Lecture Notes in Computer Science 2237
Edited by G. Goos, J. Hartmanis, and J. van Leeuwen

Springer
Berlin
Heidelberg
New York
Barcelona
Hong Kong
London
Milan
Paris
Tokyo

Philippe Codognet (Ed.)

Logic Programming

17th International Conference, ICLP 2001
Paphos, Cyprus, November 26 – December 1, 2001
Proceedings

Springer

Volume Editor

Philippe Codognet
University of Paris 6, LIP6, case 169
8 rue du Capitaine Scott, 75015 Paris, France
E-mail: Philippe.Codognet@lip6.fr

Cataloging-in-Publication Data applied for

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Logic programming : 17th international conference ; proceedings / ICLP 2001, Paphos, Cyprus, November 26 - December 1, 2001. Philippe Cocognet (ed.). - Berlin ; Heidelberg ; New York ; Barcelona ; Hong Kong ; London ; Milan ; Paris ; Tokyo : Springer, 2001
(Lecture notes in computer science ; Vol. 2237)
ISBN 3-540-42935-2

CR Subject Classification (1998): D.1.6, I.2.3, D.3, F.3, F.4.1

ISSN 0302-9743
ISBN 3-540-42935-2 Springer-Verlag Berlin Heidelberg New York

Springer-Verlag Berlin Heidelberg New York
a member of BertelsmannSpringer Science+Business Media GmbH

http://www.springer.de

Typesetting: Camera-ready by author, data conversion by Steingräber Satztechnik GmbH, Heidelberg
Printed on acid-free paper SPIN: 10845826 06/3142 5 4 3 2 1 0

Preface

A new millennium has started and new tools, heirs of three decades of research and development in the Logic Programming paradigm, are bringing new solutions to cope with the increasing complexity of today's computer systems. Computational logic in general and logic programming in particular will always play a key role in the understanding, formalizing, and development of complex software.

ICLP 2001 was the 17th International Conference on Logic Programming and continued a series of conferences initiated in Marseille, France, in 1982. This year ICLP was held in conjunction with CP 2001, the 7th International Conference on Principle and Practice of Constraint Programming. A coordinated program schedule and joint events were organized in order to maximize the interaction between these two neighboring communities. LOPSTR 2001, the 11th international workshop on Logic-based Program Synthesis and Transformation was also co-located with ICLP 2001 and CP 2001 this year, bringing together a larger community to share novel research results. Seven satellite workshops were also associated to the conference and took place on the day following the conference.

We received 64 papers, among which 23 were selected for presentation at the conference and inclusion in the conference proceedings. In addition to paper presentations, the conference program also included four invited talks and four tutorials. We chose this year to celebrate the founders of the logic programming field, namely Alain Colmerauer and Bob Kowalski, who are both celebrating their 60th birthday. The other two invited talks were given by Patrick Cousot, pioneer in the field of abstract interpretation, and Ashish Gupta, who presented his industrial experience on cross-enterprise databases at amazon.com and Tavant Technologies. The four tutorials were given by Eric Villemonte de la Clergerie, V.S. Subrahmanian, Kazunori Ueda, and Jan Wielemaker.

I would like to thank all the authors of the submitted papers, the Program Committee members, and the referees for their time and efforts spent in the reviewing process, the conference chair Tony Kakas and his team at the University of Cyprus for the excellent organization of the conference, and Toby Walsh, the CP 2001 program chair, for his constant cooperation and interaction. Last but not least, special thanks to Yoann Fabre at the University of Paris 6 for taking care of installing and maintaining the paper review system.

September 2001 Philippe Codognet

Organization

Conference Chair

Tony Kakas	University of Cyprus

Program Chair

Philippe Codognet	University of Paris

Workshop Chair

Antonio Brogi	University of Pisa

Program Committee

Krzysztof Apt	CWI, Amsterdam
Frederic Benhamou	University of Nantes
Veronica Dahl	Simon Fraser University
Bart Demoen	University of Leuven
Danny De Schreye	University of Leuven
Jürgen Dix	University of Koblenz
Gilberto Filé	University of Padova
Seif Haridi	SICS, Stockholm
Manuel Hermenegildo	Technical University of Madrid
Pat Hill	University of Leeds
Gerard Huet	INRIA
Antonis Kakas	University of Cyprus
Michael Maher	Loyola University Chicago
Kim Marriott	Monash University
Catuscia Palamidessi	Pennsylvania State University
Luis Moniz Pereira	New University of Lisbon
Andreas Podelski	Max-Planck Institute
V.S. Subrahmanian	University of Maryland
Kazunori Ueda	Waseda University, Tokyo
Pascal Van Hentenryck	Brown University
Toby Walsh	University of York

Referees

Salvador Abreu
M. Aiello
Hassan Aït-Kaci
Jose Julio Alferes
James Bailey
Francois Barthelemy
S. Billot
Philippe Blache
Annalisa Bossi
S. Brand
Gerhard Brewka
Maurice Bruynooghe
Roberto Bruni
Manuel Carro
Witold Charatonik
Livio Colussi
Agostino Cortesi
Patrick Cousot
Carlos Viegas Damásio
F. de Boer
Eric de la Clergerie
Pierangelo Dell'Acqua
Marc Denecker
Alexander Dikovsky
Mireille Ducassé
Lyndon Drake
Wolfgang Faber
Francois Fages
Gérard Ferrand
M. Garcia de la Banda
Ulrich Geske
Roberto Giacobazzi
Roberta Gori
Frederic Goualard
John Grant
Laurent Granvilliers
Gopal Gupta
Chris Hankin
Michael Hanus
Katsumi Inoue
Gerda Janssens
Arnaud Lallouet
Evelina Lamma
Antonio Fernández Leiva
Vladimir Lifschitz
Wiktor Marek
Nancy Mazur
Bernd Meyer
Laurent Michel
Dale Miller
Phan Minh Dung
Eric Monfroy
Frank Morawietz
J. J. Moreno Navarro
Hector Munoz-Avila
Hiroshi Nakashima
Dana Nau
Ilkka Niemela
Fatma Ozcan
Gerald Penn
Enrico Pontelli
António Porto
Paulo Quaresma
Christian Rétoré
Gilles Richard
Francesca Rossi
S. Ruggieri
D. Saccá
Chiaki Sakama
Vitor Santos Costa
Frederic Saubion
Francesca Scozzari
Alexander Serebrenik
Kish Shen
Jan-Georg Smaus
Harald Sondergaard
Fausto Spoto
Iain Duncan Stalker
Robert Stárk
Frieder Stolzenburg
Jürgen Stuber
Terrance Swift
Paul Tarau
Miroslaw Truszczynski
Hudson Turner
Silvio Valentini
Ruben Vandeginste
W. J. van Hoeve
Bert Van Nuffelen
Sofie Verbaeten
Christine Vrain
Enea Zaffanella
Carlo Zaniolo
Jean-Daniel Zucker

Sponsors

The Association for Logic Programming, the University of Cyprus,
Cyprus Telecommunications Authority, and IBM.

Table of Contents

Invited Speakers

Tutorials

Conference Papers

Solving the Multiplication Constraint in Several Approximation Spaces

Alain Colmerauer

Universités de la Méditerranée et de Provence
Laboratoire d'Informatique de Marseille, CNRS
13288 Marseille Cedex 9, France
alain.colmerauer@univ-mrs.fr

Given three intervals a, b, c in the real numbers and the multiplication constraint $z = xy \wedge x \in a \wedge x \in b \wedge x \in c$, we are interested in establishing and justifying formulas for computing the smallest intervals a', b', c' which substituted for a, b, c do not modify the set of solutions of the constraint.

We study three cases : (1) the well-known case where a, b, c, a', b', c' are closed intervals, (2) the case where a, b, c, a', b', c' are intervals, eventually open or not bounded, (3) the case where a, b, c, a', b', c' are intervals, eventually open or not bounded, whose lower and upper bounds, if their exist, are taken from a given finite set.

For this we introduce the general notions of approximation space, of good relation, of extension and aggregation of relations and establish three properties which can be used for solving other constraints.

P. Codognet (Ed.): ICLP 2001, LNCS 2237, p. 1, 2001.

Is Logic Really Dead or Only Just Sleeping?

Robert Kowalski

Department of Computing, Imperial College, London, UK
rak@doc.ic.ac.uk
http://www-lp.doc.ic.ac.uk/UserPages/staff/rak/rak.html

There was a time when Logic was the dominant paradigm for human reasoning. As George Boole put it around one hundred and fifty years ago, Logic was synonymous with the "Laws of Thought". Later, for most of the latter half of the twentieth century, it was the mainstream of Artificial Intelligence. But then it all went wrong. Artificial Intelligence researchers, frustrated by the lack of progress, blamed many of their problems on the logic-based approach. They argued that humans do not reason logically, and therefore machines should not be designed to reason logically either. Other approaches began to make progress where Logic was judged to have failed - approaches that were designed to simulate directly the neurological mechanisms of animal and human intelligence. Insect-like robots began to appear, and the beginning of a new Machine Intelligence was born. Logic seemed to be dieing - and to be taking Logic Programming (LP) with it.

The possible death of Logic has important implications for LP, because arguably the main argument for LP is:

- that LP is based on Logic,
- that Logic is the foundation of human reasoning, and
- that, therefore, LP is more human-oriented and user-friendly than computer languages developed mainly for machines.

It is possible to quarrel with both premises of the argument. In particular, the first premise that LP is based on Logic has been attacked by some LP researchers themselves, arguing that Logic Programs are better understood as inductive definitions. This alternative view of the foundations of LP has much technical merit, but it potentially undermines the main argument for LP. I will consider one way of rescuing the argument, by outlining how inductive definitions can be incorporated in a more general Logic of thinking, as part of a more comprehensive observation- thought-action agent cycle. However, my main concern here is with attacks against the second premise of the argument: that Logic is fundamental to human thinking. These attacks include the old, familiar ones advocating alternative symbolic approaches, most notably condition-action rules. They also include more recent ones advocating non-symbolic connectionist and situated intelligence approaches. I will examine some of these attacks and try to distinguish between those that are justified and those that are simply wrong.

I will argue that, to address these attacks and to be in a better position to fight back, Logic and LP need to be put into place: Logic within the thinking

P. Codognet (Ed.): ICLP 2001, LNCS 2237, pp. 2–3, 2001.

component of the observation-thought-action cycle of a single agent, and LP within the belief component of thought. In addition to LP, a complete model of Computation and Human Reasoning also needs: a logical goal component (which includes condition-action rules), other kinds of non-symbolic thinking, and a framework that includes other agents.

I will argue that the observation-thought-action cycle provides a more realistic framework, not only for Logic as a descriptive theory of how humans actually think, but also for Logic as a prescriptive theory of how humans and computers can reason more effectively. With such a more realistic framework, even if Logic and LP might be only half awake today, they can at worst be only sleeping, to come back with renewed and more lasting vigour in the near future.

Design of Syntactic Program Transformations by Abstract Interpretation of Semantic Transformations*

Patrick Cousot

Département d'informatique, École Normale Supérieure,
45 rue d'Ulm, 75230 Paris cedex 05, France
Patrick.Cousot@ens.fr
http://www.di.ens.fr/∼cousot/

Traditionally, static program analysis has been used for offline program transformation i.e. an abstraction of the subject program semantics is used to determine which syntactic transformations are applicable. A classical example is binding-time analysis before partial evaluation [4,5].

We present a new application of abstract interpretation to the formalization of source to source program transformations:

– The semantic transformation is understood as an abstraction of the subject program semantics. The intuition is that the transformed semantics is an approximation of the subject semantics because, most often, redundant elements of the subject semantics have been eliminated;
– The correctness of the semantic transformation is expressed by an observational abstraction. The intuition is that the subject and transformed semantics should be exactly the same when abstracting away from irrelevant hence unobserved details;
– Finally, the syntax of a program is shown to be an abstraction of its semantics (in that details of the execution are lost) so that the transformed program is an abstraction of the transformed semantics.

Abstract interpretation theory [1,2] provides the ingredients for designing a syntactic source-to-source transformation as an abstraction of a semantics-to-semantics transformation, which correctness is formally established through an observational abstraction. In particular iterative transformation algorithms are abstraction of the fixpoint semantics of the subject program.

Several examples have been studied with this perspective such as blocking command elimination [3], program reduction, constant propagation, partial evaluation, etc.

References

1. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In 6^{th} *POPL*, pages 269–282, San Antonio, TX, 1979. ACM Press.

* This work was supported in part by the european FP5 project IST-1999-20527 Daedalus.

P. Codognet (Ed.): ICLP 2001, LNCS 2237, pp. 4–5, 2001.

2. P. Cousot and R. Cousot. Abstract interpretation frameworks. *J. Logic and Comp.*, 2(4):511–547, Aug. 1992.
3. P. Cousot and R. Cousot. A case study in abstract interpretation based program transformation: Blocking command elimination. *ENTCS*, 45, 2001. `http://www.elsevier.nl/locate/entcs/volume45.html`, 23 pages.
4. N. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Int. Series in Computer Science. Prentice-Hall, June 1993.
5. N.D. Jones. An introduction to partial evaluation. *ACM Comput. Surv.*, 28(3):480–504, Sep. 1996.

X-tegration – Some Cross-Enterprise Thoughts

Ashish Gupta

Chief Strategy Office, Tavant Technologies
542 Lakeside Drive, Suite 5, Sunnyvale, CA 94085
http://www.tavant.com

One of the main contributions of the internet "revolution" is to make the concept of connectivity the efault state in the minds of individuals and enterprises. Whereas previously it was very difficult to imagine connecting the systems of an enterprise to any other enterprise, today business managers are accepting such connectivity as inevitable. This new class of applications will be referred to as XERP (cross ERP) in this talk. The author discusses the nature of XERP applications and how they differ from ERP applications. We will also discuss some of the practical problems encountered in making XERP applications a reality.

Despite the tremendous amount of publicity and hype around the first few XERp applications - namely exchanges - the fact remains that they are not very common. Yet there is a tremendous amount of software that has been built as infrastructure for XERP applications and several hundred companies formed to implement XERP applications in different spaces. Most of these have been formed using adhoc principles, poorly understood business processes, nonexistent theoretical foundations, and are frequently ahead of the customer.

We will discuss some areas where there is possibly room to explore the theoretical foundations of XERP applications. We will also discuss several distinguishing characteristics and therefore opportunities in this space. The talk will also be structured to solicit opinions of the audience given the open endedness of the topic at hand.

P. Codognet (Ed.): ICLP 2001, LNCS 2237, p. 6, 2001.

Building Real-Life Applications with Prolog

Jan Wielemaker

University of Amsterdam, SWI,
Roeterstraat 18, 1018 WB Amsterdam, The Netherlands
jan@swi.psy.uva.nl
http://www.swi.psy.uva.nl/usr/jan/

SWI-Prolog grew towards a popular free Prolog implementation. It stresses on features that make it a useful prototyping tool. SWI-Prolog is closely compatible with Quintus and other Prologs in the Edinburgh family and is used for teaching by many Universities.

XPCE/Prolog, based on XPCE (an Object Oriented GUI toolkit for dynamically typed languages), is a powerful environment for GUI prototyping and the implementation of large interactive systems. It has been used for the development of several knowledge engeneering workbenches. XPCE/Prolog is being used by various research groups in universities and industry.

P. Codognet (Ed.): ICLP 2001, LNCS 2237, p. 7, 2001.

Natural Language Tabular Parsing

Éric Villemonte de la Clergerie

ATOLL/INRIA
Domaine de Voluceau - BP 105, 78153 Le Chesnay Cedex – France
Eric.De_La_Clergerie@inria.fr
WWW Home page: http://atoll.inria.fr/~clerger

This tutorial mostly addresses the use of tabular techniques for parsing natural languages. During a tabular evaluation, traces of computations are stored in a table in order to detect loops, to share computations and to provide a final justification for answers. While widely used in Natural Language Processing to cope with the high level of ambiguity found in human languages, tabular techniques have also been developed in Functional Programming (memoization), Deductive Databases (magic-set), and Logic Programming (tabling).

We first start with a brief review of different grammatical formalisms, pointing out areas of interest for (Constraint) Logic Programming approaches.

Then, we cover various tabular techniques (CKY, Chart Parsing, Graph-Structured Stacks, Automata & Dynamic Programming) for various formalisms (Context-Free Grammars, Unification Grammars, Tree Adjoining Grammars), trying to sketch an uniform view of these techniques. We mention the relationships with tabular techniques used in LP (Magic Set and tabling).

We briefly present the notions of parse and derivation shared forests produced by tabular parsers and their transposition for LP (notion of justification or proof forest).

The tutorial is illustrated with the presentation of DyALog, a system used to compile unification grammars, TAGs and Logic Programs.

This tutorial derives from a longer one in French delivered at TALN'99, whose slides and notes may be found at http://atoll.inria.fr/~clerger/TALN99.html. Slides for this tutorial will be available at http://atoll.inria.fr/~clerger/ICLP01.html.

P. Codognet (Ed.): ICLP 2001, LNCS 2237, p. 8, 2001.

A Close Look at Constraint-Based Concurrency

Kazunori Ueda

Dept. of Information and Computer Science, Waseda University
3-4-1, Okubo, Shinjuku-ku, Tokyo 169-8555, Japan
ueda@ueda.info.waseda.ac.jp

Constraint-based concurrency, also known as the cc (concurrent constraint) formalism, is a simple framework of concurrency that features (i) asynchronous message passing, (ii) polyadicity and data structuring mechanisms, (iii) channel mobility, and (iv) nonstrictness (or computing with partial information). Needless to say, all these features originate from the use of constraints and logical variables for interprocess communication and data representation. Another feature of constraint-based concurrency is its remarkable stability; all the above features were available essentially in its present form by mid 1980's in concurrent logic programming languages.

Another well-studied framework of concurrency is *name-based concurrency* represented by the family of π-calculi, in which names represent both channels and tokens conveyed by channels. Some variants of the original π-calculus featured asynchronous communication, and some limited the use of names in pursuit of nicer semantical properties. These modifications are more or less related to the constructs of constraint-based concurrency. Integration of constraint-based and name-based concurrency can be found in proposals of calculi such as the γ-calculus, the ρ-calculus and the Fusion calculus, all of which incorporate constraints or name equation into name-based concurrency.

This tutorial takes a different, analytical approach in relating the two formalisms; we compare the rôles of logical variables and names rather than trying to integrate one into the other. Although the comparison under their original, *untyped* setting is not easy, once appropriate type systems are incorporated to both camps, name-based communication and constraint-based communication exhibit more affinities. The examples of such type systems are linear types for the π-calculus and the mode/linearity/capability systems for Guarded Horn Clauses (GHC). Both are concerned with the polarity and multiplicity of communication, and prescribe the ways in which communication channels can be used. They help in-depth understanding of the phenomena occurring in the course of computation.

The view of constraint-based concurrency that the tutorial intends to provide will complement the usual, abstract view based on *ask* and *tell* operations on a constraint store. For instance, it reveals the highly local nature of a constraint store (both at linguistic and implementation levels) which is often understood to be a global, shared entity. It also brings resource-consciousness into the logical setting. This is expected to be a step towards the deployment of cc languages as a common platform of non-sequential computation including parallel, distributed, and embedded computing.

P. Codognet (Ed.): ICLP 2001, LNCS 2237, p. 9, 2001.

Probabilistic Databases and Logic Programming

V.S. Subrahmanian

University of Maryland, Dept. of Computer Science, USA
vs@cs.umd.edu

Uncertainty occurs in the world in many ways. For instance, image processing programs identify the content of images with some levels of uncertainty. Prediction programs predict when events will occur with certain probabilities. In this tutorial, I will focus on probabilistic methods to handle uncertainty.

We will start with a quick review of how probabilistic information can be handled in a relational database setting. An extension of the relational algebra to handle probabilistic data will be described. Subsequently, we will discuss methods to handle probabilities over time in a relational database. Intuitively, when we make statements such as "Package p will be delivered sometime today", the granularity of time used will have an impact on how easy/difficult it is to store such data. If the granularity of time used is milliseconds, the above statement leads to an enormous amount of uncertain information. Techniques to store and manipulate such information will be studied.

We continue with a description of probabilistic logic programming methods that extend the above database models to logic programs. We will also discuss the incorporation of temporal uncertainty in logic programs. We will describe characterizations of such programs in terms of logical model theory, fixpoint theory, and proof theory.

The talk will conclude with some directions for future research.

P. Codognet (Ed.): ICLP 2001, LNCS 2237, p. 10, 2001.

Understanding Memory Management in Prolog Systems

Luís Fernando Castro[1] and Vítor Santos Costa[2]

[1] Dept. of Computer Science, SUNY at Stony Brook,
luis@cs.sunysb.edu
[2] COPPE/Sistemas, Universidade Federal do Rio de Janeiro,
vitor@cos.ufrj.br

Abstract. Actual performance of Prolog based applications largely depends on how the underlying system implements memory management. Most Prolog systems are based on the WAM, which is built around a set of stacks. The WAM is highly optimized to recover memory on backtracking and on tail-recursive predicates. Still, deterministic computations can create intermediate structures that can only be freed through garbage collection.

There is a significant amount of literature regarding memory management for Prolog. Unfortunately, we found relatively little data on how modern Prolog systems perform memory-wise. Open questions range from whether Prolog systems consume the same amount of space, to how effective garbage collection is in practice, and to whether we should be using sliding or copying based garbage collectors.

This work aims at investigating the practical aspects of memory management in Prolog systems. We present a methodology to compare the memory performance of such systems, and we use it to compare two different WAM-based systems, namely XSB and Yap. We suggest novel techniques for variable shunting and we propose a scheme that can improve the performance of sliding-based garbage collectors. Last, we evaluate our methodology with larger-scale applications.

1 Introduction

Prolog is a high-level, declarative language based on SLD-resolution applied over Horn clauses, a powerful subset of First Order Logic. Prolog has been used with success in such diverse fields as intelligent database processing, natural language processing, machine learning, software engineering, model checking and expert system design. Traditional Prolog systems implement a selection function that always tries the leftmost goal first and performs a depth-first search with backtracking. More recent systems support features such as co-routining for more flexible selection of goals, constraints, and tabling.

Memory allocation is quite an important issue in the implementation of Prolog systems. The first main data-structure is the Prolog internal database. Database structures are long lived and must be explicitly freed by the programmer. The run-time environment also manages several dynamic data-areas that

P. Codognet (Ed.): ICLP 2001, LNCS 2237, pp. 11–26, 2001.

grow during forward execution and contract during backtracking. In WAM-based implementations [17] these data structures are: the *Local stack*, that maintains environments, that is, call records, and choice-points, that is, open alternatives; the *Global stack* or *Heap*, that maintains data structures whose can last even after the procedure that originally created them terminates; and the *Trail*, that stores the data required to reset bindings to variables when backtracking.

Under some circumstances, namely for deterministic tail recursive calls, the WAM can reclaim space on the Local stack during forward execution [17]. Some space on the trail can also be reclaimed after pruning. Most Prolog systems therefore only reclaim global stack space either on backtracking or through garbage collection. This is unfortunate because in many applications memory consumption is dominated by the global stack.

In this work, we study memory allocation in WAM-based systems. We address what we believe are two major issues. First, we propose an initial methodology to compare the memory performance of Prolog systems, and use it to compare two different WAM-based systems. Second, we demonstrate the applicability of our methodology on larger-scale applications.

We next discuss in some more detail the main issues in memory management. We then proceed to present our methodology, and present an evaluation for the XSBand YAP systems. We start from general memory patterns and then focus on garbage collection issues. Next, we evaluate some larger scale applications. At last, we present our conclusions and propose future work.

2 Principles of Memory Management

The traditional Marseille Prolog used a single stack for the management of Prolog data areas. More efficient systems allow recovering memory during forward execution by dividing this stack into three or four stacks. In the WAM these are the Heap, the Local (or environment plus choice-point) stack, and the Trail.

The Local Stack. The *Local stack* includes environments and choice-points. Environments correspond to live clauses, and are the equivalent of activation records on imperative languages. They have two control fields in the WAM, plus a clause dependent number of slots for permanent variables, that is, for variables that were created in the body of the clause and that span several sub-goals. The WAM provides several optimizations for *deterministic* computations such as last call optimization and environment trimming. Choice-points correspond to unexplored clauses in the search tree. They are created for non-determinate goals, that is, for goals where we cannot prove at entry that at most one clause matches. In database terminology they are a snapshot: we are to recover the state of a computation at the moment of goal entry from the data in the choice-point. Choice-points include the current stack tops, program pointers, and the arguments for the current goal. Choice-points are allocated at clause entry and are recovered at backtracking or after execution of a cut. Cut is quite important, as support for last call optimization also guarantees that all environments created

after the target choice-point for the cut can be discarded. Several WAM-derived systems, such as SICStus Prolog [1] or XSB [12] maintain separate choice-point and environment stacks.

The Trail. The trail is a data structure unique to Prolog. It stores all conditional bindings, that is, all bindings that must be reset after backtracking. It grows as we perform unifications and it contracts during backtracking. Cut may also contract the trail, because it means that some bindings will no longer be conditional.

The Heap. The global stack, or *Heap*, accommodates compound terms and variables that do not fit in the environment. More precisely, we can say that the global stack stores four different kinds of objects: **(i)** free variables; **(ii)** Prolog variables bound to constants or compound terms; **(iii)** references, that is variables bound to other variables; and, **(iv)** functors. A compound term is a functor plus an ordered set of possibly bound variables. The WAM does not explicitly store the functor for pairs.

2.1 Garbage Collection

Garbage collection for Prolog was first implemented in the landmark DEC-10 Prolog system [16]. The classical approach uses a mark-and-slide collector as detailed by Appleby et al. [3] for their seminal work on the SICStus garbage collector. The marking step tags all variables reached from the current state, or from a choice-point. An extra sweep over the other stacks and two sweeps over the Heap adjust the pointers and then compact the Heap. Variable ordering is always preserved in this scheme, thus respecting stack segmentation imposed by the choice-points. Further optimizations are:

- Trail references to dead objects can, and must be, reset (or the location they point to must be made live). This is called *early reset* [3].
- The top of Heap pointer in choice-points may also point to dead objects. One solution is to create a live object at this position. A different solution is to keep track of the current choice-point while sweeping the Heap, and adjust choice-pointers accordingly.
- Reference chains can be compressed. This is called *variable shunting* [11]. A complete implementation on systems that perform pointer reversal at marking requires an extra step for the garbage collector.
- It is quite common that the garbage collector will not recover enough space. Examples are non-deterministic programs and numerical integer computations that may be heavily recursive whilst not creating many objects in the Heap. In this case, the system must be able to support *stack expansion*.

The main disadvantage of mark and sweep garbage collectors is that they take time linear on the size of the Heap. This can be a problem if the Heap grows very large. One alternative is to use copying based garbage collectors for Prolog (please refer to Demoen and Sagonas for a recent discussion [7]).

Last, one should notice that Heap garbage collection can also be used to recover cells in the Trail. This is possible through early reset, and in case of Yap by discarding unconditional bindings. XSB does not recover trail space at garbage collection: XSB performs trail trimming at cut time, and early reset does not actually discard trail entries. XSB uses early reset only to recover Heap space by releasing compound terms.

3 Methodology

Understanding the memory performance of WAM based systems is a hard task. Parameters we can experiment with include very different applications, different designs for Prolog engines that build different intermediate structures, and how much and when the engine tries to recover memory. Variables we can measure include total memory sizes, memory distribution per object type, and time for the different algorithms.

In this work we focus on what we argue are two major issues in Prolog memory allocation. The first one is to understand *how Prolog allocates objects.* To achieve this goal, we instrumented each benchmark to call the garbage collector at fixed points, and to find out how objects were distributed before and after garbage collection. Note that our interest at this point is to understand how memory is being allocated at "typical" execution points. Questions we address are types of objects being allocated, and whether there is a significant variation between Prolog engines. Towards this study we chose to use well understood benchmarks.

The second problem we focus on is *garbage collection*: we would like to know whether some well-known techniques for garbage collection of Prolog programs are worthwhile. One question of interest we discuss involves collection of aliased pointers. The second question involves improving sliding based collection, towards improving performance. The limitations of our initial methodology are shown when studying sliding-based collection, as actual differences are only significant for real, substantial applications.

The Benchmark Set. Our initial benchmark set includes a mix of deterministic and non-deterministic applications. To avoid interference with the stack protection algorithm we called garbage collection explicitly. We always place calls to the garbage collector at the end of the execution, when most of the data structures had been built but before any final cuts. Program sources can be found at `http://www.cs.sunysb.edu/~luis/gc-bench.tar.gz`. Note that the program may call the garbage collector several times. We only present statistics for the instance of garbage collection which presented the largest initial Heap usage, and mention any significant variations. The applications are Kish Shen's simulator of AND/OR parallelism, the well known Boyer theorem prover benchmark, the Gnu-Prolog compiler, Mike Carlton's chess-playing game, Bruce Holmer's program for NAND decomposition, and the Chat Natural Language Analyser and database interface.

We use two systems to perform our analysis: XSB and Yap. Both are WAM based. Yap has a Local stack, whereas XSB has a choice-point and an environment stack. Before this work, Yap only supported mark-and-slide garbage collection, whereas XSB supported both mark-and-slide and copying garbage collection.

4 Memory Management for WAM-Based Systems

Table 1 presents the first results of our analysis: we show how many choice-points were active when we called the garbage collector and how much stack space was being allocated at this point. The results indicate a clear difference between the first three deterministic benchmarks, and the remaining non-deterministic benchmarks.

For Yap, all the deterministic benchmarks show little usage of local memory. Only `gprolog` creates two choice-points, the other 4 choice-points are from the top-level. Little space is used on environments, which shows how effective last call optimization can be. Heap usage is quite high, and fully dominates memory usage. Yap actually consumes more trail than Local stack for these benchmarks, since it does not recover Trail space when executing cut.

For XSB, the numbers for the Local stack column are the sum of those for the environment and choice-point stacks. Regarding choice-points, the behavior of XSB is mostly consistent with that of Yap. XSB always uses more Local stack than Yap. The `gprolog` benchmark creates considerably more choice-points and uses much more local stack before the first run of the garbage collector. We have traced this problem to be due to a non-optimal implementation of backtracking hooks support which created unnecessary choice-points. A fix has been created and included in the system, but since the extra choice-points don't affect the rest of the results significantly, we decided to show the results as per XSB Version 2.3's behavior. The effect of Trail compaction on pruning can be seen on the differences on Trail usage on the deterministic benchmarks. XSB is able to recover Trail space almost completely before garbage collection is started.

Table 1. Memory Usage before Garbage Collection.

	Choice-Points		**Local Stack**		**Heap**		**Trail**	
Programs	**Yap**	**XSB**	**Yap**	**XSB**	**Yap**	**XSB**	**Yap**	**XSB**
`sim`	4	9	1018	2304	336580	285402	38443	4
`boyer`	4	9	86	172	408880	144000	58584	4
`gprolog`	6	369	125	13082	124882	192571	2914	2154
`nand`	264	296	4730	6306	1859	1425	1096	830
`chat`	242	293	4686	6107	4423	4028	1355	879
`chess`	12996	17968	291091	445701	122301	88361	32723	32388

The non-deterministic benchmarks present a very different story. The `nand` benchmark performs search over a shallow tree with a high branching factor. It has a very small memory footprint, even for large running-times. We were surprised by the high number of choice-points it creates. On closer analysis we found out that most choice-point are for deterministic set operations, and could have been pruned away.

Execution of `chat` follows two different stages: parsing and query optimization. In both cases, the number of choice-points and the Heap memory usage tend to increase with problem size. In the benchmark, `chat` handles relatively small problems, so the memory footprint is small. We were also surprised at finding so many choice-points for `chat`. In this case, closer analysis showed that missing cuts in `simplify/3` and `split_quants/6` were the main culprits, but even so `chat` has quite a few non-deterministic procedures. The `chess` benchmark is quite interesting: it creates a huge number of choice-points, but it also creates objects in memory. Again, more than 90% of them correspond to a deterministic procedure, `strength/3`. Notice that Trail usage is significant in the non-deterministic benchmarks. In `nand` Trail usage is close to Heap usage. This is because almost every binding is conditional and few data structures need to be created in the Heap. The ratio is smaller for the other benchmarks, but it is still important.

The Heap. The Heap has some of the more interesting results in Table 1, with significant differences between XSB and Yap. To better understand these results, Table 2 shows which objects we can find in the Heap before we enter garbage collection. The first three columns represent unbound variables, references, and their percentage over the total number of cells. The next four columns represent cells bound to constants (atoms or numbers), pairs, compound terms, and their total percentage. The last two columns represent the number of functors and the total percentage. These numbers have been obtained before garbage collection, so some (often most) of these objects are garbage.

Constants are the most popular object for most benchmarks. Prolog terms are trees, and for these benchmarks most leaves are constants, not unbound variables. The two exceptions are `gprolog` and `chess`. Both heavily use lists and (partially instantiated) compound terms. Unbound variables are only a small percentage of total Heap objects: `chat` with XSB is the one benchmark where more than 10% of the Heap is consumed with unbound variables, and this seems to be an artifact of the XSB implementation as Yap has much less unbound variables. References are important in Yap, less so in XSB.

The column on Functors gives the number of compound terms that were actually built. The correlation between the number of Functors and compound terms shows how many terms are being shared in the Prolog Heap: this sharing is particularly important in the `gprolog` compiler and in the `chess` program.

The results show huge differences between XSB and Yap. XSB performs much better in the deterministic benchmarks `sim` and especially in `boyer`, where Yap consumes three times as much memory. It performs worse in `gprolog`. XSB also performs better for the non-deterministic benchmarks, though by a smaller

Table 2. Memory Objects before Garbage Collection.

	Variables			Objects				Functors	
Programs	Unb	Refs	%	Const	Pairs	Comp	%	Functor	%
Yap									
`sim`	4267	39082	12.93%	153504	23643	57160	69.88%	57625	17.19%
`boyer`	1	94063	23.01%	194100	19	79415	66.90%	41282	10.10%
`gprolog`	1185	4012	4.19%	12881	49689	49906	90.58%	6501	5.24%
`nand`	5	2	0.38%	1162	472	110	93.81%	108	5.81%
`chat`	33	322	8.03%	1786	339	1127	73.52%	816	18.45%
`chess`	126	11039	9.13%	21872	42591	38281	84.01%	8392	6.86%
XSB									
`sim`	3918	20107	8.42%	118689	24754	59464	71.10%	58470	20.49%
`boyer`	17	5	0.02%	60710	21	41993	71.34%	41254	28.65%
`gprolog`	14393	4712	9.92%	41899	75013	49505	86.42%	7049	3.66%
`nand`	26	0	1.82%	770	472	71	92.14%	86	6.04%
`chat`	613	154	19.04%	1349	326	911	60.20%	675	16.76%
`chess`	811	4566	6.09%	13952	27805	34562	86.37%	6665	7.54%

factor. We researched into these differences and found two major contributing factors, **(i)** the compiler; and **(ii)** built-in implementation.

Regarding **(i)**, the Yap compiler traditionally allocates void variables or temporary variables in body goals in the Heap, not in the current environment. This is quite a bad idea if the program is deterministic and tail recursive, because Heap space is only recovered through garbage collection, but Local Stack will be recovered at last call. For instance, we found that the difference for the `boyer` benchmark results from the Yap compiler reserving Heap cells to store results for the `arg/3` and `functor/3` built-ins. XSB allocates the same cells in the Local stack, and manages to reduce Heap usage by a factor of three. The Yap-4.3.19 compiler addresses these problems by forcing void variables to be allocated in the Local Stack and by not initializing arguments to calls of `functor/3` and `arg/3` if we previously know the modes. Similar experience has been reported in [6]. Factor **(ii)** occurs when we have very different implementations of the same built-in. The factor explains why the number of Functor cells in the Heap differs between the two systems, as both implement copying and will create the same number of compound terms, modulo built-ins. We found that Yap would often use more space because the `write/1` and `nl/1` built-ins would leave garbage, such as terms describing the current stream, in the Heap.

5 Garbage Collection

The Heap is the dominant area in the deterministic benchmarks, and is quite important for non-deterministic benchmarks. Heap garbage collection recovers memory in the Heap and, in the case of Yap, also in the Trail. Table 3 shows how garbage collection performs on both systems.

Table 3. Effectiveness of Garbage Collection.

	Heap				Trail			
Programs	**Yap**		**XSB**		**Yap**		**XSB**	
`sim`	6558	98%	8252	98%	2	99%	4	0%
`boyer`	21	99%	39809	73%	2	99%	4	0%
`gprolog`	6247	94%	14663	93%	238	91%	1175	54.55%
`nand`	1468	21%	1332	7%	596	45%	122	14.70%
`chat`	1850	58%	2037	50%	433	68%	286	32.54%
`chess`	66798	45%	67503	24%	979	97%	26393	81.49%

Garbage collection is very effective for the deterministic benchmarks. For Yap, in all three cases more than nine in every ten cells were garbage. The `boyer` benchmark is an extreme case: Yap can recognize that almost everything is garbage at the point we force garbage collection. Garbage collection for `sim` was also performed quite at the end of the computation, and almost everything was recovered. Results for `gprolog` are somewhat worst, but well above 90%. We would expect these results to improve further as we increase benchmark size. The XSB garbage collector is somewhat less effective than Yap's. This is especially true on `boyer`. We found that environment trimming plays an important role in the efficiency of the collection for `boyer`. The test predicate stores the resulting term, which is quite large, in a local variable. This variable is dead by the time we call garbage collection. In fact, XSB does not perform environment trimming, nor does it implement any more sophisticated method to inform the garbage collector of live variables. The term ends up being marked, thus resulting in the large difference of efficiency between XSB and Yap.

The story is quite different for the non-deterministic benchmarks. The `nand` benchmark consumes little memory, and there is very little to recover. Only one in five cells is garbage. Notice that XSB recovers less cells than Yap, but that is only because it was using less cells in the first place. Ultimately, the intermediate data-structures we build in this particular query for `nand` are very small, and we build little we can discard. Parsing in `chat` is much more deterministic and we can recover quite a lot more: about half the cells are garbage. The worst result for both garbage collectors is from `chess`. Although we consume much more memory than for the other non-deterministic benchmarks, most Heap cells are still reachable at this point. This suggests that we can find non-deterministic programs which used significant amounts of Heap but where garbage collection may not perform well. Again, both Yap and XSB turn out to mark about the same number of cells in `chess`. Yap does recover more memory because in Yap temporary variables were stored in the Heap. These variables are easily recovered by the garbage collector. XSB stores the same variables in the environments, but space for the environments is protected by the choice-points and cannot be recovered. So it turns out that storing temporary variables in the Heap is a better strategy for `chess`, as space can always be recovered through garbage collection! We also experimented with removing the most obvious shallow choice-

points by introducing cuts in deterministic procedures, but our results do not show significant improvements.

Heap Objects. An analysis of the Heap objects after garbage collection revealed that the ratios between kinds of objects didn't change considerably. One interesting result is that most of the references left by Prolog were actually garbage. Even `chat` had a substantial reduction in references, eliminating about 70% of them. A second interesting observation is that the number of compound terms now closely tracks the numbers of functors for all benchmarks but `chess`. Only in the case of `chess` we still have many references to the same compound term (with 30359 compound terms and 4461 functors). This suggests that only one pointer to a compound term is actually useful. Finally, in `chat`, the number of free variables actually increases after garbage collection for Yap (from 33 to 121), whereas in XSB it strongly decreases (from 613 to 17). This is because Yap implements early reset by actually resetting the memory position to be a variable, whereas XSB stores a constant. This observation suggests that early reset is in fact quite important for `chat`.

5.1 Reference Chains

It has been shown before that reference chains are small in Prolog programs [13,14]. In fact there are few references. Table 4 shows how deep reference chains go in the two systems. We count reference chains starting both from the Heap and from Environments.

The longest chains are in `sim` and then in `chat`. `gprolog` has less variables, and also smaller reference chains. The `sim` benchmark is remarkable as it actually exhibits a substantial number of reference chains with three cells. Also, the number of double reference chains is significant. The differences between Yap and XSB for `gprolog` are probably caused by different builtin implementation.

Variable Shunting. Variable shunting is an optimization where we try to reduce the length of reference chains by jumping over a member of the chain. Reference shunting for the WAM is discussed by Sahlin and Carlsson [11]. A major problem with complete variable shunting for the WAM is that it requires an extra step. We

Table 4. Reference Chains after Garbage Collection.

	Yap				XSB			
Programs	0	1	2	3	0	1	2	3
`sim`	1006	867	256	10	921	873	248	10
`boyer`	2	4	0	0	3	5	0	0
`gprolog`	2	407	0	0	382	1075	0	0
`nand`	22	356	0	0	7	363	0	0
`chat`	238	328	26	2	45	350	22	6
`chess`	100	19	0	0	803	105	0	0

next discuss two simple transformations, easy shunting and easy undeffing, that can be implemented with little overhead. The transformations are performed at marking time. We experimented with Yap.

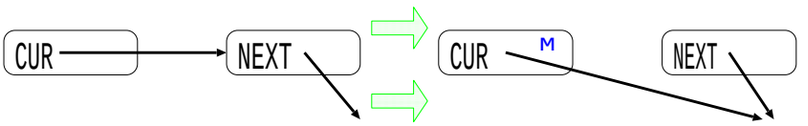


Fig. 1. Easy Shunting

Easy shunting processes two unmarked cells. The first cell, `CUR`, is an unmarked reference. It may be placed in the Heap or in the Local stack. Moreover, `CUR` may be older or younger than the current choice-point. The second variable, `NEXT`, is a reference, a compound term, or a constant, but `NEXT` must be younger than the current choice point. We know that since `NEXT` is unmarked so far, it has not been bound from a more recent choice-point (otherwise it would have been reached already). We also know that it has not been trailed so far (otherwise it would have been reached) and we know the binding is deterministic, as the cell is more recent than the current choice-point. So we know that the binding for `NEXT` is unconditional, and we can replace the value in `CUR` with the value in `NEXT`. A further improvement to easy shunting has been implemented in Yap. We would like to also process chains with marked variables, but shunting is incorrect if the marked variable was bound before the latest choice-point. Yap addresses this case by resetting all Heap variables when marking the trail (they are still marked). Therefore, all current references must have been bound at the current choice-point. We believe our result is close to what Sahlin's algorithm achieves, without the need for an extra step. In fact, Sahlin himself [11] suggests the extra step is only required because SICStus uses pointer reversal, which is not used in Yap.

The second transformation, *easy undeffing*, processes two unmarked cells. The first cell is an unmarked reference in the Heap, which must be more recent than the previous choice point. The second cell is an unbound cell in the heap, and must also be more recent than the current choice-point. The transformation simply moves the unbound variable from `NEXT` to `CUR`, that is, we reset `CUR` and set `NEXT` point at `CUR`

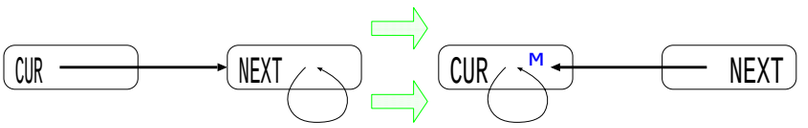


Fig. 2. Easy Undeffing

Easy undeffing is most useful if we never need to mark `NEXT`. In this case, we can actually recover a cell. In fact, our motivation for simple undeffing derives from the Prolog system's parser. When parsing Prolog terms that contain variables, we first create the variable, and then point to it from the Prolog term. There is therefore benefit in undeffing the variable to the term at garbage collection. On the other hand, there is little benefit to this optimization if `NEXT` is part of some compound term and eventually gets marked. An ideal solution would thus be to only undefine a cell if its reference count is zero. This was indeed recently proposed by Demoen [5]. On the other hand, we believe our solution catches a common case and it does have the advantage of not requiring reference counts.

