s, i.e.,* `lpda`*s that are compatible with AT and satisfy the condition C_{AT}.* □

Thus, an argumentation theory is an ordinary logic program whose rules are not labeled. The rules AT will normally contain other predicates, besides $\$\mathbf{defeated}_{AT}$, that are used to specify how the rules in $\mathcal{P}$ get defeated. For instance, the argumentation theories described in Section 4 include the binary predicates **opposes** and **overrides**. In our FLORA-2-based implementation, argumentation theories are *meta-programs*, as in Section 4, encoded using HiLog [7]; we anticipate this would be common for other implementations of ***lpda***s as well. For the purpose of defining the semantics, we assume that the argumentation theories are ground. A grounded version of AT with respect to a compatible ***lpda*** $\mathcal{P}$ is obtained by appropriately instantiating the variables and meta-predicates in AT. For instance, for the theories in Section 4 this means (i) replacing the variables $?R$ with ground rule handles (see Definition 1) followed by (ii) replacing the meta-statement $\mathbf{body}(?R, ?B), \mathbf{call}(?B)$ in rule (12) with bodies of the rules in $\mathcal{P}$ that have $?R$ as the handle ($\mathcal{P}$ may have more than one rule with the same handle).

Note that the definitions and the subsequent theory permit different subsets of the overall ***lpda*** to have different argumentation theories AT with different $\$\mathbf{defeated}_{AT}$ predicates.[5]

Definition 3. *Let $\mathcal{P}$ be an* `lpda` *and AT an argumentation theory over language $\mathcal{L}$.*

– *The* ***Herbrand Universe*** *of $\mathcal{P}$, denoted $\mathcal{U}_{\mathcal{L}}$, is the set of all ground terms built using the constants and function symbols that appear in $\mathcal{L}$. When confusion does not arise, we will simply write $\mathcal{U}$.*
– *The* ***Herbrand Base*** *of $\mathcal{P}$, denoted $\mathcal{B}_{\mathcal{L}}$ (or simply $\mathcal{B}$, when no ambiguity arises), is the set of all ground* `not` *-free literals that can be constructed using the predicates in $\mathcal{L}$.* □

[4] We say "may" for the sake of generality. If $\$\mathbf{defeated}$ does not occur in the head of any rule then the semantics of ***lpda***s reduce to ordinary logic programming.

[5] Our FLORA-2 extension also supports multiple argumentation theories.

3.1 The Well-Founded Semantics

In this section, we extend the well-founded semantics for default negation [15] to *lpda*s. Our development follows the general outline in [29]. The full version of this paper also provides the stable model semantics [16].

The following definition of partial interpretations is essentially from [15,29]. First, we assume that the language includes three special propositional constants, $\mathbf{t}$, $\mathbf{f}$, and $\mathbf{u}$, which stand for *true*, *false*, and *undefined*, respectively. We also assume the existence of the following total order on these propositions: $\mathbf{f} < \mathbf{u} < \mathbf{t}$.

Definition 4. *A* ***partial Herbrand interpretation****,* $\boldsymbol{I}$*, is simply a set of ground literals. In addition:* $\boldsymbol{I}$ *must contain both* $\mathtt{t}$ *and* $\mathtt{not}\,\mathbf{f}$*; it may contain neither* $\mathbf{u}$ *nor* $\mathtt{not}\,\mathbf{u}$*; and* L*,* $\mathtt{not}\,L$ *cannot both be in* $\boldsymbol{I}$*, for any literal* L*.*

An interpretation is ***inconsistent relative to*** *an atom* A *if both* A *and* $\mathtt{neg}\,A$ *belong to* $\boldsymbol{I}$*. Otherwise,* $\boldsymbol{I}$ *is* ***consistent relative to*** A*. An interpretation is* ***consistent*** *if it is consistent relative to every atom and* ***inconsistent*** *if it is inconsistent relative to some atom. An interpretation is* ***total*** *if, for every ground* $\mathtt{not}$ *-free literal* L *(except* $\mathbf{u}$*), either* L *or* $\mathtt{not}\,L$ *belongs to* $\boldsymbol{I}$*.*

We also define $\boldsymbol{I}^+ = \{L \mid L \in \boldsymbol{I} \text{ is a } \mathtt{not}\text{-free literal}\}$ *and* $\boldsymbol{I}^- = \{L \mid L \in \boldsymbol{I} \text{ is a } \mathtt{not}\text{-literal}\}$*. Thus,* $\boldsymbol{I} = \boldsymbol{I}^+ \cup \boldsymbol{I}^-$*.* □

Models. Next we define satisfaction for ground formulas and *lpda*s.

Definition 5. *Let* $\boldsymbol{I}$ *be a partial Herbrand interpretation,* L *a ground* $\mathtt{not}$ *-free literal, and* F*,* G *ground formulas. Then* $\boldsymbol{I}$ *maps formulas to* $\{\mathbf{t},\mathbf{f},\mathbf{u}\}$ *as follows:*

- *If* L *is a* $\mathtt{not}$ *-free literal then* $\boldsymbol{I}(L) = \mathbf{t}$ *iff* $L \in \boldsymbol{I}$*,* $\boldsymbol{I}(L) = \mathbf{f}$ *iff* $\mathtt{not}\,L \in \boldsymbol{I}$*, and* $\boldsymbol{I}(L) = \mathbf{u}$*, otherwise.*
- $\boldsymbol{I}(\mathtt{not}\,L) = \sim \boldsymbol{I}(L)$*, where* $\sim \mathbf{t} = \mathbf{f}$*,* $\sim \mathbf{f} = \mathbf{t}$*, and* $\sim \mathbf{u} = \mathbf{u}$*.*
  Note that the above two items together with Definition 4 imply that $\boldsymbol{I}(\mathbf{t}) = \mathbf{t}$*,* $\boldsymbol{I}(\mathtt{not}\,\mathbf{t}) = \mathbf{f}$*;* $\boldsymbol{I}(\mathbf{f}) = \mathbf{f}$*,* $\boldsymbol{I}(\mathtt{not}\,\mathbf{f}) = \mathbf{t}$*; and* $\boldsymbol{I}(\mathbf{u}) = \boldsymbol{I}(\mathtt{not}\,\mathbf{u}) = \mathbf{u}$*.*
- $\boldsymbol{I}(F \wedge G) = \mathtt{min}(\boldsymbol{I}(F), \boldsymbol{I}(G))$*.*
- *For a plain rule* F :- G*, define* $\boldsymbol{I}(F$:- $G) = \mathbf{t}$ *if and only if* $\boldsymbol{I}(F) \geq \boldsymbol{I}(G)$*.*
- *For a labeled rule* @r F :- G*, we define* $\boldsymbol{I}(@r\ F$:- $G) = \mathbf{t}$ *if and only if* $\boldsymbol{I}(F) \geq \mathtt{min}(\boldsymbol{I}(G), \boldsymbol{I}(\mathtt{not}\,\$\mathbf{defeated}(\mathtt{handle}(r,F))))$*.*
  Here $\mathtt{handle}(r,F)$ *is the handle (Definition 1) for the rule* @r F :- G*.* □

Definition 6. *If* $\boldsymbol{I}(F) = \boldsymbol{t}$*, where* $\boldsymbol{I}$ *is an interpretation, then we write* $\boldsymbol{I} \models F$ *and say that* $\boldsymbol{I}$ *is a* ***model*** *of* F *(or that* ***satisfies*** F*). An interpretation* $\boldsymbol{I}$ *is a model of an* `lpda` $\mathcal{P}$ *if* $\boldsymbol{I} \models R$ *for every* $R \in \mathcal{P}$*.* □

Definition 7. *Given an* `lpda` $\mathcal{P}$*, an argumentation theory* AT*, and an interpretation* $\boldsymbol{M}$*, we say that* $\boldsymbol{M}$ *is a model of* $\mathcal{P}$ *with respect to the argumentation theory* AT*, written as* $\boldsymbol{M} \models (\mathcal{P}, AT)$*, if* $\boldsymbol{M} \models \mathcal{P}$ *and* $\boldsymbol{M} \models AT$*.* □

Definition 8. *Suppose that* $\boldsymbol{M}_1$*, and* $\boldsymbol{M}_2$ *are interpretations. We define* $\boldsymbol{M}_1 \preceq \boldsymbol{M}_2$ if $\boldsymbol{M}_1^+ \subseteq \boldsymbol{M}_2^+$ and $\boldsymbol{M}_1^- \supseteq \boldsymbol{M}_2^-$*. The minimal models with respect to* $\preceq$ *are called the* ***least*** *models of* $(\mathcal{P}, AT)$*.* □

Well-Founded Models. We now define a special kind of models for *lpda*s, called the well-founded models. These models are first defined as limits of transfinite sequences of partial interpretations and then we show that they correspond to the ordinary well-founded models of certain normal logic programs that are obtained by a transformation from *lpda*s.

The quotient operator is defined analogously to [29], but with changes to adapt this concept to logic programs with defaults and argumentation theories.

Definition 9. *Let $\mathcal{Q}$ be a set of rules, which can include labeled as well as plain rules, and let $\boldsymbol{J}$ be a partial Herbrand interpretation for $\mathcal{Q}$. We define the* `lpda` ***quotient of** $\mathcal{Q}$ **by** $\boldsymbol{J}$, written as $\dfrac{\mathcal{Q}}{\boldsymbol{J}}$, by the following sequence of steps:*

1. *Replace every* `not` *-literal in the body of $\mathcal{Q}$ by its truth value in $\boldsymbol{J}$.*
2. *Replace every labeled rule of the form $@r\ L$:- $Body$ in $\mathcal{Q}$, such that $\boldsymbol{J}(\$\mathbf{defeated}(\mathtt{handle}(r, L))) = \mathbf{t}$ with the rule L :- $Body$, $\mathbf{f}$.*
   (Rule handles were introduced in Definition 1.)
3. *Replace every labeled rule ($@r\ L$:- $Body$) $\in \mathcal{Q}$ such that $\boldsymbol{J}(\$\mathbf{defeated}(\mathtt{handle}(r, L))) = \mathbf{u}$ with the rule L :- $Body$, $\mathbf{u}$.*
4. *Remove all labels from the remaining labeled rules.*

The resulting set of rules is the `lpda` *quotient $\dfrac{\mathcal{Q}}{\boldsymbol{J}}$.* □

In the next definition, $LPM(Q)$ denotes the least partial model of a `not`-free *lpda* Q. As in [29], $LPM(Q)$ is computed iteratively, by making all possible derivations using the rules in Q starting with the empty partial interpretation.

Definition 10. *The **well-founded model** of an* `lpda` *$\mathcal{P}$ with respect to the argumentation theory AT, written as $WFM(\mathcal{P}, AT)$, is defined as the limit of the following transfinite induction. Initially, $\boldsymbol{I}_0$ is the empty set. Suppose $\boldsymbol{I}_m$ has already been defined for every $m < k$, where k is an ordinal. Then:*

– $\boldsymbol{I}_k = LPM(\dfrac{\mathcal{P} \cup AT}{\boldsymbol{I}_{k-1}})$, *if k is a non-limit ordinal.*
– $\boldsymbol{I}_k = \cup_{i<k} \boldsymbol{I}_i$, *if k is a limit ordinal.* □

According to the next theorem, this limit exists. The theorem also shows that *lpda*s reduce to and can be implemented using ordinary logic programming systems that support the well-founded semantics (e.g., XSB).

Theorem 1 (Reduction). *The transfinite sequence $\langle \boldsymbol{I}_0, \boldsymbol{I}_1, \ldots \rangle$ of interpretations in Definition 10 has a (unique) limit. It is reached for some (possibly transfinite) ordinal, α, such that $\boldsymbol{I}_\alpha = \boldsymbol{I}_{\alpha+1}$. This limit, $WFM(\mathcal{P}, AT)$, is a least model of $(\mathcal{P}, AT)$. Furthermore, $WFM(\mathcal{P}, AT)$ coincides with the well-founded model of the ordinary logic program $\mathcal{P}' \cup AT$, where $\mathcal{P}'$ is obtained from $\mathcal{P}$ by changing every labeled rule* (`@r L :-Body`) $\in \mathcal{P}$ *to the plain rule* `L :-Body, not` $\$\mathbf{defeated}$(`handle(r, L)`). □

3.2 Well-Behaved Argumentation Theories

So far, argumentation theories were defined in a very general way. However, not all such theories are practically useful. This section introduces a number of *well-behavior properties* that are useful for argumentation theories to abide. These conditions involve both the argumentation theories themselves and their associated compatibility requirements (see Definition 2).

Definition 11. *An argumentation theory AT with a compatibility requirement C_{AT} **ensures consistency** relative to an atom A if for every C_{AT}-compatible* `lpda` *$\mathcal{P}$, the well-founded model of ($\mathcal{P}$,AT) is consistent relative to A (see Definition 4). We say that (AT,C_{AT}) ensures consistency, if it ensures consistency for all atoms.* □

Definition 12. *Consider an argumentation theory AT with a compatibility requirement C_{AT}. Let us further assume that AT uses a binary predicate* **opposes***, which is defined on rule handles. Literals L_1 and L_2 are said to **oppose each other** in a partial interpretation $\boldsymbol{I}$ of an* `lpda` *$\mathcal{P}$ iff* **opposes**(`handle`(r_1, L_1), `handle`(r_2, L_2)) *is true in $\boldsymbol{I}$ for all pairs of rules of the form @r_1 L_1 :- ⋯ and @r_2 L_2 :- ⋯ in $\mathcal{P}$ (i.e., rules having L_1 and L_2 in the head).*

*We say that AT **ensures strong consistency** if, for every C_{AT}-compatible* `lpda` *$\mathcal{P}$, the well-founded model $\boldsymbol{M}$ of ($\mathcal{P}$,AT) has the following property:*

> *If any pair of literals, L_1 and L_2, oppose each other in $\boldsymbol{M}$, then L_1 and L_2 cannot both be true in $\boldsymbol{M}$.*

□

Definition 13. *Consider an argumentation theory AT with a compatibility condition C_{AT}. Let us further assume that AT uses two binary predicates,* **overrides** *and* **opposes***, whose arguments are rule handles. We say that AT has the **overriding property** if, for every C_{AT}-compatible* `lpda` *$\mathcal{P}$, the following rule is true in the well-founded model of ($\mathcal{P}$, AT):*

$$\begin{aligned}\$\mathbf{defeated}(\texttt{handle}(r_2, L_2)) \texttt{ :- } & \mathit{Body}_1 \wedge \mathit{Body}_2 \\ & \wedge \mathbf{overrides}(\texttt{handle}(r_1, L_1), \texttt{handle}(r_2, L_2)) \\ & \wedge \mathbf{opposes}(\texttt{handle}(r_1, L_1), \texttt{handle}(r_2, L_2)) \\ & \wedge \texttt{not}\ \$\mathbf{defeated}(\texttt{handle}(r_1, L_1))\end{aligned} \tag{4}$$

for all pairs of rules of the form (@r_i L_i :- Body_i) $\in \mathcal{P}$, $i = 1, 2$. □

Next we develop a family of argumentation theories that obeys these properties.

4 Courteous Argumentation Theories

We now develop a family of particularly interesting argumentation theories, denoted $\mathcal{AT}^{\mathcal{C}}$, which subsumes *generalized courteous logic programs* (GCLP) [19]. Some members of this family correspond to different earlier versions of courteous logic programs [18,19]; others improve upon these previous versions by eliminating certain cases of controversial behavior. The properties of these argumentation theories are discussed at the end of this section.

Apart from the standard predicate \$**defeated**, the argumentation theories in the $\mathcal{AT}^{\mathcal{C}}$ family use two other predicates: **opposes** and **overrides**, which are normally defined by the user. Argumentation theories might include additional axioms, such as symmetry for **opposes** or transitivity for **overrides**. The **opposes** and **overrides** predicates are expected to be specified over rule handles; they may occur as facts and in the heads of rules. Other predicates in $\mathcal{AT}^{\mathcal{C}}$ represent concepts used to argue that certain rules must or must not be defeated. The variables $?R$ and $?S$ are expected to range over rule handles.

Definition of defeasibility. These rules define what it means for a rule to be defeated or to defeat another rule. A rule is *defeated* if it is *refuted* or *rebutted* by some other rule, provided that the latter rule is not *compromised*. A rule can also be defeated if it is *disqualified* for some other reason.

$$\begin{array}{lll} \$\mathbf{defeated}(?R) & \texttt{:-} & \$\texttt{defeats}(?S,?R),\ \texttt{not}\,\$\texttt{compromised}(?S). \\ \$\mathbf{defeated}(?R) & \texttt{:-} & \$\texttt{disqualified}(?R). \\ \$\texttt{defeats}(?R,?S) & \texttt{:-} & \$\texttt{refutes}(?R,?S)\ \mathbf{or}\ \$\texttt{rebuts}(?R,?S). \end{array} \tag{5}$$

The predicates `$refutes` and `$rebuts` will be defined shortly. The predicates `$compromised` and `$disqualified` can mean different things depending on the intended theory of argumentation. Here are some of the possibilities:

– No rule is compromised or disqualified. *Lpdas* with this argumentation theory are equivalent to the original courteous logic programs (GCLP).

$$\begin{array}{l} \$\texttt{compromised}(?X)\ \texttt{:-}\ \mathbf{false}. \\ \$\texttt{disqualified}(?X)\ \texttt{:-}\ \mathbf{false}. \end{array} \tag{6}$$

– A rule is compromised if it is defeated, and it is disqualified if it transitively defeats itself.[6] This choice has been the main one we have experimented with recently for practical use cases using our FLORA-2 extension.

$$\begin{array}{l} \$\texttt{compromised}(?R)\ \texttt{:-}\ \$\texttt{refuted}(?R), \$\mathbf{defeated}(?R). \\ \$\texttt{disqualified}(?X)\ \texttt{:-}\ \$\texttt{defeats}^*(?X,?X). \end{array} \tag{7}$$

Here `$defeats`$*$ denotes the transitive closure of `$defeats`.

– Another reasonable choice is

$$\begin{array}{l} \$\texttt{compromised}(?R)\ \texttt{:-}\ \$\mathbf{defeated}(?R). \\ \$\texttt{disqualified}(?X)\ \texttt{:-}\ \$\texttt{defeats}^*(?X,?X). \end{array} \tag{8}$$

Definitions for `$refutes` *and* `$rebuts`. *Refutation* of a rule, `r`, means that a higher-priority rule implies a conclusion that is incompatible with the conclusion implied by `r`. It is defined as follows:

$$\begin{array}{l} \$\texttt{refutes}(?R,?S)\ \texttt{:-}\ \$\texttt{conflict}(?R,?S),\ \mathbf{overrides}(?R,?S). \\ \$\texttt{refuted}(?R)\ \texttt{:-}\ \$\texttt{refutes}(?R2,?R). \end{array} \tag{9}$$

[6] Note that we do not require that the predicate `$defeats` is transitively closed.

Rebuttal means that a pair of rules assert conflicting conclusions, but neither derivation can be discarded or considered "more important" than the other. This intuition can be expressed in several different (not necessarily equivalent) ways. We have been experimenting with the following definitions (where (11) together with (6) defines the original GCLP):

$$\begin{aligned} \texttt{\$rebuts}(?R,?S) \ \texttt{:-}\ & \texttt{\$conflict}(?R,?S), \texttt{not}\,\texttt{\$compromised}(?R). && (10)\\ \texttt{\$rebuts}(?R,?S) \ \texttt{:-}\ & \texttt{\$conflict}(?R,?S), \texttt{not}\,\texttt{\$compromised}(?R), && (11)\\ & \texttt{not}\,\texttt{\$refuted}(?R), \texttt{not}\,\texttt{\$refuted}(?S). \end{aligned}$$

Definition of candidacy and conflict. A *candidate* rule-instance is one whose body is true in the knowledge base:

$$\texttt{\$candidate}(?R) \ \texttt{:-}\ \mathbf{body}(?R,?B),\ \mathbf{call}(?B). \qquad (12)$$

Here **body** is a meta-predicate that binds $?B$ to the body of a rule with handle $?R$. The **call** meta-predicate takes the body of a rule and poses it as a query to the knowledge base. We note that these meta-predicates can be represented as object-level predicates in HiLog [7]. We omit reviewing here the main aspects of HiLog for reasons of space and focus.

Conflicting rules are now defined as follows: two rule handles are in conflict if they are both candidates and are in opposition to each other.

$$\texttt{\$conflict}(?R,?S)\texttt{:-}\texttt{\$candidate}(?R), \texttt{\$candidate}(?S), \mathbf{opposes}(?R,?S). \quad (13)$$

Background theory for mutual exclusion. The predicate **opposes** is normally defined within the user knowledge base by a set of facts and rules. In addition, our argumentation theories require **opposes** to be symmetric and such that every literal must oppose its explicit negation (`neg`):

$$\begin{aligned} &\mathbf{opposes}(?R,?S)\ \texttt{:-}\ \mathbf{opposes}(?S,?R).\\ &\mathbf{opposes}(\texttt{handle}(?L1,?H),\ \texttt{handle}(?L2,\texttt{neg}\ ?H)). \end{aligned} \qquad (14)$$

We say that an argumentation theory ***belongs to the*** $\mathcal{AT}^{\mathcal{C}}$ ***family*** if it includes the rules (5), (9), and (12)–(14); plus either (6) or (7) or (8); and either (10) or (11). Let AT be an argumentation theory in $\mathcal{AT}^{\mathcal{C}}$ and let the compatibility requirement be as follows. An ***lpda*** $\mathcal{P}$ is compatible with AT iff:

- The set of the atoms that appear in the heads of plain (non-defeasible) rules and in the heads of labeled (defeasible) rules in $\mathcal{P}$ are disjoint.
- The \$-predicates defined by $\mathcal{AT}^{\mathcal{C}}$ (\$**defeated**, `$compromised`, `$refuted`, etc.) do not occur in the heads of the program rules (i.e., they are defined only by the rules in AT).

Theorem 2 (Well-behavior). *Let AT be an argumentation theory in $\mathcal{AT}^{\mathcal{C}}$ with the above compatibility requirement. Then AT satisfies the properties of well-behaved theories of Section 3.2; namely:*

1. *AT ensures consistency for the atoms that occur in the heads of labeled rules.*
2. *Suppose there is no literal A for which both A and* `neg` *A appear in the heads of plain rules in $\mathcal{P}$. Then AT ensures consistency (for all atoms).*
3. *If* **opposes**(`handle`(..., L_1), `handle`(..., L_2)) *is true only when neither L_1 nor L_2 occurs in the heads of plain rules, then AT ensures strong consistency.*
4. *AT has the overriding property.* □

Consider an argumentation theory, denoted AT^{GCLP}, that consists of the rules (5) – (6), (9), (11) – (14) and the following compatibility requirement. An *lpda* $\mathcal{P}$ is compatible with AT^{GCLP} iff:

– $\mathcal{P}$ contains only labeled rules.
– The \$-predicates defined by AT^{GCLP} (\$**defeated**, \$`refuted`, etc.) do not occur in the heads of the program rules.

Theorem 3 (GCLP as LPDA). *Consider AT^{GCLP} and a compatible* `lpda` *$\mathcal{P}$. Let $\mathcal{P}'$ be the program obtained from $\mathcal{P}$ using the GCLP transformation of [19].*[7] *Then the restrictions of the well-founded models of $(\mathcal{P}, AT^{GCLP})$ and of $\mathcal{P}'$ to the predicates mentioned in $\mathcal{P}$ coincide.* □

This result says that the original GCLP is essentially equivalent to LPDA with the argumentation theory AT^{GCLP}. The new formulation of GCLP has many benefits. First, it is not limited to ordinary logic programs: it extends straightforwardly to HiLog [7], F-logic [21], and other forms of logic programming. Second, *lpda*s are inherently incremental: adding new knowledge does not require changes to the already existing knowledge. In contrast, in the original approach, adding new rules or facts meant that the GCLP transformation had to be re-applied from scratch. It was substantial effort to find an equivalent incremental transformation [13]. Third, the new formulation generalizes GCLP by allowing non-defeasible rules.

Also, importantly, the new framework lets us use different argumentation theories, while the original approach had one or two built into fairly complex transformations, often making it hard to see through the complexity and to experiment. In contrast, the new approach separates the argumentation theory from program transformation, makes it much easier to see the rationale behind the different parts of the argumentation theories, greatly simplifies the implementation, and enables various optimizations and improvements.

A case in point is the following example of controversial behavior exhibited by the original GCLP in an "edge case."

```
@a p.               @b q.              @c s.
overrides(handle(a,?),handle(c,?)). opposes(handle(?,p),handle(?,s)).
overrides(handle(c,?),handle(b,?)). opposes(handle(?,q),handle(?,s)).
```

Here GCLP sanctions the model {`p, not q, not s`}. However, one might feel that the intended model should instead be {`p, q, not s`} because `c` is defeated

[7] We are glossing over the minor detail that the syntax of *lpda*s is slightly different from the syntax of GCLP used in [19].

and thus should not defeat b. Modifying the argumentation theory AT^{GCLP} is much easier than examining and modifying the complex transformation of the original GCLP. The alternative intuition about desired defeasibility behavior in the above "edge case" example can be expressed by replacing the rules (6) with (7) in the argumentation theory AT^{GCLP}.

As further illustration, we show how GCLP can be used in conjunction with HiLog's higher-order syntax and F-logic's object-oriented syntax. The example also illustrates the use of rule labels with variables.

```
@perm(?t) ?p(?usr) :- ?adm[states(?t)->?p(?usr)],?adm[controls->?p].
overrides(handle(perm(?t1),?),handle(perm(?t2),?)) :- ?t1 > ?t2.
Bob[states(2008)->neg print(Al)]. Bob[states(2009)->print(Al)].
Bob[controls->{print(?), neg print(?)}].
```

Here the first rule says that if an administrator, ?adm, has stated at time ?t that the user ?usr has a privilege, ?p, and if that administrator controls this type of privileges, then the privilege is granted. Privileges can be positive (e.g., print) or negative (e.g., neg print). This rule is defeasible and its label is non-ground. The head of the rule is a HiLog literal, because of the higher-order variable ?p, while the body has a combination of F-logic and HiLog features. The second rule is non-defeasible. It says that later pronouncements override earlier ones. The facts on line 3 say that the administrator Bob has issued conflicting statements about whether the user Al is allowed to print or not. The last fact says that Bob controls the printing privilege as well as its revocation. With the AT^{GCLP} argumentation theory, the above ***lpda*** sanctions the conclusion print(Al), as expected. It is worth pointing out here that modifying the original GCLP transformation [19] to handle this kind of programs is not a trivial matter.

5 Comparison with Other Work

The last two decades saw a great number of approaches to defeasible reasoning in logic programming. Most of these are based on Reiter's Default Logic [30], stable models [16], and only a few [19,24] use the well-founded semantics [15]. None of the works surveyed here uses the notion of argumentation theories, but [17,10,12] have goals similar to ours. Due to the sheer size of the literature on defeasible reasoning, it is not feasible to do justice to all prior work in this section. Therefore, we will focus on the more closely related work and refer the reader to a recent survey [9] for a broader discussion of the literature, including the works that we were unable to mention.

General frameworks [17,10,12]. The closest, in spirit, to our work are the logic of prioritized defaults by Gelfond and Son [17], the meta-interpretation approach of [12], and ordered logic programs of Delgrande, Schaub, and Tompits [10].

The logic of prioritized defaults [17] does not use the notion of argumentation theories, but it is made clear that the meaning of the various theories of defaults may differ from one application domain to another. This is analogous to allowing argumentation theories to vary. However, Gelfond and Son developed their language as a meta-theory, whose semantics is given by meta-interpreters. What we call an "argumentation theory" is built into meta-interpreters in [17], and no independent model theory is given. In contrast, our approach distills all the differences between the different default theories to the notion of an argumentation theory with a simple interface to the user-provided domain description, the predicate $**defeated**. This allows us to define model-theoretic semantics, including the well-founded and stable models, to unify the theories of Courteous Logic Programming, Defeasible Logic, Prioritized Defaults, and more. This also allows us to focus on the development of powerful argumentation theories, which have the expected behavior on all known "benchmarks" that we are aware of from the literature. The following is one such example.

```
@d1  neg flies :- penguin.        overrides(d1,d2).        bird.
@d2  flies :- bird.               overrides(d2,d3).        swims.
@d3  penguin :- bird, swims.      overrides(d1,d3).
```

This example was discussed in [5,17] as a case where a seemingly correct domain description yields the unintended model {`swims, bird, penguin, flies`} instead of the expected model {`swims, bird, penguin, neg flies`}. Gelfond and Son [17] argue that this happens because the above domain description is "unclear" and requires a clarification in the form of the statement **opposes**(d_2, d_3). In our opinion, however, requiring such additional domain-specific particulars is undue engineering burden. Like [5], we believe that the above domain description is sufficient by itself and, together with any of the argumentation theories of Section 4, this *lpda* has the expected behavior in our framework.

Like Gelfond and Son's work, Leone et. al. [12] set out to unify approaches to defeasible reasoning. Specifically, they present an adaptable meta-interpreter, which can be made to simulate the approaches described in [6,33] among others.

Delgrande et. al. [10] propose a framework of ordered logic programming, which can use a variety of preference handling strategies. For each strategy, this approach devises a transformation from ordered logic programs to ordinary logic programs. Each transformation is custom-made for the particular preference-handling strategy, and the approach was illustrated by showing transformations for several strategies, including two described in earlier works [33,12].

Unlike our approach, Delgrande's framework does not come with a unifying model-theoretic semantics. Instead, the definition of preferred answer sets differs from one preference-handling strategy to another. One of the more important conceptual differences between our work and [10] has to do with the nature of the variable parts of the two approaches. In our case, the variable part is the argumentation theory, which is a set of definitions for concepts that a human reasoner might use to argue why certain conclusions are to be defeated. In case of [10], the variable part is the transformation, which encodes a fairly low-level mechanism: the order of rule applications required to generate the preferred answer set.[8] Finally, we note that each program transformation in [10] needs a

[8] Note that argumentation theories can also encode rule application orderings.

compiler that contains hundreds of lines of Prolog code. Our approach requires no new software, and each argumentation theory typically contains 20-30 rules.

Defeasible Logic [25]. Defeasible Logic is related to *lpda*s in a number of ways. On one hand, [1] shows that a `not`-free subset of GCLP (which is a special case of *lpda*s) can be represented as a defeasible logic theory. On the other hand, it can be shown that Defeasible Logic programs with non-contradictory strict rules can be represented as *lpda*s with suitable argumentation theories both under the well-founded semantics [24] and under the stable model semantics [2].[9] Apart from the ability to choose argumentation theories, *lpda*s generalize Defeasible Logic in other ways. For instance, Defeasible Logic does not deal with general conflicts, i.e., situations where the opposing rules have heads that are not negations of each other. In addition, *lpda*s can use the full power of rules to define the prioritization ordering, while Defeasible Logic requires that this ordering is specified in advance.

Other logics of prioritized defaults. Many other formalizations of prioritized defaults, including [3,5,6,11,31,33,34,35], have been developed over the years. In these formalisms, priorities can be assigned either to atoms (e.g., [31]) or to rules ([3,5,6] and others), and the details vary widely. For example, most proposals specify priorities explicitly, but some (e.g., [11]) assign them implicitly, via the notion of specificity (rule r_1 is more specific than r_2 if the body of r_1 entails the body of r_2). For yet others, the mechanism for implicit prioritization is instead derived from the structure of class hierarchies [34]. In most cases, these proposals use stable models instead of the well-founded models used in the present work. However, as mentioned earlier, stable models for *lpdas* can be defined and, based on our analysis, most of the above approaches can be simulated within our framework by choosing suitable argumentation theories. Only Sakama and Inoue's approach [31] bucks the trend. The key difficulty in capturing this formalism is the way in which it defines preferences over answer sets: in [31], preferences are derived from a priority relation over atoms, while all other approaches define priorities over rules only.

Argumentation theories. A significant body of work is dedicated to development of argumentation theories. These include papers like [4,14,26], which use this term in a different sense than we do and are not closely related,[10] as well as more closely related works [27,28,20]. The focus of the latter works is development of the actual concepts that argumentation theories operate with. For instance, [27] uses Default Logic [30] to formalize the notions of defeat, defensible arguments, etc. Our work is more general in the sense that we do not stick to a particular argumentation theory and our FLORA-2-based implementation makes it easy

[9] Stable models for *lpda*s will be defined in the full version of this paper.

[10] By *arguments* these works mean proofs or sets of supporting statements, not rules that define the notion of defeasibility. The focus of [4] is non-monotonic logic in general, while [14] is a procedural approach to defeasible rules. It is unclear whether this approach can be captured as an argumentation theory in our framework.

to use different such theories. However, concrete argumentation theories used in our framework might embody notions analogous to those in [27]. For instance, the theory of Section 4 uses the notions of defeated and defensible (*rebutted*, in our terminology) arguments, although those notions are not exactly the ones developed in [27].

6 Conclusions

We presented a novel approach that unifies most of the earlier work on defeasible reasoning in logic programming (LP). The primary advantages of the approach are:

– Generalization of Courteous and other previous defeasible LP approaches to include HiLog-style higher-order and F-logic style object-oriented features.
– Much simpler implementation for Courteous and other previous defeasible LP approaches. Such an implementation is easy in a system with sufficient degree of introspection, like FLORA-2: contrast 20-30 rules per argumentation theory (e.g., Section 4) versus thousands of lines of code (e.g., [19,10]).
– Unification of almost all previous defeasible LP approaches within one theory and the ability to combine multiple such in one system.
– Improvements on original GCLP, including a direct model theory, simpler and faster incremental updating, and better control over edge case behavior.

References

1. Antoniou, G., Billington, D., Governatori, G., Maher, M.J.: Representation results for defeasible logic. ACM Trans. Comput. Log. 2(2), 255–287 (2001)
2. Antoniou, G., Maher, M.J.: Embedding defeasible logic into logic programs. In: Int'l Conference on Logic Programming, pp. 393–404 (2002)
3. Baader, F., Hollunder, B.: Priorities on defaults with prerequisites, and their application in treating specificity in terminological default logic. Journal of Automated Reasoning 15(1), 41–68 (1995)
4. Bondarenko, A., Dung, P.M., Kowalski, R.A., Toni, F.: An abstract, argumentation-theoretic approach to default reasoning. AI 93(1-2), 63–101 (1997)
5. Brewka, G., Eiter, T.: Preferred answer sets for extended logic programs. Artificial Intelligence 109, 297–356 (1999)
6. Brewka, G., Eiter, T.: Prioritizing default logic. In: Intellectics and Computational Logic – Papers in Honour of Wolfgang Bibel, pp. 27–45. Kluwer Academic Publishers, Dordrecht (2000)
7. Chen, W., Kifer, M., Warren, D.S.: HiLog: A foundation for higher-order logic programming. Journal of Logic Programming 15(3), 187–230 (1993)
8. Clark, K.L.: Negation as failure. In: Gallaire, H., Minker, J. (eds.) Logic and Data Bases, pp. 292–322. Plenum Press, New York (1978)
9. Delgrande, J., Schaub, T., Tompits, H., Wang, K.: A classification and survey of preference handling approaches in nonmonotonic reasoning. Computational Intelligence 20(12), 308–334 (2004)

10. Delgrande, J.P., Schaub, T., Tompits, H.: A framework for compiling preferences in logic programs. Theory and Practice of Logic Programming 2, 129–187 (2003)
11. Dung, P.M., Son, T.C.: An argument-based approach to reasoning with specificity. Artificial Intelligence 133(1-2), 35–85 (2001)
12. Eiter, T., Faber, W., Leone, N., Pfeifer, G.: Computing preferred answer sets by meta-interpretation in answer set programming. Theory and Practice of Logic Programming 3(4), 463–498 (2003)
13. Ganjugunte, S.: Extending reasoning infrastructure for rules on the semantic web: Well-founded negation, incremental courteous logic programs, and interoperability tools in sweetrules. Master's thesis, UMBC (2005)
14. García, A.J., Simari, G.R.: Defeasible logic programming: an argumentative approach. Theory Practice of Logic Programming 4(2), 95–138 (2004)
15. Van Gelder, A., Ross, K.A., Schlipf, J.S.: The well-founded semantics for general logic programs. Journal of the ACM 38, 620–650 (1991)
16. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: Proceedings of ICLP/SLP, pp. 1070–1080. MIT Press, Cambridge (1988)
17. Gelfond, M., Son, T.C.: Reasoning with prioritized defaults. In: Dix, J., Moniz Pereira, L., Przymusinski, T.C. (eds.) LPKR 1997. LNCS, vol. 1471, pp. 164–223. Springer, Heidelberg (1998)
18. Grosof, B.N.: Prioritized conflict handling for logic programs. In: Int'l Logic Programming Symposium, October 1997, pp. 197–211 (1997)
19. Grosof, B.N.: A courteous compiler from generalized courteous logic programs to ordinary logic programs. Technical Report Supplementary Update Follow-On to RC 21472, IBM (July 1999)
20. Karacapilidis, N.I., Papadias, D., Gordon, T.F.: An argumentation based framework for defeasible and qualitative reasoning. In: XIIIth Brazilian Symposium on Artificial Intelligence, Advances in Artificial Intelligence, pp. 1–10. Springer, Heidelberg (1996)
21. Kifer, M., Lausen, G., Wu, J.: Logical foundations of object-oriented and frame-based languages. Journal of ACM 42, 741–843 (1995)
22. Lakshmanan, L.V.S., Thirunarayan, K.: Declarative frameworks for inheritance. In: Chomicki, J., Saake, G. (eds.) Logics for Databases and Information Systems, pp. 357–388. Kluwer Academic Publishers, Dordrecht (1998)
23. Lloyd, J.W.: Foundations of Logic Programming, 2nd edn. Springer, Heidelberg (1987)
24. Maier, F., Nute, D.: Relating defeasible logic to the well-founded semantics for normal logic programs. In: Int'l Workshop on Non-monotonic Reasoning (2006)
25. Nute, D.: Defeasible logic. In: Handbook of logic in artificial intelligence and logic programming, pp. 353–395. Oxford University Press, Oxford (1994)
26. Pereira, L.M., Pinto, A.M.: Reductio ad absurdum argumentation in normal logic programs. In: ArgNMR workshop at LPNMR, pp. 96–113 (2007)
27. Prakken, H.: An argumentation framework in default logic. Annals of Mathematics and Artificial Intelligence 9(1-2), 93–132 (1993)
28. Prakken, H.: A logical framework for modelling legal argument. In: ICAIL 1993: 4th Int'l Conf. on Artificial Intelligence and Law, pp. 1–9. ACM, New York (1993)
29. Przymusinski, T.C.: Well-founded and stationary models of logic programs. Annals of Mathematics and Artificial Intelligence 12, 141–187 (1994)
30. Reiter, R.: A logic for default reasoning. Artificial Intelligence 13, 81–132 (1980)
31. Sakama, C., Inoue, K.: Prioritized logic programming and its application to commonsense reasoning. Artificial Intelligence 123(1-2), 185–222 (2000)

32. Touretzky, D.S.: The Mathematics of Inheritance Systems. Morgan Kaufmann, San Francisco (1986)
33. Wang, K., Zhou, L., Lin, F.: Alternating fixpoint theory for logic programs with priority. In: Palamidessi, C., Moniz Pereira, L., Lloyd, J.W., Dahl, V., Furbach, U., Kerber, M., Lau, K.-K., Sagiv, Y., Stuckey, P.J. (eds.) CL 2000. LNCS, vol. 1861, pp. 164–178. Springer, Heidelberg (2000)
34. Yang, G., Kifer, M.: Inheritance in rule-based frame systems: Semantics and inference. Journal on Data Semantics 2800, 69–97 (2003)
35. Zhang, Y., Wu, C.M., Bai, Y.: Implementing prioritized logic programming. AI Communications 14(4), 183–196 (2001)

Qualified Computations in Functional Logic Programming[⋆]

Rafael Caballero, Mario Rodríguez-Artalejo, and Carlos A. Romero-Díaz

Departamento de Sistemas Informáticos y Computación, Universidad Complutense
Facultad de Informática, 28040 Madrid, Spain
{rafa,mario}@sip.ucm.es and cromdia@fdi.ucm.es

Abstract. Qualification has been recently introduced as a generalization of uncertainty in the field of Logic Programming. In this paper we investigate a more expressive language for First-Order Functional Logic Programming with Constraints and Qualification. We present a Rewriting Logic which characterizes the intended semantics of programs, and a prototype implementation based on a semantically correct program transformation. Potential applications of the resulting language include flexible information retrieval. As a concrete illustration, we show how to write program rules to compute qualified answers for user queries concerning the books available in a given library.

Keywords: Constraints, Functional Logic Programming, Program Transformation, Qualification, Rewriting Logic.

1 Introduction

Various extensions of Logic Programming with uncertain reasoning capabilities have been widely investigated during the last 25 years. The recent recollection [16] reviews the evolution of the subject from the viewpoint of a committed researcher. All the proposals in the field replace classical two-valued logic by some kind of many-valued logic with more than two truth values, which are attached to computed answers and interpreted as truth degrees.

In a recent paper [14] we have presented a *Qualified Logic Programming* scheme QLP($\mathcal{D}$) parameterized by a *qualification domain* $\mathcal{D}$, a lattice of so-called *qualification values* that are attached to computed answers and interpreted as a measure of the satisfaction of certain user's expectations. QLP($\mathcal{D}$)-programs are sets of clauses of the form $A \overset{\alpha}{\leftarrow} \overline{B}$, where the head A is an atom, the body $\overline{B}$ is a conjunction of atoms, and $\alpha \in \mathcal{D}$ is called *attenuation* factor. Intuitively, α measures the maximum confidence placed on an inference performed by the clause. More precisely, any successful application of the clause attaches to the head a qualification value which cannot exceed the infimum of $\alpha \circ \beta_i \in \mathcal{D}$, where β_i are the qualification values computed for the body atoms and $\circ$ is a so-called *attenuation operator*, provided by $\mathcal{D}$.

⋆ Research partially supported by projects MERIT–FORMS (TIN2005-09027-C03-03), PROMESAS–CAM(S-0505/TIC/0407) and STAMP (TIN2008-06622-C03-01).

Uncertain Logic Programming can be expressed by particular instances of QLP($\mathcal{D}$), where the user's expectation is understood as a lower bound for the *truth degree* of the computed answer and $\mathcal{D}$ is chosen to formalize a lattice of non-classical truth values. Other choices of $\mathcal{D}$ can be designed to model other kinds of user expectations, as e.g. an upper bound for the *size* of the logical proof underlying the computed answer. As shown in [3], the QLP($\mathcal{D}$) scheme is also well suited to deal with Uncertain Logic Programming based on similarity relations in the line of [15]. Therefore, Qualified Logic Programming has a potential for flexible information retrieval applications, where the answers computed for user's queries may match the user's expectations only to some degree. As shown in [14], several useful instances of QLP($\mathcal{D}$) can be conveniently implemented by using constraint solving techniques.

In this paper we investigate an extension of QLP($\mathcal{D}$) to a more expressive scheme, supporting computation with first-order lazy functions and constraints. More precisely, we consider the first-order fragment of CFLP($\mathcal{C}$), a generic scheme for functional logic programming with constraints over a parametrically given domain $\mathcal{C}$ presented in [9]. We propose an extended scheme QCFLP($\mathcal{D}, \mathcal{C}$) where the additional parameter $\mathcal{D}$ stands for a qualification domain. QCFLP($\mathcal{D}, \mathcal{C}$)-programs are sets of conditional rewrite rules of the form $f(\overline{t}_n) \xrightarrow{\alpha} r \Leftarrow \Delta$, where the condition Δ is a conjunction of $\mathcal{C}$-constraints that may involve user defined functions, and $\alpha \in \mathcal{D}$ is an attenuation factor. As in the logic programming case, α measures the maximum confidence placed on an inference performed by the rule: any successful application of the rule attaches to the computed result a qualification value which cannot exceed the infimum of $\alpha \circ \beta_i \in \mathcal{D}$, where β_i are the qualification values computed for r and Δ, and $\circ$ is $\mathcal{D}$'s attenuation operator. QLP($\mathcal{D}$) program clauses can be easily formulated as a particular case of QCFLP($\mathcal{D}, \mathcal{C}$) program rules.

As far as we know, no related work covers the expressivity of our approach. Guadarrama et al. [6] have proposed to use real arithmetic constraints as an implementation tool for a Fuzzy Prolog, but their language does not support constraint programming as such. Starting from the field of natural language processing, Riezler [11,12] has developed quantitative and probabilistic extensions of the classical CLP($\mathcal{C}$) scheme with the aim of computing good parse trees for constraint logic grammars, but his work bears no relation to functional programming. Moreno and Pascual [10] have investigated similarity-based unification in the context of *needed narrowing* [1], a narrowing strategy using so-called *definitional trees* that underlies the operational semantics of functional logic languages such as Curry [7] and $\mathcal{TOY}$ [2], but they use neither constraints nor attenuation factors and they provide no declarative semantics.

Figure 1 below shows a small set of QCFLP($\mathcal{U}, \mathcal{R}$) program rules, called the *library program* in the rest of the paper. The concrete syntax is inspired by the functional logic language $\mathcal{TOY}$, but the ideas and results of this paper could be applied also to Curry and other similar languages. In this example, $\mathcal{U}$ stands for a particular qualification domain which supports uncertain truth values in the real interval $[0, 1]$, while $\mathcal{R}$ stands for a particular constraint domain which

```
%% Data types:
type pages, id = int
type title, author, language, genre = [char]
data vocabularyLevel = easy | medium | difficult
data readerLevel = basic | intermediate | upper | proficiency
data book = book(id, title, author, language, genre, vocabularyLevel, pages)

%% Simple library, represented as list of books:
library :: [book]
library --> [ book(1, "Tintin", "Herge", "French", "Comic", easy, 65),
              book(2, "Dune", "F. P. Herbert", "English", "SciFi", medium, 345),
              book(3, "Kritik der reinen Vernunft", "Immanuel Kant", "German",
                   "Philosophy", difficult, 1011),
              book(4, "Beim Hauten der Zwiebel", "Gunter Grass", "German",
                   "Biography", medium, 432) ]

%% Auxiliary function for computing list membership:
member(B,[]) --> false
member(B,H:_T) --> true <== B == H
member(B,H:T) --> member(B,T) <== B /= H

%% Functions for getting the explicit attributes of a given book:
getId(book(Id,_Title,_Author,_Lang,_Genre,_VocLvl,_Pages)) --> Id
getTitle(book(_Id,Title,_Author,_Lang,_Genre,_VocLvl,_Pages)) --> Title
getAuthor(book(_Id,_Title,Author,_Lang,_Genre,_VocLvl,_Pages)) --> Author
getLanguage(book(_Id,_Title,_Author,Lang,_Genre,_VocLvl,_Pages)) --> Lang
getGenre(book(_Id,_Title,_Author,_Lang,Genre,_VocLvl,_Pages)) --> Genre
getVocabularyLevel(book(_Id,_Title,_Author,_Lang,_Genre,VocLvl,_Pages)) --> VocLvl
getPages(book(_Id,_Title,_Author,_Lang,_Genre,_VocLvl,Pages)) --> Pages

%% Function for guessing the genre of a given book:
guessGenre(B) --> getGenre(B)
guessGenre(B) -0.9-> "Fantasy" <== guessGenre(B) == "SciFi"
guessGenre(B) -0.8-> "Essay" <== guessGenre(B) == "Philosophy"
guessGenre(B) -0.7-> "Essay" <== guessGenre(B) == "Biography"
guessGenre(B) -0.7-> "Adventure" <== guessGenre(B) == "Fantasy"

%% Function for guessing the reader level of a given book:
guessReaderLevel(B) --> basic <== getVocabularyLevel(B) == easy, getPages(B) < 50
guessReaderLevel(B) -0.8-> intermediate <== getVocabularyLevel(B) == easy, getPages(B) >= 50
guessReaderLevel(B) -0.9-> basic <== guessGenre(B) == "Children"
guessReaderLevel(B) -0.9-> proficiency <== getVocabularyLevel(B) == difficult,
                                           getPages(B) >= 200
guessReaderLevel(B) -0.8-> upper <== getVocabularyLevel(B) == difficult, getPages(B) < 200
guessReaderLevel(B) -0.8-> intermediate <== getVocabularyLevel(B) == medium
guessReaderLevel(B) -0.7-> upper <== getVocabularyLevel(B) == medium

%% Function for answering a particular kind of user's query:
search(Language,Genre,Level) --> getId(B) <== member(B,library),
                                              getLanguage(B) == Language,
                                              guessReaderLevel(B) == Level,
                                              guessGenre(B) == Genre
```

Fig. 1. Library with books in different languages

supports arithmetic constraints over the real numbers; see Section 2 for more details.