Table 5. Reference Chains after Shunting.

	Chains				**Total**		
	0	1	2	3	**without**	**with**	
`sim`	962	218	33	6	6558	5649	16%
`boyer`	2	4	0	0	21	21	0%
`gprolog`	2	5	0	0	6247	6046	3.3%
`nand`	22	356	0	0	1468	1468	0%
`chat`	265	313	26	2	1850	1819	1.7%
`chess`	100	19	0	0	66798	66798	0%

Table 5 shows the performance of these two simple optimizations. The three benchmarks that benefit are the ones where variables and references count, `sim`, `gprolog`, and `chat`. In all three cases the benefit stems from easy shunting, never from easy undeffing. We believe this is because we find so few unbound variables. The improvements are most impressive in the case of `sim` and of `gprolog` where Heap references almost disappear, generating a 16% space improvement for `sim`.

5.2 Garbage Collection Performance

We have so far concentrated on how well the garbage collector performs on its task, recovering Heap (and maybe Trail) cells. Actual applications, namely deterministic applications, may call the garbage collector very often. Garbage collection time may thus be a substantial component of running time. We would like to reduce this time to a minimum. Traditionally, Prolog systems rely on mark-and-slide garbage collection [3]. Such garbage collectors preserve Prolog segments and variable ordering but they require time proportional to the total amount of Heap. This is extremely inefficient if, for example, only 1% of the cells have been marked, as is the case in `boyer`. In contrast, copying garbage collectors require time proportional to the amount of live data, but can only use half of total stack space and make it harder to efficiently support choice-points.

We propose a technique that takes time proportional to the original Heap. We name it *indirect sliding*. The idea is that we use an extra, intermediate area

to store pointers to every marked cell. We then sort these pointers and use them to perform sliding. Our algorithm works as follows:

1. In the marking phase, whenever we mark a new cell, we also store a pointer to the cell in an extra data area, say $\mathcal{H}^*$;
2. if the number of pointers in $\mathcal{H}^*$ is below some threshold, then we sort them, otherwise we discard them and perform traditional sliding;
3. sweep the pointer array, using the fact that if pointer $\mathcal{H}^*[i]$ points to cell $H[j]$, cell $H[j]$ will move to $H[i]$.

XSB uses an extra buffer to store this area. Yap uses the area between Heap and Local stack. We changed Yap so that overflow is detected if the free space between stacks goes below 1/8 of total stack space. While storing indirection cells we first check if $\mathcal{H}^*$ overflowed into local space. If it does, we stop storing pointers and instead commit to standard mark-and-slide collection. Otherwise, we use indirection if less than 10% of the Heap cells were marked. XSB uses a threshold of 20%. Note that one major advantage of indirection is that we can wait until we have enough information to commit to a specific strategy.

To our knowledge, a similar idea was first proposed for the LaTTe Java virtual machine mark and sweep collector [4], where it is called selective sweeping. LaTTe names our pointer area the set of live objects. Their algorithm thus works at object level, not at cell level. This is an advantage of Java, a language where pointers can only point to objects. Sahlin's $O(nlogn)$ algorithm for Prolog works from a similar principle [10], but instead of using an extra area it requires an extra step to find a set of live blocks. We propose indirection for mark and slide collection, but indirection can be also useful for copying-based garbage collectors. The idea is to maintain $\mathcal{H}^*$ as a heap, and always mark first the largest pointer in the heap. Reverse pointers can be accessed through the Trail.

Performance Evaluation for Yap. In order the study the performance of these techniques we compared performance using three deterministic Prolog programs. The results are shown in Table 6. The `emul` benchmark is from Van Roy's benchmark set, `fsa` is van Noord's finite state machine running the test dg5 [15], and `bn_lbl_kernel` is a test run for Angelopoulos and Cussens' Markov Chain MonteCarlo's system [2]. We fix the stack space and force a fixed strategy throughout execution. Both garbage collectors are called the same number of times and collect the same amount of garbage. Times shown are the accumulated time spent on garbage collection during the whole execution of the programs.

The three examples show clear superiority of the indirection algorithm for deterministic benchmarks. The `emul` benchmark has a fixed working set of about 65KB. For the smallest memory configurations efficiency is lower than 80%, and the overhead of the sorting algorithm makes indirection slower. As we grow stack size, the number of garbage collections decreases linearly, but sliding-based garbage collection must still go through an increasing Heap. On the other hand, performance of the indirection garbage collector improves because the working set is constant. Results for indirection could be even better if not for the increase in trail size: for larger configurations indirection actually spends more

Table 6. Performance of Garbage Collection Algorithms for Yap.

	emul		**fsa**		**bn_lbl_kernel**	
heap	slide	indirect	slide	indirect	slide	indirect
512	11.33	11.57	5.75	6.91	0.99	0.57
1024	6.67	5.31	5.19	6.16	0.69	0.20
2048	4.93	2.71	3.32	3.25	0.67	0.16
4096	3.93	1.48	2.13	1.65	0.67	0.13
8192	3.70	1.04	1.60	0.76	0.62	0.11
16384	3.33	0.60	1.30	0.42	0.48	0.13
32768	3.06	0.60	1.07	0.24	0.61	0.11
65536	3.06	0.51	0.97	0.12	0.62	0.10
131072	3.01	0.24	0.96	0.10	N/A	N/A

time cleaning the Trail than the Heap. The story for `bn_lbl_kernel` is similar. Only difference is that this is a small example, so the working set oscillates between 3KB and 80KB and indirection is always better. Again, Trail sweeping dominates for the largest configurations. The `fsa` benchmark is somewhat different because the working set grows in seesaws from as little as 4KB to as much as 300KB. It is a program that can greatly benefit from generational garbage collection [9]. Indirection works badly when we have bad efficiency, so the results are better for sliding at first. As usual, indirection benefits the most from larger stacks.

Performance Evaluation for XSB. The performance evaluation of the garbage collectors in XSB was carried out by analyzing three benchmarks. `emul` is the BAM emulator presented in the previous section. The `iproto` [8] benchmark is an application of the XMC model checking system which relies heavily on the tabling mechanisms of XSB. Finally, the `justifier` builds a justification proof for a property verified by the XMC model checker.

In Table 7 we fix the garbage collection strategy and change the size of the initial size of the Heap. For all benchmarks, the copying collector is faster than the others. One important thing to notice is that the copying collector in XSB is not segment-preserving. This interacts with backtracking, so the amount of data collected may be different than for the other collectors.

For the `emul` benchmark, indirect sliding provides an interesting alternative to copying, when compared to sliding. The `iproto` is an interesting benchmark, in that the collector always marks more than 20% of the original heap. The indirection mechanism, in this case, is never used. Still, the results show that the overhead in creating the pointer buffer is small. The `justifier` benchmark has a behavior similar to `iproto`, except that the last collection is able to collect most of the heap. Indirect sliding is only faster when we start with a heap large enough so that the number of collections is small.

Table 7. Performance of Garbage Collectors for XSB

heap	emul			iproto			justifier		
Heap Size	**copy**	**slide**	**indirect**	**copy**	**slide**	**indirect**	**copy**	**slide**	**indirect**
512	1.68	2.72	3.86	3.85	4.97	4.91	21.52	25.69	28.17
1024	0.78	1.49	1.60	3.51	4.72	4.80	21.10	26.25	26.78
2048	0.34	1.00	0.80	3.14	4.23	4.26	20.98	25.94	26.80
4096	0.20	0.71	0.41	1.03	1.31	1.25	21.13	26.20	24.94
8192	0.11	0.57	0.18	0.18	0.27	0.33	4.58	5.90	6.46
16384	0.04	0.48	0.10	n/a	n/a	n/a	1.81	2.61	4.60
32768	0.02	0.49	0.04	n/a	n/a	n/a	0.69	1.33	1.91
65536	0.01	0.44	0.02	n/a	n/a	n/a	0.26	0.77	0.64
131072	0.01	0.41	0.02	n/a	n/a	n/a	0.03	0.97	0.03

6 Conclusions

We present what we believe is the first systematic and comparative analysis of memory allocation in two Prolog systems, XSB and Yap. Both systems are based on the same underlying abstract machine, the WAM, and we would expect similar memory usage. In fact, we found significant differences. One difference we expected is that Yap uses much more Trail than XSB. On the other hand, we were surprised that what was considered a relatively minor issue, allocation of temporary variables in the body of clauses, could have such a major impact on several different benchmarks. There is no always-best solution: environment allocation is the best solution for deterministic tail-recursive programs, but is worse than Heap allocation for all other programs. Fortunately, this problem can often be addressed by not initializing output arguments to built-ins. Built-in implementation is indeed a determinant factor in memory usage. Built-ins may produce the correct results and leave garbage or, even worse, unnecessary choice-points in the stacks. In the worst case this will compromise last call optimization and kill application performance for large queries. Comparative analysis, as we did for Yap and XSB, is a good way to clarify these issues.

Both systems performed similarly in garbage collection except for `boyer`, where XSB suffered significantly from not implementing variable trimming. Again an arguably minor issue showed itself to have a huge impact on system performance. Note that garbage collection does not always work for Prolog, as shown by `chess`. Our study also showed a significant interaction between cut and garbage collection. We also studied two simple alternatives to the variable shunting techniques that have been proposed for SICStus Prolog. Although few programs do benefit from them, we found programs that do, and we show that non pointer reversal-based systems can implement shunting with good results for a rather small overhead. Last, we presented indirect sliding, a solution that like copying takes time dependent on the live set, and that for a small overhead gives us the freedom to choose the best method just before we scan the Heap. Our results show significant improvements.

Our work has resulted in several improvements to both Yap's and XSB's memory management systems. We would like to improve these systems even further. One problem we found is that it is very hard to detect why two systems are consuming different amounts of memory. We would like to build better tools towards facilitating the understanding of these issues. Moreover, we have so far concentrated on Prolog, but constraint and tabling programs deserve to be studied. Namely, the results we obtained for `justifier` showed very interesting properties that require further analysis. We would also like to study how generations, as implemented in SICStus Prolog and ECLiPSe, can further improve garbage collector performance, and whether copying can indeed also benefit from indirection. Ultimately, we hope that our contributions towards a systematic approach to memory management will result in more robust logic programming systems that can support a wider number of applications well.

Acknowledgements

Our work in this paper has benefitted from discussions with Bart Demoen, Kostis Sagonas, and Paul Tarau. Vïtor Santos Costa's work was partially supported by the projects CLoP (CNPq), PLAG (FAPERJ), and by the Fundação para a Ciência e Tecnologia in Portugal through LIACC. Luïs Castro was partially supported by NSF Grant EIA-9705998. We would like to thank the anonymous referees for their detailed comments. Last, but not least, we would like to thank the XSB and the Yap user community for motivating this work.

References

1. J. Andersson, S. Andersson, K. Boortz, M. Carlsson, H. Nilsson, T. Sjoland, and J. Widén. SICStus Prolog User's Manual. Technical report, SICS, November 1997. SICS Technical Report T93-01.
2. N. Angelopoulos and J. Cussens. Markov chain monte carlo using tree-based priors on model structure. In *Proceedings of the 17th Annual Conference on Uncertainty in AI (UAI)*, Seattle, USA, 2001.
3. K. Appleby, M. Carlsson, S. Haridi, and D. Sahlin. Garbage collection for Prolog based on WAM. *Communications of the ACM*, 31(6):719–741, 1988.
4. Y. C. Chung, S.-M. Moon, K. Ebcioğlu, and D. Sahlin. Reducing sweep time for a nearly empty heap. In *Symposium on Principles of Programming Languages (POPL'00)*, pages 378–389, 2000.
5. B. Demoen. Early reset and reference counting improve variable shunting in the wam. Technical Report CW Report 298, Katholieke Universiteit Leuven, 2000.
6. B. Demoen and P.-L. Nguyen. So many WAM variations, so little time. In *Computational Logic 2000*, number 1861 in Lecture Notes in Artificial Intelligence, pages 1240–1254. Springer Verlag, 2000.
7. B. Demoen and K. F. Sagonas. Heap Garbage Collection in XSB: Practice and Experience. In *PADL*, number 1753, pages 93–108, January 2000.
8. Y. Dong, X. Du, Y. Ramakrishnan, C. Ramakrishnan, I. Ramakrishnan, S. A. Smolka, O. Sokolsky, E. W. Stark, and D. S. Warren. Fighting livelock in the i-Protocol: A comparative study of verification tools. In *TACAS'99*, March 1999.

9. R. Jones and R. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley and Sons, July 1996. Reprinted February 1997.
10. D. Sahlin. Making garbage collection independent of the amount of garbage. Research Report SICS/R–87/87008, SICS, 1987.
11. D. Sahlin and M. Carlsson. Variable shunting for the WAM. TR R91-07, SICS, Apr. 1991.
12. The XSB Group. The XSB programmer's manual, Version 2.3.
13. E. Tick. *Memory Performance of Prolog Architectures*. Kluwer Academic Publishers, Boston, 1988.
14. H. Touati and A. Despain. An empirical study of the Warren Abstract Machine. In *Fifth ICLP*, pages 114–124, San Francisco, August - September 1987. IEEE, Computer Society Press.
15. van Noord. FSA utilities: A toolbox to manipulate finite-state automata. In *WIA: International Workshop on Implementing Automata, LNCS*. Springer-Verlag, 1997.
16. D. H. D. Warren. *Applied Logic—Its Use and Implementation as a Programming Tool*. PhD thesis, Edinburgh University, 1977.
17. D. H. D. Warren. An abstract Prolog instruction set. Technical Report 309, SRI, 1983.

PALS: An Or-Parallel Implementation of Prolog on Beowulf Architectures

K. Villaverde[1], E. Pontelli[1], H. Guo[3], and G. Gupta[2]

[1] Dept. Computer Science, New Mexico State University
{kvillave,epontell}@cs.nmsu.edu
[2] Dept. Computer Science, Univ. Texas at Dallas
gupta@utdallas.edu
[3] Dept. Computer Science, SUNY Stony Brook

Abstract. This paper describes the development of the *PALS* system, an implementation of Prolog that efficiently exploits or-parallelism on *share-nothing* platforms. PALS makes use of a novel technique, called *incremental stack-splitting*. The technique builds on the stack-splitting approach, which in turn is an evolution of the stack-copying method used in a variety of parallel logic systems. This is the first distributed implementation based on the stack-splitting method ever realized. Experimental results obtained on a Beowulf system are presented and analyzed.

1 Introduction

Or-parallelism (OP) arises from the non-determinism implicit in the process of reducing a given subgoal using different clauses of the program. The non-deterministic structure of a logic programming execution is commonly depicted in the form of a *search tree* (a.k.a. *or-tree*). Each internal node represents a choice-point, i.e., an execution point where multiple clauses are available to reduce the selected subgoal. Leaves of the tree represent either failure points (i.e., resolvents where the selected subgoal does not have a matching clause) or success points (i.e., solutions to the initial goal). A sequential computation boils down to traversal of this search tree according to some predefined search strategy. While a sequential execution attempts to use one clause at the time to reduce each subgoal, eventually using backtracking to explore the use of alternative clauses, OP allows the use of different threads of execution (*computing agents*) to concurrently explore distinct alternatives emanating from a choice-point. If an unexplored branch (i.e., an untried clause to resolve a selected subgoal) is found, the agent picks it up and begins execution. This agent will stop either if it fails (reaches a failing leaf), or if it finds a solution. In case of failure, or if the solution found is not acceptable to the user, the agent will *backtrack*, i.e., move back up in the tree, looking for other choice-points with untried alternatives to explore. The agents may need to synchronize if they access the same node in the tree. Intuitively, OP allows the concurrent search of alternative solutions to the original goal. The importance of the research on efficient techniques for handling OP arises from the generality of the problem—technology originally developed

P. Codognet (Ed.): ICLP 2001, LNCS 2237, pp. 27–42, 2001.

for parallel execution of Prolog has found application in areas such as constraint programming (e.g., [17,13]) and non-monotonic reasoning (e.g., [14]).

Most research on OP execution of Prolog has focused on techniques aimed at shared-memory multiprocessors (SMMs). In this paper we are concerned with the development of execution models for exploitation of OP from Prolog programs on *Distributed Memory Architectures (DMPs)*—i.e., architectures that do not provide any centralized memory resource. The techniques we propose are immediately applicable to other systems based on the same underlying model, e.g., constraint programming [17] and non-monotonic reasoning [14] systems. Other proposals for OP on DMPs have also been recently proposed [8,18,3].

Experimental [1] and theoretical studies [15] have also demonstrated that *stack-copying*, and in particular *incremental* stack-copying, is one of the most effective implementation techniques for exploiting OP that one can devise. Stack-copying allows sharing of work between parallel agents by copying the state of one agent (which owns unexploited tasks) to another agent (which is currently idle). The idea of *incremental* stack-copying is to only copy the *difference* between the state of two agents. Incremental stack-copying has been used to implement or-parallel Prolog efficiently in a variety of systems (e.g., MUSE [1], YAP [16]), as well as to exploit parallelism from constraint systems [17] and non-monotonic reasoning systems [14]. In order to further reduce the communication during stack-copying and make its implementation efficient on share-nothing platforms, a new technique, called stack-splitting, has recently been proposed [11]. In this paper, we describe the first ever concrete implementation of stack-splitting on a DMP platform—specifically a Pentium-based Beowulf—along with a novel scheme to combine incremental copying with stack-splitting on DMPs. The *incremental stack-splitting* scheme is based on a procedure which labels choice-points and then compares the labels to determine the fragments of memory areas that need to be exchanged between agents. We also describe a scheduling scheme which is suitable to be used with this novel incremental stack-splitting scheme. Both the incremental stack-splitting and the scheduling schemes described have been implemented in the *PALS* system, a message-passing OP implementation of Prolog. In this paper we present performance results obtained from this implementation. To our knowledge, PALS is the first OP implementation of Prolog on a Beowulf architecture (built from off-the-shelf components).

2 Stack-Splitting

Relatively few efforts [18,9,3,8,7,6] have been devoted to implementing logic programming systems on DMPs. Some of the older proposals (e.g., [7,6]) relied on variations of stack-copying, while the most recent proposals (e.g., [8,18]) make use of alternative schemes. Out of these efforts only a small number have been implemented as working prototypes, and even fewer have produced acceptable speed-ups. Existing techniques developed for SMMs are mostly inadequate for the needs of DMPs. Most implementation methods require sharing of data

and/or control stacks to work correctly. Even if the need to share data stacks is eliminated—as in *stack-copying*—the need to share the control stack still exists.

2.1 The Need for a Different Stack-Copying Model

Traditional stack-copying relies on idle agents copying data structures from busy agents in order to obtain new tasks. In traditional stack-copying, as implemented in MUSE, backtracking on a choice-point which has been shared between two or more agents, requires acquiring exclusive access to the corresponding *shared frame*. Shared frames are associated to each copied choice-point and used to maintain a shared representation of the alternatives available in such choice-point. The use of shared frames with mutually exclusive access guarantees that no two agents explore the same alternative. This solution works well on SMMs—where mutual exclusion is implemented using *locks*. However, on a DMP this process is a source of overhead, since the shared area becomes a bottleneck [4].

Nevertheless, stack-copying has been recognized as one of the best representation methodologies to support OP in a DMP setting [9,3,7,6]. This is because, while the choice-points are shared (through the shared frames), at least all the other data-structures, such as the environment, the trail, and the heap are not. Other environment representation schemes proposed for OP require more extensive sharing of data structures and seem less suitable to support execution on DMPs (although some recent efforts for adapting the binding array scheme to DMPs—through the use of distributed shared-memory—have been studied [18,8]). To avoid the problem of sharing choice-points in distributed implementations, many developers have reverted back to the *scheduling on top-most choice-point* strategy [3,6,9]. This methodology transfers between agents only the highest choice-point (i.e., closer to the root) in the computation or-tree which contains unexplored alternatives. The reasoning is that untried alternatives of a choice-point created higher up in the or-tree are more likely to generate large subtrees as well as minimize the amount of computation "shared" by different agents. Furthermore, this is guaranteed to be the only choice-point with unexplored alternatives shared between agents. However, if the granularity of the branches in the top-most choice-points does not turn out to be large, then another untried alternative has to be picked and a new copying operation performed. In contrast, in *scheduling on bottom-most choice-point* more work can be found via backtracking, since more choice-points are copied during the same sharing operation. Scheduling on bottom-most choice-point is characterized by the fact that all the choice-points owned by one agent are copied during a sharing operation. Additionally, scheduling on bottom-most is closer to the depth-first search strategy used by sequential systems, and facilitates support of Prolog semantics. Research done on comparing scheduling strategies indicates that scheduling on bottom-most is superior to scheduling on top-most [5]. This is especially true for stack-copying because: (i) the number of copying operations is minimized; and, (ii) the alternatives in the choice-points copied are "cheap" sources of additional work, available via backtracking. However, the shared nature of choice-points is a major drawback for stack-copying on DMPs.

2.2 Stack-Splitting Copying Model

In the stack-copying approach, the primary reason why a choice-point has to be shared is because we want to serialize the selection of untried alternatives, so that no two agents can pick the same alternative. The shared frame is locked while the alternative is selected to achieve this effect. However, there are other simple ways of ensuring the same property: perform a splitting of the choice-points, i.e., each agent is given all the alternatives of alternate choice-points (See Fig. 1). In this case, the list of choice-points is split between the two agents. We call this operation *choice-point stack-splitting* or simply *stack-splitting*.

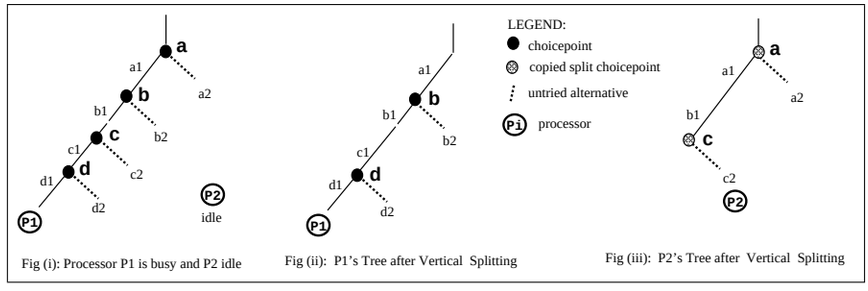


Fig. 1. Splitting of Choice-points

Stack-splitting will ensure that no two agents pick the same alternative. The need for a shared frame, as a critical section to protect the alternatives from multiple executions, has disappeared, as each stack copy has a different choice-point. All the choice-points can be evenly split in this way during the copying operation. The major advantage of stack-splitting is that scheduling on bottom-most can still be used without incurring huge communication overheads. Essentially, after splitting, the different or-parallel threads become independent of each other, and hence communication is minimized during execution. This makes the stack-splitting technique highly suitable for DMPs. Observe that alternative splitting strategies may also be designed—e.g., dividing the alternatives within each choice-point between the two agents [11].

The shared frames in the stack-copying technique are used to maintain global information related to scheduling. The shared frames provide a global description of the or-tree, and each shared frame records which agent is working in which part of the tree. This last piece of information is needed to support scheduling in stack-copying systems—work is taken from the agent that is "closer" in the or-tree, thus reducing the amount of information to be copied. The shared frames ensure accessibility of this information to all agents, providing a consistent view of the computation. However, under stack-splitting the shared frames no longer exist; scheduling and work-load information will have to be maintained in some other way. They could be kept in a global shared area, as in the case of SMMs—e.g., by building a representation of the or-tree—or distributed over multiple

agents and accessed by message passing in case of DMPs. Shared frames are also employed in MUSE [1] to detect the Prolog order of choice-points, needed to execute order-sensitive predicates (e.g., side-effects) in the correct order. As in the case of scheduling, some information regarding global ordering of choice-points needs to be maintained to execute order-sensitive predicates in the correct order. In this paper however we do not handle side-effects and order sensitive predicates. Thus, stack-splitting does not completely remove the need of a shared description of the or-tree. On the other hand, the use of stack-splitting mitigates the impact of accessing shared resources—e.g., stack-splitting allows scheduling on bottom-most which reduces the number of calls to the scheduler.

Stack-splitting has the potential to improve locality of computation, reduce communication between agents, and improve cache behavior. Indeed, the SMM implementation of stack-splitting described in [11] achieves on many benchmarks better speedups than traditional stack copying. The ability to reuse the same technology on both SMMs and DMPs is also a key to development of Prolog systems on *Clusters of SMMs*, i.e., distributed systems with SMMs as nodes.

2.3 Incremental Stack-Copying

Traditional stack-copying requires agents which share work to transfer a complete copy of the data structures representing the status of the computation. In the case of a Prolog computation, this may include transferring most of the choice-points along with copies of the other data areas (trail, heap, environments). Since Prolog computations can make use of large amounts of memory, this copying operation can become quite expensive. Existing stack-copying systems (e.g., MUSE) have introduced a variation of stack-copying, called *Incremental Stack-Copying* [1] which allows to considerably reduce the amount of data transferred during a sharing operation. The idea is to transfer only the difference between the data areas of the two agents. Incremental stack-copying, in a shared-memory context, is relatively simple to realize—the shared frames can be used to identify which choice-points are in common and which are not [1].

In the rest of the paper we describe a complete implementation of stack-splitting on a DMP platform, analyzing in detail how the various problems mentioned earlier have been tackled. In addition to the basic stack-splitting scheme, we analyze how stack-splitting can be extended to incorporate *incremental copying*, an optimization which has been deemed essential to achieve speed-ups in various classes of benchmarks. The solution we describe has been developed in a concrete implementation, realized by modifying the engine of a commercial Prolog system (ALS Prolog) and making use of MPI as communication platform. The ALS Prolog engine is based on the Warren Abstract Machine (WAM).

3 Incremental Stack-Splitting

During stack-splitting, all WAM data areas, except for the code area, are copied from the agent giving work to the idle one. Next, the parallel choice-points

are split between the two agents. Blindly copying all the stacks every time an agent shares work with another idle agent can be wasteful, since frequently the two agents already have parts of the stacks in common due to previous copying. We can take advantage of this fact to reduce the amount of copying by performing *incremental copying*. In order to figure out the incremental part that only needs to be copied during incremental stack-splitting, parallel choice-points will be *labeled*. The goal of the labeling process is to uniquely identify the original "source" of each choice-point (i.e., which agent created it), to allow unambiguous detection of copies of common choice-points.

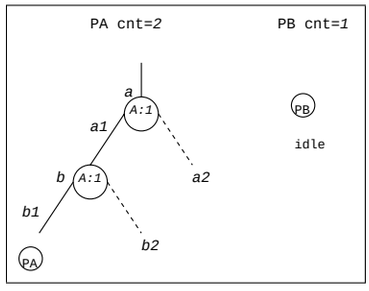


Fig. 2. A Labels its Choice-points

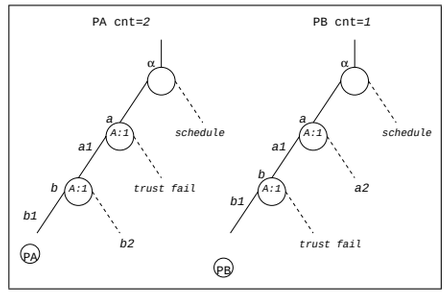


Fig. 3. A Gave Work to B

To perform labeling, each agent maintains a counter. The counter is increased by 1 every time the labeling procedure is performed. When a parallel choice-point is copied for the first time, a label for it is created. The label is composed of three parts: *(1)* agent rank, *(2)* counter, and *(3)* choice-point address. The agent rank is the rank (i.e., id) of the agent which created the choice-point. The counter is the current value of the labeling counter for the agent generating the labels. The choice-point address is the address of the choice-point which is being labeled. The labels for the parallel choice-points are recorded in a separate *label stack*, in the order they are created. Also, when a parallel choice-point is removed from the stack, its corresponding label is also removed from the label stack (this is actually integrated with the variable untrailing mechanism). Initially, the label stack in each agent is set to *empty*. Intuitively, the label stack keeps a record of changes done to the stacks since the last stack-splitting operation. Let us illustrate the stack-splitting accompanied by labeling with an example. Suppose process A has just created two parallel choice-points and process B is idle. Process A first creates labels for its two parallel choice-points. These labels have their rank and counter parts as *A:1*. Process A pushes these labels into its label stack (Fig. 2).

Process B gets all the parallel choice-points of process A along with process A label stack. Then, stack-splitting takes place: process A will keep the alternative *b2* but not *a2*, and process B will keep the alternative *a2* but not *b2*. We have designed a new WAM scheduling instruction which is placed in the next alternative field of the choice-point above which there is no more parallel work. This

scheduling instruction implements the scheduling scheme described in Section 4. To avoid taking the original alternative of a choice-point, we change its next alternative field to WAM instruction *trust_fail*. See Fig. 3. Afterwards, process B backtracks, removes choice-point *b* along with its corresponding label in the label stack, and then takes alternative *a2* of choice-point *a*.

3.1 Incremental Stack-Splitting: The Procedure

Assume process W is giving work to process I. Process W will label all its parallel choice-points which have not been labeled before and will push them into its label stack. If process I label stack is empty, then non-incremental stack-copying will need to be performed followed by stack-splitting. Process W sends its complete choice-point stack and its complete label stack to process I. Then stack-splitting is performed on all the parallel choice-points of process W. However, if process I label stack is not empty then process I sends its label stack to process W. Process W compares its label stack against the label stack from I. The objective is to find the last choice-point *ch* with a common label. In this way, processes W and I are guaranteed to have the same computation *above* the choice-point *ch*, while their computations will be different below such choice-point.

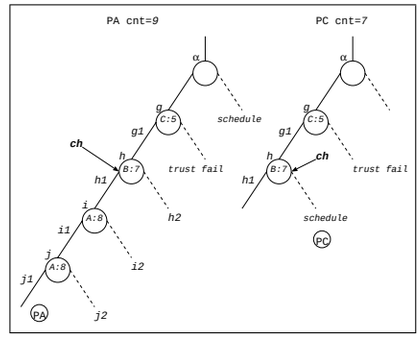


Fig. 4. Labels Comparison

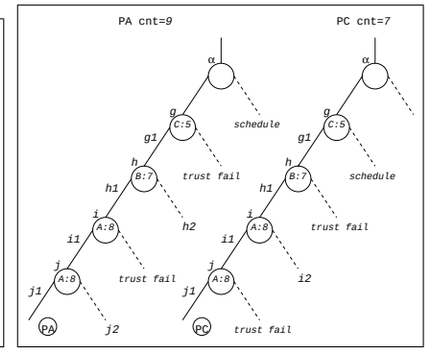


Fig. 5. Proc. A Gave Work to Proc. C

If the choice-point *ch* does not exist, then non-incremental stack-copying will need to be performed followed by stack-splitting, as described before. However, if choice-point *ch* does exist, then process I backtracks to choice-point *ch*, and performs incremental-copying. Process W sends its choice-point stack starting from choice-point *ch* to the top of its choice-point stack. Process W also sends its label stack starting from the label corresponding to *ch* to the top of its label stack. Stack-splitting is then performed on all the parallel choice-points of W.

We illustrate the above procedure by the following example. Suppose process A has three parallel choice-points and process C requests work from A. Process A first labels its last two parallel choice-points which have not been labeled before

and then increments its counter. Afterwards, process C sends its label stack to process A. Process A compares its label stack against the label stack of process C and finds the last choice-point *ch* with a common label. Above choice-point *ch*, the Prolog trees of processes A and C are equal. See Fig. 4. Now, process C backtracks to choice-point *ch*. Incremental stack-copying can then take place. Process A sends its choice-point stack starting from choice-point *ch* to the top of its choice-point stack, and stack-splitting is performed (Fig. 5).

3.2 Incremental Stack-Splitting: Challenges

Sequential Choice-points: The first issue has to do with sequential choice-points that are located among the parallel choice-points shared by two agents. If the alternatives of these choice-points are kept in both processes, we may have repeated or wrong computations. Hence, the alternatives of these choice-points should only be kept in one process (e.g., the one giving work). If the alternatives are kept in the process giving work, then the process that is receiving work should change the next alternative field of these choice-points to the instruction *trust_fail* to avoid taking the original alternatives of these choice-points.
Installation Process: The second issue has to do with the bindings of conditional variables (i.e., variables that may be bound differently in different or-parallel branches) which may not be copied during the incremental splitting process. This can be fixed by having the process giving work create a stack of all these conditional variables along with their bindings. This stack will then be sent to the process receiving work so that it can update the bindings.
Garbage Collection: When garbage collection takes place, relocation of choice-points may also occur. Hence, the labels in our label stack may no longer label the correct parallel choice-points. Therefore, we need to modify our labeling procedure so that when garbage collection on an agent takes place, the label stack of this agent is invalidated. The next time this process gives work, non-incremental stack-copying will have to take place. This solution is analogous to the one adopted in the original implementation of the MUSE system [1].

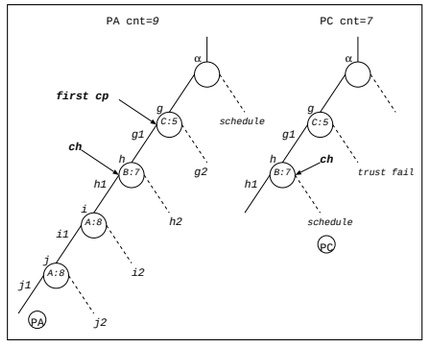


Fig. 6. Copy Nextclause from *first cp* to *ch*

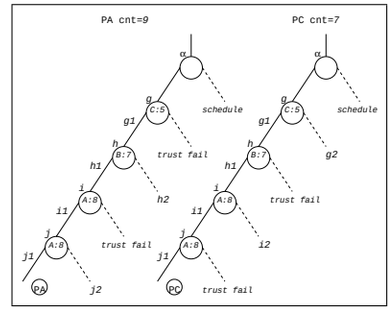


Fig. 7. C Received Next-Clause Fields

Next Clause Fields: The fourth issue arises when the next clause fields of the parallel choice-points between the first parallel choice-point *first cp* and the last choice-point *ch* with a common label in the agent giving work are not the same compared to the ones in the agent receiving work. This situation occurs after several copying and splitting operations. In this case, we cannot just copy the part of the choice-point stack between choice-point *ch* and the top of the stack and then perform the splitting. This is because the splitting will not be performed correctly. For example, suppose that in our previous example when process C requests work from process A, we have this situation (Fig. 6). We can see that choice-point *g* should be given to process C. But process C does not have the right next clause field for this choice-point. The problem can be solved by having the process giving work send all the next clause fields between its first choice-point *first cp* and choice-point *ch* to the process receiving work. Then the splitting of all parallel choice-points can take place correctly. See Fig. 7.

4 Scheduling

The main objective of a scheduling strategy is to balance the amount of parallel work done by different agents. Additionally, work distribution among agents should be done with minimal communication overhead. These two goals are somewhat at odds with each other, since achieving perfect balance may result in a very complex scheduling strategy with considerable communication overhead, while a simple scheduling strategy which re-distributes work less often will incur low communication overhead but poor balancing of work. Therefore, it is obvious that there is an intrinsic contradiction between distributing parallel work as even as possible and minimizing the distribution overhead. Thus our main goal is to find a trade-off point that results in a reasonable scheduling strategy.

We adopt a simple distributed algorithm to implement a scheduling strategy in PALS. A data structure—the *load vector*—is introduced to indicate the work loads of different agents. The work load of an agent is approximated by the number of parallel choice-points present in its local computation tree. Each agent keeps a work load vector V in its local memory, and the value of $V[i]$ represents the work load of the agent with rank i. Based on the work load vector, an idle agent can request parallel work from other agent with the greatest work load, so that parallel work can be fairly distributed. The load vector is updated at runtime. When stack-splitting is performed, a `Load_Info` message with updated load information will be broadcasted to all the agents so that each agent has the latest information of work load distribution. Additionally, load information is attached with each incoming message. For example: when a `Request_Work` message is received from agent P_1, the value of P_1's work load, 0, can be inferred.

Based on its work load each agent can be in one of two states: *scheduling* state or *running* state. An agent that is running, occasionally checks whether there are incoming messages. Two possible types of messages are checked by the running agent: one is `Request_Work` message sent by an idle agent, and the other is `Send_Load_Info` message, which is sent when stack-splitting occurs. The idle agent in scheduling state is also called a scheduling agent.

The distributed scheduling algorithm mainly consists of two parts: one is for the scheduling agent, and the other is for the running agent. An idle agent wants to get work as soon as possible from another agent, preferably the one that has the largest amount of work. The scheduling agent searches through its local load vector for the agent with the greatest work load, and then sends a `Request_Work` message to that agent asking for work. If all the other agents have no work, then the execution of the current query is finished and the agent halts. When a running agent receives a `Request_Work` message, stack-splitting will be performed if the running agent's work load is greater than the splitting threshold, otherwise, a `Reply_Without_Work` message with a positive work load value will be sent as a reply. If a scheduling agent receives a `Request_Work` message, a `Reply_Without_Work` message with work load 0 will be sent as a reply. The running agent's algorithm can be briefly described as follows: each incoming message can be either a `Send_LoadInfo` message—i.e., a notification of a change in load for some processors—or a `Request_Work` message—i.e., a request for sharing, which is accepted if the local load is above a given threshold. At fixed time intervals (which can be selected at initialization of the system) the agent examines the content of its message queue for eventual pending messages. `Send_LoadInfo` messages are quickly processed to update the local view of the overall load in the system. Messages of the type `Request_Work` are handled as described above. Observe that the concrete implementation actually checks for the presence of the two types of messages with different frequency (i.e., request for work messages are considered less frequently than requests for load update).

5 Implementation and Performance

Stack-Splitting: The stack-splitting procedure has been implemented by modifying the commercial ALS Prolog system, using the MPI library for message passing. The only major data structures added to the ALS system are: the label stack, the load vector, and buffers in order transfer information. The whole system runs on a truly distributed machine (a network of 32 Pentium II nodes connected by Myrinet-SAN Switches). All communication—during scheduling, copying, splitting, etc.— is done using explicit message passing via MPI.

The benchmarks used to test our system are standard benchmarks drawn from the pool of programs frequently used to evaluate OP systems (e.g., *Queens*, *Knight*, *Solitaire*). The benchmarks selected are simple but provide sufficiently different program structures to validate the parallel engine. The timing results in seconds from our incremental stack-splitting system are presented in Table 1. The modifications made to the ALS WAM are very localized and reduced to the minimum. This has allowed us to keep a clean design—that can be easily ported to other WAM-based implementations—and to contain the parallel overhead—our engine on a single processor is on average 5% slower than ALS WAM. The corresponding speed-ups are presented in Fig. 8 (with label *incremental*).

Note that for benchmarks with substantial running time the speed-ups are quite good, while for programs with smaller running time the speed-ups deteri-

Table 1. Timings for Incremental Stack-Splitting (Time in sec.)

Benchmark	**# Processors**					
	1	**2**	**4**	**8**	**16**	**32**
Knight	159.950	81.615	40.929	20.754	10.939	8.248
Send More	61.817	32.953	17.317	8.931	4.923	3.916
8 Puzzle	27.810	15.387	8.442	10.522	3.128	5.940
Solitaire	5.909	3.538	1.811	1.003	0.628	0.535
10 Queens	4.572	2.418	1.380	0.821	1.043	0.905
Hamilton	3.175	1.807	0.952	0.610	0.458	0.486
Map Coloring	1.113	0.702	0.430	0.319	0.318	0.348
8 Queens	0.185	0.162	0.166	0.208	0.169	0.180

orate. This is consistent with our belief that DMP implementations should be used for parallelizing programs with coarse-grained parallelism. For programs with small running times, there is not enough work to offset the communication costs on DMPs. Nevertheless, our system is reasonably efficient, given that even for small benchmarks it can produce speed-ups. It is also interesting to observe that in no cases we have observed slow-downs due to parallel execution—thanks to simple granularity control mechanisms embedded in the scheduler. For some benchmarks the speedup graphs are somewhat irregular – specially the *8 Puzzle*. We believe that the reason behind this hides in the scheduling strategy used.

One of the objectives of the experiments performed is to validate the effectiveness of incremental stack-splitting for efficient exploitation of parallelism on DMPs. In particular, there are two aspects that we were interested in exploring: *(i)* verifying the effectiveness of stack-splitting versus a more "direct" implementation of stack-copying (i.e., keeping single copies of choice-points around the system); *(ii)* verifying the impact of *incremental* splitting. Validity of stack-splitting vs. stack-copying can be inferred from the experiments described in the next subsection: a direct implementation of stack-copying would produce the same amount of communication traffic as some of the variations of scheduling tested, and thus incur the same kind of problems described next. In order to evaluate the impact of incrementality, we have measured the performance of the system on the selected benchmarks without the use of incremental splitting—i.e., each time a sharing operation takes place, a complete copy of the data areas is performed. The results obtained from this experiment are in Fig. 8: the figure compares the speed-ups observed with and without incremental copying. We can observe that incremental stack-splitting obtains higher speed-ups than the non-incremental stack-copying. The difference is more significant in benchmarks with a large number of choice-points, where incrementality is applied more frequently.

Scheduling: One of the major reasons to adopt stack-splitting is the ability to perform scheduling on bottom-most choice-point. Other DMP implementations of OP have resorted to scheduling on the top-most choice-point, where only

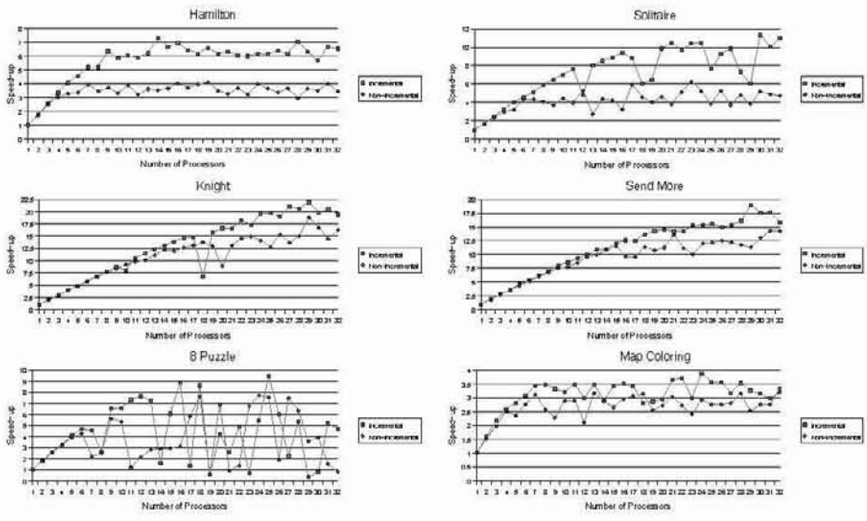


Fig. 8. Incremental Stack-Splitting vs. Non-Incremental Stack-Splitting

the oldest choice-point with unexplored alternatives is exchanged between processors. Top-most scheduling will share only one choice-point at the time, thus relieving the engine from the need of controlling access to shared choice-points.

To validate the effectiveness of our claim, we have developed a top-most scheduler for our system and compared its performance with that of the incremental stack-splitting with bottom-most scheduling. Fig. 9 compares the speed-ups observed using the two different schedulers. In the figure we have reported the behavior only of those benchmarks where significant differences in performance have been recorded. In all other benchmarks, top-most and bottom-most scheduling provide similar results, as a small number of choice-points are cre-

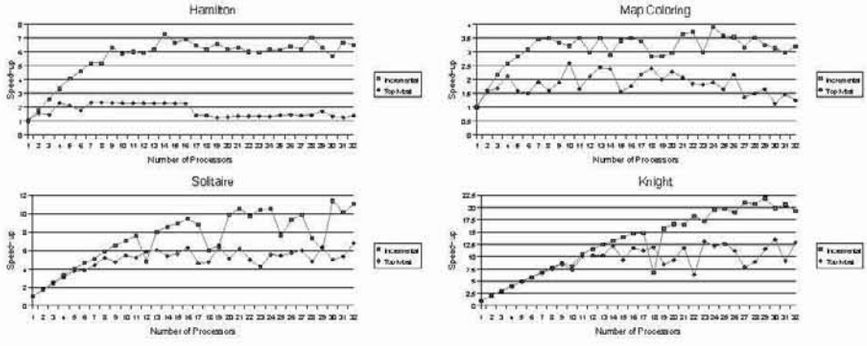


Fig. 9. Incremental Stack-Splitting vs. Top Most Scheduling

ated and only one at a time is shared between processors. As we can observe from Fig. 9, bottom-most scheduling provides a sustained speed-up considerably higher than top-most scheduling. This is due to the reduced number of calls to the scheduler performed during the execution—processors spend a higher fraction of their time doing useful work compared to scheduling on top-most.

Another aspect of our implementation that we are interested in validating is the performance of the distributed scheduler. As mentioned in Sect. 4, our scheduler is based on keeping in each processor an "approximated" view of the load in each other processor. The risk that this method may encounter is that a processor may have out-of-date information concerning the load in other processors, and as a consequence it may try to request work from idle processors or ignore processors that may have unexplored alternatives. Fig. 10 provides some information concerning the number of attempts that a processor needs to perform before receiving work. The figure on the left measures the average number of requests that a processor has to send; as we can see, the number is very small (1 or 2 requests are typically sufficient) and such number is generally better if we adopt bottom-most scheduling. The figure on the right shows the maximum number of requests observed; these numbers tend to grow towards the end of the computation (when less work is available)—nevertheless, typically only one or two processors achieve these maximum values, while the majority of the processors remain close to the average number of attempts.

To further validate our scheduling approach, we have compared it with an alternative scheduling scheme developed in PALS. This alternative scheme is an implementation of a *centralized* scheduling algorithm, designed following the guidelines of the scheduler used in Opera [7]. In the centralized approach, only one processor, called *central*, is in charge of keeping track of the load information. Idle processors send their requests for work directly to the central processor. In turn, the central processor is in charge of implementing a matchmaking algorithm between idle and busy processors. When stack-splitting occurs, only the central processor is informed about the load information update. Fig. 11 compares the speed-ups achieved using centralized scheduling with the speed-ups observed using the distributed scheduling approach.[1] As evident from the figure, the speed-ups observed in centralized scheduling are almost negligible—this is due to the inability of the scheduling method to promptly respond to the requests for new work. Also, the use of a reasonably fast network (Myrinet) leads to the creation of a severe bottleneck at the level of the centralized scheduler.

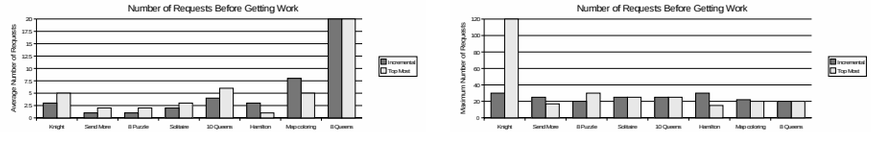


Fig. 10. Average and Maximum Number of Tries to Acquire Work

[1] We had to limit the experiments to a smaller number of CPUs due to unavailability of half of the machine at that time.

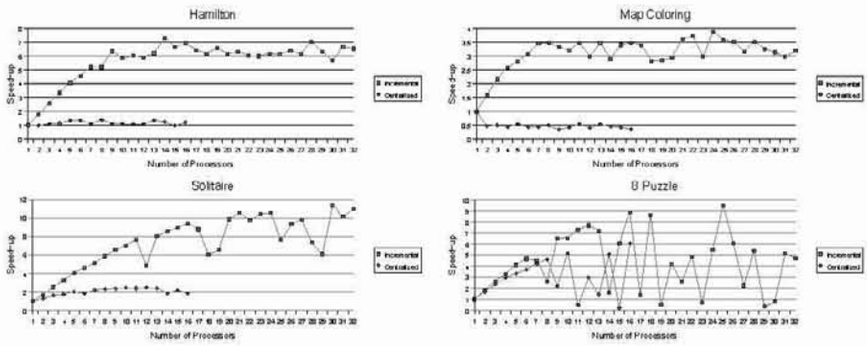


Fig. 11. Incremental Stack-Splitting vs. Centralized Scheduling

The results presented in [5] suggest that random selection of work may also provide a simple and effective alternative when searching for work. We have experimented with this idea, by modifying the scheduler to select any busy processor for scheduling. The idea is to avoid bottleneck situations where multiple idle processors are concentrating their requests for work towards the same busy processor. We have named this new version of the scheduler *Random Scheduler*. In this version, an idle processor searches its load vector for the next processor with load greater than a given small threshold. Fig. 12 compares the speed-ups observed in the Random scheduler with those from the standard bottom-most scheduling with selection of processor with highest load. The results indicate that the Random scheduler is less effective. This suggests that selecting work from the processor with highest load is not a severe bottleneck and sending requests to lightly loaded processors may increase the number of calls to the scheduler.

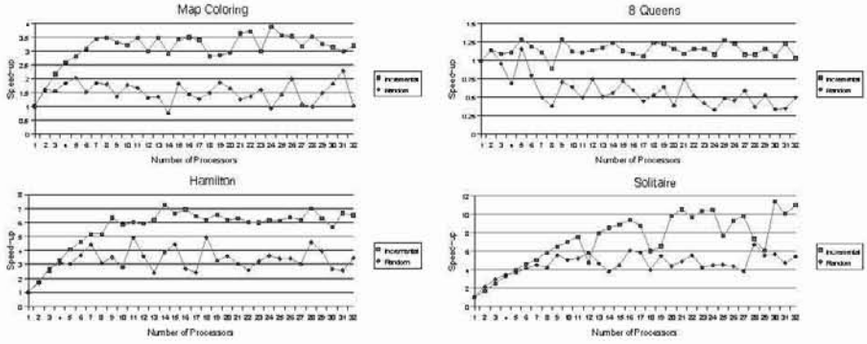


Fig. 12. Incremental Stack-Splitting vs. Random Scheduling

6 Related Work and Conclusions

In this paper we proposed a novel scheme to implement incremental stack-splitting for OP on DMPs. The novel method allows to take advantage of the higher locality and independence of computation threads allowed by stack-splitting, without losing the advantages of incremental copying. The incremental stack-splitting scheme presented is based on a procedure which labels parallel choice-points and then compares the labels to determine the incremental WAM areas to be copied. Furthermore, we described a scheduling strategy for incremental stack-splitting. The incremental stack-splitting scheme and the scheduling strategy have been implemented in the ALS Prolog system, and performance results from this implementation were reported. To our knowledge, PALS is the first ever or-parallel implementation of Prolog on Beowulf systems.

A relatively small number of proposals can be found in the literature dealing with execution of Prolog on DMPs. Some of the existing environment representation models proposed (e.g., Conery's Closed Environments) have been designed with distributed memory in mind, but they have never been concretized in actual implementations. Most of the older systems implemented on DMPs [7,6] are based on stack copying and have been designed with respect to a specialized architecture (Transputers). Their schedulers are tailored for this class of architectures and they all resort to top scheduling to reduce communication costs. PDP [3] makes use of a recomputation approach to deal with OP, and has also been developed on Transputers. MUSE version on switch based multiprocessors [2] (e.g., Butterfly) gives good speedups for very coarse grain applications but uses distributed shared-memory techniques. Only in recent years a renovated effort towards developing models for generic DMP architectures have emerged. These include DAOS [8] and Dorpp [18] based on variations of the binding arrays method and relying on distributed shared-memory technology; DAOS has not reported any implementation result, while Dorpp has been executed on simulators (with fairly good results). In contrast to DAOS and Dorpp, we opted to continue using stack copying with a fully distributed scheduler. For comparison of stack-splitting with other existing approaches see [11].

Acknowledgments: We are grateful for the help received from K. Bowen, C. Houpt, and V. Santos Costa. This work has been partially supported by NSF grants CCR-9875279, CCR-9900320, CDA-9729848, EIA-9810732, CCR-9820852, and HRD-9906130, and by a fellowship from the Dept. of Education.

References

1. K.A.M. Ali and R. Karlsson. The Muse Or-parallel Prolog Model and its Performance. In *N. American Conf. on Logic Prog.*, pages 757–776. MIT Press, 1990.
2. K.A.M. Ali, R. Karlsson, and S. Mudambi Performance of Muse on Switch-Based Multiprocessors Machines. *New Generation Computing*, 11(1):81-103, 1992.
3. L. Araujo and J. Ruz. A Parallel Prolog System for Distributed Memory. *J. of Logic Programming*, 33(1):49–79, 1998.
4. H. Babu. Porting MUSE on ipsc860. Master's thesis, NMSU, 1996.

5. A.J. Beaumont and D. H. D. Warren. Scheduling Speculative Work in Or-Parallel Prolog Systems. In *ICLP*, pages 135–149, 1993. MIT Press.
6. V. Benjumea and J.M. Troya. An OR Parallel Prolog Model for Distributed Memory Systems. In *PLILP*, pages 291–301, 1993. Springer Verlag.
7. J. Briat et al. OPERA: Or-Parallel Prolog System on Supernode. In *Implementations of Distributed Prolog*, pages 45–64. J. Wiley & Sons, New York, 1992.
8. L.F. Castro et al. DAOS: Scalable And-Or Parallelism. In *Euro-Par*, pages 899–908, 1999. Springer Verlag.
9. W-K. Foong. *Combining and- and or-parallelism in Logic Programs: a distributed approach*. PhD thesis, University of Melbourne, 1995.
10. G. Gupta and E. Pontelli. Optimization Schemas for Parallel Implementation of Nondeterministic Languages and Systems. In *IPPS*, 1997. IEEE Computer Society.
11. G. Gupta and E. Pontelli. Stack-splitting: A Simple Technique for Implementing Or-parallelism on Distributed Machines. In *ICLP*. MIT Press, pages 290–304, 1999.
12. G. Gupta, E. Pontelli, M. Carlsson, M. Hermenegildo, and K.M. Ali. Parallel Execution of Prolog Programs: a Survey. *ACM TOPLAS*, 2001. (to appear).
13. L. Perron. Search Procedures and Parallelism in Constraint Programming. In *PPDP*, pages 346–360, 1999. Springer Verlag.
14. E. Pontelli and O. El-Kathib. Construction and Optimization of a Parallel Engine for Answer Set Programming. In *PADL*, pages 288–303, 2001. Springer Verlag.
15. D. Ranjan, E. Pontelli, and G. Gupta. On the Complexity of Or-Parallelism. *NGC*, 17(3):285–308, 1999.
16. R. Rocha, F. Silva, and V. Santos Costa. YapOr: an Or-Parallel Prolog System based on Environment Copying. In *EPPIA*, pages 178–193, 1999, Springer Verlag.
17. C. Schulte. Parallel Search Made Simple. In *TRICS*, number TRA9/00, pages 41–57, University of Singapore, 2000.
18. F. Silva and P. Watson. Or-Parallel Prolog on a Distributed Memory Architecture. *Journal of Logic Programming*, 43(2):173–186, 2000.

On a Tabling Engine That Can Exploit Or-Parallelism*

Ricardo Rocha[1], Fernando Silva[1], and Vítor Santos Costa[2]

[1] DCC-FC & LIACC, University of Porto, Portugal.
{ricroc,fds}@ncc.up.pt.
[2] COPPE Systems, University of Rio de Janeiro, Brazil.
vitor@cos.ufrj.br.

Abstract. Tabling is an implementation technique that improves the declarativeness and expressiveness of Prolog by reusing solutions to goals. Quite a few interesting applications of tabling have been developed in the last few years, and several are by nature non-deterministic. This raises the question of whether parallel search techniques can be used to improve the performance of tabled applications.
In this work we demonstrate that the mechanisms proposed to parallelize search in the context of SLD resolution naturally generalize to parallel tabled computations, and that resulting systems can achieve good performance on multi-processors. To do so, we present the OPTYap parallel engine. In our system individual SLG engines communicate data through stack copying. Completion is detected through a novel parallel completion algorithm that builds upon the data structures proposed for or-parallelism. Scheduling is simplified by building on previous research on or-parallelism. We show initial performance results for our implementation. Our best result is for an actual application, model checking, where we obtain linear speedups.

Keywords: Parallel Logic Programming, Or-Parallelism, Tabling.

1 Introduction

The past years have seen wide effort at increasing Prolog's declarativeness and expressiveness. *Tabling* or *memoing* is one such proposal that has been gaining in popularity. In a nutshell, tabling consists of storing intermediate answers for subgoals so that they can be reused when a repeated subgoal appears. Work on SLG resolution [3], as implemented in the XSB System [15], proved the viability of tabling technology for application areas such as natural language processing, knowledge based systems, model checking, or program analysis. Tabling based models are able to reduce the search space, avoid looping, and always terminate for programs with the *bounded term-size property* [4].

* Work partially supported by CLoP (CNPq), PLAG (FAPERJ) and by Fundação para a Ciência e Tecnologia.

P. Codognet (Ed.): ICLP 2001, LNCS 2237, pp. 43–58, 2001.

Tabling works for both deterministic and non-deterministic applications, but it has frequently been used to reduce search space. This rises the question of whether further efficiency improvements may be achievable through parallelism. Freire and colleagues [7] were the first to propose that tabled goals could indeed be a source of implicit parallelism. In their model, each tabled subgoal is computed independently in a separate computational thread, a *generator thread*. Each generator thread is the sole responsible for fully exploiting its subgoal and obtain the complete set of answers. This model restricts parallelism to concurrent execution of generator threads. Parallelism arising from non-tabled subgoals or from alternative clauses is not exploited.

Our suggestion is that we should exploit parallelism from both tabled and non-tabled subgoals. By doing so we can both extract more parallelism, and reuse the mature technology for tabling and parallelism. Towards this goal, we previously proposed two computational models to combine tabling with or-parallelism [12], *Or-Parallelism within Tabling (OPT)* and *Tabling within Or-Parallelism (TOP)* models.

This paper presents an implementation for the OPT model, the OPTYap system. To the best of our knowledge, OPTYap is the first available system that can exploit parallelism from tabled programs. The OPT model considers tabling as the base component of the system. Each computational worker behaves as a full sequential tabling engine. The or-parallel component of the system is triggered to allow synchronized access to the shared part of the search space or to schedule work.

From the beginning, we aimed at developing an or-parallel tabling system that, when executed with a single worker, runs as fast or faster than current sequential tabling systems as otherwise, parallel performance would not be significant and fair. To achieve these goals, OPTYap builds on YapOr [13] and YapTab [14] engines. YapOr is an or-parallel engine that extends Yap's efficient sequential engine [16]. It is based on the environment copy model, as first implemented in Muse [1]. YapTab is a sequential tabling engine that extends Yap's execution model to support tabled evaluation. YapTab's implementation is largely based on the ground-breaking SLG-WAM work used in the XSB system [15].

The remainder of the paper is organized as follows. First, we briefly introduce the basic tabling definitions and the SLG-WAM. Next, we present the OPT computational model and discuss its implementation framework. We then present the new data areas, data structures and algorithms to extend the Yap Prolog system to support sequential and parallel tabling. Last, we present some early performance data and terminate by outlining some conclusions and further work.

2 Tabling and the SLG-WAM

Tabling is about storing and reusing intermediate answers for goals. In variant-based tabling, whenever a tabled subgoal S is called for the first time, an entry for S is allocated in the *table space*. This entry will collect all the answers found for S. Repeated calls to *variants* of S are resolved by consuming the answers already

stored in the table. Meanwhile, as new answers are generated, they are inserted into the table and returned to all variant subgoals. Within this model, the nodes in the search space are classified as either *generator nodes*, corresponding to first calls to tabled subgoals, *consumer nodes*, corresponding to variant calls to tabled subgoals, and *interior nodes*, corresponding to non-tabled predicates.

Tabling based evaluation has four main types of operations for definite programs. The *Tabled Subgoal Call* operation checks if the subgoal is in the table and if not, inserts it and allocates a new generator node. Otherwise, allocates a consumer node and starts consuming the available answers. The *New Answer* operation verifies whether a newly generated answer is already in the table, and if not, inserts it. The *Answer Resolution* operation consumes the next newly found answer, if any. The *Completion* operation determines whether a tabled subgoal is completely evaluated, and if not, schedules a possible resolution to continue the execution.

Space for a subgoal can be reclaimed when the subgoal has been *completely evaluated*. A subgoal is said to be completely evaluated when all its possible resolutions have been performed, that is, when no more answers can be generated and the variant subgoals have consumed all the available answers. Note that a number of subgoals may be mutually dependent, forming a *strongly connected component* (or *SCC*) [15], and therefore can only be completed together. The completion operation is thus performed at the *leader* of the SCC, that is, by the oldest subgoal in the SCC, when all possible resolutions have been made for all subgoals in the SCC. Hence, in order to efficiently evaluate programs one needs an efficient and dynamic detection scheme to determine when all the subgoals in a SCC have been completely evaluated.

The implementation of tabling in XSB Prolog was attained by extending the WAM [17] into the SLG-WAM [15]. In short, the SLG-WAM introduces a new set of instructions to deal with the operations above, a special mechanism to allow suspension and resumption of computations, and two new memory areas: a *table space*, used to save the answers for tabled subgoals; and a *completion stack*, used to detect when a set of subgoals is completely evaluated. The SLG-WAM also introduced the concepts of freeze registers and forward trail to handle suspension [15].

3 Or-Parallelism within Tabling

The OPT model [12] divides the search tree into a public and several private regions, one per worker. Workers in their private region execute nearly as in sequential tabling. Workers exploiting the public region of the search tree must be able to synchronize in order to ensure the correctness of the tabling operations. When a worker runs out of alternatives to exploit, it enters in scheduling mode. The YapOr scheduler is used to search for busy workers with unexploited work. Alternatives should be made available for parallel execution, regardless of whether they originate from generator, consumer or interior nodes.

Parallel execution requires significant changes to the SLG-WAM. Synchronization is required **(i)** when backtracking to public generator or interior nodes to take the next available alternative; **(ii)** when backtracking to public consumer nodes to take the next unconsumed answer; or, **(iii)** when inserting new answers into the table space. In a parallel tabling system, the relative positions of generator and consumer nodes are not as clear as for sequential systems. Hence we need novel algorithms to determine whether a node is a leader node and to determine whether a SCC can be completed.

OPTYap uses environment copying for or-parallelism and the SLG-WAM for tabling because these are, respectively, two of the most successful or-parallel and tabling engines. Copying is a popular and effective approach to or-parallelism that minimizes actual changes to the WAM. To share work we use *incremental copying* [1], that is, we only copy *differences* between stacks.

In contrast to copying, the SLG-WAM requires significant changes to the WAM in order to support freezing of goals. These changes introduce overheads, namely in trailing and in stack manipulation. Demoen and Sagonas addressed the problems by suggesting CAT [5] and more recently, CHAT [6]. These two models reduce overheads by copying parts of stacks, instead of freezing. Although there is an attractive analogy between copying and CAT or CHAT, a more detailed analysis shows significant drawbacks. First, both assume separate choice-point and local stacks. Second, both rely on an incremental saving technique to reduce copying overheads. Unfortunately, the technique assumes that completion always takes place at generator nodes. As we shall see, these assumptions do not hold true for parallel tabling. Last, both may incur in substantial slowdowns for some applications. We therefore used the SLG-WAM in our work.

Rather different approaches to tabling have also been proposed recently [18,9]. In both cases, the main idea is to recompute tabled goals, instead of suspending. Unfortunately, the process of retrying alternatives may cause redundant recomputations of non-tabled subgoals that appear in the body of a looping alternative and redundant consumption of answers if the looping alternative contains more than one variant subgoal call. Parallel recomputation is harder because we do not know beforehand if a tabled alternative needs to be recomputed: a conservative approach may lose parallelism, and an optimistic approach may lead to even more redundant computation.

4 The Sequential Tabling Engine

Next, we review the main principles of the YapTab design (please refer to [14,11] for more details). YapTab implements two tabling scheduling strategies, batched and local [8], and in our initial design it only considers positive programs. Tables are implemented using tries as proposed in [10]. We reconsidered decisions in the original SLG-WAM that can be a potential source of parallel overheads. Namely, YapTab considers that control of leader detection and scheduling of unconsumed answers should be performed through the consumer nodes. Hence, YapTab associates a new data structure, the *dependency frame*, to consumer

nodes. In contrast, the SLG-WAM associates this control with generator nodes. We argue that managing dependencies at the level of the consumer nodes is a more intuitive approach that we can take advantage of.

The introduction of this new data structure allows us to reduce the number of extra fields in tabled choice points and to eliminate the need for a separate completion stack. Furthermore, allocating the data-structure in a separate area simplifies the implementation of parallelism.

To benefit from the philosophy behind the dependency frame data structure, we redesigned the algorithms related with suspension, resumption and completion. We next present YapTab's main data structures and algorithms. We assume a batched scheduling strategy implementation [8] (please refer to [11] for the implementation of local scheduling).

Generator and Consumer Nodes. YapTab implementation stores generator nodes as standard nodes plus a pointer to the corresponding subgoal frame. In contrast to the SLG-WAM, we adjust the freeze registers by using the top of stack values kept in the consumer choice points. YapTab also implements consumer nodes as standard nodes plus a pointer to a *dependency frame*. The dependency frames are linked together to form the *dependency list of consumer nodes*. Additionally, dependency frames store information to efficiently check for completion points, replacing the need for a separate completion stack [15], as we discuss next.

Completion and Leader Nodes. The completion operation takes place when a generator node exhausts all alternatives and finds itself as a leader node. We designed novel algorithms to quickly determine whether a generator node is a leader node.

Our key idea is that each dependency frame holds a pointer to the presumed leader node of its SCC, and that the youngest consumer node always knows the leader for the current SCC. Hence, our leader node algorithm must always compute leader node information when first creating a new consumer node, say $\mathcal{C}$. To do so, we first hypothesize that the current leader node is $\mathcal{C}$'s generator node, say $\mathcal{G}$. Next, for all consumer nodes between $\mathcal{C}$ and $\mathcal{G}$, we check whether they depend on an older generator node. Consider that the oldest dependency is for $\mathcal{G}'$. If this is the case, then $\mathcal{G}'$ is the leader node, otherwise our hypothesis was correct and the leader is indeed $\mathcal{G}$.

Whenever we backtrack to a generator that it also the current leader node, we must check whether there are younger consumer nodes with unconsumed answers. This is implemented by going through the chain of dependency frames looking for a frame with unconsumed answers. If there is such a frame, we resume the computation to the corresponding consumer node. Otherwise, we perform completion. Completion includes **(i)** marking all the subgoals in the SCC as completed; **(ii)** deallocating all younger dependency frames; **(iii)** adjusting the freeze registers; and **(iv)** backtracking to the previous node to continue the execution.

Answer Resolution. Answer resolution has to be performed whenever the computation fails and is resumed at a consumer choice point. The implementation must guarantee that every answer is consumed once and just once. First, we check the table space for unconsumed answers for the subgoal at hand. If there are new answers, we load the next available answer and proceed with execution. Otherwise, we schedule for a backtracking node. If this is the first time that backtracking from that consumer node takes place, then it is performed as usual to the previous node. Otherwise, we know that the computation has been resumed from an older generator node $\mathcal{G}$ during an unsuccessful completion operation. Therefore, backtracking must be done to the next consumer node that has unconsumed answers and that is younger than $\mathcal{G}$. If there are no such consumer nodes then backtracking must be done to the generator node $\mathcal{G}$.

5 The Or-Parallel Tabling Engine

The OPTYap engine is based on the YapTab engine. However, new data structures and algorithms were required to support parallel execution. Next, we describe the main design and implementation decisions.

Memory Management. The efficiency of a parallel system largely depends on how concurrent handling of shared data is achieved and synchronized. Page faults and memory cache misses are a major source of overhead regarding data access or update in parallel systems. OPTYap tries to avoid these overheads by adopting a page-based organization scheme to split memory among different data structures, in a way similar to Bonwick's Slab memory allocator [2].

Our experience showed that the table space is a key data area open to concurrent access operations in a parallel tabling environment. To maximize parallelism, whilst minimizing overheads, accessing and updating the table space must be carefully controlled. Read/write locks are the ideal implementation scheme for this purpose. OPTYap implements four alternative locking schemes to deal with concurrent accesses to the table data structures. Our results suggested that concurrent table access is best handled by schemes that lock table data only when writing to the table is likely.

Leader Nodes. Or-parallel systems execute alternatives early. As a result, it is possible that generators will execute earlier, and in a different branch than in sequential execution. In the worst case, different workers may execute the generator and the consumer goals. Workers may have consumer nodes while not having the corresponding generators in their branches. Or, the owner of a generator node may have consumers being executed by several different workers. This may induce complex dependencies between workers, hence requiring a more elaborate completion operation that may involve branches created by several workers.

OPTYap allows completion to take place at any node, not only at generator nodes. In order to allow a very flexible completion algorithm we introduce a new

concept, the *generator dependency node* (or *GDN*). Its purpose is to signal the nodes that are candidates to be leader nodes, therefore representing a similar role as that of the generator nodes for sequential tabling. The GDN is calculated whenever a new consumer node, say $\mathcal{C}$, is allocated. It is defined as the youngest node $\mathcal{D}$ on the current branch of $\mathcal{C}$, that is an ancestor of the generator node $\mathcal{G}$ for $\mathcal{C}$. Figure 1 presents three different situations that better illustrate the GDN concept. $\mathcal{WG}$ is the worker that allocated the generator node $\mathcal{G}$, $\mathcal{WC}$ is the worker that is allocating a consumer node $\mathcal{C}$, and the node pointed by the black arrow is the GDN for the new consumer.

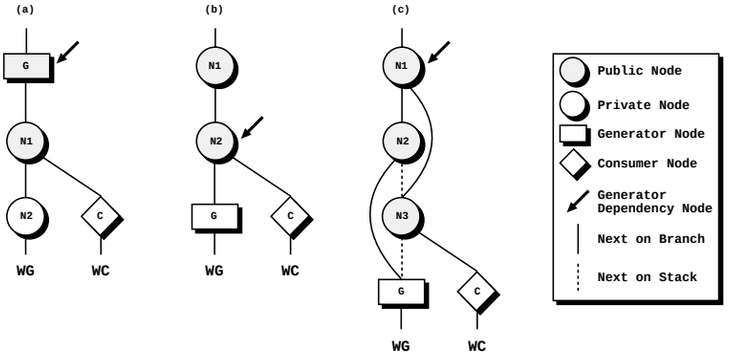


Fig. 1. Spotting the generator dependency node.

In situation **(a)**, the generator node $\mathcal{G}$ is on $\mathcal{C}$'s branch, and thus, $\mathcal{G}$ is the GDN. In situation **(b)**, nodes $\mathcal{N}_1$ and $\mathcal{N}_2$ are on $\mathcal{C}$'s branch, and both contain a branch leading to $\mathcal{G}$. As $\mathcal{N}_2$ is the youngest node of both, it is the GDN. In situation **(c)**, $\mathcal{N}_1$ is the unique node that belongs to $\mathcal{C}$'s branch and that also contains $\mathcal{G}$ in a branch below. $\mathcal{N}_2$ contains $\mathcal{G}$ in a branch below, but it is not on $\mathcal{C}$'s branch, while $\mathcal{N}_3$ is on $\mathcal{C}$'s branch, but it does not contain $\mathcal{G}$ in a branch below. Therefore, $\mathcal{N}_1$ is the GDN. Notice that in both cases **(b)** and **(c)** the GDN can be a generator, a consumer or an interior node.

The procedure to compute the leader node information when allocating a dependency frame for a new consumer node now hypothesizes that the leader node for the consumer node at hand is its GDN, and not its generator node.

The Control Flow. OPTYap's execution control mainly flows through four procedures. The process of completely evaluating SCCs is accomplished by the `completion()` and `answer_resolution()` procedures, while parallel synchronization is achieved by the `getwork()` and `scheduler()` procedures. Here we focus on the flow of control in engine mode, that is on the `completion()`, `answer_resolution()` and `getwork()` procedures, and discuss scheduling later. Figure 2 presents a general overview of how control flows between the three procedures and how it flows within each procedure.

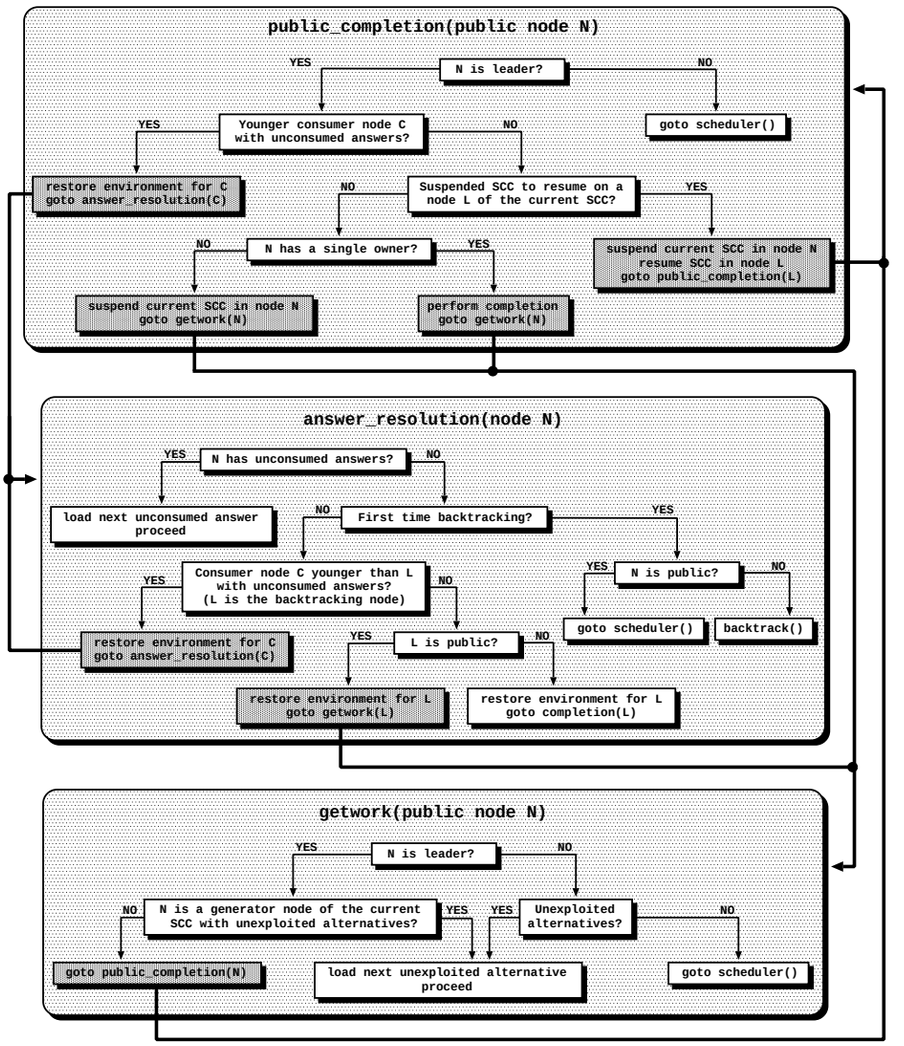


Fig. 2. The flow of control in a parallel tabled evaluation.

Public Completion. Different paths may be followed when a worker $\mathcal{W}$ reaches a leader node for a SCC $\mathcal{S}$. The simplest case is when the node is private. In this case, we proceed as for sequential tabling. Otherwise, the node is public, and there *may* exist dependencies on branches explored by other workers. Therefore, even when all younger consumer nodes on $\mathcal{W}$'s stacks do not have unconsumed answers, completion *cannot* be performed. The reason for this is that the other workers can still influence $\mathcal{S}$. For instance, these workers may find new answers for a consumer node in $\mathcal{S}$, in which case the consumer must be resumed to

consume the new answers. As a result, in order to allow $\mathcal{W}$ to continue execution it becomes necessary to *suspend the SCC* at hand.

Suspending in this context is obviously different from suspending consumer nodes. Consumer nodes are suspended due to tabled evaluation. SCCs are suspended due to or-parallel execution. Suspending a SCC includes saving the SCC's stacks to a proper space, leaving in the leader node a reference to where the stacks were saved, and readjusting the freeze registers and the stack and frame pointers. If the worker did not suspend the SCC, hence not saving the stacks, any future sharing work operation might damage the SCC's stacks and therefore make delayed completion unworkable.

To deal with the new particularities arising with concurrent evaluation a novel completion procedure, `public_completion()`, implements completion detection for public leader nodes. As for private nodes, whenever a public node finds that it is a leader, it starts to check for younger consumer nodes with unconsumed answers. If there is such a node, we resume the computation to it. Otherwise, it checks for suspended SCCs in the scope of its SCC. A suspended SCC should be resumed if it contains consumer nodes with unconsumed answers. To resume a suspended SCC a worker needs to copy the saved stacks to the correct position in its own stacks, and thus, it has to suspend its current SCC first.

We thus adopted the strategy of resuming suspended SCCs *only when the worker finds itself at a leader node*, since this is a decision point where the worker either completes or suspends the current SCC. Hence, if the worker resumes a suspended SCC it does not introduce further dependencies. This is not the case if the worker would resume a suspended SCC $\mathcal{R}$ as soon as it reached the node where it had suspended. In that situation, the worker would have to suspend its current SCC $\mathcal{S}$, and after resuming $\mathcal{R}$ it would probably have to also resume $\mathcal{S}$ to continue its execution. A first disadvantage is that the worker would have to make more suspensions and resumptions. Moreover, if we resume earlier, $\mathcal{R}$ may include consumer nodes with unconsumed answers that are common with $\mathcal{S}$. More importantly, suspending in non-leader nodes leads to further complexity. Answers can be found in upper branches for suspensions made in lower nodes, and this can be very difficult to manage.

A SCC $\mathcal{S}$ is completely evaluated when **(i)** there are no unconsumed answers in any consumer node in its scope, that is, in any consumer node belonging to $\mathcal{S}$ or in any consumer node within a SCC suspended in a node belonging to $\mathcal{S}$; and **(ii)** there is only a single worker owning its leader node $\mathcal{L}$. We say that a worker *owns* a node $\mathcal{N}$ when it holds $\mathcal{N}$ on its stacks (this is true even if $\mathcal{N}$ is not the worker's current branch). Completing a SCC includes **(i)** marking all dependent subgoals as complete; **(ii)** releasing the frames belonging to the complete branches, including the branches in suspended SCCs; **(iii)** releasing the frozen stacks and the memory space used to hold the stacks from suspended SCCs; and **(iv)** readjusting the freeze registers and the whole set of stack and frame pointers.

Our public completion algorithm has two major advantages. One is that the worker checking for completion determines if its current SCC is completely eval-

uated or not without requiring any explicit communication or synchronization with other workers. The other is that it uses the SCC as the unit for suspension. This latter advantage is very important since it simplifies the management of dependencies arising from branches not on stack. A leader node determines the position from where dependencies may exist in younger branches. As a suspension unit includes the whole SCC and suspension only occurs in leader node positions, we can simply use the leader node to represent the whole scope of a suspended SCC, and therefore simplify its management.

Answer Resolution. The answer resolution operation for the parallel environment essentially uses the same algorithm as previously described for private nodes.

Getwork. The last flow control procedure. It contributes to the progress of a parallel tabled evaluation by moving to effective work. The usual way to execute `getwork()` is through failure to the youngest public node on the current branch. We can distinguish two blocks of code in the `getwork()` procedure. The first block detects completion points and therefore makes the computation flow to the `public_completion()` procedure. The second block corresponds to or-parallel execution. It synchronizes to check for available alternatives and executes the next one, if any. Otherwise, it invokes the scheduler.

The `getwork()` procedure detects a completion point when $\mathcal{N}$ is the leader node pointed by the top dependency frame. The exception is if $\mathcal{N}$ is itself a generator node for a consumer node within the current SCC and it contains unexploited alternatives. In such cases, the current SCC is not fully exploited. Hence, we should exploit first the available alternatives, and only then invoke completion.

Scheduling Work. Scheduling work is the scheduler's task. It is about efficiently distributing the available work for exploitation between the running workers. In a parallel tabling environment we have the extra constraint of keeping the correctness of sequential tabling semantics. A worker enters in scheduling mode when it runs out of work and returns to execution whenever a new piece of unexploited work is assigned to it by the scheduler.

The scheduler for the OPTYap engine is mainly based on YapOr's scheduler. All the scheduler strategies implemented for YapOr were used in OPTYap. However, extensions were introduced in order to preserve the correctness of tabling semantics. These extensions allow support for leader nodes, frozen stack segments, and suspended SCCs. The OPTYap model was designed to enclose the computation within a SCC until the SCC was suspended or completely evaluated. Thus, OPTYap introduces the constraint that the *computation cannot flow outside the current SCC, and workers cannot be scheduled to execute at nodes older than their current leader node.* Therefore, when scheduling for the nearest node with unexploited alternatives, if it is found that the current leader node is younger than the potential nearest node with unexploited alternatives, then the current leader node is the node scheduled to proceed with the evaluation.

Moving In the Tree. The next case is when the process above does not return any node to proceed execution. The scheduler then starts searching for busy workers that can be requested for work. If such a worker $\mathcal{B}$ is found, then the requesting worker moves up to the lowest node that is common to $\mathcal{B}$, in order to become partially consistent with part of $\mathcal{B}$. Otherwise, no busy worker was found, and the scheduler moves the idle worker to a better position in the search tree. Therefore, we can enumerate three different situations for a worker to move up to a node $\mathcal{N}$: **(i)** $\mathcal{N}$ is the nearest node with unexploited alternatives; **(ii)** $\mathcal{N}$ is the lowest node common with the busy worker we found; or **(iii)** $\mathcal{N}$ corresponds to a better position in the search tree.

The process of moving up in the search tree from a current node $\mathcal{N}_0$ to a target node $\mathcal{N}_f$ is implemented by the `move_up_one_node()` procedure. This procedure is invoked for each node that has to be traversed until reaching $\mathcal{N}_f$. The presence of frozen stack segments or the presence of suspended SCCs in the nodes being traversed influences and can even abort the usual moving up process.

Assume that the idle worker $\mathcal{W}$ is currently positioned at $\mathcal{N}_i$ and that it wants to move up one node. Initially, the procedure checks for frozen nodes on the stack to infer whether $\mathcal{W}$ is moving within the SCC. If so, $\mathcal{W}$ is simply deleted from member of $\mathcal{N}_i$. The interesting case is when $\mathcal{W}$ is not within a SCC. If $\mathcal{N}_i$ holds a suspended SCC, then $\mathcal{W}$ can safely resume it. If resumption does not take place, the procedure proceeds to check whether $\mathcal{N}_i$ is a consumer node. Being this the case, $\mathcal{W}$ is deleted from member of $\mathcal{N}_i$ and if $\mathcal{W}$ is the unique owner of $\mathcal{N}_i$ then the suspended SCCs in $\mathcal{N}_i$ can be completed. Completion can be safely performed over the suspended SCCs in $\mathcal{N}_i$ not only because the SCCs are completely evaluated, as none was previously resumed, but also because no more dependencies exist, as there are no more branches below $\mathcal{N}_i$. The reasons given to complete the suspended SCCs in $\mathcal{N}_i$ hold even if $\mathcal{N}_i$ is not a consumer node, as long as $\mathcal{W}$ is the unique owner of $\mathcal{N}_i$. In such case, if $\mathcal{N}_i$ is a generator node then its correspondent subgoal can be also marked as completed. Otherwise, $\mathcal{W}$ is simply deleted from being member and owner of $\mathcal{N}_i$.

6 Initial Performance Evaluation

The environment for our experiments consists of a shared memory parallel machine, a 200 MHz PentiumPro with 4 processors, 128 MBytes of main memory, 256 KBytes of cache and running the linux-2.2.12 kernel. The machine was otherwise idle while benchmarking.

YapOr, YapTab and OPTYap are based on Yap's 4.2.1 engine. Note that sequential execution would be somewhat better with more recent Yap engines. We used the same compilation flags for Yap, YapOr, YapTab and OPTYap. Regarding XSB Prolog, we used version 2.3 with the default configuration and the default execution parameters (chat engine and batched scheduling).

Non-Tabled Benchmarks. To put the performance results in perspective we first use a common set of non-tabled benchmark programs to evaluate how the original

Yap Prolog engine compares against the several Yap extensions and against the most well-known tabling engine, XSB Prolog. The benchmarks include the n-queens problem, the puzzle and cubes problems from Evan Tick's book, an hamiltonian graph problem and a naïve sort algorithm. All benchmarks find all solutions for the problem.

Table 1 shows the base running times, in milliseconds, for Yap, YapOr, YapTab, OPTYap and XSB for the set of non-tabled benchmarks. In parentheses, it shows the overhead over the Yap running times. The results indicate that YapOr, YapTab and OPTYap introduce, on average, an overhead of about 6%, 8% and 12% respectively over standard Yap. Regarding XSB, the results show that, on average, XSB is 1.9 times slower than Yap, a result mainly due to the faster Yap engine.

Table 1. Running times on non-tabled programs.

Program	**Yap**	**YapOr**	**YapTab**	**OPTYap**	**XSB**
9-queens	584	604(1.03)	605(1.04)	626(1.07)	1100(1.88)
cubes	170	170(1.00)	173(1.02)	175(1.03)	329(1.94)
ham	371	402(1.08)	399(1.08)	432(1.16)	659(1.78)
nsort	310	330(1.06)	328(1.06)	354(1.14)	629(2.03)
puzzle	1633	1818(1.11)	1934(1.18)	1950(1.19)	3059(1.87)
Average		(1.06)	(1.08)	(1.12)	(1.90)

YapOr overheads result from handling the work load register and from testing operations that **(i)** verify whether a node is shared or private, **(ii)** check for sharing requests, and **(iii)** check for backtracking messages due to cut operations. On the other hand, YapTab overheads are due to the handling of the freeze registers and support of the forward trail. OPTYap overheads result from both.

Since OPTYap is based on the same environment model as the one used by YapOr, we then compare OPTYap's parallel performance with that of YapOr. Table 2 shows the speedups relative to the single worker case for YapOr and OPTYap with 2, 3 and 4 workers. Each speedup corresponds to the best execution time obtained in a set of 3 runs. The results show that OPTYap maintains YapOr's behavior in exploiting or-parallelism in non-tabled programs, despite that it includes all the machinery required to support tabled programs.

Tabled Benchmarks. We then use a set of tabled benchmark programs to measure the performance of the tabling engines in discussion. The benchmarks include two transition systems from XMC specs[1], a same generation problem for a 24x24x2 data cylinder, and two path problems that find the transitive closure of different graph topologies. All benchmarks find all the solutions for the problem.

[1] We are thankful to C.R. Ramakrishnan for providing us these benchmarks.

Table 2. Speedups for YapOr and OPTYap on non-tabled programs.