The program rules are intended to encode expert knowledge for computing qualified answers to user's queries concerning the books available in a simplified library, represented as a list of objects of type `book`. The various `get` functions extract the explicit values of book attributes. Functions `guessGenre` and `guessReaderLevel` perform qualified inferences relying on analogies between different genres and heuristic rules to estimate reader levels on the basis of other features of a given book, respectively. For instance, the second rule for `guessGenre` infers the genre `"Fantasy"` with attenuation `0.9` for a book `B` whose genre is already known to be `"SciFi"`. Some program rules, as e.g. those of the auxiliary function `member`, have attached no explicit attenuation factor. By convention, this is understood as the implicit attachment of the attenuation factor `1.0`, the top value of $\mathcal{U}$. For any instance of the QCFLP($\mathcal{D}, \mathcal{C}$) scheme, a similar convention allows to view CFLP($\mathcal{C}$)-program rules as QCFLP($\mathcal{D}, \mathcal{C}$)-program rules whose attached qualification is optimal.

The last rule for function `search` encodes a method for computing qualified answers to a particular kind of user's queries. Therefore, the queries can be formulated as goals to be solved by the program fragment. For instance, answering the query of a user who wants to find a book of genre `"Essay"`, language `"German"` and user level `intermediate` with a certainty degree of at least 0.65 can be formulated as the goal:

```
(search("German","Essay",intermediate) == R) # W | W >= 0.65
```

The techniques presented in Section 4 can be used to translate the QCFLP($\mathcal{U}, \mathcal{R}$) program rules and goal into the CFLP($\mathcal{R}$) language, which is implemented in the $\mathcal{TOY}$ system. Solving the translated goal in $\mathcal{TOY}$ computes the answer $\{R \mapsto 4\}\{0.65 \le W, W \le 0.7\}$, ensuring that the library book with `id` 4 satisfies the query's requirements with any certainty degree in the interval [0.65,0.7], in particular 0.7. The computation uses the 4th program rule of `guessGenre` to obtain `"Essay"` as the book's genre with qualification 0.7, and the 6th program rule of `guessReaderLevel` to obtain `intermediate` as the reader level with qualification 0.8.

The rest of the paper is organized as follows. In Section 2 we recall known proposals concerning qualification and constraint domains, and we introduce a technical notion needed to relate both kinds of domains for the purposes of this paper. In Section 3 we present the generic scheme QCFLP($\mathcal{D}, \mathcal{C}$) announced in this introduction, and we formalize a special Rewriting Logic which characterizes the declarative semantics of QCFLP($\mathcal{D}, \mathcal{C}$)-programs. In Section 4 we present a semantically correct program transformation converting QCFLP($\mathcal{D}, \mathcal{C}$) programs and goals into the qualification-free CFLP($\mathcal{C}$) programming scheme, which is supported by existing systems such as $\mathcal{TOY}$. Section 5 concludes and points to some lines of planned future work. The Technical Report [4] includes full proofs of the main results in this paper, as well as some additional results concerning alternative characterizations of program semantics.

2 Qualification and Constraint Domains

A *Qualification Domain* is any algebraic structure $\mathcal{D} = \langle D, \trianglelefteq, \mathbf{b}, \mathbf{t}, \circ \rangle$ such that D is a set of elements called *qualification values*, $\langle D, \trianglelefteq, \mathbf{b}, \mathbf{t} \rangle$ is a lattice with extreme points $\mathbf{b}$ and $\mathbf{t}$ w.r.t. the partial ordering $\trianglelefteq$ and $\circ : D \times D \to D$ is a so-called *attenuation operation* satisfying the axioms stated in [14]. When convenient, D will be also noted as $D_{\mathcal{D}}$.

The intended use of qualification domains has been explained in the introduction. The examples in this paper will use a particular qualification domain $\mathcal{U}$ whose values represent certainty degrees in the sense of fuzzy logic. Formally, $\mathcal{U} = \langle U, \leq, 0, 1, \times \rangle$, where $U = [0, 1] = \{d \in \mathbb{R} \mid 0 \leq d \leq 1\}$, $\leq$ is the usual numerical ordering, and $\times$ is the multiplication operation. In this domain, the bottom and top elements are $\mathbf{b} = 0$ and $\mathbf{t} = 1$, and the infimum of a finite $S \subseteq U$ is the minimum value $\min(S)$, understood as 1 if $S = \emptyset$. The reader is referred to [14] for other useful instances of qualification domains.

Constraint domains are used in Constraint Logic Programming and its extensions as a tool to provide data values, primitive operations and constraints tailored to domain-oriented applications. Various formalizations of this notion are known. In this paper, constraint domains are related to signatures of the form $\Sigma = \langle DC, PF, DF \rangle$ where $DC = \bigcup_{n \in \mathbb{N}} DC^n$, $PF = \bigcup_{n \in \mathbb{N}} PF^n$ and $DF = \bigcup_{n \in \mathbb{N}} DF^n$ are mutually disjoint sets of *data constructor* symbols, *primitive function* symbols and *defined function* symbols, respectively, ranked by arities. Given a signature Σ, a symbol $\bot$ to note the *undefined value*, a set B of *basic values* u and a countably infinite set $\mathcal{V}ar$ of variables X, we define the notions listed below, where $\overline{o}_n$ abbreviates the n-tuple of syntactic objects $o_1, \ldots, o_n$.

- *Expressions* $e \in \mathrm{Exp}_\bot(\Sigma, B, \mathcal{V}ar)$ have the syntax $e ::= \bot | X | u | h(\overline{e}_n)$, where $h \in DC^n \cup PF^n \cup DF^n$. In the case $n = 0$, $h(\overline{e}_n)$ is written simply as h.
- *Constructor Terms* $t \in \mathrm{Term}_\bot(\Sigma, B, \mathcal{V}ar)$ have the syntax $e ::= \bot | X | u | c(\overline{t}_n)$, where $c \in DC^n$. They will be called just terms in the sequel.
- *Total Expressions* $e \in \mathrm{Exp}(\Sigma, B, \mathcal{V}ar)$ and *Total Terms* $t \in \mathrm{Term}(\Sigma, B, \mathcal{V}ar)$ have a similar syntax, with the $\bot$ case omitted.
- An expression or term (total or not) is called *ground* iff it includes no occurrences of variables. $\mathrm{Exp}_\bot(\Sigma, B)$ stands for the set of all ground expressions. The notations $\mathrm{Term}_\bot(\Sigma, B)$, $\mathrm{Exp}(\Sigma, B)$ and $\mathrm{Term}(\Sigma, B)$ have a similar meaning.
- We note as $\sqsubseteq$ the *information ordering*, defined as the least partial ordering over $\mathrm{Exp}_\bot(\Sigma, B, \mathcal{V}ar)$ compatible with contexts and verifying $\bot \sqsubseteq e$ for all $e \in \mathrm{Exp}_\bot(\Sigma, B, \mathcal{V}ar)$.
- Substitutions are defined as mappings $\sigma : \mathcal{V}ar \to \mathrm{Term}_\bot(\Sigma, B, \mathcal{V}ar)$ assigning not necessarily total terms to variables. They can be represented as sets of bindings $X \mapsto t$ and extended to act over other syntactic objects o. The *domain* $\mathrm{dom}(\sigma)$ and *variable range* $\mathrm{vran}(\sigma)$ are defined in the usual way. We will write $o\sigma$ for the result of applying σ to o. The *composition* $\sigma\sigma'$ of two substitutions is such that $o(\sigma\sigma')$ equals $(o\sigma)\sigma'$.

By adapting the definition found in Section 2.2 of [9] to a first-order setting, we formalize a constraint domain of signature Σ as any algebraic structure of the form $\mathcal{C} = \langle C, \{p^\mathcal{C} \mid p \in PF\} \rangle$ such that:

1. The carrier set C is $\mathrm{Term}_\bot(\Sigma, B)$ for a certain set B of *basic values*. When convenient, we note B and C as $B_\mathcal{C}$ and $C_\mathcal{C}$, respectively.
2. $p^\mathcal{C} \subseteq C^n \times C$, written simply as $p^\mathcal{C} \subseteq C$ in the case $n = 0$, is called the *interpretation* of p in $\mathcal{C}$. We will write $p^\mathcal{C}(\overline{t}_n) \to t$ (or simply $p^\mathcal{C} \to t$ if $n = 0$) to indicate that $(\overline{t}_n, t) \in p^\mathcal{C}$.

3. Each primitive interpretation $p^{\mathcal{C}}$ has *monotonic* and *radical* behavior w.r.t. the information ordering $\sqsubseteq$, in the technical sense defined in [4].

Note that symbols $h \in DC \cup DF$ are given no interpretation in $\mathcal{C}$. As we will see in Section 3, symbols in $c \in DC$ are interpreted as free constructors, and the interpretation of symbols $f \in DF$ is program-dependent. We assume that any signature Σ includes two nullary constructors *true* and *false* for the boolean values, and a binary symbol $== \in PF^2$ used in infix notation and interpreted as *strict equality*; see [9] for details. For the examples in this paper we will use a constraint domain $\mathcal{R}$ whose set of basic elements is $C_{\mathcal{R}} = \mathbb{R}$ and whose primitives functions correspond to the usual arithmetic operations $+, \times, \ldots$ and the usual boolean-valued comparison operations $\leq, <, \ldots$ over $\mathbb{R}$. Other useful instances of constraint domains can be found in [9].

Atomic constraints over $\mathcal{C}$ have the form $p(\overline{e}_n) == v$ with $p \in PF^n$, $e_i \in \mathrm{Exp}_\perp(\Sigma, B, \mathcal{V}ar)$ and $v \in \mathcal{V}ar \cup DC^0 \cup B_{\mathcal{C}}$. Atomic constraints of the form $p(\overline{e}_n) == true$ are abbreviated as $p(\overline{e}_n)$. In particular, $(e_1 == e_2) == true$ is abbreviated as $e_1 == e_2$. Atomic constraints of the form $(e_1 == e_2) == false$ are abbreviated as $e_1 \mathrel{/=} e_2$.

Compound constraints are built from atomic constraints using logical conjunction, existential quantification, and sometimes other logical operations. Constraints without occurrences of symbols $f \in DF$ are called *primitive.* We will note atomic constraints as δ, sets of atomic constraints as Δ, atomic primitive constraints as π, and sets of atomic primitive constraints as Π. When interpreting sets of constraints, we will treat them as the conjunction of their members.

Ground substitutions η such that $X\eta \in \mathrm{Term}_\perp(\Sigma, B)$ for all $X \in \mathrm{dom}(\eta)$ are called *variable valuations* over $\mathcal{C}$. The set of all possible variable valuations is noted $\mathrm{Val}_{\mathcal{C}}$. The *solution set* $\mathrm{Sol}_{\mathcal{C}}(\Pi) \subseteq \mathrm{Val}_{\mathcal{C}}$ includes as members those valuations η such that $\pi\eta$ is true in $\mathcal{C}$ for all $\pi \in \Pi$; see [9] for a formal definition. In case that $\mathrm{Sol}_{\mathcal{C}}(\Pi) = \emptyset$ we say that Π is *unsatisfiable* and we write $\mathrm{Unsat}_{\mathcal{C}}(\Pi)$. In case that $\mathrm{Sol}_{\mathcal{C}}(\Pi) \subseteq \mathrm{Sol}_{\mathcal{C}}(\pi)$ we say that π is *entailed* by Π in $\mathcal{C}$ and we write $\Pi \models_{\mathcal{C}} \pi$. Note that the notions defined in this paragraph only make sense for primitive constraints.

In this paper we are interested in pairs consisting of a qualification domain and a constraint domain that are related in the following technical sense:

Definition 1 (Expressing $\mathcal{D}$ in $\mathcal{C}$). *A qualification domain $\mathcal{D}$ with carrier set $D_{\mathcal{D}}$ is expressible in a constraint domain $\mathcal{C}$ with carrier set $C_{\mathcal{C}}$ if $D_{\mathcal{D}} \setminus \{\mathbf{b}\} \subseteq C_{\mathcal{C}}$ and the two following requirements are satisfied:*

1. *There is a primitive $\mathcal{C}$-constraint $\mathsf{qVal}(X)$ depending on the variable X, such that $Sol_{\mathcal{C}}(\mathsf{qVal}(X)) = \{\eta \in Val_{\mathcal{C}} \mid \eta(X) \in D_{\mathcal{D}} \setminus \{\mathbf{b}\}\}$.*
2. *There is a primitive $\mathcal{C}$-constraint $\mathsf{qBound}(X, Y, Z)$ depending on the variables X, Y, Z, such that any $\eta \in Val_{\mathcal{C}}$ such that $\eta(X)$, $\eta(Y)$, $\eta(Z) \in D_{\mathcal{D}} \setminus \{\mathbf{b}\}$ verifies $\eta \in Sol_{\mathcal{C}}(\mathsf{qBound}(X, Y, Z)) \iff \eta(X) \trianglelefteq \eta(Y) \circ \eta(Z)$.* □

Intuitively, $\mathsf{qBound}(X, Y, Z)$ encodes the $\mathcal{D}$-statement $X \trianglelefteq Y \circ Z$ as a $\mathcal{C}$-constraint. As convenient notations, we will write $\ulcorner X \trianglelefteq Y \circ Z \urcorner$, $\ulcorner X \trianglelefteq Y \urcorner$ and $\ulcorner X \trianglerighteq Y \urcorner$ in place of $\mathsf{qBound}(X,Y,Z)$, $\mathsf{qBound}(X,\mathbf{t},Y)$ and $\mathsf{qBound}(Y,\mathbf{t},X)$, respectively. In the sequel, $\mathcal{C}$-constraints of the form $\ulcorner\kappa\urcorner$ are called *qualification constraints*, and Ω is used as notation for sets of qualification constraints. We also write $\mathrm{Val}_{\mathcal{D}}$ for the set of all $\mu \in \mathrm{Val}_{\mathcal{C}}$ such that $X\mu \in D_{\mathcal{D}} \setminus \{\mathbf{b}\}$ for all $X \in \mathrm{dom}(\mu)$, called *$\mathcal{D}$-valuations.*

Note that $\mathcal{U}$ can be expressed in $\mathcal{R}$, because $D_{\mathcal{U}} \setminus \{0\} = (0,1] \subseteq \mathbb{R} \subseteq C_{\mathcal{R}}$, $\mathsf{qVal}(X)$ can be built as the $\mathcal{R}$-constraint $0 < X \wedge X \leq 1$ and $\ulcorner X \trianglelefteq Y \circ Z\urcorner$ can be built as the $\mathcal{R}$-constraint $X \leq Y \times Z$. Other instances of qualification domains presented in [14] are also expressible in $\mathcal{R}$.

3 A Qualified Declarative Programming Scheme

In this section we present the scheme QCFLP($\mathcal{D},\mathcal{C}$) announced in the Introduction and its declarative semantics. The parameters $\mathcal{D}$ and $\mathcal{C}$ stand for a qualification domain and a constraint domain with certain signature Σ, respectively. By convention, we only allow those instances of the scheme verifying that $\mathcal{D}$ is expressible in $\mathcal{C}$ in the sense of Definition 1. For example, QCFLP($\mathcal{U},\mathcal{R}$) is an allowed instance.

A QCFLP($\mathcal{D},\mathcal{C}$)-program is a set $\mathcal{P}$ of program rules. A program rule has the form $f(\overline{t}_n) \xrightarrow{\alpha} r \Leftarrow \Delta$ where $f \in DF^n$, $\overline{t}_n$ is a linear sequence of Σ-terms (where each variable occurs just once), $\alpha \in D_{\mathcal{D}} \setminus \{\mathbf{b}\}$ is an attenuation factor, r is a Σ-expression and Δ is a sequence of atomic $\mathcal{C}$-constraints δ_j $(1 \leq j \leq m)$, interpreted as conjunction. The undefined symbol $\perp$ is not allowed to occur in program rules. The library program shown in Figure 1 serves as an example of QCFLP($\mathcal{U},\mathcal{R}$)-program. Leaving aside the attenuation factors, this is clearly not a confluent conditional term rewriting system. Certain program rules, as e.g. those for `guessGenre`, are intended to specify the behavior of *non-deterministic functions.* As argued elsewhere [13], the semantics of non-deterministic functions for the purposes of Functional Logic Programming is not suitably described by ordinary rewriting. We overcome this difficulty by designing a formal inference system noted QCRWL($\mathcal{D},\mathcal{C}$) and called *Qualified Constrained Rewriting Logic.* First, we define the kind of statements that can be inferred in this logic:

Definition 2 (qc-Statements). *Assume a partial Σ-expression e, partial Σ-terms $t, t', \overline{t}_n$, a qualification value $d \in D_{\mathcal{D}} \setminus \{\mathbf{b}\}$, an atomic $\mathcal{C}$-constraint δ and a finite set of atomic primitive $\mathcal{C}$-constraints Π. A* qualified constrained statement *(briefly, qc-statement) φ must have one of the following two forms:*

1. qc-production $(e \to t)\sharp d \Leftarrow \Pi$*. Such a qc-statement is called* trivial *iff either t is $\perp$ or else $Unsat_{\mathcal{C}}(\Pi)$. Its intuitive meaning is that a rewrite sequence $e \to^* t'$ using program rules and with attached qualification value d is allowed in our intended semantics for some $t' \sqsupseteq t$, under the assumption that Π holds. By convention, qc-productions of the form $(f(\overline{t}_n) \to t)\sharp d \Leftarrow \Pi$ with $f \in DF^n$ are called* qc-facts.
2. qc-atom $\delta\sharp d \Leftarrow \Pi$*. Such a qc-statement is called* trivial *iff $Unsat_{\mathcal{C}}(\Pi)$. Its intuitive meaning is that δ is entailed by the program rules with attached qualification value d, under the assumption that Π holds.* □

QTI $\dfrac{}{\varphi}$ if φ is a trivial qc-statement.

QRR $\dfrac{}{(v \to v)\sharp d \Leftarrow \Pi}$ if $v \in \mathcal{V}ar \cup B_{\mathcal{C}}$ and $d \in D_{\mathcal{D}} \setminus \{\mathbf{b}\}$.

QDC $\dfrac{(\ (e_i \to t_i)\sharp d_i \Leftarrow \Pi\ )_{i=1\dots n}}{(c(\overline{e}_n) \to c(\overline{t}_n))\sharp d \Leftarrow \Pi}$ if $c \in DC^n$ and $d \in D_{\mathcal{D}} \setminus \{\mathbf{b}\}$ verifies $d \trianglelefteq d_i$ $(1 \le i \le n)$.

QDF$_{\mathcal{P}}$ $\dfrac{(\ (e_i \to t_i)\sharp d_i \Leftarrow \Pi\ )_{i=1\dots n} \quad (r \to t)\sharp d'_0 \Leftarrow \Pi \quad (\delta_j \sharp d'_j \Leftarrow \Pi)_{j=1\dots m}}{(f(\overline{e}_n) \to t)\sharp d \Leftarrow \Pi}$

if $f \in DF^n$ and $(f(\overline{t}_n) \xrightarrow{\alpha} r \Leftarrow \delta_1, \dots, \delta_m) \in [\mathcal{P}]_\perp$ where $[\mathcal{P}]_\perp = \{R_l\theta \mid R_l$ is a rule in $\mathcal{P}$ and θ is a substitution$\}$ is the set of program rule instances, and $d \in D_{\mathcal{D}} \setminus \{\mathbf{b}\}$ verifies $d \trianglelefteq d_i$ $(1 \le i \le n)$, $d \trianglelefteq \alpha \circ d'_j$ $(0 \le j \le m)$.

QPF $\dfrac{(\ (e_i \to t_i)\sharp d_i \Leftarrow \Pi\ )_{i=1\dots n}}{(p(\overline{e}_n) \to v)\sharp d \Leftarrow \Pi}$ if $p \in PF^n$, $v \in \mathcal{V}ar \cup DC^0 \cup B_{\mathcal{C}}$,

$\Pi \models_{\mathcal{C}} p(\overline{t}_n) \to v$ and $d \in D_{\mathcal{D}} \setminus \{\mathbf{b}\}$ verifies $d \trianglelefteq d_i$ $(1 \le i \le n)$.

QAC $\dfrac{(\ (e_i \to t_i)\sharp d_i \Leftarrow \Pi\ )_{i=1\dots n}}{(p(\overline{e}_n) == v)\sharp d \Leftarrow \Pi}$ if $p \in PF^n$, $v \in \mathcal{V}ar \cup DC^0 \cup B_{\mathcal{C}}$,

$\Pi \models_{\mathcal{C}} p(\overline{t}_n) == v$ and $d \in D_{\mathcal{D}} \setminus \{\mathbf{b}\}$ verifies $d \trianglelefteq d_i$ $(1 \le i \le n)$.

Fig. 2. Qualified Constrained Rewriting Logic

Next, we define QCRWL$(\mathcal{D}, \mathcal{C})$ as the formal system consisting of the six inference rules displayed in Fig. 2. They are based on the first-order fragment of the Constrained Rewriting Logic presented in [9], suitably extended to manage attached qualification values. These inference rules formalize provability of qc-statements according to their intuitive meanings. In particular, **QDF$_{\mathcal{P}}$** formalizes the applications of a program rule instance to infer that $f(\overline{e}_n)$ returns a result t with qualification d. Note that d is bounded by the qualifications d_i corresponding to the evaluation of e_i, and also by $\alpha \circ d'_j$ corresponding to the evaluation of the right hand side and the conditions of the rule attenuated by α.

In the sequel we use the notation $\mathcal{P} \vdash_{\mathcal{D},\mathcal{C}} \varphi$ to indicate that φ can be inferred from $\mathcal{P}$ in QCRWL$(\mathcal{D}, \mathcal{C})$. By convention, we agree that no other inference rule is used whenever **QTI** is applicable. Therefore, trivial qc-statements can only be inferred by rule **QTI**. As usual in formal inference systems, QCRWL$(\mathcal{D}, \mathcal{C})$ proofs can be represented as trees whose nodes correspond to inference steps. For example, if $\mathcal{P}$ is the library program, Π is empty, and ψ is

```
(guessGenre(book(4,"Beim Hauten der Zwiebel","Gunter Grass",
    "German","Biography", medium, 432)) --> "Essay")#0.7
```

then $\mathcal{P} \vdash_{\mathcal{U},\mathcal{R}} \psi \Leftarrow \Pi$ with a proof tree whose root inference can be chosen as $\mathbf{QDF}_{\mathcal{P}}$ using a suitable instance of the 4th program rule for `guessGenre`.

Extending ideas from [9], it is possible to define *qc-interpretations* as sets $\mathcal{I}$ of qc-facts that verify certain closure conditions. Moreover, *models* of $\mathcal{P}$ can be defined to be those interpretations that satisfy the program rules in a suitable sense. The following result can be proved:

Theorem 1 (Least Program Model). *For any QCFLP($\mathcal{D}$,$\mathcal{C}$)-program $\mathcal{P}$, $S_{\mathcal{P}} = \{\varphi \mid \varphi$ is a qc-fact and $\mathcal{P} \vdash_{\mathcal{D},\mathcal{C}} \varphi\}$ is the least model of $\mathcal{P}$ w.r.t. set inclusion. An alternative characterization of $S_{\mathcal{P}}$ as least fixpoint is also possible.* □

Assume now a QCFLP($\mathcal{D}$,$\mathcal{C}$)-program $\mathcal{P}$ and a countable set $\mathcal{W}ar$ of so-called *qualification variables*, disjoint from $\mathcal{V}ar$ and $\mathcal{C}$'s signature Σ. Then:

Definition 3 (Goals and their Solutions).

1. *A* goal *G for $\mathcal{P}$ has the form $\delta_1\sharp W_1, \ldots, \delta_m\sharp W_m \;[]\; W_1 \trianglerighteq \beta_1, \ldots, W_m \trianglerighteq \beta_m$, abbreviated as $(\ \delta_i\sharp W_i,\ W_i \trianglerighteq \beta_i\ )_{i=1\ldots m}$, where $\delta_i\sharp W_i$ $(1 \leq i \leq m)$ are atomic $\mathcal{C}$-constraints annotated with different qualification variables W_i, and $W_i \trianglerighteq \beta_i$ are so-called* threshold conditions*, with $\beta_i \in D_{\mathcal{D}} \setminus \{\mathbf{b}\}$ $(1 \leq i \leq m)$.*
2. *A* solution *for G is any triple $\langle \sigma, \mu, \Pi \rangle$ such that σ is a substitution, μ is a $\mathcal{D}$-valuation, Π is a finite set of atomic primitive $\mathcal{C}$-constraints, and the following two conditions hold for all $1 \leq i \leq m$: $W_i\mu = d_i \trianglerighteq \beta_i$, and $\mathcal{P} \vdash_{\mathcal{D},\mathcal{C}} (\delta_i\sigma)\sharp d_i \Leftarrow \Pi$. The set of all solutions for G is noted $Sol_{\mathcal{P}}(G)$.* □

Thanks to Theorem 1, solutions of $\mathcal{P}$ are valid in the least model $S_{\mathcal{P}}$ and hence in all models of $\mathcal{P}$. A goal for the library program and one solution for it have been presented in the Introduction. In this particular example, $\Pi = \emptyset$ and the QCRWL($\mathcal{U}$, $\mathcal{R}$) proof needed to check the solution according to Definition 3 can be formalized by following the intuitive ideas sketched in the Introduction.

4 Implementation by Program Transformation

Goal solving in instances of the CFLP($\mathcal{C}$) scheme from [9] has been formalized by means of *constrained narrowing* procedures as e.g. [8,5], and is supported by systems such as Curry [7] and $\mathcal{TOY}$ [2]. In this section we present a semantically correct transformation from QCFLP($\mathcal{D}$,$\mathcal{C}$) into the first-order fragment of CFLP($\mathcal{C}$) which can be used for implementing goal solving in QCFLP($\mathcal{D}$,$\mathcal{C}$).

By abuse of notation, the first-order fragment of the CFLP($\mathcal{C}$) scheme will be noted simply as CFLP($\mathcal{C}$) in the sequel. A formal description of CFLP($\mathcal{C}$) is easily derived from the previous Section 3 by simply omitting everything related to qualification domains and values. Programs $\mathcal{P}$ are sets of program rules of the form $f(\bar{t}_n) \to r \Leftarrow \Delta$, with no attenuation factors attached. Program semantics is characterized by a Constrained Rewriting Logic CRWL($\mathcal{C}$) where c-statements can be derived from a given program. A c-statement may be a c-production $e \to t \Leftarrow \Pi$ or a c-atom $\delta \Leftarrow \Pi$. The six inference rules **RL** of CRWL($\mathcal{C}$) are easy to derive from the corresponding rules **QRL** of QCRWL($\mathcal{D}$,$\mathcal{C}$). For instance, the CRWL($\mathcal{C}$) rule derived from **QAC** by forgetting qualifications is:

$$\textbf{AC} \quad \frac{(\ e_i \to t_i \Leftarrow \Pi\ )_{i=1\dots n}}{p(\overline{e}_n) == v \Leftarrow \Pi} \quad \begin{array}{l}\text{if } p \in PF^n,\ v \in \mathcal{V}ar \cup DC^0 \cup B_{\mathcal{C}} \\ \text{and } \Pi \models_{\mathcal{C}} p(\overline{t}_n) == v.\end{array}$$

The notation $\mathcal{P} \vdash_{\mathcal{C}} \varphi$ indicates that φ can be inferred from $\mathcal{P}$ in CRWL($\mathcal{C}$). In analogy to Theorem 1, it is possible to prove that the least model of $\mathcal{P}$ w.r.t. set inclusion can be characterized as $S_{\mathcal{P}} = \{\varphi \mid \varphi \text{ is a c-fact and } \mathcal{P} \vdash_{\mathcal{C}} \varphi\}$. In analogy to Definition 3, goals G for a CFLP($\mathcal{C}$)-program $\mathcal{P}$ have the form $\delta_1, \dots, \delta_m$ where δ_j are atomic $\mathcal{C}$-constraints, and $Sol_{\mathcal{P}}(G)$ is defined as the set of all the pairs $\langle \sigma, \Pi \rangle$ such that σ is a substitution, Π is a finite set of atomic primitive $\mathcal{C}$-constraints, and $\mathcal{P} \vdash_{\mathcal{C}} \delta_j\sigma \Leftarrow \Pi$ holds for $1 \leq j \leq m$.

The transformation goes from a source signature Σ into a target signature Σ' such that each $f \in DF^n$ in Σ becomes $f' \in DF^{n+1}$ in Σ', and all the other symbols in Σ remain the same in Σ'. It works by introducing fresh variables W to represent the qualification values attached to the results of calls to defined functions, as well as qualification constraints to be imposed on such variables. There are four groups of transformation rules displayed in Figure 3. Let us comment them in order.

Transforming an expression e yields a triple $e^{\mathcal{T}} = (e', \Omega, \mathcal{W})$, where Ω is a set of qualification constraints and $\mathcal{W}$ is the set of qualification variables occurring in e' at outermost positions. The qualification value attached to e cannot exceed the infimum in $\mathcal{D}$ of the values of the variables $W \in \mathcal{W}$, and $e^{\mathcal{T}}$ is computed by recursion on e's syntactic structure as specified by the transformation rules **TAE**, **TCE**$_1$ and **TCE**$_2$. Note that **TCE**$_2$ introduces a new qualification variable W for each call to a defined function $f \in DF^n$ and builds a set Ω' of qualification constraints ensuring that W must be interpreted as a qualification value not greater than the qualification values attached to f's arguments. **TCE**$_1$ deals with calls to constructors and primitive functions just by collecting information from the arguments, and **TAE** is self-explanatory.

Unconditional productions and atomic constraints are transformed by means of **TP** and **TA**, respectively, relying on the transformation of expressions in the obvious way. Relying on **TP** and **TA**, **TCS** transforms a qc-statement of the form $\psi\sharp d \Leftarrow \Pi$ into a c-statement whose conditional part includes, in addition to Π, the qualification constraints Ω coming from $\psi^{\mathcal{T}}$ and extra qualification constraints ensuring that d is not greater than allowed by ψ's qualification.

Program rules are transformed by **TPR**. Transforming the left-hand side $f(\overline{t}_n)$ introduces a fresh symbol $f' \in DF^{n+1}$ and a fresh qualification variable W. The transformed right-hand side r' comes from $r^{\mathcal{T}}$, and the transformed conditions are obtained from the constraints coming from $r^{\mathcal{T}}$ and ${\delta_i}^{\mathcal{T}}(1 \leq i \leq m)$ by adding extra qualification constraints to be imposed on W, namely $\mathsf{qVal}(W)$ and $(\ulcorner W \trianglelefteq \alpha \circ W' \urcorner)_{W' \in \mathcal{W}'}$, for $\mathcal{W}' = \mathcal{W}_r$ and $\mathcal{W}' = \mathcal{W}_i$ $(1 \leq i \leq m)$. By convention, $(\ulcorner W \trianglelefteq \alpha \circ W' \urcorner)_{W' \in \mathcal{W}'}$ is understood as $\ulcorner W \trianglelefteq \alpha \urcorner$ in case that $\mathcal{W}' = \emptyset$. The idea is that W's value cannot exceed the infimum in $\mathcal{D}$ of all the values $\alpha \circ \beta$, for the different β coming from the qualifications of r and δ_i $(1 \leq i \leq m)$. The result of applying **TPR** to all the program rules of a program $\mathcal{P}$ will be noted as $\mathcal{P}^{\mathcal{T}}$.

Transforming Expressions

TAE $$\frac{}{v^{\mathcal{T}} = (v, \emptyset, \emptyset)} \quad \text{if } v \in \mathcal{V}ar \cup B_{\mathcal{C}}$$

TCE$_1$ $$\frac{(\ e_i{}^{\mathcal{T}} = (e'_i, \Omega_i, \mathcal{W}_i)\ )_{i=1\ldots n}}{h(\overline{e}_n)^{\mathcal{T}} = (h(\overline{e'}_n),\ \bigcup_{i=1}^{n} \Omega_i,\ \bigcup_{i=1}^{n} \mathcal{W}_i)} \quad \text{if } h \in DC^n \cup PF^n$$

TCE$_2$ $$\frac{(\ e_i{}^{\mathcal{T}} = (e'_i, \Omega_i, \mathcal{W}_i)\ )_{i=1\ldots n}}{f(\overline{e}_n)^{\mathcal{T}} = (f'(\overline{e'}_n, W), \Omega', \{W\})}$$

if $f \in DF^n$ and W is a fresh variable,
where $\Omega' = (\bigcup_{i=1}^{n} \Omega_i) \cup \{\mathsf{qVal}(W)\} \cup \{\ulcorner W \trianglelefteqslant W' \urcorner \mid W' \in \bigcup_{i=1}^{n} \mathcal{W}_i\}$.

Transforming qc-Statements

TP $$\frac{e^{\mathcal{T}} = (e', \Omega, \mathcal{W})}{(e \to t)^{\mathcal{T}} = (e' \to t,\ \Omega, \mathcal{W})}$$

TA $$\frac{(\ e_i{}^{\mathcal{T}} = (e'_i, \Omega_i, \mathcal{W}_i)\ )_{i=1\ldots n}}{(p(\overline{e}_n) == v)^{\mathcal{T}} = (\ p(\overline{e'}_n) == v, \bigcup_{i=1}^{n} \Omega_i, \bigcup_{i=1}^{n} \mathcal{W}_i\ )}$$

if $p \in PF^n$, $v \in \mathcal{V}ar \cup DC^0 \cup B_{\mathcal{C}}$.

TCS $$\frac{\psi^{\mathcal{T}} = (\psi', \Omega, \mathcal{W})}{(\psi\sharp d \Leftarrow \Pi)^{\mathcal{T}} = (\psi' \Leftarrow \Pi, \Omega \cup \{\ulcorner d \trianglelefteqslant W \urcorner \mid W \in \mathcal{W}\})}$$

if ψ is of the form $e \to t$ or $p(\overline{e}_n) == v$ and $d \in D_{\mathcal{D}}$.

Transforming Program Rules

TPR $$\frac{r^{\mathcal{T}} = (r', \Omega_r, \mathcal{W}_r) \qquad (\ \delta_i{}^{\mathcal{T}} = (\delta'_i, \Omega_i, \mathcal{W}_i)\ )_{i=1\ldots m}}{\begin{array}{l}(f(\overline{t}_n) \xrightarrow{\alpha} r \Leftarrow \delta_1, \ldots, \delta_m)^{\mathcal{T}} = \\ \quad f'(\overline{t}_n, W) \to r' \Leftarrow \mathsf{qVal}(W),\ \Omega_r,\ (\ulcorner W \trianglelefteqslant \alpha \circ W' \urcorner)_{W' \in \mathcal{W}_r}, \\ \qquad\qquad (\ \Omega_i,\ (\ulcorner W \trianglelefteqslant \alpha \circ W' \urcorner)_{W' \in \mathcal{W}_i},\ \delta'_i\ )_{i=1\ldots m}\end{array}}$$

where W is a fresh variable.

Transforming Goals

TG $$\frac{(\ \delta_i{}^{\mathcal{T}} = (\delta'_i, \Omega_i, \mathcal{W}_i)\ )_{i=1\ldots m}}{\begin{array}{l}((\ \delta_i\sharp W_i, W_i \trianglerighteqslant \beta_i\ )_{i=1\ldots m})^{\mathcal{T}} = \\ \quad (\ \Omega_i,\ \mathsf{qVal}(W_i),\ (\ulcorner W_i \trianglelefteqslant W \urcorner)_{W \in \mathcal{W}_i},\ \ulcorner W_i \trianglerighteqslant \beta_i \urcorner,\ \delta'_i\ )_{i=1\ldots m}\end{array}}$$

Fig. 3. Transformation rules

Finally, **TG** transforms a goal $(\ \delta_i \sharp W_i, W_i \trianglerighteq \beta_i\ )_{i=1\ldots m}$ by transforming each atomic constraint δ_i and adding $\mathsf{qVal}(W_i)$, $(\ulcorner W_i \trianglelefteq W' \urcorner)_{W' \in \mathcal{W}'_i}$ and $\ulcorner W_i \trianglerighteq \beta_i \urcorner$ $(1 \le i \le m)$ to ensure that each W_i is interpreted as a qualification value not bigger than the qualification computed for δ_i and satisfying the threshold condition $W_i \trianglerighteq \beta_i$. In case that $\mathcal{W}'_i = \emptyset$, $(\ulcorner W_i \trianglelefteq W' \urcorner)_{W' \in \mathcal{W}'_i}$ is understood as $\ulcorner W_i \trianglelefteq \mathbf{t} \urcorner$.

Program semantics in QCFLP($\mathcal{D},\mathcal{C}$) and CFLP($\mathcal{C}$) is characterized by derivability in QCRWL($\mathcal{D},\mathcal{C}$) and CRWL($\mathcal{C}$), respectively. Therefore, the following theorem proves the semantic correctness of the program transformation:

Theorem 2. *Let $\mathcal{P}$ be a QCFLP($\mathcal{D},\mathcal{C}$)-program and $\psi \sharp d \Leftarrow \Pi$ a qc-statement such that $(\psi \sharp d \Leftarrow \Pi)^{\mathcal{T}} = (\psi' \Leftarrow \Pi, \Omega')$. Then the two following statements are equivalent:*

1. *$\mathcal{P} \vdash_{\mathcal{D},\mathcal{C}} \psi \sharp d \Leftarrow \Pi$.*
2. *$\mathcal{P}^{\mathcal{T}} \vdash_{\mathcal{C}} \psi' \rho \Leftarrow \Pi$ for some $\rho \in Sol_{\mathcal{C}}(\Omega')$ such that $dom(\rho) = var(\Omega')$.*

Proof. (Sketch; a full proof can be found in [4] as Proof of Theorem 3).

[1. $\Rightarrow$ 2.] *(Transformation completeness).* Assume $\mathcal{P} \vdash_{\mathcal{D},\mathcal{C}} \psi \sharp d \Leftarrow \Pi$ by means of a QCRWL($\mathcal{D},\mathcal{C}$) proof tree T with k nodes. By induction on k we show the existence of a CRWL($\mathcal{C}$) proof tree T' witnessing $\mathcal{P}^{\mathcal{T}} \vdash_{\mathcal{C}} \psi' \rho \Leftarrow \Pi$ for some $\rho \in \mathrm{Sol}_{\mathcal{C}}(\Omega)$ such that $\mathrm{dom}(\rho) = \mathrm{var}(\Omega')$. In the base case $k = 1$, T contains just one root node inferred by a QCRWL($\mathcal{D},\mathcal{C}$) inference rule **QRL** other than $\mathbf{QDF}_{\mathcal{P}}$ and with no premises. Then T' can be easily built as a proof tree which also contains just one root node inferred by the QCRWL($\mathcal{D},\mathcal{C}$) inference rule **RL** with no premises. In the inductive case $k > 1$ the QCRWL($\mathcal{D},\mathcal{C}$) inference rule **QRL** applied at T's root can be neither **QTI** nor **QRR**. Here we argue only for the case where **QRL** is **QAC**. In this case ψ has the form $p(\overline{e}_n) == v$ and according to Figure 2 the inference step at T's root has the form:

$$\frac{(\ (e_i \to t_i) \sharp d_i \Leftarrow \Pi\ )_{i=1\ldots n}}{(p(\overline{e}_n) == v) \sharp d \Leftarrow \Pi}$$

where $v \in \mathcal{V}ar \cup DC^0 \cup B_{\mathcal{C}}$, $\Pi \models_{\mathcal{C}} p(\overline{t}_n) == v$ and $d \in D_{\mathcal{D}} \setminus \{\mathbf{b}\}$ verifies $d \trianglelefteq d_i$ $(1 \le i \le n)$. Assume $(\ e_i{}^{\mathcal{T}} = (e'_i, \Omega_i, \mathcal{W}_i)\ )_{i=1\ldots n}$, using different fresh variables W in each case. Then the transformation rules **TA** and **TCS** yield $((p(\overline{e}_n) == v) \sharp d \Leftarrow \Pi)^{\mathcal{T}} = p(\overline{e'}_n) == v \Leftarrow \Pi, \Omega'$ and $((e_i \to t_i) \sharp d \Leftarrow \Pi)^{\mathcal{T}} = e'_i \to t_i \Leftarrow \Pi, \Omega'_i$, where $\Omega' = \bigcup_{i=1}^n \Omega_i \cup \{\ulcorner d \trianglelefteq W \urcorner \mid W \in \bigcup_{i=1}^n \mathcal{W}_i\}$ and $\Omega'_i = \Omega_i \cup \{\ulcorner d_i \trianglelefteq W \urcorner \mid W \in \mathcal{W}_i\}$. For each $1 \le i \le n$, $\mathcal{P} \vdash_{\mathcal{D},\mathcal{C}} (e_i \to t_i) \sharp d_i \Leftarrow \Pi$ is witnessed by a QCRWL($\mathcal{D},\mathcal{C}$) proof tree T_i which is subtree of T and has less than k nodes. Therefore, by induction hypothesis we get CRWL($\mathcal{C}$) proof trees T'_i $(1 \le i \le n)$ witnessing $\mathcal{P}^{\mathcal{T}} \vdash_{\mathcal{C}} (e'_i \to t_i)\rho_i \Leftarrow \Pi$ for certain $\rho_i \in \mathrm{Sol}_{\mathcal{C}}(\Omega_i)$ such that $\mathrm{dom}(\rho_i) = \mathrm{var}(\Omega'_i)$. Consider $\rho = \biguplus_{i=1}^n \rho_i \in \mathrm{Val}_{\mathcal{D}}$, which is is well defined because the sets $\mathrm{var}(\Omega'_i), 1 \le i \le n$, are pairwise disjoint. Note that $\mathrm{dom}(\rho) = \bigcup_{i=1}^n \mathrm{dom}(\rho_i) = \bigcup_{i=1}^n \mathrm{var}(\Omega'_i) = \mathrm{war}(\Omega')$. Moreover, $\rho \in \mathrm{Sol}_{\mathcal{C}}(\Omega')$. In fact, for each $1 \le i \le n$, $\rho \in \mathrm{Sol}_{\mathcal{C}}(\Omega_i)$ follows from $\rho_i \in \mathrm{Sol}_{\mathcal{C}}(\Omega_i)$; and for each $1 \le i \le n$ and each $W \in \mathcal{W}_i$, $\rho \in \mathrm{Sol}_{\mathcal{C}}(\ulcorner d \trianglelefteq W \urcorner)$ follows from $d \trianglelefteq d_i$

(ensured by the **QAC** inference at T's root) and $\rho \in \mathrm{Sol}_{\mathcal{C}}(\ulcorner d_i \trianglelefteq W \urcorner)$ (ensured by $\rho_i \in \mathrm{Sol}_{\mathcal{C}}(\Omega'_i)$ and $\ulcorner d_i \trianglelefteq W \urcorner \in \Omega'_i$). Finally, $\mathcal{P}^{\mathcal{T}} \vdash_{\mathcal{C}} ((p(\overline{e'}_n) == v)\rho \Leftarrow \Pi$ is witnessed by a proof tree T' whose root inference using **AC** has the form

$$\frac{((\ e'_i \to t_i)\rho \Leftarrow \Pi\ )_{i=1\ldots n}}{(p(\overline{e'}_n) == v)\rho \Leftarrow \Pi}$$

and where each premise $(e'_i \to t_i)\rho \Leftarrow \Pi$ is identical to $(e'_i \to t_i)\rho_i \Leftarrow \Pi$ and therefore proved by the CRWL($\mathcal{C}$) proof tree T'_i.

[2. ⇒ 1.] *(Transformation soundness).* Assume $\rho \in \mathrm{Sol}_{\mathcal{C}}(\Omega')$ such that $\mathrm{dom}(\rho) = \mathrm{var}(\Omega')$ and $\mathcal{P}^{\mathcal{T}} \vdash_{\mathcal{C}} \psi'\rho \Leftarrow \Pi$ by means of a CRWL($\mathcal{C}$) proof tree T' with k nodes. Reasoning by induction on k we show the existence of a QCRWL($\mathcal{D},\mathcal{C}$) proof tree T witnessing $\mathcal{P} \vdash_{\mathcal{D},\mathcal{C}} \psi\sharp d \Leftarrow \Pi$. The base case $k = 1$ is easy. For the inductive case $k > 1$ we distinguish cases according to the CRWL($\mathcal{C}$) inference rule **RL** applied at the root of T'. Here we argue only for the case where **RL** is **AC**. In this case ψ, ψ', Ω', the proof tree T' and the subtrees T'_i of T' proving the premises of the **AC** inference at the root of T' have the forms described in the first part of the proof. For each $1 \le i \le n$, let $d_i = d$ and $\rho_i = \rho\upharpoonright\mathrm{var}(\Omega'_i)$. Then $\rho_i \in \mathrm{Sol}_{\mathcal{C}}(\Omega'_i)$ follows from $\rho \in \mathrm{Sol}_{\mathcal{C}}(\Omega')$. Moreover, $(e'_i \to t_i)\rho_i \Leftarrow \Pi$ is identical to the i-th premise of the **AC** inference at the root of T', and therefore $\mathcal{P}^{\mathcal{T}} \vdash_{\mathcal{C}} (e'_i \to t_i)\rho_i \Leftarrow \Pi$ is witnessed by T'_i, which has less than k nodes. By induction hypothesis we can obtain QCRWL($\mathcal{D},\mathcal{C}$) proof trees T_i witnessing $\mathcal{P} \vdash_{\mathcal{D},\mathcal{C}} (e_i \to t_i)\sharp d_i \Leftarrow \Pi$. Since $d = d_i$, the conditions $d \trianglelefteq d_i$ $(1 \le i \le n)$ hold trivially, and T can be built as a QCRWL($\mathcal{D},\mathcal{C}$) proof tree having the form described in the beginning, with the inference rule **QAC** applied at the root and the proof trees T_i witnessing the premisses. □

Using Theorem 2 we can prove that the transformation of goals specified in Figure 3 preserves solutions in the sense of the following result.

Theorem 3. *Let G be a goal for a given QCFLP($\mathcal{D},\mathcal{C}$)-program $\mathcal{P}$. Then, the two following statements are equivalent:*

1. *$\langle \sigma, \mu, \Pi \rangle \in Sol_{\mathcal{P}}(G)$.*
2. *$\langle \sigma \uplus \mu \uplus \rho, \Pi \rangle \in Sol_{\mathcal{P}^{\mathcal{T}}}(G^{\mathcal{T}})$ for some $\rho \in Val_{\mathcal{D}}$ such that $dom(\rho)$ is the set of new variables W introduced by the transformation of G.*

Proof. Let $G = (\ \delta_i\sharp W_i, W_i \trianglerighteq \beta_i\ )_{i=1\ldots m}$, σ and μ be given. For $i = 1\ldots m$, consider $\delta_i{}^{\mathcal{T}} = (\delta'_i, \Omega_i, \mathcal{W}_i)$ and $\Omega'_i = \Omega_i \cup \{\ulcorner W_i \trianglelefteq W \urcorner \mid W \in \mathcal{W}_i\}$. According to Fig. 3, $G^{\mathcal{T}} = (\Omega'_i, \mathsf{qVal}(W_i), \ulcorner W_i \trianglerighteq \beta_i \urcorner, \delta'_i)_{i=1\ldots m}$. Then, because of Def. 3(2) and the analogous notion of solution for CFLP($\mathcal{C}$) goals explained in Sect. 3, the two statements of the theorem can be reformulated as follows:

(a) $W_i\mu \trianglerighteq \beta_i$ and $\mathcal{P} \vdash_{\mathcal{D},\mathcal{C}} \delta_i\sigma\sharp W_i\mu \Leftarrow \Pi$ hold for $i = 1\ldots m$.