	YapOr			OPTYap		
Program	**2**	**3**	**4**	**2**	**3**	**4**
9-queens	1.99	2.99	3.94	2.00	2.99	3.96
cubes	2.00	2.98	3.95	1.98	2.96	3.97
ham	2.00	2.95	3.90	1.97	2.93	3.78
nsort	1.97	2.92	3.83	1.97	2.92	3.80
puzzle	2.02	3.03	4.02	1.98	2.97	3.94
Average	2.00	2.97	3.93	1.98	2.95	3.89

Table 3 shows the base running times, in milliseconds, for YapTab, OPTYap and XSB for the set of tabled benchmarks. In parentheses, it shows the overhead over the YapTab running times. The results indicate that OPTYap introduce, on average, an overhead of about 17% over YapTab for tabled programs, which is much worse than the overhead of 5% for non-tabled programs. The difference results from locking requests to handle the data structures introduced by tabling. Locks are require to insert new trie nodes into the table space, and to update subgoal and dependency frame pointers to tabled answers. We observed that the benchmarks that deal with more tabled answers per time unit are the ones that perform more locking operations and in consequence introduce further overheads.

Table 3. Running times on tabled programs.

Program	**YapTab**	**OPTYap**	**XSB**
xmc-sieve	2851	3226(1.13)	3560(1.25)
xmc-iproto	2438	2736(1.22)	4481(1.84)
same-gen	16598	17034(1.03)	25390(1.82)
path-grid	1069	1240(1.16)	3610(3.38)
path-chain	102	136(1.33)	271(2.66)
Average		(1.17)	(2.19)

Regarding XSB, the results show that, on average, YapTab is slightly more than twice as fast as XSB, surprisingly a better result than for non-tabled benchmarks. In particular, XSB shows the worst behavior for the two programs that are more table intensive. We believe that the XSB performance may be caused by overheads in their tabling implementation. XSB must support negated literals, and also has recently been extended to support attributed variables and especially subsumption.

Parallel Tabled Benchmarks. To assess the performance of OPTYap when running the tabled programs in parallel, we ran OPTYap for the same set of tabled

Table 4. Speedups for OPTYap on tabled programs.

	Number of Workers		
Program	**2**	**3**	**4**
xmc-sieve	2.00	3.00	3.99
xmc-iproto	1.90	2.78	3.64
same-gen	2.04	2.84	3.86
path-grid	1.82	2.54	3.10
Average	1.94	2.79	3.65
path-chain	0.92	0.86	0.78

programs with varying number of workers. Table 4 shows the speedups relative to the single worker case for OPTYap with 2, 3 and 4 workers. Each speedup corresponds to the best execution time obtained in a set of 3 runs. The table is divided in two blocks: the upper block groups the benchmarks that showed potential for parallel execution, whilst the lower block includes the benchmark that do not show any gains when run in parallel.

Globally, our results show quite good speedups for the upper block programs, especially considering that the execution times were obtained in a multiprocess environment. In particular, *xmc-sieve* achieves linear speedups up to 4 workers. The *same-gen* benchmark presents also excellent results up to 4 workers and *xmc-iproto* and *path-grid* show a slightly slowdown with the increase in the number of workers. On the other hand, the `path-chain` benchmark does not show any speedup at all.

Through experimentation, we observed that workers are busy for more than 95% of the execution time, even for 4 workers. In general, slowdowns are not caused because workers became idle and start searching for work, as usually happens with parallel execution of non-tabled programs. Here the problem seems more complex: workers do have available work, but there is a lot of contention to access that work.

Closer analysis suggested that there are two main reasons that constraint speedups. One relates with massive table access to insert and consume answers. As trie structures are a compact data structure, the presence of massive table access increases the number of contention points. The other relates with the sequencing in the order that answers are found. There are answers that can only be found when other answers are also found, and the process of finding such answers cannot be anticipated. This incurs in high overheads related with SCC suspensions and resumptions.

7 Conclusions

In this paper we have presented the design and implementation of OPTYap. To the best of our knowledge, OPTYap is the first parallel tabling engine for

logic programming systems. OPTYap extends the Yap Prolog system both with the SLG-WAM, initially implemented for XSB Prolog, and with environment copying, initially implemented in the Muse or-parallel system.

First results show that OPTYap introduces low overheads for sequential execution, and that it compares favorably with current versions of XSB. Moreover, the results showed that OPTYap maintains YapOr's effective speedups in exploiting or-parallelism in non-tabled programs. For parallel execution of tabled programs, OPTYap showed linear speedups for a well known application of XSB, and quite good results globally. These results emphasize our belief that tabling and parallelism are a very good match.

On the other hand, there are tabled programs where OPTYap may not speedup up execution. Parallel execution of tabled programs may have different characteristics than traditional or-parallel programs. In general, tabling tends to decrease the height of the search tree, whilst increasing its breadth. We therefore believe that improvements in scheduling and on concurrent access to tries may be fundamental for scalable performance. We plan to investigate this issue further, also by studying more programs.

References

1. K. Ali and R. Karlsson. The Muse Approach to OR-Parallel Prolog. *Journal of Parallel Programming*, 19(2):129–162, 1990.
2. Jeff Bonwick. The Slab Allocator: An Object-Caching Kernel Memory Allocator. In *USENIX Summer 1994*, pages 87–98, 1994.
3. W. Chen and D. S. Warren. Query Evaluation under the Well Founded Semantics. In *Proceedings of PODS*, pages 168–179, 1993.
4. W. Chen and D. S. Warren. Tabled Evaluation with Delaying for General Logic Programs. *Journal of the ACM*, 43(1):20–74, 1996.
5. B. Demoen and K. Sagonas. CAT: the Copying Approach to Tabling. In *Proceedings of PLILP*, number 1490 in LNCS, pages 21–35. Springer-Verlag, 1998.
6. B. Demoen and K. Sagonas. CHAT: the Copy-Hybrid Approach to Tabling. In *Proceedings of PADL*, number 1551 in LNCS, pages 106–121. Springer-Verlag, 1999.
7. J. Freire, R. Hu, T. Swift, and D. S. Warren. Exploiting Parallelism in Tabled Evaluations. In *Proceedings of PLILP*, pages 115–132. Springer-Verlag, 1995.
8. J. Freire, T. Swift, and D. S. Warren. Beyond Depth-First: Improving Tabled Logic Programs through Alternative Scheduling Strategies. In *Proceedings of PLILP*, pages 243–258. Springer-Verlag, 1996.
9. Hai-Feng Guo and G. Gupta. A New Tabling Scheme with Dynamic Reordering of Alternatives. In *Workshop on Parallelism and Implementation Technology for (Constraint) Logic Languages*, 2000.
10. I. V. Ramakrishnan, P. Rao, K. Sagonas, T. Swift, and D. S. Warren. Efficient Tabling Mechanisms for Logic Programs. In *Proceedings of ICLP*, pages 687–711. The MIT Press, 1995.
11. R. Rocha. *On Applying Or-Parallelism and Tabling to Logic Programs*. PhD thesis, Computer Science Department, University of Porto, 2001.
12. R. Rocha, F. Silva, and V. Santos Costa. Or-Parallelism within Tabling. In *Proceedings of PADL*, number 1551 in LNCS, pages 137–151. Springer-Verlag, 1999.

13. R. Rocha, F. Silva, and V. Santos Costa. YapOr: an Or-Parallel Prolog System Based on Environment Copying. In *Proceedings of EPIA*, number 1695 in LNAI, pages 178–192. Springer-Verlag, 1999.
14. R. Rocha, F. Silva, and V. Santos Costa. YapTab: A Tabling Engine Designed to Support Parallelism. In *Proceedings of TAPD*, pages 77–87, 2000.
15. K. Sagonas and T. Swift. An Abstract Machine for Tabled Execution of Fixed-Order Stratified Logic Programs. *Journal of ACM Transactions on Programming Languages and Systems*, 1998.
16. Vítor Santos Costa. Optimising Bytecode Emulation for Prolog. In *Proceedings of PPDP*, number 1702 in LNCS, pages 261–267. Springer-Verlag, 1999.
17. David H. D. Warren. An Abstract Prolog Instruction Set. Technical Note 309, SRI International, 1983.
18. Neng-Fa Zhou. Implementation of a Linear Tabling Mechanism. In *Proceedings of PADL*, number 1753 in LNCS, pages 109–123. Springer Verlag, 2000.

Revisiting the *Cardinality* Operator and Introducing the *Cardinality-Path* Constraint Family

Nicolas Beldiceanu and Mats Carlsson

SICS, Lägerhyddsvägen 18, SE-75237 Uppsala, Sweden
{nicolas,matsc}@sics.se

Abstract. This paper revisits the classical *cardinality* operator introducing new propagation rules that operate on variables that occur in more than one constraint. It also introduces a restricted case of the *cardinality* operator which is characterized by a structure of sliding constraints on consecutive variables. We call it *cardinality-path* and take advantage of these restrictions in order to come up with more efficient propagation algorithms. From an application point of view the *cardinality-path* constraint allows to express a host of regulation constraints occurring in personnel planning problems. We have used the meta-programming services of Prolog in order to implement the *cardinality-path* constraint within SICStus Prolog.

1 Introduction

Since its introduction 10 years ago, the *cardinality* operator [3] has been recognized as a generic concept which was progressively integrated in most of the modern constraint systems. It has the form $\text{cardinality}(C, \{CTR_1(V_{11},..,V_{1k_1}),..,CTR_n(V_{n1},..,V_{nk_n})\})$ where C is a domain variable[1] and $\{CTR_1(V_{11},..,V_{1k_1}),..,CTR_n(V_{n1},..,V_{nk_n})\}$ is a set of constraints. The *cardinality* operator holds iff:

$$C = \sum_{i=1}^{n} \#CTR_i(V_{i1},..,V_{ik_i}), \tag{1}$$

where $\#CTR_i(V_{i1},..,V_{ik_i})$ is equal to 1 if constraint $CTR_i(V_{i1},..,V_{ik_i})$ holds and 0 otherwise. From an operational point of view the *cardinality* operator used entailment [4] in order to implement the corresponding propagation. However a fundamental weakness of the previous propagation scheme is that it assumes each constraint to be independent. In practice this is a bit too optimistic assumption since, very often, the same variables occur in all the constraints of the *cardinality* operator. The first contribution of this paper is to provide new pruning rules for the *cardinality* operator that take advantage of the fact that some variables occur in more than one constraint.

[1] A domain variable is a variable that ranges over a finite set of integers; dom(*V*), min(*V*) and max(*V*) respectively denote the set of possible values of variable *V*, the minimum value of *V* and the maximum value of *V*.

P. Codognet (Ed.): ICLP 2001, LNCS 2237, pp. 59–73, 2001.

The second part of this paper introduces a restricted case of the *cardinality* operator which is characterized by a structure of sliding constraints on consecutive variables. We call it *cardinality-path* and present generic propagation algorithms for this family of constraints. It regroups a set of global constraints that were described in [1]. The *cardinality-path* family constraint has the form cardinality_path$(C, \{V_1,..,V_n\}, CTR)$ where C is a domain variable, $\{V_1,..,V_n\}$ is a collection of domain variables and CTR is a k -ary elementary constraint $(2 \leq k \leq n)$. The constraint holds iff:

$$C = \sum_{i=1}^{n-k+1} \#CTR(V_i,..,V_{i+k-1}), \tag{2}$$

where $\#CTR(V_i,..,V_{i+k-1})$ is equal to 1 if constraint $CTR(V_i,..,V_{i+k-1})$ holds and 0 otherwise. Condition (2) expresses the fact that the *cardinality-path* constraint holds if exactly C constraints out of the set $\{CTR(V_1,..,V_k), CTR(V_2,..,V_{k+1}),.., CTR(V_{n-k+1},..,V_n)\}$ are satisfied.

The first and second generic propagation algorithms that we present for the *cardinality-path* constraint take partially advantage of this specific structure in order to derive stronger pruning than the one that can be achieved by those rules described in [3, pages 749-751]. The third propagation algorithm combines the second algorithm with a special case of the new pruning rule introduced for the *cardinality* operator in order to derive stronger propagation.

The *cardinality-path* constraint family is also useful for those over-constrained problems having the structure described in the previous paragraph. In this case it allows to get an upper bound of the maximum number of constraints that hold and to propagate in order to try to achieve this upper bound.

In order to make all our propagation algorithms generic, constraint CTR is defined by the following functions:

- enforce_$CTR(V_i,..,V_{i+k-1})$: adds constraint $CTR(V_i,..,V_{i+k-1})$ to the constraint store,
- enforce_*NOT*_$CTR(V_i,..,V_{i+k-1})$: adds the negation[2] of constraint $CTR(V_i,..,V_{i+k-1})$ to the constraint store.

The previous functions trigger constraint propagation that will be carried on until saturation. Failure detection should be independent from the order in which constraints CTR are posted. In addition we use also the following primitives:

- create_choice_point: creates a choice point in order to be able to return to the current state later on,
- backtrack: restores the state of the domain variables and of the constraint store as it was on the last call to create_choice_point.

The next section presents new propagation rules for the *cardinality* operator. Sect. 3 describes the different aspects of *cardinality-path*: Sect. 3.1 first provides some instances of the *cardinality_path* constraint family; Sect. 3.2 and 3.3 show how to compute a lower and an upper bound of the number of elementary constraints that hold; Sect. 3.4 indicates how to prune variables $V_1,..,V_n$ according to the minimum and maximum value of C; finally, Sect. 3.5 shows how to partially integrate external

[2] The negation of constraint CTR is denoted $\neg CTR$; $\neg CTR(V_1,..,V_k)$ holds iff $CTR(V_1,..,V_k)$ does not hold.

constraints within the previous propagation algorithms. The last section provides for the *cardinality-path* constraint family a new pruning rule which combine the new pruning rule introduced for the *cardinality* operator in Sect. 2 with the algorithm described in Sect. 3.4.

2 Revisiting the *Cardinality* Operator

Consider the *cardinality* operator $\text{cardinality}(C, \{CTR_1(V_{11},..,V_{1k_1}),..,CTR_n(V_{n1},..,V_{nk_n})\})$ and let $V_1,..,V_p$ be a set of variables which all occur in all the different constraints $CTR_1,..,CTR_n$. Let $\text{dom}(V_j|CTR_i)$ denote the domain of variable V_j $(1 \le j \le p)$ under the assumption that CTR_i $(1 \le i \le n)$ is enforced[3]. The new pruning rule is based on the following observation: a value $val \in \text{dom}(V_j)$ can remain in $\text{dom}(V_j)$ only if $val \in \text{dom}(V_j|CTR_i)$ for at least $\min(C)$ different values of i. In fact, this rule is a generalization of constructive disjunction [4], [5]. This leads to the following algorithm.

```
1  nfail:=0;
2  FOR i:=1 TO n DO
3    create_choice_point;
4    IF enforce_CTR(Vi1,..,Vik_i) fails THEN nfail:=nfail+1;
5    ELSE
6      FOR j:=1 TO p DO
7        FOR all val∈dom(Vj) such that count_val[j,val][4]<min(C) DO
8          count_val[j,val]:=count_val[j,val]+1;
9    backtrack;
10   IF nfail>n-min(C) THEN RETURN fail;

11 adjust max(C) to n-nfail.
12 FOR j:=1 TO p DO
13   FOR all val∈dom(Vj) such that count_val[j,val]<min(C) DO
14     remove value val from Vj;
15 RETURN delay;
```

The first part of the algorithm initializes the `count_val` matrix (lines 1-10) while the last part (lines 11-15) exploits the `count_val` matrix in order to prune. A similar procedure performs pruning according to the negation of the different constraints $CTR_i(1 \le i \le n)$. If one is only interested by adjusting the minimum and maximum value of variables V_i $(1 \le i \le p)$ then we can instead use the following simplified version of the previous algorithm.

---

[3] $\text{dom}(V_j|CTR_i)$ is empty if CTR_i leads to a contradiction.

[4] We assume `count_val[j,val]` to be initialized to 0.

```
 1  nfail:=0;
 2  FOR i:=1 TO n DO
 3    create_choice_point;
 4    IF enforce_CTR(Vi1,..,Vik_i) fails THEN nfail:=nfail+1;
 5    ELSE
 6      FOR j:=1 TO p DO
 7        min_val[j,i]:=min(Vj);
 8        max_val[j,i]:=max(Vj);
 9    backtrack;
10    IF nfail>n-min(C) THEN RETURN fail;

11  adjust max(C) to n-nfail.
12  FOR j:=1 TO p DO
13    adjust min(Vj) to the min(C) smallest value of min_val[j,1..n][5];
14    adjust max(Vj) to the min(C) largest  value of max_val[j,1..n][6];
15  RETURN delay;
```

Line 13 removes all values $val \in \mathrm{dom}(V_j)$ which are smaller than the $\min(C)$ smallest value of $\min(\mathrm{dom}(V_j|CTR_i))$ $(1 \leq i \leq n)$, while line 14 discards all values $val \in \mathrm{dom}(V_j)$ which are greater than the $\min(C)$ largest value of $\max(\mathrm{dom}(V_j|CTR_i))$ $(1 \leq i \leq n)$.

3 The *Cardinality-Path* Constraint Family

3.1 Instances of the *Cardinality-Path* Constraint Family

The purpose of this paragraph is to provide various concrete examples of the *cardinality-path* constraint family. These examples are given in Table 1 and a possible practical use is provided for each of them at the end of this subsection.

The first column of Table 1 provides the arity k of the elementary constraint CTR, while the second column describes a member of the family in terms of the parameters of the *cardinality-path* family: it gives the initial lower and upper values for variable C and defines the elementary constraint CTR. Finally the last column of Table 1 describes the parameters of a family member and provides an example where the constraint holds. In order to make these examples more readable, two spaces after the value of a variable V_i $(1 \leq i \leq n-k+1)$ indicate that constraint $CTR(V_i,..,V_{i+k-1})$ does hold. For instance, the example $\mathrm{change}(3, \{4,4,\ 3,\ 4,\ 1\}, \neq)$ of the first row of Table 1 denotes that all the next three following constraints hold $4 \neq 3$, $3 \neq 4$, $4 \neq 1$, and that constraint $4 \neq 4$ does not hold. Note that the meaning of a family member can be derived from Condition (2) and from the second column of Table 1.

---

[5] We assume `min_val[j,1..n]` to be initialized to `max(Vj)`.

[6] We assume `max_val[j,1..n]` to be initialized to `min(Vj)`.

Table 1. Members of the *cardinality-path* constraint family

Arity	Bound for C and elementary constraint *CTR*	Member and example of a solution
2	$C : 0..n-1$ $X_i \neq X_{i+1}$	$\text{change}(C, \{V_1,..,V_n\}, \neq)$ $\text{change}(3, \{4,4,\ 3,\ 4,\ 1\}, \neq)$
2	$C : 0..n-1$ $(X_i + 1) \bmod L \neq X_{i+1}$	$\text{cyclic\_change}(C,\ L, \{V_1,..,V_n\})$ $\text{cyclic\_change}(2,\ 5, \{2,3,4,0,\ 2,3,\ 1\})$
2	$C : 0..n-1$ $(X_i + 1) \bmod L \neq X_{i+1} \wedge$ $X_i < L \ \wedge \ X_{i+1} < L$	$\text{cyclic\_change\_joker}(C,\ L, \{V_1,..,V_n\})$ $\text{cyclic\_change\_joker}(2,\ 4, \{3,0,\ 2,4,4,4,3,\ 1,4\})$
2	$C : 0..n-1$ $\lvert X_i - X_{i+1} \rvert > T$	$\text{smooth}(C, T, \{V_1,..,V_n\})$ $\text{smooth}(1,2, \{1,3,4,5,\ 2\})$
3	$C : 0..n-2$ $X_i = 0 \wedge X_{i+1} = 0 \wedge X_{i+2} \neq 0$	$\text{number\_of\_rest}(C, \{V_1,..,V_n\})$ $\text{number\_of\_rest}(2, \{2,0,\ 0,1,1,0,2,0,\ 0,1,2\})$
k	$C = n-k+1$ $low \leq \sum_{j=i}^{i+k-1} (X_j \ in \ Values)$[7] $\leq up$	$\text{among\_seq}(low, up, k, \{V_1,..,V_n\}, Values)$ $\text{among\_seq}(1,2,4, \{9,\ 2,\ 4,\ 5,\ 5,7,2\}, \{0,2,4,6,8\})$
k	$C = n-k+1$ $low \leq \sum_{j=i}^{i+k-1} X_j \leq up$	$\text{sliding\_sum}(low, up, k, \{V_1,..,V_n\})$ $\text{sliding\_sum}(3,7,4, \{1,\ 4,\ 2,\ 0,\ 0,3,4\})$
k	$C : atleast..atmost$ $low \leq \sum_{j=i}^{i+k-1} X_j \leq up$	$\text{relaxed\_sliding\_sum}(atleast, atmost, low, up, k, \{V_1,..,V_n\})$ $\text{relaxed\_sliding\_sum}(3,4,3,7,4, \{2,4,\ 2,\ 0,\ 0,3,4\})$

Constraints *change*, *cyclic_change*, *cyclic_change_joker*, *smooth*, *among_seq*, *sliding_sum* and *relaxed_sliding_sum* were respectively described at pages 43, 44, 45, 46, 40, 41 and 42 of [1]. From a practical point of view, the constraints of the previous table can be used for the following purpose:

- *change* can be used for timetabling problems in order to put an upper limit on the number of changes during a given period,
- *cyclic_change* may be used for personnel cyclic timetabling problems where each person has to work according to cycles that have to be sometimes broken,
- *cyclic_change_joker* may be used in the same context as the *cycle_change* constraint with the additional interpretation that holidays (i.e. those values that are greater than or equal to L) are not subject to cyclic constraint,

[7] $X \ in \ Values$ is equal to 1 if $X \in Values$, and 0 otherwise.

- *smooth* can be used to put a limit on the number of drastic variations on a given attribute (for example the number of persons working on consecutive weeks),
- *number_of_rest* allows controlling the number of rest days over a period of work, where a rest day is a period of at least two consecutive days off and one work day,
- *among_seq* may be used to express frequency constraints for producing goods for which one can have different variants (for example the car sequencing problem [2]),
- *sliding_sum* allows to restrict the total number of working hours on periods of consecutive days,
- *relaxed_sliding_sum* has the same utility as the *sliding_sum* constraint, but in addition allows expressing the fact that the rule may be broken sometimes.

More complete examples of utilization of the previous constraints can be found in [1]. The next two subsections indicate respectively how to compute a lower and an upper bound for the number of elementary constraints that hold.

3.2 Computing a Lower Bound of the Minimum Number of Elementary Constraints That Hold

The following greedy algorithm returns in `min_break` the minimum number of elementary constraints that hold. It tries to impose the negation of constraint CTR on consecutive variables as long as no failure occurs. A failure will correspond to the fact that posting a new constraint $\neg CTR$ leads to a contradiction. In order to keep the propagation implied by `enforce_NOT_CTR(Ui,..,Ui+k-1)` (line 10) local to the constraints we state, we duplicate variables $V_1,..,V_n$. However, usual saturation is used for the constraints we enforce; in particular they can trigger each other until no further deduction is possible. Variables $U_1,..,U_n$ will be deallocated when the last backtrack occurs.

```
 1  exist_choice_point:=1;
 2  create_choice_point;
 3  copy variables V1..Vn to U1..Un;
 4  min_break:=0;
 5  FOR i:=1 TO n-k+1 DO
 6    IF exist_choice_point=0 THEN
 7      exist_choice_point:=1;
 8      create_choice_point;
 9    END;
10    IF enforce_NOT_CTR(Ui,..,Ui+k-1) fails THEN
11      backtrack;
12      exist_choice_point:=0;
13      min_break:=min_break+1;
14    ENDIF;
15  ENDFOR;
16  IF exist_choice_point THEN backtrack END;
```

Let's call a *maximal sequence*, a sequence of consecutive variables $V_r,..,V_s$ $(r \geq 1,\ s \leq n,\ s-r+1 \geq k)$ such that:

- Propagation on the conjunction of constraints $\neg CTR(V_r,..,V_{r+k-1}),.., \neg CTR(V_{s-k+1},..,V_s)$ does not find a contradiction[8],
- s is equal to n or the propagation on the conjunction of constraints $\neg CTR(V_r,..,V_{r+k-1}),.., \neg CTR(V_{s-k+2},..,V_{s+1})$ finds a contradiction.

The greedy algorithm constructs a suite of maximal sequences of consecutive variables. It returns a valid lower bound since stopping a maximum sequence earlier will not allow expanding the next maximum sequence further on to the right. The lower bound may not be sharp since:

- It depends whether the propagation algorithm associated to constraint $\neg CTR$ is complete[9] or not.
- It depends on if we have a global propagation algorithm, which can take into account or not the fact that consecutive constraints partially overlap (i.e. have $k-1$ variables in common). For example, consider $\neg CTR$ being the constraint $V_i + V_{i+1} + V_{i+2} + V_{i+3} = 2$. Furthermore assume we have four 0-1 domain variables V_1,V_2,V_3,V_4. Suppose now that we apply $\neg CTR$ on each sliding sequence of four consecutive variables of the series $0,V_1,V_2,V_3,V_4$ (e.g. we have the two constraints $0+V_1+V_2+V_3=2$ and $V_1+V_2+V_3+V_4=2$). If each of the previous constraint is propagated in an independent way we will miss the fact that V_4 is equal to 0.
- If $\neg CTR$ is a constraint that involves more than 2 variables (i.e. $k>2$) then, the fact that we backtrack after a failure restores the domain of the variables to their initial state; however, since two consecutive maximum sequences have $k-2$ variables in common, there is an interaction that is ignored by our algorithm.

We illustrate the previous algorithm with an example of the *cyclic_change* constraint that was introduced in [1, page 44] and described in row 2 of Table 1. The *cyclic_change* constraint is a member of the *cardinality-path* family constraint where CTR is the following binary constraint: $(X_i+1) \bmod L \neq X_{i+1}$. L is a strictly positive integer. Constraint cyclic_change$(2,5,\{2,3,4,0,2,3,1\})$ holds since $(X_i+1) \bmod 5 \neq X_{i+1}$ is verified exactly 2 times, namely $(0+1) \bmod 5 \neq 2$ and $(3+1) \bmod 5 \neq 1$.

Let's assume we have the constraint cyclic_change$(C,5,\{V_1,V_2,V_3,V_4,V_5,V_6,V_7,V_8,V_9\})$ with the following initial domains: $V_1:\{0,3\}$, $V_2:\{2,3,4\}$, $V_3:\{0,4\}$, $V_4:\{0,1,2,3,4\}$, $V_5:\{0,1,2,3\}$, $V_6:\{0,2,4\}$, $V_7:\{0,1,2\}$, $V_8:\{0,1,2\}$, $V_9:\{0,4\}$. Table 2 gives the 2 maximum sequences V_1,V_2,V_3,V_4,V_5 and V_6,V_7,V_8 built by the algorithm in order to evaluate the minimum number of constraints that hold, namely 2 in this case. For each maximum sequence, a line in the

---

[8] This does not mean that there is a solution for this conjunction of constraints since the propagation may be incomplete.

[9] A propagation algorithm for constraint $CTR(V_1,..,V_n)$ is called complete if after propagation $\forall i \in \{1,2,..,n\}$, $\forall v \in dom(V_i)$, there exists at least one feasible solution for $CTR(V_1,..,V_n)$ with $V_i = v$.

table represents the constraint that is currently added and the state of the domains after posting that constraint.

Table 2. Maximum sequences of consecutive variables built by the algorithm

First maximal sequence	
	$V_1 : \{0,3\}$
$(V_1 + 1) \bmod 5 = V_2$:	$V_1 : \{3\}\ V_2 : \{4\}$
$(V_2 + 1) \bmod 5 = V_3$:	$V_1 : \{3\}\ V_2 : \{4\}\ V_3 : \{0\}$
$(V_3 + 1) \bmod 5 = V_4$:	$V_1 : \{3\}\ V_2 : \{4\}\ V_3 : \{0\}\ V_4 : \{1\}$
$(V_4 + 1) \bmod 5 = V_5$:	$V_1 : \{3\}\ V_2 : \{4\}\ V_3 : \{0\}\ V_4 : \{1\}\ V_5 : \{2\}$
$(V_5 + 1) \bmod 5 = V_6$:	contradiction
Second maximal sequence	
	$V_6 : \{0,2,4\}$
$(V_6 + 1) \bmod 5 = V_7$:	$V_6 : \{0,4\}\ V_7 : \{0,1\}$
$(V_7 + 1) \bmod 5 = V_8$:	$V_6 : \{0,4\}\ V_7 : \{0,1\}\ V_8 : \{1,2\}$
$(V_8 + 1) \bmod 5 = V_9$:	contradiction

3.3 Computing an Upper Bound of the Maximum Number of Elementary Constraints That Hold

We derive an upper bound (5) of the maximum number of elementary constraints that hold from the following two identities (3) and (4):

$$D = \sum_{i=1}^{n-k+1} \#\neg CTR(V_i,..,V_{i+k-1}), \tag{3}$$

$$C + D = n - k + 1, \tag{4}$$

$$\max(C) \leq n - k + 1 - \min(D). \tag{5}$$

Identity (3) introduces quantity D, which is the number of times that the negation of constraint CTR holds on variables $V_1,..,V_n$ (i.e. the number of discontinuities). Identity (4) states that the number of elementary constraints that hold plus the number of constraints that do not hold is equal to the total number of constraints $n-k+1$. Finally, Inequality (5) expresses the upper bound of the maximum number of elementary constraints that hold in term of the lower bound of the minimum number of continuities. In order to evaluate $\min(D)$, we use the algorithm described in Sect. 3.2, where we replace `enforce_NOT_CTR` by `enforce_CTR`.

3.4 Pruning According to the Minimum and Maximum Number of Elementary Constraints That Hold

We use the following algorithm in order to prune variables $V_1,..,V_n$ according to the maximum value of variable C. We remove values that otherwise would cause a too big number of elementary constraints CTR to hold.

```
 1  i:=1;
 2  FOR inc:=1 TO -1 (STEP -2) DO
 3    exist_choice_point:=1;
 4    create_choice_point;
 5    copy variables V1..Vn to U1..Un;
 6    min_break:=0;
 7    WHILE 1≤i AND i≤n-k+1 DO
 8      IF inc=1 THEN before[i]:=min_break
 9               ELSE after [i]:=min_break ENDIF;
10      IF exist_choice_point=0 THEN
11        exist_choice_point:=1;
12        create_choice_point;
13      ENDIF;
14      IF enforce_NOT_CTR(Ui,..,Ui+k-1) fails THEN
15        backtrack;
16        exist_choice_point:=0;
17        min_break:=min_break+1;
18      ENDIF;
19      IF inc=1 THEN record dom(Ui+k-1) in vbefore[i+k-1]
20               ELSE record dom(Ui    ) in vafter [i    ] ENDIF;
21      i:=i+inc;
22    ENDWHILE;
23    IF exist_choice_point THEN backtrack END;
24    i:=n-k+1;
25  ENDFOR;

26  IF min_break=max(C) THEN
27    FOR i:=1 TO n-k+1 DO
28      IF before[i]+after[i]+1>max(C) THEN
29        enforce_NOT_CTR(Vi,..,Vi+k-1);
30      ENDIF;
31    ENFOR;
32  ENDIF;

33  IF max(C)-min_break≤1 THEN
34    FOR i:=k TO n-k+1 DO
35      IF before[i-k+2]+after[i-1]+2>max(C) THEN
36        remove values v not in vbefore[i]∪vafter[i] from Vi;
37      ENDIF;
38    ENDFOR;
39  ENDIF;
```

When `inc` is equal to 1, the first part of the algorithm (lines 1 to 25) computes the minimum number of constraints that hold in the set of constraints $\{CTR(V_1,..,V_k)..,CTR(V_{i-1},..,V_{i+k-2})\}$ for each i between 1[10] and $n-k+1$. This number is recorded in `before`[i]. We also initialize the sets of values `vbefore`[i] $(k \le i \le n)$ to the values that are still in the domain of variable V_i just after propagating con-

[10] When i is equal to 1, the previous set is empty and `before`[i] is equal to 0.

straint $\neg CTR(V_{i-k+1},..,V_i)$ [11]. This is the first constraint that mentions variable V_i when we scan the constraints from left to right.

When `inc` is equal to –1, the first part of the algorithm (lines 1 to 25) computes the minimum number of constraints that hold in the set of constraints $\{CTR(V_{i+1},..,V_{i+k}),..,CTR(V_{n-k+1},..,V_n)\}$ for each i between 1 and $n-k+1$ [12]. This number is stored in after$[i]$. We also initialize the sets of values `vafter`$[i]$ $(n-k+1 \geq i \geq 1)$ to the values that are still in the domain of variable V_i just after propagating constraint $\neg CTR(V_i,..,V_{i+k-1})$ [13]. This is the first constraint that mentions variable V_i when we scan the constraints from right to left.

The `min_break` counter (line 17) is processed twice by the +1 and –1 loops (line 2). Because of the hypothesis made in the introduction "failure detection should be independent from the order in which constraints *CTR* are posted" we get twice the same value for the `min_break` counter.

The second part of the algorithm (lines 26 to 32) enforces constraint $\neg CTR(V_i,..,V_{i+k-1})$ to hold if the minimum number of constraints that hold in the set $\{CTR(V_1,..,V_k),..,CTR(V_{i-1},..,V_{i+k-2})\}$ plus the minimum number of constraints that hold in the set $\{CTR(V_{i+1},..,V_{i+k}),..,CTR(V_{n-k+1},..,V_n)\}$ is just equal to the maximum possible number of elementary constraints that hold.

The third part of the algorithm (lines 33 to 39) removes from a variable the values that cause two distinct additional elementary constraints to hold. We now prove that the third part of the algorithm removes only values that would lead to a failure of the *cardinality-path* constraint.

CASE 1: Assume that posting constraint $\neg CTR(V_{i-k+1},..,V_i)$ or constraint $\neg CTR(V_i,..,V_{i+k-1})$ did generate a failure (line 14). Since we backtrack (line 15), `vbfore`$[i]$ (line 19) or `vafter`$[i]$ (line 20) would contain all values of variable V_i. We derive from this fact that no value will be removed from variable V_i (line 36).

CASE 2: Let us assume that posting constraint $\neg CTR(V_{i-k+1},..,V_i)$ and constraint $\neg CTR(V_i,..,V_{i+k-1})$ did not generate any failure (line 14). In this case, we show that if $V_i \notin \texttt{vbefore}[i] \cup \texttt{vafter}[i]$ then the minimum number of constraints of $\{CTR(V_1,..,V_k),..,CTR(V_{n-k+1},..,V_n)\}$ that hold is greater than or equal to $\texttt{before}[i-k+2]+\texttt{after}[i-1]+2$.

---

[11] If $\neg CTR(V_{i-k+1},..,V_i)$ finds a contradiction then `vbefore`$[i]$ is initialized to the initial domain of variable V_i.

[12] When i is equal to $n-k+1$, the previous set of constraints is empty and `after`$[i]$ is equal to 0.

[13] If $\neg CTR(V_i,..,V_{i+k-1})$ finds a contradiction then `vafter`$[i]$ is initialized to the initial domain of variable V_i.

Let us note:

- $\min_{CTR}(a,b)$ the minimum number of constraints that hold in the conjunction of constraints $CTR(V_a,..,V_{a+k-1}), CTR(V_{a+1},..,V_{a+k}),..,CTR(V_b,..,V_{b+k-1})$, where $1 \le a \le b \le n-k+1$.
- f the smallest value less than or equal to $i-k+1$ such that the following two conditions are true:
  - No failure was detected during the first iteration of the algorithm (when `inc=1`) on the conjunction of constraints $\{\neg CTR(V_f,..,V_{f+k-1}), \neg CTR(V_{f+1},..,V_{f+k}),..,\neg CTR(V_{i-k+1},..,V_i)\}$,
  - $f=1$ or a failure was detected after stating $\neg CTR(V_{f-1},..,V_{f+k-2})$.
- l the largest value greater than or equal to i such that the following two conditions are true:
  - No failure was detected during the second iteration of the algorithm (when `inc=-1`) on the conjunction of constraints $\{\neg CTR(V_l,..,V_{l+k-1}), \neg CTR(V_{l-1},..,V_{l+k-2}),..,\neg CTR(V_i,..,V_{i+k-1})\}$,
  - $l=n-k+1$ or a failure was detected after stating $\neg CTR(V_{l+1},..,V_{l+k})$.

We have:

Partitioning the set of constraints $\{CTR(V_1,..,V_k),..,CTR(V_{i-k+1},..,V_i)\}$ in the set $\{CTR(V_1,..,V_k),..,CTR(V_{f-1},..,V_{f+k-2})\}$ and in $\{CTR(V_f,..,V_{f+k-1}),..,CTR(V_{i-k+1},..,V_i)\}$ leads to $\min_{CTR}(1,i-k+1) \ge \min_{CTR}(1,f-1) + \min_{CTR}(f,i-k+1)$.

From the definition of f we have that: $\min_{CTR}(1,f-1) \ge \texttt{before}[f] = \texttt{before}[i-k+2]$.

Since $V_i \notin \texttt{vbefore}[i]$ we also have that: $\min_{CTR}(f,i-k+1) \ge 1$.

So we conclude that: $\min_{CTR}(1,i-k+1) \ge \texttt{before}[i-k+2]+1$.

In a similar way, we have:

Partitioning the set of constraints $\{CTR(V_i,..,V_{i+k-1}),..,CTR(V_{n-k+1},..,V_n)\}$ in the set $\{CTR(V_i,..,V_{i+k-1}),..,CTR(V_l,..,V_{l+k-1})\}$ and in $\{CTR(V_{l+1},..,V_{l+k}),..,CTR(V_{n-k+1},..,V_n)\}$ leads to $\min_{CTR}(i,n-k+1) \ge \min_{CTR}(i,l) + \min_{CTR}(l+1,n-k+1)$.

Since $V_i \notin \texttt{vafter}[i]$ we also have that: $\min_{CTR}(i,l) \ge 1$.

From the definition of l we have that: $\min_{CTR}(l+1,n-k+1) \ge \texttt{after}[l] = \texttt{after}[i-1]$.

So we conclude that: $\min_{CTR}(i,n-k+1) \ge 1 + \texttt{after}[i-1]$.

So the minimum number of constraints that hold in $\{CTR(V_1,..,V_k),..,CTR(V_{n-k+1},..,V_n)\}$ is greater than or equal to $\texttt{before}[i-k+2] + \texttt{after}[i-1] + 2$. ❑

Table 3 gives an example of execution of the previous algorithm on the example introduced at the end of Sect. 3.

Table 3. Tables `before[]`, `after[]`, `vbefore[]`, `vafter[]` built by the algorithm

variables	V_1	V_2	V_3	V_4	V_5	V_6	V_7	V_8	V_9
domains	0,3	2,3,4	0,4	0,1,2,3,4	0,1,2,3	0,2,4	0,1,2	0,1,2	0,4
before[]	0	0	0	0	0	1	1	1	
vbefore[]		4	0	1	2	0,2,4	0,1	1,2	0,4
after[]	2	2	1	1	1	1	1	0	
vafter[]	3	3,4	0,4	2	3	0,4	0,1	0,1,2	

If at most two constraints should hold then part 2 (lines 26-32) will perform the following pruning:

- since `before`[1]+`after`[1]+1=3>2, line 29 imposes constraint $(V_1+1) \bmod 5 = V_2$, which fixes V_1 to 3 and V_2 to 4,
- since `before`[2]+`after`[2]+1=3>2, line 29 imposes constraint $(V_2+1) \bmod 5 = V_3$, which fixes V_3 to 0,
- since `before`[6]+`after`[6]+1=3>2, line 29 imposes constraint $(V_6+1) \bmod 5 = V_7$, which removes value 2 from variables V_6 and V_7,
- since `before`[7]+`after`[7]+1=3>2, line 29 imposes constraint $(V_7+1) \bmod 5 = V_8$, which removes value 0 from variable V_8.

If at most two constraints should hold then part 3 (lines 33-39) will perform the following pruning:

- since `before`[2]+`after`[1]+2=4>2, line 36 removes values in $\mathrm{dom}(V_2) - (\mathtt{vbefore}[2] \cup \mathtt{vafter}[2]) = \{2,3,4\} - \{3,4\} = \{2\}$ from variable V_2,
- since `before`[3]+`after`[2]+2=4>2, line 36 removes values in $\mathrm{dom}(V_3) - (\mathtt{vbefore}[3] \cup \mathtt{vafter}[3]) = \{0,4\} - \{0,4\} = \varnothing$ from variable V_3,
- since `before`[4]+`after`[3]+2=3>2, line 36 removes values in $\mathrm{dom}(V_4) - (\mathtt{vbefore}[4] \cup \mathtt{vafter}[4]) = \{0,1,2,3,4\} - \{1,2\} = \{0,3,4\}$ from variable V_4,
- since `before`[5]+`after`[4]+2=3>2, line 36 removes values in $\mathrm{dom}(V_5) - (\mathtt{vbefore}[5] \cup \mathtt{vafter}[5]) = \{0,1,2,3\} - \{2,3\} = \{0,1\}$ from variable V_5,
- since `before`[6]+`after`[5]+2=4>2, line 36 removes values in $\mathrm{dom}(V_6) - (\mathtt{vbefore}[6] \cup \mathtt{vafter}[6]) = \{0,2,4\} - \{0,2,4\} = \varnothing$ from variable V_6,
- since `before`[7]+`after`[6]+2=4>2, line 36 removes values in $\mathrm{dom}(V_7) - (\mathtt{vbefore}[7] \cup \mathtt{vafter}[7]) = \{0,1,2\} - \{0,1\} = \{2\}$ from variable V_7,
- since `before`[8]+`after`[7]+2=4>2, line 36 removes values in $\mathrm{dom}(V_8) - (\mathtt{vbefore}[8] \cup \mathtt{vafter}[8]) = \{0,1,2\} - \{0,1,2\} = \varnothing$ from variable V_8.

Finally, if at most three constraints should hold then part 3 (lines 33-39) will perform the following pruning:

- since `before`[2]+`after`[1]+2=4>3, line 36 removes values in $\mathrm{dom}(V_2) - (\mathtt{vbefore}[2] \cup \mathtt{vafter}[2]) = \{2,3,4\} - \{3,4\} = \{2\}$ from variable V_2,
- since `before`[3]+`after`[2]+2=4>3, line 36 removes values in $\mathrm{dom}(V_3) - (\mathtt{vbefore}[3] \cup \mathtt{vafter}[3]) = \{0,4\} - \{0,4\} = \varnothing$ from variable V_3,

- since `before`[6]+`after`[5]+2=4>3, line 36 removes values in $\mathrm{dom}(V_6) - (\texttt{vbefore}[6] \cup \texttt{vafter}[6]) = \{0,2,4\} - \{0,2,4\} = \varnothing$ from variable V_6,
- since `before`[7]+`after`[6]+2=4>3, line 36 removes values in $\mathrm{dom}(V_7) - (\texttt{vbefore}[7] \cup \texttt{vafter}[7]) = \{0,1,2\} - \{0,1\} = \{2\}$ from variable V_7,
- since `before`[8]+`after`[7]+2=4>3, line 36 removes values in $\mathrm{dom}(V_8) - (\texttt{vbefore}[8] \cup \texttt{vafter}[8]) = \{0,1,2\} - \{0,1,2\} = \varnothing$ from variable V_8.

A similar pruning for variables $V_1,..,V_n$ is done according to the minimum value of variable C. For this purpose we use the same algorithm, where we replace `enforce_NOT_CTR` by `enforce_CTR` and `max(C)` by `n-k+1-min(C)`. The previous quantity is the maximum number of constraints of the form $\neg CTR$ that hold.

3.5 Integrating Partially the "External World"

The purpose of this paragraph is to show how to partially integrate external constraints within some of the propagation algorithms of the *cardinality-path* constraint family. This is not a very common approach, since usually most of the constraint algorithms are local to a given constraint. However, this is especially relevant for getting stronger propagation.

The algorithm of Sect. 3.2, which computes a lower bound of the minimum number of elementary constraints that hold, can be modified as follows. We do not duplicate (line 3) any more the variables $V_1,..,V_n$, but work directly on them. This will result in waking the constraints we state inside the algorithm but also the external constraints mentioning a variable for which the domain is reduced. Finally, this may produce shorter maximum sequences than those obtained by the original algorithm. If this were the case, this would allow getting an improved lower bound of the minimum number of elementary constraints that hold.

4 Redundant Constraints for *Cardinality-Path*

This section shows how to generate redundant *cardinality* constraints for the *cardinality-path* constraint family. The goal is then to use the algorithms presented in Sect. 2 in order to perform additional propagation from these redundant constraints. The idea is as follows:

- first consider all the constraints $CTR(V_{i-k+1},..,V_i),..,CTR(V_i,..,V_{i+k-1})$ mentioning a given variable V_i $(k \le i \le n-k+1)$ of the *cardinality-path* constraint family,
- secondly, use the information provided by `before`[] and `after`[] in order to evaluate the minimum number of constraints CTR that hold both in $CTR(V_1,..,V_k),..,CTR(V_{i-k},..,V_{i-1})$ and in $CTR(V_{i+1},..,V_{i+k}),..,CTR(V_{n-k+1},..,V_n)$; this is equal to `before`[$i$–$k$+1]+`after`[$i$]. Since at most $\max(C)$ constraints hold in the whole set of constraints $CTR(V_1,..,V_k),..,CTR(V_{n-k+1},..,V_n)$ it follows that at most $\max(C) - \texttt{before}[i-k+1] - \texttt{after}[i]$ constraints hold in $CTR(V_{i-k+1},..,V_i),..,CTR(V_i,..,V_{i+k-1})$. Given that the previous set contains k con-

straints we have that at least $k-\max(C)+$ `before[`i–k+1`]+after[`i`]` constraints from $CTR(V_{i-k+1},..,V_i),..,CTR(V_i,..,V_{i+k-1})$ should fail.

– finally, since V_i is a variable which occurs in all constraints $CTR(V_{i-k+1},..,V_i),..,CTR(V_i,..,V_{i+k-1})$, and given that at least $k-\max(C)+$ `before[`i–k+1`]+after[`i`]` such constraints should fail we adapt the algorithm of Sect. 2 in the following way in order to try to prune variable V_i .

```
1  min_c:=k-max(C)+before[i-k+1]+after[i];
2  FOR j:=i-k+1 TO i DO
3    create_choice_point;
4    IF enforce_NOT_CTR(Vj,..,Vj+k-1)≠fail THEN
5      FOR all val∈dom(Vi) such that count_val[val][14]<min_c DO
6        count_val[val]:=count_val[val]+1;
7    backtrack;
8  FOR all val∈dom(Vi) such that count_val[val]<min_c DO
9    remove value val from Vi;
10 RETURN delay;
```

As an illustrative example consider the constraint relaxed_sliding_sum$(2,3,0,9,3,\{V_1,V_2,V_3,V_4,V_5\})$[15] where the initial domains of variables V_1,V_2,V_3,V_4,V_5 respectively are 1..9, 4..9, 1..9, 3..9 and 1..9. It imposes that at least 2 and at most 3 constraints of the form $0 \le V_i+V_{i+1}+V_{i+2} \le 9\ (1 \le i \le 3)$ to hold. The standard pruning introduced in Sect. 3 reduces the maximum value of V_3 to 5, but the previous propagation technique further restricts the maximum value of V_3 to 4. This is because two constraints enforce the maximum value of V_3 to be less than or equal to value 4: $0 \le V_1+V_2+V_3 \le 9$ imposes $\max(V_3) \le 4$, while $0 \le V_2+V_3+V_4 \le 9$ enforces $\max(V_3) \le 2$.

5 Conclusion and Open Questions

As a first contribution we have introduced for the *cardinality* operator new propagation rules which are not only based on entailment, as has been the case for the last 10 years. In fact, these rules can be regarded as a generalization of constructive disjunction [4]. As a second contribution, we have presented generic propagation algorithms for the *cardinality-path* constraint family. We have also shown how to extend these propagation algorithms in order to consider the influence of external constraints that share some variable with the *cardinality-path* constraint. As one can observe, one of the main advantages of generic propagation algorithms is that they can be applied to all constraints having some internal structure [1] in common. One should notice that the algorithms presented in this paper are still valid when the elementary constraints *CTR* do not all correspond to the same constraint. However from the *cardinality-path* constraint family perspective this is somehow a weakness since it means that we did not take advantage from the fact that we have the same constraint *CTR* .

[14] We assume `count_val[val]` to be initialized to 0.

[15] This constraint was introduced in the last entry of Table 1 of Sect. 3.1.

The following example shows that, even when the arity of the elementary constraint is equal to 2, our algorithm does not always find out that no solution exists. If we consider constraint cardinality_path$(1, \{0, V_1, V_2, V_3, 0\}, \neq)$ such that V_1, V_2, V_3 are 0-1 domain variables, then there is no solution since the number of satisfied disequality constraints is even. However the current algorithm seems to make a complete pruning in the case of binary constraints such as the *less* or the *greater* constraints.
From the previous remarks one can ask the following questions:

- For which class of binary constraints does our algorithm lead to complete pruning?
- For which class of binary constraints is there no need to perform saturation in order to get a complete pruning? In this case, propagation concerns only the elementary constraint that is currently posted and not the elementary constraints that were already posted.
- How to extend our algorithm in order to get more propagation for some classes of elementary constraints?
- Is it valid to apply the modification indicated in Sect. 3.5 to the pruning algorithm given in Sect. 3.4?

Acknowledgements

Thanks to Per Mildner and Emmanuel Poder for useful observations on an earlier draft of this paper as well as to anonymous reviewers for their comments.

References

1. Beldiceanu, N.: Global Constraints as Graph Properties on Structured Network of Elementary Constraints of the Same Type. SICS Technical Report T2000/01, (2000).
2. Dincbas, M., Simonis, H., Van Hentenryck, P.: Solving the Car-Sequencing Problem in Constraint Logic Programming. *ECAI 1988*, 290-295, (1988).
3. Van Hentenryck, P., Deville, Y.: The Cardinality Operator: A New Logical Connective for Constraint Logic Programming. *ICLP 1991*, 745-759, (1991).
4. Van Hentenryck, P., Saraswat, V., Deville, Y.: Design, Implementation and Evaluation of the Constraint Language cc(*FD*). In *A. Podelski, ed., Constraints: Basics and Trends*, vol. 910 of Lecture Notes in Computer Science, Springer-Verlag, (1995).
5. Würtz, J., Müller, T.: Constructive Disjunction Revisited. In *20th German Annual Conf. On AI*, LNAI 1137, 377-386, eds. G. Görz and S. Hölldobler, Springer-Verlag, (1996).

Optimizing Compilation of Constraint Handling Rules

Christian Holzbaur[1], María García de la Banda[2], David Jeffery[2], and Peter J. Stuckey[3]

[1] Dept. of Medical Cybernetics and Art. Intelligence, University of Vienna, Austria
christian@ai.univie.ac.at
[2] School of Comp. Sci. & Soft. Eng., Monash University, Australia
{mbanda,dgj}@csse.monash.edu.au
[3] Dept. of Comp. Sci. & Soft. Eng., University of Melbourne, Australia
pjs@cs.mu.oz.au

Abstract. CHRs are a multi-headed committed choice constraint language, commonly applied for writing incremental constraint solvers. CHRs are usually implemented as a language extension that compiles to the underlying language. In this paper we discuss the optimizing compilation of Constraint Handling Rules (CHRs). In particualr, we show how we can use different kinds of information in the compilation of CHRs in order to obtain access efficiency, and a better translation of the CHR rules into the underlying language. The kinds of information used include the types, modes, determinism, functional dependencies and symmetries of the CHR constraints. We also show how to analyze CHR programs to determine information about functional dependencies, symmetries and other kinds of information supporting optimizations.

1 Introduction

Constraint handling rules [3] (CHRs) are a very flexible formalism for writing incremental constraint solvers and other reactive systems. In effect, the rules define transitions from one constraint set to an equivalent constraint set. Transitions serve to simplify constraints and detect satisfiability and unsatisfiability. CHRs have been used extensively (see e.g. [4]). Efficient implementations are already available for the languages SICStus Prolog and Eclipse Prolog, and will soon appear for others such as Java [5] and HAL [2].

In this paper we discuss how to improve the compilation of CHRs by using additional information derived either from declarations provided by the user or from the analysis of the constraint handling rules themselves. The major improvements we discuss over previous papers [4] on CHR compilation are:

- general index structures which are specialized for the particular joins required in the CHR execution. Previous CHR compilation was restricted to two kinds of indexes: simple lists of constraints for given `Name/Arity` and lists indexed by the variables involved. For ground usage of CHRs this meant that only list indexes were used.

P. Codognet (Ed.): ICLP 2001, LNCS 2237, pp. 74–89, 2001.

- continuation optimization, where we use matching information from rules earlier in the execution to avoid matching later rules.
- optimizations that take into account algebraic properties such as functional dependencies, symmetries and the set semantics of the constraints.

We illustrate the advantages of the various optimizations experimentally on a number of small example programs in the HAL implementation of CHRs. We also discuss how the extra information required by HAL in defining CHRs (that is, type, mode and determinism information) is used to improve the execution.

In part, some of the motivation of this work revolves around a difference between CHRs in Prolog and in HAL. HAL is a typed language which does not (presently) support attributed variables. Prolog implementations of CHRs rely on the use of attributed variables to provide efficient indexing into the constraint store. Hence, we are critically interested in determining efficient index structures for storing constraints in the HAL implementation of CHRs. An important benefit of using specific index structures is that CHRs which are completely ground can still be efficiently indexed. This is not exploited in the current Prolog implementations. As many CHR solvers only use ground constraints this is an important issue.

2 Constraint Handling Rules and HAL

Constraint Handling Rules manipulate a global multiset of primitive constraints, using multiset rewrite rules which can take three forms

$$\begin{aligned} \textit{simplification}\ & [\textit{name@}]\ c_1, \ldots, c_n \iff g \mid d_1, \ldots, d_m \\ \textit{propagation}\ & [\textit{name@}]\ c_1, \ldots, c_n \implies g \mid d_1, \ldots, d_m \\ \textit{simpagation}\ & [\textit{name@}]\ c_1, \ldots, c_l \setminus c_{l+1}, \ldots, c_n \iff g \mid d_1, \ldots, d_m \end{aligned}$$

where *name* is an optional rule name, $c_1, \ldots, c_n$ are CHR constraints, g is a conjunction of constraints from the underlying language, and $d_1, \ldots, d_m$ is a conjunction of CHR constraints and constraints of the underlying language. The guard part g is optional. If omitted, it is equivalent to $g \equiv \mathit{true}$.

The simplification rule states that given a constraint multiset $\{c'_1, \ldots, c'_n\}$ and substitution θ matching the multiset $\{c_1, \ldots, c_n\}$, i.e. $\{c'_1, \ldots, c'_n\} = \theta(\{c_1, \ldots, c_n\})$, where the execution of $\theta(g)$ succeeds, then we can replace $\{c'_1, \ldots, c'_n\}$ by multiset $\theta(\{d_1, \ldots, d_m\})$. The propagation rule states that, for a matching constraint multiset $\{c'_1, \ldots, c'_n\}$ where $\theta(g)$ succeeds, we should add $\theta(\{d_1, \ldots, d_m\})$. The simpagation rules states that, given a matching constraint multiset $\{c'_1, \ldots, c'_n\}$ where $\theta(g)$ succeeds, we can replace $\{c'_{l+1}, \ldots, c'_n\}$ by $\theta(\{d_1, \ldots, d_m\})$. A *CHR program* is a sequence of CHRs.

The operational semantics of CHRs exhaustively apply rules to the global multiset of constraints, being careful not to apply propagation rules twice on the same constraints (to avoid infinite propagation). For more details see e.g. [1]. Although CHRs have a logical reading (see e.g. [3]) and programmers are encouraged to write confluent CHR programs, there are applications where a pre-

dictable order of rule applications is important. Hence, their textual order is used to resolve rule applicability conflicts in favor of earlier rules.

In this paper we focus on the implementation of CHRs in a programming language, such as HAL [2], which requires programmers to provide type, mode and determinism information. A simple example of a HAL CHR program to compute the greatest common divisor of two positive numbers a and b (using the goal `gcd(`a`), gcd(`b`)`) is given below.

```
:- module gcd.                                              (L1)
:- import int.                                              (L2)
:- export constraint gcd(int).                              (L3)
:- mode gcd(in) is det.                                     (L4)
base @ gcd(0) <=> true.                                     (L5)
pair @ gcd(N) \ gcd(M) <=> M >= N | gcd(M-N).               (L6)
```

The first line ($L1$) states that the file defines the module `gcd`. Line ($L2$) imports the standard library module `int` which provides (ground) arithmetic and comparison predicates for the type `int`. Line ($L3$) exports the CHR constraint `gcd/1` which has one argument, an `int`. This is the *type* declaration for `gcd/1`. Line ($L4$) is an example of a *mode of usage* declaration. The CHR constraint `gcd/1`'s first argument has mode `in` meaning that it will be fixed (ground) when called. The second part of the declaration "`is det`" is a determinism statement. It indicates that `gcd/1` always succeeds exactly once (for each separate call). For more details on types, modes and determinism see [2,6].

Lines ($L5$) and ($L6$) are the two CHRs defining the `gcd/1` constraint. The first rule is a simplification rule. It states that a constraint of the form `gcd(0)` should be removed from the constraint store to ensure termination. The second rule is a simpagation rule. It states that given two different `gcd/1` constraints in the store, such that one `gcd(M)` has a greater argument than the other `gcd(N)` we should remove the larger (the one after the \), and add a new `gcd/1` constraint with argument `M-N`. Together these rules mimic Euclid's algorithm.

The requirement of the HAL compiler to always have correct mode and determinism information means that CHR constraints can only have declared modes that do not change the instantiation state of their arguments,[1] since the compiler will be unable to statically determine when rules fire. The same restriction applies to dynamically scheduled goals in HAL (see [2]).[2]

3 Optimizing the Basic Compilation of CHRs

Essentially, the execution of CHRs is as follows. Every time a new constraint (the *active constraint*) is placed in the store, we search for a rule that can fire given this new constraint, i.e., a rule for which there is now a set of constraints

[1] They may actually change the instantiation state but this cannot be made visible to the mode system.

[2] Unlike dynamically scheduled goals in HAL, CHR constraints can have `multi` or `nondet` determinism.

that matches its left hand side. The first such rule (in the textual order they appear in the program) is fired.

Given this scheme, the bulk of the execution time for a CHR

$$c_1, \ldots, c_l[, \backslash]c_{l+1}, \ldots, c_n \iff g \mid d_1, \ldots, d_m$$

is spent in determining partner constraints $c'_1, \ldots, c'_{i-1}, c'_{i+1}, \ldots, c'_n$ for an active constraint c'_i to match the left hand side of the CHR. Hence, for each rule and each occurrence of a constraint, we are interested in generating efficient code for searching for partners that will cause the rule to fire. We will then link this code together to form the entire program for the constraint. A more detailed description of the overall process is given in [4], which is the basis for the SICStus Prolog version of CHRs. In the rest of this section, when applicable, we will show how different kinds of compile-time information can be used to improve the resulting code in the HAL version of CHRs.

3.1 Join Ordering

The left hand side of a rule together with the guard defines a multi-way join with selections (the guard) that could be processed in many possible ways, starting from the active constraint. This problem has been extensively addressed in the database literature. However, most of this work is not applicable since in the database context they assume the existence of information on cardinality of relations (number of stored constraints) and selectivity of various attributes. Since we are dealing with a programming language we have no access to such information, nor reasonable approximations. Another important difference is that, often, we are only looking for the first possible join partner, rather than all. In the SICStus CHR version, the calculation of partner constraints is performed in textual order and guards are evaluated once all partners have been identified. In HAL we determine an optimal join order and guard scheduling using, in particular, mode information.

Since we have no cardinality or selectivity information we will select a join ordering by using the number of unknown attributes in the join to estimate its cost. We assume an initial set *Fixed* of known variables (which arises from the active constraint), together with the set of (as yet unprocessed) partner constraints and guards. The algorithm measure shown in Figure 1, takes as inputs the set *Fixed*, the sequence *Partners* of partner constraints in a particular order, the set *FDs* of functional dependencies and the set *Guards* of guards, and returns the triple $(Measure, Goal, Lookups)$. *Measure* is an ordered pair representing the cost of the join for the particular order given by *Partners*. It is made up of the weighted sum $(n-1)w_1 + (n-2)w_2 + \cdots + 1w_{n-1}$ of the costs w_i for each individual join with a partner constraint. The cost of an individual join is defined as a pair: the number of arguments in the new partner which are unfixed before the join; followed by the (negative of) the number of arguments which are fixed before the join. *Goal* gives the ordering of partner constraints and guards (with guards scheduled as early as possible). Finally, *Lookups* gives

```
measure(Fixed,Partners,FDs,Guards)
  Lookups := ∅; Goal := true; score := (0,0); sum := (0,0)
  while true
    repeat
      Fixed0 := Fixed
      foreach g ∈ Guards
        if invars(g) ⊆ Fixed
          Goal := Goal,g; Fixed := Fixed ∪ outvars(g); Guards := Guards \ {g}
    until Fixed = Fixed0
    if Partners = ∅ return (score,Goal,Lookups)
    let Partners ≡ p(x̄),Partners1
    Partners := Partners1
    FDp := {p(x̄) :: fd ∈ FDs}
    Fixedp := fdclose(Fixed,FDp)
    fixedx := x̄ ∩ Fixedp
    cost := (|x̄ \ fixedx|,−|fixedx|)
    score := score + sum + cost; sum := sum + cost
    Lookups := Lookups ∪ {p((x_i ∈ fixedx ? x_i : _) | x_i ∈ x̄)}
    Fixed := Fixed ∪ x̄
    Goal := Goal,p(x̄);
  endwhile
```

Fig. 1. Algorithm for evaluating join ordering

the queries. Queries will be made from partner constraints, where a variable name indicates a fixed value, and an underscore (_) indicates an unfixed value. For example, query `p(X,_,X,Y,_)` indicates a search for `p/5` constraints with a given value in the first, third, and fourth argument positions, the values in the first and third position being the same.

Here we see the usefulness of mode information which allows us to schedule guards as early as possible. For simplicity, we treat mode information in the form of two functions: *invars* and *outvars* which return the set of input and output arguments of a procedure. We also assume that each guard has exactly one mode (it is straightforward to extend the approach to multiple modes and more complex instantiations). Functional dependencies are represented as $p(\bar{x}) :: S \rightsquigarrow x$ where $S \cup \{x\} \subseteq \bar{x}$ meaning that for constraint p fixing all the variables in S means there is at most one solution to the variable x. The function fdclose($Fixed$,FDs) closes a set of fixed variables $Fixed$ under the functional dependencies. fdclose($Fixed$,FDs) is the least set $F \supseteq Fixed$ such that for each $p(\bar{x}) :: S \rightsquigarrow x \in FDs$ such that $S \subseteq F$ then $x \in F$.

Example 1. Consider the compilation of the rule:

```
p(X,Y), q(Y,Z,T,U), flag, r(X,X,U) \ s(W) ==> W = U + 1, linear(Z) | p(Z,W).
```

for active constraint `p(X,Y)` and $Fixed = \{X, Y\}$. The scores calculated for the left-to-right partner order illustrated in the rule are $(3,-1)$, $(0,0)$, $(0,-2)$, $(0,-1)$ for a total cost of $(12,-9)$[3] together with goal

[3] Note that the cost of `r(X,X,U)` is $(0,-2)$ because `W = U + 1` is executed before `r(X,X,U)` thus grounding `U`. Also note that `X` is counted only once.

```
q(Y,Z,T,U), W = U + 1, linear(Z), flag, r(X,X,U), s(W)
```

and lookups `q(Y,_,_,_)`, `flag`, `r(X,X,U)`, `s(W)`. An optimal order has cost $(5, -7)$ resulting in goal

```
flag, r(X,X,U), W = U + 1, s(W), q(Y,Z,T,U), linear(Z)
```

and lookups `flag`, `r(X,X,_)`, `s(W)`, `q(Y,_,_,U)`.

For active constraint `q(Y,Z,T,U)`, the optimal order has cost $(2, -8)$ resulting in goal

```
W = U + 1, linear(Z), s(W), flag, p(X,Y), r(X,X,U)
```

and lookups `s(W)`, `flag`, `p(_,Y)`, `r(X,X,U)`.

For rules with large left hand sides where examining all permutations is too expensive we can instead greedily search for a permutation of the partners that is likely to be cost effective. In practice, we have not required this as left hand sides of CHRs are usually small.

3.2 Index Selection

Once join orderings have been selected, we must determine for each constraint a set of lookups of constraints of that form in the store. We then select an index or set of indexes for that constraint that will efficiently support the lookups required. Finally, we choose a data structure to implement each index. Mode information is crucial to the selection of index data structures. If the terms being indexed on are not ground, then we cannot use tree indexes since variable bindings will change the correct position of data.[4]

The current SICStus Prolog CHR implementation uses only two index mechanisms: Constraints for a given `Functor/Arity` are grouped, and variables shared between heads in a rule *index* the constraint store because matching constraints must correspondingly share a (attributed) variable. In the HAL CHR version, we put some extra emphasis on indexes for ground data:

The first step in this process is *lookup reduction*. Given a set of lookups for constraint `p/`k we reduce the number of lookups by using information about properties of `p/`k:

- lookup generalization: rather than build specialized indexes for lookups that share variables we simply use more general indexes. Thus, we replace any lookup $\mathtt{p}(v_1, \ldots, v_k)$ where v_i and v_j are the same variable by a lookup $\mathtt{p}(v_1, \ldots, v_{j-1}, v'_j, v_{j+1}, \ldots, v_k)$ where v'_j is a new variable. Of course, we must add an extra guard $v_i = v_j$ for rules where we use generalized lookups. For example, the lookup `eq(X,X)` can use the lookup for `eq(X,Y)`, followed by the guard `X = Y`.

[4] Currently HAL only supports CHRs with fixed arguments (although these might be variables from another (non-Herbrand) solver).

- functional dependency reduction: we can use functional dependencies to reduce the requirement for indexes. We can replace any lookup $\mathtt{p}(v_1,\ldots,v_k)$ where there is a functional dependency $p(x_1,\ldots,x_k)::\{x_{i_1},\ldots,x_{i_m}\} \rightsquigarrow x_j$ and $v_{i_1},\ldots,v_{i_m}$ are input (as opposed to anonymous) variables to the query by the lookup $\mathtt{p}(v_1,\ldots,v_{j-1},\_,v_{j+1},\ldots,v_k)$. For example, consider the constraint `bounds(X,L,U)` which stores the lower `L` and upper `U` bounds for a constrained integer variable `X`. Given functional dependency $bounds(X,L,U)::X \rightsquigarrow L$, the lookup `bounds(X,L,_)` can be replaced by `bounds(X,_,_)`.
- symmetry reduction: if $\mathtt{p}/k$ is symmetric on arguments i and j we have two symmetric lookups $\mathtt{p}(v_1,\ldots,v_i,\ldots,v_j,\ldots,v_k)$ and $\mathtt{p}(v'_1,\ldots,v'_i,\ldots,v'_j,\ldots,v'_k)$ where $v_l = v'_l$ for $1 \leq l \leq k, l \neq i, l \neq j$ and $v_i = v'_j$ and $v_j = v'_i$ then remove one of the symmetric lookups. For example, if `eq/2` is symmetric the lookup `eq(_,Y)` can use the index for `eq(X,_)`.

We discuss how we generate functional dependency and symmetry information in Section 5. We can now choose the data structures for the indexes that support the remaining lookups. The default choice is a balanced binary search tree (BST). Note that using a BST we can sometimes merge two indexes, for example, a BST for `eq(X,Y)` can also efficiently answer `eq(X,_)` queries.

Normally, the index will return an iterator which iterates through the multiset of constraints that match the lookup. Conceptually, each index thus returns a list iterator of constraints matching the lookup.[5] We can use functional dependencies to determine when this multiset can have at most one element. This is the case for a lookup $\mathtt{p}(v_1,\ldots,v_k)$ with fixed variables $v_{i_1},\ldots,v_{i_m}$ such that $\mathsf{fdclose}(\{x_{i_1},\ldots,x_{i_m}\},FDp) \supseteq \{x_1,\ldots,x_k\}$ where FDp are the functional dependencies for $\mathtt{p}/k$. For example, the lookup `bounds(X,_,_)` returns at most one constraint given the functional dependencies: $bounds(X,L,U) :: X \rightsquigarrow L$ and $bounds(X,L,U) :: X \rightsquigarrow U$. Iterators with at most one entry can return a `yesno` iterator rather than a list.

Since, in general, we may need to store multiple copies of identical constraints (CHR rules accept multisets rather than sets of constraints) each constraint needs to be stored with a unique identifier, called the *constraint number*. Code for the constraint will generate a new identifier for each new active constraint.

Each index for $\mathtt{p}(v_1,\ldots,v_k)$, where say the fixed variables are $v_{i_1},\ldots,v_{i_m}$, needs to support operations for initializing a new index, inserting and deleting constraints from the index and returning an iterator over the index for a given lookup. Note that the constraint number is an important extra argument for index manipulation. The compiler generates code for the predicates `p_insert_constraint` and `p_delete_constraint` which insert and delete the constraint `p` from each of the indexes in which it is involved.

Example 2. Suppose a CHR constraint `eq/2` has lookups `eq(X,_)` and `eq(_,Y)`, and `eq/2` is known to be symmetric in its two arguments. We can remove the

[5] Some complexities arise with the insertion and deletion of constraints *during* the execution of the iterator, because once a rule commits, the changes to the store regarding the removal of constraints and the addition of the active constraint have to take effect to implement an "immediate update view".

```
gcd_3(M,CN1) :-
 (gcd_index_exists_iteration(N,CN2),
  M >= N, CN1 != CN2 -> %% guard
     gcd_delete_constraint(M,CN1),
     gcd(M-N), %% RHS
     gcd_3_succ_cont(M,CN1)
    ;   gcd_3_fail_cont(M,CN1) ).

gcd_2(N,CN1) :-
 gcd_index_iteration_init(I0),
 gcd_2_forall_iterate(N,CN1,I0).

gcd_2_forall_iterate(N,CN1,I0) :-
  gcd_iteration_last(I0),
  gcd_2_succ_cont(N,CN1)).
gcd_2_forall_iterate(N,CN1,I0) :-
  gcd_iteration_next(I0, M, CN2, I1),
  (M >= N, CN1 != CN2 -> %% guard
     gcd_delete_constraint(M,CN2),
     gcd(M-N) %% RHS
    ; true), %% rule did not apply
 gcd_2_forall_iterate(N,CN1,I1).
```

Fig. 2. Code for existential partner search and universal partner search.

lookup `eq(_,Y)` in favor of the symmetric `eq(X,_)`, and then use a single balanced tree index for `eq(X,Y)` to store `eq/2` constraints since this can also efficiently retrieve constraints of the form `eq(X,_)`.

3.3 Code Generation for Individual Occurrences of Active Constraints

Once we have determined the join order for each rule and each active constraint, and the indexes available for each constraint, we are ready to generate code for each occurrence of the active constraint. Two kinds of searches for partners arise: A universal search iterates over all possible partners. This is required for propagation rules where the rule fires for each possible matching partner. An existential search looks for only the first possible set of matching partners. This is sufficient for simplification rules where the constraints found will be deleted.

We can split the constraints appearing on the left-hand-side of any kind of rule into two sets: those that are deleted by the rule (*Remove*), and those that are not (*Keep*). The partner search uses universal search behavior, up to and including the first constraint in the join which appears in *Remove*. From then on the search is existential. If the constraint has a functional dependency that ensures that there can be only one matching solution, we can replace universal search by existential search.

For each partner constraint we need to choose an available index for finding the matching partners. Since we have no selectivity or cardinality information, we simply choose the index with the largest intersection with the lookup.

Example 3. Consider the compilation of the 3rd occurrence of a `gcd/1` constraint in the program in the introduction (the second occurrence in ($L6$)) which is to be removed. Since the active constraint is in *Remove* the entire search is existential. The compilation produces the code `gcd_3` shown in Figure 2. The predicate `gcd_index_exists_iteration` iterates non-deterministically through the `gcd/1` constraints in the store using the index (on no arguments). It returns the value of the `gcd/1` argument as well as its constraint number. Next, the guard is checked. Additionally, we check that the two `gcd/1` constraints are in

fact different by comparing their constraint numbers (`CN1 != CN2`). If a partner is found, the active constraint is removed from the store, and the body is called. Afterwards, the success continuation for this occurrence is called. If no partner is found the failure continuation is called.

The compilation for second occurrence of a `gcd/1` constraint (the first occurrence in ($L6$)) requires universal search for partners. The compilation produces the code `gcd_2` shown in Figure 2. The predicate `gcd_index_iteration_init`, returns an iterator of `gcd/1` constraints resulting from looking up the index. Calls to `gcd_iteration_last` and `gcd_iteration_next` succeed if the iterator is finished and return values of the last and next `gcd/1` constraint (and its constraint number) as well as the new iterator.

3.4 Joining the Code Generated for Each Constraint Occurrence

After generating the code for each individual occurrence, we must join it all together in one piece of code. The code is ordered according to the textual order of the associated constraint occurrences except for simpagation rules where occurrences after the \ symbol are ordered earlier than those before the symbol (since they will then be deleted, thus reducing the number of constraints in the store). Let the order of occurrences be $o_1, \ldots, o_m$. The simplest way to join the individual rule code for a constraint $\mathtt{p}/k$ is as follows: Code for $\mathtt{p}/k$ creates a new constraint number and calls the first occurrence of code $\mathtt{p\_}o_1/k+1$. The fail continuation for $\mathtt{p\_}o_j/k+1$ is set to $\mathtt{p\_}o_{j+1}/k+1$. The success continuation for $\mathtt{p\_}o_j/k+1$ is also set to $\mathtt{p\_}o_{j+1}/k+1$ unless the active constraint for this occurrence is in *Remove* in which case the success continuation is *true*, since the active constraint has been deleted.

Example 4. For the `gcd` program the order of the occurrences is $1, 3, 2$. The fail continuations simply reflect the order in which the occurrences are processed: `gcd_1` continues to `gcd_3` which continues to `gcd_2` which continues to `true`. Clearly, the success continuation for occurrences 1 and 3 of `gcd/1` are `true` since the active constraint is deleted. The success continuation of `gcd_2` is `true` since it is last. The remaining code for `gcd/1` is given in Figure 3.[6]

4 Improving CHR Compilation

In the previous section we examined the basics steps for compiling CHRs taking advantage of type, mode, functional dependency and symmetries information. In this section we explore other kinds of optimizations based on analysis of CHRs.

4.1 Continuation Optimization

We can improve the simple strategy for joining the code generated for each occurrence of a constraint by noticing correspondences between rule matchings

[6] Note that later compiler passes remove the overhead of chain rules and empty rules.

```
gcd(N) :-
   new_constraint_number(CN1),       gcd_1_succ_cont(_,_).
   gcd_insert_constraint(N,CN1),     gcd_1_fail_cont(N,CN1) :- gcd_3(N,CN1).
   gcd_1(N,CN1).
gcd_1(N,CN1) :-                      gcd_3_succ_cont(_,_).
   (N = 0 -> %% Guard                gcd_3_fail_cont(N,CN1) :- gcd_2(N,CN1).
       gcd_delete_constraint(N,CN1),
       true, %% RHS                 gcd_2_succ_cont(_,_).
       gcd_1_succ_cont(N,CN1)        gcd_2_fail_cont(_,_).
    ;  gcd_1_fail_cont(N,CN1)).
```

Fig. 3. Initial code, code for first occurrence and continuation code for `gcd`.

for various occurrences. Suppose we have two consecutive occurrences with active constraints, partner constraints and guards given by the triples $(p(\bar{x}), c, g)$ and $(p(\bar{y}), c', g')$ respectively. Suppose we can prove that $\models (\bar{x} = \bar{y} \wedge (\bar{\exists}_{\bar{y}} c' \wedge g')) \rightarrow \bar{\exists}_{\bar{x}} c \wedge g$ (where $\bar{\exists}_V F$ indicates the existential quantification of F for all its variables not in set V). Then, anytime the first occurrence fails to match the second occurrence will also fail to match, since the store has not changed meanwhile. Hence, the fail continuation for the first occurrence can skip over the second occurrence. We can use whatever reasoning we please to prove the implication. Currently, both the SICStus and HAL version of the CHR compiler use very simple implication reasoning about identical constraints and `true`.

Example 5. Consider the following rules which manipulate `bounds(X,L,U)` constraints.

```
ne @ bounds(X,L,U) ==> U >= L.
red @ bounds(X,L1,U1) \ bounds(X,L2,U2)<=>L1 >= L2, U1 <= U2 | true.
int @ bounds(X,L1,U1), bounds(X,L2,U2) <=> bounds(X,max(L1,L2),min(U1,U2)).
```

For the 4th and 5th occurrences in rule `intersect` the implication

$$(X_4 = X_5 \wedge \exists L2_4, U2_4 bounds(X_4, L2_4, U2_4)) \rightarrow \exists L1_5, U1_5 bounds(X_5, L1_5, U1_5)$$

(where we use subscripts to indicate which is the active occurrence) holds. Hence, the 5th occurrence will never succeed if the 4th fails. Since if the 4th succeeds the active constraint is deleted, the 5th occurrence can be omitted entirely.

4.2 Late Storage

The first action in processing a new active constraint is to add it to the store, so that when it fires, the store has already been updated. In practice, this is inefficient since it may quite often be immediately removed. We can delay the addition of the active constraint until just before executing a right-hand-side that does not delete the active constraint, and can affect the store (i.e., may make use of the CHR constraints in the store).

Example 6. Consider the compilation of `gcd/1`. The first and third occurrences delete the active constraint. Thus, the new `gcd/1` constraint need not be stored before they are executed. It is only required to be stored just before the code for the second occurrence. The call to `gcd_insert_constraint` can be moved to the beginning of `gcd_2`, while the calls to `gcd_delete_constraint` in `gcd_1` and `gcd_3` can be removed.

This information can be inferred with a simple pre-analysis. For simplicity, we can consider a rule as *rhs-affects-store* if its right-hand-side calls a CHR constraint, or a local predicate which calls constraints (directly or indirectly), or (to be safe) an external predicate which is not a library predicate.

4.3 Set Semantics

Although CHRs use a multiset semantics, often the constraints defined by CHRs have a set semantics, where the number of copies of a constraint does not matter. In the HAL version, indexes for constraints with set semantics can take advantage of this information (by not worrying about duplicates). We can recognize constraints with set semantics in two different ways.

A constraint `p/`k has *set* semantics if there is a rule which explicitly removes duplicates of constraints. That is, if there exists a rule of the form

$$p(x_1, \ldots, x_k) \setminus p(y_1, \ldots, y_k) \iff g \mid true$$

such that $\models x_1 = y_1 \wedge \cdots x_k = y_k \rightarrow \bar{\exists}_{\bar{x} \cup \bar{y}} g$ which occurs before any rule requires `p/`k to be stored or which can match two identical copies of `p/`k. For instance, the rule `red` from Example 5 ensures that any new active `bounds/3` constraint identical to one already in the store will be deleted (it also deletes other redundant bounds information).

A constraint also has set semantics if all rules in which it appears behave the same even if duplicates are present. This is a very common case since CHRs are used to build constraint solvers which (by definition) should treat constraint multisets as sets. Thus, a constraint `p/`k also has *set* semantics if

- there are no rules which can match two identical copies of `p/`k
- there are no rules that delete a constraint `p/`k without deleting all identical copies.
- there are no rules with occurrences of `p/`k that can generate constraints (on the rhs) which do not have set semantics.

A simple fixpoint analysis can detect such constraints starting from the assumption that all constraints have set semantics.

For constraints `p/`k having this form we can safely add a rule of the form

$$p(x_1, \ldots, x_k) \setminus p(x_1, \ldots, x_k) \iff true.$$

This will avoid redundant work when duplicate constraints are added.

Example 7. Consider a constraint `eq/2` (for equality) defined by the CHR
`eq(X,Y),bounds(X,LX,UX),bounds(Y,LY,UY) ==> bounds(Y,LX,UX),bounds(X,LY,UY).`

Then, since `bounds/3` has set semantics, `eq/2` also has set semantics.

5 Determining Functional Dependencies and Symmetries

In previous sections we have either explained how to determine the information used for an optimization (as in the case of rules which are rhs-affects-store) or assumed it was given by the user or inferred by the compiler in the usual way (as in type, mode and determinism). The only two exceptions (functional dependencies and symmetries) were delayed in order not to clutter the explanation of CHR compilation. The following two sections examine how to determine these two properties.

5.1 Functional Dependencies

Functional dependencies occur frequently in CHRs since they encode functions using relations. Suppose `p/`k need not be stored before occurrences in a rule of the form

$$p(x_1, \ldots, x_l, y_{l+1}, \ldots, y_k)[, \backslash]p(x_1, \ldots, x_l, z_{l+1}, \ldots, z_k) \iff d_1, \ldots, d_m$$

where $x_i, 1 \leq i \leq l$ and $y_i, z_i, l+1 \leq i \leq k$ are distinct variables. Then, this rule ensures that there is at most one constraint in the store of the form $p(x_1, \ldots, x_l, \_, \ldots, \_)$ at any time. This corresponds to the functional dependencies $p(x_1, \ldots, x_k) :: (x_1, \ldots, x_l) \rightsquigarrow x_i, l+1 \leq i \leq k$. For example, the rule `int` of Example 5 illustrates the functional dependencies $bounds(X, L, U) :: X \rightsquigarrow L$ and $bounds(X, L, U) :: X \rightsquigarrow U$.

We can detect more functional dependencies if we consider multiple rules of the same kind. For example, the rules

$$p(x_1, \ldots, x_l, y_{l+1}, \ldots, y_k)[, \backslash]p(x_1, \ldots, x_l, z_{l+1}, \ldots, z_k) \iff g_1 | d_1, \ldots, d_m$$
$$p(x_1, \ldots, x_l, y'_{l+1}, \ldots, y'_k)[, \backslash]p(x_1, \ldots, x_l, z'_{l+1}, \ldots, z'_k) \iff g_2 | d'_1, \ldots, d'_{m'}$$

also lead to functional dependencies if $\models (\bar{y} = \bar{y}' \wedge \bar{z} = \bar{z}' \rightarrow (g_1 \vee g_2)$ is provable.

Example 8. The second rule for `gcd/1` written twice illustrates the functional dependency $\texttt{gcd}(N) :: \emptyset \rightsquigarrow N$ since $N = M' \wedge M = N' \rightarrow (M \geq N \vee M' \geq N')$ holds:

```
gcd(N) \ gcd(M) <=> M >= N | gcd(M - N).
gcd(N') \ gcd(M') <=> M' >= N' | gcd(M' - N').
```

Making use of this functional dependency for `gcd/1` we can use a single global `yesno` integer value (`$Gcd`) to store the (at most one) `gcd/1` constraint, we can replace the forall iteration by exists iteration, and remove the constraint numbers entirely. The resulting code (after unfolding) is

```
gcd(X) :-
    (X = 0 -> true              %% occ 1: guard -> rhs
    ; (yes(N) = $Gcd, X >= N    %% occ 3: gcd_index_exists_iteration, guard
        gcd(X-N)                %% occ 3: rhs
    ; (yes(M) = $Gcd, M >= X    %% occ 2: gcd_forall_iterate, guard
        $Gcd := yes(X),         %% occ 2: gcd_insert_constraint
        gcd(M-X)                %% occ 2: rhs
    ; $Gcd := yes(X)))).        %% late insert
```

5.2 Symmetry

Symmetry also occurs reasonably often in CHRs. There are multiple ways of detecting symmetries. A rule of the form

$$p(x_1, x_2, \ldots, x_k) \implies p(x_2, x_1, \ldots, x_k)$$

that occurs before any rule that requires `p/`k to be inserted induces a symmetry for constraint $p(x_1, \ldots, x_k)$ on x_1 and x_2, providing that no rule eliminates $p(x_1, x_2, \ldots, x_k)$ and not $p(x_2, x_1, \ldots, x_k)$.

Example 9. Consider a `!=/2` constraint defined by the rules:

```
neqset @ X != Y \ X != Y <=> true.
neqsym @ X != Y ==> Y != X.
neqlower @ X != Y, bounds(X,VX,VX), bounds(Y,VX,UY) ==> bounds(Y,VX+1,UY).
nequpper @ X != Y, bounds(X,VX,VX), bounds(Y,LY,VX) ==> bounds(Y,LY,VX-1).
```

the rule `neqsym @ X != Y => Y != X` illustrates the symmetry of `!=/2` w.r.t. `X` and `Y`, since in addition no rule deletes a (non-duplicate) `!=/2` constraint.

A constraint may be symmetric without a specific symmetry adding rule. The general case is complicated and, for brevity, we simply give an example.

Example 10. The rule in Example 7 and its rewriting with $\{X \mapsto Y, Y \mapsto X\}$ are logically equivalent (they are variants illustrated by the reordering of the rule).

```
eq(X,Y),bounds(X,LX,UX),bounds(Y,LY,UY) ==> bounds(Y,LX,UX),bounds(X,LY,UY).
eq(Y,X),bounds(Y,LY,UY),bounds(X,LX,UX) ==> bounds(X,LY,UY),bounds(Y,LX,UX).
```

Hence, since this is the only rule for `eq/2`, the `eq/2` constraint is symmetric.

6 Experimental Results

Our initial version of the HAL CHR compiler performs only some of the optimizations outlined above, including join ordering and continuation optimization and late storage. We do not yet have the automated analysis to support discovery of functional dependencies, set semantics and symmetries, nor specialized indexes (which rely on this information). It is ongoing work to improve the compiler, to perform the appropriate analyses, and make use of the information during the compilation.

To get an estimate on the benefits achievable through the optimizing compilation, we have modified the code produced by the current compiler by hand, in a way as close as possible to how we foresee its implementation. The comparisons against the SICStus CHR versions primarily serve as a simple "reality check" for the HAL version in its infancy. Any deductions beyond that would have to consider all the differences between SICStus and HAL producing Mercury code.

Table 1. Execution times (ms) for various optimized versions of CHR programs.