(b) There exists $\rho \in \mathrm{Val}_{\mathcal{D}}$ with $\mathrm{dom}(\rho) = \bigcup_{i=1}^{m} \mathrm{var}(\Omega_i)$ such that $\rho \in \mathrm{Sol}_{\mathcal{C}}(\Omega'_i\mu)$, $W_i\mu \trianglerighteq \beta_i$ and $\mathcal{P}^{\mathcal{T}} \vdash_{\mathcal{C}} (\delta'_i\sigma)\rho \Leftarrow \Pi$ hold for $i = 1\ldots m$.

$[(a) \Rightarrow (b)]$ Assume (a). Note that $\delta_i\sigma\sharp W_i\mu \Leftarrow \Pi^{\mathcal{T}}$ is $\delta_i'\sigma \Leftarrow \Pi, \Omega_i'\mu$. Applying Theorem 2 (with $\psi = \delta_i\sigma$, $d = W_i\mu$ and Π) we obtain $\mathcal{P}^{\mathcal{T}} \vdash_{\mathcal{C}} (\delta_i'\sigma)\rho_i \Leftarrow \Pi$ for some $\rho_i \in \mathrm{Sol}_{\mathcal{C}}(\Omega_i'\mu)$ with $\mathrm{dom}(\rho_i) = \mathrm{var}(\Omega_i'\mu) = \mathrm{var}(\Omega_i)$. Then (b) holds for $\rho = \biguplus_{i=1}^{m} \rho_i$.
$[(b) \Rightarrow (a)]$ Assume (b). Let $\rho_i = \rho\restriction\mathrm{var}(\Omega_i)$, $i = 1 \ldots m$. Note that (b) ensures $\mathcal{P}^{\mathcal{T}} \vdash_{\mathcal{C}} (\delta_i'\sigma)\rho_i \Leftarrow \Pi$ and $\rho \in \mathrm{Sol}_{\mathcal{C}}(\Omega_i'\mu)$. Then Theorem 2 can be applied (again with $\psi = \delta_i\sigma$, $d = W_i\mu$ and Π) to obtain $\mathcal{P} \vdash_{\mathcal{D},\mathcal{C}} \delta_i\sigma\sharp W_i\mu \Leftarrow \Pi$. Therefore, (a) holds. $\square$

As an example of goal solving via the transformation, we consider again the *library program* $\mathcal{P}$ and the goal G discussed in the Introduction. Both belong to the instance QCFLP($\mathcal{U}, \mathcal{R}$) of our scheme. Their translation into CFLP($\mathcal{R}$) can be executed in the $\mathcal{TOY}$ system [2] after loading the Real Domain Constraints library (`cflpr`). The source and translated code are publicly available at `gpd.sip.ucm.es/cromdia/qlp`. Solving the transformed goal in $\mathcal{TOY}$ computes the answer announced in the Introduction as follows:

```
Toy(R)> qVal([W]), W>=0.65, search("German","Essay",intermediate,W) == R
      { R -> 4 }
      { W=<0.7, W>=0.65 }
sol.1, more solutions (y/n/d/a) [y]? no
```

The best qualification value for `W` provided by the answer constraints is `0.7`.

5 Conclusions and Future Work

The work in this paper is based on the scheme CFLP($\mathcal{C}$) for functional logic programming with constraints presented in [9]. Our main results are: a new programming scheme QCFLP($\mathcal{D}, \mathcal{C}$) extending the first-order fragment of CFLP($\mathcal{C}$) with qualified computation capabilities; a rewriting logic QCRWL($\mathcal{D}, \mathcal{C}$) characterizing QCFLP($\mathcal{D}, \mathcal{C}$)-program semantics; and a transformation of QCFLP($\mathcal{D}, \mathcal{C}$) into CFLP($\mathcal{C}$) preserving program semantics and goal solutions, that can be used as a correct implementation technique. Existing CFLP($\mathcal{C}$) systems such as $\mathcal{TOY}$ [2] and Curry [7] that use definitional trees as an efficient implementation tool can easily adopt the implementation, since the structure of definitional trees is quite obviously preserved by the transformation.

As argued in the Introduction, our scheme is more expressive than the main related approaches we are aware of. By means of an example dealing with a simplified library, we have shown that instances of QCFLP($\mathcal{D}, \mathcal{C}$) can serve as a declarative language for flexible information retrieval problems, where qualified (rather than exact) answers to user's queries can be helpful.

As future work we plan to extend QCFLP($\mathcal{D}, \mathcal{C}$) and the program transformation in order to provide explicit support for similarity-based reasoning, as well as the higher-order programming features available in CFLP($\mathcal{C}$). We also plan to analyze the complexity of the program transformation and to embed it as part of an enhanced version of the $\mathcal{TOY}$ system. Finally, we plan further research on flexible information retrieval applications, using different instances of our scheme.

References

1. Antoy, S., Echahed, R., Hanus, M.: A needed narrowing strategy. Journal of the ACM 47(4), 776–822 (2000)
2. Arenas, P., Fernández, A.J., Gil, A., López-Fraguas, F.J., Rodríguez-Artalejo, M., Sáenz-Pérez, F.: $\mathcal{TOY}$, a multiparadigm declarative language. version 2.3.1. Caballero, R., Sánchez, J. (eds.) (2007), http://toy.sourceforge.net
3. Caballero, R., Rodríguez-Artalejo, M., Romero-Díaz, C.A.: Similarity-based reasoning in qualified logic programming. In: PPDP 2008: Proceedings of the 10th international ACM SIGPLAN conference on Principles and Practice of Declarative Programming, pp. 185–194. ACM, New York (2008)
4. Caballero, R., Rodríguez-Artalejo, M., Romero-Díaz, C.A.: A generic scheme for qualified constraint functional logic programming. Technical Report SIC-1-09, Universidad Complutense, Departamento de Sistemas Informáticos y Computación, Madrid, Spain (2009), http://gpd.sip.ucm.es/cromdia/works
5. del Vado Vírseda, R.: Declarative constraint programming with definitional trees. In: Gramlich, B. (ed.) FroCos 2005. LNCS, vol. 3717, pp. 184–199. Springer, Heidelberg (2005)
6. Guadarrama, S., Muñoz, S., Vaucheret, C.: Fuzzy prolog: A new approach using soft constraint propagation. Fuzzy Sets and Systems 144(1), 127–150 (2004)
7. Hanus, M.: Curry: an integrated functional logic language, version 0.8.2. Hanus, M. (ed.) (2006), http://www.informatik.uni-kiel.de/~curry/report.html
8. López-Fraguas, F.J., Rodríguez-Artalejo, M., del Vado-Virseda, R.: A lazy narrowing calculus for declarative constraint programming. In: Proceedings of the 6th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP 2004), pp. 43–54. ACM Press, New York (2004)
9. López-Fraguas, F.J., Rodríguez-Artalejo, M., del Vado-Vírseda, R.: A new generic scheme for functional logic programming with constraints. Journal of Higher-Order and Symbolic Computation 20(1-2), 73–122 (2007)
10. Moreno, G., Pascual, V.: Formal properties of needed narrowing with similarity relations. Electronic Notes in Theoretical Computer Science 188, 21–35 (2007)
11. Riezler, S.: Quantitative constraint logic programming for weighted grammar applications. In: Retoré, C. (ed.) LACL 1996. LNCS, vol. 1328, pp. 346–365. Springer, Heidelberg (1997)
12. Riezler, S.: Probabilistic Constraint Logic Programming. PhD thesis, Neuphilologischen Fakultät del Universität Tübingen (1998)
13. Rodríguez-Artalejo, M.: Functional and constraint logic programming. In: Comon, H., Marché, C., Treinen, R. (eds.) CCL 1999. LNCS, vol. 2002, pp. 202–270. Springer, Heidelberg (2001)
14. Rodríguez-Artalejo, M., Romero-Díaz, C.A.: Quantitative logic programming revisited. In: Garrigue, J., Hermenegildo, M.V. (eds.) FLOPS 2008. LNCS, vol. 4989, pp. 272–288. Springer, Heidelberg (2008)
15. Sessa, M.I.: Approximate reasoning by similarity-based SLD resolution. Theoretical Computer Science 275(1-2), 389–426 (2002)
16. Subrahmanian, V.S.: Uncertainty in logic programming: Some recollections. Association for Logic Programming Newsletter 20(2) (2007)

Logic Programs under Three-Valued Łukasiewicz Semantics

Steffen Hölldobler and Carroline Dewi Puspa Kencana Ramli

International Center for Computational Logic,
TU Dresden, 01062 Dresden, Germany
sh@iccl.tu-dresden.de
http://www.computational-logic.org/~sh/

Abstract. If logic programs are interpreted over a three-valued logic, then often Kleene's strong three-valued logic with complete equivalence and Fitting's associated immediate consequence operator is used. However, in such a logic the least fixed point of the Fitting operator is not necessarily a model for the program under consideration. Moreover, the model intersection property does not hold. In this paper, we consider the three-valued Łukasiewicz semantics and show that fixed points of the Fitting operator are also models for the program under consideration and that the model intersection property holds. Moreover, we review a slightly different immediate consequence operator first introduced by Stenning and van Lambalgen and relate it to the Fitting operator under Łukasiewicz semantics. Some examples are discussed to support the claim that Łukasiewicz semantics and the Stenning and van Lambalgen operator is better suited to model commonsense and human reasoning.

Keywords: Three Valued Logic Programs, Łukasiewicz Semantics.

1 Introduction

When interpreting logic programs (with negation) under a three-valued semantics, then it appears that with some exceptions (see e.g. [10]) mainly the semantics defined by Fitting in [7] is considered (see e.g. [1]) in the logic programming literature up to now. This semantics combines Kleene's strong three-valued logic for negation, conjunction, disjunction and implication with complete equivalence, which was also introduced by Kleene (see [13]). Complete equivalence was used by Fitting to ensure that a formula of the form $F \leftrightarrow F$ is mapped to true under an interpretation, which maps F to neither true nor false (see [7], p.300). Under the Fitting semantics, the law of equivalence ($F \leftrightarrow G$ is semantically equivalent to $(F \leftarrow G) \wedge (G \leftarrow F)$) does not hold anymore. This is somewhat surprising as Fitting suggests a completion-based approach ([5]), where the if-halves of the definitions in a logic program are completed by adding their corresponding only-if-halves. Under the Fitting semantics, a completed definition $p \leftrightarrow q$ may be mapped to true under an interpretation, which maps neither $p \leftarrow q$ nor $q \leftarrow p$ to true.

The Fitting semantics was also considered in a recent book by Stenning and van Lambalgen [18], where they argue in favor of a completion-based logic-programming approach to model human reasoning. Stenning and van Lambalgen introduce an immediate consequence operator, which is slightly different from the one defined by Fitting in [7], and claim that for a given propositional logic program the least fixed point of this operator is the minimal model of the program (Lemma 4(1.) in [18]). Looking into this result we found that the least fixed point may not even be a model for the program (see [12]) and that this stems from the fact that the Fitting semantics does not admit the law of equivalence.

From these observations two questions arose: Why did Fitting combine Kleene's strong three-valued logic with complete equivalence? Is there an alternative semantics under which the results proven in [7] hold and which admits also the law of equivalence?

We can answer the former question only partially: questions of computability[1] and, in particular, termination[2] may have been the driving force. As for the latter, we believe that the Łukasiewicz semantics [15] may be a good candidate.

After reviewing three-valued logics in Section 2 and stating some preliminaries in Section 3 we investigate Fitting's immediate consequence operator in Section 4. In particular, we show that under the Łukasiewicz semantics, a fixed point of the Fitting operator is not only a model for the completion of a given program, but for the program itself. Moreover, we show that the model intersection property holds for logic programs (with negation) under the Łukasiewicz semantics.

In Section 5 we review Stenning and van Lambalgen's immediate consequence operator under Łukasiewicz semantics. The main difference between the Fitting and the Stenning and van Lambalgen operator is the observation that whereas Fitting assumes all undefined predicates to be false within the completion process, Stenning and van Lambalgen allow the user to control which otherwise undefined predicates shall be mapped to false. In order to do so, they introduce so-called negative facts and modify the notion of completion accordingly. In Section 6 we present two examples from commonsense and human reasoning to support the claim that the Stenning and van Lambalgen operator may be better suited for these reasoning tasks than the Fitting operator. In the final Section 7 we summarize our findings and point to some future and related work.

2 Three-Valued Logics

In 1920, the Polish philosopher Łukasiewicz introduced the first three-valued logic [15]. The truth values are not only true or false, but there exists a third, intermediate value. A formula is allowed to be neither true nor false. We can interpret the intermediate truth value as possibility: the truth value is not decided yet but possibly decided at some later time. In this paper, we symbolize truth- and falsehood by $\top$ and $\bot$, respectively. We call the third truth value *undecided* and use the symbol u to denote it.

Łukasiewicz used the following principles and definitions to assign values to formulas, where $\equiv$ denotes semantic equivalence:

[1] Personal communication with Melvin Fitting.

[2] Personal communication with Pascal Hitzler.

Table 1. A truth table for three-valued logics. The indices K and Ł refer to Kleene's and Łukasiewicz's logic, respectively. $\leftrightarrow_C$ denotes the complete equivalence used by Fitting.

F	G	$\neg F$	$F \wedge G$	$F \vee G$	$F \leftarrow_K G$	$F \leftrightarrow_K G$	$F \leftrightarrow_C G$	$F \leftarrow_Ł G$	$F \leftrightarrow_Ł G$
$\top$	$\top$	$\bot$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$
$\top$	$\bot$	$\bot$	$\bot$	$\top$	$\top$	$\bot$	$\bot$	$\top$	$\bot$
$\top$	u	$\bot$	u	$\top$	$\top$	u	$\bot$	$\top$	u
$\bot$	$\top$	$\top$	$\bot$	$\top$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
$\bot$	$\bot$	$\top$	$\bot$	$\bot$	$\top$	$\top$	$\top$	$\top$	$\top$
$\bot$	u	$\top$	$\bot$	u	u	u	$\bot$	u	u
u	$\top$	u	u	$\top$	u	u	$\bot$	u	u
u	$\bot$	u	$\bot$	u	$\top$	u	$\bot$	$\top$	u
u	u	u	u	u	u	u	$\top$	$\top$	$\top$

Table 2. Some common laws under Łukasiewicz, Kleene and Fitting semantics

Laws		Łukasiewicz	Kleene	Fitting
Equivalence	$F \leftrightarrow G \equiv (F \rightarrow G) \wedge (G \rightarrow F)$	Yes	Yes	No
Implication	$F \rightarrow G \equiv \neg F \vee G$	No	Yes	Yes
Syllogism	$(F \rightarrow G) \wedge (G \rightarrow H) \equiv F \rightarrow H$	No	Yes	Yes
Excluded Middle	$F \vee \neg F \equiv \top$	No	No	No
Contradiction	$F \wedge \neg F \equiv \bot$	No	No	No

1. The principles of identity and non-identity: $(\bot \leftrightarrow \bot) \equiv (\top \leftrightarrow \top) \equiv \top$, $(\top \leftrightarrow \bot) \equiv (\bot \leftrightarrow \top) \equiv \bot$,
   $(\bot \leftrightarrow u) \equiv (u \leftrightarrow \bot) \equiv (\top \leftrightarrow u) \equiv (u \leftrightarrow \top) \equiv u$, $(u \leftrightarrow u) \equiv \top$.
2. The principles of implication:
   $(\bot \leftarrow \bot) \equiv (\top \leftarrow \bot) \equiv (\top \leftarrow \top) \equiv \top$, $(\bot \leftarrow \top) \equiv \bot$,
   $(u \leftarrow \bot) \equiv (\top \leftarrow u) \equiv (u \leftarrow u) \equiv \top$, $(\bot \leftarrow u) \equiv (u \leftarrow \top) \equiv u$.
3. The definitions of negation, disjunction and conjunction:
   $\neg A \equiv (\bot \leftarrow A)$, $A \vee B \equiv (B \leftarrow (B \leftarrow A))$, $A \wedge B \equiv \neg(\neg A \vee \neg B)$.

Later, in 1952, Kleene proposed an alternative three-valued logic with the truth values true, false, and undefined. He distinguishes between weak and strong three-valued logics. For our paper only the latter is of interest. It is similar to the Łukasiewicz logic, but differs in the semantics of implication and equivalence, viz., $u \leftrightarrow u \equiv u$ and $u \leftarrow u \equiv u$. Kleene also introduced a *complete equivalence* where $(F \leftrightarrow G) \equiv \top$ if and only if both F and G have the same logical value, else $(F \leftrightarrow G) \equiv \bot$.

The semantics of the connectives is summarized in Table 1. In the Łukasiewicz logic [15] the set of connectives is $\{\neg, \wedge, \vee, \leftarrow_Ł, \leftrightarrow_Ł\}$, in Kleene's strong three-valued logic [13] the set of connectives is $\{\neg, \wedge, \vee, \leftarrow_K, \leftrightarrow_K\}$, and in the Fitting logic [7] the set of connectives is $\{\neg, \wedge, \vee, \leftarrow_K, \leftrightarrow_C\}$. Table 2 gives an overview over some common laws which do not always hold with respect to the Łukasiewicz, Kleene and Fitting logics considered in this paper. Other laws like impotency, commutativity, associativity, absorption, distributivity, double negation, de Morgan and contraposition hold under Kleene, Łukasiewicz and Fitting logics.

3 Preliminaries

In this section we recall some notations and terminologies based on [14].

3.1 First-Order Language

We consider an *alphabet* consisting of (finite or countably infinite) disjoint sets of variables, constants, function symbols, predicate symbols, connectives $\{\neg, \vee, \wedge, \leftarrow, \leftrightarrow\}$, quantifiers $\{\forall, \exists\}$, and punctuation symbols {"(", ",", ")"}. In this paper we will use upper case letters to denote variables and lower case letters to denote constants, function- and predicate symbols. Terms, atoms, literals and formulas are defined as usual. To avoid having formulas cluttered with brackets, we adopt the following precedence hierarchy to order the connectives: $\neg > \{\vee, \wedge\} > \leftarrow > \leftrightarrow$. The *language* given by an alphabet consists of the set of all formulas constructed from the symbols occurring in the alphabet. A *sentence* is a formula without free variables. Finally, we extend our language by the symbols $\top$ and $\bot$ denoting a valid and an unsatisfiable formula, respectively.

3.2 Logic Programs

A *(program) clause* is an expression of the form $A \leftarrow B_1 \wedge \cdots \wedge B_n$, where $n \geq 1$, A is an atom, and each B_i, $1 \leq i \leq n$, is either a literal (i.e., atom or negated atom) or $\top$. A is called *head* and $B_1 \wedge \cdots \wedge B_n$ *body* of the clause. One should note that the body of a clause must not be empty. A clause of the form $A \leftarrow \top$ is called a *positive fact*.

A *(logic) program* is a finite set of clauses. *ground*$(\mathcal{P})$ denotes the set of all ground instances of the program $\mathcal{P}$. In many cases, *ground*$(\mathcal{P})$ is infinite, but for propositional or datalog programs *ground*$(\mathcal{P})$ is finite. In the sequel we will consider *ground*$(\mathcal{P})$ as a substitute for $\mathcal{P}$, thus ignoring unification issues.

We assume that each non-propositional program contains at least one constant symbol. Moreover, the language $\mathcal{L}$ underlying a program $\mathcal{P}$ shall contain precisely the relation, function and constant symbols occurring in $\mathcal{P}$, and no others.

3.3 Interpretations and Models

The declarative semantics of a logic program is given by a model-theoretic semantics of formulas in the underlying language. We represent interpretations by pairs $\langle I^\top, I^\bot \rangle$, where the set $I^\top$ contains all atoms which are mapped to $\top$, the set $I^\bot$ contains all atoms which are mapped to $\bot$, and $I^\top \cap I^\bot = \emptyset$. All atoms which occur neither in $I^\top$ nor $I^\bot$ are mapped to u. The logical value of formulas can be derived from Table 1 as usual. We use $I_Ł$, I_K and I_F to denote that an interpretation I uses the Łukasiewicz, Kleene or Fitting semantics, respectively. let $\mathcal{I}$ denote the set of all interpretations. One should observe that $(\mathcal{I}, \subseteq)$ is a complete semi-lattice (see [7]).

Let I be an interpretation of a language $\mathcal{L}$ and let F be a sentence of $\mathcal{L}$. I is a *model* for F if F is true with respect to I (i.e., $I(F) = \top$). Let $\mathcal{S}$ be a set of sentences of a language $\mathcal{L}$ and let I be an interpretation of $\mathcal{L}$. We say I is a *model* for $\mathcal{S}$ if I is a model for each sentence of $\mathcal{S}$. Two sentences F and G are said to be *semantically equivalent* if and only if both have same truth value under all interpretations.

3.4 Program Completion

Let *ground*$(\mathcal{P})$ be a logic program. Consider the following transformation:

1. All clauses with the same head (ground atom) $A \leftarrow Body_1,\ A \leftarrow Body_2, \ldots$ are replaced by the single expression $A \leftarrow Body_1 \vee Body_2 \vee \ldots$.
2. If a ground atom A is not the head of any clause in *ground*$(\mathcal{P})$ then add $A \leftarrow \bot$, where $\bot$ denotes an unsatisfiable formula.
3. All occurrences of $\leftarrow$ are replaced by $\leftrightarrow$.

The resulting set of formulas is called *completion of ground*$(\mathcal{P})$ and is denoted by *comp*$($*ground*$(\mathcal{P}))$. One should observe that in step 1 there may be infinitely many clauses with the same head resulting in a countable disjunction. However, its semantic behavior is unproblematic.

4 The Fitting Operator

In this section we will discuss Fitting's immediate consequence operator [7] under the Łukasiewicz semantics. We will show that replacing the Fitting semantics with the Łukasiewicz semantics does not change the behaviors of the Fitting operator. But in addition each model of the completion of a program coincides with a model of the program itself.

Let I be an interpretation and $\mathcal{P}$ a program. *Fitting's immediate consequence operator* is defined as follows: $\Phi_{F,\mathcal{P}}(I) = \langle J^{\top}, J^{\bot} \rangle$, where

$$\begin{aligned} J^{\top} &= \{A \mid \text{there exists } A \leftarrow Body \in ground(\mathcal{P}) \text{ with } I(Body) = \top\} \text{ and} \\ J^{\bot} &= \{A \mid \text{for all } A \leftarrow Body \in ground(\mathcal{P}) \text{ we find } I(Body) = \bot\}. \end{aligned}$$

Please recall that the body of the program is a conjunction of literals and, hence, $I_{Ł}(Body) = I_K(Body) = I_F(Body)$ according to Table 1.

Fitting shows in [7] that $\Phi_{F,\mathcal{P}}$ is monotone on $(\mathcal{I}, \subseteq)$. Moreover, from [19] and [16] follows that for finite *ground*$(\mathcal{P})$ the operator $\Phi_{F,\mathcal{P}}$ is also continuous. We call a program $\mathcal{P}$ *F-acceptable* if $\Phi_{F,\mathcal{P}}$ is continuous.

Given a program $\mathcal{P}$. An interpretation I is said to be *fixed point of* $\Phi_{F,\mathcal{P}}$ iff $I = \Phi_{F,\mathcal{P}}(I)$. If $\Phi_{F,\mathcal{P}}$ is continuous, then it admits a least fixed point denoted by *lfp*$(\Phi_{F,\mathcal{P}})$. It can be computed by iterating $\Phi_{F,\mathcal{P}}$ starting with the empty interpretation as follows, where ω is an arbitrary limit ordinal:

$$\begin{aligned} \Phi_{F,\mathcal{P}} \uparrow_0 &= \langle \emptyset, \emptyset \rangle\,, \\ \Phi_{F,\mathcal{P}} \uparrow_{(\alpha+1)} &= \Phi_{F,\mathcal{P}}(\Phi_{F,\mathcal{P}} \uparrow_\alpha), \\ \Phi_{F,\mathcal{P}} \uparrow_\omega &= \bigcup \{\Phi_{F,\mathcal{P}} \uparrow_\alpha \mid \alpha < \omega\}. \end{aligned}$$

As examples consider the programs $\mathcal{P}_1 = ground(\mathcal{P}_1) = \{p \leftarrow q\}$ and $\mathcal{P}_2 = ground(\mathcal{P}_2) = \{p \leftarrow q,\ q \leftarrow p\}$. Their completions are $comp(ground(\mathcal{P}_1)) = \{p \leftrightarrow q,\ q \leftrightarrow \bot\}$ and $comp(ground(\mathcal{P}_2)) = \{p \leftrightarrow q,\ q \leftrightarrow p\}$. In both cases, the Fitting operator is continuous and we obtain the least fixed points $lfp(\Phi_{F,\mathcal{P}_1}) = \langle \emptyset, \{p, q\} \rangle$ and $lfp(\Phi_{F,\mathcal{P}_2}) = \langle \emptyset, \emptyset \rangle$. It is easy to verify that the least fixed points are models of the completions under the Fitting semantics, which is no coincidence as formally proven in [7]. This property holds also under the Łukasiewicz semantics.

Proposition 1. *Let $\mathcal{P}$ be a program.*

1. *$I_Ł$ is a fixed point of $\Phi_{F,\mathcal{P}}$ iff $I_Ł$ is a model of* $\text{comp}(\text{ground}(\mathcal{P}))$.
2. *If $I_Ł = \mathit{lfp}(\Phi_{F,\mathcal{P}})$ then $I_Ł$ is the least model of* $\text{comp}(\text{ground}(\mathcal{P}))$.

Proof. 1. To show the if-part, suppose I is a fixed point of $\Phi_{F,\mathcal{P}}$. As shown in [7], in this case I is a model of $\mathit{comp}(\mathit{ground}(\mathcal{P}))$ under the Fitting semantics. Comparing the columns labeled $F \leftrightarrow_C G$ and $F \leftrightarrow_Ł G$ in Table 1 we observe that if $I(F \leftrightarrow_C G) = \top$ then $I(F \leftrightarrow_Ł G) = \top$. Consequently, I is also model for $\mathit{comp}(\mathit{ground}(\mathcal{P}))$ under the Łukasiewicz semantics.
To show the only-if-part, suppose $I_Ł(\mathit{comp}(\mathit{ground}(\mathcal{P}))) = \top$. In this case we have to show that $I_Ł = \langle I^\top, I^\bot \rangle$ is a fixed point of $\Phi_{F,\mathcal{P}}$, i.e., $\Phi_{F,\mathcal{P}}(I_Ł) = I_Ł$. Let $\Phi_{F,\mathcal{P}}(I_Ł) = J = \langle J^\top, J^\bot \rangle$. $J = I$ if and only if $J^\top = I^\top$ and $J^\bot = I^\bot$. We distinguish four cases:
   (a) Suppose $A \in I^\top$, i.e., $I_Ł(A) = \top$. Because $I_Ł(\mathit{comp}(\mathit{ground}(\mathcal{P}))) = \top$ we find $A \leftrightarrow \mathit{Body}_1 \vee \mathit{Body}_2 \vee \ldots \in \mathit{comp}(\mathit{ground}(\mathcal{P}))$ such that $I_Ł(\mathit{Body}_1 \vee \mathit{Body}_2 \vee \ldots) = \top$. Hence, there exists $A \leftarrow \mathit{Body}_i \in \mathit{ground}(\mathcal{P})$, $i \geq 1$, such that $I_Ł(\mathit{Body}_i) = \top$. Therefore, $A \in J^\top$.
   (b) Suppose $A \in J^\top$. By the definition of $\Phi_{F,\mathcal{P}}$, we find $A \leftarrow \mathit{Body}_i \in \mathit{ground}(\mathcal{P})$, $i \geq 1$, such that $I_Ł(\mathit{Body}_i) = \top$. Hence, we find $A \leftrightarrow \mathit{Body}_1 \vee \mathit{Body}_2 \vee \ldots \in \mathit{comp}(\mathit{ground}(\mathcal{P}))$ and $I_Ł(\mathit{Body}_1 \vee \mathit{Body}_2 \vee \ldots) = \top$. Because $I_Ł(\mathit{comp}(\mathit{ground}(\mathcal{P}))) = \top$, we find $I_Ł(A) = \top$. Hence, $A \in I^\top$.
   (c) Suppose $A \in I^\bot$, i.e., $I_Ł(A) = \bot$. Because $I_Ł(\mathit{comp}(\mathit{ground}(\mathcal{P}))) = \top$ we find $A \leftrightarrow F \in \mathit{comp}(\mathit{ground}(\mathcal{P}))$ such that $I_Ł(F) = \bot$. In this case either $F = \bot$ or $F = \mathit{Body}_1 \vee \mathit{Body}_2 \vee \ldots$ and for all $i \geq 1$ we find $I_Ł(\mathit{Body}_i) = \bot$. By definition of $\Phi_{F,\mathcal{P}}$ we find $A \in J^\bot$ in either case.
   (d) Suppose $A \in J^\bot$. By the definition of $\Phi_{F,\mathcal{P}}$ we find for all $A \leftarrow \mathit{Body}_i \in \mathit{ground}(\mathcal{P})$, $i \geq 1$, that $I_Ł(\mathit{Body}_i) = \bot$. Hence, with $F = \bot \vee \mathit{Body}_1 \vee \mathit{Body}_2 \vee \ldots$ we find $I_Ł(F) = \bot$. Because $I_Ł(\mathit{comp}(\mathit{ground}(\mathcal{P}))) = \top$ and $A \leftrightarrow F \in \mathit{comp}(\mathit{ground}(\mathcal{P}))$ we conclude $I_Ł(A) = \bot$. Consequently, $A \in I^\bot$.

2. Suppose $I_Ł = \mathit{lfp}(\Phi_{F,\mathcal{P}})$ and $I_Ł$ is not the least model of $\mathit{comp}(\mathit{ground}(\mathcal{P}))$. Then we find an interpretation $J_Ł$ such that $J_Ł(\mathit{comp}(\mathit{ground}(\mathcal{P}))) = \top$ and $J_Ł \subset I_Ł$. By 1., $J_Ł$ will be a fixed point of $\Phi_{F,\mathcal{P}}$, which contradicts the assumption that $I_Ł$ is the least fixed point of $\Phi_{F,\mathcal{P}}$. □

A fixed point of the Fitting operator under the Fitting semantics is a model of the completion of the program, but it is not necessarily a model of the program itself. Reconsider $\mathcal{P}_2 = \{p \leftarrow q,\ q \leftarrow p\}$. $\mathit{lfp}(\Phi_{F,\mathcal{P}_2}) = \langle \emptyset, \emptyset \rangle$ is not a model for $\mathcal{P}_2$. This is because under Fitting semantics, if p and q are mapped to u, then both implications are mapped to u as well. However, under the Łukasiewicz semantics, if p and q are mapped to u, then both implications are mapped to $\top$. Hence, $\mathit{lfp}(\Phi_{F,\mathcal{P}_2}) = \langle \emptyset, \emptyset \rangle$ is a model for $\mathcal{P}_2$ under the Łukasiewicz semantics.

Proposition 2. *Let $\mathcal{P}$ be a program.*
If $I_Ł(\text{comp}(\text{ground}(\mathcal{P}))) = \top$, then $I_Ł(\text{ground}(\mathcal{P})) = \top$.

Proof. If $I_Ł(\mathit{comp}(\mathit{ground}(\mathcal{P}))) = \top$, then for all $A \leftrightarrow F \in \mathit{comp}(\mathit{ground}(\mathcal{P}))$ we find $I_Ł(A \leftrightarrow F) = \top$. By the law of equivalence we conclude $I_Ł((A \leftarrow F) \wedge (F \leftarrow$

$A)) = \top$ and, consequently, $I_Ł(A \leftarrow F) = \top$. If $F = \bot$ then $ground(\mathcal{P})$ does not contain a clause with head A. Otherwise, $F = Body_1 \vee Body_2 \vee \ldots$ and we distinguish three cases:

1. If $I_Ł(A) = \top$, then we find $I_Ł(A \leftarrow Body_i) = \top$ for all $A \leftarrow Body_i \in ground(P)$.
2. If $I_Ł(A) = \bot$, then for all $i \geq 1$ we find $I_Ł(Body_i) = \bot$ and, consequently, $I_Ł(A \leftarrow Body_i) = \top$ for all $A \leftarrow Body_i \in ground(P)$.
3. If $I_Ł(A) = u$ then either $I_Ł(F) = \bot$ or $I_Ł(F) = u$. The former possibility being similar to case 2. we concentrate on the latter. If $I_Ł(F) = u$ then for at least one i we find $I_Ł(Body_i) = u$ and for all $i \geq 1$ either $I_Ł(Body_i) = u$ or $I_Ł(Body_i) = \bot$. In any case, we find $I_Ł(A \leftarrow Body_i) = \top$ for all $A \leftarrow Body_i \in ground(\mathcal{P})$. □

Corollary 1. *Let $\mathcal{P}$ be a program.*
If $I_Ł$ is a fixed point of $\Phi_{F,\mathcal{P}}$ then $I_Ł(\text{ground}(\mathcal{P})) = \top$.

Proof. The corollary follows immediately from Propositions 1 and 2. □

Although a fixed point of the Fitting operator is not always a model of the given program under the Fitting semantics, the program itself may have models. Returning to the example $\mathcal{P}_2 = \{p \leftarrow q,\ q \leftarrow p\}$, its minimal models under the Fitting semantics are $\langle\emptyset, \{p, q\}\rangle$ and $\langle\{p, q\}, \emptyset\rangle$. Their intersection $\langle\emptyset, \emptyset\rangle$ is no model of $\mathcal{P}_2$ under the Fitting semantics. In other words, the model intersection property does not hold under the Fitting semantics. Under the Łukasiewicz semantics, however, $\langle\emptyset, \emptyset\rangle$ is a model for $\mathcal{P}_2$ and, as we will show in the following, the model intersection property does hold under the Łukasiewicz semantics.

Proposition 3. *Let $\mathcal{P}$ be a program. If $I_Ł = \langle I^\top, I^\bot\rangle$ is a model of* ground($\mathcal{P}$)*, then $I'_Ł = \langle I^\top, \emptyset\rangle$ is also a model of* ground($\mathcal{P}$).

Proof. Let $\mathcal{P}$ be a program. Suppose $I_Ł = \langle I^\top, I^\bot\rangle$ is a model of $ground(\mathcal{P})$. Let $A \leftarrow Body$ be a clause in $ground(\mathcal{P})$. In order to show $I'_Ł(A \leftarrow Body) = \top$ we distinguish three cases:

1. If $A \in I^\top$, then $I'_Ł(A \leftarrow Body) = \top$.
2. If $A \in I^\bot$, then $I_Ł(A) = \bot$ and $I'_Ł(A) = u$. Because $I_Ł(A \leftarrow Body) = \top$ we conclude that $I_Ł(Body) = \bot$. Hence, we find a literal C in $Body$ with $I_Ł(C) = \bot$. For each literal B occurring in $Body$ we find:
   (a) if B is an atom and $B \in I^\top$, then $I_Ł(B) = \top$ and $I'_Ł(B) = \top$,
   (b) if B is an atom and $B \in I^\bot$, then $I_Ł(B) = \bot$ and $I'_Ł(B) = u$,
   (c) if B is an atom and $B \notin I^\top \cup I^\bot$, then $I'_Ł(B) = I_Ł(B) = u$,
   (d) if B is of the form $\neg B'$ and $B' \in I^\top$, then $I_Ł(B) = \bot$ and $I'_Ł(B) = \bot$,
   (e) if B is of the form $\neg B'$ and $B' \in I^\bot$, then $I_Ł(B) = \top$ and $I'_Ł(B) = u$,
   (f) if B is of the from $\neg B'$ and $B' \notin I^\top \cup I^\bot$, then $I'_Ł(B) = I_Ł(B) = u$,
   Because C must belong to either case (b) or (d) and, hence, $I'_Ł(C)$ is either u or $\bot$, we conclude that $I'_Ł(Body)$ is either $\bot$ or u as well. Because $I'_Ł(A) = u$ we conclude that $I'_Ł(A \leftarrow Body) = \top$.
3. If $A \notin I^\top \cup I^\bot$, then $I_Ł(A) = I'_Ł(A) = u$. Because $I_Ł(A \leftarrow Body) = \top$ we distinguish two cases:

(a) If $I_Ł(\mathit{Body}) = \bot$, then we conclude as in case 2. that $I'_Ł(\mathit{Body})$ is either $\bot$ or u and, consequently, $I'_Ł(A \leftarrow \mathit{Body}) = \top$.
(b) If $I_Ł(\mathit{Body}) = u$, then *Body* must contain a literal B with $I_Ł(B) = u$. In this case, $I'_Ł(B) = u$ as well and, consequently, $I'_Ł(\mathit{Body})$ is either $\bot$ or u. As in the previous sub-case we conclude that $I'_Ł(A \leftarrow \mathit{Body}) = \top$. □

As an example consider the program $\mathcal{P}_3 = \{p \leftarrow q \wedge \neg r\}$. In the remainder of this paragraph all models are considered under the Łukasiewicz semantics. $\langle\{p, q\}, \{r\}\rangle$ is a model for $\mathcal{P}_3$, and so is $\langle\{p, q\}, \emptyset\rangle$. $\langle\{p, r\}, \{q\}\rangle$ is a model for $\mathcal{P}_3$, and so is $\langle\{p, r\}, \emptyset\rangle$. $\langle\{r\}, \{q\}\rangle$ is a model for $\mathcal{P}_3$, and so is $\langle\{r\}, \emptyset\rangle$. $\langle\emptyset, \emptyset\rangle$ is the least model of $\mathcal{P}_3$.

Proposition 4. *Let $I_{Ł1} = \left\langle I_1^\top, \emptyset\right\rangle$ and $I_{Ł2} = \left\langle I_2^\top, \emptyset\right\rangle$ be two models for a program $\mathcal{P}$. Then $I_{Ł3} = \left\langle I_1^\top \cap I_2^\top, \emptyset\right\rangle$ is a model for $\mathcal{P}$ as well.*

Proof. Suppose $I_{Ł3} = \left\langle I_3^\top, I_3^\bot\right\rangle = \left\langle I_1^\top \cap I_2^\top, \emptyset\right\rangle$ is not a model for $\mathcal{P}$. Then we find $A \leftarrow \mathit{Body} \in \mathcal{P}$ such that $I_{Ł3}(A \leftarrow \mathit{Body}) \neq \top$. According to Table 1 one of the following cases must hold:

1. $I_{Ł3}(A) = \bot$ and $I_{Ł3}(\mathit{Body}) = \top$.
2. $I_{Ł3}(A) = \bot$ and $I_{Ł3}(\mathit{Body}) = u$.
3. $I_{Ł3}(A) = u$ and $I_{Ł3}(\mathit{Body}) = \top$.

Because $I_3^\bot = \emptyset$ we find $I_{Ł3}(A) \neq \bot$ and, consequently, cases 1. and 2. cannot apply. Therefore, we turn our attention to case 3. If $I_{Ł3}(A) = u$ then there must exist $j \in \{1, 2\}$ such that $I_{Łj}(A) = u$. Because $I_{Łj}$ is a model for $\mathcal{P}$ we find $I_{Łj}(A \leftarrow \mathit{Body}) = \top$ and, thus, $I_{Łj}(\mathit{Body})$ is either u or $\bot$. In this case, $\mathit{Body} \neq \top$. Let $\mathit{Body} = B_1 \wedge \ldots \wedge B_n$ with $n \geq 1$.

Because $I_{Ł3}(\mathit{Body}) = \top$ and $I_3^\bot = \emptyset$ we find for all $1 \leq i \leq n$ that B_i is an atom with $I_{Ł3}(B_i) = \top$. Hence, $\{B_1, \ldots, B_n\} \subseteq I_3^\top$ and, consequently, $\{B_1, \ldots, B_n\} \subseteq I_j^\top$, which contradicts the assumption that $I_{Łj}(\mathit{Body})$ is either u or $\bot$. □

Proposition 4 does not hold for arbitrary models of $\mathcal{P}$. For instance, suppose $\mathcal{P}_4 = \{p \leftarrow q_1 \wedge r_1,\ p \leftarrow q_2 \wedge r_2\}$, $I_{Ł1} = \langle\emptyset, \{p, q_1, r_2\}\rangle$ and $I_{Ł2} = \langle\emptyset, \{p, q_2, r_1\}\rangle$. We can easily show that $I_{Ł1}$ and $I_{Ł2}$ are models for $\mathcal{P}_4$. Their intersection $\langle\emptyset, \{p\}\rangle$, however, is not a model for $\mathcal{P}_4$.

Proposition 5. *Let $\mathcal{M}_Ł$ be the set of all models of a program $\mathcal{P}$ under the Łukasiewicz semantics. Then, $\bigcap \mathcal{M}_Ł$ is a model for $\mathcal{P}$ as well.*

Proof. The result follows immediately from Propositions 3 and 4. □

The least model of $\mathcal{P}_4$ under the Łukasiewicz semantics is $\langle\emptyset, \emptyset\rangle$, whereas the least model of $\mathcal{P}_5 = \{p \leftarrow \top,\ q \leftarrow p,\ r \leftarrow q \wedge \neg s\}$ under the Łukasiewicz semantics is $\langle\{p, q\}, \emptyset\rangle$. The last example also exhibits that the least fixed point of the Fitting operator is not necessarily the least model of the underlying program because $\mathit{lfp}(\Phi_{F,\mathcal{P}_4}) = \langle\{p, q, r\}, \{s\}\rangle$.

5 The Stenning and van Lambalgen Operator

In their quest for models of human reasoning Stenning and van Lambalgen [18] have introduced an immediate consequence operator for propositional programs, which differs slightly from the Fitting operator. Here, we extend the operator to first-order programs. Let I be an interpretation and $\mathcal{P}$ be a program. *Stenning and van Lambalgen's immediate consequence operator* is defined as follows: $\Phi_{SvL,\mathcal{P}}(I) = \langle J^\top, J^\bot \rangle$, where

$$\begin{aligned} J^\top &= \{A \mid \text{there exists } A \leftarrow \mathit{Body} \in \mathit{ground}(\mathcal{P}) \text{ with } I(\mathit{Body}) = \top\} \text{ and} \\ J^\bot &= \{A \mid \text{there exists } A \leftarrow \mathit{Body} \in \mathit{ground}(\mathcal{P}) \text{ and} \\ &\qquad \text{for all } A \leftarrow \mathit{Body} \in \mathit{ground}(\mathcal{P}) \text{ we find } I(\mathit{Body}) = \bot\} \end{aligned}$$

and the difference to the Fitting operator has been highlighted. Stenning and van Lambalgen consider programs under the Fitting semantics. In addition, Stenning and van Lambalgen allow so-called *negative facts* of the form $A \leftarrow \bot$ as program clauses. An *extended (logic) program* is a finite set of clauses and negative facts.

Stenning and van Lambalgen show in [18] that $\Phi_{SvL,\mathcal{P}}$ is monotone on $(\mathcal{I}, \subseteq)$. Moreover, from [19] and [16] follows that for finite $\mathit{ground}(\mathcal{P})$ the operator $\Phi_{SvL,\mathcal{P}}$ is also continuous. We call a program $\mathcal{P}$ *SvL-acceptable* if $\Phi_{SvL,\mathcal{P}}$ is continuous.

If $\Phi_{SvL,\mathcal{P}}$ is continuous then we can compute the least fixed point of $\Phi_{SvL,\mathcal{P}}$ by iterating $\Phi_{SvL,\mathcal{P}}$ starting from empty interpretation. Let I be the least fixed point of $\Phi_{SvL,\mathcal{P}}$ and let

$$I_0 = \langle \emptyset, \emptyset \rangle \tag{1}$$

$$I_\alpha = \Phi_{SvL,\mathcal{P}}(I_{\alpha-1}) \text{ for every non-limit ordinal } \alpha > 0 \tag{2}$$

$$I_\alpha = \bigcup_{\beta<\alpha} I_\beta \text{ for every limit ordinal } \alpha \tag{3}$$

Then for some ordinal ω we find $I = I_\omega$.

Before discussing further properties of the new operator we reconsider $\mathcal{P}_1 = \{p \leftarrow q\}$. Its completion is $\mathit{comp}(\mathit{ground}(\mathcal{P}_1)) = \{p \leftrightarrow q,\ q \leftrightarrow \bot\}$. $\Phi_{SvL,\mathcal{P}}$ admits a least fixed point for $\mathcal{P}_1$ and we obtain $\mathit{lfp}(\Phi_{SvL,\mathcal{P}_1}) = \langle \emptyset, \emptyset \rangle$. One should note that this result differs from $\mathit{lfp}(\Phi_{F,\mathcal{P}_1}) = \langle \emptyset, \{p, q\} \rangle$. Now consider $\mathcal{P}_1' = \{p \leftarrow q,\ q \leftarrow \bot\}$. Its completion is $\mathit{comp}(\mathit{ground}(\mathcal{P}_1')) = \{p \leftrightarrow q,\ q \leftrightarrow \bot\} = \mathit{comp}(\mathit{ground}(\mathcal{P}_1))$ and $\mathit{lfp}(\Phi_{SvL,\mathcal{P}_1'}) = \mathit{lfp}(\Phi_{F,\mathcal{P}_1}) = \langle \emptyset, \{p, q\} \rangle$. Thus, by adding negative facts, Stenning and van Lambalgen's operator can simulate Fitting's operator. But it is more liberal in that if there is no clause with head A in the extended program, then its meaning remains undefined.

Obviously, completion as defined in Section 3.4 is unsuitable for extended programs $\mathcal{P}$. If we omit step 2. in the completion transformation, then the resulting set of formulas is called *weak completion of* $\mathit{ground}(\mathcal{P})$ and is denoted by $\mathit{wcomp}(\mathit{ground}(\mathcal{P}))$. Returning to the examples, we find $\mathit{wcomp}(\mathit{ground}(\mathcal{P}_1)) = \{p \leftrightarrow q\}$ and $\mathit{wcomp}(\mathit{ground}(\mathcal{P}_1'))$ $= \{p \leftrightarrow q,\ q \leftrightarrow \bot\}$.

In the following we relate the Stenning and van Lambalgen operator and weak completion under the Łukasiewicz semantics.

Lemma 1. *Let $I_Ł$ be the least fixed point of $\Phi_{SvL,\mathcal{P}}$ and $J_Ł$ be a model of* wcomp(ground($\mathcal{P}$)) *then $I_Ł \subseteq J_Ł$.*

Proof. Let $I_Ł = \langle I^\top, I^\perp \rangle$ be the least fixed point of $\Phi_{SvL,\mathcal{P}}$ and $J_Ł = \langle J^\top, J^\perp \rangle$ be a model of *wcomp*(*ground*($\mathcal{P}$)). $I_Ł \subseteq J_Ł$ iff $I^\top \subseteq J^\top$ and $I^\perp \subseteq J^\perp$ iff the following propositions hold: (i) if $I_Ł(A) = \top$, then $J_Ł(A) = \top$ and (ii) if $I_Ł(A) = \perp$, then $J_Ł(A) = \perp$. By transfinite induction it can be shown that for every ordinal α and every atom A we find: (iii) if $I_\alpha(A) = \top$, then $J_Ł(A) = \top$ and (iv) if $I_\alpha(A) = \perp$, then $J_Ł(A) = \perp$. The claim follows immediately by the definition of least fixed point of $\Phi_{SvL,\mathcal{P}}$ because it implies that there is an ordinal ω such that $I_Ł = I_\omega$. □

Proposition 6. *Let $\mathcal{P}$ be an extended program. If $I_Ł$ is the least fixed point of $\Phi_{SvL,\mathcal{P}}$, then $I_Ł$ is a minimal model of* wcomp(ground($\mathcal{P}$)).

Proof. First we will show that $I_Ł$ is a model of *wcomp*(*ground*($\mathcal{P}$)). Let's pick an arbitrary formula $(A \leftrightarrow F) \in$ *wcomp*(*ground*($\mathcal{P}$)). In order to show that $I_Ł(A \leftrightarrow F) = \top$ we consider three cases according to the truth value of A in $I_Ł$:

a) If $I_Ł(A) = \top$, then according to the definition of $\Phi_{SvL,\mathcal{P}}$, there exists a clause $(A \leftarrow Body_i) \in ground(\mathcal{P})$ such that $I_Ł(Body_i) = \top$. Because $Body_i$ is one of the disjuncts of F, this implies $I_Ł(F) = \top$ and hence $I_Ł(A \leftrightarrow F) = \top$.
b) If $I_Ł(A) = \perp$, then according to the definition of $\Phi_{SvL,\mathcal{P}}$, there is a clause $(A \leftarrow Body_i) \in ground(\mathcal{P})$ and for every clause $(A \leftarrow Body_i) \in ground(\mathcal{P})$ we have $I_Ł(Body_i) = \perp$ for all i. Consequently, all disjuncts in F are false under $I_Ł$ and, therefore, $I_Ł(F) = \perp$. Hence, $I_Ł(A \leftrightarrow F) = \top$.
c) If $I_Ł(A) = u$, then according to the definition of $\Phi_{SvL,\mathcal{P}}$ there is no clause $(A \leftarrow Body_i) \in ground(\mathcal{P})$ with $I_Ł(Body_i) = \top$ and there are some clauses $(A \leftarrow Body_j) \in ground(\mathcal{P})$ with $I_Ł(Body_j) \neq \perp$. So none of the disjuncts in F is true, but it is also not the case that all of them are false. Therefore $I_Ł(F) = u$ and $I_Ł(A \leftrightarrow F) = \top$.

To prove that $I_Ł$ is a minimal model of *wcomp*(*ground*($\mathcal{P}$)), let $I_Ł = \langle I_Ł^\top, I_Ł^\perp \rangle$. By Lemma 1 we learn that any model $J_Ł = \langle J_Ł^\top, J_Ł^\perp \rangle$ of *wcomp*(*ground*($\mathcal{P}$)) will be such that $I_Ł^\top \subseteq J_Ł^\top$ and $I_Ł^\perp \subseteq J_Ł^\perp$. Hence, no proper subset of $I_Ł$ can be a model of *wcomp*(*ground*($\mathcal{P}$)). Consequently, $I_Ł$ is a minimal model of *wcomp*(*ground*($\mathcal{P}$)). □

Proposition 7. *Let $\mathcal{P}$ be an extended program. If $I_Ł$ is a minimal model of* wcomp(ground($\mathcal{P}$)), *then $I_Ł$ is the least fixed point of $\Phi_{SvL,\mathcal{P}}$.*

Proof. Let $I_Ł = \langle I_Ł^\top, I_Ł^\perp \rangle$ be a minimal model of *wcomp*(*ground*($\mathcal{P}$)) and let $J_Ł = \langle J_Ł^\top, J_Ł^\perp \rangle$ be the least fixed point of $\Phi_{SvL,\mathcal{P}}$. By Lemma 1 we know that $J_Ł^\top \subseteq I_Ł^\top$ and $J_Ł^\perp \subseteq I_Ł^\perp$. Further, by Proposition 6 we have that $J_Ł$ is a minimal model of *wcomp*(*ground*($\mathcal{P}$)). But then it must be the case that $I_Ł = J_Ł$ because otherwise we have a conflict with the minimality of $I_Ł$. □

Corollary 2. *Let $\mathcal{P}$ be an extended program. $I_Ł$ is the least fixed point of $\Phi_{SvL,\mathcal{P}}$ iff $I_Ł$ is the least model of* wcomp(ground($\mathcal{P}$)).

Proof. Follows from Propositions 6 and 7 and the fact that the least fixed point of $\Phi_{SvL,\mathcal{P}}$ is unique. □

One should observe, that Corollary 2 does not hold if we consider $comp(ground(\mathcal{P}))$ and the Fitting semantics instead of the Łukasiewicz semantics. As an example consider again $\mathcal{P}_1 = \{p \leftarrow q\}$ and let $I = \langle\emptyset, \{p,q\}\rangle$. I_F is a model for $comp(\mathcal{P}_1)$, but $\Phi_{SvL,\mathcal{P}_1}(I) = \langle\emptyset, \{p\}\rangle \neq I$. This is counter example for Lemma 4(3) in [18].

Proposition 8. *Let $\mathcal{P}$ be an extended program.*
If $I_Ł(\text{wcomp}(\text{ground}(\mathcal{P}))) = \top$, then $I_Ł(\text{ground}(\mathcal{P})) = \top$.

Proof. If $I_Ł(wcomp(ground(\mathcal{P}))) = \top$, then for all $A \leftrightarrow F \in wcomp(ground(\mathcal{P}))$ we find $I_Ł(A \leftrightarrow F) = \top$. By the law of equivalence we conclude $I_Ł((A \leftarrow F) \wedge (F \leftarrow A)) = \top$ and, consequently, $I_Ł(A \leftarrow F) = \top$. Let $F = Body_1 \vee Body_2 \vee \ldots$. We distinguish three cases:

1. If $I_Ł(A) = \top$, then we find $I_Ł(A \leftarrow Body_i) = \top$ for all $A \leftarrow Body_i \in ground(P)$.
2. If $I_Ł(A) = \bot$, then for all $i \geq 1$ we find $I_Ł(Body_i) = \bot$ and, consequently, $I_Ł(A \leftarrow Body_i) = \top$ for all $A \leftarrow Body_i \in ground(P)$.
3. If $I_Ł(A) = u$ then either $I_Ł(F) = \bot$ or $I_Ł(F) = u$. The former possibility being similar to case 2. we concentrate on the latter. If $I_Ł(F) = u$ then we find an i with $I_Ł(Body_1) = u$ and for all $i \geq 1$ either $I_Ł(Body_i) = u$ or $I_Ł(Body_i) = \bot$. In any case, we find $I_Ł(A \leftarrow Body_i) = \top$ for all $A \leftarrow Body_i \in ground(\mathcal{P})$. □

From Proposition 6 and Proposition 8 we can derive Corollary 3 for the Stenning and Lambalgen operator.

Corollary 3. *Let $\mathcal{P}$ be an extended program.*
If $I_Ł$ is the least fixed point of $\Phi_{SvL,\mathcal{P}}$ then $I_Ł(\text{ground}(\mathcal{P})) = \top$.

Proof. The corollary follows immediately from Propositions 6 and 8. □

One should observe that contrary to Lemma 4(1.) of [18] this corollary does not hold under the Fitting semantics. Reconsider $\mathcal{P}_1 = \{p \leftarrow q\}$, then $lfp(\Phi_{SvL,\mathcal{P}_1}) = \langle\emptyset,\emptyset\rangle$ and, thus, both p and q are mapped to u. Under this interpretation $\mathcal{P}_1$ is mapped to u as well. One should also note that the least fixed point of the Stenning and van Lambalgen operator for a given program $\mathcal{P}$ is not necessarily the least model of $\mathcal{P}$ under the Fitting semantics. Reconsidering $\mathcal{P}_1' = \{p \leftarrow q,\ q \leftarrow \bot\}$ we find $lfp(\Phi_{SvL,\mathcal{P}_1'}) = \langle\emptyset, \{p,q\}\rangle$ whereas the least model of $\mathcal{P}_1'$ under the Łukasiewicz semantics is $\langle\emptyset,\emptyset\rangle$.

6 Two Examples

In this section we present two examples to illustrate the difference between the Fitting and the Stenning and van Lambalgen operator. Suppose we want to model an agent driving a car. One rule would be that he may cross an intersection if the traffic light shows green and there is no unusual situation:

$$cross \leftarrow green, \neg unusual\_situation.$$

An unusual situation occurs if an ambulance wants to cross the intersection from a different direction:

$$unusual\_situation \leftarrow ambulance\_crossing.$$

In addition, suppose that the green light is indeed on:

$$green \leftarrow \top.$$

Let $\mathcal{P}_6$ be the set of these clauses. It is easy to see that

$$\mathit{lfp}(\Phi_{F,\mathcal{P}_6}) = \langle\{green, cross\}, \{unusual\_situation, ambulance\_crossing\}\rangle\,.$$

Hence, not knowing anything about an ambulance, our agent will assume that no ambulance is present, hit the accelerator, and speed into the intersection. One should observe that not knowing anything about an ambulance may be caused by the fact that the agent's camera is blurred or the agent's microphone is damaged. His assumption that no ambulance is present is made by default. On the other hand,

$$\mathit{lfp}(\Phi_{SvL,\mathcal{P}_6}) = \langle\{green\}, \emptyset\}\rangle\,.$$

In this case, the agent doesn't know whether he may cross the intersection. Inspecting his rules he may find that in order to satisfy the conditions for the first rule, he must verify that no ambulance is crossing. In doing so, he may extend $\mathcal{P}_6$ to $\mathcal{P}_6' = \mathcal{P}_6 \cup \{ambulance\_crossing \leftarrow \bot\}$ yielding

$$\mathit{lfp}(\Phi_{SvL,\mathcal{P}_{6'}}) = \langle\{green, cross\}, \{unusual\_situation, ambulance\_crossing\}\rangle\,.$$

Now, the agent can safely cross the intersection.

The second example is taken from [4]. Byrne has confronted individuals with sentences like *If Marian has an essay to write, she will study late in the library. She does not have an essay to write. If she has textbooks to read, she will study late in the library.* The individuals are then asked to draw conclusions. In this example, only 4% of the individuals conclude that Marian will not study late in the library. Although Byrne uses these and similar examples to conclude that (classical) logic is inadequate for human reasoning, Stenning and van Lambalgen have argued in [18] that the use of three-valued logic programs under completion semantics is indeed adequate for human reasoning. They represent the scenario by

$$\mathcal{P}_7 = \{l \leftarrow e \wedge \neg ab_1,\ e \leftarrow \bot,\ ab_1 \leftarrow \bot,\ l \leftarrow t \wedge \neg ab_2, ab_2 \leftarrow \bot\},$$

where l denotes that Marian will study late in the library, e denotes that she has an essay to write, t denotes that she has a textbook to read, and ab denotes abnormality. In this case, we find $\mathit{lfp}(\Phi_{SvL,\mathcal{P}_7}) = \langle\emptyset, \{ab_1, ab_2, e\}\rangle$, from which we conclude that it is unknown whether Marian will study late in the library. On the other hand, $\mathit{lfp}(\Phi_{F,\mathcal{P}_7}) = \langle\emptyset, \{ab_1, ab_2, e, t, l\}\rangle$. Using the Fitting operator one would conclude that Marian will not study late in the library. Thus, this operator leads to a wrong answer with respect to the discussed scenario from human reasoning, whereas the Stenning and van Lambalgen operator does not.

Table 3. A comparison between the Fitting and the Łukasiewicz semantics for logic programs. We have highlighted the results which were obtained by formal proofs or by counter examples in this paper. The result marked by † was formally proven in [7]. The result marked by * was not proven formally in [18] nor in this paper, but we conjecture that it holds.

Property	Fitting	Łukasiewicz
Model Intersection	No	Yes
Fixed points of $\Phi_{F,\mathcal{P}}$ are models of $comp(ground(\mathcal{P}))$	Yes†	Yes
Fixed points of $\Phi_{F,\mathcal{P}}$ are models of $\mathcal{P}$	No	Yes
The least fixed point of $\Phi_{SvL,\mathcal{P}}$ is the least model of $wcomp(ground(\mathcal{P}))$	Yes*	Yes
The least fixed point of $\Phi_{SvL,\mathcal{P}}$ is a model of $\mathcal{P}$	No	Yes

7 Conclusion

Table 3 compares the Fitting and Łukasiewicz semantics for logic programs as discussed in this paper. In [18] many more examples are given to support the claim that human reasoning can be adequately modelled using completion-based propositional logic programs and the Stenning and van Lambalgen operator. Here, we have extended this approach to first-order programs and have given rigorous proofs of some of the properties of the operator under Łukasiewicz semantics.

Naish in [17] considers yet another three-valued semantics, which differs from the Fitting and Łukasiewics semantics studied in this paper as far as the truth table for the implication is concerned. Although Naish shows several model intersection results for his logic, these results do not subsume our model intersection result nor is our result an immediate consequence of Naish's results. Likewise, Naish introduces new immediate consequence operators, but they differ from the Stenning and van Lambalgen operator studied in this paper and, again, the results by Naish do not subsume our results nor are our results immediate consequences of Naish's results. There is an underlying reason for the differences: Naish focuses on programming and debugging, whereas the work by Stenning and van Lambalgen, which underlies this paper, focuses on human reasoning.

In recent years, the Fitting semantics for logic programs has not been used much. It has been overtaken in interest by the well-founded semantics [20] and stable model semantics [9]. The latter extends the former in a well-understood manner, and provides a two-valued semantics for logic programs. Both capture transitive closure and other recursive rule behavior and, thus, are useful for programming. However, there are trade-offs between the Fitting semantics and well-founded semantics. The ability of well-founded semantics to capture properties like graph reachability means that it cannot be modelled by a finite first-order theory such as completion. Well-founded semantics also has a higher complexity than the Fitting semantics. The relationship of the Fitting semantics and the well-founded semantics is brought forward in [11] using level mappings. These are mappings from Herbrand bases to ordinals, i.e., they induce orderings on the set of ground atoms while disallowing infinite descending chains. The result shows that well-founded semantics is a stratified version of the Fitting semantics.

It has been argued recently in [18] that a completion-based approach captures many aspects of commonsense reasoning. Unlike most approaches to logically modelling commonsense reasoning which rely on introspection to characterize common sense, Stenning and van Lambalgan base their model on the large corpus of cognitive science. The result is already helping logic programming to be re-examined in fields such as medical decision-making.

In [18] and [12] connectionist implementations of the Stenning and van Lambalgen operator are given. The latter is based on the core method (connectionist model generation using recurrent networks with feed-forward core, see e.g. [2]), which has been applied to propositional, first-order, multi-valued as well as modal logic programs (see e.g. [3,6]).

The role of negative facts in extended logic programs needs to be discussed. The name *negative fact* is considered only with respect to the (weak) completion of a program as, otherwise, a negative fact like $A \leftarrow \bot$ is also mapped to true by interpretations which map A to u or $\top$. If in addition a program contains a clause with head A, then negative facts can be eliminated without changing the semantics of the program. This is hardly the intention of a negative fact in human reasoning, where an individual may gather some support for a fact as well as its negation. An alternative idea would be to add $\bot \leftarrow A$ to a program and treat this as a constraint, but this needs to be investigated in the future.

We would like to find a syntactic characterization of SvL-acceptability and relate it to corresponding characterizations of F-acceptability. Likewise, we would like to find conditions under which the Stenning and van Lambalgen operator is a contraction and relate it to corresponding findings with respect to the Fitting operator (see [8]).

Last but not least it remains to be seen which semantics is better suited for logic programming, common sense as well as human reasoning. It appears that the Łukasiewicz semantics has nicer theoretical properties, but we still have to investigate how this semantics relates to questions concerning computability and termination. It also appears that the Łukasiewicz semantics gives more flexibility than the Fitting semantics concerning common sense reasoning problems. As far as human reasoning is concerned we would like to find out how individuals treat implications where the premise as well as the conclusion are undefined as this is the distinctive feature between the Łukasiewicz and the Fitting semantics.

Acknowledgement. The authors would like to thank Bertram Fronhöfer for many fruitful discussions and the anonymous referees for many useful comments and suggestions.

References

1. Apt, K.R., Pedreschi, D.: Reasoning about termination of pure Prolog programs. In: Information and Computation (1993)
2. Bader, S., Hölldobler, S.: The core method: Connectionist model generation. In: Kollias, S.D., Stafylopatis, A., Duch, W., Oja, E. (eds.) ICANN 2006. LNCS, vol. 4132, pp. 1–13. Springer, Heidelberg (2006)
3. Bader, S., Hitzler, P., Hölldobler, S.: Connectionist model generation: A first-order approach. Neurocomputing 71, 2420–2432 (2008)
4. Byrne, R.M.J.: Suppressing valid inferences with conditionals. Cognition 31, 61–83 (1989)
5. Clark, K.L.: Negation as failure. In: Gallaire, H., Minker, J. (eds.) Logic and Databases, pp. 293–322. Plenum, New York (1978)

6. d'Avila Garcez, A.S., Broda, K., Gabbay, D.M.: Neural-Symbolic Learning Systems: Foundations and Applications. Springer, Heidelberg (2002)
7. Fitting, M.: A Kripke–Kleene semantics for logic programs. Journal of Logic Programming 2(4), 295–312 (1985)
8. Fitting, M.: Metric methods – three examples and a theorem. Journal of Logic Programming 21(3), 113–127 (1994)
9. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: Proceedings of the International Joint Conference and Symposium on Logic Programming, pp. 1070–1080. MIT Press, Cambridge (1988)
10. Hitzler, P., Seda, A.K.: Characterizations of classes of programs by three-valued operators. In: Gelfond, M., Leone, N., Pfeifer, G. (eds.) LPNMR 1999. LNCS, vol. 1730, pp. 357–371. Springer, Heidelberg (1999)
11. Hitzler, P., Wendt, M.: The well-founded semantics is a stratified fitting semantics. In: Jarke, M., Koehler, J., Lakemeyer, G. (eds.) KI 2002. LNCS, vol. 2479, pp. 205–221. Springer, Heidelberg (2002)
12. Hölldobler, S., Kencana Ramli, C.D.: Logics and networks for human reasoning. Technical report, International Center for Computational Logic, TU Dresden (submitted, 2009)
13. Kleene, S.C.: Introduction to Metamathematics. North-Holland, Amsterdam (1952)
14. Lloyd, J.W.: Foundations of Logic Programming. Springer, Berlin (1993)
15. Łukasiewicz, J.: O logice trójwartościowej. Ruch Filozoficzny 5, 169–171 (1920); On Three-Valued Logic. In: Borkowski, L. (ed.) Jan Łukasiewicz Selected Works, pp. 87–88. North Holland (1990) (English translation)
16. Mycroft, A.: Logic programs and many-valued logic. In: Fontet, M., Mehlhorn, K. (eds.) STACS 1984. LNCS, vol. 166, pp. 274–286. Springer, Heidelberg (1984)
17. Naish, L.: A three-valued semantics for logic programmers. Theory and Practice of Logic Programming 6(5), 509–538 (2006)
18. Stenning, K., van Lambalgen, M.: Human Reasoning and Cognitive Science. MIT Press, Cambridge (2008)
19. Stoy, J.E.: Denotational Semantics. MIT Press, Cambridge (1977)
20. van Gelder, A., Ross, K.A., Schlipf, J.S.: The well-founded semantics for general logic programs. Journal of the ACM 38, 620–650 (1991)

Execution Control for CHR
PhD Summary

Leslie De Koninck[*]

Department of Computer Science, K.U.Leuven, Belgium
Leslie.DeKoninck@cs.kuleuven.be

Abstract. This is a summary of the PhD of the author [1], which deals with the topic of execution control for Constraint Handling Rules.

1 Introduction

Constraint Handling Rules (CHR) [2] is a rule-based language, designed for the implementation of solvers for special-purpose constraints in terms of general-purpose constraints for which solvers are readily available. CHRs are rules that operate on a (multi-)set of *user-defined* constraints and either *simplify* them into more basic, *built-in* constraints, or *propagate* implied constraints. CHRs can be *multi-headed*, that is, they may apply to a combination of user-defined constraints, and have an optional *guard* which may prevent a rule instance from firing. A classic example CHR program is given next.

Example 1 (`leq`*).* The CHR program given below, called `leq`, implements a solver for the $\leq$ constraint in terms of the built-in equality constraint (=). Our version is slightly different from the usual description in that we assume that CHR constraints are kept in a set rather than a multi-set, and as a consequence, the `idempotence` rule is no longer needed.

```
reflexivity  @ X ≤ X <=> true.
antisymmetry @ X ≤ Y, Y ≤ X <=> X = Y.
transitivity @ X ≤ Y, Y ≤ Z ==> X ≤ Z.
```

The first rule has as name `reflexivity` (before `@`) and is a *simplification* rule that states that any user-defined constraint of the form $a \leq a$ can be 'simplified' to (i.e. replaced by) `true`. The second rule, `antisymmetry`, states that two constraints $a \leq b$ and $b \leq a$ can be replaced with the built-in constrain $a = b$, constraining the arguments to be equal. Finally, rule `transitivity` is a *propagation* rule, which says given constraints $a \leq b$ and $b \leq c$ we can add a new constraint $a \leq c$ without removing anything. The constraints before the arrow are called the *heads* of the rule; those after the arrow the rule *body*. □

CHR is very flexible for specifying a constraint solvers' logic using language concepts such as multi-headed rules, guards and multi-set semantics. However,

[*] Research funded by a PhD grant of the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen).

as will be shown next, flexible execution control is almost completely lacking. In Sect. 2, we extend CHR with rule priorities into CHR^{rp} to deal with this problem. Sect. 3 relates CHR^{rp} with the Logical Algorithms formalism by Ganzinger and McAllester [3]. In Sect. 4, we add search to CHR^{rp}. Sect. 5 concludes.

2 CHR with Rule Priorities

In general, given a set of constraints to be solved, multiple rule instances may apply. For example, given the constraints $\{a \leq b, b \leq a\}$ we can apply either the `antisymmetry` rule which replaces both constraints by $a = b$, or we could apply the `transitivity` rule which adds the constraint $a \leq a$. We can also observe that some choices of rule instances lead to a solution, i.e. a state in which no more rules apply, faster than others. Therefore, it is essential to be able to control which rule instance is applied next in every state of the execution.

Most current implementations of CHR follow the *refined operational semantics* of CHR [4], which is based on the concept of an *active constraint.* The active constraint is a user-defined constraint that traverses all occurrences of its constraint symbol (e.g. $\leq /2$) in the program in textual order, and tries to find a rule instance in which it matches the occurrence under consideration. This may require looking up *partner constraints* that match with the remaining heads of the rule at hand. After applying a rule, the active constraint may be removed and other constraints may be activated.

By cleverly formulating the logic of a CHR solver, one can obtain a desired execution strategy by relying on the refined operational semantics. However, it is neither flexible nor desirable to encode the execution control in the program logic. Therefore, we propose a more high-level approach to execution control that clearly separates the logic and control aspects of a CHR solver. In particular, we extend CHR with rule priorities into CHR^{rp} [5]. In CHR^{rp} a rule instance is applicable if it is in regular CHR and no higher priority rule instance is.

Example 2 (Dijkstra's Shortest Path). A CHR^{rp} implementation of Dijkstra's single-source shortest path algorithm is given below.

```
  1 :: source(V) ==> dist(V,0).
  1 :: dist(V,D₁) \ dist(V,D₂) <=> D₁ =< D₂ | true.
D+2 :: dist(V,D), edge(V,C,U) ==> dist(U,D+C).
```

The input consists of a set of directed weighted edges, represented as `edge`/3 constraints where the first and last arguments respectively denote the begin and end nodes, and the middle argument represents the weight. The source node is given by the `source`/1 constraint. The algorithm keeps track of upper-bounds on the shortest path distances to the different nodes, represented by `dist`/2 constraint whose arguments are respectively the node in question and the distance. Eventually, the distance upper-bounds become tight.

The first rule initiates the algorithm by creating a zero distance upper-bound to the source node. The second rule removes redundant distance bounds. Both these rules have a *static* priority of 1 (before the : :). Finally, the last rule has a *dynamic* priority, that is, it depends on the actual constraints that form an instance of the rule. It generates new distance upper-bounds. The priorities ensure these upper-bounds are created in the order required by Dijkstra's algorithm.

In [6], we presented an optimized implementation of CHR$^{\text{rp}}$. The basic compilation schema is based on that for the refined semantics of CHR, extended with support for priorities. Many optimizations that apply to regular CHR are still possible in CHR$^{\text{rp}}$. For instance, we applied to CHR$^{\text{rp}}$ the important *late storage* optimization of CHR which postpones the storage of a constraint, and refined it further. In our experience, many of the analyses required for the optimizations were considerably simpler in CHR$^{\text{rp}}$ because rule priorities are more global than rule order in the refined operational semantics, which is only obeyed by the active constraint.

3 Logical Algorithms: A CHR$^{\text{rp}}$ Meta-complexity Result

Logical Algorithms [3] is a formalism proposed by Ganzinger and McAllester for the purpose of facilitating the derivation of the complexity of algorithms stated in a high-level logical language. The formalism consists of a theoretical bottom-up LP language (LA), accompanied by a meta-complexity theorem. In [7], we investigated the differences and correspondences between the Logical Algorithms language and CHR$^{\text{rp}}$: LA operates on a set of ground facts, whereas CHR operates on a multi-set of possibly non-ground constraints; LA supports a form of monotonic negation with negative heads and negated facts; CHR$^{(\text{rp})}$ makes use of a built-in constraint theory, whereas LA has no such thing. Apart from the latter, these differences can easily be overcome by program transformations.

Of more interest is the meta-complexity result for LA. It gives the time complexity of a LA program in terms of the number of (partial) rule instances and some related measures. The result is supported by an implementation proposal that gives strong complexity guarantees. We have presented a new meta-complexity result for CHR$^{\text{rp}}$, inspired by the one for LA and based on a similar implementation proposal. It states that the time complexity of a CHR$^{\text{rp}}$ program is assymptotically bounded by

$$C_{\mathsf{ask}} \cdot (A_{\mathsf{s}} + P_{\mathsf{s}} + (A_{\mathsf{d}} + P_{\mathsf{d}}) \cdot \log N) + B \cdot C_{\mathsf{tell}} \cdot (K + C_{\mathsf{ask}} \cdot S)$$

Here, the left-most summand represents the cost of finding an applicable rule instance and applying it. Among others, it takes into account the number of partial rule instances (P_{s} and P_{d} for static and dynamic priority rules respectively) and the number of distinct dynamic priorities (N). The right-most summand represents the cost of dealing with built-in constraints. The result is more generally applicable and more accurate than previous approaches. In particular, in comparison with [8], our approach supports propagation rules and is based on an optimized implementation. We generalize on the Logical Algorithms result by supporting a built-in constraint theory.

4 Adding Search, and Branch Priorities

Regular CHR does not support search as rules are applied in a committed-choice way. However, there exists a language extension for CHR, called CHR$^\vee$, which extends CHR with disjunction in the rule bodies [9]. An example CHR$^\vee$ rule is

```
domain(X,[V|Vs]) <=> X = V ∨ domain(X,Vs).
```

which can be used as a labeling procedure for a finite-domain constraint solver. When a CHR$^\vee$ rule with disjunction in the rule body is applied to an execution state, it is replaced by a number of alternative execution states (as many as there are disjuncts in the body). Each of these may lead to an alternative solution to the constraint problem.

The execution of a CHR$^\vee$ program can be represented as a search tree. By adding rule priorities to CHR$^\vee$, we can *shape* the search tree, that is, we can determine where and how branches are formed. What is missing is a way to state the *exploration strategy*, i.e., the order in which different alternatives are processed. Therefore, we have proposed CHR$^{\mathrm{brp}}_\vee$: CHR$^\vee$ with **b**ranch and **r**ule **p**riorities. The rule priorities are as in CHR$^{\mathrm{rp}}$. The branch priorities determine which alternative is processed next: at every branching point in the search tree, each of the newly created branches is assigned a branch priority and rules are applied only to states of the highest priority branch.

Example 3 (Depth-first and breadth-first). We extend the labeling rule given earlier with branch and rule priorities:

```
(D,N) :: domain(X,[V|Vs],N) <=> D+1 :: X = V ∨ D :: domain(X,Vs,N-1).
```

Here we extended the `domain` constraint with an extra argument (`N`) that denotes the number of elements in the domain. The syntax is as follows: the priority expression `(D,N)` consists of two parts, namely, the branch priority `D` and the rule priority `N`. Here, `D` is actually matched with the branch priority of the current branch, which allows us to use this priority when creating new branches in the body. The rule priority is such that domains with fewer elements are labeled first. In the body, we assign new branch priorities to the different disjuncts (before the `::`). In this case, the branch priority of the alternative formed by adding `X = V` to the constraints to be solved, is one more than that of the state that created this alternative. The branch priority of the other alternative remains unchanged.

Now if a smaller number indicates a higher branch priority, then the above code implements a breadth-first exploration strategy. If on the other hand a smaller number indicates a lower branch priority, it implements a depth-first strategy. In CHR$^{\mathrm{brp}}_\vee$, the order between branch priorities is user-defined. Note that in the above code, all alternative values of a domain appear at the same depth as far as the branch priorities are concerned. □

5 Conclusion

Constraint Handling Rules are a flexible means to implement application-specific constraint solvers on top of existing solvers. However, the language lacks facilities for specifying the execution control. In the thesis, we extended CHR with such facilities in such a way that the logic and control aspects of a CHR solver can be clearly separated. In particular, we made the following contributions:

- We extended CHR with rule priorities into CHR^{rp}. An optimized implementation of CHR^{rp} was presented, that is able to compete with the state-of-the-art in regular CHR.
- We investigated the relationship between CHR^{rp} and the Logical Algorithms framework by Ganzinger and McAllester. We have shown that the LA language is subsubmed by CHR^{rp} and proved a new meta-complexity result for CHR^{rp} that is more widely applicable and more accurate than previous results.
- We combined CHR^{rp} with $\text{CHR}^{\vee}$ and added branch priorities to allow for the specification of an exploration strategy. We also showed how to add support for conflict-directed backjumping on top of the resulting framework.

Apart from these topics, the thesis also discusses a fourth topic, namely join ordering for CHR, which is left out of this summary for space reasons.

References

1. De Koninck, L.: Execution control for Constraint Handling Rules. Ph.D thesis, K.U.Leuven (2008)
2. Frühwirth, T.: Theory and practice of Constraint Handling Rules. J. Log. Program. 37(1-3), 95–138 (1998)
3. Ganzinger, H., McAllester, D.A.: Logical algorithms. In: Stuckey, P.J. (ed.) ICLP 2002. LNCS, vol. 2401, pp. 209–223. Springer, Heidelberg (2002)
4. Duck, G.J., Stuckey, P.J., García de la Banda, M., Holzbaur, C.: The refined operational semantics of Constraint Handling Rules. In: Demoen, B., Lifschitz, V. (eds.) ICLP 2004. LNCS, vol. 3132, pp. 90–104. Springer, Heidelberg (2004)
5. De Koninck, L., Schrijvers, T., Demoen, B.: User-definable rule priorities for CHR. In: Leuschel, M., Podelski, A. (eds.) PPDP 2007, pp. 25–36. ACM, New York (2007)
6. De Koninck, L., Stuckey, P.J., Duck, G.J.: Optimizing compilation of CHR with rule priorities. In: Garrigue, J., Hermenegildo, M.V. (eds.) FLOPS 2008. LNCS, vol. 4989, pp. 32–47. Springer, Heidelberg (2008)
7. De Koninck, L.: Logical Algorithms meets CHR: A meta-complexity theorem for Constraint Handling Rules with rule priorities. In: TPLP (to appear, 2009)
8. Frühwirth, T.: As time goes by II: More automatic complexity analysis of concurrent rule programs. In: Quantitative Aspects of Programming Languages: Selected Papers. ENTCS, vol. 59. Elsevier, Amsterdam (2002)
9. Abdennadher, S., Schütz, H.: $\text{CHR}^{\vee}$: A flexible query language. In: Andreasen, T., Christiansen, H., Larsen, H.L. (eds.) FQAS 1998. LNCS, vol. 1495, pp. 1–14. Springer, Heidelberg (1998)

Demand-Driven Normalisation for ACD Term Rewriting

Leslie De Koninck[1], Gregory J. Duck[2], and Peter J. Stuckey[2]

[1] Department of Computer Science, K.U.Leuven, Belgium
Leslie.DeKoninck@cs.kuleuven.be
[2] NICTA Victoria Laboratory, Australia
{gjd,pjs}@cs.mu.oz.au

Abstract. ACD Term Rewriting (ACDTR) is term rewriting modulo associativity, commutativity, and a limited form of distributivity called conjunctive context. Previous work presented an implementation for ACDTR based on bottom-up eager normalisation, extended to support the conjunctive context. This paper investigates the possibility of using a demand-driven normalisation strategy for ACDTR. Again, dealing with the conjunctive context proves to be challenging. The alternative normalisation strategy is compared with the current form of eager normalisation and potential further improvements on the strategy are investigated.

1 Introduction

ACD Term rewriting (ACDTR) [1] is term rewriting modulo the equational theory E consisting of the following equivalences:

$$\begin{array}{lr} (\textit{associativity}) & (X \circ Y) \circ Z \approx_E X \circ (Y \circ Z) \\ (\textit{commutativity}) & X \circ Y \approx_E Y \circ X \\ (\textit{distributivity}) & P \wedge f(Q_1, \ldots, Q_i, \ldots, Q_n) \approx_E P \wedge f(Q_1, \ldots, P \wedge Q_i, \ldots, Q_n) \end{array}$$

for any *associative commutative* (AC) operator $\circ$, functor f/n and $i \in [1..n]$. ACDTR *simplification* rules are of the form "$H \iff B$" where H and B are terms. These rules rewrite terms matching H into B. In [1], rules may have guards; these are not considered in this paper to simplify the presentation. The distributivity property is used by *simpagation* rules of the form "$C \setminus H \iff B$" in which C is matched with terms in the *conjunctive context* (CC) of H, that is, the terms that appear conjoined with a superterm of H.

ACDTR is implemented in the Cadmium system [2] which is used in the G12 project [3] to map high-level constraint models onto low-level executable ones. ACDTR subsumes Constraint Handling Rules (CHR) [4] and so it inherits the latter's applications. The Cadmium system uses bottom-up eager normalisation, which is common for (AC) term rewriting, but incomplete because of the conjunctive context. Indeed, when normalising from the bottom up, the CC of a term may not be in normal form. In [2], this problem is dealt with by an event mechanism that causes conjuncts to be renormalised in case their CC changes in a relevant way. This event mechanism is an improvement over an earlier naive renormalisation policy described in [1]. This paper presents a top-down, demand-driven (lazy) normalisation strategy for ACDTR. First, an example:

Example 1. Consider the rule "$\texttt{f}(X) \iff \texttt{a}$" and *goal* term $\texttt{f}(\texttt{a}(\texttt{complex}, \texttt{term}))$. The argument of the $\texttt{f}/1$ term is not relevant for the above rule. However, when using eager bottom-up normalisation, this argument is first normalised, but finds itself being discarded later on. Lazy normalisation avoids this. □

When compared to the Cadmium system [2], the present work differs as follows: (1) normalisation proceeds from the top down and in a demand-driven way; (2) rules that rewrite AC terms are considered as multi-headed rules that apply to any term matching one of the heads, provided terms matching the remaining heads can be found amongst its siblings. E.g. the rule "$\texttt{a} + \texttt{b} \iff \texttt{c}$" is considered to apply to $\texttt{a}/0$ and $\texttt{b}/0$ terms rather than to $+$ terms; (3) we use an *active* conjunctive context as an alternative for the events of [2]. The idea is that the active term (i.e. the term being normalised) looks for terms that can be rewritten given that it is part of their conjunctive context.

While the implementation in [2] is based on techniques common in (AC) term rewriting, extended to support the conjunctive context, the principles behind the implementation presented in this paper are largely inspired by implementation techniques for CHR. In particular, the active CC and active AC operands are inspired by how applicable rule instances are found in CHR by means of an active constraint. The main contribution of this paper is that it shows the feasibility of a demand-driven normalisation policy for ACD term rewriting, with a discussion of the challenges such a policy introduces, in particular w.r.t. the CC.

2 Preliminaries

We assume some familiarity with term rewriting, see [5]. We consider AC terms to be flattened, e.g. $A \circ (B \circ C)$ with $\circ$ an AC operator, is represented as $\circ(A, B, C)$ and similar for any nesting of AC terms. We say that a rule "$H_1 \wedge \ldots \wedge H_m \setminus H_{m+1} \circ \ldots \circ H_n \iff B$" *applies* to a term S, subterm of the goal term T, if S matches with one of the heads $H_1, \ldots, H_n$ (modulo AC) and terms matching the remaining heads can be found among the subterms of T, such that the following conditions hold: (1) all terms matching the heads are different, (2) if $m + 1 < n$, the terms that match the heads $H_{m+1}, \ldots, H_n$ have a common parent which is a $\circ$ term, and (3) the terms matching the heads $H_1 \ldots, H_m$ appear in conjunction with $H_{m+1}, \ldots, H_n$ or one of their ancestors. We define that a rule *rewrites* a term S if the above conditions hold, and S matches one of the heads $H_{m+1}, \ldots, H_n$. In analogy with CHR terminology, we refer to the heads $H_1, \ldots, H_m$ as the *kept* heads, and to $H_{m+1}, \ldots, H_n$ as the *removed* heads.

A term S, subterm of the goal term T, may depend on this goal term in that some rules only apply to S or its subterms because it is part of T. The *conjunctive context* of a term T is the set of all terms that appear in conjunction with (a superterm of) T. The *relevant context* of a term T is its conjunctive context, plus its siblings if T's parent is an AC term. For example, given the goal term $\texttt{f}(\texttt{a} \wedge \texttt{g}(\texttt{b} + \texttt{c}) \wedge \texttt{d})$, the conjective context of subterm $\texttt{b}$ is $\{\texttt{a}, \texttt{d}\}$ and the relevant context is $\{\texttt{a}, \texttt{c}, \texttt{d}\}$. The relevant context contains exactly those terms that are relevant for enabling rule applications. In [2], only the CC of a term is important.

Here, we consider rules of the form "$H_1 \wedge \ldots \wedge H_m \setminus H_{m+1} \circ \ldots \circ H_n \iff B$" with $\circ$ an AC operator to rewrite any term matching one of the heads $H_{m+1}, \ldots, H_n$. In contrast, in [2], such a rule is considered to rewrite a $\circ$ term. The view we take here allows for more fine-grained on-demand rewrites during matching. Finally, we define the *inverse conjunctive context* of a term T as the set of terms whose conjunctive context contains T.

3 Demand-Driven Normalisation

Now we define what it means for a term to be in *normal form* and *toplevel normal form*. The toplevel normal form roughly corresponds to the (weak) head normal form in the lambda calculus and is less restrictive than the normal form.

Definition 1 (Normal form). *A term S, subterm of the goal term T, is in normal form if no rule rewrites S or its subterms.*

Definition 2 (Toplevel normal form). *A term $T = f(T_1, \ldots, T_n)$ has toplevel functor f/n. A term is in toplevel normal form if further normalisation of the term does not change its toplevel functor.*

The *status* of a term is either *normalised*, *toplevel normalised* or *unnormalised*. If a term's relevant context changes, its status may also change. E.g. given the rules "$\mathtt{f}(X) \wedge \mathtt{u} \iff X$" and "$\mathtt{t} \setminus \mathtt{a} \iff \mathtt{b}$" then term $\mathtt{f}(\mathtt{a})$ has status *normalised* as part of the goal term $\mathtt{f}(\mathtt{a}) \wedge \mathtt{n}$ whereas it has status *unnormalised* as part of the goal term $\mathtt{f}(\mathtt{a}) \wedge \mathtt{u}$ and status *toplevel normalised* as part of the goal term $\mathtt{f}(\mathtt{a}) \wedge \mathtt{t}$. We use the normalisation status of a term to decide if a rule can apply to a given term without having to normalise it completely. For example, a rule "$\mathtt{f}(X) \iff \mathtt{a}$" cannot be applied to a term $\mathtt{g}(\mathtt{b})$ if this term is in toplevel normal form. In particular, we do not need to normalise its argument ($\mathtt{b}$).

We now propose a rewrite strategy that ensures a term is in toplevel normal form. After applying it (sequentially) to all operands of an AC term, these operands are all in toplevel normal form. This is non-trivial as rewriting an AC operand changes the relevant context of its siblings, and so a term that was in toplevel normal form before, may no longer be so after such a context change.

We first describe how a given subject term is matched with a pattern term. We distinguish between the case that the pattern is a *linear* variable, a *nonlinear* variable, and a non-variable term. Variables are linear if they appear only once in the rule heads. In case of a linear variable, a match is trivially found and we assert a binding between the variable and the term with which it is matched. In case of a nonlinear variable, we first check if we already have a binding for this variable. If so, we match the term in question with the term bound to the variable (modulo AC). Otherwise, we normalise the term because at that point, it is not yet known which terms the variable need to be matched with further on. Therefore, we take a safe approach and normalise the term. While this deviates from lazy matching, there is no unique lazy way in general to match with nonlinear variables anyway: we can apply rewrites to any of the terms being matched in case they are not equal. Finally, in case of matching with a non-variable term, we check whether the functors of subject and pattern correspond. If so, we continue by matching the arguments of subject and pattern. Otherwise, we try to rewrite the subject term and if this succeeds, we try to match it with the pattern again.

To normalise a term, we first ensure it is in toplevel normal form by exhaustively applying rules to it. Next, we traverse all the term's arguments and recursively normalise them. A rule application starts with a matching phase, followed by a rewrite step if successful. First, we match the *active* term with a rule head, applying rules to its subterms if necessary. If the rule head is a kept head, we continue by looking for terms matching the rule's removed heads, and then for terms matching the remaining kept heads. The latter are in the CC of the terms matching the removed heads. Otherwise, we first match with the remaining removed heads, and then with the kept heads. During matching, no rules are applied to any term other than the subterms of the active term.

4 Optimisation

We have a prototype implementation of the described ideas, consisting of a Prolog front-end responsible for program analysis and preprocessing, and a Java back-end that interprets an internal representation of the program rules. We now present some optimisations that have been implemented.

Variables in a rule's body that also appear in its head, are bound to terms during matching. After a rule firing, these terms may appear in a new context which affects their normalisation status. Sometimes we can keep the normalisation status, namely if no terms are added to their relevant context. This optimisation is a generalisation of the conjunction collector optimisation of [2].

When terms are duplicated, we can either copy their representation, or use a shared representation for the duplicates. In standard term rewriting using a bottom-up normalisation strategy, the duplicates are already in normal form, and so we can easily share their representation as we do not need to perform rewrites on them. The Cadmium system [2] also uses a form of sharing, but rewrites are not performed on multiple occurrences of a shared term simultaneously.

Sharing causes some problems. Firstly, a rule may only apply to a term because of its context and different occurrences of a term may have a different context. Also, because we use a flattened representation for AC terms, the *result* of a rule application may be context dependent. We support sharing with simultaneous rewrites of all occurrences of a shared term. However, if a rewrite depends on a shared term's context, its representation is copied first, and the rewrite takes place on this non-shared copy. We allow AC terms to be in a non-flattened form temporarily, and flatten such terms on demand while matching.

Example 2. As an example of sharing, let there be given the following program:

$$\begin{array}{rclcrcl} \mathtt{f}(X) & \Longleftrightarrow & \mathtt{g}(X,X) & \qquad & \mathtt{c} \setminus \mathtt{b} & \Longleftrightarrow & \mathtt{a} \\ \mathtt{g}(\mathtt{a} \wedge X, Y) & \Longleftrightarrow & \mathtt{h}(X,Y) & \qquad & \mathtt{a} \setminus \mathtt{c} & \Longleftrightarrow & \mathtt{d} \end{array}$$

and goal term $\mathtt{f}(\mathtt{b} \wedge \mathtt{c})$, which is first rewritten into $\mathtt{g}(\mathtt{b} \wedge \mathtt{c}, \mathtt{b} \wedge \mathtt{c})$ using a shared representation for the $\wedge$ term. While matching this term with the rule for $\mathtt{g}/2$, we rewrite `b` into `a`, which results in $\mathtt{g}(\mathtt{a} \wedge \mathtt{c}, \mathtt{a} \wedge \mathtt{c})$. Since `b` is only shared via its parent, we can perform this rewrite for all occurrences of `b` simultaneously. Next, we rewrite the goal term into $\mathtt{h}(\mathtt{c}, \mathtt{a} \wedge \mathtt{c})$ where the `c` terms are shared. Finally, we rewrite the second occurrence of `c` into `d`. This rewrite depends on a shared term's context, so we copy its representation first. The result is $\mathtt{h}(\mathtt{c}, \mathtt{a} \wedge \mathtt{d})$. □

A final optimisation concerns inverse conjunctive context lookups. The inverse conjunctive context of a term T is computed in a demand-driven way and from the top down, i.e. a term is always considered before its subterms. We use indexing on the functor symbols appearing in a term to reduce the number of terms that are considered. We further optimise our approach by only indexing those terms that have already been considered while matching.

5 Conclusion

This work is strongly related to work on lazy evaluation in functional languages. It is known that lazy evaluation leads to better termination behaviour. It may also reduce the number of rule applications if subterms of a term being rewritten are discarded. In [1], it was shown that a bottom-up eager normalisation strategy for ACD term rewriting is incomplete because a term's conjunctive context might not be in normal form. In the approach we take here, it holds that if a conjunction term is in toplevel normal form, then so are all of its conjuncts. This means that often, terms in the conjunctive context are (at least) in toplevel normal form.

Acknowledgements. Research funded by a PhD grant of the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen). NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence Program.

References

1. Duck, G.J., Stuckey, P.J., Brand, S.: ACD term rewriting. In: Etalle, S., Truszczyński, M. (eds.) ICLP 2006. LNCS, vol. 4079, pp. 117–131. Springer, Heidelberg (2006)
2. Duck, G.J., De Koninck, L., Stuckey, P.J.: Cadmium: An implementation of ACD term rewriting. In: Garcia de la Banda, M., Pontelli, E. (eds.) ICLP 2008. LNCS, vol. 5366, pp. 531–545. Springer, Heidelberg (2008)
3. Stuckey, P.J., et al.: The G12 project: Mapping solver independent models to efficient solutions. In: Gabbrielli, M., Gupta, G. (eds.) ICLP 2005. LNCS, vol. 3668, pp. 9–13. Springer, Heidelberg (2005)
4. Frühwirth, T.: Theory and practice of Constraint Handling Rules. Journal of Logic Programming 37(1-3), 95–138 (1998)
5. Baader, F., Nipkow, T.: Term rewriting and all that. Cambridge Univ. Press, Cambridge (1998)

One More Decidable Class of Finitely Ground Programs

Yuliya Lierler and Vladimir Lifschitz

Department of Computer Sciences, University of Texas at Austin
{yuliya,vl}@cs.utexas.edu

Abstract. When a logic program is processed by an answer set solver, the first task is to generate its instantiation. In a recent paper, Calimeri *et el.* made the idea of efficient instantiation precise for the case of disjunctive programs with function symbols, and introduced the class of "finitely ground" programs that can be efficiently instantiated. Since that class is undecidable, it is important to find its large decidable subsets. In this paper, we introduce such a subset—the class of argument-restricted programs. It includes, in particular, all finite domain programs, ω-restricted programs, and λ-restricted programs.

1 Introduction

When an answer set solver, such as SMODELS[1] or DLV[2], starts processing a logic program Π, the first task is to generate an instantiation of Π—a program without variables that has the same answer sets as Π. In the course of instantiation, the rules of Π are grounded and simplified. Efficient instantiation algorithms expect that each rule of the input program is safe, in the sense that every variable occurring in the rule occurs in the positive part of its body. Some solvers impose stronger restrictions and expect that the given program is ω-restricted [1] or, more generally, λ-restricted [2].

For a program containing function symbols, however, even safety does not guarantee the possibility of instantiating the program efficiently. In fact, a safe program with functions can have infinite answer sets as, for instance, the program

$$\begin{array}{l} p(0) \\ p(f(X)) \leftarrow p(X). \end{array} \tag{1}$$

Such a program cannot be instantiated in a computationally meaningful way. In [3], the idea of efficient (or "intelligent") instantiation is made precise for disjunctive programs with function symbols. Efficient instantiation, as understood in that paper, is applicable to the logic programs that the authors call *finitely ground*. A program without function symbols is finitely ground if and only if it is safe. The program

[1] LPARSE+SMODELS: http://www.tcs.hut.fi/Software/smodels/
[2] DLV: http://www.dbai.tuwien.ac.at/proj/dlv/

$$\begin{array}{l} p(0) \\ q(f(X)) \leftarrow p(X) \end{array} \tag{2}$$

is finitely ground, but program (1) is not. Every finitely ground program has finitely many answer sets, and each of them is finite [3, Corollary 1]. Furthermore, there exists an algorithm for computing the answer sets of an arbitrary finitely ground program [3, Theorem 2].