Benchmark	Query	Orig	+yesno	+det	+nn	hand	SICS
`gcd`	(5000000,3)	1912	1592	529	222	192	10271
`gcd`	(10000000,3)	7725	6766	1596	666	574	20522
`gcd`	(15000000,3)	11471	10216	2129	891	766	30770

Benchmark	Query	Orig	+tree	+sym	+eq	SICS	SICS_v
`interval`	(10,1)	1870	550	620	600	17491	1298
`interval`	(12,1)	9400	1710	1860	1890	72950	3725
`interval`	(14,1)	124000	19100	20765	20520	876550	32318
`interval`	(10,2)	2460	840	890	605	21834	1656
`interval`	(12,2)	12495	2710	2810	1850	92475	4795
`interval`	(14,2)	165595	32270	35375	20535	1115910	42382
`dfa`	180	57	56	54	=	57435	=
`dfa`	300	126	98	88	=	180220	=
`dfa`	3000	10576	1209	1061	=	too long	=

We compare the performance on 3 small programs:

- `gcd` as described in the paper, where the query (a,b) is `gcd(`a`),gcd(`b`)`.
- `interval`: a simple bounds propagation solver on N-queens; where the query (a, b) is for a queens with each constraint added b times (usually 1, just here to illustrate the possible benefits from set semantics).
- `dfa`: a visual parser for DFAs building the DFA from individual graphics elements, e.g. circles, lines and text boxes. The constraints are all ground, and the compilation involves a single (indexable) lookup $line(\_, Y)$, has a single symmetry $line(X, Y) = line(Y, X)$ and no constraints have set semantics. In this program the rules are large multi-ways joins, e.g., the rule to detect an arrow from one state to another is:

```
circle(C1,R1), circle(C2,R2) \
line(P1,P2), line(P3,P2), line(P4,P2), text(P5,T) <=>
    point_on_circle(P1,C1,R1), point_on_circle(P2,C2,R2),
    midpoint(P1,P2,P12), near(P12,P5) | arrow(P1,P2,T).
```

  The query a finds a (constant) small DFA (of 10 elements) in a large set of a graphical elements (to illustrate indexing).

The results are shown in Table 1. All timings are the average over 20 runs on a dual Pentium II-400MHz with 384M of RAM running under Linux RedHat 5.2 with kernel version 2.2, and are given in milliseconds. SICStus Prolog 3.8.4 is run under compact code (there is no fastcode for Linux).

For `gcd` we first give times for the original output of the compiler *Orig*. In the version *+yesno* the list storage of constraints is replaced by a *+yesno* structure (using the functional dependency). We can see a significant improvement here by just avoiding some loop overhead. Next in *+det* the determinism of produced

code is modified to take into account the functional dependency. Here we can really see the benefits of taking advantage of the functional dependency. Finally in $+nn$ constraint numbers are completely removed (and this massively simplifies the resulting code). We also give *hand* which uses the code in Example 8 (as a lower bound on where optimization can reach), and $SICS$ the original code in SICStus Prolog 3.8.4.

In the second experiment we give: the original code $Orig$, $+tree$ where all list indexes have been replaced by 234 trees, $+sym$ where symmetric constraints have the symmetry handled by indexes, and $+eq$ where set semantics optimizations are applied (note that an = means the code is identical. i.e. there was no scope for the optimization). Finally, we compare with the SICStus Prolog CHR compiler $SICS$, and for the `interval` example, a nonground version of the program $SICS\_v$ which uses attributed variable indexing (the other benchmarks involve only ground constraints).

The advantage of join ordering is illustrated by the difference between HAL and SICStus on `dfa`, the principle difference here is simply the better join ordering and early guard scheduling of HAL.

Adding indexes is clearly important when there are a significant number of constraints and effective lookups. In `dfa`, since there is only one indexed lookup, if the constraint stores are too small the overhead of the trees nullifies the advantages of the lookup. As the number of elements grows the advantages become clear.

Handling symmetry turned out to be disappointing. While it can reduce the number of indexes, it doubles their size and hence the possible benefit is limited. The overhead of managing symmetry in the index overwhelms the advantages when the constraint store is small, the advantages only becomes visible when the constraint store grows very large (`dfa` 3000). The handling of set semantics is of considerable benefit when duplicate constraints are actually added, and doesn't add significant overhead when there are no duplicate constraints, hence it seems worthwhile.

7 Conclusion and Future Work

The core of compiling CHRs is a multi-way join compilation. But, unlike the usual database case, we have no information on the cardinality of relations and index selectivity. We show how to use type and mode information to compile efficient joins, and automatically utilize appropriate indexes for supporting the joins. We show how functional dependencies and symmetries can improve this compilation process. We further investigate how, by analyzing the CHRs themselves we can find other opportunities for improving compilation, as well as determined functional dependencies, symmetries and other algebraic features of the CHR constraints.

References

1. S. Abdennadher. Operational semantics and confluence of constraint propagation rules. In *Proceedings of the Third International Conference on Principles and Practice of Constraint Programming*, pages 252–266, 1997.
2. B. Demoen, M. García de la Banda, W. Harvey, K. Marriott, and P.J. Stuckey. An overview of HAL. In *Proceedings of the Fourth International Conference on Principles and Practices of Constraint Programming*, pages 174–188, 1999.
3. T. Frühwirth. Theory and practice of constraint handling rules. *Journal of Logic Programming*, 37(1–3):95–138, 1998.
4. C. Holzbaur and T. Frühwirth. Constraint handling rules, special issue. *Journal of Applied Artificial Intelligence*, 14(4), 2000.
5. JACK: Java constraint kit. http://www.fast.de/~mandel/jack/.
6. Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury: an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29:17–64, 1996.

Building Constraint Solvers with HAL

María García de la Banda[1], David Jeffery[1], Kim Marriott[1],
Nicholas Nethercote[2], Peter J. Stuckey[2], and Christian Holzbaur[3]

[1] School of Comp. Sci. & Soft. Eng., Monash University, Australia.
{mbanda,dgj,marriott}@csse.monash.edu.au
[2] Dept. of Comp. Sci. & Soft. Eng., University of Melbourne, Australia.
{njn,pjs}@cs.mu.oz.au
[3] Dept. of Medical Cybernetics and Art. Intel., University of Vienna, Austria
christian@ai.univie.ac.at

Abstract. Experience using constraint programming to solve real-life problems has shown that finding an efficient solution to a problem often requires experimentation with different constraint solvers or even building a problem-specific solver. HAL is a new constraint logic programming language expressly designed to facilitate this process. In this paper we examine different ways of building solvers in HAL. We explain how type classes can be used to specify solver interfaces, allowing the constraint programmer to support modelling of a constraint problem independently of a particular solver, leading to easy "plug and play" experimentation. We compare a number of different ways of writing a simple solver in HAL: using dynamic scheduling, constraint handling rules and building on an existing solver. We also examine how external solvers may be interfaced with HAL, and approaches for removing interface overhead.

1 Introduction

There is no single best technique for solving combinatorial optimization and constraint satisfaction problems. Thus, constraint programmers would like to be able to easily experiment with different constraint solvers and to readily develop new problem-specific constraint solvers. The new constraint logic programming (CLP) language HAL [3] has been specifically designed to allow the user to easily experiment with different constraint solvers over the same domain, to support extension of solvers and construction of hybrid solvers, and to call procedures (in particular, solvers) written in other languages with little overhead.

In order to do so, HAL provides semi-optional type, mode and determinism declarations for predicates and functions. These allow the generation of efficient target code, ensure that solvers and other procedures are being used in the correct way, and facilitate efficient integration with foreign language procedures. Type information also means that predicate and function overloading can be resolved at compile-time, allowing a natural syntax for constraints. To facilitate writing simple constraint solvers, extending existing solvers and combining them, HAL provides dynamic scheduling by way of a specialized delay construct which supports definition of "propagators." Finally, HAL provides "global variables"

P. Codognet (Ed.): ICLP 2001, LNCS 2237, pp. 90–104, 2001.

which allow efficient implementation of a persistent constraint store. They behave in a similar manner to C's static variables and are only visible within the module in which they are defined.

The initial design of HAL was described in [3]. The current paper extends this in five main ways. First, we describe how the addition of type classes provides a natural way of specifying a constraint solver's capabilities and, therefore, support for "plug and play" with solvers. Second, we give a more detailed description of how HAL supports solver dependent dynamic scheduling and its use in writing solvers. Third, we describe how to integrate foreign language solvers into HAL and some programming techniques for reducing the runtime overhead of the solver interface. Fourth, we discuss the integration of constraint handling rules (CHRs) into HAL. Finally, we compare the efficiency of solvers written in HAL using CHRs, dynamic scheduling and type classes with comparable solvers written in SICStus and compare the overhead of HAL's external solver interface for CPLEX with that of ECLiPSe.

Thus, the main focus of the current paper is how to provide generic mechanisms such as type classes, dynamic scheduling and CHRs for structuring, writing and extending constraint solvers in a constraint programming language with type, mode and determinism declarations and what, if any, performance advantage is provided by this additional information.

2 The HAL Language

In this section we provide a brief overview of HAL [3], a CLP language which is compiled to the logic programming language Mercury [15].[1] The basic HAL syntax follows the standard CLP syntax, with variables, rules and predicates defined as usual (see, e.g., [14] for an introduction to CLP). The module system in HAL is similar to that of Mercury. A module is defined in a file, it `import`s the modules it uses and has `export` annotations on the declarations of the objects that it wishes to be visible to those importing the module. Selective importation is also possible. The core language supports the basic integer, float, string, and character data types plus polymorphic constructor types (such as lists) based on these basic types. This support is, however, limited to assignment, testing for equality, and construction and deconstruction of ground terms. More sophisticated constraint solving is provided by importing a constraint solver for each type involved.

As a simple example, the following program is a HAL version of the now classic CLP program `mortgage`.

```
:- module mortgage.                                              (L1)
:- import simplex.                                               (L2)
:- export pred mortgage(cfloat,cfloat,cfloat,cfloat,cfloat).     (L3)
:-        mode mortgage(oo,oo,oo,oo,oo) is nondet.               (L4)
```

[1] The key difference between them is that Mercury does not support constraints and constraint solvers. In fact, Mercury only provides a limited form of unification.

```
mortgage(P,0.0,I,R,P).                                          (R1)
mortgage(P,T,I,R,B) :- T >= 1.0, mortgage(P+P*I-R,T-1.0,I,R,B).  (R2)
```

Line ($L1$) states that this file defines the module `mortgage`. Line ($L2$) imports a module called `simplex`. This module provides a simplex-based linear arithmetic constraint solver for constrained floats, called `cfloat`s. Line ($L3$) exports the predicate `mortgage` which takes five `cfloat`s as arguments. This is the *type* declaration for `mortgage`. A type specifies the representation format of a variable. Thus, for example, the type system distinguishes between constrained floats (`cfloat`) and standard numerical floats (`float`) since they have a different representation. Types are defined using (polymorphic) regular tree type statements. For instance, the `list/1` constructor type is defined by

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

Line ($L4$) provides a *mode* declaration for `mortgage`. Mode declarations associate a mode with each argument of a predicate. A mode is a mapping $Inst_1 \rightarrow Inst_2$ where $Inst_1$ and $Inst_2$ describe the instantiation of an argument on call and on success from the predicate, respectively. The *base* instantiations are `new`, `old` and `ground`. Variable X is `new` if it has not been seen by the constraint solver, `old` if it has, and `ground` if X is constrained to take a fixed value. Note that `old` is interpreted as `ground` for variables of non-solver types (i.e., types for which there is no solver). The *base* modes are mappings from one base instantiation to another: we use two letter codes (`oo`, `no`, `og`, `gg`, `ng`) based on the first letter of the instantiation, e.g. `ng` is `new`→`ground`. The standard modes `in` and `out` are renamings of `gg` and `ng`, respectively. Therefore, line ($L4$) declares that each argument of `mortgage` has mode `oo`, i.e., takes an `old` variable and returns an `old` variable.

More sophisticated instantiations (lying between `old` and `ground`) may be used to describe the state of complex terms. Instantiation definitions look like type definitions. For example, the instantiation definition

```
:- instdef fixed_length_list -> ([] ; [old | fixed_length_list]).
```

indicates that the variable is bound to either an empty list or a list with an `old` head and a tail with the same instantiation.

Line ($L4$) also states the determinism for this mode of `mortgage`, i.e., how many answers it may have. We use the Mercury hierarchy: `nondet` means any number of solutions; `multi` at least one solution; `semidet` at most one solution; `det` exactly one solution, `failure` no solution and `erroneous` a run-time error.

The rest of the file contains the standard two rules defining `mortgage`.

3 Constraint Solvers and Type Classes

Type classes [13,17] support *constrained* polymorphism by allowing the programmer to write code which relies on a parametric type having certain associated predicates and functions. More precisely, a *type class* is a name for a set of types for which certain predicates and/or functions, called the *methods*, are defined.

Type classes were first introduced in functional programming languages Haskell and Clean, while Mercury and CProlog were the first logic programming languages to include them [12,4]. We have recently extended HAL to provide type classes similar to those in Mercury. One major motivation is that they provide a natural way of specifying a constraint solver's capabilities and, therefore, support for "plug and play" with solvers.

A `class` declaration defines a new type class. It gives the names of the type variables which are parameters to the type class, and the methods which form its interface. As an example, one of the most important built-in type classes in HAL is that defining types which support equality testing:

```
:- class eq(T) where [
        pred T = T,
        mode oo = oo is semidet ].
```

Instances of this class can be specified, for example, by the declaration

```
:- instance eq(int).
```

which declares the `int` type to be an instance of the `eq/1` type class. For this to be correct, the module must either define the method `=/2` with type `int=int` and mode `oo=oo is semidet` in the current module or indicate that it is a renaming of some other predicate.

We note that all types in HAL (like Mercury) have an associated "equality" for modes `in=out` and `out=in`, since these correspond to an assignment. Most types also support testing for equality, the main exception being for types that contain higher-order predicates. Thus, by default, HAL automatically generates instance declarations of the above form and the definition of `=/2` methods for all constructor types which contain types supporting equality.

Type classes allow us to naturally capture the notion of a type having an associated constraint solver: It is a type for which there is a method for initialising variables and a method for true equality. Thus, we define the `solver/1` type class to be:

```
:- class solver(T) <= eq(T) where [
        pred init(T),
        mode init(no) is det ].
```

This indicates that the `solver/1` type class provides an initialisation method `init/1` and that `solver/1` is a subclass of `eq/1` and, thus, any instance of `solver/1` must also be an instance of `eq/1`. Therefore, for type `T` to be in the `solver/1` type class, there must exist predicates `init/1` and `=/2` for this type with mode and determinism as shown.

Constructor types can be automatically declared to be instances of the `solver/1` type class using the notation `deriving solver`. The compiler then automatically generates an appropriate instance declaration and the predicate `init/1`. Variables whose type is not an instance of `solver/1` are not true logic variables, i.e., they are like Mercury terms since they must either be `new` or bound to a functor of the type. Thus, the type declaration given earlier for lists defines Mercury terms which have a fixed length while

```
:- typedef hlist(T) -> []; [T|hlist(T)] deriving solver.
```

defines true "Herbrand" lists.

Class constraints can appear as part of a predicate or function's type signature. They constrain the variables in the type signature to belong to particular type classes. Class constraints are checked and inferred during the type checking phase except for those of type classes `solver/1` and `eq/1` which must be treated specially because they might vary for different modes of the same predicate. In the case of `solver/1`, this will be true if the HAL compiler inserts appropriate calls to `init/1` for some modes (those in which the argument is initially `new`) but not in others. In the case of `eq/1`, this will be true if equalities are found to be assignments or deconstructions in some modes but true equalities in others. As a result, it is not until after mode checking that we can determine which variables in the type signature should be instances of `eq/1` and/or `solver/1`. Unfortunately, mode checking requires type checking to have taken place. Hence, the HAL compiler includes an additional phase after mode checking, where newly inferred `solver/1` and `eq/1` class constraints are added to the inferred types of procedures for modes that require them. Note that, unlike for other classes, if the declared type for a predicate does not contain the inferred class constraints, this is not considered an error, unless the predicate is exported.[2]

To illustrate the problem, consider the predicate

```
:- pred append(list(T),list(T),list(T)).
:- mode append(in,in,out) is det.
:- mode append(in,out,in) is semidet.
append([],Y,Y).
append([A|X1], Y, [A|Z1]) :- append(X1,Y,Z1).
```

During mode checking, the predicate `append` is compiled into two different procedures, one for each mode of usage (indicated by the keyword `implemented_by`). Conceptually, the code after mode checking is

```
:- pred append(list(T),list(T),list(T)) implemented_by [append_1, append_2].
:- mode append_1(in,in,out) is det.
append_1(X,Y,Z) :- X =:= [], Z := Y.
append_1(X,Y,Z) :- X =: [A|X1], append_1(X1,Y,Z1), Z := [A|Z1].
:- mode append_2(in,out,in) is semidet.
append_2(X,Y,Z) :- X =:= [], Y := Z.
append_2(X,Y,Z) :- X =: [A|X1], Z =: [B|Z1], A =:= B, append_2(X1,Y,Z1).
```

where `=:=`, `:=`, `=:` indicate calls to `=/2` with mode `(in,in)`, `(out,in)` and `(in,out)`, respectively. It is only now that we see that for the second mode the parametric type `T` must allow equality testing (be an instance of the `eq/1` class), because we need to compare `A` and `B`. Thus, in an additional phase of type inference the HAL compiler infers

```
:- pred append_2(list(T),list(T),list(T)) <= eq(T).
```

[2] Exported predicates need to have all their information available to ensure correct modular compilation. We plan to remove this restriction when the compiler fully supports cross module optimizing compilation [1].

Any predicates calling `append_2` will also inherit the `eq(T)` class constraint in their type.

This is a new problem for type classes and multi-moded predicates which does not arise in functional programming. While the same problem arises in Mercury for equality, it is side-stepped by not supporting an anologue of the `eq/1` class: effectively all types are required to support equality for mode `=(in,in)`. This may lead to run-time errors (e.g., when using `append_2` on lists of predicates). Since such errors are caught at compile-time by our two phase scheme, we believe our approach provides a better solution.

HAL provides a hierarchy of pre-defined type classes for common constraint domains which derive from the `solver` type class: `bool_solver`, `linfloat_solver`, `float_solver`, `linint_solver`, and `int_solver`. They provide a standard interface to solvers, thus facilitating "plug and play" experimentation by allowing separate compilation of the constraint models from the solvers that they use. As a result, we can rewrite the type declaration for `mortgage` to

```
:- export pred mortgage(T, T, T, T, T) <= float_solver(T).
```

thus allowing it to use any solver defined as an instance of `float_solver`.

Other important subclasses of the `solver` type class are `herbrand` (which includes as instances all constructor types declared as `deriving solver`) and its subclass the `prolog` type class. The role of the `herbrand` type class is to distinguish between constructor types and other user defined solver types. The `prolog` class requires the type to support a number of non-logical operations commonly used in Prolog style programming. For instance, it provides `var/1` and `nonvar/1` to test if a variable is still uninstantiated or not and standard functions to access the components of a term such as `functor`. It also provides the method `===/2` which succeeds only if both its arguments are variables and constrained to be equal.

A constructor type can be declared to support the Prolog built-ins by annotating the type declaration with `deriving prolog` rather than `deriving solver`. In this case, the compiler automatically generates definitions for a `prolog` class methods as well as those for the `solver` class. Distinguishing between `herbrand` and `prolog` allows the HAL compiler to differentiate between types which are used logically from those which are not (useful for optimization).

4 Dynamic Scheduling

An important feature of the HAL language is a form of "persistent" dynamic scheduling designed specifically to support constraint solving. A delay construct is of the form

$$cond_1 \texttt{ ==> } goal_1 \texttt{ || } \cdots \texttt{ || } cond_n \texttt{ ==> } goal_n$$

where the goal $goal_i$ will be executed when delay condition $cond_i$ is satisfied. By default, delayed goals remain active and are reexecuted whenever their delay condition becomes true again. This is useful, for example, if the delay condition

is "the lower bound has changed." However, delayed goals may also contain calls to the special predicate `kill/0` which kills all delayed goals in the immediate surrounding delay construct; that is, these goals will no longer be active.

The delay construct of HAL is designed to be extensible, so that programmers can build constraint solvers that support delay. In order to do so, one must create an instance of the `delay/2` type class defined as follows:

```
:- class delay_id(I) where [
        pred get_id(I),
        mode get_id(out) is det,
        pred kill(I),
        mode kill(in) is det ].
:- class delay(D,I) <= delay_id(I) where [
        pred delay(D, I, pred),
        mode delay(oo, in, in(pred is semidet)) is semidet ].
```

where type `I` represents the unique identifier (id) of each delay construct, `get_id/1` returns an unused id, `kill/1` causes all goals delayed for the input id to no longer wake up, type `D` represents the supported delay conditions, and `delay/3` takes a delay condition, an id and a goal,[3] and stores the information in order to execute the goal whenever the delay condition holds.

The separation of the delay type class into two parts allows different solver types to share delay ids. Thus, we can build delay constructs which involve more than one solver as long as they use a common delay id (the original design of delay [3] did not allow this).

The HAL compiler translates the delay construct into the base delay methods provided by the classes. Thus, the delay construct shown above is translated into:

`get_id(Id), delay(`$cond_1$`,Id,`$goal_1$`), ..., delay(`$cond_n$`,Id,`$goal_n$`)`

where each call to `kill/0` in a $goal_i$ is replaced by a call to `kill(Id)`.

Most modern logic programming languages allow predicates or goals to delay until a particular Herbrand variable is bound or is unified with another variable. In HAL a programmer can declare this by including `deriving delay` in the declaration for a constructor type. As when deriving from `solver/1` or `prolog/1`, the compiler will automatically generate the appropriate methods and instance declaration for that type. All such types use the common delay conditions `bound(X), touched(X)` and the common delay id type `system_delay_id` and its system defined instance of `delay_id`. Note that `system_delay_id` can also be used in programmer defined solvers.

As an example of the use of delay in constructing constraint solvers, the following program containst the code for (part of) a simple Boolean constraint solver.[4]

[3] To simplify analysis, each $goal_i$ must be `semidet` and may not change the instantiation of variables. As a result, delayed code cannot invalidate the mode and determinism checking when woken up.

[4] Note the `touched` delayed goals are included only for illustration, they are not used in the experiments.

```
:- module bool_delay.
:- instance bool_solver(boolv).
:- export_abstract typedef boolv -> ( f ; t ) deriving [prolog,delay].
:- export func true --> boolv.
true --> t.
:- export pred and(boolv,boolv,boolv).
:-        mode and(oo,oo,oo) is semidet.
and(X,Y,Z) :-
     ( bound(X) ==> kill, (X = f -> Z = f ; Y = Z)
     | bound(Y) ==> kill, (Y = f -> Z = f ; X = Z)
     | bound(Z) ==> kill, (Z = t -> X = t, Y = t ; notboth(X,Y))).
:- export func false --> boolv.
false --> f.
:- pred notboth(boolv,boolv).
:- mode notboth(oo,oo) is semidet.
notboth(X,Y) :-
     ( bound(X) ==> kill, (X = t -> Y = f ; true)
     | bound(Y) ==> kill, (Y = t -> X = f ; true)
     | touched(X) ==> (X === Y -> kill, X = f ; true)
     | touched(Y) ==> (X === Y -> kill, X = f ; true)).
```

The constructor type `boolv` is used to represent Booleans. Notice how the class functions `true` and `false` are simply defined to return the appropriate value, while the `and` predicate delays until one argument has a fixed value, and then constrains the other arguments appropriately. In the case of `notboth` we also test if two variables are identical. Hence, `boolv` must be declared as an instance of both the `prolog` type class and the `delay` type class (and, hence, implicitly as an instance of the `solver` type class.)

5 Using External Solvers from HAL

One of the main design requirements on the HAL language is that it should readily support integration into foreign language applications and, in particular, allow constraint solvers written in other languages to be called with relatively little overhead. An example of such a solver is CPLEX [10], a simplex based solver supporting linear arithmetic constraints.[5] This section details our experience integrating CPLEX into HAL.

The HAL interface for CPLEX is built on top of three Mercury predicates: the function `initialise_cplex` which returns a CPLEX solver instance CP, the predicate `add_column(`CP`,`n`)` which adds n columns to the tableau, and the predicate `add_equality(`CP`, [(`c_1`,`v_1`),...,(`c_n`,`v_n`)], `b`)` which adds the equation $c_1 \cdot v_1 + \cdots + c_n \cdot v_n = b$ to the tableau. These predicates wrap the C interface functions of CPLEX. This is easy to do since C code can be directly written as part of a Mercury predicate body. These predicates also handle trailing and restoration of choice points. This is done by using the higher-order predicate

[5] CPLEX also provides routines for mixed integer programming and barrier methods but we have not yet integrated these.

`trail/1` which places its argument (a predicate closure), on the function trail to be called in the event of backtracking to `trail/1`.

A naive way to write the interface in HAL is as follows.

```
:- module cplex.
:- instance linfloat_solver(cfloat).
:- import int.
:- export_abstract typedef cfloat -> col(int).
:- reinst_old cfloat = ground.
:- glob_var CPLEX has_type cplex_instance init_value initialise_cplex.
:- glob_var VarNum has_type int init_value 0.
:- export_only pred init(cfloat).
:-              mode init(no) is det.
init(V) :- V = col($VarNum), $VarNum := $VarNum + 1, add_column($CPLEX,1).
:- export_only pred cfloat = cfloat.
:-              mode oo = oo is semidet.
V1 = V2 :- add_equality($CPLEX, [(1.0,V1),(-1.0,V2)], 0.0).
:- export func cfloat + cfloat --> cfloat.
V1 + V2 --> V3 :-
      init(V3),
      trust_det add_equality($CPLEX, [(1.0,V1),(1.0,V2),(-1.0,V3)], 0.0).
:- export func float x cfloat --> cfloat.
C x V1 --> V2 :-
      init(V2),
      trustdet add_equality($CPLEX, [(C,V1),(-1.0,V2)], 0.0).
:- coerce coerce_float(float) --> cfloat.
:- export func coerce_float(float) --> cfloat.
coerce_float(C) --> V :-
      init(V),
      trust_det add_equality($CPLEX, [(1.0,V)], C).
```

The solver type `cfloat` is a wrapped integer giving the column number of the variable in the CPLEX tableau. It is exported abstractly to provide an abstract data type, and declared to be an instance of the linear arithmetic constraint solver class `linfloat_solver`. The `reinst_old` declaration states that the instantiation `old` for `cfloat`s must be interpreted as `ground` inside this module reflecting their internal implementation. We use two global variables: `CPLEX` for storing the CPLEX instance, and `VarNum` for storing the number of variables (columns) in the solver.

The predicate `init/1` simply increments the counter `VarNum` and adds a column to the CPLEX tableau. The `=/2` predicate adds an equality to the CPLEX tableau. Both are designated as `export_only`, which makes them visible outside the module, but not inside. This avoids confusion with the internal view of `cfloat`s as wrapped integers rather than the external view as float variables. The function `+/2` initialises a new variable to be the result of the addition and adds an equality constraint to compute the result. The `trust_det` annotation allows the compiler to pass the determinism check (the solver author knows that this call to `add_equality` will not fail). The linear multiplication function `x/2` is defined similarly to `+/2`.

Since `cfloat`s are constrained floats, it is convenient to be able to use floating point constants in place of `cfloat`s. HAL allows the solver programmer to specify the automatic coercion of a base type to a solver type. In our example, the `coerce` directive declares that `coerce_float` is a coercion function and the next three lines give its type, mode and definition.

Unfortunately, this naive interface has a high overhead. One issue is that many arithmetic constraints are simple assignments or tests which do not require the power of a linear constraint solver. Thus, we can improve the interface by only passing "real" constraints to the solver and "solving" simple assignments and tests in the interface functions themselves. This can be done easily by redefining the `cfloat` type to wrap either a true variable or a constant value and redefining our interface functions to handle the different cases appropriately.

Another issue is that the interface splits complex linear equations into a large number of intermediate constraints and variables. A better approach is to have `+` and `x` build up a data structure representing the linear constraint. More precisely, we can redefine `cfloat` to be this data structure and for `+/2`, `x/2`, `init/1` and `coerce_float/1` to build the data structure. As a by product, this data structure can also be used to track constants and perform tests and assignments in the interface. The modified code is:

```
:- export_abstract typedef cfloat -> cfloat(float,list(cterm)).
:-                 typedef cterm  -> (float,int).
init(V) :- V = cfloat(0.0,[(1.0,$VarNum)]),
            $VarNum := $VarNum + 1, add_column($CPLEX,1).
cfloat(C1, Vs1) = cfloat(C2, Vs2) :-
            negate_coeffs(Vs2, NewVs2),
            append(Vs1, NewVs2, Terms),
            add_equality($CPLEX, Terms, C2-C1).
cfloat(C1, Vs1) + cfloat(C2, Vs2) -->
            cfloat(C1+C2,Vs) :- append(Vs1, Vs2, Vs).
C x cfloat(F, Vs) --> cfloat(C*F,NewVs) :- multiply_coeffs(C, Vs, NewVs).
coerce_float(C) --> cfloat(C, []).
```

Also, in external solvers such as CPLEX that are not specialized for incremental satisfiability checking, the usual CLP approach of checking satisfiability after each new constraint is added, may be expensive. We can therefore improve performance by "batching" constraints and requiring the programmer to explicitly call the solver to check for satisfiability.

6 Using Constraint Handling Rules (CHRs)

Constraint Handling Rules (CHRs) have proven to be a very flexible formalism for writing incremental constraint solvers and other reactive systems. In effect, the rules define transitions from one constraint set to an equivalent constraint set. Rules are repeatedly applied until no new rule can be applied. Once applied, a rule cannot be undone. For more details the interested reader is referred to [6].

The simplest kind of rule is a *propagation* rule of the form

$$lhs \texttt{ ==> } guard \texttt{ | } rhs$$

where lhs is a conjunction of CHR constraints, $guard$ is a conjunction of constraints of the underlying language (in practice this is any goal not involving CHR constraints) and rhs is a conjunction of CHR constraints and constraints of the underlying language. The rule states that if there is a set S appearing in the global CHR constraint store G that matches lhs such that goal $guard$ is entailed by the current constraints, then we should add the rhs to the store. *Simplification rules* have a similar form (replacing the `==>` with a `<=>`) and behavior except that the matching set S is deleted from G. A syntactic extension allows only part of the lhs to be eliminated by a simplification rule:

$$lhs_1 \setminus lhs_2 \texttt{ <=> } guard \texttt{ | } rhs$$

indicates that only the set matching lhs_2 is eliminated.

Efficient implementations of CHRs are provided for SICStus Prolog, Eclipse Prolog (see [5]) and Java [11]. Recently, they have also been integrated into HAL [8]. As in most implementations, HAL CHRs sit on top of the "host" language. More exactly, they may contain HAL code and are essentially compiled into HAL in a pre-processing stage of the HAL compiler. As a consequence, CHR constraints defined in HAL require the programmer to provide type, mode and determinism declarations.

The following program implements part of a Boolean solver implemented in HAL using CHRs.[6]

```
:- module bool_chr.
:- instance bool_solver(boolv).
:- export_abstract
   typedef boolv -> wrap(int).
:- reinst_old boolv = ground.

:- glob_var VNum
   has_type int init_value 0.

:- export_only
   pred init(boolv).
:- mode init(no) is det.
init(V) :- V = wrap($VNum),
           $VNum := $VNum + 1.
```

```
:- export constraint true(boolv).
:-        mode true(oo) is semidet.
:- export constraint false(boolv).
:-        mode false(oo) is semidet.
true(X), false(X) <=> fail.
:- export constraint
          and(boolv,boolv,boolv).
:-        mode and(oo,oo,oo) is semidet.
true(X) \ and(X,Y,Z) <=> Y = Z.
true(Y) \ and(X,Y,Z) <=> X = Z.
false(X) \ and(X,Y,Z) <=> false(Z).
false(Y) \ and(X,Y,Z) <=> false(Z).
false(Z) \ and(X,Y,Z) <=> notboth(X,Y).
true(Z) \ and(X,Y,Z) <=> true(X), true(Y).
```

In this case `boolv`s are simply variable indices[7] and Boolean constraints and values are implemented using CHR constraints. Initialization simply builds a new term and increments the Boolean variable counter `VNum` which is a global variable. The `constraint` declaration is like a `pred` declaration except it indicates that it is a CHR predicate. The mode and determinism for each CHR constraint are defined as usual. The remaining parts are CHRs. The first rule

[6] Somewhat simplified for ease of exposition.

[7] HAL does not yet support CHRs on Herbrand types

states that if a variable is given both truth values, true and false, we should fail. The next rule (for `and/3`) states that if the first argument is true we can replace the constraint by an equality of the remaining arguments.

In HAL, CHR constraints must have a mode which does not change the instantiation of their arguments (like `oo` or `in`) to preserve mode safety, since the compiler is unlikely to statically determine when rules fire. Predicates appearing in the guard must also be `det` or `semidet` and not alter the instantiation of variables appearing in the left hand side of the CHR (this means they are implied by the store). This is a weak restriction since, typically, guards are simple tests.

7 Evaluation

Our first experiment has three aims. First, it illustrates the use of type classes for "plug and play" with solvers. Second, it determines the overhead of using type classes when implementing solvers. Third, it evaluates the efficiency of the generic solver writing constructs supported by HAL: dynamic scheduling and CHRs. For this experiment we created three implementations of a propagation-based Boolean constraint solver: using dynamic scheduling (*dyn*); using CHRs (*chr*); and using conversion to integer constraints (*int*).[8] We give two results for each HAL solver: $solv_t$ which uses type classes for separate compilation, where each query module was compiled separately from the solver, and joined at link time; and $solv_i$ where the query module imported the solver, and was compiled with this knowledge, so removing the overhead of type classes. It is important to note that type classes allowed us to use identical code for the benchmarks: only at linking time did we need to choose which solver to use.

To evaluate the efficiency of HAL,[9] we also built comparable solvers in SICStus Prolog: using the generic `when` delay mechanism ($SICS_w$) closest to our generic delay mechanism, using the CHRs of SICStus ($SICS_c$); and using the `clfd` integer propagation solver ($SICS_z$). Finally, for interest, we provide two more SICStus solvers: ($SICS_b$) a dynamic scheduling solver using the highly restricted but efficient `block` mechanism of SICStus, and ($SICS_v$) where the ground variable numbers in the CHR solver are replaced by Prolog variables, allowing the use of attribute variable indices.

The comparison uses five simple Boolean benchmarks (most from [2]): the first `pigeon`n`-`m places n pigeons in m pigeon holes (the 24-24 query succeeds, while 8-7 fails); `schur`n Schurs's lemma for n (see [2]) (the 13 query is the largest n that succeeds); `queens`n the Boolean version of this classic problem; `mycie`n`-`m which colors a 5-colorable graph (taken from [16]) with n nodes and m edges with 4 colours; and `fulladder` which searches for a single faulty gate in a n bit adder (see e.g. [14] for the case of 1 bit).

[8] More exactly, we use the integer propagation solver described in [7] which is implemented in C and interfaced to HAL using the methodology described in Section 5.

[9] It is much easier to build highly flexible but inefficient mechanisms for defining solvers.

Table 1. Comparison of Boolean solvers for dynamic scheduling.

Benchmark	Var	Con	Search	Dynamic Scheduling			
				dyn_t	dyn_i	$SICS_w$	$SICS_b$
mycie23_71	184	583	19717	1855	1816	34769	1920
fulladder	135	413	1046	472	471	5181	147
pigeon24_24	1152	13896	576	805	822	2258	56
pigeon8_7	112	444	24296	931	901	16870	843
queens18	972	13440	42168	8904	8818	125250	7316
schur13	178	456	57	22	18	118	4
schur14	203	525	450	98	112	1308	63

Table 2. Comparison of Boolean solvers using an existing integer solver and CHRs.

Benchmark	CHRs				Integer		
	chr_t	chr_i	$SICS_c$	$SICS_v$	int_t	int_i	$SICS_z$
mycie23_71	25073	25070	200613	76567	1279	1251	5339
fulladder	4240	4270	24770	13840	178	175	313
pigeon24_24	71455	70785	107366	71048	81	72	957
pigeon8_7	11313	11176	61126	36251	765	726	3166
queens18	504750	511620	1350636	263433	4522	4438	12363
schur13	53	63	450	201	9	7	51
schur14	823	830	5966	2319	55	52	278

Table 1 gives an indication of how much work the solvers are performing for each benchmark. Var is the number of variables initialised by the solver, Con is the number of Boolean constraints, and Search is the number of labeling steps performed (using the default labeling strategy to find a first solution). Note that each solver implements *exactly* the same propagation strength on Boolean constraints and, thus, for each benchmark each different solver performs exactly the same search. All timings are the average over 10 runs on a dual Pentium II-400MHz with 384M of RAM running under Linux RedHat 5.2 with kernel version 2.2, and are given in milliseconds. SICStus Prolog 3.8.4 is run under compact code (no fastcode for Linux).

From Table 1 it is clear that the generic delay mechanism implemented in HAL is reasonably efficient. In comparison with the propagation happening in C in the integer solver, the dynamic scheduled version is only 4 times slower. It also compares well with the generic dynamic scheduling of SICStus. However, the `block` based dynamic scheduling of SICStus illustrates how delay that is tightly tied to the execution mechanism can be very efficient.

Table 2 shows that the CHR solver mechanism for HAL (at least for this example) is significantly faster than the SICStus equivalent, so much so that in this case even the use of attributed variable indexing does not regain the

Table 3. Executing CPLEX using the various HAL interfaces.

Bench	*naive*		*constants*			*datastructures*				*ECL*	
	Con	*inc*	Con	*inc*	*batch*	Con	*inc*	*batch*	*+opt*	Con	*batch*
`fib`	2557	1329000	465	49010	5950	233	25280	2910	2690	232	7650
`laplace`	347	9070	298	5140	1550	20	950	210	210	90	1070
`matmul`	nonlinear		684	142830	15020	216	27390	2950	2270	432	10050
`mortgage`	nonlinear		482	89940	18200	2	810	750	1520	240	6390

difference except in the biggest examples.[10] We are currently working on adding indices to HAL CHRs.

Examination of both tables shows that the type class mechanism does not add substantial overhead to the use of constraint solvers: The overhead of type classes varies up to 3.5% (ignoring the 28% on very small times), and the average overhead is just 2%.

Note that this experiment is not meant to be an indication of the merits of the different approaches since, for building different solvers, each approach has its place.

Our second experiment compares the speed of the HAL interfaces defined in Section 5: the *naive* interface, the interface *constant* that keeps track of when `cfloat`s are constants and solves assignments and test in the interface itself, and the interface *datastructures* which builds data structures to handle functions calls, and only sends constraints at predicate calls. For the last two, we run the solver incrementally (solving after every constraint addition) and in batch mode (explicitly calling a `solve` predicate). For the last interface we also provide a version (`+opt`) which implements a simple type of partial evaluation by making use of Mercury to aggresively inline predicates and functions even across module boundaries. Finally, we compare against the ECLiPSe [9] interface (*ECL*) to the CPLEX solver, which also batches constraints.

The benchmarks are standard small linear arithmetic examples (see e.g. [3]). The table gives the number of constraints sent to CPLEX by each solver (Con), and execution times in milliseconds for 100 executions of the program. The last two benchmarks involve nonlinear constraints (not handled by CPLEX) if constants are not kept track of.

It is clear from Table 3 that the naive interface is impractical. Tracking constants and performing assignments and tests in the interface itself significantly improves speed. The move to using data structures to build linear expressions is clearly important in practice. Using this technique, the number of constraints passed to CPLEX for `mortgage` is reduced to just 2. Finally, batching is clearly worthwhile in these examples.

This experiment shows that the external solver interface for CPLEX is considerably faster than that provided by ECLiPSe and we believe that there is still

[10] Note that using bindings to represent `true` and `false` would result in a more efficient SICStus CHR Boolean solver, but the equivalent is not possible in HAL (at present).

considerable scope for improvement. Inlining of predicates and functions across module boundaries provides substantial improvement, but we believe that we can do even better by partially evaluating away many of the calls to solver interface and building the arguments to the calls to CPLEX at compile time if the constraint is known. Similarly, we would like to automatically perform "batching" by making HAL introduce satisfiability checks just before a choice point is created. An important lesson from the second experiment is that it is vital for a CLP language to allow easy experimentation with the interface to external solvers, since the choice of interface can make a crucial difference to performance. Our experience with HAL has been very positive in this regard.

References

1. F. Bueno, M. Garcia de la Banda, M. Hermenegildo, K. Marriott, G. Puebla, and P.J. Stuckey. A model for inter-module analysis and optimizing compilation. In *Procs of LOPSTR2000*, volume 2042 of *LNCS*, pages 86–102, 2001.
2. P. Codognet and D. Diaz. Boolean constraint solving using `clp(FD)`. In *Procs. of ILPS'1993*, pages 525–539. MIT Press, 1993.
3. B. Demoen, M. García de la Banda, W. Harvey, K. Marriott, and P.J. Stuckey. An overview of HAL. In *Procs. of PPCP'99*, LNCS, pages 174–188, 1999.
4. A.J. Fernández and B.C. Ruiz Jiménez. Una semántica operacional para CProlog. In *Proceedings of II Jornadas de Informática*, pages 21–30, 1996.
5. T. Frühwirth. CHR home page. www.informatik.uni-muenchen.de/~fruehwir/chr/.
6. T. Frühwirth. Theory and practice of constraint handling rules. *Journal of Logic Programming*, 37:95–138, 1998.
7. W. Harvey and P.J. Stuckey. Constraint representation for propagation. In *Procs. of PPCP'98*, LNCS, pages 235–249. Springer-Verlag, 1998.
8. C. Holzbaur, P.J. Stuckey, M. García de la Banda, and D. Jeffery. Optimizing compilation of constraint handling rules. In *Procs. of ICLP17*, LNCS, 2001.
9. IC PARC. ECLiPSe prolog home page. http://www.icparc.ic.ac.uk/eclipse/.
10. ILOG. CPLEX product page. http://www.ilog.com/products/cplex/.
11. Jack: Java constraint kit. http://www.fast.de/ mandel/jack/.
12. D. Jeffery, F. Henderson, and Z. Somogyi. Type classes in Mercury. Technical Report 98/13, University of Melbourne, Australia, 1998.
13. S. Kaes. Parametric overloading in polymorphic programming languages. In *ESOP'88 Programming Languages and Systems*, volume 300 of *LNCS*, pages 131–141, 1988.
14. K. Marriott and P.J. Stuckey. *Programming with Constraints: an Introduction*. MIT Press, 1998.
15. Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury: an efficient purely declarative logic programming language. *JLP*, 29:17–64, 1996.
16. M. Trick. mat.gsia.cmu.edu/COLOR/color.html.
17. P. Wadler and S. Blott. How to make *ad-hoc* polymorphism less *ad-hoc*. In *Proc. 16th ACM POPL*, pages 60–76, 1989.

Practical Aspects for a Working Compile Time Garbage Collection System for Mercury

Nancy Mazur[1], Peter Ross[2], Gerda Janssens[1], and Maurice Bruynooghe[1]

[1] Department of Computer Science, K.U.Leuven
Celestijnenlaan, 200A, B–3001 Heverlee, Belgium
{nancy,gerda,maurice}@cs.kuleuven.ac.be
[2] Mission Critical, Drève Richelle, 161, Bât. N
B–1410 Waterloo, Belgium
petdr@miscrit.be

Abstract. Compile-time garbage collection (CTGC) is still a very uncommon feature within compilers. In previous work we have developed a compile-time structure reuse system for Mercury, a logic programming language. This system indicates which datastructures can safely be reused at run-time. As preliminary experiments were promising, we have continued this work and have now a working and well performing near-to-ship CTGC-system built into the Melbourne Mercury Compiler (MMC).
In this paper we present the multiple design decisions leading to this system, we report the results of using CTGC for a set of benchmarks, including a real-world program, and finally we discuss further possible improvements. Benchmarks show substantial memory savings and a noticeable reduction in execution time.

1 Introduction

Modern programming languages typically limit the possibilities of the programmer to manage memory directly. In such cases allocation and deallocation is delegated to the run-time system and its garbage collector, at the expense of possible run-time overhead. Declarative languages go even further by prohibiting destructive updates. This increases the run-time overhead considerably: new datastructures are created instead of updating existing ones, hence garbage collection will be needed more often.

Special techniques have been developed to overcome this handicap and to improve the memory usage, both for logic programming languages [10,14,18] and functional languages [21,16]. Some of the approaches depend on a combination of special language constructs and analyses using unique objects [19,1,22], some are solely based on compiler analyses [13,16], and others combine it with special memory layout techniques [21]. In this work we develop a purely analysis based memory management system.

Mercury, a modern logic programming language with declarations [19] profiles itself as a general purpose programming language for large industrial projects.

P. Codognet (Ed.): ICLP 2001, LNCS 2237, pp. 105–119, 2001.

Memory requirements are therefore high. Hence we believe it is a useful research goal to develop a CTGC-system for this language. In addition, mastering it for Mercury should be a useful stepping stone for systems such as Ciao Prolog [12] (which has optional declarations and includes the impurities of Prolog) and HAL [8] (a Mercury-based constraint language).

The intention of the CTGC-system is to discover at compile-time when data is not referenced anymore, and how it can best be reused. Mulkers et al. [18] have developed an analysis for Prolog which detects when memory cells become available for reuse. This analysis was first adapted to languages with declarations [3] and then refined for use in the presence of program modules [15]. A first prototype implementation was made to measure the potential of the analysis for detecting dead memory cells. As the results of the prototype were promising [15], we have continued this work and implemented a full CTGC-system for the Melbourne Mercury Compiler (MMC), focusing on minimizing the memory usage of a program. In this paper we present the different design decisions that had to be taken to obtain noticeable memory savings, while remaining easy to implement within the MMC and with acceptable compilation overhead. A series of benchmarks are given, measuring not only the global effect of CTGC, but also the effect of the different decisions during the CTGC analysis.

After presenting some background in Section 2, we first solve the problem of deciding how to perform reuse once it is known which cells might die (Section 3). Section 4 presents low-level additions required to increase precision and speed, and obtain the first acceptable results for a set of benchmarks (Section 5). Using cell-caching (Section 6) more memory savings can be obtained. Finally improvements related to other work are suggested (Section 7), followed by a conclusion (Section 8).

2 Background

2.1 Mercury

Mercury [11] is a logic programming language with types, modes and determinism declarations. Its type system is based on a polymorphic many-sorted logic and its mode-system does not allow partially instantiated datastructures.

The analysis performed by our CTGC-system is at the level of the *High Level Data Structure* (HLDS) constructed by the MMC. Within this structure, predicates are *normalized*, i.e. all atoms appearing in the program have distinct variables as arguments, and all unifications $X = Y$ are explicited as one of (1) a test $X == Y$ (both are ground terms), (2) an assignment $X := Y$, (X is free, Y is ground) (3) a construction $X \Leftarrow f(Y_1, \ldots, Y_n)$ (X is free, all Y_i are ground), or (4) a deconstruction $X \Rightarrow f(Y_1, \ldots, Y_n)$ (X is ground, all Y_i are free) [11]. Within the HLDS, the atoms of a clause body are ordered such that the body is well moded. In the paper, we will use the explicit modes.

Just like in the HLDS we will use the notion of a *procedure*, i.e. a combination of one predicate with *one* mode, and thus talk about the analysis of a procedure.

2.2 General Structure of the CTGC-System

The CTGC-system consists of a data-flow analysis, followed by a reuse analysis and ended by a code generation pass (similar to [10]).

The data-flow analysis is performed to obtain structure-sharing information (expressed as possible aliases [3]) and to detect when heap cells become *dead* and are therefore *available for reuse*. It is based on abstract interpretation [2] using a so called *default call pattern* for each of the procedures to be analysed. This default call pattern makes minimal realistic assumptions: the inputs to a procedure are in no way aliased, and only the outputs will be used after the call to the procedure. The data-flow analysis requires a fixpoint computation to deal with recursive predicate definitions. For more details, see [3,15].

Next the reuse analysis decides which reuses are possible (see Section 2.5). Different versions can then be created for the different reuses detected. While the underlying concepts were already developed in [15], the pragmatics of our implementation are discussed in this paper.

Finally, low-level code corresponding to the detected reuses is generated.

As Mercury allows programming with modules, the CTGC-system processes each module independently. Interface files are used to allow analysis information (structure-sharing and reuse information) generated while processing one module to be used when processing other modules.

2.3 Data Representation

The purpose of the CTGC-system is to identify which objects on the heap, so called *datastructures*, become dead and can therefore be reused. In order to understand what these objects are, we will clarify the way typed terms are usually[1] represented in the MMC. Consider the following types:

```
:- type dir ---> north ; south ; east ; west.
:- type example ---> a(int, dir) ; b(example).
```

Terms of primitive types such as integers, chars, floats[2] and pointers to strings are represented as single machine words. Terms of types such as `dir`, in which every alternative is a constant are equivalent to enumerated types in other languages. Mercury represents them as consecutive integers starting from zero, and stores them in a single machine word. Terms of types such as `example` are stored on the heap. The pointer to the actual term on the heap is tagged [9]. This tag is used to indicate the function symbol of the term. Terms of types having more function symbols than a single tag can distinguish use secondary tags.

Figure 1 shows the representation of a variable `A` bound to `b(a(3,east))`. In this paper `ha1`, `hy1`,... denote heap cells, whereas `sa`, `sx`,... are registers or stack locations.

[1] The MMC compiles to different back-ends, the most common being ANSI-C. Higher-level back-ends, such as Java or .NET, use different low level representations, yet the theory of recycling heap cells remains the same.

[2] Depending on the word-size, these might have a boxed representation.

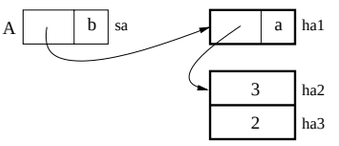


Fig. 1. `A = b(a(3,east))`.

```
:- pred convert1(example, example).
:- mode convert1(in, out) is semidet.
convert1(X,Y):- X => b(X1),
                X1 => a(A1, _),
                Y1 <= a(A1, north),
                Y <= b(Y1).
```

Fig. 2. Conversion-procedure.

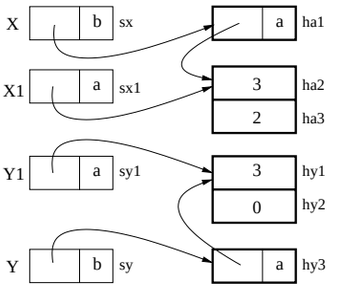


Fig. 3. No reuse

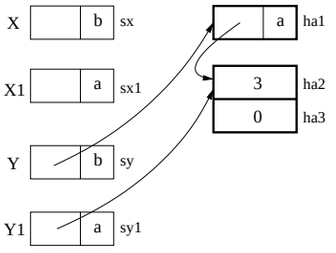


Fig. 4. Reuse

2.4 Data Reuse

Figure 3 shows the memory layout when calling `convert1(A, B)` (Fig. 2), where A is bound to `b(a(3,east)` (Fig. 1).

After deconstructing the input, new heap cells (`hy1`, `hy2` and `hy3`) are allocated to create `Y`, and the content of `X` is partially copied into those cells. If it can be shown at compile-time that after this procedure call the term pointed at by `X` will not be referenced during the rest of the program (thus becoming available for reuse), then the deconstruction statements perform the last access ever to the concerned heap cells (`ha1`, `ha2`, `ha3`) after which they become garbage, and can be (re)used for `Y` (Fig. 4, the contents of `sx` and `sx1` are no longer relevant).

The optimization could go further and detect that this reuse only requires the update of one heap cell, namely `ha3`. Yet currently we mainly focus on the memory usage of a program, execution time being only of indirect importance. Therefore we do not try to optimize the number of field updates in the presence of reuse. See also Section 7.

2.5 Types of Reuse

During reuse analysis we make a distinction between different kinds of reuses [15]. The procedure shown in Fig. 2 has *direct reuse* (of `X`): it might contain the last reference to the cells of `X` which can then be reused by `Y`. However, the reuse is *conditional*: if the caller's environment does not correspond to the default call pattern (e.g. keeping a reference to one of the cells marked for reuse), reuse is

```
:- type field1 ---> field1(int, int, int).
:- type field2 ---> field2(int, int).
:- type list(T) ---> [] ; [T | list(T)].
:- pred convert2(list(field1), list(field2)).
:- mode convert2(in, out) is det.

convert2(List0, List):-
    ( % switch on List0
        List0 => [], List  <= []
    ;
        List0 => [Field1 | Rest0],      % (d1)
        Field1 => field1(A, B, _C),     % (d2)
        Field2 <= field2(A, B),         % (c1)
        convert2(Rest0, Rest),
        List <= [Field2 | Rest]         % (c2)
    ).
```

Fig. 5. Converting lists.

not allowed. As this imposes very harsh restrictions on the reuse possibilities, we introduced the notion of *reuse conditions* which express the minimal conditions a call pattern has to meet so that reuse is safe. These conditions are expressed in terms of the variables involved (here `X`). If the reuse is independent of the calling environment (`X` being a local variable), then we have *unconditional* reuse.

Given the reuse conditions, the next step of the reuse analysis is to check for *indirect reuses*. Consider the following procedure:

```
:- pred generate(example).
:- mode generate(out) is semidet.
generate(Y):- generate_2(X), convert1(X, Y).
```

Assuming the default call pattern for `generate`, the call to `convert1` meets the condition that after that call `X` will not be used anymore. Hence, a reuse-version of `convert1` can be called and we say that `generate` has *indirect reuse*. Moreover, `X` is a local variable, so reusing it will always be safe as it is independent of the call pattern to `generate`. This is an example of *unconditional* (indirect) reuse. If `X` would have been an input variable to `generate`, the indirect reuse would be conditional and additional reuse conditions would be formulated.

3 A Working Reuse Decision Approach

Consider the predicate in Fig. 5 which converts a list of data-elements into a new list of data-elements. While the data-flow analysis spots the datastructures that can potentially be reused, it is up to the reuse analysis to select those reuses (direct and indirect) that yield the most interesting saving w.r.t. memory usage (and indirectly execution time).

3.1 Deciding Direct Reuse

A first restriction we impose is to limit reuses to *local reuses*, i.e. a dead cell can only be reused in the same procedure as where it is last accessed (deconstructed). In Section 6 we discuss techniques of how to lift this restriction. Furthermore, we consider that dead structures can only be reused by at most one new structure. Using the terminology of Debray [7], we limit ourselves to the *simple* reuse problem. It is not difficult to remove this limitation, but it makes the reuse decisions more complex. We plan to lift this restriction in the future.

The data-flow analysis of the example identifies the deconstructed datastructures (at `d1`, resp. `d2`) as available for reuse. The procedure also contains two constructions (`c1` and `c2`) where the memory from the dead cells could be reused.

Each of the combinations yields an acceptable reuse-scheme. Yet, which one is the most interesting? It has been shown that this problem [7] can be reformulated as an instance of the maximum weight matching problem for a weighted bipartite graph. However for simplicity of implementation we have reduced this general matching problem to two orthogonal decisions: imposing constraints on the allowed reuses, and using simple strategies to select amongst different candidates for reuse. We will discuss each of these.

Constraints on allowed reuses. Constraints allow one to express common characteristics between the dead and the newly constructed cell and reflect the restrictions which can be imposed by the back-end to which a Mercury program is compiled.

We have implemented the following constraints:

- Almost matching arities. This constraint expresses the intuition that it can be worthwhile to reuse a dead cell, even if not all memory-words are reused. This is indeed interesting if it can be guaranteed that the superfluous words will be collected by the run-time garbage collector within a reasonable delay. In our example, allowing a difference of size one allows `c1` and `c2` to reuse the memory available from either `d1` or `d2`.
- Matching arities. If the run-time system is not powerful enough to be used with *almost matching arities*, then a more restrictive constraint can be used: only allow reuse between constructors having the same arity. This means that in our example only `d1` can be reused (by either `c1` or `c2`).
- Label-preserving. Using the Java or .NET back-end, it is not possible to change the type of run-time objects, therefore reuse is only allowed if the dead and new cell have the same constructor (label). For the example this means that the cell from `d1` can only be reused in `c2`.

Selection strategies. When a cell can reuse different dead cells, a choice has to be made (e.g. `c1` can either reuse the cell available from `d1` or `d2`). Some choices yield better results than others. We have experimented with two simple strategies:

- Lifo. Traverse the body of the procedure and assign the reuses using a last-in-first-out selection strategy. This means that when a choice is left for a

given construction, choose the cell which died most recently. The intuition is that after deconstructing a variable, it is very likely that a new similar cell will be constructed in the same context.
e.g. If `c1` is allowed to reuse the cells from `d1` or `d2`, then according to this strategy, `Field1` will be reused for constructing `Field2` and `List0` for `List`.

- Random. The intuition behind the lifo-strategy might not always be true, for example in the presence of a disjunction[3]. Therefore we have added a simple strategy which randomly selects the dead cell amongst all the candidates.

3.2 Deciding Indirect Reuse

In order to decide whether a call to a procedure can be substituted by a call to a reuse version of that procedure, we must be sure that such substitution is safe. This is tested by checking the reuse-conditions (under the assumption of a default call pattern). If it is safe to call the reuse-version we have to decide whether we will do so or not.

Here we have decided for simplicity by always calling the reuse-version of a procedure if it is safe to do so. In Section 7 we discuss the drawbacks and suggest a possible better solution.

Suppose that for our previous example we would only allow the reuse of the list-cells (`d1` by `c2`). Such reuse is conditional: the list cell only dies iff it is not needed within the caller's context. This condition has to be checked for the recursive call. Under the default call pattern (see Section 2.4) `Rest0` is dead at the moment of the recursive call, hence the condition is satisfied, and the recursive call can safely be substituted by a call to its reuse version[4].

3.3 Splitting into Different Versions

Once the possible direct and indirect reuses have been decided, there is one remaining decision left: how many versions of a given procedure should be created? In our example, we might have detected three reuses: `List0` reused by `List`, `Field1` by `Field2`, and the indirect reuse (the recursive call to the reuse version). We can generate 4 interesting versions of the initial procedure: a version with no reuse, a version reusing only `List0`, a version reusing `Field1` and a version reusing both (where the reuse versions also include the recursive reuse call). In general, for a procedure with n possible direct reuses, 2^n interesting versions can be created. In our implementation we limit the number of versions to at most two: a version which imposes no conditions on the caller (containing all possible unconditional reuses), and a version containing all detected reuses. In Section 7 we briefly discuss other possibilities.

[3] e.g. `X => f(..), ( ... Y <= f(..) ; ... ), Z <= f(..)`: as the first branch of the disjunction might not always be executed, it is more interesting to allow `Z` to reuse `X` than `Y`.

[4] Note that this indirect reuse is in itself conditional: it can only be allowed if the list-cells of `Rest0` are not needed in the callers context.

4 Low Level Additions

Given the previous decisions, a first CTGC-system was implemented. Although good results were obtained for small programs (e.g. naive-reverse), we ran into problems when analysing large ones:

- imprecision in the alias analysis had the effect that relatively few cells were recognized as dead.
- the number of aliases collected within a procedure became huge. This slowed down the operations manipulating them and the CTGC process became too time consuming.

4.1 Enhancing the Aliasing Precision

The underlying analysis for deriving alias-information uses the concept of `top` which expresses that all data parts might be aliased. This is a safe abstraction in the case of total lack of knowledge about the possible existing aliases at some program point. Once generated, this lack of information propagates rapidly as all primitive operations manipulating it yield `top` as well.

Such a `top` is generated in the presence of language constructs with which the analysis cannot cope yet. These are procedures defined in terms of foreign code (c, C++), higher-order calls and typeclasses. It is also generated for procedures which are defined in other modules that have not yet been analysed and for which no interface files have been generated yet.

To obtain a usable CTGC-system, techniques were needed to limit the creation and propagation of `top`. In our implementation, three techniques are used:

1. *Using heuristics.* Based on the type- and mode- declaration of a procedure, one can derive whether it can create additional aliases or not, without looking at the procedure's body. This is the case when a procedure uses unique objects (declared `di` or `uo` [11]), or only has unique output variables[5] or when the non-unique output arguments are of a type for which sharing is not possible (integers, enums, chars, etc.). In all these cases, it is safe to conclude that the procedure will not introduce new aliases.
2. *Manual aliasing annotation for foreign code.* Important parts of the Mercury standard library consist of procedures which are defined in terms of foreign code. With the intention to be used mainly in this standard library, we have extended the Mercury language such that foreign code can be manually annotated with aliasing-information.
3. *Manual iteration for mutual dependent modules.* The current compilation-scheme of Mercury is not yet able to cope with mutual dependent modules. Consider a module A in which some procedures are expressed in terms of procedures declared in a module B, and vice versa. The normal compilation scheme is to compile one of the files, and then the other one. In the presence

[5] A procedure call cannot create additional aliases between input variables as they must be ground at the moment when the procedure is called.

of an optimizing compiler this is not enough. At the moment the first module is compiled, nothing is known from the second one, yielding bad precision for the first one. This bad precision will propagate further to the second file as the second file relies on the first one. Bueno et al. [5] propose a new compilation scheme which is able to handle these cases. As this requires quite some work, we make a work around by allowing manually controlled incremental compilation.

4.2 Making Compilation Faster: Widening the Aliasing

While it is interesting to have more precise aliasing information than simply `top`, having more aliases also slows down the system. Now one can argue that speed is not a major requirement of a CTGC-system as it is primarely intended to be used only at the final compilation phase of a program, but even for our benchmarks we were not ready to wait hours for a module to compile. Therefore, in order to produce a usable CTGC-system we have added a widening operator [6] which acts upon the aliases produced[6].

During the data-flow analysis, a datastructure is represented by its full path down the term it is part of. Such a path is a concatenation of selectors which selects the functor and the exact argument position in the functor[7]. Aliases are expressed as pairs of datastructures.

To illustrate this, let us consider the following definition of a tree type:

```
:- type tree ---> e ; two(int,tree,tree)
                  ; three(int,int,tree,tree,tree).
```

After the construction `V <= three(2,3,two(0,e,e),A,A)` (where `A` is a variable bound to another tree-term), the path $(three, 3) \cdot (two, 1)$ selects in `V` the zero-integer. The path $(three, 3)$ selects in `V` the whole datastructure corresponding to the first subtree (namely `two(0,e,e)`). In `V`, the positions corresponding with the paths $(three, 4)$ and $(three, 5)$ are aliased.

For the aliasing information, we introduced *type widening* that consists of replacing a full path of normal selectors by one selector, a so called *type selector*. The meaning of a *type selector* is as follows: instead of selecting one specific subterm of a term, it will select all the subterms which have the type expressed by the selector. In our example, the paths $(three, 1)$, $(three, 2)$, and $(three, 3) \cdot (two, 1)$ all select integer elements of `V`. With type widening, all these selectors are reduced to the selector (int), i.e. the type of the subterms which they select. The alias in `V` (ie. between $(three, 4)$ and $(three, 5)$) becomes an alias between $(tree)$ and $(tree)$, hence expressing that all subtrees of `V` might be aliased. If other aliases between subtrees of `V` exist, then they will all be replaced by this one single alias, hence making the overall size of the set of aliases smaller.

[6] This widening operator can be enabled on a per-module base. The user can also specify the threshold at which widening should be performed: e.g. only widen if the size of the set of aliases exceeds 1000.

[7] Infinite paths are avoided by simplifying full type trees to type graphs. This is beyond the scope of this paper.

This widening leads to a considerable speed-up of the CTGC-system (compilation of some modules taking almost one hour was now reduced to less than a minute). Our results suggest that the overall precision remains sufficient in order to detect the expected reuses for our benchmarks.

5 First Results

We have evaluated the effectiveness of our CTGC-system by comparing memory usage and measuring compilation times. We have used toy benchmarks and one real-life program. All the experiments were run on an Intel-Pentium III (600Mhz) with 256MB RAM, using Debian Linux 2.3.99, under a usual workload. The CTGC-system was integrated into version 0.9.1 of the MMC. The reported memory information is obtained using the MMC memory profiler. This profiler counts the total number of memory words that are allocated on the heap[8]. The timings are averages of 10 runs each time. All the benchmarks are compiled using a non-optimized Mercury standard library w.r.t. memory usage (hence no reuse in the library predicates[9]). This allows us to focus on the reuse occurring in the actual code of the benchmarks.

The toy benchmarks comprise *nrev* (naive reverse of a list of 3000 integers), *qsort* (quick sort of a sorted list of 10000 integers), and *argo_cnters* (a benchmark counting various properties of a file, also used in [15]). Table 1 shows the results. These are independent of the CTGC configuration used, as they all yield the same results here. For each of the benchmarks every possible reuse is detected, yielding the expected savings in memory usage and execution time.

Table 1. Toy benchmarks. C = compilation time. M = number of allocated words. R = execution time. m = relative reduction in memory usage.

	No Reuse			Reuse			
module	C (sec)	M (Word)	R (sec)	C (sec)	M (Word)	m (%)	R (sec)
nrev	1.49	9M	1.51	11.79	6k	-99.9	0.32
qsort	1.40	50M	36.63	11.29	20k	-99.9	27.22
argo_cnters	4.53	3.00M	0.35	16.38	2.60M	-13.3	0.32

Next to small benchmarks, we found it important to evaluate the system on a real-life program, where the different constraints and strategies do make a difference. The program we used is a ray tracer program developed for the ICFP'2000 programming contest [17] where it ended up fourth. This program transforms a given scene description into a rendered image. It is a CPU- and memory-intensive process, and therefore an ideal candidate for our CTGC-system to be tested on. A complete description of this program can be found at [20].

[8] Note that this count is independent of any run-time garbage collection.

[9] Normally, a Mercury system with CTGC would also have the library modules compiled with CTGC in the same way as user modules.

The program consists of 20 modules (5700 lines of code), containing mostly deterministic predicates. All modules could be compiled without widening, except for one: *peephole*. This module manipulates complex constructors and generates up to 11K aliases. Without type-widening, the compilation of *peephole* takes 160 minutes. With type-widening (at 500 aliases), it only takes 40 seconds. The compilation of the whole program with CTGC (and widening) takes 5 minutes, compared to 1 minute for a normal compilation. As some of these modules depend on each other, the technique of manually iterating the compilation was used to obtain better results. For this benchmark, the compilation had to be repeated 3 times to reach a fixpoint (for a total time of 15 minutes). Each time every module was recompiled. In a smart compilation environment, most of the recompilations could be avoided.

To measure the effects of the different constraints and strategies we have compiled the ray tracer with different CTGC-configurations. The first row of Table 2 shows the number of memory words and the execution time (in seconds) needed to render a set of 27 different scene descriptions (ranging from simple scenes, to more complex ones) using a version of the ray tracer without CTGC. Rows 1 to 9 show the relative memory usage and execution time of ray tracers compiled using different CTGC-configurations for the same set of scene descriptions:

- Using the matching arities (*match*) or label-preserving (*same cons*) constraints, up to 24% memory can be saved globally. For some scene descriptions, this can go up to 30%. There is also a noticeable speedup (14%).
- Using almost matching arities within a distance of one (*within 1*) or two (*within 2*), much less memory is saved (only 10%) with hardly any speedup. The bad memory usage is not surprising as none of the selection strategies takes into account the correspondance of the arities between a new cell and the available dead cells. The bad timings are also explicable: with non-matching arities, reuse leaves space-leaks which cannot immediately be detected by the current run-time garbage collector, hence the garbage collector will be called more often. Improvements to the garbage collector are required.
- Globally, using the *random* selection strategy yields slightly worse results than *lifo*. For some scene descriptions though, results are better, but without spectacular differences.
- Row 9 shows the results of a ray tracer compiled using a version of the Mercury standard library *with* CTGC. There is hardly any difference with Row 1, where libraries were used without CTGC. This is due to the fact that the ray tracer makes a limited use of these libraries.

Finally, a version of the ray tracer was built without type-widening (*lifo* and *matching arities*). Compared to row 1 in Table 2 the overall memory usage difference is less than 1%. The execution times are comparable.

Table 2. ICFP-ray tracer using different CTGC-configurations.

	Configuration			Memory (kWord)	(%)	Time (sec)	(%)
0	no CTGC			1024795.51	-	362.31	-
1	lifo	match		776707.92	-24.21	311.85	-13.93
2	lifo	same cons		791742.06	-22.74	313.57	-13.45
3	lifo	within 1		916642.90	-10.55	361.84	-0.13
4	lifo	within 2		917847.97	-10.44	359.90	-0.67
5	random	match		780838.58	-23.81	310.75	-14.23
6	random	same cons		795872.67	-22.34	312.70	-13.69
7	random	within 1		920764.26	-10.15	359.14	-0.87
8	random	within 2		921969.35	-10.03	355.08	-2.00
9	lifo	match	libs	775607.04	-24.32	320.32	-11.59
10	lifo	match	cc	513901.37	**-49.85**	301.66	-16.74
11	lifo	same cons	cc	542626.80	**-47.05**	304.20	-16.04
12	lifo	within 1	cc	845603.55	-17.49	375.79	3.72
13	lifo	within 2	cc	864722.90	-15.62	370.49	2.26
14	random	match	cc	518032.04	**-49.45**	299.45	-17.35
15	random	same cons	cc	546757.48	**-46.65**	302.79	-16.43
16	random	within 1	cc	849724.90	-17.08	363.90	0.44
17	random	within 2	cc	868844.29	-15.22	391.68	8.11

6 Non-local Reuse: Cell Cache

Currently we have assumed that all dying datastructures must be reused locally, i.e. within the same procedure in which they die. Hence quite some interesting possibilities of reuse could be missed.

We see three ways to achieve non-local reuses as well. The first and the most difficult is to extend the data-flow analysis to handle non-local reuse. The analysis would have to propagate possible dead cells and thus become quite complex. It would also require intensive changes in the internal calling convention of procedures within the MMC as the address of the cells to be reused would have to be passed between procedures. The second approach is to combine reuse analysis with inlining in such a way that the cell death and subsequent reuse end up in the same procedure. The third approach, which is the one we implemented, is to *cache dead cells*. Whenever a cell dies unconditionally and cannot be reused locally, we mark it as *cacheable*. At runtime the address of the cell as well as its size will be recorded in a cache (or free list). Before each memory allocation the runtime system will first check the cell cache to see if a cell of the correct size is available and use that cell instead of allocating a new cell. This operation increases the time taken to allocate a memory cell in the case of the cell cache being empty, and hence should only be a win if the cell cache occupancy rate is high. It also avoids new allocations so the overall cost of the runtime garbage collection system should go down due to smaller heap sizes and less frequent need for garbage collection.

The *cc*-entries of Table 2 (Rows 10-17) show the results of CTGC-configurations combined with the cell cache technique. Compared to the basic CTGC-configurations, cell caching always increases memory savings, going up to 49% (for some scenes even 70%). In the case of label-preserving or matching arities constraints, execution time drops slightly. On the other hand, using almost matching arities combined with cell caching increases the execution time.

7 Further Improvements

In the near future, we intend to explore a number of improvements to our system. First, for some procedures, several possibilities of reuse are discovered, each one imposing its own reuse conditions. Taken together, these reuse conditions are too restrictive on the caller, hence hardly any calling environment is able to satisfy them, and no reuse is performed at all. A *top-down call-dependent version splitting pass* could aid in generating more useful reuse-versions of procedures, and avoid the generation of the useless ones.

A second problem is the too absorbant effect of the notion of `top` currently used in the alias information. Once `top` is encountered, it propagates all throughout the remainder of the code. Instead of `top`, we could use *topmost substitutions* [4]: e.g. generating all possible combinations of aliases between the arguments of a called predicate, based on the types of these arguments, either explicitly or in a more compact form (using type-selectors or keeping sets of variables, stating that these variables might be aliased to each other in any possible way).

In this paper we mainly focussed on memory savings, reasoning that saving memory implies less garbage collection, hence diminishes the execution time. If execution time is of primary concern, than more sophisticated reuse strategies will be needed. In the near future we will adopt the use of weighted graphs [7] where the weights can be adjusted for minimizing memory usage or execution time (taking into account the fields that do not need to be updated). We will also consider splitting dead cells and reusing them for different new cells.

In [10] the focus on execution time is even greater, trying to discover almost every field not requiring an update, going even beyond the boundaries of single procedures. This is indeed important in Prolog, where the determinism of procedures is not necessarely known at analysis time, and where given the underlying data-flow analysis, each cell update requires extra care in the case the value has to be reset upon backtracking. In Mercury, where determinism *is* known at compilation time, and where the analysis explicitly takes into account backtracking, this is not a major issue. Therefore, it is not our immediate intention to try to avoid every possible cell update.

8 Conclusion

This paper describes a complete working compile-time garbage collection system for Mercury, a logic programming language with declarations. The system

consists of three passes: data-flow analysis, reuse decision, and low level code generation. The data-flow analysis based on [15] detects which cells become available for reuse. This paper presents easy implementable restrictions, constraints and strategies for selecting realistic reuses. In order to obtain a workable CTGC-system, low level improvements were introduced.

A major contribution of this work is the integration of the CTGC system in the Melbourne Mercury Compiler and its evaluation. Some small benchmarks were used, but also one real-life complex program, a ray tracer. Average global memory savings of up to 49% were obtained, with a speedup of up to 17%. It would be interesting to compare these results with the total potential of reuse within the program. This total potential could be approximated using the techniques used in our first prototype [15] to predict the amount of reuse.

Beside the proposed improvements the system could also be adapted to handle higher order calls and type classes properly (instead of generating `top` aliasing, and not allowing reuse). Yet given the fact that many higher order calls are specialized away by the compiler, we currently do not believe that the overhead needed to deal with these language constructs is worthwhile.

References

1. Y. Bekkers and P. Tarau. Monadic constructs for logic programming. In J. Lloyd, editor, *Proceedings of the International Symposium on Logic Programming*, pages 51–65, Cambridge, Dec. 4–7 1995. MIT Press.
2. M. Bruynooghe. A practical framework for the abstract interpretation of logic programs. *Journal of Logic Programming*, 10(2):91–124, Feb. 1991.
3. M. Bruynooghe, G. Janssens, and A. Kågedal. Live-structure analysis for logic programming languages with declarations. In L. Naish, editor, *Proceedings of the Fourteenth International Conference on Logic Programming (ICLP'97)*, pages 33–47, Leuven, Belgium, 1997. MIT Press.
4. F. Bueno, D. Cabeza, M. Hermenegildo, and G. Puebla. Global Analysis of Standard Prolog Programs. In *European Symposium on Programming*, number 1058 in LNCS, pages 108–124, Sweden, April 1996. Springer-Verlag.
5. F. Bueno, M. García de la Banda, M. Hermenegildo, K. Marriott, G. Puebla, and P. J. Stuckey. A model for inter-module analysis and optimizing compilation. In *Tenth International Workshop on Logic-based Program Synthesis and Transformation*, London, UK, 2000. to appear.
6. P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In M. Bruynooghe and M. Wirsing, editors, *Proceedings of the Fourth International Symposium on Programming Language Implementation and Logic Programming*, pages 269–295, Leuven, Belgium, 1992. LNCS 631, Springer-Verlag.
7. S. K. Debray. On copy avoidance in single assignment languages. In D. S. Warren, editor, *Proceedings of the Tenth International Conference on Logic Programming*, pages 393–407, Budapest, Hungary, 1993. The MIT Press.
8. B. Demoen, M. García de la Banda, W. Harvey, K. Marriott, and P. J. Stuckey. An overview of HAL. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming*, pages 174–188, Virginia, USA, October 1999. Springer Verlag.

9. T. Dowd, Z. Somogyi, F. Henderson, T. Conway, and D. Jeffery. Run Time Type Information in Mercury. In *Principles and Practice of Declarative Programming*, pages 224–243, 1999.
10. G. Gudjónsson and W. H. Winsborough. Compile-time memory reuse in logic programming languages through update in place. *ACM Transactions on Programming Languages and Systems*, 21(3):430–501, May 1999.
11. F. Henderson, T. Conway, Z. Somogyi, and D. Jeffery. The Mercury language reference manual. Technical Report 96/10, Dept. of Computer Science, University of Melbourne, February 1996.
12. M. Hermenegildo, F. Bueno, G. Puebla, and P. López. Program Analysis, Debugging and Optimization Using the Ciao System Preprocessor. In D. D. Schreye, editor, *1999 International Conference on Logic Programming*, pages 52–66, Cambridge, MA, December 1999. MIT Press.
13. A. Kågedal and S. Debray. A practical approach to structure reuse of arrays in single assignment languages. In L. Naish, editor, *Proceedings of the 14th International Conference on Logic Programming*, pages 18–32, Cambridge, July 8–11 1997. MIT Press.
14. F. Kluźniak. Compile-time garbage collection for ground Prolog. In R. A. Kowalski and K. A. Bowen, editors, *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 1490–1505, Seattle, 1988. MIT Press, Cambridge.
15. N. Mazur, G. Janssens, and M. Bruynooghe. A module based analysis for memory reuse in Mercury. In J. Lloyd, V. Dahl, U. Furbach, M. Kerber, K.-K. Lau, C. Palamidessi, L. Moniz Pereira, Y. Sagiv, and P. J. Stuckey, editors, *Computational Logic - CL 2000, First International Conference*, London, UK, July 2000, Proceedings, volume 1861 of *Lecture Notes in Artificial Intelligence*, pages 1255–1269. Springer-Verlag, 2000.
16. M. Mohnen. Optimising the Memory Management of Higher–Order Functional Programs. Technical Report AIB-97-13, RWTH Aachen, 1997. PhD Thesis.
17. G. Morrisett and J. Reppy. The third annual ICFP programming contest. In Conjunction with the 2000 International Conference on Functional Programming, http://www.cs.cornell.edu/icfp/, 2000.
18. A. Mulkers, W. Winsborough, and M. Bruynooghe. Live-structure dataow analysis for Prolog. *ACM Transactions on Programming Languages and Systems*, 16(2):205–258, Mar. 1994.
19. Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. The Journal of Logic Programming, 29(1–3):17–64, October-December 1996.
20. The Mercury Team. ICFP 2000: The merry mercurians. Description of the Mercury entry to the ICFP'2000 programming contest, http://www.mercury.cs.mu.oz.au/information/events/icfp2000.html.
21. M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.
22. P. Wadler. The essence of functional programming. In *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–14, Albequerque, New Mexico, Jan. 1992.

Positive Boolean Functions as Multiheaded Clauses

Jacob M. Howe*,[1] and Andy King[2]

[1] Department of Computing, City University, London, UK
[2] Computing Laboratory, University of Kent, Canterbury, UK
{j.m.howe, a.m.king}@ukc.ac.uk

Abstract. Boolean functions are ubiquitous in the analysis of (constraint) logic programs. The domain of positive Boolean functions, Pos, has been used for expressing, for example, groundness, finiteness and sharing dependencies. The performance of an analyser based on Boolean functions is critically dependent on the way in which the functions are represented. This paper discusses multiheaded clauses as a representation of positive Boolean functions. The domain operations for multiheaded clauses are conceptually simple and can be implemented straightforwardly in Prolog. Moreover these operations generalise those for the less algorithmically complex operations of propositional Horn clauses, leading to naturally stratified algorithms. The multiheaded clause representation is used to build a pos-based groundness analyser. The analyser performs surprisingly well and scales smoothly, not requiring widening to analyse any program in the benchmark suite.
Keywords. Abstract interpretation, (constraint) logic programs, Boolean functions, groundness analysis.

1 Introduction

Boolean functions play an important role in the practice of static analysis. Many analyses are couched in terms of Boolean functions, and manipulation of these functions is crucial to the performance of any implementation. In particular, positive Boolean functions have been applied to the analysis of logic programs for properties such as groundness, rigidity [15], finiteness [3] and sharing [8]. This paper advocates representing positive Boolean functions as multiheaded clauses and argues that Prolog is well suited to their manipulation.

The choice of abstract domain for a particular application involves the striking of a balance between efficiency and precision. The various properties tracked using positive Boolean functions give rise in practice to different forms of Boolean function. Hence, in some applications, restricting to a more computationally tractable subclass of Pos can have a significant impact on precision (for example, goal-independent analysis of library code), whilst in others little precision is lost (for example, goal-dependent groundness analysis). Elsewhere, the authors have

* Work supported by EPSRC Grant GR/MO8769.

P. Codognet (Ed.): ICLP 2001, LNCS 2237, pp. 120–134, 2001.

discussed various subclasses of Pos and their computational properties [17,19]. Here, with an eye to a wider range of applications, the authors adapt techniques from these subclasses to Pos.

Traditionally, Boolean function manipulation has been performed using binary decision diagrams (BDDs). Groundness analysis is one of the most important topics in the static analysis of (constraint) logic programs and from a logic programming point of view this analysis is the most practical test of Boolean function manipulation. BDD-based analysers have consistently outperformed those based on other representations of Boolean functions [1,2,10,24] for groundness analysis, but there has been a continuous stream of work on representations amenable to Prolog implementation [7], in particular for the subclass of definite positive functions, Def [12,13,19]. The majority of these implementations, included those based on BDDs, require widening to analyse large benchmarks.

The Def-based groundness analyser described in [19] does not require widening and was designed so that the most frequently called domain operations are the most lightweight. The same design methodology suggests that a Pos-based analyser should represent Boolean functions as conjunctions of multiheaded clauses. In fact, in [1] (reduced) conjunctive normal form, (R)CNF, was investigated, and "performed reasonably well", but was ultimately rejected since BDDs performed 40% faster and, in C (their implementation language), conjunctive normal form is no easier to code than BDDs. Surprisingly, conjunctive normal forms have not been considered since. This paper revisits clausal representations of Pos since, in Prolog, clausal representations are much easier to code than BDDs and following the methodology of [19] the clausal representation lends itself to efficient implementation based on entailment checking.

The importance of the choice of representation is clearly illustrated by the subtle difference between multiheaded clauses and RCNF. The RCNF representation is reduced in the sense that no clause subsumes another. This reduction makes meet for RCNF quadratic in the size of the representation. The multiheaded clause representation may contain redundant clauses, enabling meet to be constant time. This is an important issue for performance since meet is by far the most frequently applied operation. Neither multiheaded clause nor RCNF representations are in a canonical form, therefore equivalence cannot be detected by straightforward syntactic identity. In [1] equivalence for RCNF is determined by computing the dual Blake canonical form of the formulae and then testing for syntactic identity. The dual Blake canonical form may be exponentially larger than the RCNF representation and must always be completely computed. Therefore the method is not amenable to filtering through lower complexity algorithms. Logical entailment, rather than syntactic equivalence, is more flexible. In practice, entailment of formulae can often be detected using an incomplete low complexity algorithm. Using such a check, many calls to the worst case algorithm can be filtered out. It is this stratified use of entailment checking that enables an analyser based on multiheaded clauses to scale surprisingly well. Speed is achieved by exploiting Prolog technology – by using a nonground rep-

resentation entailment checking can be implemented efficiently using renaming and block declarations, whilst meet reduces to list concatenation implemented using difference lists. The major themes and contributions of this work are:

- Pos functions can be naturally expressed as multiheaded clauses, which are particularly straightforward to understand, manipulate and code.
- The entailment checking algorithm (which is potentially exponential in the number of variables) is stratified so that checks for naturally occurring subclasses of formulae take quadratic time (in the size of the formulae); in particular the forward chaining algorithm for propositional Horn clauses is subsumed.
- The domain operations for multiheaded clauses may be coded succinctly and efficiently in Prolog, resulting in fast Pos-based goal-dependent and goal-independent groundness analysers which do not require widening for any program in the benchmark suite.
- If widening is required, the representation may be simply and naturally widened to Def or to the simpler domain EPos.
- The analysers again demonstrate the value of a principled approach to the design of a static analysis.
- An experimental evaluation of the analysers is given illustrating that a clausal representation of Pos coded in Prolog gives performances comparable to BDD representations coded in C.

The rest of this paper is structured as follows: Section 2 introduces the necessary technical background material. Section 3 details multiheaded clauses. Section 4 gives algorithms for the abstract operations of Pos represented as multiheaded clauses. Section 5 describes Pos-based groundness analysers implemented with Boolean functions represented as multiheaded clauses. Section 6 gives an experimental evaluation of these analysers. Section 7 reviews related work and Section 8 concludes.

2 Preliminaries

A Boolean function is a function $f : \mathsf{Bool}^n \to \mathsf{Bool}$ where $n \geq 0$. Let V denote a denumerable universe of variables. A Boolean function can be represented by a propositional formula over $X \subseteq V$ where $|X| = n$. The set of propositional formulae over X is denoted by Bool_X. Throughout this paper, Boolean functions and propositional formulae are used interchangeably without worrying about the distinction. The convention of identifying a truth assignment with the set of variables M that it maps to *true* is also followed. Specifically, a map $\psi_X(M) : \wp(X) \to \mathsf{Bool}_X$ is introduced defined by: $\psi_X(M) = (\wedge M) \wedge \neg(\vee(X \setminus M))$. In addition, the formula $\wedge Y$ is often abbreviated as Y.

Definition 1. The map $model_X : \mathsf{Bool}_X \to \wp(\wp(X))$ is defined by: $model_X(f) = \{M \subseteq X \mid \psi_X(M) \models f\}$. Also, $countermodel_X : \mathsf{Bool}_X \to \wp(\wp(X))$ is defined by: $countermodel_X(f) = \wp(\wp(X)) \setminus model_X(f)$. Observe that $model_X$ is bijective, hence $model_X^{-1} : \wp(\wp(X)) \to \mathsf{Bool}_X$ is well defined.

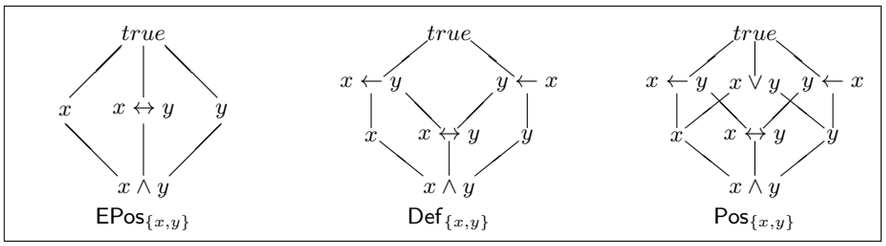


Fig. 1. Hasse diagrams

Example 1. If $X = \{x, y\}$, then the function $\{\langle true, true\rangle \mapsto true, \langle true, false\rangle \mapsto false, \langle false, true\rangle \mapsto false, \langle false, false\rangle \mapsto false\}$ can be represented by the formula $x \wedge y$. Also, $model_X(x \wedge y) = \{\{x, y\}\}$ and $model_X(x \vee y) = \{\{x\}, \{y\}, \{x, y\}\}$.

The focus of this paper is on the use of subclasses of Bool_X in tracing dependencies. These subclasses are defined below:

Definition 2. A function f is positive iff $X \in model_X(f)$. Pos_X is the set of positive Boolean functions over X. A function f is definite iff $M \cap M' \in model_X(f)$ for all $M, M' \in model_X(f)$. Def_X is the set of positive functions over X that are definite. A function f is GE iff f is definite positive and for all $M, M' \in model_{var(f)}(f)$, $|M \backslash M'| \neq 1$. EPos_X is the set of GE functions over X.

Note that $\mathsf{EPos}_X \subseteq \mathsf{Def}_X \subseteq \mathsf{Pos}_X$. Also notice that $\mathsf{EPos}_X = \{\wedge F \mid F \subseteq X \cup E_X\}$, where $E_X = \{x \leftrightarrow y \mid x, y \in X\}$.