It appears then that "finitely ground" is a property that a reasonable answer set solver can expect of its input. Unfortunately, the class of finitely ground programs is not decidable [3, Theorem 5]. This fact led the authors to the problem of describing large decidable subclasses of that class. As a step in this direction, they defined a decidable class of "finite domain" programs, and showed that every finite domain program is finitely ground [3, Theorems 6, 7].

In this paper, we introduce another decidable class of finitely ground programs, *argument-restricted* programs, which is a proper superset of the class of finite domain programs. For instance, the program

$$\begin{array}{l} p(f(X)) \leftarrow q(X) \\ q(X) \leftarrow p(X), r(X) \end{array} \tag{3}$$

is argument-restricted, but not finite domain program. The new class is also a superset of λ-restricted programs (and consequently of ω-restricted programs, in view of Theorem 1 from [2]). For instance, the program

$$\begin{array}{l} p(X) \leftarrow q(X) \\ q(X) \leftarrow p(X) \end{array} \tag{4}$$

is argument-restricted (as any safe program without function symbols), but not λ-restricted.

Figure 1 illustrates the relationships between the classes of logic programs mentioned above. The broken line shows the boundary of the important, but undecidable, class of finitely ground programs. In the picture, the class of finite domain programs and the class of λ-restricted programs partially overlap: the former contains program (4), but not (3); the latter contains (3), but not (4).

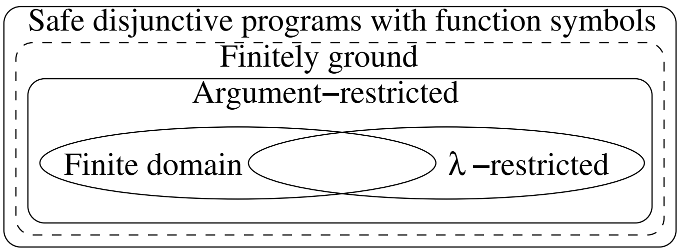


Fig. 1. Classes of logic programs

2 Argument Rankings

We consider disjunctive logic programs—finite sets of rules of the form

$$A_1; \ldots ; A_l \leftarrow A_{l+1}, \ldots , A_m,\ \textit{not}\ A_{m+1}, \ldots ,\ \textit{not}\ A_n \tag{5}$$

($n \geq m \geq l \geq 0$), where each A_i is an atom, possibly containing function symbols. The *positive body* of a rule (5) is the list $A_{l+1}, \ldots , A_m$. A program Π is *safe* if every variable occurring in a rule of Π occurs also in the positive body of that rule. Recall that *grounding* a logic program replaces each rule with all its instances obtained by substituting ground terms, formed from the object and function symbols occurring in the program, for all variables. The *answer sets* of a program are answer sets of the result of its grounding [4].

The definition of an argument ranking below, which is the main definition introduced in this paper, uses the following terminology and notation. For any atom $p(t_1, \ldots , t_n)$, by $p(t_1, \ldots , t_n)^0$ we denote its predicate symbol p, and by $p(t_1, \ldots , t_n)^i$, where $1 \leq i \leq n$, we denote its argument term t_i. As in [3], an *argument* is an expression of the form $p[i]$, where i is one of the argument positions $1, \ldots , n$. Finally, the *depth* of a variable X in a term t that contains X, denoted by $d(X, t)$, is defined recursively, as follows:

$$d(X,t) = \begin{cases} 0, & \text{if } t \text{ is } X, \\ 1 + \max\limits_{i\,:\,t_i \text{ contains } X} d(X, t_i), & \text{if } t \text{ is } f(t_1, \ldots , t_n). \end{cases}$$

An *argument ranking* for a program Π is a function α from arguments to integers such that, for every rule R of Π, every atom A occurring in the head of R, and every variable X occurring in an argument term A^i, the positive body of R contains an atom B such that X occurs in an argument term B^j satisfying the condition

$$\alpha\left(A^0[i]\right) - \alpha\left(B^0[j]\right) \geq d(X, A^i) - d(X, B^j). \tag{6}$$

A safe program is *argument-restricted* if it has an argument ranking.

Example 1. If a safe program Π does not contain function symbols in the heads of rules then it is argument-restricted, because its argument ranking can be defined by $\alpha(p[i]) = 0$ for all arguments $p[i]$. Indeed, the right-hand side of (6) for such a program is nonpositive, because $d(X, A^i) = 0$.

Example 2. Program (1) is not argument-restricted. In fact, any program containing the second rule of (1) is not argument-restricted, because for that rule condition (6) turns into $\alpha(p[1]) - \alpha(p[1]) \geq 1 - 0$.

Example 3. Program (2) is argument-restricted: take $\alpha(p[1]) = 0$, $\alpha(q[1]) = 1$.

Example 4. Program (3) is argument-restricted: take $\alpha(p[1]) = 1$, $\alpha(q[1]) = \alpha(r[1]) = 0$.

Example 5. The one-rule program $p(X, f(X)) \leftarrow p(X,X)$ is argument-restricted: take $\alpha(p[1]) = 0$, $\alpha(p[2]) = 1$.

It is clear that adding the same number to all values of an argument ranking produces another argument ranking for the same program. It follows that any argument-restricted program has an argument ranking with nonnegative values.

3 Properties of Argument-Restricted Programs

Theorem 1. *The set of argument-restricted programs is decidable.*

The easiest proof refers to the fact that the definition of an argument ranking, viewed as a condition on the values of $\alpha(p[i])$, can be encoded in difference logic [5]. A polynomial-time decision method for the set of argument-restricted programs is described in the next section.

The concept of a finitely ground program is defined in [3, Section 3].

Theorem 2. *Every argument-restricted program is finitely ground.*

The concept of a finite domain program is defined in [3, Section 5].

Theorem 3. *Every finite domain program is argument-restricted.*

As mentioned in the introduction, program (3) is a counterexample showing that the converse does not hold. The one-rule program $p(f(X)) \leftarrow p(g(X))$ and the program from Example 5 provide counterexamples as well: they are not finite domain programs, but they are argument-restricted.

The concept of a λ-restricted program is defined in [2, Section 2].

Theorem 4. *Every λ-restricted program is argument-restricted.*

As mentioned in the introduction, program (4) is a counterexample showing that the converse does not hold. The argument-restricted program from Example 5 is not λ-restricted either.

The definition of a λ-restricted program and the definition of an argument-restricted program are similar to each other in the sense that each of them refers to the existence of a number-valued function with certain properties. The difference is that the function is defined on predicate symbols p in the first case, and on arguments $p[i]$ in the second case. We know from Example 5 that the possibility of assigning different values to $p[1]$ and $p[2]$ is essential. Furthermore, the definition of a λ-restricted program does not take into account the depth of nesting of function symbols within a term. If we replace an occurrence of a term—say, $f(X)$— in a λ-restricted program with another term containing the same variables—say, X or $f(f(X))$—the result will be λ-restricted as well.

4 Checking Whether a Program Is Argument-Restricted

Recall that the definition of an argument ranking (Section 2) involves a condition on every

(i) rule R of the given program,
(ii) atom A occurring in the head of R,
(iii) argument position i of A, and
(iv) variable X occurring in A^i.

The inequality (6) in that condition can be rewritten as

$$\alpha\left(A^0[i]\right) \geq \alpha\left(B^0[j]\right) + d(X, A^i) - d(X, B^j). \tag{7}$$

For any R, A, i, X satisfying (i)–(iv), by $D_{R,A,i,X}(\alpha)$ we denote the list of the right-hand sides of inequalities (7) for all atoms B in the positive body of R and the argument positions j such that X occurs in B^j. Define the operator Ω on the set U of functions from arguments to nonnegative integers by the formula

$$\Omega(\alpha)(p[i]) = \max\left(\max_{R,A,X\,:\,A^0=p}\left(\min D_{R,A,i,X}(\alpha)\right), 0\right).$$

A function $\alpha \in U$ is an argument ranking for Π iff $\alpha \geq \Omega(\alpha)$.

The operator Ω is monotone. It follows that if Π is argument-restricted then the set of its nonnegative argument rankings has the least element $\alpha_{\min}$, and that $\alpha_{\min} = \Omega^i(0)$ for the smallest i such that $\Omega^{i+1}(0) = \Omega^i(0)$.

On the other hand, we can show that, for any argument-restricted Π, all values of $\alpha_{\min}$ do not exceed the number M defined as the product of the total number of arguments and the largest of the numbers $d(X, t)$ for the terms t occurring in the heads of rules and for the variables X occurring in t.

It follows that we can determine whether Π is argument-restricted by iterating Ω on 0 until

- $\Omega^{i+1}(0) = \Omega^i(0)$ —then $\alpha_{\min}$ is found, or
- one of the values of $\Omega^i(0)$ exceeds M —then Π is not argument-restricted.

Acknowledgements. Thanks to Martin Gebser, Tomi Janhunen, Joohyung Lee, Nicola Leone, Ilkka Niemelä, Wanwan Ren, Fangkai Yang for useful discussions. This research was partially supported by the NSF under Grant IIS-0712113.

References

1. Syrjänen, T.: Omega-restricted logic programs. In: Eiter, T., Faber, W., Truszczyński, M. (eds.) LPNMR 2001. LNCS, vol. 2173, pp. 267–279. Springer, Heidelberg (2001)
2. Gebser, M., Schaub, T., Thiele, S.: Gringo: A new grounder for answer set programming. In: Baral, C., Brewka, G., Schlipf, J. (eds.) LPNMR 2007. LNCS, vol. 4483, pp. 266–271. Springer, Heidelberg (2007)
3. Calimeri, F., Cozza, S., Ianni, G., Leone, N.: Computable functions in ASP: theory and implementation. In: Garcia de la Banda, M., Pontelli, E. (eds.) ICLP 2008. LNCS, vol. 5366, pp. 407–424. Springer, Heidelberg (2008)
4. Gelfond, M., Lifschitz, V.: Classical negation in logic programs and disjunctive databases. New Generation Computing 9, 365–385 (1991)
5. Mahfoudh, M., Niebert, P., Asarin, E., Maler, O.: A satisfiability checker for difference logic. In: Proceedings of International Conference on the Theory and Applications of Satisfiability Testing (SAT), pp. 222–230 (2002)

Optimizing Compilation and Computational Complexity of Constraint Handling Rules
Ph.D. Thesis Summary

Jon Sneyers

K.U. Leuven, Belgium
jon.sneyers@cs.kuleuven.be

1 Introduction

Constraint Handling Rules [1,2] is a high-level programming language extension based on multi-headed committed-choice multiset rewrite rules. It can be used as a stand-alone language or as an extension to an existing host language. CHR systems have been implemented for nearly every Prolog system, and there are also CHR systems for Haskell, Java and C.

In the past few years there has been quite some research interest in improving the performance of CHR systems. This has lead to the introduction of several novel compiler optimizations aimed at improving the computational complexity (mostly time complexity) of CHR programs. Recently at least four Ph.D. theses focussed on this topic: Tom Schrijvers [3] and Gregory Duck [4] worked mainly on optimizing compilation, while Leslie De Koninck[1] [5] worked on computational complexity and compilation techniques for variants of CHR that deviate from the de facto standard refined operational semantics ω_r [6]. This paper gives a brief overview of the main results of the Ph.D. thesis of Jon Sneyers [7].

2 Optimizing Compilation

The first part of the thesis [7] introduces CHR and gives an overview of the state-of-the-art in optimizing CHR compilation. In the second part, a number of novel compiler optimizations are introduced. They are briefly discussed in this section. Most of these optimizations are necessary for achieving the main complexity results (discussed in part three of the thesis and Section 3 of this summary). They are implemented in the Leuven CHR system [3] in hProlog [8].

Guard reasoning. The abstract operational semantics ω_t of CHR is very nondeterministic. For example, the order in which rules are applied is not specified at all. Instantiations of the abstract operational semantics — for example the refined operational semantics — remove sources of nondeterminism and in that sense they give more execution control to the programmer. Rule guards may be redundant under a more instantiated semantics while being necessary in the

[1] A summary of Leslie's Ph.D. thesis can be found elsewhere in this volume.

abstract semantics. Expert CHR programmers tend to remove such redundant guards. Although this improves performance, it also destroys the local character of the logical reading of CHR rules: in order to understand the meaning of a rule, the entire program and the details of the instantiated operational semantics have to be taken into account. As a solution, we propose compiler optimizations that automatically detect and remove redundant guards.

Memory reuse. Repeatedly replacing a constraint with a new one is a typical pattern that, in current CHR implementations, does not have the space complexity one might expect. The extra space can be reclaimed using (host language) garbage collection, but this comes at a cost in execution time. Indeed, CHR programmers often see that more than half of the total runtime is spent on garbage collection. We therefore introduce two compiler optimizations, *in-place update* and *suspension reuse*, that drastically reduce the memory footprint of CHR programs. Both optimizations reuse suspension terms, the internal representation of CHR constraints, and avoid redundant indexing operations. The optimizations are defined formally and their correctness is proved. They were implemented and significant memory savings and speedups were measured.

Join ordering. A crucial aspect of CHR compilation is finding matching rules efficiently. Given an active constraint, searching for matching partner constraints corresponds to joining relations—a well-studied topic in the context of databases. The performance of join methods is determined by the efficiency of the indexing techniques and by join ordering. In the refined operational semantics ω_r, a different join ordering can be used for each active occurrence in a rule. Given a CHR program $\mathcal{P}$ and one of its active occurrences a, a join ordering strategy $\prec$ imposes a total order $\prec_a^{\mathcal{P}}$ on the partner constraints of a. We formulate a generic cost model to evaluate join orderings and we propose static and dynamic heuristics to implement a join ordering strategy.

3 Computational Complexity

As a stand-alone programming language CHR is Turing-complete. In fact, several subclasses of CHR are already Turing-complete. These computability properties of CHR are discussed in the beginning of part three of the thesis [7]. The rest of part three is mostly devoted to the computational complexity of CHR [9].

In order to investigate the computational complexity of CHR, we have introduced abstract CHR machines. These machines essentially execute one CHR rule (or more exactly, one ω_t transition) in every step. We define the time complexity of a CHR machine to be the number of steps it takes. This is unrealistic since finding an applicable CHR rule takes more than constant time in general. We thus have to investigate the relation between CHR machines and more realistic models of computation, in particular the RAM machine.

We now state the main results. Because of space limitations we have to refer to [7] for the underlying definitions, lemmas, and proofs. We just give the definitions of determined partner constraints and the dependency rank of a constraint occurrence, because they are central to the formulation of the theorems.

Definition 1 (determined partner). *Given a join ordering strategy $\prec$, a CHR program $\mathcal{P}$, and a set of valid goals $\mathcal{VG}$, we say an occurrence c is* determined *by the j-th occurrence of constraint a iff for all execution states σ that occur in a derivation $d \in \Delta^{\mathcal{H}}_{\omega_r}|_{\mathbb{G}}$ for some valid goal $\mathbb{G} \in \mathcal{VG}$, the following holds: if σ is of the form $\langle [a\#i\!:\!j|\mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T}\rangle_n$ (that is, the occurrence subprocedure for the j-th occurrence of constraint a is about to be executed), then a set semantic functional dependency for c holds in state σ, where the key arguments of c are fixed by a and all partners x for which $x \prec^{\mathcal{P}}_{a} c$.*

In other words, a partner constraint c is determined by a given (active) constraint occurrence of a if the following holds: whenever the partner constraint c is looked up, there is at most one match that needs to be considered.

Definition 2 (dependency rank). *The dependency rank of an (active) occurrence a is the number of non-determined partner constraints of a.*

Complexity meta-theorem. The dependency rank of an occurrence corresponds to the "real" nesting depth of the lookup iterations. Given a constraint store of size S, the worst-case complexity of searching for matching partner constraints of an occurrence with dependency rank d is $O(S^d)$. This observation leads to the following complexity meta-theorems, which improve upon earlier results [10]:

Theorem 1. *Given a CHR program $\mathcal{P}$ and a ω_t derivation d of length T which has a corresponding ω_r derivation, for which the maximal store size is S, m is the maximum dependency rank of the active occurrences in $\mathcal{P}$, and p is the number of propagation rule applications in d; assuming the host language constraints used in the guards and bodies of the rules of $\mathcal{P}$ can be evaluated in constant time; the Leuven CHR system compiles $\mathcal{P}$ to hProlog code which has, for the given derivation d, a time complexity $O(TS^{m+1})$ and a space complexity $O(S + p)$.*

Theorem 2. *If in the previous theorem, the CHR program is ground (i.e. all constraint arguments are ground), then $O(TS^m)$ time complexity can be achieved.*

Complexity-wise completeness. Now we show that "everything can be done in CHR with the right complexity". Given an arbitrary RAM machine program, we can simulate it in CHR using the simulator program RAMSIMUL (Fig. 1). The program takes $O(T + S)$ rule applications to simulate a RAM-machine with time complexity T and space complexity S. Using the above complexity meta-theorem we then show the following theorem:

Theorem 3. *An ω_t derivation for the program* RAMSIMUL*, with T steps and maximal store size S, can be executed in $O(T)$ time and $O(S)$ space.*

Proof. Follows from Theorem 2: the program RAMSIMUL is ground, it has no propagation rules so $p = 0$, and there is a join ordering strategy such that the maximal dependency rank $m = 0$.

```
i(L,init,A), m(A,B), maxm(M) \ c(L) <=> initm(M+1,B,L).
      initm(A,B,L) <=> A =< B | m(A,0), initm(A+1,B,L).
      initm(A,B,L), m(B,X) <=> A > B | m(B,0), maxm(B), c(L+1).
i(L,cnst,B,A) \ m(A,X), c(L) <=> m(A,B), c(L+1).
i(L,add,B,A), m(B,Y) \ m(A,X), c(L) <=> m(A,X+Y), c(L+1).
i(L,sub,B,A), m(B,Y) \ m(A,X), c(L) <=> m(A,X-Y), c(L+1).
i(L,mul,B,A), m(B,Y) \ m(A,X), c(L) <=> m(A,X*Y), c(L+1).
i(L,div,B,A), m(B,Y) \ m(A,X), c(L) <=> m(A,X//Y), c(L+1).
i(L,mov,B,A), m(B,Y) \ m(A,_), c(L) <=> m(A,Y), c(L+1).
i(L,imv,B,A), m(B,C), m(C,Y) \ m(A,_), c(L) <=> m(A,Y), c(L+1).
i(L,mvi,B,A), m(B,Y), m(A,C) \ m(C,_), c(L) <=> m(C,Y), c(L+1).
i(L,jmp,A) \ c(L) <=> c(A).
i(L,cjmp,A,J), m(A,0) \ c(L) <=> c(J).
i(L,cjmp,A,J), m(A,X) \ c(L) <=> X =\= 0 | c(L+1).
i(L,halt) \ c(L) <=> true.
```

Fig. 1. RAMSIMUL: Simulator of standard RAM machines

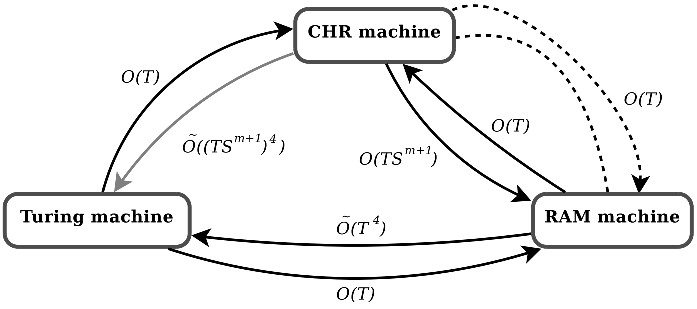


Fig. 2. Time complexity relationships between Turing, RAM, and CHR machines

Figure [2] gives an overview. We conclude that "everything can be done in CHR":

Corollary 1. *For every (RAM machine) algorithm which uses at least as much time as it uses space, a CHR program exists which can be executed in the Leuven CHR system in hProlog, with time and space complexity within a constant from the original complexities.*

4 Discussion and Conclusion

One may expect to pay *some* performance penalty for using a very high-level language like CHR. Therefore, it is good to have a complexity-wise completeness result, which essentially proves that one can always get the asymptotic time and space complexity right in CHR. Of course the constant factors are also important in practice. These have also been investigated in [7]. The general construction described above (using a RAM machine simulator) predictably yields very large constant factors — about four orders of magnitude between CHR(hProlog) and assembler code. However, using more elegant high-level CHR programs to implement an algorithm, a much more acceptable performance was measured: the CHR(hProlog) programs (for Union-find and Dijkstra's algorithm with Fibonacci heaps) were 'only' about one order of magnitude slower than direct implementations in C, and they used about 3 to 10 times as much space.

For other declarative programming languages, it remains a challenge to prove a similarly strong complexity-wise completeness result. In [7], a first attempt was made to "port" the result to some other declarative languages. For Prolog, Haskell, and Maude [11] we could not find a way to achieve complexity-wise completeness within the pure fragment of the language. In Jess [12] we did get complexity-wise completeness, but with a much worse constant factor (about 30 times slower than CHR). Other languages still have to be investigated.

Acknowledgments. This research was funded by a Ph.D. grant of the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen). Most of the work presented in my Ph.D. thesis was joint work with Tom Schrijvers, Bart Demoen, Peter Van Weert, and Leslie De Koninck.

References

1. Frühwirth, T.: Constraint Handling Rules. Cambridge University Press, Cambridge (2009)
2. Sneyers, J., Van Weert, P., Schrijvers, T., De Koninck, L.: As time goes by: Constraint Handling Rules — A survey of CHR research between 1998 and 2007. In: Theory and Practice of Logic Programming (2009)
3. Schrijvers, T.: Analyses, optimizations and extensions of Constraint Handling Rules. Ph.D thesis, K.U.Leuven, Leuven, Belgium (June 2005)
4. Duck, G.J.: Compilation of Constraint Handling Rules. Ph.D thesis, University of Melbourne, Victoria, Australia (December 2005)
5. De Koninck, L.: Execution control for Constraint Handling Rules. Ph.D thesis, K.U.Leuven, Leuven, Belgium (November 2008)
6. Duck, G.J., Stuckey, P.J., García de la Banda, M., Holzbaur, C.: The refined operational semantics of Constraint Handling Rules. In: Demoen, B., Lifschitz, V. (eds.) ICLP 2004. LNCS, vol. 3132, pp. 90–104. Springer, Heidelberg (2004)
7. Sneyers, J.: Optimizing Compilation and Computational Complexity of Constraint Handling Rules. Ph.D thesis, K.U.Leuven, Leuven, Belgium (November 2008)
8. Demoen, B., Nguyen, P.L.: So many WAM variations, so little time. In: Palamidessi, C., Moniz Pereira, L., Lloyd, J.W., Dahl, V., Furbach, U., Kerber, M., Lau, K.-K., Sagiv, Y., Stuckey, P.J. (eds.) CL 2000. LNCS, vol. 1861, pp. 1240–1254. Springer, Heidelberg (2000)
9. Sneyers, J., Schrijvers, T., Demoen, B.: The computational power and complexity of Constraint Handling Rules. ACM Trans. Program. Lang. Syst. 31(2) (2009)
10. Frühwirth, T.: As time goes by: Automatic complexity analysis of simplification rules. In: Proc. 8th Intl. Conf. Princ. Knowledge Representation and Reasoning, pp. 547–557. Morgan Kaufmann, San Francisco (2002)
11. Clavel, M., Durán, F., Eker, S., et al.: Maude: Specification and programming in rewriting logic. Theoretical Computer Science 285(2), 187–243 (2002)
12. Friedman-Hill, E.: Jess in Action: Java Rule-Based Systems. Manning Publications (2003)

Proving Termination by Invariance Relations

Paolo Pilozzi[⋆] and Danny De Schreye

Dept. of Computer Science, K.U.Leuven, Belgium
paolo.pilozzi@cs.kuleuven.be, danny.deschreye@cs.kuleuven.be

Abstract. We propose a new constraint-based approach to termination analysis, applicable to Logic Programming (LP) and Constraint Handling Rules (CHR). Our approach further extends the existing constraint-based approaches for LP based on polynomial interpretations and introduces a whole new level of expressivity. We can handle problems such as bounded increase and integer arithmetic, elegantly. Furthermore, we are able to prove termination of programs that only terminate for subsets of the considered queries. Examples are algorithms that manipulate graphs and that only terminate if the graph in the input is cycle-free. This information cannot be represented, using the existing techniques in termination analysis. Therefore, we introduce invariance relations, representing relations among terms that hold on atoms during calls to the program. These relations can also be derived in a constraint-based manner and they can be used as a basis for a more expressive interpretation of the atoms of the program. We discuss our technique in the context of CHR, solving an important class of open problems containing transitivity rules. We also demonstrate the technique in an LP context and show that it is more powerful than existing constraint-based approaches.

The following CHR program [3, 7], computes the transitive closure of a graph.

$$transitivity \ @ \ arc(X,Y), arc(Y,Z) \Rightarrow arc(X,Z).$$

This program is representative for a large class of practical programs in CHR that cannot be handled using any existing automated technique. For it to be proven terminating, the query graph may not contain cycles. Consider for example a query of two constraints, $arc(a,b)$ and $arc(b,c)$, representing a cycle-free graph, consisting of three nodes: a, b and c. The transitivity rule is applicable to these constraints, adding the constraint $arc(a,c)$. Since a fire-once policy prevents multiple applications of a propagation rule on the same combination of constraints, the program terminates. If on the other hand, the query graph contains a cycle, then arcs are added progressively along that cycle, ad infinitum.

To tackle the problem of programs that only terminate for some subset of the considered queries, we present in the next paragraphs a new approach to constraint-based termination analysis. The technique is based on polynomial interpretations [5], however, we allow more expressive polynomials than the ones used in [5]. Our approach is applicable to both LP and CHR programs. First, we discuss our approach for CHR programs containing a transitivity rule. Then, we show the technique's applicability to LP and argue that it is more powerful than existing approaches.

⋆ Supported by I.W.T. Flanders - Belgium.

To prove termination automatically, we develop verifiable sufficient conditions [2,6]. To express such conditions, we require information on possible calls to a program. Such information comes in the form of a *call set.* In LP, this corresponds to the set of atoms, selected in some derivation of the program, for some query. In CHR, it corresponds to the atoms used to fire a rule in some computation of the program for some query. To specify the intended use of a program, we use a *query set*, finitely represented using query patterns. From it, we select atoms to compose queries. Then, by application of abstract interpretation, we derive from query patterns, call patterns [4,7] as a representation of the call set.

In [5], termination is proven using a polynomial interpretation. There, the interpretation maps atoms and terms to $\mathbb{N}$-closed polynomials, using a *polynomial level mapping*, $|.|$, for atoms and a *polynomial norm*, $\|.\|$, for terms. The call set has to be rigid w.r.t. this interpretation, meaning that only polynomials can be constructed using the instantiated parts of atoms and terms. To verify this, syntactic conditions are constructed, based on call patterns [1].

In [5], the $\mathbb{N}$-closed polynomials that are used, are constructed from positive monomials. For example, a polynomial $a_0 + a_1.X_1 + \cdots + a_n.X_n$. However, for programs such as our running example, we require negative monomials to prove termination. We allow therefore polynomials that are the difference of two (linear) positive polynomials: $a_0^1+a_1^1.X_1+\cdots+a_n^1.X_n-(a_0^2+a_1^2.X_1+\cdots+a_n^2.X_n)$. For such polynomials, $\mathbb{N}$-closedness is not trivial: there is obviously a positive integer assignment to variables and coefficients, resulting in a negative integer outcome. To guarantee that such polynomials are $\mathbb{N}$-closed, $a_0^1 + a_1^1.X_1 + \cdots + a_n^1.X_n \geq a_0^2 + a_1^2.X_1 + \cdots + a_n^2.X_n$ must hold during the execution of the program. In the next paragraph, we derive this information, using *invariance relations.*

To prove termination of a CHR program containing a transitivity rule, we need to be able to represent more restricted kinds of queries. Call patterns [4] provide information on the instantiated parts of atoms in the call set, however, do not represent relations that hold among these parts. To represent this information, we introduce invariance relations, holding on atoms in the query or the call set.

To prove termination of our running example, we must guarantee the use of a cycle-free query graph. In such a graph, one can always find an ordering on nodes, such that every arc points from a strictly smaller sized node to a bigger sized node. Thus, the queries we can use are in $\{\bigwedge_{i=1,\ldots,n} arc(t_i', t_i'') \mid \forall i : t_i'' \succ t_i'\}$.

Using an invariance relation $I_{arc/2}^{query} = \{(t_1, t_2) \mid t_1, t_2 \in Term_P, t_{max} \succ t_2 \succ t_1 \succ t_{min}\}$, we can formulate cycle-freeness on the level of individual query atoms. The constants, t_{max} and t_{min}, fix the domain of terms that can be used as nodes. The ordering on terms in arcs is expressed on individual atoms. In contrast to the invariant formulated on the query, the latter invariance relation therefore represents a relation holding on atoms that can be used in a query, while the former invariant represents actual queries.

Given the invariance relation $I_{arc/2}^{query}$ on query atoms, we aim to derive an invariance relation $I_{arc/2}^{callset}$ on the atoms used in calls. For that purpose, we will now assume that $I_{arc/2}^{query}$ is not just an invariance relation, but actually specifies the query atoms of interest. Since we also assume that the query atoms are a subset of the call atoms, the invariance relation $I^{callset}_{arc/2}$ on the call atoms must hold on the query atoms. Thus, $I^{query}_{arc/2} \subseteq I^{callset}_{arc/2}$. More explicitly, we get the condition $\forall t_1, t_2 \in Term_P : (t_1, t_2) \in I^{query}_{arc/2} \rightarrow (t_1, t_2) \in I^{callset}_{arc/2}$, meaning that the invariance relation on query atoms must be more restrictive than the one on atoms in the call set. On the basis of CHR rules, we derive rule conditions. For the transitivity rule, we get that $\forall t_1, t_2, t_3 \in Term_P : (t_1, t_2) \in I^{callset}_{arc/2} \wedge (t_2, t_3) \in I^{callset}_{arc/2} \rightarrow (t_1, t_3) \in I^{callset}_{arc/2}$, meaning that the invariance relation holding on $arc/2$ atoms, has to remain valid under transitivity.

We derive invariance relations on call atoms in a constraint-based manner, using a polynomial interpretation. That is, we reformulate the conditions by using a symbolic polynomial form for invariance relations in the same way as this is done for interargument relations in [5]. The invariance relation in polynomial form for the query is $I^{query}_{arc/2} = \{(t_1, t_2) \mid \|t_{max}\| > \|t_2\| \wedge \|t_2\| > \|t_1\| \wedge \|t_1\| > \|t_{min}\|\}$.

For the invariance relation on the atoms used in calls, we provide the same level of expressivity as for the query. Thus for the $arc/2$ constraints, we express their invariance relation using three symbolic polynomial inequalities: $I^{callset}_{arc/2} = \{(t_1, t_2) \mid ii^1_0 + ii^1_1.\|t_1\| + ii^1_2.\|t_2\| \geq io^1_0 + io^1_1.\|t_1\| + io^1_2.\|t_2\| \wedge ii^2_0 + ii^2_1.\|t_1\| + ii^2_2.\|t_2\| \geq io^2_0 + io^2_1.\|t_1\| + io^2_2.\|t_2\| \wedge ii^3_0 + ii^3_1.\|t_1\| + ii^3_2.\|t_2\| \geq io^3_0 + io^3_1.\|t_1\| + io^3_2.\|t_2\|\}$. To find values for the symbolic coefficients, we formulate the invariance conditions in terms of these symbolic polynomials. Thus, $\forall t_1, t_2 \in Term_P$:

$$\|t_{max}\| > \|t_2\| \wedge \|t_2\| > \|t_1\| \wedge \|t_1\| > \|t_{min}\| \rightarrow ii^1_0 + ii^1_1.\|t_1\| + ii^1_2.\|t_2\| \geq io^1_0 + io^1_1.\|t_1\| + io^1_2.\|t_2\| \wedge ii^2_0 + ii^2_1.\|t_1\| + ii^2_2.\|t_2\| \geq io^2_0 + io^2_1.\|t_1\| + io^2_2.\|t_2\| \wedge ii^3_0 + ii^3_1.\|t_1\| + ii^3_2.\|t_2\| \geq io^3_0 + io^3_1.\|t_1\| + io^3_2.\|t_2\|$$

For the condition related to the transitivity rule, we get that $\forall t_1, t_2, t_3 \in Term_P$:

$$ii^1_0 + ii^1_1.\|t_1\| + ii^1_2.\|t_2\| \geq io^1_0 + io^1_1.\|t_1\| + io^1_2.\|t_2\| \wedge ii^2_0 + ii^2_1.\|t_1\| + ii^2_2.\|t_2\| \geq io^2_0 + io^2_1.\|t_1\| + io^2_2.\|t_2\| \wedge ii^3_0 + ii^3_1.\|t_1\| + ii^3_2.\|t_2\| \geq io^3_0 + io^3_1.\|t_1\| + io^3_2.\|t_2\| \wedge ii^1_0 + ii^1_1.\|t_2\| + ii^1_2.\|t_3\| \geq io^1_0 + io^1_1.\|t_2\| + io^1_2.\|t_3\| \wedge ii^2_0 + ii^2_1.\|t_2\| + ii^2_2.\|t_3\| \geq io^2_0 + io^2_1.\|t_2\| + io^2_2.\|t_3\| \wedge ii^3_0 + ii^3_1.\|t_2\| + ii^3_2.\|t_3\| \geq io^3_0 + io^3_1.\|t_2\| + io^3_2.\|t_3\| \rightarrow ii^1_0 + ii^1_1.\|t_1\| + ii^1_2.\|t_3\| \geq io^1_0 + io^1_1.\|t_1\| + io^1_2.\|t_3\| \wedge ii^2_0 + ii^2_1.\|t_1\| + ii^2_2.\|t_3\| \geq io^2_0 + io^2_1.\|t_1\| + io^2_2.\|t_3\| \wedge ii^3_0 + ii^3_1.\|t_1\| + ii^3_2.\|t_3\| \geq io^3_0 + io^3_1.\|t_1\| + io^3_2.\|t_3\|$$

These conditions are similar to the ones obtained in [5] and thus can be solved in a similar way. That is, by transforming them to a system of Diophantine constraints on the symbolic coefficients and solving the resulting system by an existing constraint solver. One possible solution, results in an invariance relation,

$$I^{callset}_{arc/2} = \{(t_1, t_2) \mid \|t_{max}\| + 0.\|t_1\| + 0.\|t_2\| \geq 0 + 0.\|t_1\| + 1.\|t_2\| \wedge 0 + 0.\|t_1\| + 1.\|t_2\| \geq 0 + 1.\|t_1\| + 0.\|t_2\| \wedge 0 + 1.\|t_1\| + 0.\|t_2\| \geq \|t_{min}\| + 0.\|t_1\| + 0.\|t_2\|\}$$

Thus, the invariance relation on the query is preserved under transitivity. That is, the bounds, t_{max} and t_{min}, remain to be bounds for the added arcs and these arcs keep pointing from strictly smaller sized nodes to bigger sized nodes.

To prove the transitivity program terminating, we obtain the following termination conditions, according to [6]. The scope of the decrease conditions is restricted by invariance relations holding on the atoms used in the head.

$$\forall t_1, t_2, t_3 : (t_1, t_2) \in I^{callset}_{arc/2} \wedge (t_2, t_3) \in I^{callset}_{arc/2} \rightarrow arc(t_1, t_2) \succ arc(t_1, t_3)$$
$$\forall t_1, t_2, t_3 : (t_1, t_2) \in I^{callset}_{arc/2} \wedge (t_2, t_3) \in I^{callset}_{arc/2} \rightarrow arc(t_2, t_3) \succ arc(t_1, t_3)$$

To prove validity of these conditions, we require an interpretation, mapping $arc/2$ atoms to a polynomial of the form $|arc(t_1, t_2)| = (\|t_{max}\| - \|t_{min}\|) - (\|t_2\| - \|t_1\|)$. Since, $(\|t_{max}\| - \|t_{min}\|) - (\|t_2\| - \|t_1\|) \geq 0$ is implied by $I^{callset}_{arc/2}$, we map all $arc/2$ atoms in the call set, to $\mathbb{N}$-closed polynomials.

To obtain an integrated approach, incorporating invariance and termination conditions, we introduce a symbolic form for level mappings, parameterizable by invariance relations. For $arc/2$ atoms, we have $|arc(t_1, t_2)| = ii_0 + ii_1.\|t_1\| + ii_2.\|t_2\| - (io_0 + io_1.\|t_1\| + io_2.\|t_2\|)$ as its symbolic form, which depends on the invariance relation holding on $arc/2$ atoms in the call set. Thus, $\forall t_1, t_2 \in Term_P$:

$$ii^1_0 + ii^1_1.\|t_1\| + ii^1_2.\|t_2\| \geq io^1_0 + io^1_1.\|t_1\| + io^1_2.\|t_2\| \wedge ii^2_0 + ii^2_1.\|t_1\| + ii^2_2.\|t_2\| \geq$$
$$io^2_0 + io^2_1.\|t_1\| + io^2_2.\|t_2\| \wedge ii^3_0 + ii^3_1.\|t_1\| + ii^3_2.\|t_2\| \geq io^3_0 + io^3_1.\|t_1\| + io^3_2.\|t_2\| \rightarrow$$
$$ii_0 + ii_1.\|t_1\| + ii_2.\|t_3\| \geq io_0 + io_1.\|t_1\| + io_2.\|t_3\|$$

Notice that the symbolic form, used in existing constraint-based approaches [5], is implied by default. After all, $ii_0 + ii_1.\|t_1\| + ii_2.\|t_2\| \geq 0$ is trivially implied, since only polynomials are constructed with positive integer coefficients. Therefore, if termination can be proven by existing systems, we can prove it as well.

As termination conditions, we obtain now:

$$\forall t_1, t_2, t_3 : (t_1, t_2) \in I^{callset}_{arc/2} \wedge (t_2, t_3) \in I^{callset}_{arc/2} \rightarrow$$
$$ii_0{+}ii_1.\|t_1\|{+}ii_2.\|t_2\|{-}(io_0{+}io_1.\|t_1\|{+}io_2.\|t_2\|) \geq ii_0{+}ii_1.\|t_1\|{+}ii_2.\|t_3\|{-}(io_0{+}io_1.\|t_1\|{+}io_2.\|t_3\|)$$
$$\forall t_1, t_2, t_3 : (t_1, t_2) \in I^{callset}_{arc/2} \wedge (t_2, t_3) \in I^{callset}_{arc/2} \rightarrow$$
$$ii_0{+}ii_1.\|t_2\|{+}ii_2.\|t_3\|{-}(io_0{+}io_1.\|t_2\|{+}io_2.\|t_3\|) \geq ii_0{+}ii_1.\|t_1\|{+}ii_2.\|t_3\|{-}(io_0{+}io_1.\|t_1\|{+}io_2.\|t_3\|)$$

Solving these conditions using the techniques in [5], provides us with the aforementioned interpretation for $arc/2$ constraints, proving termination.

Our technique is also applicable to LP. Similarly, invariance relations for atoms used in calls, can be derived on the basis of an invariance relation for the query atoms. We demonstrate this on the following LP program.

$$a(Max, Max). \qquad a(N, Max) \ :- \ neq(N, Max), a(s(N), Max).$$

Currently, no technique for termination analysis of LP can handle such programs. Since a call to $neq/2$ also succeeds whenever the first term of an $a/2$ atom is bigger than the second, the program will run forever, as for such queries the first argument has no upper bound. Thus, in order to prove termination, we have to consider that a query consists of $a(t_1, t_2)$ atoms, where $t_1 \prec t_2$, represented using an invariance relation, $I^{query}_{a/2} = \{(t_1, t_2) \mid t_1, t_2 \in Term_P, \|t_2\| > \|t_1\|\}$.

Consequently, to derive $I^{callset}_{a/2}$, we initially set it to a general symbolic form, $\{(t_1, t_2) \mid t_1, t_2 \in Term_P \wedge ii^1_0 + ii^1_1.\|t_1\| + ii^1_2.\|t_2\| \geq io^1_0 + io^1_1.\|t_1\| + io^1_2.\|t_2\|\}$

and then, we formulate invariance conditions. To express $I^{query}_{a/2} \subseteq I^{callset}_{a/2}$, we get that $\forall t_1, t_2 \in Term_P : \|t_2\| > \|t_1\| \rightarrow ii^1_0 + ii^1_1.\|t_1\| + ii^1_2.\|t_2\| \geq io^1_0 + io^1_1.\|t_1\| + io^1_2.\|t_2\|$. For the clause, we obtain that $\forall t_1, t_2 \in Term_P : ii^1_0 + ii^1_1.\|t_1\| + ii^1_2.\|t_2\| \geq io^1_0 + io^1_1.\|t_1\| + io^1_2.\|t_2\| \wedge (\|t_1\| > \|t_2\| \vee \|t_2\| > \|t_1\|) \rightarrow ii^1_0 + ii^1_1.\|s(t_1)\| + ii^1_2.\|t_2\| \geq io^1_0 + io^1_1.\|s(t_1)\| + io^1_2.\|t_2\|$. Here, $R_{neq/2} = \{(t_1, t_2) \mid \|t_1\| > \|t_2\| \vee \|t_2\| > \|t_1\|\}$ estimates the effect of a call to $neq/2$. Then, a suitable polynomial level mapping for $a/2$ atoms is derived on the basis of $I^{callset}_{a/2}$. That is, $\forall t_1, t_2 \in Term_P : ii^1_0 + ii^1_1.\|t_1\| + ii^1_2.\|t_2\| \geq io^1_0 + io^1_1.\|t_1\| + io^1_2.\|t_2\| \rightarrow ii_0 + ii_1.\|t_1\| + ii_2.\|t_2\| \geq io_0 + io_1.\|t_1\| + io_2.\|t_2\|$.

To prove termination, we must show that all recursive body atoms are smaller than the atom used in the head [2]. This results for the LP program, in the following condition: $\forall t_1, t_2 \in Term_P : ii^1_0 + ii^1_1.\|t_1\| + ii^1_2.\|t_2\| \geq io^1_0 + io^1_1.\|t_1\| + io^1_2.\|t_2\| \wedge (\|t_1\| > \|t_2\| \vee \|t_2\| > \|t_1\|) \rightarrow ii_0 + ii_1.\|t_1\| + ii_2.\|t_2\| - (io_0 + io_1.\|t_1\| + io_2.\|t_2\|) \geq ii_0 + ii_1.\|s(t_1)\| + ii_2.\|t_2\| - (io_0 + io_1.\|s(t_1)\| + io_2.\|t_2\|)$. When solving the termination, invariance and interargument conditions, we obtain $|a(t_1, t_2)| = \|t_2\| - \|t_1\|$ as one possible level mapping for $a/2$ atoms, proving termination.

To conclude, we proposed a new constraint-based approach, more powerful than existing approaches in LP. Our approach can handle both LP and CHR programs containing bounded increase, integer arithmetic or programs that only terminate for subsets of considered queries. We overcame these problems by introducing invariance relations, resulting in a useful specification for relations holding among the instantiated parts of call atoms and thus complementing call types. As such, we are able to represent more restricted kinds of queries, related to the use of domains or even global properties of queries, such as cycle-freeness of a query graph. In the continuation of this work, we will formalize this technique and further evaluate it, in the context of CHR and LP.

References

1. Bossi, A., Cocco, N., Fabris, M.: Proving termination of logic programs by exploiting term properties. In: Abramsky, S. (ed.) CAAP 1991 and TAPSOFT 1991. LNCS, vol. 493, pp. 153–180. Springer, Heidelberg (1991)
2. De Schreye, D., Decorte, S.: Termination of logic programs: the never-ending story. JLP 19-20, 199–260 (1994)
3. Frühwirth, T.: Theory and practice of constraint handling rules. JLP 37(1-3), 95–138 (1998)
4. Janssens, G., Bruynooghe, M.: Deriving descriptions of possible values of program variables by means of abstract interpretation. JLP 13(2-3), 205–258 (1992)
5. Nguyen, M.T., Schreye, D.: Polynomial interpretations as a basis for termination analysis of logic programs. In: Gabbrielli, M., Gupta, G. (eds.) ICLP 2005. LNCS, vol. 3668, pp. 311–325. Springer, Heidelberg (2005)
6. Pilozzi, P., De Schreye, D.: Termination analysis of CHR revisited. In: Garcia de la Banda, M., Pontelli, E. (eds.) ICLP 2008. LNCS, vol. 5366, pp. 501–515. Springer, Heidelberg (2008)
7. Schrijvers, T.: Analyses, Optimizations and Extensions of Constraint Handling Rules: Ph.D. summary. In: Gabbrielli, M., Gupta, G. (eds.) ICLP 2005. LNCS, vol. 3668, pp. 435–436. Springer, Heidelberg (2005)

Automating Termination Proofs for CHR

Paolo Pilozzi[⋆] and Danny De Schreye

Dept. of Computer Science, K.U.Leuven, Belgium
paolo.pilozzi@cs.kuleuven.be, danny.deschreye@cs.kuleuven.be

Abstract. We propose a constraint-based approach towards automated termination analysis of Constraint Handling Rules (CHR). Similar to such approaches for Logic Programming (LP), we define a symbolic level mapping on atoms of the program and express termination conditions using these. Then, we search for an assignment to the symbolic coefficients, validating the termination conditions. However, different from the approaches developed for LP, termination of CHR programs is concerned with multi-headed rules, while the existing constraint-based approaches in LP are developed for definite programs. These cannot be adapted directly to a multi-headed context. In the following we discuss a constraint-based approach for CHR programs and show how such an approach can be obtained by an elegant reuse of existing techniques in LP. We evaluate the approach, using our implementation, CHRisTA.

A CHR program is a finite set of rules, syntactically named by "*rulename* @". There are essentially two kinds of rules in a CHR program: *simplification* and *propagation* rules [2]. Both are discussed in the next example.

Example 1 (Constraint Handling Rules). The program below is a simple genetic algorithm. The CHR constraints $\alpha/2, \beta/2$ and $f/2$ are first-order relations on terms and represent respectively, alpha-males, beta-males and females. Their first arguments represent their generation and their second arguments, genes.

$$\begin{array}{l} r_1 \ @ \ \alpha(N_1,G_1), f(s(N_2),G_2) \Rightarrow \mathit{less}(N_2,N_1) \mid \mathit{recombine}(G_1,G_2,G), \beta(N_2,G). \\ r_2 \ @ \ \alpha(N_1,G_1), f(s(N_2),G_2) \Rightarrow \mathit{less}(N_2,N_1) \mid \mathit{recombine}(G_1,G_2,G), f(N_2,G). \\ \ s \ @ \ \alpha(N_1,G_1), \beta(s(N_2),G_2) \Leftrightarrow \mathit{less}(N_2,N_1), \mathit{fitter}(G_2,G_1) \mid \alpha(s(N_2),G_2). \end{array}$$

The reproduction rules, r_1 and r_2, are *propagation rules* and the selection rule, s, is a *simplification rule.* In general, a simplification rule takes the form $H_k \setminus H_r \Leftrightarrow G \mid B, C$ or $H_r \Leftrightarrow G \mid B, C$ if no kept head is present. A propagation rule takes the form $H_k \Rightarrow G \mid B, C$. Here, H_k, H_r and C are conjunctions of kept, removed and added CHR constraints and G and B conjunctions of built-ins.

The reproduction rules guarantee that only alpha-males can reproduce with younger females, producing as such new beta-males or females of an even younger generation. The third rule performs selection and thus replaces alpha-males by fitter and younger beta-males. This behavior is accomplished through the use of built-ins provided in the host-language, Prolog. The built-in $\mathit{less}/2$ succeeds if term-size of its first argument is smaller than term-size of its second,

[⋆] Supported by I.W.T. Flanders - Belgium.

recombine/3 defines mutations and crossover of genes and *fitter*/2 succeeds if its first argument contains a fitter gene than its second.

For a given query, different computations may exist. This is due to the non-deterministic choice of mates in the reproduction rules and the non-deterministic removal of alpha-males in the selection rule. Alpha-males will therefore mate with a non-deterministic subset of the female population to produce offspring. To prove termination, we require finiteness of all computations from a given query. If this query is composed of males and females with given generations, thus ground first arguments, termination is guaranteed due to a limited number of children produced. That is, every generation can only produce younger generations and since generations have a lower bound, this process dies out. □

Until recently, two complementary techniques existed to prove termination of CHR programs: A technique for CHR without propagation [1] and a technique for CHR with propagation [7]. These techniques were generalized in [6], resulting in a single approach, able to prove new classes of CHR programs terminating. However, [6] only describes the theory of the termination analysis. In the current paper, we discuss how the technique in [6] can be automated. Also, we evaluate the approach and discuss extensions.

To prove termination of CHR programs, separate conditions exist for propagation rules and simplification rules [6]. In these conditions, built-ins are left unconsidered. We assume them to terminate on their own and not to introduce new CHR constraints. However, built-ins may impose interargument relations. These relations are assumed to be given and represent the success set of a built-in, according to its specification in the host-language. An interargument relation therefore estimates the effect of answer substitutions of built-ins in a rule.

To express these relations, we resort to *reduction pairs* $(\succeq, \succ)$ [5], consisting of a *quasi-order* $\succeq$: a reflexive ($s \succeq s$) and transitive ($s \succeq u \succeq v \rightarrow s \succeq v$) binary relation; and a *well-founded order* $\succ$: a transitive relation without infinite chains ($s_0 \succ s_1 \succ \dots$); that are compatible ($s \succ u \succeq v \rightarrow s \succ v$). We define the associated equivalence relation $s \approx t \leftrightarrow s \succeq t \wedge t \succeq s$. Thus, for the $less/2$ predicate, we can define its interargument relation as $R_{less/2} = \{(t_1, t_2) \mid t_1, t_2 \in \mathrm{Term}_P \wedge t_1 \prec t_2\}$. The interargument relations of the other built-ins, *recombine*/3 and *fitter*/2, are not discussed. They are not required to prove termination.

As mentioned, separate conditions exists for propagation and simplification rules. For propagation rules, the level value of any head constraint must be strictly larger than the level value of any added body CHR constraint. Thus the conditions for the propagation rules in the running example, take the form:

$$\forall \bar{X} : (N_2, N_1) \in R_{less/2} \rightarrow \alpha(N_1, G_1) \succ \beta(N_2, G) \wedge f(s(N_2), G_2) \succ \beta(N_2, G)$$
$$\forall \bar{X} : (N_2, N_1) \in R_{less/2} \rightarrow \alpha(N_1, G_1) \succ f(N_2, G) \wedge f(s(N_2), G_2) \succ f(N_2, G)$$

Notice that by $\forall \bar{X}$, we mean all assignments of terms constructible from the program, to the variables occurring in the conditions. The scope of these assignments is restricted by interargument relations in the pre-condition. To prove termination, we have to find a reduction pair, validating these conditions.

For simplification rules, the condition is more complex. The condition requires a multi-set decrease between removed heads and added body CHR constraints. Such an ordering is induced on a reduction pair. Let $(\succeq, \succ)$ be a reduction pair for the elements of two multi-sets X and Y. Let n_r^X be the number of elements in X that are equivalent to some term r, given the reduction pair $(\succeq, \succ)$. Then, X has a strictly larger multi-set size than Y, denoted $X \succ_m Y$, if there exists an r such that $n_r^X > n_r^Y$ and such that for any $q \succ r : n_q^X = n_q^Y$. We call r and q ranks and represent them using a term or a constraint. These denote an equivalence class of terms or constraints, given $(\succeq, \succ)$. In the context of a multi-set decrease, we call r the decreasing rank. Thus we obtain for the selection rule, the decrease condition:

$$\forall \bar{X} : (N_2, N_1) \in R_{less/2} \rightarrow \{\alpha(N_1, G_1), \beta(s(N_2), G_2)\} \succ_m \{\alpha(s(N_2), G_2)\}$$

The condition on simplification rules has a second requirement. No CHR constraint, added by propagation on the added CHR constraints of a simplification rule, may undo the multi-set decrease. Thus for every match of an added CHR constraint with a head of a propagation rule, we add to the pre-condition, the match and the built-ins of a renamed variant of the propagation rule. Then, in the conclusion, we express decrease conditions, using r as a representation for the decreasing rank, associated to the multi-set decrease. We get:

$$\begin{gathered}\forall \bar{X} : (N_2, N_1) \in R_{less/2} \wedge \alpha(s(N_2), G_2) \approx \alpha(N_1', G_1') \wedge (N_2', N_1') \in \\ R_{less/2} \wedge \alpha(s(N_2), G_2) \approx \alpha(N_1'', G_1'') \wedge (N_2'', N_1'') \in R_{less/2} \rightarrow \\ \{\alpha(N_1, G_1), \beta(s(N_2), G_2)\} \succ_m \{\alpha(s(N_2), G_2)\} \wedge r \succ \beta(N_2', G') \wedge r \succ f(N_2'', G'')\end{gathered}$$

To express the conditions for simplification rules using the reduction pair, multiple disjunctive conditions turn up, all implying the multi-set decrease of the original condition. Consider for example the multi-set decrease between $\{A, B\} \succ_m \{C, D\}$, given a reduction pair $(\succeq, \succ)$. Then, $A \succ B \wedge A \succ C \wedge A \succ D \rightarrow \{A, B\} \succ_m \{C, D\}$, but also $B \succ A \wedge B \succ C \wedge B \succ D \rightarrow \{A, B\} \succ_m \{C, D\}$ and others. For every such instance, a different decreasing rank can be determined. So if we represent the instances of the condition on the selection rule of our running example, then we get after filling in the decreasing rank:

$$\begin{gathered}\forall \bar{X} : (N_2, N_1) \in R_{less/2} \wedge \alpha(s(N_2), G_2) \approx \alpha(N_1', G_1') \wedge (N_2', N_1') \in \\ R_{less/2} \wedge \alpha(s(N_2), G_2) \approx \alpha(N_1'', G_1'') \wedge (N_2'', N_1'') \in R_{less/2} \rightarrow \alpha(N_1, G_1) \succeq \\ \beta(s(N_2), G_2) \wedge \alpha(N_1, G_1) \succeq \alpha(s(N_2), G_2) \wedge \alpha(N_1, G_1) \succ \\ \beta(N_2', G') \wedge \alpha(N_1, G_1) \succ f(N_2'', G'') \\ \forall \bar{X} : (N_2, N_1) \in R_{less/2} \wedge \alpha(s(N_2), G_2) \approx \alpha(N_1', G_1') \wedge (N_2', N_1') \in \\ R_{less/2} \wedge \alpha(s(N_2), G_2) \approx \alpha(N_1'', G_1'') \wedge (N_2'', N_1'') \in R_{less/2} \rightarrow \\ \beta(s(N_2), G_2) \succeq \alpha(N_1, G_1) \wedge \beta(s(N_2), G_2) \succeq \alpha(s(N_2), G_2) \wedge \beta(s(N_2), G_2) \succ \\ \beta(N_2', G') \wedge \beta(s(N_2), G_2) \succ f(N_2'', G'')\end{gathered}$$

Validity of one of these conditions is sufficient to prove validity of the original condition containing the multi-set decrease. Notice that if multiple simplification rules are present in a CHR program, that for each of these rules a decrease condition is formulated containing a multi-set decrease. For every one of these conditions, several instances may exist. Therefore, for every combination of instances,

Table 1. Results in *seconds* of CHRisTA on terminating CHR programs

ackermann	–	$constr_{02}$	1.43	$constr_{15}$	7.85	*fib*	1.53	*pathc*	–
average	1.40	$constr_{03}$	1.52	$constr_{16}$	3.47	*gcd*	1.43	*power*	1.64
binlog	1.34	$constr_{04}$	3.22	$constr_{17}$	4.32	$genint_1$	1.47	$primes_1$	1.51
booland	0.16	$constr_{05}$	3.21	$constr_{18}$	4.39	$genint_2$	–	$primes_2$	2.74
boolcard	1.52	$constr_{06}$	3.15	$constr_{19}$	5.35	$genint_3$	–	*revlist*	1.58
$compl_1$	1.49	$constr_{07}$	2.44	$constr_{20}$	8.48	*joinlists*	6.79	*sum*	3.98
$compl_2$	18.56	$constr_{08}$	2.47	$constr_{21}$	4.90	*linpoleq*	1.42	*succadd*	1.95
$compl_3$	20.44	$constr_{09}$	2.55	$constr_{22}$	3.79	*max*	0.35	*tak*	5.74
$compl_4$	5.04	$constr_{10}$	2.58	*convert*	1.45	*mean*	21.26	*toyama*	0.15
$compl_5$	3.67	$constr_{11}$	7.59	*dfsearch*	1.44	*mergesort*	4.25	*weight*	1.56
$compl_6$	6.77	$constr_{12}$	5.11	*diff*	1.51	*modulo*	1.48	*zebra*	1.66
concat	–	$constr_{13}$	4.93	*even*	2.67	*nqueen*	6.00	*ztoa*	–
$constr_{01}$	1.54	$constr_{14}$	7.97	*factorial*	1.50	*oddeven*	1.45		

together with the conditions on propagation rules, we obtain a disjunctive system of conditions, each one of them implying validity of the original system. The resulting systems of conditions take a similar form as the system of conditions derived for a logic program and thus can be solved by techniques developed in termination analysis for LP. For example by a constraint-based approach using polynomial interpretations [5].

We implemented a constraint-based approach with polynomial interpretations for CHR on top of Prolog. We called the system CHRisTA (CHR Termination Analyzer) and implemented it in CHR(SWI-Prolog)[1]. The system lacks a call set analyzer, to be a fully-automated termination analyzer. Thus, to handle non-ground queries, we provide the system with call types, such as the ones derived by call set analyzers for LP [3]. We obtain promising results with our analyzer, being able to prove termination of the entire benchmark of [1] and [7]. Furthermore, we can handle the programs, $compl_i$, not in the scope of [1] or [7].

The constructed programs, $constr_i$, are included to demonstrate the impact of increasing numbers of heads and bodies. $constr_1$ contains a single-headed simplification rule with one body CHR constraint. The second and third program contain two body CHR constraints. $constr_4$ until $constr_{11}$ contain two heads and one body CHR constraint and the remaining programs, two heads and two body CHR constraints. The impact of an increasing number of heads is bigger than the impact of an increasing number of bodies. This is because the number of multi-set instances is mainly determined by the number of heads. The differences between programs containing the same numbers of heads and bodies is because termination is proven using different underlying multi-set instances.

The order in which these instances are tried is pre-defined. In our case, we first try the instances corresponding to the condition on simplification rules formulated in [7]. It expresses a more restricted form of a multi-set decrease, corresponding to a strict subset of the multi-set instances that we obtain. Since

[1] http://www.cs.kuleuven.be/∼paolo/CHRisTA/index.html

the set of instances from [7] is much smaller and has proven its applicability on many practical problems in CHR, we try these instances first.

In the table, containing our evaluation results, there are still some classes of CHR programs that cannot be handled using CHRisTA. These are marked with '–'. A first important class of problems are non-primitive recursive programs, such as the *ackermann* program. Such programs require a lexicographical interpretation for atoms, which CHRisTA cannot handle as yet. However, similar to how the problem is solved in LP, it can be solved in CHR by application of the underlying ideas of the Dependency Pair Approach [4].

To prove problems such as *concat* terminating, we require more expressive polynomial interpretations, e.g. a mixed polynomial form. In CHRisTA we provide only the linear form. However, as in [5], more expressive forms can easily be incorporated. Another limitation is that simplification rules cannot contain more than a total number of five head and body constraints. Multi-set instances for bigger rules were not pre-computed, but can be added to the list of multi-set instances. This is why we cannot handle $genint_3$.

Problems regarding bounded increase, such as $genint_2$, cannot be handled. Even in LP, such programs cause problems for analyzers. For programs that only terminate for a subset of the considered class of queries, such as *pathc*, no solutions exist. Finally, there are programs such as *ztoa* that cannot be proven terminating due to no clear concept of a success set in CHR.

To conclude, we have shown how termination for CHR programs can be proved automatically, using a constraint-based approach. We have implemented the approach in a system called CHRisTA and obtained good results when evaluating it. Since solutions exist for many of the considered problem classes, we will direct future work towards solving these issues first and further extending our analyzer.

References

1. Frühwirth, T.: Proving Termination of Constraint Solver Programs. In: New Trends in Constraints, pp. 298–317 (2000)
2. Frühwirth, T.: Theory and practice of constraint handling rules. JLP 37(1-3), 95–138 (1998)
3. Janssens, G., Bruynooghe, M.: Deriving descriptions of possible values of program variables by means of abstract interpretation. JLP 13(2-3), 205–258 (1992)
4. Nguyen, M.T., Giesl, P., Schneider-Kamp, J., De Schreye, D.: Termination analysis of logic programs based on dependency graphs. In: King, A. (ed.) LOPSTR 2007. LNCS, vol. 4915, pp. 8–22. Springer, Heidelberg (2008)
5. Nguyen, M.T., De Schreye, D.: Polynomial interpretations as a basis for termination analysis of logic programs. In: Gabbrielli, M., Gupta, G. (eds.) ICLP 2005. LNCS, vol. 3668, pp. 311–325. Springer, Heidelberg (2005)
6. Pilozzi, P., De Schreye, D.: Termination analysis of CHR revisited. In: Garcia de la Banda, M., Pontelli, E. (eds.) ICLP 2008. LNCS, vol. 5366, pp. 501–515. Springer, Heidelberg (2008)
7. Voets, D., Pilozzi, P., De Schreye, D.: A new approach to termination analysis of Constraint Handling Rules. In: Pre-proceedings LOPSTR 2008, pp. 28–42 (2008)

An Overview of FORCES: An INRIA Project on Declarative Formalisms for Emergent Systems

Jesús Aranda[1], Gerard Assayag[4], Carlos Olarte[1,3], Jorge A. Pérez[2], Camilo Rueda[3,4], Mauricio Toro[3], and Frank D. Valencia[1]

[1] INRIA and CNRS-LIX, École Polytechnique
[2] Dept. of Computer Science, University of Bologna
[3] Pontificia Universidad Javeriana Cali
[4] Institut de Recherche et Coordination Acoustique/Musique, IRCAM

Abstract. The FORCES project aims at providing robust and *declarative formalisms* for analyzing systems in the emerging areas of *Security Protocols, Biological Systems* and *Multimedia Semantic Interaction*. This short paper describes FORCES's motivations, results and future research directions.

Introduction. FORCES (FORmalisms from Concurrency for Emergent Systems) is an ongoing project funded by the *Equipes Associées* programme of INRIA. It is carried by the INRIA team COMETE (France), the IRCAM Music Representation Team (France) and the team AVISPA (Colombia). The main goal of FORCES is to provide robust *declarative formalisms* for modeling systems from emergent application areas of computer science in which our teams have been working during recent years: Namely, *Security Protocols, Biological Systems* and *Multimedia Semantic Interaction*.

Process calculi are formalisms that treat communicating processes much like the λ-calculus treats computable functions: The structure of terms reflects the structure of processes and process evolution is represented by term reduction. *Concurrent Constraint Programming* (CCP) based calculi [1] are computational models that combine the operational view of process calculi with a *declarative* one based upon logic.

Some of the members of FORCES developed and used `ntcc` [2], a timed CCP calculus, to predict the behavior of systems from Security Protocols [3], Systems Biology [4] and Multimedia Semantic Interaction [5]. Although these areas differ significantly from one another, there is a crucial commonality in the analysis we wanted to perform in them: *Reachability* i.e., whether a system reaches a particular state. The `ntcc` calculus provides several reasoning tools for reachability analysis. These include a temporal logic, a proof system, verification techniques, and a denotational semantics.

Nevertheless, we have learned from our modeling experience and theoretical studies that `ntcc` is not sufficiently robust for these applications. E.g., some security protocols use a mechanism to allow communication of *nonces* (i.e., uniquely generated random number). The `ntcc` calculus can at best express this mechanism indirectly [6]. Also, `ntcc` lacks constructs for quantitative information, which are essential for biological systems. Furthermore, we have identified musical settings exhibiting complex non-regular timed behavior that cannot be expressed in `ntcc`.

Our research strategy in FORCES has been, with the benefit of hindsight, to develop declarative formalisms for modeling systems from the above-mentioned areas as suitable *extensions* or *specializations* of ntcc. Our expertise in ntcc as well as our modeling experience have been fundamental for guiding our research.

This short paper provides an overview of FORCES. Further information can be found at http://www.lix.polytechnique.fr/comete/Forces.

Declarative Models of Security Protocols. A fundamental ability for security protocols is that of generating and communicating private *nonces*; process calculi for security therefore include mechanisms for creating and communicating local names. Neither ntcc nor its predecessor tcc [7] features such mechanisms.

As a remedy to this, in [3] we introduced the Universal Timed CCP process calculus (utcc): a generalization of tcc that allows for the communication of local names (or *links*). This additional expressiveness paves the way for the declarative modeling of a wider class of systems, most notably dynamic ones.

We have endowed utcc with a number of reasoning techniques for reachability analysis. A *symbolic semantics* was defined to deal with problematic operational aspects involving infinitely many substitutions which often arise when modeling security protocols. The semantics uses temporal constraints to finitely represent infinitely-many substitutions; it has been used to exhibit secrecy flaws in some security protocols [3]. The utcc calculus also enjoys a *declarative view* of processes as First-Order Temporal Logic (FLTL) formulae [8]. This allows for reachability analysis of utcc processes using FLTL techniques. For instance, in [3] we used the FLTL formulae representing the model of a protocol to know if it reaches a state where the attacker knows a secret.

We also defined a denotational semantics for utcc [9]. This way, processes can be represented as partial closure operators. As an application of the semantics, we identified a language for security protocols that can be represented as closure operators over a cryptographic constraint system. We showed that the least fixed point of such an operator may then be used to check a secrecy property in a protocol. To our knowledge, this is the first denotational account in the context of calculi for security protocols.

This way, our work has brought new semantic insights into the verification of security protocols, and is related to the research in security protocols from areas closely related to CCP. Namely, Constraint Programming (e.g. [10]) and Logic Programming (e.g. [11,12]). To our knowledge there is no work on Security Protocols that takes advantage of the reasoning techniques of CCP.

Declarative Models of Biological Systems. Quantitative information is fundamental for biological systems. For example, behavior in most biochemical reactions is highly dependent on the presence of a certain amount of the substances involved. Very often, information is *partial* as obtaining exact values for parameterizing models is difficult. Unpredictable behavior is thus an inherent condition of the biologic phenomena, and one counts with *partial behavioral information* for describing system interactions. This partial information not only ignores elements on *how* reactions occur (e.g. what components actually interact), but also on *when* such reactions commonly happen (e.g. the relative speeds of the interacting components).

While the notion of partial quantitative information is central to CCP via constraints, partial behavioral information is actually the novelty of ntcc via non-deterministic and asynchronous operators. Our teams have already explored these advantages by analyzing mechanisms for cellular transport and genetic regulatory networks [4,13].

A drawback of these models is their lack of explicit quantitative information. As hinted at above, a fundamental feature of any model of biological systems is the capability of exploiting any available quantitative information. In biological systems this is often represented as *stochastic behavior*. One then has a set of reactions each endowed with a rate representing their propensity or speed. When considering their execution, a race between them takes place and the fastest action is executed.

We have taken initial steps on the inclusion of stochastic information into an explicitly timed concurrent constraint process language [14]. We defined stochastic events in terms of the time units defined by the language: this provides great flexibility for modeling and allows for a clean semantics. Most importantly, by considering stochastic information and adhering to explicit discrete time, it is possible to reason about processes using quantitative logics (both discrete and continuous), while retaining the simplicity of calculi such as `ntcc` for deriving qualitative reasoning techniques (such as denotational semantics and proof systems). We plan to consolidate the framework outlined in [14], and to apply it to study systems such as the modeled in [15].

Declarative Models of Multimedia Semantic Interaction. Interactivity in multimedia systems has become increasingly important. The aim is to devise ways for the machine to be an effective and active partner in a collective behavior constructed dynamically by many actors. In complex forms of multimedia interaction the machine is always adapting its behavior according to the information derived from the activity of the other partners who, in turn, adapt theirs according to the computer actions.

Constructing multimedia systems is thus a challenging task. Their core depends on powerful and consistent concurrent agents architectures. In this setting, `ntcc` has much to offer. Complex dynamic agent synchronization scenarios can be modeled cleanly and compositionally based on the synchronization mechanism provided by blocking ask constructs. Also, interactions on which little information is available can be conveniently represented using non-determinism. Most importantly, safety properties of the model, crucial in performance settings, can be formally proved to hold.

Quantitative information is also important in musical settings. In [16], we proposed `pntcc`, the first `ntcc` extension featuring probabilistic and non-deterministic choices. `pntcc` advocates the specification of probabilistic, reactive systems within non-deterministic environments. The calculus is equipped with an operational semantics that ensures consistent interactions between both kinds of choices. The semantics is also crucial in the definition of logic-based verification capabilities over system specifications. We have used `pntcc` for analyzing a scenario of interactive music improvisation (see the extended version of [16]). Probabilitistic information was shown to be useful to obtain a quantitative measure of the quality of an improvised sequence and to enhance the control the modeler has over the whole process.

Interactive scores [17] are models for reactive music systems where weakly defined temporal relations between components specify a hierarchy of loosely coupled music processes. Although the hierarchical structure has been treated as static in previous works, there is no reason it should be so. In [18], we propose a model for dynamic interactive scores where interactive points can be defined to adapt the score depending on the information inferred from the environment (say, a set of performers). We then broaden the interaction mechanisms available for the composer.

In [19] we proposed rtcc, a model of real-time concurrent constraint programming which adds to ntcc the means for specifying and modeling real-time behavior. This calculus has constructs for modeling strong preemption and for defining delays within the same time unit. The operational semantics of rtcc supports resources, limited time and true concurrency. We showed the applicability of the rtcc calculus by giving more faithful models of various musical situations previously modeled in other CCP calculi.

In multimedia interaction settings the real-time execution of models is central. We have implemented a first prototype of an interpreter for ntcc specifications providing real-time interaction with the models developed in the calculus. The tool, called NtccRT [20], also allows for the integration with music composition environments such as OpenMusic (OM, [21]). NtccRT provides a means to write a ntcc specification graphically (using OM) and then to compile it as an standalone program interacting with Midishare [22]. The tool is available at http://ntccrt.sourceforge.net.

Future Directions: Automatic Verification. Presently there are no automatic, nor machine-assisted, tools for the simulation and verification of concurrent systems specified in ntcc. Since we deal with complex and large systems, these tools are essential to our intended applications. In fact, the issue of automatic support has received little attention in the case of CCP formalisms. To our knowledge only Villanueva et al (e.g. [23,24]) have addressed automatic verification but in the context of finite-state CCP systems. Several applications of ntcc are, however, inherently infinite-state. Automatic verification of large systems, not to mention infinite systems, is challenging because of the state explosion problem it poses—i.e., the number of states a system has is exponential in the number of processes.

We will take up this challenge by identifying ntcc fragments amenable to automatic verification and by developing techniques and tools to machine assist the verification of system properties in ntcc. We envisage two main complementary approaches for our purposes: (1) Automaton-based and symbolic techniques, and (2) Static and abstract interpretation techniques. We plan to use the automaton representations of processes used to prove the the decidability of the verification problem for ntcc [6], together with the symbolic approach in [3] to ameliorate the state explosion problem. Finally, we expect to develop static and abstract interpretation techniques to extract representative information from system specifications. Such information can be used to reason about essential properties of systems behavior.

We plan to use Security Protocols to test the above-mentioned techniques and tools. In fact, the analysis of Security Protocols is typically carried out using symbolic verification techniques thus making them ideal application candidates. It is also worth noticing that computer simulation plays a fundamental role for Biological Systems because of their inherent complexity. We also plan to develop an ntcc simulation tool and use it as a test bench for (abstractions of) biological systems. We expect that the declarative and parametric nature of ntcc should provide bench biologists with a tool for computing with and analyzing these systems that is intuitive.

References

1. Saraswat, V.A., Rinard, M., Panangaden, P.: The semantic foundations of concurrent constraint programming. In: Proc. of POPL 1991. ACM Press, New York (1991)
2. Nielsen, M., Palamidessi, C., Valencia, F.D.: Temporal concurrent constraint programming: Denotation, logic and applications. Nordic Journal of Computing 9(1) (2002)
3. Olarte, C., Valencia, F.D.: Universal concurrent constraint programing: Symbolic semantics and applications to security. In: Proc. of SAC 2008. ACM, New York (2008)
4. Gutiérrez, J., Pérez, J.A., Rueda, C., Valencia, F.: Timed concurrent constraint programming for analysing biological systems. Electron. Notes Theor. Comput. Sci. 171(2) (2007)
5. Rueda, C., Valencia, F.D.: A temporal concurrent constraint calculus as an audio processing framework. In: Proc. of SMC 2005 (2005)
6. Valencia, F.D.: Decidability of infinite-state Timed CCP processes and first-order LTL. Theor. Comput. Sci. 330(3), 577–607 (2005)
7. Saraswat, V., Jagadeesan, R., Gupta, V.: Foundations of timed concurrent constraint programming. In: Proc. of LICS 1994. IEEE CS, Los Alamitos (1994)
8. Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems: Specification. Springer, Heidelberg (1991)
9. Olarte, C., Valencia, F.D.: The expressivity of universal timed CCP: Undecidability of monadic FLTL and closure operators for security. In: Proc. of PPDP 2008 (2008)
10. Bella, G., Bistarelli, S.: Soft constraint programming to analysing security protocols. TPLP 4(5-6), 545–572 (2004)
11. Abadi, M., Blanchet, B.: Analyzing Security Protocols with Secrecy Types and Logic Programs. Journal of the ACM 52(1) (2005)
12. Millen, J.K.: The interrogator: A tool for cryptographic protocol security. In: Proc. of IEEE Symposium on Security and Privacy, pp. 134–141 (1984)
13. Arbeláez, A., Gutiérrez, J., Pérez, J.A.: Timed Concurrent Constraint Programming in Systems Biology. Newsletter of the ALP 19(4) (2006)
14. Aranda, J., Pérez, J., Rueda, C., Valencia, F.: Stochastic behavior and explicit discrete time in concurrent constraint programming. In: Garcia de la Banda, M., Pontelli, E. (eds.) ICLP 2008. LNCS, vol. 5366, pp. 682–686. Springer, Heidelberg (2008)
15. Cardelli, L., Gardner, P., Kahramanogullari, O.: A process model of rho gtp-binding proteins in the context of phagocytosis. Electr. Notes Theor. Comput. Sci. 194(3), 87–102 (2008)
16. Pérez, J.A., Rueda, C.: Non-determinism and Probabilities in Timed Concurrent Constraint Programming. In: Garcia de la Banda, M., Pontelli, E. (eds.) ICLP 2008. LNCS, vol. 5366, pp. 677–681. Springer, Heidelberg (2008)
17. Allombert, A., Assayag, G., Desainte-Catherine, M.: A system of interactive scores based on Petri nets. In: Proc. of SMC 2007 (2007)
18. Olarte, C., Rueda, C.: A declarative language for dynamic multimedia interaction systems. In: Proc of. MCM 2009. Springer, Heidelberg (to appear, 2009)
19. Sarria, G., Rueda, C.: Real-time concurrent constraint programming. In: Proc. of CLEI 2008 (2008)
20. Toro, M., Rueda, C., Assayag, G., Agón, C.: NtccRT: A Concurrent Constraint Framework for Real-Time Interaction. In: Proc. of International Computer Music Conference (2009)
21. Bresson, J., Agon, C., Assayag, G.: Openmusic 5: A cross-platform release of the computer-assisted composition environment. In: Proc. of Brazilian Symposium on Computer Music (2005)
22. Fober, D., Orlarey, Y., S.L.: Midishare: une architecture logicielle pour la musique. Hermes, 175–194 (2004)
23. Alpuente, M., Gallardo, M., Pimentel, E., Villanueva, A.: Verifying Real-Time Properties of tccp Programs. Journal of Universal Computer Science 12(11), 1551–1573 (2006)
24. Falaschi, M., Villanueva, A.: Automatic verification of timed concurrent constraint programs. TPLP 6(3), 265–300 (2006)

An Engine for Computing Well-Founded Models

Terrance Swift

CENTRIA — Universidade Nova de Lisboa
tswift@cs.sunysb.edu

Abstract. The seemingly simple choice of whether to use call variance or call subsumption in a tabled evaluation deeply affects an evaluation's properties. Most tabling implementations have supported only call variance or, in the case of XSB Prolog, supported call subsumption only for stratified programs. However, call subsumption has proven critical for (sub-)model generation as required for some kinds of program analysis (e.g. type analysis) and for semantic web applications such as RDF inference. At the same time, the lack of well-founded negation has prevented the use of call subsumption in producing residual programs, and has limited its use in semantic web applications that require negation (e.g. evaluation of OWL ontologies). This paper describes an engine for evaluating normal programs under the well-founded semantics (WFS) in which the evaluation method can be based on a mixture of call subsumption and call variance, chosen at the predicate level. The implementation has been thoroughly tested for both local and batched evaluation and is available in version 3.2 of XSB.

Keywords: Tabling, WAM.

Tabled evaluations can differ in their subgoal reuse strategy. Given a selected tabled subgoal G, answers may be resolved when there is a subgoal in the table that subsumes G, in which case *call subsumption* is used, or only when a variant of G is in the table, where *call variance* is used. Call variance preserves the instantiation patterns of selected subgoals in an evaluation, making it efficient for query-oriented applications and suitable for tabling meta-interpreters. Call subsumption, on the other hand, often evaluates only the most general subgoals, giving it a more bottom-up flavor. Call subsumption is therefore suitable for (sub-)model generation as required for stable model generation, for certain type analyses, or for some semantic web applications. Call subsumption is harder to implement than call variance and is usually not supported in tabling implementations. Previous versions of XSB implemented call subsumption for stratified programs [3], and allowed the strategy of subsumption or variance to be declared on a predicate basis. This paper describes how XSB's implementation of call subsumption is extended to support full well-founded semantics The extended implementation has been thoroughly tested, and is available in XSB version 3.2.

A Motivating Example. Computing A-box entailment from a standard OWL wine ontology (`www.w3.org/TR/2003/CR-owl-guide-20030818/wine`) provides a striking example of a use of call subsumption for a semantic web application. When translated into datalog by KAON2 (`http://kaon2.semanticweb.org`) the ontology is a highly recursive datalog program. Computing a query using call subsumption in XSB terminated correctly in 10 seconds, while call variance terminated with memory errors after 100+ seconds. The difference was due to the lack of relevancy in query evaluation: essentially the entire (sub-)model of the wine program needed to be constructed. Because call subsumption avoided recomputing subsumed calls, it saved space and time, and is quite competitive with special-purpose ontology tools [4]. While this is a single example, two conclusions are clear. Call subsumption can be critical for "bottom-up" computations that do not benefit from relevancy. Also, since ontologies in general require negation when they are translated into datalog, evaluating WFS using call subsumption is an important problem.

1 Implementation

We briefly describe the main ideas of the engine as implemented in the SLG-WAM of XSB. Because of space limitations, we must assume a general knowledge of tabling and its implementation. As background, [3] describes call subsumption in the SLG-WAM, while [5] describes the overall SLG-WAM architecture for WFS and [2] the data structures used for well-founded residual programs.

We begin by describing actions of the SLG-WAM on definite programs. When call variance is used, if a tabled subgoal G is new to an evaluation, it is associated with a generator choice point to backtrack through program clauses. If G was previously selected and completed, the engine simply backtracks through answers in an answer trie. If G was previouslyselected but is not completed a consumer choice point is created that will backtrack through answers using an *answer return list* for G. The answer return list is needed to backtrack efficiently through the dynamically changing answer trie.

Call subsumption extends the cases an implementation must support. Let subgoal G subsume a subgoal $G\theta$. If $G\theta$ is selected before G no special action is taken – $G\theta$ and G are evaluated just as with call variance, However if $G\theta$ is selected *after* G, two subcases arise. If G is completed, $G\theta$ simply backtracks through answers for G failing on those that do not unify with $G\theta$. If G is not completed, then a consumer choice point is created for $G\theta$ (as with call variance) along with a *subsumed subgoal frame* in the table. In addition, a special answer return list is created for $G\theta$ pointing to each answer in the answer trie for G that unifies with $G\theta$. When $G\theta$ consumes the final answer in its answer return list, the engine traverses the answer trie for G to generate a new answer return list for $G\theta$, consisting of answers that unify with $G\theta$ *and* were added after the previous list was generated for $G\theta$. Figure 1 schematically illustrates the relation of G and $G\theta$. [3] described an important optimization where nodes in an answer trie are associated with time stamps, and the time stamps manipulated to avoid unnecessary search through the trie when regenerating answer lists for subsumed subgoals. Note that this mechanism supports those stratified programs where no ground negative subgoal $G\theta$ occurs in the same SCC as G (i.e. the same set of mutually dependent subgoals).

To evaluate WFS, situations must be handled that arise when G and $not(G\theta)$ are mutually dependent (and neither is completed). The program

```
:- table win/1 as subsumptive.
win(X):- move(X,Y),tnot(win(Y)).
```

and query `win(X)` under varying extensions of `move/2` serves as a running example. (`tnot/1` is XSB's predicate for tabled negation).

Changes to DELAYING**.** Consider the extension `move(a,b) move(b,a)`. Evaluation of the goal `win(X)` creates the goals `win(a)` and `win(b)`. The two subsumed goals are created after `win(X)` is called and before it is completed, so for each subsumed subgoal a consumer choice point is created along with a subsumed subgoal frame which is inserted in a *consumer subgoal chain* in the subgoal frame for `win(X)`. However, since the program is non-stratified, both `win(a)` and `win(b)` are undefined in WFS and are treated as *conditional answers*, denoted `win(a):-tnot(win(b))` | and `win(b):-tnot(win(a))` | — where the "|" symbol indicates that the preceding literal is delayed. To represent delay lists when call subsumption is used, `tnot/1` must be changed to determine whether a negated goal is subsumed by another goal, so that the delay list will properly contain a ground literal, rather than the subsuming literal `tnot(win(X))`. To effect this, the goal *tnot(G)* determines whether there is a subsuming goal for *G*, and if not associates the delay element with *G*'s (producer) subgoal frame. Otherwise, *tnot(G)* determines whether the subsuming goal is completed or not, creates the subsumed subgoal frame if necessary, and associates the delay element with the subsumed subgoal frame for *G* – in this case `win(a)` or `win(b)`.

Changes to SIMPLIFICATION**.** Next, consider the extension `move(a,b) move(b,a) move(b,c)` and the evaluation of the query `win(a)`. In this case, no delaying is necessary for either call variance or call subsumption: `win(a)` does not subsume `win(b)` or `win(c)`. `win(c)` is determined to be false, and this false value causes `win(b)` to be determined to be true and `win(a)` false before any conditional answers are created. However, if call subsumption were used for the goal `win(X)`, all goals would be in the same mutually dependent SCC, so that DELAYING and SIMPLIFICATION must both occur.

SIMPLIFICATION operations are initiated in two cases [5]. When a subgoal G is completed with no answers (fails), any answers conditional on G must be deleted; and any answers conditional on *tnot(G)* must have *tnot(G)* removed from their delay lists. Similarly, when an unconditional answer A is derived, any answers conditional on A must be simplified. Each case can initiate a chain of SIMPLIFICATION operations, since removing an answer can cause a goal to fail; while removing a literal from a delay list can cause an answer (or a ground subgoal) to become unconditionally true. Figure 1 shows the supporting data structures. A subgoal G has an answer Ans in its answer trie along with a subsumed goal $G\theta$. Ans contains $tnot(Gdep)$ in its delay list. To perform simplification, the subgoal $Gdep$ contains a list of backpointers to each answer (such as Ans) containing $tnot(Gdep)$ in its delay list (conditional answers have backpointers similar to those for subgoals). In addition, pointers from delay lists to answers (through Delayinfo structures) and from answers to (producing) subgoals are used to traverse table space and propagate SIMPLIFICATION operations.

Both cases in which SIMPLIFICATION is initiated are affected by call subsumption. To handle the case initiated by a failing subgoal G, the engine checks the subgoal frame to see whether G has been declared to use call subsumption. If so, the engine must check whether there are any subgoals subsumed by G that now fail. The chain of subsumed subgoals is traversed, and each subsumed subgoal frame checked for a non-null backpointers cell. The subsumed subgoal is checked for backpointers, and a SIMPLIFICATION operation executed aif the subgoal fails. In our running example,

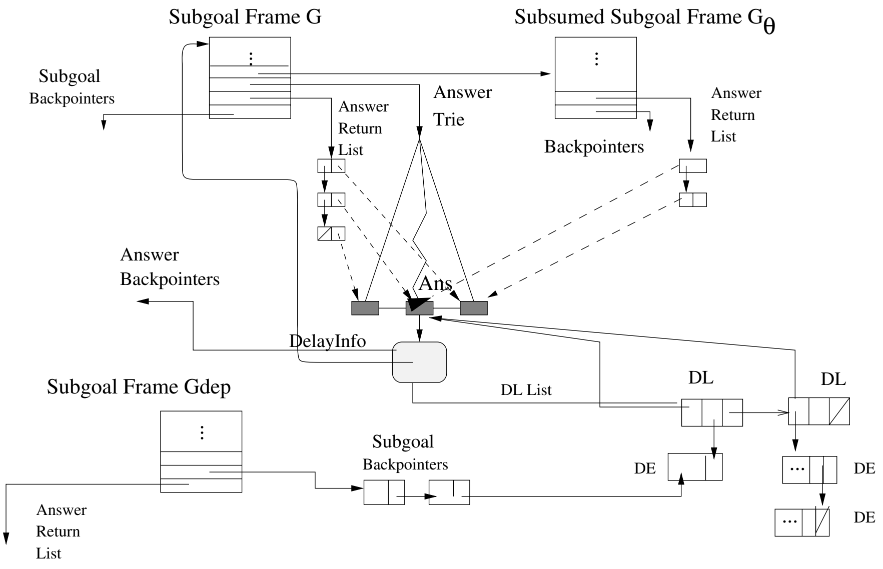


Fig. 1. Schematic Table Space for Call Subsumption with Conditional Answers

when the subsumed subgoal `win(c)` fails, a SIMPLIFICATION operation is performed using backpointers of `win(c)` to remove the answer `win(b):- tnot(win(c))` |, in turn propagating a SIMPLIFICATION operation to make the answer `win(a):- tnot(win(b)` | unconditional. Consider the case of the second simplification where an unconditional answer A is derived, other answers having either A or *tnot(A)* in their delay lists need to be simplified. Answers having A in their delay lists are obtained through the backpointers list off of the answer A itself, and so are not affected by call subsumption. On the other hand, obtaining answers having $tnot(A)$ in their delay list is more difficult using call subsumption, as they depend on the subgoal A which may be subsumed. As shown in Figure 1, there is a pointer from an answer to its producer (or variant) subgoal frame, but not to frames of any subsumed subgoals. However while A may not need to initiate simplification through its producing call, it may need to initiate simplification through a subsumed call. Again using our running example, when `win(b)` becomes true, the answer `win(a) :- tnot(win(b))` | must be deleted. This occurs through the backpointers of the subgoal `win(b)`, but as mentioned there is no pointer from the answer `win(b)` (for `win(X)`) to the subsumed subgoal frame for `win(b)`. Fortunately, the trie data structures make the check for possible subsumed subgoals efficient. Tabled subgoals are stored in a trie just as answers are, with subgoal frames (including those for subsumed subgoals) as leaves of the subgoal trie. A term is constructed from the answer substitution A, and trie indexing is used to determine whether A is also a subsumed subgoal frame: if so, simplification will be performed.

Performance. Tests of `win/1` provide information about the speed of basic operations. The table below illustrates CPU times in seconds and table space in bytes for `win/1` using call variance and call subsumption. When `move/2` is a chain, call subsumption is significantly slower than call variance. For the goal `win(1)` this is due to the fact that `tnot/1`, currently written largely in Prolog, is more complicated for call subsumption. In addition, call subsumption for the goal `win(X)` treats all subgoals as if they were in the same SCC, and must delay and simplify each answer. When `move/2` is a cycle, the time to add conditional answers dominates all strategies. For the goal `win(X)`, call subsumption must perform the same amount of delaying for the chain and for the cycle (and the same amount of delaying as call variance for the cycle). However, note that for the chain, 50,000 extra simplifications are performed in a negligible amount of time. Also, call subsumption requires slightly more table space than call variance, when there is no actual subsumption to exploit. With `win(X)` goals, call subsumption saves some table space, but unlike the wine example, the savings are limited in this example as there is at most a single answer per subsumed goal.

	call variance `win(1)`	call subsump. `win(1)`	call subsump. `win(X)`
50000 chain	0.19 / 5,582,396	0.53 / 5,982,596	1.0 / 3,653,620
50000 cycle	1.14 / 9,985,548	0.72 / 10,585,940	0.98 / 7,654,660

2 Discussion

In terms of related work, a global answer table has been implemented in YAP for definite programs [1], and allows the sharing of answers between different subgoals of a tabled predicate. Global tables and call subsumption can both reduce the size of tables, but each has advantages that the other does not. A global table can allow sharing of answers between subgoals that unify, even if neither subsumes the other. Call subsumption, on the other hand can reduce computation time as well as space.

The implementation described is intended to be robust. Accordingly, before distributing the described implementation in XSB, it was run on a suite of programs testing WFS, residual programs, and tabled constraints. The test suite contained over 12,000 lines of code, and was run on various platforms under local and batched evaluation for 32-bit and 64-bit compilations. The implementation described here makes minor changes to scheduling, relatively straightforward changes to table data structures and access routines, and more complex changes to simplification instructions. It is important to note that call subsumption does not affect a tabling system's mechanisms for suspending and resuming a computation – the aspect of tabling that is most intimately connected with the WAM data structures. This means that the approach described here is (in principle) applicable to systems such as Ciao, Mercury, B-Prolog and ALS that support tabling outside of the SLG-WAM.

References

1. Costa, J., Rocha, R.: One table fits all. In: PADL, pp. 195–208 (2009)
2. Cui, B., Swift, T., Warren, D.S.: From tabling to transformation: Implementing non-ground residual programs. In: Implementations of Declarative Languages (1999)
3. Johnson, E., Ramakrishnan, C.R., Ramakrishnan, I.V., Rao, P.: A space-efficient engine for subsumption based tabled evaluation of logic programs. In: 4th International Symposium on Functional and Logic Programming (1999)
4. Liang, S., Fodor, P., Wan, H., Kifer, M.: OpenRuleBench: An analysis of the performance of rule engines. In: WWW: Semantic Data Track (2009)
5. Sagonas, K., Swift, T., Warren, D.S.: An abstract machine for efficiently computing queries to well-founded models. JLP 45(1-3), 1–41 (2000)

Incremental Answer Completion in the SLG-WAM

Terrance Swift, Alexandre Miguel Pinto, and Luís Moniz Pereira

Centro de Inteligência Artificial, Universidade Nova de Lisboa
tswift@cs.sunysb.edu, amp@di.fct.unl.pt, lmp@di.fct.unl.pt

Abstract. The SLG-WAM of XSB Prolog soundly implements the Well-Founded Semantics (WFS) for logic programs, but in a few pathological cases its engine treats atoms as undefined that are true or false in WFS. The reason for this is that the XSB does not implement the SLG ANSWER COMPLETION operation in its engine, the SLG-WAM – rather ANSWER COMPLETION must be performed by post-processing the table. This engine-level omission has not proven significant for applications so far, but the need for ANSWER COMPLETION is becoming important as XSB is more often used to produce well-founded residues of highly non-stratified programs. However, due to its complexity, care must be taken when adding ANSWER COMPLETION to an engine. In the worst case, the cost of each ANSWER COMPLETION operation is proportional to the size of a program P, so that the operation must be invoked as rarely as possible, and when invoked the operation must traverse as small a fragment as possible of P. We examine the complexity of ANSWER COMPLETION; and then describe its implementation and performance in XSB's SLG-WAM such that the invocations of the operation are restricted, and which is limited in scope to Strongly Connected Components within a tabled evaluation's Subgoal Dependency Graph.

Designers of logic programming engines must weigh the usefulness of operations against the burden of complexity they require. Perhaps the best known example is the *occurs check* in unification. Prologs derived from the WAM do not perform occurs check between two terms, since its cost may be exponential in the size of the terms. Rather, the occurs check must be explicitly invoked through the ISO predicate `unify_with_occurs_check/2` or a similar mechanism. For evaluating normal programs using tabling, checking for certain positive loops involves similar considerations. While most positive loops can be efficiently checked, positive subloops within larger negative loops are more difficult to detect, and account for the complexity of evaluating a program P according to WFS, which is $atoms(P) \times size(P)$, where $atoms(P)$ is the number of atoms of P and $size(P)$ is the number of rules of P.

As implemented in XSB, the SLG-WAM detects positive loops between tabled subgoals so that answers are not added to a table unless they are true, or are involved in a loop through negation and so are undefined at the time of their addition (termed *conditional answers*). As shown in Theorem 1 below, this sort of evaluation can be done in time linear in $size(P)$. However, a situation can arise where certain conditional answers are later determined to be true or false. This determination may break a negative loop, which uncovers a positive loop and makes the answers false. Within SLG, this situation is addressed by the ANSWER COMPLETION operation, which is not implemented within the currently available version of the SLG-WAM. So far, the lack of ANSWER COMPLETION has not proven a problem for most programs. However, the SLG-WAM is increasingly being used to produce well-founded residues for highly non-stratified programs for applications involving intelligent agents (e.g. [2]), where the need for ANSWER COMPLETION is greater.

This paper examines issues involved in adding ANSWER COMPLETION to the SLG-WAM. We illustrate the situation of a positive loop begin uncovered when a negative loop is resolved through a concrete example, and then we provide a formal result on the contribution ANSWER COMPLETION makes to the complexity of computing WFS. We introduce an algorithm for efficiently performing ANSWER COMPLETION (subject to its complexity), and discuss performance results obtained by implementing it in the SLG-WAM. Due to space requirements, we must assume knowledge of tabled evaluation of WFS through SLG resolution [1] as well as certain data structures of the SLG-WAM [3].

Example 1. The following program is soundly, but not completely, evaluated by the SLG-WAM, where `tnot/1` indicates that tabled negation is used:

```
:- table p/1,r/0,s/0.
p(X):- tnot(s).      p(X):- p(X).
   s:- tnot(r).         s:- p(X).        r:- tnot(s),r.
```

The well founded model for this program has true atoms $\{s\}$ and false atoms $\{r, p(X)\}$. Recall that literals that do not have a proof and that are involved in loops over default negation are considered *undefined* in WFS. Unproved literals involved only in positive loops, i.e., without negations, are *unsupported* and, hence, *false* in WFS. Accordingly, `p(X)`, whose second clause fails, is *false*; however, a query to $p(X)$ in XSB indicates that $p(X)$ is *undefined*. The reason is that during evaluation the engine detects a strongly connected component (SCC) of mutually dependent goals containing $p(X)$, r and s, along with negative dependencies, and no answers for any of these goals. In such a situation, the SLG-WAM delays negative literals and continues execution. Here, the literal `tnot(s)` in the rule `p(X):- tnot(s)` is delayed, producing an answer `p(X):- tnot(s) |`, indicating that `p(X)` is conditional on a *delay list*, here `tnot(s)`. That answer is returned to the goal `p(X)` in the clause `p(X):- p(X)` and a conditional answer `p(X):- p(X) |` is derived. Later, a positive loop is detected for `r`, causing its truth value to become *false*. The failure of `r` causes `s` to become *true*, and SIMPLIFICATION removes the answer `p(X):- tnot(s) |`. At this stage, however, no further simplification is possible for `p(X) :- p(X) |`, which is now unsupported.

The ANSWER COMPLETION operation addresses such cases by detecting positive loops in dependencies among conditional answers. More precisely, ANSWER COMPLETION marks *false* sets of answers that are not *supported*: i.e. conditional answers for completed subgoals that contain only positive, and no negative dependencies in their delay lists. The creation of unsupported answers are uncommon in the SLG-WAM because its evaluation is *delay minimal* – that is, the engine performs no unnecessary DELAYING operations [4]. Delay minimality reduces the need for simplification of dependencies among answers, and thereby the chances of uncovering positive loops among answers, as with the answer `p(X):- p(X) |` above.

1 Complexity

We begin by showing that queries to programs that do not need ANSWER COMPLETION can be evaluated in $\mathcal{O}(size(P))$. Such programs include stratified ones, and also non-stratified programs that contain no positive loops within negative SCCs in their dynamic dependency graphs[1].

Theorem 1. *Let Q be a query to a finite ground normal program P. Under a cost model with constant time access to all subgoals, nodes, and delay elements of each SLG forest in an evaluation, and constant time access to each clause in P, a partial SLG evaluation that does not perform* ANSWER COMPLETION *can be constructed that is linear in the size of P.*

The algorithm ITERATE ANSWER COMPLETION below iteratively applies ANSWER COMPLETION operations, calling Check Supported Answers() to perform a check for positive loops. Check Supported Answers() is an adaptation of Tarjan's algorithm for SCC detection (cf. `http://en.wikipedia.org/wiki/Tarjan's_strongly_connected_components_algorithm`), which is linear in $size(P)$. Note that in the worst case, ANSWER COMPLETION operations iteratively need to be applied, and that each time it is applied, a single atom would be found *false*. In that case, program evaluation would have a cost proportional to $atoms(P) \times size(P)$, which is equivalent to the known complexity for WFS.

2 Implementation of ANSWER COMPLETION

Within an SLG evaluation, a tabled subgoal can be marked as *complete*, which indicates that all possible answers have been produced for the subgoal, although SIMPLIFICATION and ANSWER COMPLETION operations may remain to simplify or delete conditional answers. Completed subgoals do not require execution stack space, but only table space, so that completing subgoals as early as possible is a critical step for engine efficiency. Accordingly the SLG-WAM performs *incremental completion* via a completion instruction, which maintains information about mutually dependent sets of subgoals (SCCs), and completes each SCC when all applicable operations have been performed. In addition to marking each subgoal S in an SCC as complete, if S failed (has no answers) the completion instruction may initiate SIMPLIFICATION for conditional answers that depend negatively on S. In terms of ANSWER COMPLETION, observe that any positive loop among the delayed literals of conditional answers must be contained within the SCC being completed, as each delayed literal was a selected literal before it was delayed. This incremental approach has several benefits. Performing ANSWER COMPLETION operation within the completion instruction restricts the space that any such operation needs to search. In addition, performing ANSWER COMPLETION after all other SIMPLIFICATION operations have been performed on answers within the SCC similarly reduces search space. As a final optimization, ANSWER COMPLETION is not required unless delaying has been performed within the SCC, a fact that

[1] The proof of Theorem 1 is contained in an appendix of a fuller version of this paper available on request.

is easily maintained using data structures in the SLG-WAM's *Completion Stack*, which maintains information about SCCs. The pseudo code for Iterate Answer Completion(), which traverses all subgoals in the SCC using the *Completion Stack*, and checks each answer for support, deleting unsupported answers from the table and invoking SIMPLIFICATION operations, is presented in Figure 1. SIMPLIFICATION may remove further negative loops, and uncover new unsupported other answers as a side-effect. In such case, the ANSWER COMPLETION procedure should be executed once more, and this is guaranteed by the use of the $reached\_fixed\_point$ flag. A fixed-point is reached when all answers within the scope of the SCC are known to be supported.

```
Algorithm Iterate Answer Completion($CompletionStack$)

$reached\_fixed\_point = FALSE$;
while not $reached\_fixed\_point$
    $reached\_fixed\_point = TRUE$;
    foreach subgoal $S$ in the $Completion\ Stack$
        foreach answer $A$ for subgoal $S$
            if not Check_Supported_Answer($A$)          /* $A$ is unsupported */
                $reached\_fixed\_point = FALSE$;
                delete $A$;
                propagate $A$'s deletion's simplifications;
```

Fig. 1. Algorithm Iterate Answer Completion

Check Supported Answer. This procedure (Figure 2) does the actual check of whether a (positive) answer is unsupported. It detects positive loops whenever it encounters an answer that has already been visited and which is in the SCC. In this case, the algorithm terminates returning $FALSE$ to indicate the answer is unsupported. On the other hand, if the answer has been visited but is not part of the SCC, it means such answer has been produced during some other branch of query-solving and was therefore, rightfully supported and stored in the table: the algorithm terminates returning $TRUE$. Checking a non-visited answer consists of 1) marking it as visited; 2) adding it to the state of the search (stored in the $Completion\ Stack$); and then 3) traversing all the Delay Elements (literals) of the Delay Lists for the answer recursively checking each in turn for supportedness. Whenever an answer is determined to be unsupported, all Delay Lists containing (Delay Elements that reference) it are deleted, which may cause further simplification and iterations of ANSWER COMPLETION.

The above algorithm has been implemented within the completion instruction of XSB. Full performance analysis is still underway. Preliminary results indicate advantages of our heuristics: traditional benchmarks like `win/1` either do not use SIMPLIFICATION or use it seldom so that there is no overhead for ANSWER COMPLETION. A stress test that performs a large number of repetitions of Example 1 shows an overhead of at most 18%. Example 1 is actually representative of the typical situation where ANSWER COMPLETION is needed. This is so because it contains (at least) two rules for some literal (in this case $p(X)$) where the first one depends on a loop through negation (rendering $p(X)$ *undefined*) and the second one depend on a positive loop, which

```
Algorithm Check Supported Answer($Answer$)

if $Answer$ has already been visited
    if $Answer$ is in the $SupportCheckStack$ return FALSE;
    else return TRUE;
else
    mark $Answer$ as visited;
    push $Answer$ onto the $SupportCheckStack$;
    mark $Answer$ as $supported\_unknown$;
    foreach Delay List $DL$ for $Answer$
        if $Answer$ is $supported\_true$ exit loop;
        mark $DL$ as $supported\_true$;
        foreach Delay Element $DE$ in the Delay List $DL$
            if $DL$ is not $supported\_true$ exit loop;
            if $DE$ is positive and it is in the $SupportCheckStack$
                recursively call $Check\ Supported\ Answer(Answer\ of\ DE)$
                if $Answer\ of\ DE$ is not $supported\_true$
                    mark $DL$ as $supported\_false$;
    if $DL$ is $supported\_false$
        remove $DL$ from $Answer$'s DLs list
        if $Answer$'s DLs list is now empty
            delete $Answer$ node;
            simplify away unsupported positives of $Answer$;
        else mark $Answer$ as $supported\_true$;
    if the $Answer$ node was deleted return $TRUE$;
    else return $FALSE$;
```

Fig. 2. Algorithm Check Supported Answer

is unsupported. The "undefinedness" coming from the first clause is passed on to the $p(X)$ in the body of the second one. Only ANSWER COMPLETION can then be used to clean away the delay list with $p(X)$ from the answer coming from the second clause for $p(X)$. The "pathological" nature of this example follows from the, until now, XSB's SLG-WAM inability to rightfully detect and simplify away unsupported literals such as $p(X)$.

3 Conclusions

WFS is used in an increasing number of applications, from intelligent agents, to inheritance in object logics, to supply-chain analysis. However, the abstract complexity of WFS is a concern when embedding into the semantic core of a programming language like Prolog. Theorem 1 shows that the non-linearity of WFS can be separated from other parts of an engine for WFS; and the optimizations of the algorithm presented here, together with the preliminary performance results, underscore the suitability of WFS for general-purpose programming.

References

1. Chen, W., Warren, D.S.: Tabled Evaluation with Delaying for General Logic Programs. JACM 43, 20–74 (1996)
2. Pereira, L.M.: G Lopes. Prospective logic agents. In: Neves, J., Santos, M.F., Machado, J.M. (eds.) EPIA 2007. LNCS (LNAI), vol. 4874, pp. 73–86. Springer, Heidelberg (2007)
3. Sagonas, K., Swift, T.: An abstract machine for tabled execution of fixed-order stratified logic programs. TOPLAS 20(3), 586–635 (1998)
4. Sagonas, K., Swift, T., Warren, D.S.: The limits of fixed-order computation. Theoretical Computer Science 254(1-2), 465–499 (2000)

Research Summary: Tabled Evaluation for Transaction Logic Programs

Paul Fodor

State University of New York at Stony Brook, USA

1 Introduction

In my thesis, I present problems and techniques in tabling Transaction Logic ($\mathcal{TR}$). $\mathcal{TR}$ is an extension of classical logic programming with backtrackable state updates and it has a top-down evaluation algorithm similar to Prolog's SLD derivation extended with execution paths of states instead of a single global state. This backward chaining algorithm can be very inefficient by re-computing the same transactional queries more than once, or can enter into infinite loops by visiting the same paths of states an infinite number of times when computing answers to recursive programs. We solve these problems by memoizing (caching) the calls, call initial states, unifications (answers) and return states in a searchable structure for the Sequential Transaction Logic, respective building a graph for the query and tabling the nodes ready for current execution for the Concurrent Transaction Logic. Important problems of tabling $\mathcal{TR}$ are to store, index, update, query and resume states into memory. I implemented and measured the efficiency of multiple data structures used in tabling programs with backtrackable updates in XSB Prolog. My thesis studies the data structures and their performance for various applications of TR, such as, artificial intelligence planning, NP-complete graph algorithms (Hamiltonian cycle, clique, shortest consuming paths, connected components) and active databases. One of the most promising techniques was storing logs (i.e., inserts and deletes relative to a materialized state) into individual tries (optimized for querying), while keeping a global page trie as a common index for restarting.

2 Tabling Transaction Logic

Logic programming tabling is notoriously inefficient when working with lists and other structured terms. Most of the time, such complex terms are non-discriminate arguments or terms not used for the tabling, i.e., large states represented as lists and accumulating results. An intuitive and simple solution for this problem is to use the program database for such large arguments, and backtrackable updates to change them. $\mathcal{TR}$ [1,2,3] is a general logic designed specifically for the rule-based paradigm intended to provide a declarative account for state-changing actions. The elementary backtrackable transitions **ins/1** and **del/1** specify basic updates of the current state of the database executed by an oracle that takes the database into a new state by inserting or deleting an atom in the extended database.

Complex formulas are built out of atomic formulas and elementary transitions using connectives and quantifiers (similar to those in logic programming, with the exception of the concurrent conjunction operator "|". The truth of $\mathcal{TR}$ formulas is determined over paths, where a path is a finite sequence of states. A formula π is true over a path $<s_1, ..., s_n>$ if it can be executed starting with state s_1 changing the database into states s_2, s_3,.. and finally terminate at the state s_n.

Tabling for logic programming [4,5] is a technique to reuse branches of the computation for all the calls of equivalent queries (variant or subsumtive equivalence). For the tabling of transaction logic we pair calls and answer unifications with their initial and returning states. This call-answer table is consulted whenever a new call to a tabled predicate is issued. If the call issued in an initial database state is similar to a tabled call issued in the same initial state, then the set of answers and returning states may be used to satisfy the call. If there is no entry in the call table for the call, then it is entered into the table and is resolved against program clauses using the SLD-like resolution. As each answer is derived during this process, is inserted into the table entry associated with the call. After the answer is added to this set, it is scheduled to be returned to all equivalent previous calls stored in a lookup table. If no answer was found, then the evaluation fails and the execution backtracks.

Common issues in tabling of transaction logic are: querying facts in the current state, checking if the current state was encountered before followed by insertion, resuming suspended calls by also changing the current state with a return state from the answer tables, tabling logs. vs. materialized states and optimizations. We implemented and measured the performance of various data structures for storing states: from states as ordered lists, materialized states stored on a trie with explicit state identifiers, indexing states with hash incremental functions, to logs as tries and global state trie.

Our experiments on applications of $\mathcal{TR}$ in planning and Hamiltonian paths showed that the performance of the system evolved from infeasible for states stored as lists, to a few seconds for logs represented as tries. Space is still a very important issue because tries are not the most optimal representation for sets of states. Selective materialization might increase the sharing, but it imposes a heavy toll on computation (for instance, checking for existence of a state <inslog1, dellog1, relativeState1> in the pool of previously seen states requires materialization and comparison of multiple states in time liniar with the size of the state.

The $\mathcal{TR}$ tabling technique can be extended to tabling of Concurrent Transaction Logic programs by building a graph for the program and the query, and memoizing the "hot" vertices (e.g., vertices possible to execute). We currently developed this technique only for the propositional logic programs, remaining to extend it to the predicative case.

References

1. Bonner, A.J., Kifer, M.: An overview of transaction logic. Theoretical Computer Science 133(2), 205–265 (1994)
2. Bonner, A.J., Kifer, M.: Transaction logic programming (or, a logic of procedural and declarative knowledge). Technical report, University of Toronto (1995)
3. Bonner, A.J., Kifer, M.: Concurrency and communication in transaction logic. In: JICSLP, pp. 142–156 (1996)
4. Tamaki, H., Sato, T.: Old resolution with tabulation. In: ICLP (1988)
5. Warren, D.S.: Memoing for logic programs. Communications ACM 35(3), 93–111 (1992)

Research Summary: Logic Programming for Massively Distributed Systems

Michael P. Ashley-Rollman

Carnegie Mellon University, Pittsburgh, PA 15213
mpa@cs.cmu.edu

1 Introduction and Problem Description

My research focuses on finding a better way to program massively distributed systems. Programming these systems is crucial as we move into a world where they are increasingly necessary. Unforunately, existing concurrency models in modern languages are very difficult for programmers to understand and to reason about. Many programmers have an extremely hard time writing and debugging concurrent programs, let alone massively distributed ones. Race conditions, in particular, are among the most challenging bugs to find, understand, and resolve.

I am currently looking at this problem in the context of Claytronics [5], a system of modular robots intended to create a programmable material. These robots form *ensembles*, network-varying massively distributed systems, with the added complexities of moving nodes and real world uncertainties. These ensembles are expected to eventually contain millions of nodes.

2 Background and Overview of the Existing Literature

Some work on using logic programming for distributed systems has come out of Berkeley in recent years. In particular, P2 [6] and SNLog [2] use a logic programming approach for overlay networks and sensor networks, respectively. These systems are not designed to handle the frequent network topology changes present in ensembles. Furthermore, they lack the formal semantics necessary to prove things about their programs. Other approachs, outside of logic programming, also exist and have similar short-comings.

3 Current Status of the Research

I am exploring alternative programming models through the development of new programming languages and paradigms designed around concurrency. I have created a Datalog-like programming language called Meld [1], designed to run on ensembles. Meld is based upon the idea that a programmer should program an ensemble as a single entity and not need to worry about implementation details such as where various parts of the program should execute or what messages need to be sent between nodes. Instead, the programmer focuses on the logic of the program, expressing this as a set of rules for changing the state of the ensemble based upon the current state. The programmer is able to implicitly specify which nodes the state is stored on, leaving Meld to collect state from various nodes when rules apply and to update the state across nodes accordingly. Additionally, when the state of the ensemble changes, Meld automatically updates the program state by determining which rules no longer apply and reverting the state changes they caused. Among other features, this provides a basic fault-tolerant behavior to any Meld program.

4 Preliminary Results

Meld has been successful in the Claytronics project. It was used to implement a number of complex applications, including an ensemble localization program [4], which builds a shared coordinate system, and a shape change program [3]. The Meld implementation of the shape change program has been proven correct and has been demonstated running on one million simulated nodes.

5 Open Issues and Expected Achievements

Unfortunately, Meld still has many shortcomings, particularly revolving around expressivity. Meld lacks any linear or temporal components, inhibiting the ability to write programs with dynamic state. It is not clear how to implement a temporal component in Meld given that modules may move at any time, possibly preventing the derivation of some facts at a given time, with no obvious way to derive them later. Adding a linear component is equally challenging, given how Meld automatically maintains state. Meld automatically derives and underives facts as the observed state of the physical world changes. How this would extend to linear facts is unclear.

I am interested in exploring approaches for supporting these sorts of features in Meld. I have experience trying to implement algorithms with dynamic state in them and have found Meld to be an impractical way of doing this. Finding a solution would greatly increase the utility of Meld in programming ensembles as it would permit common types of algorithms, such as gradient decent, to be efficiently implemented. In addition to solving these problems, I am also interested in other ways of enhancing Meld.

References

1. Ashley-Rollman, M.P., Lee, P., Goldstein, S.C., Pillai, P., Campbell, J.D.: A language for large ensembles of independently executing nodes. In: Hill, P.M., Warren, D.S. (eds.) ICLP 2009. LNCS, vol. 5649, pp. 265–280. Springer, Heidelberg (2009)
2. Chu, D., Tavakoli, A., Popa, L., Hellerstein, J.: Entirely declarative sensor network systems. In: Proceedings of Very Large Data Bases (2006)
3. Dewey, D., Srinivasa, S.S., Ashley-Rollman, M.P., Rosa, M.D., Pillai, P., Mowry, T.C., Campbell, J.D., Goldstein, S.C.: Generalizing metamodules to simplify planning in modular robotic systems. In: Proceedings of IEEE/RSJ 2008 International Conference on Intelligent Robots and Systems IROS 2008, Nice, France (September 2008)

4. Funiak, S., Pillai, P., Ashley-Rollman, M.P., Campbell, J.D., Goldstein, S.C.: Distributed localization of modular robot ensembles. In: Proceedings of Robotics: Science and Systems (June 2008)
5. Goldstein, S.C., Mowry, T.C.: Claytronics: An instance of programmable matter. In: Wild and Crazy Ideas Session of ASPLOS, Boston, MA (October 2004)
6. Loo, B.T., Condie, T., Garofalakis, M., Gay, D.E., Hellerstein, J.M., Maniatis, P., Ramakrishnan, R., Roscoe, T., Stoica, I.: Declarative networking: language, execution and optimization. In: Proc. of the 2006 ACM SIGMOD int'l conf. on Management of data, pp. 97–108. ACM Press, New York (2006)

Research Summary: Extending Elimination Algorithms for Functional Constraints to Solve Two Integer Variables per Inequality

Chendong Li

Department of Computer Science, Texas Tech University
2500 Broadway, Lubbock, TX 79409, USA
chendong.li@ttu.edu

1 Introduction and Problem Description

Binary constraint is an important constraint class that has been studied in both Constraint (Logic) Programming community [5] and Mathematical Programming community [3,4]. In [6] we studied *Functional Constraints* in Binary Constraint Satisfaction Problems (BCSPs) and developed efficient variable elimination algorithms based on variable substitutions (For details please refer to [6]). On the basis of our variable elimination algorithms for *Functional Constraints*, we further explored a special case of BCSPs that consisted of only *Bi-Functional Constraints*, which were called $\mathcal{I}$ *Constraints* in our experiments. Afterward we extended our elimination algorithms to solve integer programs with two variables per inequality. These problems are known as TVPIs [1] that have been studied in [3,4].

To solve TVPI systems with our elimination algorithms, we have to do following mappings between TVPIs and $\mathcal{I}$ *Constraints*: 1) We need to establish the one-to-one corresponding relationship between a TVPI and an $\mathcal{I}$ *Constraint*. This can be done by using the corresponding binary equation (which is an $\mathcal{I}$ *Constraint*) of the TVPI to represent the boundary of its solution space for each TVPI. 2) We need to find an effective way to map the $\mathcal{I}$*nfinite* $\mathcal{D}$*omain* of the TVPI systems to the $\mathcal{F}$*inite* $\mathcal{D}$*omain* of BCSPs without loosing solutions. By investigating properties of TVPIs, we observed a regular pattern repeat of domains of the two variables involved in any TVPIs. Based on this characteristic, we proposed a new representation form for linear inequalities — `segments`. Under this representation, we employed novel operations for `segments` and further developed our new elimination algorithms based on Motzkin Elimination over real variables.

2 Research Goals and Current Status

The purpose of this research is to develop a system of efficient variable elimination algorithms for a specific class of binary constraints, in particular to obtain

[1] A TVPI is a binary linear inequality that can be written in the form $ax \leq by + c$ where $a, b, c \in \mathbb{Q}$ $x, y \in \mathbb{Z}$.

fast elimination algorithms to improve the state-of-the-art problem solving techniques. In addition, we hope to apply our new elimination algorithms to different scientific fields. Our short goal is to compare the performance of our algorithms with ILOG CPLEX 9.1 optimizer. We anticipate that our elimination algorithms can perform better than CPLEX under certain conditions.

As there are no benchmarks on TVPIs, we need to design our own problem generator. By now we have implemented a TVPI system generator to generate random TVPIs with given parameters. In the ILOG CPLEX 9.1, we have tested the random TVPI systems. In terms of our algorithms, we have implemented the main part of our elimination algorithms. After implementing all the algorithms, we will carry out empirical studies on the new algorithms.

3 Preliminary Results and Open Issues

For BCSPs that consist of only $\mathcal{I}$ *Constraints*, we had obtained preliminary results on DELL PowerEdge 1850 (two 3.6GHz Intel Xeon CPUs) in Linux. From experimental results, we first confirmed the conclusion that $\mathcal{I}$ *Constraints* were polynomially solvable made by P. David in [2] and the worst case time complexity was $\mathcal{O}(ed)$ proposed by P. V. Hentenryck etc. in [5]. Moreover, we concluded that our algorithms were much faster that the general solver implemented based on AC2001/3.1 [1].

On the tests on random generated TVPI systems, although we observe ILOG CPLEX 9.1 can solve the problems pretty fast, we expect our algorithms are able to perform better than CPLEX. However, an apparent problem of our algorithms is the representations of TVPIs, which may be extremely large especially when a TVPI system has n variables with more than $n * (n-1)/2$ constraints. Furthermore, very large coefficients can result in out of memory for the system. How to represent TVPIs with more efficiency is of great importance when we intend to solve any TVPI systems, which means coefficients can be as large as possible.

References

1. Bessiere, C., Regin, J.C., Yap, R.H.C., Zhang, Y.: An Optimal Coarse-grained Arc Consistency Algorithm. Artificial Intelligence 165(2), 165–185 (2005)
2. David, P.: When functional and bijective constraints make a CSP polynomial. In: Bajcsy, R. (ed.) Proceedings of IJCAI 1993, vol. 1, pp. 224–229. Morgan Kaufmann, San Fransisco (1993)
3. Sewell, E.C.: Binary integer programs with two variables per inequality. Math. Program. 75, 467–476 (1996)
4. Simon, A., King, A., Howe, J.: Two Variables per Linear Inequality as an Abstract Domain. In: Leuschel, M.A. (ed.) LOPSTR 2002. LNCS, vol. 2664, pp. 71–89. Springer, Heidelberg (2003)
5. Van Hentenryck, P., Deville, Y., Teng, C.M.: A Generic Arc-consistency Algorithm and its Specializations. Artificial Intelligence 58, 291–321 (1992)
6. Zhang, Y., Yap, R.H.C., Li, C., Marisetti, S.: Efficient Algorithms for Functional Constraints. In: Banda, M., Pontelli, E. (eds.) ICLP 2008. LNCS, vol. 5366, pp. 606–620. Springer, Heidelberg (2008)

Research Summary
A Cognitive Architecture for a Service Robot: An Answer Set Programming Approach

Jianmin Ji

Multi-Agent Systems Lab., School of Computer Science and Technology,
University of Science and Technology of China, HeFei, 230027, P. R. China
jizheng@mail.ustc.edu.cn

1 Introduction

Service robot is one of the most promising directions of Robotics and full of challenges. Most of current work in Robotics concentrate on low-level functions, while in AI there are notable achievements on high-level functions. It would be interesting to integrate state of the art AI (particularly, KR) techniques and test if they are sufficient for developing a "good enough" service robot. New challenges will be identified and attacked on the basis of this investigation.

After more than twenty years of research, Answer Set Programming (ASP) has become a popular tool for knowledge representation and reasoning. Gelfond [1] suggested to use ASP for the design and implementation of deliberative agents. Following the perspective, we propose a cognitive architecture for our home service robot [2] in Fig. 1.

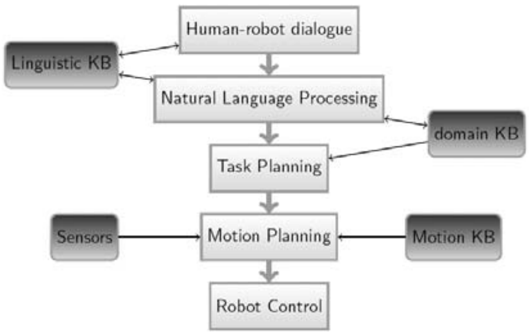


Fig. 1. An architecture for a home robot

Task Planning is a key part of the architecture, which is used to (1) access various kinds of information provided by users from Natural Language Processing; (2) compute a plan to fulfill the command with the help of information proved by the user; (3) pass sequences of atomic actions to Motion Planning.

We implemented it by ASP, thus providing us a versatile and robust platform for integrating modern KR techniques. Another benefit is that, both connections of Task Planning are close and smooth, as logical formulas generated from Natural Language Processing can be easily imported to it, and sequences of atomic actions can be easily understood by Motion Planning, which helps the robot work more automatically and makes the design of the robot more easily. To the best of our knowledge, we are the first one in this kind, and the only team that constructs the architecture based on ASP.

We have implemented basic components of the architecture. Currently, the robot can accomplish the command with the help of information proved by users in the form of natural language. For example, the user said "The book is on the table. Give Jim the book.". From Natural Language Processing, we can get $i\_on(book, table)$ and $g\_give(agent, Jim, book)$. Task Planning is implemented as a answer set program with a KB for classical planning, the command is translated to the corresponding goal state, if we have $location(table, 5, 0)$ in KB, then we can derive $location(book, 5, 0)$. cmodels is used to compute the answer set, which will contain a plan: $move(5, 0), catch(book, 1), move(0, 2), putdown(3)$. Then we pass this sequence of actions to Motion Planning to accomplish the task.

We have successfully qualified for a participation in the @home competition of RoboCup09[1]. The aim of RoboCup@Home is to foster mobile autonomous service robotics and natural human-robot interaction. There is no standard scenario, but something that people encounter in daily life. The competition is a series of tests, which will steadily increase in complexity. We can test the idea and compare with other work in the competition.

So far the work is preliminary, but as a versatile and robust platform, we can integrate many modern KR techniques to solve problems for service robot.

Diagnosis [3] is one of our next goals. There are many uncertainties (external events and actions may fail) during the execution, diagnosis can explain the discrepancy between abnormal observations and correct system behavior, which helps the robot fix the problem. Another important work is to extract ASP rules from tradition knowledge bases automatically, which can ease the design of the robot and make it more flexible.

References

1. Gelfond, M.: Answer set programming and the design of deliberative agents. In: Demoen, B., Lifschitz, V. (eds.) ICLP 2004. LNCS, vol. 3132, pp. 19–26. Springer, Heidelberg (2004)
2. Chen, X., Ji, J., Jiang, J., Jin, G.: Wrighteagle team description for robocup@home 2009. Technical report, Department of Computer Science and Technology, University of Science and Technology of China (2009), http://wrighteagle.org/en/robocup/atHome/
3. Eiter, T., Faber, W., Leone, N., Pfeifer, G.: The diagnosis frontend of the dlv system. AI Communications 12(1-2), 99–111 (1999)

[1] http://www.ai.rug.nl/robocupathome/

Research Summary: Termination of CHR

Paolo Pilozzi*

Dept. of Computer Science, K.U. Leuven, Belgium

1 Problem Description

Constraint Handling Rules (CHR) [7] is a high-level programming language, designed for the easy implementation of custom constraint solvers. CHR is a language-extension, built on top of Java, C and half a dozen Prolog-variants. The language is currently used in many projects [7].

In contrast to earlier approaches, CHR has a "no box"-approach and thus can implement any sort of solvers. Of course this affects the efficiency of the obtained solver. In order to obtain the same level of efficiency, we need analysis tools to detect the properties of a CHR program that lead to optimizations [7].

In CHR, termination is one of the most important properties of a program. Other important properties, such as confluence can be decided [7] given termination. But also for the programmer this is an important property. CHR is a multi-headed language in which loops are often much more difficult to detect than in single-headed languages such as Prolog.

2 Goal of the Research

Central to my research is the development of new, scalable techniques for automated termination analysis of CHR programs. Until recently, for CHR, only one approach existed [1], applicable only to CHR programs without propagation. I will therefore search for new conditions applicable to general CHR programs.

Usually for declarative languages, one designs necessary and sufficient conditions to characterize the termination behavior. Since such conditions are usually difficult to automate, one then develops from these conditions, sufficient conditions that are suitable for automation. The current approaches in LP use polynomial interpretations [2]. Since the atoms and termination conditions of a CHR program are similar to those of an LP program, we want to apply this technique as well. To find an interpretation validating the conditions, I intend to use a constraint-based approach [2]. In a next phase I want to refine our approach with the dependency pair approach from [3]. To obtain a scalable approach, I want to consider loops. In LP, these are detected using a strongly connected component (SCC) analysis. Such an approach seems also feasible for CHR.

Still many practical programs are likely to be out of scope of the aforementioned techniques. Examples of such programs are programs with bounded increase and programs which terminate for subsets of the considered queries. We intend to solve these issues by developing new techniques.

* Supported by I.W.T. Flanders - Belgium.

3 Current Status

By now, we studied the theoretical aspects regarding termination of CHR, thoroughly [4,8]. This lead to conditions discussed in [4] proving large classes of practical programs terminating. On the basis of these conditions, we derived verifiable conditions, implemented in a system called CHRisTA [5]. Our approach already scales to large(r) programs by using a SCC analysis.

4 Recent Advances and Expected Achievements

When using a more expressive polynomial form [2], we can improve our analysis considerably. By integration of the underlying principles of the Dependency Pair approach [3], we hope to be able to handle an important class of problems, implementing non-primitive recursion. In that way we will obtain a state of the art termination analyzer for CHR, such as these exist for LP today.

To be able to handle programs with bounded increase or that terminate only for subsets of the considered queries, we have developed a new approach [6]. Using this approach, we are developing an automated analysis for CHR on top of LP, to obtain as such the most powerful integrated termination tool for CHR(LP).

References

1. Frühwirth, T.: Proving Termination of Constraint Solver Programs. In: New Trends in Contraints, Joint ERCIM/Compulog Net Workshop, Selected papers, pp. 298–317 (2000)
2. Thang Nguyen, M., De Schreye, D.: Polynomial Interpretations as a Basis for Termination Analysis of Logic Programs. In: Gabbrielli, M., Gupta, G. (eds.) ICLP 2005. LNCS, vol. 3668, pp. 311–325. Springer, Heidelberg (2005)
3. Nguyen, M.T., Giesl, J., Schneider-Kamp, P., De Schreye, D.: Termination Analysis of Logic Programs Based on Dependency Graphs. In: King, A. (ed.) LOPSTR 2007. LNCS, vol. 4915, pp. 8–22. Springer, Heidelberg (2008)
4. Pilozzi, P., De Schreye, D.: Termination analysis of CHR revisited. In: ICLP 2008: Proceedings of the 24th International Conference on Logic Programming, pp. 501–515 (2008)
5. Pilozzi, P., De Schreye, D.: Automating Termination Proofs for CHR. In: ICLP 2009: Proceedings of the 25th International Conference on Logic Programming (accepted for publication, 2009)
6. Pilozzi, P., De Schreye, D.: Termination Analysis by Invariance Relations. In: ICLP 2009: Proceedings of the 25th International Conference on Logic Programming (accepted for publication, 2009)
7. Sneyers, J., Van Weert, P., Schrijvers, T., De Koninck, L.: As Time Goes By: Constraint Handling Rules – A Survey of CHR Research between 1998 and 2007. Submitted to Theory and Practice of Logic Programming (2009)
8. Voets, D., Pilozzi, P., De Schreye, D.: A new approach to termination analysis of CHR. In: King, A. (ed.) LOPSTR 2007. LNCS, vol. 4915, pp. 28–42. Springer, Heidelberg (2008)

Research Summary: Intelligent Natural Language Processing Techniques and Tools

Alessio Paolucci

Department of Computer Science
University of L'Aquila, Coppito 67100, L'Aquila, Italy
alessio.paolucci@univaq.it

1 Summary

My research path started with my master thesis (supervisor Prof. Stefania Costantini) about a neurobiologically-inspired proposal in the field of natural language processing. In more detail, we proposed the "Semantic Enhanced DCGs" (for short SE-DCGs) extension to the well-known DCGs to allow for parallel syntactic and semantic analysis, and generate semantically-based description of the sentence at hand. The analysis carried out through SE-DCG's was called "syntactic-semantic fully informed analysis", and it was designed to be as close as possible (at least in principle) to the results in the context of neuroscience that I had revised and studied. As proof-of-concept, I implemented the prototype of semantic search engine, the Mnemosine system. Mnemosine is able to interact with a user in natural language and to provide contextual answer at different levels of detail. Mnemosine has been applied to a practical case-study, i.e., to the WikiPedia Web pages. A brief overview of this work was presented during CICL 08 [1].

With the admission to the Ph.D track (again under the supervision of Prof. Stefania Costantini), the research path was redefined so as to consider the many interesting topics that emerged from previous experience and from open problems in the field. In fact, many intelligent systems have to deal with knowledge expressed in natural language, either extracted from books, web pages and documents in general, or expressed by human users. Knowledge acquisition from these sources is a challenging matter, and many attempts are presently under way towards automatically translating natural language sentences into an appropriate knowledge representation formalism [2]. The selection of a suitable formalism plays an important role but first-order logic, that would under many respects represent a natural choice, is actually not appropriate for expressing various kinds of knowledge, i.e., for dealing with default statements, normative statements with exceptions, etc. Recent work has investigated the usability of non-monotonic logics, like Answer Set Programming (ASP)[3].

Translating natural language sentences into a logic knowledge representation is a key point on the applications side as well. In fact, designing applications such as semantic search engines implies obtaining a machine-processable form of the extracted knowledge that makes it possible to perform reasoning on the data so as to suitably answer (possibly in natural language) to the user's queries, as such an engine should interact with the user like a personal agent. We have practically demonstrated in previous work [1] that for extracting semantic information from a large dataset like wikipedia, a reasoning process on the data is needed, e.g., for semantic disambiguation of concepts.

A central aspect of knowledge acquisition is related to the automation of the process. Recent work in this direction has been presented in [2] [4] and [3].

At this point (first year of Ph.D), the present goal of my research is to investigate the automatization of the translation from natural language sentences to answer set theories.

To this aim, I am considering to use and enhance SE-DCG grammars, with some advantages: efficient parsing, 'on the flight' semantic analysis of sentences (that performs significant disambiguation in many cases), and automatic generation of lambda-calculus expressions from template ones, thus improving the effectiveness and generality of the translation process. Adopting SE-DCG's results in a fully logical framework, though as future work we envisage an integration of positive aspects of CCG's and SE- DCG's. My recent works outlines a method for translating natural language sentences into ASP, so as to be able to reason on the extracted knowledge. In particular, I have extended the [3] approach by adopting a semantically enhanced efficient context-free parser and by introducing a new more abstract intermediate representation to be instantiated on practical cases. Also, I have outlined a fully automated translation methodology based upon our previous work [1].

The main expected achievement is for now the development of a solid theoretical and practical framework to cope with the translation from natural language sentences to ASP theory in a fully automatized way.

References

1. S. Costantini, Paolucci, A.: Semantically augmented dcg analysis for next-generation search engine
2. Bos, J., Markert, K.: Recognising textual entailment with logical inference, pp. 628–635 (2005)
3. Baral, C., Dzifcak, J., Son, T.C.: Using answer set programming and lambda calculus to characterize natural language sentences with normatives and exceptions, 818–823 (2008)
4. Moldovan, D.I., Harabagiu, S.M., Girju, R., Morarescu, P., Lacatusu, V.F., Novischi, A., Badulescu, A., Bolohan, O.: Lcc tools for question answering, TREC

Stochastic Reasoning with Models of Agent Behavior

Gerardo I. Simari

University of Maryland College Park, College Park, MD 20742, USA

1. Problem Description and Motivation. There is a constant need to reason about diverse cultures all over the world. For example, a health care organization anxious about the spread of diarrhea (or any other disease) in Kenya might wish to understand socio-economic-cultural-environmental aspects of Kenyan society that cause the diseases to spread extensively in some parts of the country and not in others. In most cases, the spread of diseases is not due to biological factors alone, but to social behaviors, environmental factors, and economic and educational aspects of the disease-stricken community. The problem we are tackling is that of how to best *reason about actions using models of the behaviors of such agents* (where "agent" means any entity).

2. Progress Made to Date.
(a) Initial research on Action Probabilistic Logic Programs (APLPs): In [7] we presented APLPs, the basic language for expressing behavioral models based on rules that express that certain actions will be taken by an agent with a given probability interval depending on the current state of the world. We proposed a linear programming based approach for expressing the constraints on sets of action atoms that are expressed by an APLP, as well as the statement of the fundamental problem to be solved in order to be able to make predictions using this framework, *i.e.*, computing the most probable set of actions (or worlds) given a program and a current state (the MPW problem). ***(b) Research on efficiently and accurately finding an MPW in APLPs:*** In [2] we focused on effectively solving the MPW problem; we investigated ways of reducing the number of variables in the set of linear constraints, introducing two provably correct algorithms for finding an *equivalence class* of worlds that are solutions to the problem (classes subsume all worlds that are in them). Furthermore, we investigated the application of a heuristic for further reducing the number of variables in the computation, this time surrendering accuracy in favor of greater scalability. The heuristic algorithm was shown to be both scalable and reasonably accurate by a set of preliminary empirical evaluations. Finally, we have developed algorithms for MPW that leverage the fact that users are not always interested in discerning the *entire set* of possible actions [6]. Both exact and approximation algorithms are provided, as well as empirical evaluations of how they perform. ***(c) Evaluating Behavior of Agents with Respect to Promises Made:*** In [5], we develop a formal theory to quantitatively evaluate how well an agent has fulfilled its past promises and use that as a predictor of whether it will keep its current (as yet unfulfilled promises), taking into account three important factors not considered before: partial fulfillment, late fulfillment, and fulfillment of a promise that is "similar" to the promise that was made. The goal of this line of work is to integrate evaluation of promises in the type of rules like the ones discussed above. ***(d) Applications:*** Our work has found application in real-world behavior modeling efforts. In [8], we describe how the APLP framework can be applied to model the behaviors of various stake-holders in the Afghan drug economy, deriving probabilistic rules

much in the same way as an expert in any domain could do. In [9], we describe how our work fits in the "big picture" of modeling agent behavior within the CARA architecture.

3. Proposed Plan of Research. Research towards obtaining a PhD is expected to be complete by the end of the 2009-2010 academic year. The work described in the *Progress* section will be integrated into a thesis describing tractable methods for building models of agent behavior, where agents can be any entity whose behavior is of interest (real-world individuals and groups, as well as software agents).

4. Related Work. Past work on adversarial reasoning in AI has focused primarily on games such as Chess, Bridge, Clue, and Poker, where the rules of the game are well articulated. Reasoning about real world adversaries can be viewed as a complex game-tree search problem, but it is difficult to know the "rules" of the game and, often, even the variables constituting the state are unclear, let alone their values. Past adversarial reasoning work in AI can be a great asset even in real-world cultural reasoning, but a major problem is to identify the adversary's objectives and payoffs as well as the rules that the adversary adheres to, and determine how best to "play" the adversary given our knowledge of his behavioral rules. Probabilistic logic programming, which is the basis of our rule-based models, was introduced in [4] and later studied by many authors such as [3]. All works to date on this topic have addressed a problem different from MPW: checking whether a given annotated formula is entailed by a PLP, which usually boils down to finding out if all interpretations that satisfy the PLP assign a probability in accordance with the annotation. Finally, the closest work to solving large linear programs is that of [1], who use column generation methods to solve probabilistic satisfiability (PSAT); however, PSAT is easier computationally than MPW, since MPW computations require solving the linear program *once for each world*.

References

1. Jaumard, B., Hansen, P., de Aragão, M.P.: Column generation methods for probabilistic logic. In: Proc. of IPCOC 1990, pp. 313–331. Univ. of Waterloo Press (1990)
2. Khuller, S., Martinez, M.V., Nau, D., Simari, G.I., Sliva, A., Subrahmanian, V.S.: Computing most probable worlds of action probabilistic logic programs: scalable estimation for $10^{30,000}$ worlds. AMAI 51(2), 295–331 (2007)
3. Lukasiewicz, T.: Probabilistic logic programming. In: Proc. of ECAI 1998, pp. 388–392 (1998)
4. Ng, R.T., Subrahmanian, V.S.: Probabilistic logic programming. Information and Computation 101(2), 150–201 (1992)
5. Simari, G.I., Broecheler, M., Subrahmanian, V.S., Kraus, S.: Promises made, promises broken: An axiomatic and quantitative treatment of fulfillment. In: Proc. of KR 2008, pp. 59–68 (2008)
6. Simari, G. I., Martinez, V., Sliva, A., and Subrahmanian, V. S. Focused Most Probable World Computations in Probabilistic Logic Programs. Under review (journal)
7. Simari, G.I., Sliva, A., Nau, D., Subrahmanian, V.S.: A stochastic language for modelling opponent agents. In: Proc. of AAMAS 2006, pp. 244–246. ACM, New York (2006)
8. Sliva, A., Martinez, V., Simari, G.I., Subrahmanian, V.S.: Soma models of the behaviors of stakeholders in the afghan drug economy: A preliminary report. In: Proc. of ICCCD 2007, pp. 78–86. AAAI Press, Menlo Park (2007)
9. Subrahmanian, V.S., Albanese, M., Martinez, V., Reforgiato, D., Simari, G.I., Sliva, A., Udrea, O., Wilkenfeld, J.: CARA: A Cultural Reasoning Architecture. IEEE Intelligent Systems 22(2), 12–16 (2007)

Research Summary

Analysing Graph Transformation Systems Using Extended Methods from Constraint Handling Rules

Frank Raiser

Faculty of Engineering and Computer Sciences, Ulm University, Germany
frank.raiser@uni-ulm.de

1 Motivation

Constraint Handling Rules (CHR) [1] has become a general-purpose rule-based programming language throughout the last decade. The relations to many other formalisms have been investigated [2] and often results could be transferred from CHR to other formalisms, or vice versa.

Graph Transformation Systems (GTS) [3], which have been developed in the 60ies and have become increasingly popular, have not been compared to CHR before. GTS and CHR appear to be very similar on a cursory glance, as they are both non-deterministic rule-based state transition systems. However, the fact that confluence is decidable in CHR [1] and undecidable in GTS [4] warrants a closer investigation of the two formalisms. Hence, I want to apply analysis methods of CHR to GTS, concentrating on confluence analysis.

2 Existing Work

A solid mathematical basis for algebraic graph transformation systems is given in [3] that is based on category theory. CHR has been compared with several other formalisms [2], but to the best of my knowledge, there either was no comparison of confluence, or confluence of terminating systems was decidable in both. A direct comparison of GTS with CHR has not appeared in the literature before.

3 Goals

Through this thesis I want to gain further insights into the relation between these two important rule-based formalisms. I want to embed GTS in CHR and compare the approaches to deciding confluence. I also want to find out which elements of CHR are responsible for the decidability of confluence. Furthermore, I want to investigate similar analysis methods, like operational equivalence, that have been developed for CHR and transfer them to GTS.

4 Preliminary Results

I have succeeded in embedding GTS in CHR [5]. I also managed to give a characterization of the sufficient criterion for confluence of GTS in CHR using the notion of observable confluence [6]. The embedding also proved viable to transfer the notion of operational equivalence to GTS [7].

Confluence analysis requires to test CHR states for equivalence. The comparison of GTS confluence with that in CHR, together with results on the linear logic semantics of CHR gave further insights into state equivalence in CHR. Recently, I have succeeded in providing an axiomatic definition for it [8] with two significant results for CHR research: firstly, the correspondence between state equivalence and rule application, that has been taken for granted for over a decade, could be proved for the first time. And secondly, this work provides the foundation for a new view on CHR's operational semantics as a state transition system over equivalence states. Thus, the investigation of state equivalence in the context of confluence gave rise to results relevant to the operational semantics.

5 Open Issues and Expected Achievements

An open issue is to extend the notion of operational equivalence to get a stronger characterization of equivalence of terminating GTS, similar to my confluence characterization. Furthermore, using my observations on confluence I intend to determine if a subclass of GTS exists that more closely resembles CHR such that confluence in this subclass becomes decidable.

References

1. Frühwirth, T.: Constraint Handling Rules. Cambridge University Press, Cambridge (to appear, 2009)
2. Sneyers, J., Van Weert, P., Schrijvers, T., De Koninck, L.: As time goes by: Constraint Handling Rules – A survey of CHR research between 1998 and 2007. Accepted by Journal of Theory and Practice of Logic Programming (2008)
3. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Springer, Heidelberg (2006)
4. Plump, D.: Confluence of graph transformation revisited. In: Middeldorp, A., van Oostrom, V., van Raamsdonk, F., de Vrijer, R. (eds.) Processes, Terms and Cycles: Steps on the Road to Infinity. LNCS, vol. 3838, pp. 280–308. Springer, Heidelberg (2005)
5. Raiser, F.: Graph Transformation Systems in CHR. In: Dahl, V., Niemelä, I. (eds.) ICLP 2007. LNCS, vol. 4670, pp. 240–254. Springer, Heidelberg (2007)
6. Raiser, F., Frühwirth, T.: Strong joinability analysis for graph transformation systems in CHR. In: 5th International Workshop on Computing with Terms and Graphs, TERMGRAPH 2009 (2009)
7. Raiser, F., Frühwirth, T.: Operational equivalence of graph transformation systems. In: Sixth International Workshop on Constraint Handling Rules (submitted, 2009)
8. Raiser, F., Betz, H., Frühwirth, T.: Equivalence of CHR states revisited. In: Sixth International Workshop on Constraint Handling Rules (submitted, 2009)

Modular Action Language $\mathcal{ALM}$

Daniela Inclezan

Texas Tech University, Computer Science Department
Lubbock, TX 79409 USA
daniela.inclezan@ttu.edu

1. Introduction and Problem Description

The field of knowledge representation experienced substantial progress in representing dynamic domains during the last decades. Action languages were created to describe transition diagrams in a mathematically accurate and concise manner. Solutions for the frame, ramification and qualification problems were discovered. We now have a good understanding of how to describe effects of actions and action executability conditions. However, the issue of describing objects of dynamic domains, including actions and fluents, remains almost unaddressed. The traditional approach, in which such objects are represented as constants or terms, doesn't allow for an elaboration tolerant and scalable represention of action properties. In previous action languages it is impossible to describe objects of the domain in terms of other, already defined, objects. This and similar features can be achieved by adding modularity to action languages. This would enable programmers to create libraries of knowledge about dynamic domains.

2. Background and Overview of Existing Literature

In the last 5 years, several attempts have been made towards the introduction of modularity in logic programming. Lifschitz and Ren [1] used the concept of renaming to add modularity to $\mathcal{C}+$, an action language based on the *causality principle* stating that *"Everything true in the world must be caused"*. Gustaffson and Kvarnström [2] used an object-oriented framework to describe dynamic domains. Gelfond [3] added modularity to logic programming language CR-Prolog. Other ideas for applying modularity to logic programming were the introduction of templates [4] or macros [5].

3. Goal of the Research

Our goal is to develop methodology and tools for representing a large body of knowledge about dynamic domains. In particular, we intend to create a simple yet powerful modular action language, $\mathcal{ALM}$, and use it to create libraries of knowledge about a number of recurrent dynamic domains. We will test the generality of our approach by using our $\mathcal{ALM}$ libraries to represent several well-known problems and novel scenarios, and perform reasoning using the obtained representations. $\mathcal{ALM}$ will be based on an extension $\mathcal{AL}_d$ that will add *defined fluents* to action language $\mathcal{AL}$. $\mathcal{AL}$ incorporates the *inertia* axiom stating that *"Things tend to stay as they are"*, and is closely connected to ASP. Our methodology will use $\mathcal{AL}$'s connection to ASP for reasoning, planning, and diagnosis.

4. Current Status of the Research

So far, we have a good understanding of the conceptual design of the language. The syntax of $\mathcal{ALM}$ is intuitively defined via a number of examples. We define reusable *modules* of knowledge about dynamic domains as collections of classes describing sorts, fluents and actions of the domain. A class of objects describes properties of those objects and axioms about them. Modules can be refined by adding new classes or refining existing classes. The definition of a *dynamic system* consists of class declarations imported from modules, the system's sorted universe, and its collection of concrete actions. The semantics of a system description in $\mathcal{ALM}$ is given by mapping it into a system description in $\mathcal{AL}_d$.

5. Preliminary Results Accomplished

Our first efforts focused on illustrating $\mathcal{ALM}$ by representing primitive domains. For example, we created module *Basic Move* to represent the movement of things from one area to another. We then showed how this module could be refined by adding new sorts (*cities*, *countries*). Another refinement allowed us to represent movements in both the vertical and horizontal coordinate systems. We also showed how action class *carry* can be defined as a special case of class *move.* These basic modules were used to represent various simple scenarios, including *Blocks World.*

6. Open Issues and Expected Achievements

The syntax and semantics of $\mathcal{ALM}$ remain to be finalized. More dynamic domains should be represented in our language. The representation of well-known problems in $\mathcal{ALM}$ should be compared to their representation in other modular action languages, to check that $\mathcal{ALM}$ is indeed a simpler and more powerful language. The generality and elaboration tolerance of our language should be tested by verifying the applicability of our libraries to novel problems. An implementation of $\mathcal{ALM}$ should be provided as well. Finally, we will consider extending our language by adding other important abstractions such as *activities* or *intentions.*

Acknowledgments. The author would like to thank Prof. Michael Gelfond for all his help and support.

References

1. Lifschitz, V., Ren, W.: A Modular Action Description Language. In: Proceedings of AAAI 2006, pp. 853–859 (2006)
2. Gustafsson, J., Kvarnström, J.: Elaboration Tolerance through Object-Orientation. Artificial Intelligence 153(1-2), 239–285 (2004)
3. Gelfond, M.: Going Places – Notes on a Modular Development of Knowledge about Travel. In: Working Notes of the AAAI 2006 Spring Symposium (2006)
4. Calimeri, F., Ianni, G., Ielpa, G., Pietramala, A., Santoro, M.C.: A System with Template Answer Set Programs. In: Proceedings of JELIA 2004 (2004)
5. Baral, C., Dzifcak, J., Takahashi, H.: Macros, macro calls and use of ensembles in modular answer set programming. In: Etalle, S., Truszczyński, M. (eds.) ICLP 2006. LNCS, vol. 4079, pp. 376–390. Springer, Heidelberg (2006)

Answering Questions from Natural Language Using A-Prolog

Yana Todorova

Texas Tech University, Lubbock, TX 79409 USA
yana.todorova@ttu.edu

Introduction and Problem Description. The topic of this research is the development of methodology for building computer systems capable of answering questions from natural language (NL) texts. Existing methodologies take the NL text and question as an input, transform it to a logical form, and use reasoning systems to obtain the answer. Sometimes little or no commonsense knowledge is added to the original input. However, we believe that most answers are commonsense answers and we want to investigate how additional knowledge can be used to produce them.

Background and Overview of the Existing Literature. Several question answering systems (QAS) have been proposed in the past. Some of them use rigorous approaches, but some are satisfied with a sloppier and more procedural methods of knowledge representation and reasoning. We want a rigorous approach that uses commonsense and other background knowledge and one that is based on nonmonotonic reasoning techniques. The following are examples of the state of the art of QAS. LCC QA system [1] is hybrid, not always well defined, but powerful. Nutcracker [2] uses *first-order logic* (FOL) reasoning tools and expresses different NL phenomena. However, since FOL is monotonic formalism, it does not allow commonsense default reasoning. Mueller's system [3] uses *Event Calculus* and scripts. DD system and ASU QA system [4] perform commonsense reasoning. Their language of choice is *A-Prolog*. This is a language for knowledge representation, nonmonotonic reasoning, and declarative problem solving [5]. It is suitable for representing knowledge and for reasoning in so called dynamic domains.

Goal of the Research. Our main goal is to develop a methodology for building a reliable QA system with commonsense knowledge and test it on simple domains expanding those of the DD and the ASU QA systems. Both of them are only applicable to very limited linguistic and knowledge domain. Two tasks are required: (1) build logic forms and (2) build knowledge bases for commonsense domains and test them. First, we select a non-trivial motion domain, which expands the previous work by adding a difficult task of reasoning about cardinality. More domains will be considered later. Second, we test if natural language processing can be enhanced by the use of A-Prolog reasoning methods.

Current Status of the Research. To get from English text to its logic form, people normally use existing Natural Language Processing (NLP) systems. However, such systems produce incorrect analysis. Therefore, we decided to develop a simple controlled natural language, A-CL. It has a restricted grammar and a limited vocabulary for expressing motion scenarios. Next, we translate the A-CL texts and questions into our input language. We use the tool Boxer [6], which parses our A-CL sentences and generates semantic representations of them. The vocabulary of A-CL includes Verbs of Motion [7]. We further classify them into two types of motion verbs: *enter verbs* and *leave verbs*. We combine those verbs with prepositions, such as *into*, *to*, and *from*. Thus, we allow phrasal verbs, such as *come in* and *come out*.

Preliminary Results Accomplished. We started the development of a QA system, called A-QAS. It uses knowledge represented in A-Prolog to answer questions from natural language formulated in A-CL. Our preliminary results are the development of language A-CL; some improvement of Boxer's logic form; the translation from improved Boxer's logic form to A-Prolog input; and some axiomatization of reasoning about motion.

Open Issues and Expected Achievements. Our next steps are, first, to expand the vocabulary and the grammar rules of A-CL and to see what linguistic phenomena appear as a result. Second, to translate larger input texts and questions into A-Prolog and to observe if anaphora resolution can be handled the same way as with smaller inputs. There are two open issues that remain. First, can we leverage existing NLP systems by adding A-Prolog knowledge to produce high quality logic forms for our language and its extensions? Second, can we formalize relevant commonsense knowledge in A-Prolog in a reusable and elaboration tolerant way? This work is the first step towards answering these questions. Finally, our expected achievements are to develop a QA system with high reliability and to extend our controlled language with other related domains.

Acknowledgments. The author would like to thank Michael Gelfond for his help.

References

1. Clark, C., Harabagiu, S., Maiorano, S., Moldovan, D.: COGEX: A Logic Prover for Question Answering. In: Proc. of HLT-NAACL, pp. 87–93 (2003)
2. Bos, J., Markert, K.: Recognising textual entailment with logical inference. In: Proceeding of the Conference on Empirical Methods in Natural Language Processing (EMNLP), pp. 628–635 (2005)
3. Mueller, E.: Understanding script-based stories using commonsense reasoning. Cognitive Systems Research 5(4), 307–340 (2004)

4. Balduccini, M., Baral, C., Lierler, J.: Knowledge Representation and Question Answering. In: Handbook of Knowledge Representation, ch. 1, Elsevier, Amsterdam (2007)
5. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: Logic Programming: Proc. of the Fifth Int'l Conf. and Symp., pp. 1070–1080 (1988)
6. Curran, J., Clark, S., Bos, J.: Linguistically Motivated Large-Scale NLP with C&C and Boxer. In: Proceedings of the ACL 2007 Demonstrations Session (ACL 2007 demo), pp. 29–32 (2007)
7. Levin, B.: English Verb Classes and Alternations: A Preliminary Investigation. University of Chicago Press (1993)

Belief Logic Programming

Hui Wan

State University of New York at Stony Brook,
Stony Brook, NY 11794, U.S.A.

Various forms of quantitative logic programming have been widely used for dealing with uncertainty and inconsistency in knowledge representation. A less explored issue in quantitative logic programming is combining correlated pieces of information. Most works disregard correlation or assume that all sources are independent. Others make an effort to take some forms of correlation into account, but in an ad hoc manner.

Probability is widely used to model uncertainty, but some problems arise when several information sources disagree. For example, one source might claim that the probability of a bullish stock market is in the interval $[0.5, 0.6]$, while the other might suggest that it is in the interval $[0.2, 0.3]$. What should one conclude? Some approaches [3] compute the intersection, yielding $\varnothing$. Others [10] combine the uncertainty ranges, e.g., $[min(0.5, 0.2), max(0.6, 0.3)]$. However, such combination methods cannot be explained probabilistically. Another well-established way of dealing with uncertainty is Fuzzy Logic [14]. It has been successfully applied in many domains, but it remains controversial due to some of its properties [5,6].

Dempster-Shafer theory of evidence [4,12] has also been central to many approaches to quantitative logic programming [1,2,8,11,13]. This theory is based on belief functions [12] which represents degrees of belief in various statements. The belief functions from different sources must be combined in order to obtain more accurate information. The difficulty is that the contributing sources are sometimes not independent, so correlation of evidence is necessary.

To the best of our knowledge, almost all existing works avoid correlating structural information contained in rules, which often leads to counter-intuitive behavior when such correlation is essential for correctness. A few notable exceptions include Baldwin's [1,2], Lakshmanan's [9] and Kersting's [7] approaches. Kersting et al. provide a very general, albeit impractical, framework, which could, in principle, be used to handle correlation. However, combination of two inconsistent conclusions is hard to explain in probability theory. Both Baldwin's and Lakshmanan's methods assume that every pair of rules with the same head have the same correlation. Consequently their methods are inadequate when the correlation between two sources changes from case to case. Most other approaches simply restrict logic dependencies to avoid combining sources that are not independent [8,13]. Those that do not, might yield incorrect or inaccurate results when combining sources that are correlated due to overlapping belief derivation paths. For example, consider two rules A `:-` $B \wedge C$ and A `:-` $B \wedge D$, each asserting its conclusion with certainty 0.5. The approach in [10] would directly combine the certainty factors for A derived from the two rules as if they are independent, assigning A a combined certainty, which is likely going to be too high. Clearly, the independence assumption does not hold here, as both rules rely on the same fact B.

The goal of my dissertation research is to enable correlation of evidence in combining multiple information sources. We have proposed a novel form of quantitative reasoning, called *Belief Logic Programming* (BLP), which is able to account for correlation of evidence obtained from non-independent and possibly contradictory information sources. The semantics of BLP is based on belief functions and Dempster-Shafer theory [4,12], but it is not tied to any particular rule for combining evidence: any natural rule for combining evidence can be used. Along with the semantics, we developed a query evaluation algorithm, which utilizes information supplied in the query to speed up computation.

The plan for future work is to extend BLP to programs with cyclic logic dependency among atoms. Another possible direction is to extend the algorithms to deal with non-ground rules and queries, and try to make them optimized based on belief factors given in the query.

References

1. Baldwin, J.F.: Support logic programming. Intl. Journal of Intelligent Systems 1, 73–104 (1986)
2. Baldwin, J.F.: Evidential support logic programming. Fuzzy Sets and Systems 24(1), 1–26 (1987)
3. Dekhtyar, A., Subrahmanian, V.S.: Hybrid probabilistic programs. J. of Logic Programming 43, 391–405 (1997)
4. Dempster, A.P.: Upper and lower probabilities induced by a multi-valued mapping. Ann. Mathematical Statistics 38 (1967)
5. Elkan, C.: The paradoxical success of fuzzy logic. In: IEEE Expert, pp. 698–703 (1993)
6. Halpern, J.Y.: Reasoning About Uncertainty. MIT Press, Cambridge (2003)
7. Kersting, K., De Raedt, L.: Bayesian logic programs. Technical report, Albert-Ludwigs University at Freiburg (2001)
8. Kifer, M., Li, A.: On the semantics of rule-based expert systems with uncertainty. In: Gyssens, M., Van Gucht, D., Paredaens, J. (eds.) ICDT 1988. LNCS, vol. 326, pp. 102–117. Springer, Heidelberg (1988)
9. Lakshmanan, L.V.S., Shiri, N.: A parametric approach to deductive databases with uncertainty. IEEE Trans. on Knowledge and Data Engineering 13(4), 554–570 (2001)
10. Ng, R.T.: Reasoning with uncertainty in deductive databases and logic programs. Intl. Journal of Uncertainty, Fuzziness and Knowledge-Based Systems 5(3), 261–316 (1997)
11. Ruthven, I., Lalmas, M.: Using dempster-shafers theory of evidence to combine aspects of information use. J. of Intelligent Systems 19, 267–301 (2002)
12. Shafer, G.: A Mathematical Theory of Evidence. Princeton University Press, Princeton (1976)
13. Shenoy, P.P., Shafer, G.: Axioms for probability and belief-function propagation. In: Uncertainty in Artificial Intelligence, pp. 169–198. North-Holland, Amsterdam (1990)
14. Zadeh, L.A.: Fuzzy sets. Information Control 8, 338–353 (1965)

Logic-Statistic Models with Constraints for Biological Sequence Analysis

Christian Theil Have

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
cth@ruc.dk

1 Introduction and Problem Description

This project aims to investigate biologically inspired, logic-statistic models with constraints. The complexity and expressiveness of models with different kinds of constraints will be examined and algorithms to efficiently cope with inference in and training of such models will be explored. The models will be evaluated with regards to their applicability to biological sequence analysis.

Statistical models for biological sequence analysis are usually based on variants of HMMs and occasionally more expressive models like PCFGs. The size of the data often prohibits the more expressive models, so careful choice of good independence assumptions is paramount. Realistic biological models may include complicated interactions between different aspects such as codon frequencies, RNA structure and phylogenetic information. Constraints can be a way of explicitly combining aspects of such models in a more intuitive and modular way and at a higher abstraction level. The declarative nature of constraints permits a degree of freedom of implementation, allowing for potential optimizations.

Constraints are usually embedded in the model either as structure, parameters (soft, data-driven) or a combination, but can also be applied in inferencing. Consider as example a generic genefinder model, where we would like to infer the most likely sequence of *hidden* states, representing the genes and non-genes, that best explains a given *observable* sequence of nucleotides. It might be that we know that the DNA sequence for a particular organism contains at least 8000 genes, but the proposed most likely sequence has less than 8000 genes. In this case, the constraint could be used to guide the inference procedure. It has recently been suggested that this can be formulated as constraint satisfaction problem [1].

2 Background and Overview of the Existing Literature

This project is part the larger project LOgic-STatistic Modeling and Analysis of Biological Sequence Data (LoSt). Logic-statistic models can be expressed as stochastic logic programs using the PRISM language [2], which is an extension of Prolog where values of special variables are determined by random switches rather than usual unification. PRISM includes efficient procedures for inference and parameter estimation. Stochastic logic programs can have constraints, usually in the form of equality between unified logic variables. Stochastic selection of values for such variables may lead to unification failure and resulting failed derivations must be taken into account in parameter estimation. PRISM does this using an adaptation of Cussen's FAM algorithm [3].

3 Goal of the Research

The goal is to investigate the applicability of logic-statistic models with constraints with regard to their ability to express and efficiently deal with the problems of biological sequence analysis.

4 Current Status of the Research

The research is at a very early stage where the ideas are currently being refined.

5 Preliminary Results Accomplished

A context-sensitive grammar formalism, "Stochastic Definite Clause Grammars", was implemented using PRISM and utilizing its facilities for handling failures.

6 Open Issues and Expected Achievements

I hope to find that logic-statistic models with constraints will make it easier to express complex biological models and that the achieved compositional problem structure will allow certain optimizations. I think that statistical inference and constraint solving can complement each other and that interesting techniques may be found in their intersection. Different logic-statistic frameworks may have distinct features with regard to different kinds of constraints and these features should be investigated further.

Soft constraints seems to be a nice fit to statistical models since probabilities are much like preferences or weights. In the example in section 1 we might be only 80% certain that the constraint holds. If the probability of tagging at least 8000 genes is sufficiently low, then it might be preferable to break the constraint. Using soft constraints in the context of both inference and parameter learning seems like a very interesting direction to pursue.

References

1. Petit, M., Christiansen, H.: Viterbi computation for a constrained hidden markov model. Actes JFPC (2009)
2. Sato, T., Kameya, Y.: New advances in logic-based probabilistic modeling by prism. In: De Raedt, L., Frasconi, P., Kersting, K., Muggleton, S. (eds.) Probabilistic Inductive Logic Programming. LNCS, vol. 4911, pp. 118–155. Springer, Heidelberg (2008)
3. Cussens, J.: Parameter estimation in stochastic logic programs. Machine Learning 44(3), 245 (2001)

Fusion of Logic Programming and Description Logics

Mantas Šimkus

Institute of Information Systems, Vienna University of Technology
Favoritenstaße 9-11, Vienna, Austria
simkus@kr.tuwien.ac.at

Overview. Broadly, our research concerns the fusion of Logic Programming (LP) and Description Logics (DLs), two solid knowledge representation paradigms with largely dual properties. The two feature different semantics and orthogonal expressivity, and they apply different reasoning techniques. Finding ways to exploit the positive features of both has become an important research topic, which has also been fueled by the prospect of applications in the Semantic Web [10]. In the latter context, ontologies written in expressive DLs are to be used to provide a common conceptualization of web resources, which can then be accessed by LP-based automated agents providing services to end-users.

We are concentrating on two ways of integrating LP and DLs. The first one is through the identification of expressive but still decidable fragments of logic programs with features that allow to capture some important DLs. In other words, we aim at obtaining a common host language for both the rules and expressive DLs. The second way concerns knowledge representation languages where the two paradigms are loosely integrated. In such formalisms, knowledge bases consist of two separate but interacting components: a DL ontology and a logic program, see, e.g., [8,5,11,4]. Integration in both cases is highly nontrivial. Naive approaches quickly lead to undecidability, and for the second approach, it is not always apparent how to give an intuitive semantics that would integrate the closed- and open-world assumptions of the two paradigms.

Contributions. The focus until now has been mostly on the first approach, leading to the identification of $\mathbb{FDNC}$ and *bidirectional* logic programs [7,6]. Both are decidable families of logic programs allowing for function symbols, disjunction, and negation under the answer set semantics.

Function symbols play a crucial role in these languages. They allow to simulate existential quantification, a fundamental feature of expressive DLs, enabling us, for example, to capture $\mathcal{SHI}$, the DL closely related to the Semantic Web ontology language OWL-lite. The availability of function symbols is also helpful when modeling indefinite time, reasoning about actions and change, and supports a generic representation of recursive data structures (see [1,2] for related approaches). However, allowing function symbols in answer set programming poses major technical challenges, as their unrestricted presence causes high undecidability. Overcoming this is not trivial, and requires carefully crafted restrictions and novel reasoning methods that had not been explored in this setting.

To show decidability and complexity results for the introduced languages we have used the *knot* and *automata* techniques. The former approach was introduced in [7] and is closely related to the *mosaic* technique previously applied in Modal Logics, while the latter is well known in the context of Modal and Dynamic Logics. Both approaches are novel in the context of logic programs, and transferring them to this field is not straightforward.

In line with the second approach to combining LP and DLs, we have contributed with knot-based algorithms and complexity results for *conjunctive query answering* over DL knowledge bases [9,3]. While query answering is a topic of interest on its own in DLs, it is also of major importance for integrating DLs and LP because many important approaches make use of conjunctive queries in their semantics [8,11,4]. Hence, new complexity results and reasoning methods for query answering provide new insights for the existing integration approaches.

Future work. There are several issues for the future work. Indeed, the preliminary investigations of $\mathbb{FDNC}$ and bidirectional programs concentrated on decidability and complexity issues. However, a thorough investigation of the knowledge representation aspects of the two languages still remains to be done. This involves exploring their expressive power, finding new encoding of DLs, identifying new applications, and comparing them with existing languages. We will also try to extend these languages, and to find expressive fragments exhibiting lower complexity. Finally, we intend to consider translations from the two languages to function-free logic programs for which efficient reasoners exist.

References

1. Bonatti, P.: Reasoning with infinite stable models. Artif. Intell. 156(1), 75–111 (2004)
2. Calimeri, F., Cozza, S., Ianni, G., Leone, N.: Computable functions in ASP: Theory and implementation. In: Garcia de la Banda, M., Pontelli, E. (eds.) ICLP 2008. LNCS, vol. 5366, pp. 407–424. Springer, Heidelberg (2008)
3. Eiter, T., Gottlob, G., Ortiz, M., Šimkus, M.: Query answering in the description logic Horn-$\mathcal{SHIQ}$. In: Hölldobler, S., Lutz, C., Wansing, H. (eds.) JELIA 2008. LNCS, vol. 5293, pp. 166–179. Springer, Heidelberg (2008)
4. Eiter, T., Ianni, G., Krennwallner, T., Schindlauer, R.: Exploiting conjunctive queries in description logic programs. In: Annals of Mathematics and Artificial Intelligence (2009) (Published online: 27 January 2009)
5. Eiter, T., Lukasiewicz, T., Schindlauer, R., Tompits, H.: Combining answer set programming with description logics for the semantic web. In: KR 2004, pp. 141–151 (2004)
6. Eiter, T., Šimkus, M.: Bidirectional answer set programs with function symbols. In: Proc. of IJCAI 2009 (to appear, 2009)
7. Eiter, T., Šimkus, M.: FDNC: Decidable non-monotonic disjunctive logic programs with function symbols. ACM Trans. Comput. Log. (to appear, 2009)
8. Levy, A.Y., Rousset, M.-C.: Combining Horn rules and description logics in CARIN. Artificial Intelligence 104(1–2), 165–209 (1998)
9. Ortiz, M., Šimkus, M., Eiter, T.: Worst-case optimal conjunctive query answering for an expressive description logic without inverses. In: Proc. of AAAI 2008, pp. 504–510 (2008)
10. Patel-Schneider, P., Hayes, P., Horrocks, I.: OWL Web Ontology Language semantics and abstract syntax. W3C Recommendation (February 2004), http://www.w3.org/TR/owl-semantics/
11. Rosati, R.: DL+log: Tight integration of description logics and disjunctive datalog. In: Proc. of KR 2006, pp. 68–98 (2006)

Research Summary: Non-termination Analysis of Logic Programs

Dean Voets*

Department of Computer Science, K.U. Leuven, Belgium

1 Introduction and Related Work

An important advantage of Logic Programming (LP) is that a declarative programming style leads to more understandable and less error-prone computations. However, a declarative programming style also leads to less efficient, and in the extreme case, in non-terminating computations. Therefore, an important aspect of proving correctness of a program, is the analysis of the termination behavior of the program for a class of queries.

Termination analysis is a well studied field in LP. [4] presents an extensive overview of the work up till 1994. However, many powerful techniques have been developed in the last decade (e.g. [2]).

Non-termination analysis is a much less studied problem. In [3], a non-termination analyzer, *NTI*, is introduced. The main motivation for this work, is checking the precision of the termination analysis techniques. As far as we know, this is the only automated technique for proving non-termination in LP.

Very recently, another technique to analyze the termination behavior was introduced in [5]. This approach does not produce a termination or non-termination proof, but predicts the termination behavior based on a symbolic derivation tree. Experiments show that these predictions are extremely precise.

2 Goal of the Research

The goal of my research is to develop a new non-termination analyzer, based on the symbolic derivation trees introduced for termination prediction. A first version of this non-termination analyzer is already developed and implemented by me. The analyzer proves non-termination of pure logic programs and classes of queries described using modes. I want to extend this technique in several ways, so that it is applicable to more realistic Prolog programs.

Because many programs have input arguments that also contain variables, using modes to describe classes of queries is too restricted. Therefore, I want to extend this technique so that types can be used to describe these queries.

Another issue is the restriction to pure logic programs. Although most termination and non-termination analysis techniques are restricted to such programs, an analyzer must handle some non-logical features to analyze more realistic programs. The most important extension of our technique, is an extension for programs containing arithmetics.

* Supported by the Fund for Scientific Research - FWO-project G0561-08.

In order to apply this analysis on larger Prolog programs, it is important to apply some kind of modular approach. Many modular approaches have been discussed in the literature (e.g. [2]). I want to adapt one of these techniques so that it fits our non-termination analyzer.

3 Current Status of the Research

In [6], I introduced a new non-termination condition for logic programs and classes of moded queries. Non-termination is proven by showing that a path in the symbolic derivation tree can be applied infinitely often. Such a path can be identified by checking three basic properties. This approach has been implemented in the analyzer $P2P$[1].

$P2P$ first constructs the symbolic derivation tree. Then, it checks if this tree contains a path that satisfies the three properties implying non-termiation.

I tested $P2P$ on a benchmark of 48 small to medium sized programs. These programs are the non-terminating pure logic programs from a benchmark[2] representing different challenges in both termination and non-termination analysis. Our tool proves non-termination of all benchmark programs and thus, significantly improves on the results of the only other non-termination analyzer NTI.

4 Open Issues and Expected Achievements

The open issues I want to solve are discussed in the second section.

The first priority is the extension for programs with arithmetics. The idea is to rewrite the arithmetic expressions in the path, so that existing finite domain solvers can be used to infer domains for which the path loops. Both the theory and the implementation of this extension are in a preliminary phase.

The extension for types could be done by applying ideas similar to the ones used in abstract interpretation [1].

References

1. Bruynooghe, M.: A practical framework for the abstract interpretation of logic programs. J. Log. Program. 10(1/2/3&4), 91–124 (1991)
2. Nguyen, M.T., Giesl, J., Schneider-Kamp, P., De Schreye, D.: Termination analysis of logic programs based on dependency graphs, pp. 8–22 (2008)
3. Payet, É., Mesnard, F.: Nontermination inference of logic programs. ACM Transactions on Programming Languages and Systems 28(2), 256–289 (2006)
4. De Schreye, D., Decorte, S.: Termination of logic programs: The neverending story. J. Log. Program. 19(20), 199–260 (1994)
5. Shen, Y.-D., De Schreye, D., Voets, D.: Termination prediction for general logic programs: Submitted. Report CW 536 (2009) (Accepted for TPLP)
6. Voets, D., De Schreye, D.: A new approach to non-termination analysis of logic programs. Report CW 536 (2009) (Accepted for ICLP 2009)

[1] Available at www.cs.kuleuven.be/˜dean/

[2] www.lri.fr/˜marche/termination-competition/

Study of Random Logic Programs

Gayathri Namasivayam

Department of Computer Science, University of Kentucky, Lexington, KY
40506-0046, USA

1 Introduction

The aim of this research is to generate random logic programs and study their properties such as the occurrence of an easy-hard-easy pattern and the probability of existence of stable models for these programs. The primary focus is on generating random logic programs that can be classified to be hard for ASP solvers based on the time taken and the number of choice points generated by them in finding a stable model. Our study and generation of random logic programs have been motivated by two key reasons: hard random logic programs can be used as benchmarks for evaluating the algorithms used in existing ASP solvers, and experimental analysis as well as theoretical studies of the properties of these random logic programs can provide us with insights for improving the design of the heuristics/algorithms in these solvers.

2 Background and Overview of the Existing Literature

The main inspiration to generate random logic programs comes from the extensive experimental and theoretical studies done in the area of random satisfiability (SAT). They include the study of the properties of random SAT instances such as phase transition, the easy-hard-easy pattern and the correlation of the hard region with the point at which the probability of generating a satisfiable random SAT theory is 0.5. These studies have led the path to significant advances in building efficient programs for solving SAT theories.

Random logic programs were initially studied by Yuting Zhao and Fangzhen Lin[3]. However, random logic programs were introduced earlier on by Wong, Schilpf and Truszczyński [1] as well as Yuting Zhao [2]. Fangzhen Lin and Yuting Zhao proposed two models for generating random logic programs based on if every rule in the program has the same fixed length (i.e., number of literals) or have varying mixed lengths. The fixed length programs were generated with the following three parameters: number of atoms N, rule density α that specifies that each program has α times N rules, and the length of each rule K. The probability distribution λ that specifies the probability of occurrence of a rule with a fixed number of literals in its body is a parameter used in the mixed length programs in addition to the other parameters N and α. Fangzhen Lin and Yuting Zhao generated fixed and mixed length random logic programs with an increasing number of rules and with all other parameters being equal.

The solvers demonstrate an easy-hard-easy pattern on these randomly generated logic programs by being able to solve logic programs that are generated with a certain α (i.e., say α_S) relatively slow when compared to those programs that are generated with $\alpha << \alpha_S$ or $\alpha >> \alpha_S$.

3 Current Status of the Research

Our work is related to that done by Yuting Zhao and Fangzhen Lin [3] on generating and studying random logic programs. However, we generate random logic programs using a different methodology and a different set of parameters.

We focus on generating random logic programs with rules of same length. We generate rules of lengths 2 and 3 since combinatorial problems in the class NP can be modeled as logic programs that primarily consists of rules having at most 2 literals each. We experimentally compute the probability of the existence of a stable model, the average time as well as the average number of choice points taken by ASP solvers, and the average number of stable models for logic programs that we randomly generate using the same parameters. An easy-hard-easy pattern is observed by randomly generating programs with an increasing number of rules and with all other parameters being the same. We generate tight random logic programs having rules of length 2 of the form $a \leftarrow not(b)$ where a and b are atoms, and produce a Clark's completion of these programs. Since, models of these completion theories are stable models of the corresponding tight logic programs, we use SAT solvers to find stable models of these logic programs. We observe a similar easy-hard-easy pattern for SAT solvers.

We also observe that random logic programs that are generated with certain parameters are harder for ASP solvers when compared to those generated with certain others parameters. We currently have identified properties of the random logic programs that are generated by us, and have been able to relate some of these properties to the occurrence of the easy-hard-easy pattern.

4 Open Issues and Expected Achievements

We would like to provide theoretical analysis to support the existence of the properties identified in the random logic programs generated by us. In the future we would like to be able design heuristics or modify existing heuristics to further aid the existing solvers in solving hard randomly generated logic programs.

References

1. Schilpf, J., Truszczyński, M., Wong, D.: On the distribution of programs with stable models, 05171 Abstracts Collection – Nonmonotonic Reasoning. Answer Set Programming and Constraints (2005)
2. Zhao, Y.: Random generated logic programs, 05171 Abstracts Collection – Nonmonotonic Reasoning, Answer Set Programming and Constraints (2005)
3. Zhao, Y., Lin, F.: Answer set programming phase transition: A study on randomly generated programs. In: Palamidessi, C. (ed.) ICLP 2003. LNCS, vol. 2916, pp. 239–253. Springer, Heidelberg (2003)

Locally Distributed Predicates: A Programming Facility for Distributed State Detection

Michael De Rosa

School of Computer Science, Carnegie Mellon University
mderosa@cs.cmu.edu

1 Introduction

The rise of high-performance computing and internet-scale applications has spurred a renewed interest in distributed computing. Such distributed applications can range from multi-hop routing algorithms used in wireless mesh networks [1] to volunteer-driven parallel data analysis efforts [2][3]. The property of distributed systems that makes them both powerful and challenging is that they are composed of multiple autonomous, interconnected entities. Managing the interactions between these entities is the primary challenge of distributed application development.

Distributed programming is inherently more difficult than single-threaded applications, due to two complicating factors. The first of these is the need for an executing thread of a distributed program located at one computational node to access state located at a different node. Accessing remote state information is quantitatively different than accessing local memory, in that latency and failure rates are much higher in the remote case. What makes accessing remote state qualitatively different is that remote state can be modified by another component of the distributed system, raising issues of consistent data state. This is further complicated by the second factor, asynchronicity. As multiple distributed components may operate independently, there is no guarantee that state obtained from multiple nodes represents a consistent view of the distributed process, as varying rates of computation and message latency can result in the reception of out-of-date information.

To overcome these difficulties, a set of general techniques for inspecting the state of a distributed system have been developed. The most widely used technique is that of a distributed snapshot [4], in which the state of the entire system is captured in a *consistent snapshot*, which reflects a possible serialization of the parallel event stream of the distributed nodes. Distributed snapshots are a useful tool, but they require a centralized aggregation point, and are typically quite heavy-weight. In many cases, the complete state of a distributed system is unnecessary; rather it is some property of the state that one wishes to test. Development of techniques for evaluating these tests produced global predicate evaluation [5], which allows a program to evaluate a single predicate over the entire distributed system.

While global predicates allow a programmer to encode queries over the state of an entire distributed system, in distributed systems with a sparse, multihop communications topology there exists another class of distributed predicates, which we call *locally distributed predicates*. These are predicates over the local neighborhood of a particular node, bounded to a finite number of communication hops. These locally distributed predicates allow a programmer to describe the state configuration of a bounded subgraph of a distributed system. Locally distributed predicates differ from global predicates in two important respects. An important difference is that, as locally distributed predicates do not encompass the entire distributed system, there may be multiple matching subgraphs for a particular predicate. Another difference is that a locally distributed property can describe not only the logical state of the entities in a distributed system, but also their topological configuration, a property that is inherently ignored by global predicates. This prevents global predicates from efficiently detecting predicates that rely on the state and topology of a small number of nodes. Such locally distributed predicates are important in a variety of applications, including distributed debugging, multi-entity coordination, and resource discovery.

I have developed an initial set of algorithms to represent and detect locally distributed predicates in the context of debugging large ensembles of modular robots [6]. These predicates allow programmers to describe the distributed state subsets that signify an error. I have developed a simple representation language that can express logical, topological, and temporal relationships between multiple communicating robots.

My thesis research centers on the development of a concise language for representing locally distributed predicates, and the use of this language in a variety of application domains. As part of this work, I plan to fully characterize both the formal semantics and performance properties of the language.

References

1. Karp, B., Kung, H.T.: Greedy Perimeter State Routing for Wireless Networks. In: Proceedings of the Sixth Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom 2000), Boston, Massachusetts, pp. 243–254 (2000)
2. Anderson, D.P., Cobb, J., Korpela, E., Lebofsky, M., Werthimer, D.: SETI@home: an experiment in public-resource computing. Communications of the ACM 45(11), 56–61 (2002)
3. Larson, S.M., Snow, C.D., Shirts, M.R., Pande, V.S.: Folding@Home and Genome@Home: Using distributed computing to tackle previously intractable problems in computational biology. In: Computational Genomics, Horizon Press (2002)
4. Chandy, K.M., Lamport, L.: Distributed snapshots: Determining global states in distributed systems. ACM Transactions on Computer Systems 3(1), 63–75 (1985)
5. Cooper, R., Marzullo, K.: Consistent Detection of Global Predicates. In: Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging, pp. 167–174 (1991)
6. De Rosa, M., Goldstein, S., Lee, P., Campbell, J., Pillai, P.: Distributed Watchpoints: Debugging Large Multi-Robot Systems. In: Proceedings of the IEEE International Conference on Robotics and Automation (2007)

Capturing Fair Computations on Concurrent Constraint Language

Paola Campli and Stefano Bistarelli*

Dipartimento di Scienze, Universitá "G. d'Annunzio" di Chieti-Pescara, Italy

Introduction and Problem Description. This paper contains an extension of the Concurrent Constraint language (CC) in order to guarantee a *fair* criterion of selection among parallel agents and to restrict or remove the possible unwanted behavior of a program.

Background and Overview of the Existing Literature. - Fairness in programming languages: the most common notions of fairness are given by Nissim Francez in [1]; **weak fairness** requires that if an agent is *continuously enabled* then it must *eventually* proceed, while **strong fairness** requires that if an agent can proceed *infinitely often* then it must *eventually* proceed (be executed).
- Concurrent Constraint Programming [3] is a programming paradigm that concerns the behavior of a set of concurrent agents with a shared store (σ), which is a conjunction of constraints (relations among a specified set of variables).The concurrent agents communicate with the store, by either checking if it *entails* a given constraint (*ask* operation) or adding a new constraint to it (*tell* operation). The behavior of the agents in a parallel execution is given by the following transition rule:

$$\frac{\langle A_1,\sigma\rangle\rightarrow\langle A_1',\sigma'\rangle}{\langle A_1\|A_2,\sigma\rangle\rightarrow\langle A_1'\|A_2,\sigma'\rangle}\quad\frac{\langle A_1,\sigma\rangle\rightarrow\langle A_1',\sigma'\rangle}{\langle A_2\|A_1,\sigma\rangle\rightarrow\langle A_2\|A_1',\sigma'\rangle}\quad\text{parallelism (1)}$$

$$\frac{\langle A_1,\sigma\rangle\rightarrow\langle success,\sigma'\rangle}{\langle A_1\|A_2,\sigma\rangle\rightarrow\langle A_2,\sigma'\rangle}\quad\frac{\langle A_1,\sigma\rangle\rightarrow\langle success,\sigma'\rangle}{\langle A_2\|A_1,\sigma\rangle\rightarrow\langle A_2,\sigma'\rangle}\quad\text{parallelism (2)}$$

Goal of the Research. The parallelism operator in Concurrent Constraint performs a selection among the concurrent agents. We aim to define a new transition rule in order to provide a *fair* criterion of selection among a finite number of parallel agents.

Current Status of the Research. We develop an extension of the CC language with fairness properties for finite computations by modifying the parallel operator $\|$ to use quantitative metrics that provides a more accurate way to establish which of the agents can succeed.The metric we use is based on a *Fair carpooling scheduling algorithm* [2]. Carpooling consists in sharing a car among a driver and one or more passengers to divide the costs of the trip in a fair way. Let U the total cost of the trip. We define m as the largest number of people who ever ride together at a time in the carpool and n as the number of participants in a

* Supervisor.

given day ($n \in [1 \ldots m]$). Each day we calculate the member's scores. In the first day each member has score zero; in the following days the driver will increase his score of $U(m-1)/m$, while the remaining $m-1$ passengers decrease their score of U/m. As proved by [2] this algorithm is fair, because at the end of the carpooling, each member meet the same cost.

Since we need to associate a value to more than two agents, we extend the semantic with the new operator $\|_m$ (where m is a finite number of agents. We include an array $k[]$ that permits to keep track of the actions performed by each agent. We represent the driver with the agent A_i, while the passengers are the remaining $(m-1)$ agents. We define $\alpha = U(m-1)/m$ and $\beta = U/m$. We add α to the previous value of $(k[i])$ of the agent A_i and we subtract β to the previous value $(k[j])$ of the other agents $A_j \quad \forall j \in [1, \ldots, m], \quad j \neq i$.

We also insert in the precondition of the rule a guard $(k_i \leq k_j)$ to establish which of the agents can succeed (that is, the one with a lower score). Notice that in $\|_m(A_1 \ldots A_m)$ we consider only the n enabled agents $(n \leq m)$.

In the initial phase, the enabled agent A_i succeed in $A_i^{'}$ with value α. The other agents instead assume the value $-\beta$. In next steps we sum the new values with that of the array $k[]$. We obtain the new transition rule:

$$\frac{k_i \leq k_j \quad \forall \quad j = 1, .., m, i \neq j \quad \langle A_i, \sigma \rangle \rightarrow \langle A_i^{'}, \sigma^{'} \rangle}{\langle \|_m \ (A_1, .., A_i, .., A_m), [k_1, .., k_i, .., k_m], \sigma \rangle \rightarrow \langle \|_m \ (A_1, .., A_i^{'}, .., A_m), [k_1 - \beta, ..k_i + \alpha, .., k_m - \beta], \sigma^{'} \rangle}$$

Since $\|_m$ is an associative and commutative operator, we can omit the rewriting of the second rule of Parallelism (1). We use the same criterion also for the Parallelism (2) rule:

$$\frac{k_i \leq k_j \quad \forall \quad j = 1, \ldots, m, i \neq j \qquad \langle A_i, \sigma \rangle \quad \rightarrow \quad \langle success, \sigma^{'} \rangle}{\langle \|_m \ (A_1 .. A_i .. A_m), [k_1, .., k_i, .., k_m], \sigma \rangle \rightarrow \langle \|_{m-1} \ (A_1, .., A_{i-1}, A_{i+1}, .., A_m), [k_1, .., k_{i-1}, k_{i+1}, .., k_m], \sigma^{'} \rangle}$$

The new rules respect strong fairness and weak fairness because the continuously enabled agents will eventually succeed.

Open Issues and Expected Achievements. The aim of our research is to guarantee equitable computations also in the current extensions of Concurrent Constraint. Moreover we plan to provide quantitative valuations by using soft constraints and to consider fairness for web services applications as a level of preference; we'll use a semiring structure to measure how much the service is fair. To do this we plan to use the same indexes that are used to measure the inequality in economics.

References

1. Francez, N.: FAIRNESS (text and monographs in computer science). Springer, Heidelberg (1986)
2. Fagin, R., Williams, J.H.: A Fair Carpool Scheduling Algorithm, International Business Machine Corporation, pp. 133–139 (1983)
3. Saraswat, V.A., Rinard, M.: Concurrent constraint programming. In: POPL 1990 (1990)

Constraint Based Languages for Biological Reactions

Marco Bottalico[*] and Stefano Bistarelli[**]

Dipartimento di Scienze, Università "G. d'Annunzio" di Chieti-Pescara, Italy
bottalic@sci.unich.it, bista@sci.unich.it

Introduction and Problem Description. In this paper, we describe two techniques based on Concurrent Constraint Programming, to model biological systems: timed process calculus (ntcc) and stochastic Concurrent Constraint Programming (sCCP). We build a simple biochemical reaction, from an enzymatic reaction and we write it be using the different models.

Background and Overview of the Existing Literature. Systems biology are a new science integrating experimental activity and mathematical modeling, which studies the dynamical behaviors of biological systems.

The most important features that we model are the basic time operators in the two languages: $tell_\lambda$ and ask_λ.

Biochemical reactions are chemical reactions involving mainly proteins. In a cell, there are many different proteins, hence, the number of reactions that can take place, can be very high. All the interactions that take place in a cell, can be used to create a diagram, obtaining a biochemical reaction network (BRN).

We will examine in the following, one of the thirteen enzymatic reaction of the blood coagulation, in the generic form:

$$XI + XIIa \underset{k_{-1}}{\overset{k_1}{\rightleftharpoons}} XI : XIIa \xrightharpoonup{k_2} XI + XIa \tag{1}$$

where XI is the enzyme (E) that binds substrate $(S) = XIIa$, to form an enzyme-substrate complex $(ES) = XI : XIIa$. After that we have the formation of product $(P) = XIa$ and the release of the unchanged enzyme $(E) = XI$, ready for a new reaction.

In literature there are any programs to model these features; we considered the followings:

– sCCP[2] is obtained by adding a stochastic duration to the instruction interacting with the constraint store C, i.e. ask and tell. The most important features added in the sCCP is the continuous random variable T, associated with each instruction. It represents the time needed to perform the corresponding operations in the store. T is exponentially distributed, and its probability density function is $f(\tau) = \lambda e^{-\lambda\tau}$ where λ is a positive real number (rate of the exponential random variable) representing the expected

[*] Student.

[**] Supervisor.

frequency per unit of time. The duration of an ask or a tell can depend on the state of the store at the moment of the execution.

- In ntcc[1], time is conceptually divided into discrete intervals. In a time unit, a process P gets an input c (a constraint) from the environment; it executes with this input as the initial store, and it outputs the resulting store d to the environment, when it reaches its resting point. The resting point determines a residual process Q, which is then executed in the next time unit. Information is not automatically transferred from one time unit to the following but by using the "next" operator.

Goal of the Research. We want to model biochemical reactions with the three cited languages, based on Concurrent Constraint programming, in order to explain the differences of the models and the features that we plan to implement, to have the programs as similar as possible to the real behaviuour.

Current Status of the Research. In our research we plan to modify and enrich the syntax of the previous programs, by extending different operators, that enable us to model the different relations among various kinds of biological entities. The most important are new temporal variables, to organize ask and tell in the store, and a way to model the collaborative procedures in one unique execution, i.e. several executions at the same exact time.

Open Issues and Expected Achievements. We are also interested in studying another language based on CCP, i.e. the Hybrid CC[5], and the differences between this language and the others [2][1], that we have already examined.

References

1. Gutiérrez, J., Pérez, J.A., Rueda, C., Valencia, F.D.: Timed Concurrent Constraint Programming for Analysing Biological Systems
2. Bertolussi, L., Policritti, A.: Modeling Biological systems in Stochastic Concurrent Constraint Programming
3. Gupta, V., Jagadeesan, R., Saraswat, V., Bobrow, D.G.: Programming in Hybrid constraint languages
4. Bertolussi, L.: Ph.D. Thesis. Mathematical Modeling of Biological Systems, ch. 3, pp. 26–50 (2007)
5. Gupta, V., Jagadeesan, R., Saraswat, V., Bobrow, D.: Programming in hybrid constraint languages

Author Index

Alberti, Marco 387
Aranda, Jesús 509
Ashley-Rollman, Michael P. 265, 527
Assayag, Gerard 509

Bistarelli, Stefano 559, 561
Bottalico, Marco 561
Broecheler, Matthias 40

Caballero, Rafael 449
Campbell, Jason D. 265
Campli, Paola 559
Carro, Manuel 190
Chen, Xiaoping 130
Chico de Guzmán, Pablo 190
Christiansen, Henning 55, 70
Costa, Jorge 205
Costa, Vítor Santos 372

Dal Palù, Alessandro 115
Dao-Tran, Minh 145
De Raedt, Luc 39
De Koninck, Leslie 479, 484
Delgrande, James 160
Denecker, Marc 22, 296
De Rosa, Michael 557
De Schreye, Danny 220, 499, 504
Dovier, Agostino 115
Ducassé, Mireille 38
Duck, Gregory J. 484
Dung, Phan Minh 84

Eiter, Thomas 145, 342
Erdem, Esra 342
Erdoğan, Halit 342

Ferreira, Michel 372
Fink, Michael 145, 342
Fodor, Paul 432, 525

Gallagher, John P. 55
Gavanelli, Marco 387
Gebser, Martin 235, 250
Goldstein, Seth Copen 265
Grosof, Benjamin 432

Hanh, Do Duc 84
Hermenegildo, Manuel 190, 281
Hildebrandt, Thomas 417
Hölldobler, Steffen 464

Inclezan, Daniela 542

Ji, Jianmin 130, 532

Kaminski, Roland 250
Kaufmann, Benjamin 250
Kencana Ramli, Carroline Dewi Puspa 464
Kifer, Michael 432
King, Andy 37
Krennwallner, Thomas 145

Lamma, Evelina 387
Lee, Peter 265
Li, Chendong 530
Liang, Senlin 432
Lierler, Yuliya 489
Lifschitz, Vladimir 489
Lin, Fangzhen 130
López, Hugo A. 417
Lopez-García, Pedro 281

Mera, Edison 281
Moniz Pereira, Luís 519
Moura, Paulo 23
Mungall, Chris 1

Namasivayam, Gayathri 555

Olarte, Carlos 509
Ostrowski, Max 235

Paolucci, Alessio 536
Pérez, Jorge A. 509
Pillai, Padmanabhan 265
Pilozzi, Paolo 499, 504, 534
Pinto, Alexandre Miguel 519
Pontelli, Enrico 99, 115

Raimundo, João 205
Raiser, Frank 540
Rocha, Ricardo 205
Rodríguez-Artalejo, Mario 449
Romero-Díaz, Carlos A. 449
Rossi, Gianfranco 115
Rueda, Camilo 509

Sakama, Chiaki 99
Sarna-Starosta, Beata 357
Sato, Taisuke 24
Schaub, Torsten 160, 235, 250, 312
Schrijvers, Tom 357
Simari, Gerardo I. 40, 538
Šimkus, Mantas 551
Sneyers, Jon 494
Son, Tran Cao 99
Stuckey, Peter J. 484
Subrahmanian, V.S. 40
Swift, Terrance 514, 519

Thang, Phan Minh 84
Theil Have, Christian 549
Thiele, Sven 312
Thielscher, Michael 327
Todorova, Yana 544
Tompits, Hans 160
Toro, Mauricio 509
Torp Lassen, Ole 70
Truszczyński, Mirosław 175

Valencia, Frank D. 509
Vaz, David 372
Vlaeminck, Hanne 296
Voets, Dean 220, 553

Wan, Hui 432, 547
Wielemaker, Jan 36
Wittocx, Johan 296
Woltran, Stefan 160

Zhou, Neng-Fa 402