Example 2. Suppose $X = \{x, y, z\}$ and consider the following table, which states, for some Boolean functions, whether they are in EPos_X, Def_X or Pos_X and also gives $model_X$.

f	EPos_X	Def_X	Pos_X	$model_X(f)$
$false$				$\emptyset$
$x \wedge y$	•	•	•	$\{\{x, y\}, \{x, y, z\}\}$
$x \vee y$			•	$\{\{x\}, \{y\}, \{x, y\}, \{x, z\}, \{y, z\}, \{x, y, z\}\}$
$x \leftarrow y$		•	•	$\{\emptyset, \{x\}, \{z\}, \{x, y\}, \{x, z\}, \{x, y, z\}\}$
$x \vee (y \leftarrow z)$			•	$\{\emptyset, \{x\}, \{y\}, \{x, y\}, \{x, z\}, \{y, z\}, \{x, y, z\}\}$
$true$	•	•	•	$\{\emptyset, \{x\}, \{y\}, \{z\}, \{x, y\}, \{x, z\}, \{y, z\}, \{x, y, z\}\}$

Note that $x \vee y$ is not in Def_X (since its set of models is not closed under intersection) and that $false$ is neither in EPos_X, nor Pos_X, nor Def_X.

The 4-tuple $\langle \mathsf{Pos}_X, \models, \wedge, \vee\rangle$ is a finite lattice, where $true$ is the top element and $\wedge X$ is the bottom element. The set of (free) variables in a syntactic object o is denoted by var(o). Existential quantification is defined by Schröder's Elimination Principle, that is, $\exists x.f = f[x \mapsto true] \vee f[x \mapsto false]$. Also, $\exists\{y_1, \ldots, y_n\}.f$ (project out) abbreviates $\exists y_1 \ldots . \exists y_n.f$ and $\overline{\exists} Y.f$ (project

onto) denotes $\exists var(f) \setminus Y.f$. Two functions f, f' are equivalent, $f \equiv f'$ if and only if $f \models f'$ and $f' \models f$. Finally, for any $f \in \mathsf{Bool}_X$, $coneg(f) = model_X^{-1}(\{X \setminus M \mid M \in model_X(f)\})$.

3 Pos as Multiheaded Clauses

A Boolean function is positive if and only if every clause in its conjunctive normal form representation contains at least one positive literal. A clause is described as multiheaded if it contains one or more positive literals. In this paper, multiheaded clauses are written as implications with the body a conjunction of variables and the head a disjunction of variables. That is, a multiheaded clause has the form:

$$y_1 \wedge ... \wedge y_n \rightarrow x_1 \vee .. \vee x_m$$

Observe that the y_i and the x_j are distinct variables, otherwise the clause is equivalent to $true$. Let $f \in$ MHC denote that f is represented as a conjunction of multiheaded clauses.

Proposition 1. For every $f \in \mathsf{Pos}$ there is $f' \in$ MHC such that $f \equiv f'$.

Proof. It is well known that any Boolean formula is equivalent to another in conjunctive normal form. Suppose $f \equiv f'$, where f' is in conjunctive normal form. Since f is positive, every clause of f' must contain at least one positive literal, hence $f' \in$ MHC. ■

In the case that $m = 1$ the multiheaded clause is simply a propositional Horn clause. This suggests that the algorithms to calculate the domain operations might perform well if they naturally specialise to efficient propositional Horn clause algorithms. This will be the case for entailment checking. Moreover, the multiheaded clauses representation is particularly amenable to widening. If widening is required, the representation may be restricted in linear time so that clauses with more than, say n, heads are discarded. If $n = 1$, this widening corresponds to restricting to Def.

4 Domain Operations for Multiheaded Clauses

This section gives algorithms for the domain operations of Pos represented as multiheaded clauses. Meet ($\wedge$) is simply conjunction of clauses and is constant time; the other domain operations described are join ($\vee$), relative pseudo-complement ($\rightarrow$), entailment checking ($\models$) and projection out ($\exists$). The algorithms form the basis of the groundness analyser whose implementation is described in the next section.

4.1 Join

Consider $f = f_1 \vee f_2$, where $f_1, f_2 \in$ MHC. Suppose $f_1 = c_1 \wedge ... \wedge c_n$ and $f_2 = d_1 \wedge ... \wedge d_m$. Then, distributing, $f \equiv f' = \wedge_{i=1}^{n}(\wedge_{j=1}^{m}(c_i \vee d_j))$. Suppose

$c_i = y_1 \wedge ... \wedge y_k \rightarrow x_1 \vee ... \vee x_l$ and $d_i = u_1 \wedge ... \wedge u_p \rightarrow v_1 \vee ... \vee v_q$. Then $c_i \vee d_i \equiv y_1 \wedge ... \wedge y_k \wedge u_1 \wedge ... \wedge u_p \rightarrow x_1 \vee ... \vee x_l \vee v_1 \vee ... \vee v_q \in$ MHC. Hence $f' \in$ MHC. Since the above involves a quadratic blowup in the size of the representation, join is quadratic in the size of the input formulae.

4.2 Relative Pseudo-complement

Relative pseudo-complement has recently been used to support backward reasoning. In particular to trace control flow backward (right to left) to infer moding properties of initial queries [20].

Consider $f = f_1 \rightarrow f_2$, where $f_1, f_2 \in$ MHC. Suppose $f_1 = c_1 \wedge ... \wedge c_n$ and $f_2 = d_1 \wedge ... \wedge d_m$. Then, $f \equiv f' = \wedge_{j=1}^{m}(\vee_{i=1}^{n}(c_i \rightarrow d_j))$. Suppose $c_i = y_1 \wedge ... \wedge y_k \rightarrow x_1 \vee ... \vee x_l$ and $d_i = u_1 \wedge ... \wedge u_p \rightarrow v_1 \vee ... \vee v_q$. Then

$$c_i \rightarrow d_i = \begin{cases} \wedge_{i=1}^{l}(x_i \wedge u_1 \wedge ... \wedge u_p \rightarrow v_1 \vee ... \vee v_q) \\ \wedge \wedge_{j=1}^{k}(u_1 \wedge ... \wedge u_p \rightarrow y_j \vee v_1 \vee ... \vee v_q) \end{cases}$$

Hence $f' \in$ MHC. Given that the size of f' is exponential in the size of f_1, the operation is exponential. However, it should be noted that many analyses using positive Boolean functions (including groundness) do not require this operation to be calculated. In such cases the cost of this operation is not a drawback.

4.3 Entailment Checking

Entailment checking for positive Boolean functions represented in conjunctive normal form is *co-NP* complete [1]. However, as exploited in SAT solving, many of the Boolean functions that arise in practice can be checked for satisfiability with low complexity algorithms. This observation is exploited by the two algorithms detailed below. The first, *entailslite*, is incomplete and takes quadratic time in the size of the input. The second, *entailsheavy*, adds case splitting to the first algorithm to obtain completeness (which is required to guarantee termination in the fixpoint engine). This stratified algorithm usually only requires *entailslite* to be invoked once.

The *entailslite* algorithm (seen Figure 2) is an incomplete test that a multiheaded clause is entailed by a conjunction of multiheaded clauses: $\wedge_{i=1}^{l} B_i \rightarrow H_i \models B \rightarrow H$, where $B = y_1 \wedge ... \wedge y_n$ and $H = x_1 \vee ... \vee x_m$. It works by propagating deterministic bindings in an attempt to detect contradiction. The algorithm terminates either when a contradiction is found or when no more bindings can be propagated: then $Flag$ is returned. Notice that this algorithm contains forward chaining for propositional Horn clauses as a special case. Also notice that the variables are assigned values only once. The auxiliary rename produces a syntactic variant of a term which does not share any variables with the original term.

The algorithm *entailsheavy* (see Figure 3) applies case splitting if *entailslite* does not detect entailment. The number of cases is potentially exponential in

```
process entailslite(∧_{i=1}^{l} B_i → H_i, B → H)
        Flag := false;
        for i = 1 to m do x_i := false;
        for j = 1 to n do y_j := true;
        for k = 1 to l do
            spawn forward(B_k, H_k, Flag);
            spawn backward(B_k, H_k, Flag);
        return Flag.

process forward(B, H, Flag)
        block until every x ∈ B bound
            if ∧B ≡ true then spawn maketrue(H, Flag)
            else stop.

process backward(B, H, Flag)
        block until every y ∈ H bound
            if ∨H ≡ false then spawn makefalse(B, Flag)
            else stop.
process maketrue(H = {y_1, ..., y_m}, Flag)
        block until y_i ∈ H changes for some i ∈ {1, ..., m}
            if ∨H ≡ false then Flag := true; stop
            else if ∨H ≡ true then stop
            else if ∨H ≡ y_i for some i ∈ {1, ..., m} then y_i := true; stop
            else suspend.

process makefalse(B = {x_1, ..., x_n}, Flag)
        block until x_i ∈ B changes for some i ∈ {1, ..., n}
            if ∧B ≡ true then Flag := true; stop
            else if ∧B ≡ false then stop
            else if ∧B ≡ x_i for some i ∈ {1, ..., n} then x_i := false; stop
            else suspend.
```

Fig. 2. The *entailslite* Algorithm

the number of variables left unbound by *entailslite*. However, propagation occurs after each binding, therefore deep case splitting is rarely required. A more intelligent splitting strategy (as in SAT solving) could be applied, but the naïve strategy performs more than adequately.

Proposition 2. The algorithm *entailslite* is sound, but not complete for entailment checking. The algorithm *entailsheavy* is both sound and complete.

4.4 Projection

As in [19], projection is calculated using a Fourier-Motzkin style algorithm. The projection of a single variable out of a pair of clauses, one of which contains the variable in the body and the other in its head is performed by syllogising as

```
process entailsheavy(F, f)
        Flag :=entailslite(F, f);
        if Flag = true then return true
        else V := var(F)={x_1, ..., x_n};
        if V = ∅ then return false
        else do
            rename(F ∧ f)=(F' ∧ f');
            Flag' := entailsheavy({x'_1 ↦ true}F, {x'_1 ↦ true}f);
            if Flag' = true then
                rename(F ∧ f)=(F'' ∧ f'');
                return entailsheavy({x''_1 ↦ false}F'', {x''_1 ↦ false}f'')
            else return false.
```

Fig. 3. The *entailsheavy* Algorithm

follows:

$$\exists z.\begin{pmatrix} y_1 \wedge ... \wedge y_p \rightarrow z \vee x_1 \vee ... \vee x_q \\ \wedge\ z \wedge y_{p+1} \wedge ... \wedge y_n \rightarrow x_{q+1} \vee ... \vee x_m \end{pmatrix} = y_1 \wedge ... \wedge y_n \rightarrow x_1 \vee ... \vee x_m$$

The correctness and completeness of this is easily confirmed using Schröder elimination, hence the algorithm below is also correct and complete. In general, each variable is eliminated in turn, as follows. Suppose z is to be projected out of f.

1. All those clauses with z in the head are found, giving $\{C_i \mid i \in I\}$ where I is a (possibly empty) index set.
2. All those clauses with z in the body are found, giving $\{D_j \mid j \in J\}$ where J is a (possibly empty) index set.
3. These clauses of f are replaced by $\{\exists z.C_i \wedge D_j | i \in I, j \in J\}$
4. A compact representation is maintained by eliminating redundant clauses (absorption).

Step 4 means that the algorithm is parameterised by the compaction process. Compaction does not necessarily have to remove all redundant clauses (or indeed any), hence a tradeoff can be made between keeping the representation small and the cost of this maintenance. In projecting out a single variable, syllogising gives a quadratic blowup in the size of the representation. Thus the basic cost of projecting out a single variable is quadratic. However, the compaction step takes as its input a representation quadratic in the size of the original and the overall cost is dependent on the compaction algorithm. In the implementation, *entailslite* is used for compaction therefore the cost of projecting out a single variable is quartic. Because of the size blowup, projecting an arbitrary function onto a finite set of variables is exponential.

5 A Pos-Based Groundness Analyser

To assess the representation, two Pos-based groundness analysers built on multiheaded clauses were implemented in Prolog: one goal-dependent and one goal-independent. The analysers illustrate the ease with which the multiheaded clause

representation can be used. The analysers perform surprisingly well compared with other Pos analysers (including those with BDD-based Boolean function manipulation coded in C) and compared with analysers using more computationally tractable domains. This section details the Prolog implementation.

5.1 A GEP Representation

As in [2,19], the analyser maintains a factorised representation, that is, as a product of subdomains. The factorisation is encoded in the call and answer patterns. A call (or answer) pattern is a pair $\langle a, f \rangle$ where a is an atom and $f \in \mathsf{Pos}$. Normally the arguments of a are distinct variables. The formula f is a conjunction (list) of multiheaded clauses. In a non-ground representation the arguments of a can be instantiated and aliased to express simple dependency information [17]. For example, if $a = p(x_1, ..., x_5)$, then the atom $p(x_1, true, x_1, x_4, true)$ represents a coupled with the formula $(x_1 \leftrightarrow x_3) \wedge x_2 \wedge x_5$. This enables the abstraction $\langle p(x_1, ..., x_5), f_1 \rangle$ to be collapsed to $\langle p(x_1, true, x_1, x_4, true), f_2 \rangle$ where $f_1 = (x_1 \leftrightarrow x_3) \wedge x_2 \wedge x_5 \wedge f_2$. This encoding leads to a more compact representation and is similar to the GER factorisation of ROBDDs proposed by Bagnara and Schachte [2]. The representation of call and answer patterns described above is called GEP (groundness, equivalences and propositional clauses) where the atom captures the first two properties and the formula the latter.

The GEP representation is advantageous since it gives a compact representation whilst incurring little overhead when the representation is non-ground. The compactness of the representation affects memory usage and the complexity of domain operations. As demonstrated in [17], many dependencies arising in groundness analysis fall into the GE component. By using the GEP representation, many calls to expensive domain operations are avoided. Note that (as in [19]) the analyser does not maintain the factorisation strictly. Dependencies that could be encoded in the GE component may exist in the P component – the advantage of this is that the implementor may choose to update the GE component only when most computationally convenient.

5.2 Domain Operations for the GEP Representation

Meet. The meet of the pairs $\langle a_1, f_1 \rangle$ and $\langle a_2, f_2 \rangle$ can be computed by unifying a_1 and a_2 and concatenating f_1 and f_2.

Renaming. The objects that require renaming are formulae and call (answer) pattern GEP pairs. If a dynamic database is used to store the pairs, then renaming is automatically applied each time a pair is looked-up in the database. Formulae can be renamed with a single call to the Prolog builtin copy_term.

Entailment. Entailment checking works on three levels each called under a negation so as not produce any problematic bindings. The first entailment check operates only on the GE component (and is complete for this component). Entailment of the functions encoded in the GE component is denoted $a_1 \models a_2$. To test this, bind each distinct variable in a_1 to a distinct ground constant, resulting

in a'_1. If, after this has been performed, a'_1 may be unified with a_2, then $a_1 \models a_2$. Otherwise $a_1 \not\models a_2$. The second entailment check is only applied to formula in the P component. This implements the (incomplete) *entailslite* algorithm described in section 4.3. The propagating processes are realised using block declarations. A single pass over the formulae sets up the process and each clause results in two processes at any one time. The cost of suspending and resuming these processes is constant time, so propagation is achieved with very little overhead. The third entailment check implements a variant of the *entailsheavy* algorithm described in section 4.3. Copy_term produces a renamed formulae with new variables such that if any of the original variables have processes blocked on them, then the new variables will have copies of the processes blocked on them. This saves repeating work in the calls to *entailslite*.

Projection. Projection is only applied to formulae in the P component. It is performed using the algorithm given in section 4.4. Clauses produced by projection that are equivalent to *true* (that is, the intersection of the head and body variables is nonempty) are immediately discarded. The compaction step is based on the *entailslite* algorithm. However, as the purpose of compaction is to prevent an explosion in the size of the representation, compaction is only performed if the representation after syllogising is larger than beforehand. Since *entailslite* is incomplete some redundant clauses may be retained, however this is more than compensated by the reduced complexity of compaction.

Join. Calculating the join of the pairs $\langle a_1, f_1\rangle$ and $\langle a_2, f_2\rangle$ is complicated by the way that join interacts with renaming. Specifically, in a non-ground representation, call (answer) patterns would be typically stored in a dynamic database so that $var(a_1) \cap var(a_2) = \emptyset$. Hence $\langle a_1, f_1\rangle$ (or equivalently $\langle a_2, f_2\rangle$) have to be appropriately renamed before the join is calculated. This is achieved as follows. Plotkin's anti-unification algorithm [22] is used to compute the most specific atom a that generalises a_1 and a_2. (But observe that if $a_1 \models a_2$, a_2 is a most specific generalisation of the atoms.) The basic idea is to reformulate a_1 as a pair $\langle a'_1, f'_1\rangle$ which satisfies two properties: a'_1 is a syntactic variant of a; the pair represents the same dependency information as $\langle a_1, true\rangle$. A pair $\langle a'_2, f'_2\rangle$ reformulating a_2 is likewise constructed. The atoms a, a'_1 and a'_2 are unified and the formula $f' = (f_1 \wedge f'_1) \vee (f_2 \wedge f'_2)$ is calculated. This calculation is filtered by entailment checking. If $f_1 \wedge f'_1 \models f_2 \wedge f'_2$ can be detected using *entailslite*, then $f' = f_2 \wedge f'_2$ (and symmetrically). In this case the entailment check saves a call to join (and the associated projection) and the creation of a new data-structure, f'. Otherwise the join f' is computed as in section 4.1. Redundant clauses are removed from f' using *entailslite* to give f, and thereby the join $\langle a, f\rangle$.

5.3 Fixpoint Algorithms

The goal-dependent analyser is driven by an induced magic based iteration strategy, refining that used in [19]. Induced magic was introduced in [5], where a meta-interpreter for semi-naïve, goal-dependent, bottom-up evaluation is pre-

sented. Simple optimisations can significantly impact on performance. In particular, as noted in [18], evaluations resulting from new calls should be performed before those resulting from new answers, and a call to solve for one rule should finish before another call to solve for another rule starts. These optimisations have been incorporated into the induced magic framework by using an explicit redo list storing those call and answer patterns which have changed, thereby defining the clauses which need to be reevaluated. The goal-independent analyser is based on semi-naïve iteration. Neither of these analysers has exploited condensing [16,21].

6 Experimental Evaluation

To assess the feasibility of multiheaded clauses as a representation of positive Boolean functions, the Pos-based groundness analysers were tested on a large benchmark suite.

BDD representations of Boolean functions have been popular for the implementation of Pos-based groundness analysers. For this reason an analyser using a BDD package has also been instrumented. The BDD package available does not employ a GER factorisation. However, it should be noted that turning off the GEP factorisation with the multiheaded clause analyser does not greatly affect its performance. This is a strength of clausal representations. An RCNF analyser was also implemented in Prolog to aid the assessment of MHC. The three goal-dependent analysers share the same fixpoint algorithm and therefore run in lock-step.

The analysers are coded in SICStus Prolog 3.8.6 with the exception for the domain operations for BDD-based Pos, which were written in C by Schachte [23], and compiled with O2 level of optimisation. The analysers were run on a 296MHz Sun UltraSPARC-II with 1GByte of RAM running Solaris 7. Programs are abstracted following the elegant (two program) scheme of [4] to guarantee correctness. Programs containing disjunctions are normalised to definite clauses. Timeouts were set at two minutes.

Table 1 presents the experimental results for the larger programs in the benchmark suite. The columns detail the following information, file: the program name; size: the number of abstract clauses; abs: the time require to read, parse, normalise and abstract the program. For goal-dependent analysis the fixpoint times for the MHC, RCNF and BDD analysers are given, along with count: the number of ground argument positions in the call and answer patterns found by the analyser. For goal-independent analysis, the fixpoint times for MHC are given, along with the number of ground arguments in the success patterns. Timeout is denoted by '–'. The goal-independent counts are occasional larger than the goal-dependent counts owing to the presence of code unreachable from the initial query.

Multiheaded clauses perform consistently better than RCNF for goal-dependent analysis. This is unsurprising given the cost of meet and the relative expense of equivalence checking via dual Blake canonical form, together with the

Table 1. Timing and Precision Results

			goal-dep.				goal-indep.	
file	size	abs	MHC	RCNF	BDD	count	MHC	count
bridge.clpr	68	0.09	0.00	0.12	0.03	24	0.08	34
conman.pl	76	0.05	0.00	0.00	0.03	6	0.01	6
unify.pl	77	0.05	0.07	0.29	0.08	70	0.09	19
kalah.pl	78	0.05	0.02	0.11	0.04	199	0.02	42
nbody.pl	85	0.07	0.05	0.13	0.06	113	0.04	57
peep.pl	85	0.12	0.03	0.08	0.04	10	0.02	8
sdda.pl	89	0.06	0.04	0.07	0.05	17	0.02	4
bryant.pl	94	0.07	0.32	2.38	0.15	99	0.28	9
boyer.pl	95	0.08	0.05	0.07	0.04	3	0.02	5
read.pl	101	0.09	0.05	0.23	0.08	99	0.03	37
qplan.pl	108	0.09	0.03	0.25	0.07	216	0.05	27
trs.pl	108	0.13	0.10	2.28	0.26	13	0.04	7
press.pl	109	0.09	0.11	0.27	0.12	53	0.04	32
reducer.pl	113	0.07	0.08	0.17	0.09	41	0.05	21
parser_dcg.pl	122	0.09	0.09	0.29	0.08	43	0.04	24
simple_analyzer.pl	140	0.10	0.16	0.48	0.13	89	0.10	31
dbqas.pl	143	0.09	0.03	0.04	0.04	18	0.03	24
ann.pl	146	0.11	0.16	0.43	0.10	71	0.09	12
asm.pl	160	0.17	0.05	0.19	0.09	90	0.14	16
nand.pl	179	0.14	0.05	1.46	0.14	402	0.68	16
lnprolog.pl	220	0.10	0.08	0.19	0.12	143	0.07	31
ili.pl	221	0.15	0.55	1.63	0.13	4	0.15	5
strips.pl	240	0.22	0.03	0.07	0.08	142	0.06	36
sim.pl	244	0.22	1.09	24.78	0.25	100	0.62	33
rubik.pl	255	0.21	0.22	25.32	0.20	158	0.16	51
chat_parser.pl	281	0.36	0.29	1.75	0.26	505	0.30	128
sim_v5-2.pl	288	0.23	0.07	0.33	0.16	457	0.10	37
peval.pl	332	0.18	0.64	4.62	0.16	27	1.30	17
aircraft.pl	395	0.54	0.15	0.70	0.41	687	0.12	196
essln.pl	595	0.48	0.19	20.72	0.37	162	0.30	75
chat_80.pl	883	1.43	0.88	4.28	0.84	855	0.64	339
aqua_c.pl	3928	3.55	7.68	67.04	–	1285	6.59	458

filtering applied to join in MHC. MHC compares favourably with BDDs, especially considering that the BDD operations exploit memoisation and are coded in C. In terms of runtime, MHC and BDDs give similar results, although as would be expected, the different representations performed differently on different programs. For example, BDD perform well on sim.pl, whereas MHC perform well on sim_v5-2.pl. The MHC analyser appears to scale smoothly for both goal-dependent and goal-independent analysis. Of course, any Pos-based analyser can be broken using the schema from [6,14]; the analyser can deal with the arity 14 case of [6] before timeout (that is, a single predicate requiring 16384 iterations).

The major cost in entailment checking is incurred through case splitting in *entailsheavy*. Instrumentation has revealed that the total number of times *entail-*

slite is invoked in checking $F \models f$ almost never exceeds $|\mathrm{var}(F)|$. Therefore in practice *entailsheavy* exhibits cubic behaviour in the size of the input formulae. Further instrumentation has shown that the maximum number of heads observed in a clause is four. These maxima occur infrequently. Since most clauses have few heads, typically only a small number of bindings have to be made before propagation binds sufficient variables to return the $Flag$. The calls to *entailsheavy* typically do not detect entailment, as the vast majority of entailments are detected using *entailslite*. As disentailment is demonstrated by the discovery of a single countermodel, the binding of a small number of variables to their value in a countermodel is often enough to generate the rest of this countermodel via propagation. This helps to explain the success of the stratified entailment check.

7 Related Work

The efficiency of groundness analysis depends on the way dependencies are represented and implemented. The representation decides the algorithmic complexity of the domain operations but the implementation can introduce a prohibitive constant factor or even push the complexity into a higher class if there is not a good match between the representation and the implementation language. Efficient BDD-based Pos analysis are usually implemented in languages with mutable data-structures such as C [24] or SML [10,11]. State-of-the-art BDD-based groundness analysers employ a GE factorisation [2] which keeps simple definite information separate from dependency information. This leads to a particularly dense representation (meant informally, a small number of nodes/clauses in the representation) and is therefore an important implementation tactic.

The density of the representation is as important to Prolog as it is to C: the density determines the size of the inputs to the domain operations, as well as impacting on memory usage. The dual Blake canonical form representation of Def functions [1,9] is attractive as it is amenable to Prolog implementation [12] and it gives a unique representation for every Def function (up to variable ordered). However, its requirement to make transitive variable dependencies explicit can compromise density. For example, the function $(x \leftarrow y) \wedge (y \leftarrow z)$ is represented as $(x \leftarrow (y \vee z)) \wedge (y \leftarrow z)$. Because of this Howe and King [19] present a (non-orthogonal [1]) clausal representation of Def as conjunctions of propositional clauses, but do not maintain a canonical form. Therefore entailment checking is required to detect stability.

Recently, Genaim and Codish [13] have proposed a dual representation for Def. For function f, the models of $coneg(f)$ are named and f is represented by a tuple recording for each variable of f which of these models the variable is in. For example, the models of $coneg(x \rightarrow y)$ are $\{\{x, y\}, \{x\}, \emptyset\}$. Naming the three models a, b, c respectively, f is represented by $\langle ab, a\rangle$. This representation cleverly allows ACI1 unification theory to be used for the domain operations and elegantly supports a GE factorisation. Promising experimental results are reported [13], but a widening is required to analyse the aqua_c benchmark.

Codish and Demoen [7] describe a model based Prolog implementation technique for Pos that would encode $x_1 \leftrightarrow (x_2 \wedge x_3)$ as three tuples $\langle true, true, true\rangle$, $\langle false, \_, false\rangle$, $\langle false, false, \_\rangle$. The technique performs well against BDD-based Pos analysis of its era [24] but it does not scale smoothly to the larger benchmarks. Heaton *et al.* [17] therefore propose EPos, a sub-domain of Def, that can only propagate dependencies of the form $(x_1 \leftrightarrow x_2) \wedge x_3$ across procedure boundaries. This information is precisely that contained in one of the fields of the GE factorisation. The main finding of [17] is that this sub-domain retains reasonably precision for goal-dependent analysis and possesses good scaling behaviour.

8 Conclusion

Positive Boolean functions can be naturally expressed as multiheaded clauses which are straightforward to understand, manipulate and code in Prolog. Multiheaded clauses have been used as the basis for efficient goal-dependent and goal-independent Pos-based groundness analysers. The key to the success of these analysers is their constant time meet and their use of entailment checking succinctly and efficiently coded using block declarations. Entailment checking is stratified so that many entailments are detected using a low complexity algorithm. The full exponential algorithm is only applied when necessary for detecting stability, and even then the number of case splits is typically very small. The analysers do not require widening for any of the benchmarks; however, natural widenings to Def or to EPos are available if required [6,14]. This work illustrates the subtlety of choosing a representation and its associated operations, even for a well known domain. Minor changes to the representation can have a significant impact on performance if they affect frequently occurring operations. It also demonstrates the effectiveness of stratifying high complexity operations to avoid expensive computation whenever possible. The intelligent application of the simple entailment checking algorithm is the heart of the analyser presented in this paper.

References

1. T. Armstrong, K. Marriott, P. Schachte, and H. Søndergaard. Two Classes of Boolean Functions for Dependency Analysis. *Science of Computer Programming*, 31(1):3–45, 1998.
2. R. Bagnara and P. Schachte. Factorizing Equivalent Variable Pairs in ROBDD-Based Implementations of Pos. In *Seventh International Conference on Algebraic Methodology and Software Technology*, volume 1548 of *Lecture Notes in Computer Science*, pages 471–485. Springer-Verlag, 1999.
3. P. Bigot, S. Debray, and K. Marriott. Understanding Finiteness Analysis using Abstract Interpretation. In *Joint International Conference and Symposium on Logic Programming*, pages 735–749. MIT Press, 1992.
4. F. Bueno, D. Cabeza, M. Hermenegildo, and G. Puebla. Global Analysis of Standard Prolog Programs. In *European Symposium on Programming*, volume 1058 of *Lecture Notes in Computer Science*, pages 108–124. Springer-Verlag, 1996.

5. M. Codish. Efficient Goal Directed Bottom-up Evaluation of Logic Programs. *Journal of Logic Programming*, 38(3):355–370, 1999.
6. M. Codish. Worst-Case Groundness Analysis using Positive Boolean Functions. *Journal of Logic Programming*, 41(1):125–128, 1999.
7. M. Codish and B. Demoen. Analysing Logic Programs using "prop"-ositional Logic Programs and a Magic Wand. *Journal of Logic Programming*, 25(3):249–274, 1995.
8. M. Codish, H. Søndergaard, and P. Stuckey. Sharing and Groundness Dependencies in Logic Programs. *ACM Transactions on Programming Languages and Systems*, 21(5):948–976, 1999.
9. P. Dart. On Derived Dependencies and Connected Databases. *Journal of Logic Programming*, 11(1–2):163–188, 1991.
10. C. Fecht. *Abstrakte Interpretation logischer Programme: Theorie, Implementierung, Generierung*. PhD thesis, Universität des Saarlandes, 1997.
11. C. Fecht and H. Seidl. A Faster Solver for General Systems of Equations. *Science of Computer Programming*, 35(2-3):137–162, 1999.
12. M. García de la Banda, M. Hermenegildo, M. Bruynooghe, V. Dumortier, G. Janssens, and W. Simoens. Global Analysis of Constraint Logic Programs. *ACM Transactions on Programming Languages and Systems*, 18(5):564–614, 1996.
13. S. Genaim and M. Codish. The Def-inite Approach to Dependency Analysis. In *European Symposium on Programming*, volume 2028 of *Lecture Notes in Computer Science*, pages 417–32. Springer-Verlag, 2001.
14. S. Genaim, J. M. Howe, and M. Codish. Worst-Case Groundness Analysis using Definite Boolean Functions. *Theory and Practice of Logic Programming*, 2001. Forthcoming.
15. R. Giacobazzi, S. Debray, and G. Levi. Generalized Semantics and Abstract Interpretation for Constraint Logic Programs. *Journal of Logic Programming*, 25(3):191–247, 1995.
16. R. Giacobazzi and F. Scozzari. A Logical Model for Relational Abstract Domains. *ACM Transactions on Programming Languages and Systems*, 20(5):1067–1109, 1998.
17. A. Heaton, M. Abo-Zaed, M. Codish, and A. King. A Simple Polynomial Groundness Analysis for Logic Programs. *Journal of Logic Programming*, 45(1–3):143–156, 2000.
18. M. Hermenegildo, G. Puebla, K. Marriot, and P. Stuckey. Incremental Analysis of Constraint Logic Programs. *ACM Transaction on Programming Languages and Systems*, 22(2):187–223, 2000.
19. J. M. Howe and A. King. Implementing Groundness Analysis with Definite Boolean Functions. In *European Symposium on Programming*, volume 1782 of *Lecture Notes in Computer Science*, pages 200–214. Springer-Verlag, 2000. Available at http://www.cs.ukc.ac.uk/pubs/2000/949/.
20. A. King and L. Lu. A Backwards Analysis for Constraint Logic Programs. Technical Report 4-01, University of Kent, 2001.
21. K. Marriott and H. Søndergaard. Precise and Efficient Groundness Analysis for Logic Programs. *ACM Letters on Programming Languages and Systems*, 2(4):181–196, 1993.
22. G. Plotkin. A Note on Inductive Generalisation. *Machine Intelligence*, 5:153–163, 1970.
23. P. Schachte. *Precise and Efficient Static Analysis of Logic Programs*. PhD thesis, Department of Computer Science, The University of Melbourne, Australia, 1999.
24. P. Van Hentenryck, A. Cortesi, and B. Le Charlier. Evaluation of the Domain *Prop*. *Journal of Logic Programming*, 23(3):237–278, 1995.

Higher-Precision Groundness Analysis

Michael Codish[1], Samir Genaim[1], Harald Søndergaard[2], and Peter J. Stuckey[2]

[1] Dept. of Computer Science, Ben-Gurion Univ. of the Negev, Beer-Sheva, Israel
[2] Dept. of Comp. Science and Software Eng., Univ. of Melbourne, Australia

Abstract. Groundness analysis of logic programs using Pos-based abstract interpretation is one of the clear success stories of the last decade in the area of logic program analysis. In this work we identify two problems with the Pos domain, the multiplicity and sign problems, that arise independently in groundness and uniqueness analysis. We describe how these problems can be solved using an analysis based on a domain Size for inferring term size relations. However this solution has its own shortcomings because it involves a widening operator which leads to a loss of Pos information. Inspired by Pos, Size and the LSign domain for abstract linear arithmetic constraints we introduce a new domain LPos, and show how it can be used for groundness and uniqueness analysis. The idea is to use the sign information of LSign to improve the widening of Size so that it does not lose Pos information. We prove that the resulting analyses using LPos are uniformly more precise than those using Pos.

1 Introduction

Groundness analysis of logic programs using Pos-based [7,18,19] abstract interpretation [9] is one of the success stories of the last decade in the area of logic program analysis. Moreover, Pos can be the basis for many other applications such as finiteness analysis, rigidity analysis, type analysis, and suspension analysis (for concurrent logic programs). It has also been applied in the context of constraint logic programs to determine when variables have obtained a final unique value [2]. Here we shall refer to such an analysis as *uniqueness* analysis.

The abstract domain Pos [7,18] consists of the positive Boolean functions, ordered by logical consequence. For groundness analysis, the idea is to express groundness information about a set of substitutions as a Boolean function. For example, the logical reading of the formula $\psi = x \wedge (y \rightarrow z)$ for groundness analysis is: on application of any substitution, x is ground and if y becomes ground then also z does. More formally, we say that a Boolean function ψ describes a substitution θ if any set of variables that might become ground by further instantiating θ is a model of ψ. For uniqueness analysis, the situation is similar, but the Boolean function is interpreted as a statement about uniqueness dependencies. For example, if any two of the three variables in the linear constraint $x = 2y + 3z$ have a unique value then so does the third. This is described by $((x \wedge y) \rightarrow z) \wedge ((y \wedge z) \rightarrow x) \wedge ((z \wedge x) \rightarrow y)$.

Pos is attractive because its structure and corresponding abstract operations are simple and Pos-based analysers are efficient in practice [12,22]. But

P. Codognet (Ed.): ICLP 2001, LNCS 2237, pp. 135–149, 2001.

the domain has its limits when we consider precision of analysis. This paper exposes two shortcomings and illustrates their effect on groundness and uniqueness analyses for logic and constraint logic programs. The first problem relates to the fact that a Boolean function does not capture information about multiplicity of variables, so the term equations $[x|xs] = [x|ys]$ and $[x|xs] = [x, x|ys]$ are both approximated by $(x \wedge xs) \leftrightarrow (x \wedge ys)$. The second problem arises because Pos abstractions for arithmetic constraints ignore the signs of coefficients, so the systems of constraints $(x-y-z=0) \wedge (x-y=3)$ and $(x-y-z=0) \wedge (x+y=3)$ are both given the Pos description $(x \leftrightarrow y) \wedge (x \rightarrow z)$.

We look at several possible solutions to the two problems. For groundness analysis, the use of size relations [3,11,23] addresses the multiplicity problem, and can provide groundness information. A substitution is approximated by a constraint on term sizes. For example, the inequality $x \leq y + z$ describes a substitution for which the size of the term x is bound to is smaller than or equal to the sum of the sizes of the terms y and z are bound to (for some appropriately chosen measure of term size). Groundness dependencies are captured, since any substitution which makes y and z ground must make x ground to satisfy the size constraint. While the abstract domain Size of size constraints can be shown to be at least as precise as Pos, it has infinite ascending chains and so must be applied in combination with a widening operator [11]. Widening sometimes results in the loss of groundness information so that in many cases a Pos based groundness analysis is more precise than a size relations based analysis.

For uniqueness analysis, the abstract domain LSign [20], where linear constraints are abstracted by replacing coefficients by their signs, can provide the needed information about signs of coefficients. Intuitively an LSign description $\oplus x + \ominus y + \ominus z = 0 \wedge \oplus x + \oplus y = \oplus$ defines constraints C equivalent to $a_1x - a_2y - a_3z = 0 \wedge a_4x + a_5y = a_6,\ a_i > 0, 1 \leq i \leq 6$, for example $(x-y-z=0) \wedge (x+y=3)$. The coefficient information is enough to determine the uniqueness information $(x \leftrightarrow y) \wedge (x \leftrightarrow z)$. Unfortunately, LSign is aimed at capturing a different kind of information, namely whether a represented constraint must be satisfiable or not, and LSign operations quickly lose information important to uniqueness analysis.

Inspired by Size and LSign we propose a new domain, LPos, which improves on Pos, both for groundness analysis and for uniqueness analysis. The idea is to use the sign information of LSign to improve the widening of Size so that it does not lose Pos information. LPos uses the same style of description as LSign and, importantly, shares the use of an abstract Fourier elimination algorithm for the abstract projection operation. It differs crucially from LSign by using operations which maintain better uniqueness information.

2 Problems with Pos

Recall that a Boolean function is *positive* if it is satisfied by the assignment of *true* to all of its variables. The abstract domain Pos consists of the set of positive Boolean functions (together with *false* as a bottom element) ordered by

```
bf(Tree, List):-
   bfq(s(zero),[Tree|QT],QT List).

bfq(zero,Q,Q,[]).
bfq(s(N),[nil|QH], QT, Es):-
   bfq(N,QH, QT, Es).
bfq(s(N),[t(L,E,R)|QH],[L,R|QT],[E|Es]):-
   bfq(s(s(N)), QH, QT, Es).
```

$$
\begin{aligned}
&\mathsf{bf}(u, v) = \\
&\quad \exists qt : \mathsf{bfq}(true, u \wedge qt, qt, v) \\
&\mathsf{bfq}(w, x, y, z) = \\
&\quad [w \wedge (x \leftrightarrow y) \wedge z] \\
&\vee\ [\exists n, qh : (w \leftrightarrow n) \wedge (x \leftrightarrow qh) \wedge \\
&\quad \mathsf{bfq}(n, qh, y, z)] \\
&\vee\ [\exists n, l, e, r, qt, qh, es : \\
&\quad (w \leftrightarrow n) \wedge (x \leftrightarrow l \wedge e \wedge r \wedge qh) \wedge \\
&\quad (y \leftrightarrow l \wedge r \wedge qt) \wedge (z \leftrightarrow e \wedge es) \wedge \\
&\quad \mathsf{bfq}(n, qh, qt, es)]
\end{aligned}
$$

Fig. 1. Breadth-first traversal of binary trees (left) with Pos analysis (right).

logical consequence [18,7,19]. The abstract operations that arise in analyses are: Boolean conjunction, as a greatest lower bound, Boolean disjunction, as a least upper bound, and existential quantification, for projection. There is a substantial body of literature on this type of analysis, for example, [1,4,8]. In this section we describe two problems with the Pos domain.

The multiplicity problem: The first problem arises in groundness analysis of logic programs. For groundness analysis, a Boolean function ψ describes a substitution θ if any set of variables that might become ground by further instantiating θ is a model of ψ. The unification of two terms t and s is described in Pos by the Boolean formula $(x_1 \wedge x_2 \wedge \ldots \wedge x_n) \leftrightarrow (y_1 \wedge y_2 \wedge \ldots \wedge y_m)$ where $\{x_1, x_2, \ldots x_n\}$ and $\{y_1, y_2, \ldots y_m\}$ are the sets of variables in t and s.

The multiplicity problem for Pos is a loss of groundness information because the abstraction function (of a Herbrand constraint) ignores the multiplicity of variables in terms. For example, the two term equations $[x|xs] = [x|ys]$ and $[x|xs] = [x, x|ys]$ are both described by the Boolean function $(x \wedge xs) \leftrightarrow (x \wedge ys)$. The inability of a Pos analysis to distinguish the two explains why it cannot find the (otherwise correct) description $xs \leftrightarrow ys$ for the first equation.

Of course neither equation is likely to appear in real-world programs. But a similar phenomenon *is* found, less transparently, in real programs. Consider the program given in Figure 1 (left side), for breadth-first traversal of binary trees. The program uses a queue of tree nodes which is represented in the first three arguments of `bfq/4`—the first argument represents the number of elements in the queue, and the second and third represent a difference list of the elements on the queue. This setup ensures that the program can be executed in both directions. A groundness analysis using Pos consists of translating the program into the recursive definition of a Boolean function illustrated in Figure 1 (right side) and then finding a closed form using Kleene iteration. This yields a description $w \wedge (x \leftrightarrow (y \wedge z))$ for $\mathsf{bfq}(w, x, y, z)$ indicating that in any answer to a query to $\mathsf{bfq}/4$, the first argument is ground, and the second is ground if and only if the third and fourth both are. This result is precise. But, consider now the analysis of the predicate $\mathsf{bf}/2$. The result for $\mathsf{bf}(u, v)$ is obtained from the result for $\mathsf{bfq}/4$ as follows: $\exists qt : (u \wedge qt) \leftrightarrow (qt \wedge v)$ which is $true$. Hence the Pos-based analysis

```
mg(P,T,R,B) :- T = 1, B = (105/100)*P-R.
mg(P,T,R,B) :- T > 1, T1 = T-1, P1 = (105/100)*P-R, mg(P1,T1,R,B).
```

Fig. 2. Mortgage repayment in $\mathsf{CLP}(\mathbb{R}_{\mathsf{Lin}})$.

of $\mathsf{bf}/2$ yields no useful groundness information, even though in fact the first argument to $\mathsf{bf}/2$ is ground if and only if the second is.

The sign problem: The sign problem for Pos arises in the context of uniqueness analysis of linear arithmetic constraints [6,2]. In this context, a Boolean function of the form $(x \wedge y) \rightarrow z$ expresses that if the values of the variables x and y become uniquely determined, then the value of the variable z will have been determined as a result. A constraint such as $x = y - 1$ is described by the Pos function $x \leftrightarrow y$ and a constraint such as $x = y + 3z - 1$ by $((x \wedge y) \rightarrow z) \wedge ((y \wedge z) \rightarrow x) \wedge ((z \wedge x) \rightarrow y)$. A problem is that the Pos abstraction of an arithmetic constraint ignores the signs of the coefficients in the arithmetic constraints. In particular, it is not possible for Pos to discover linear independence amongst constraints determined by these signs.

Figure 2 shows a $\mathsf{CLP}(\mathbb{R}_{\mathsf{Lin}})$ program to calculate mortgage repayments[1]. Uniqueness analysis using Pos begins by translating the program into a (possibly recursive) definition of a positive Boolean function, just as for groundness analysis. The difference is in the translation of the primitive constraints which are abstracted according to the following abstraction function:

$$\alpha(c) = \begin{cases} true & \text{if } c \text{ is inequality} \\ \nabla(vars(c)) & \text{if } c \text{ is an equation} \end{cases}$$

where $\nabla(S) = \bigwedge_{v \in S} ((\bigwedge_{v' \in S \setminus \{v\}} v') \rightarrow v)$. The abstract version of Figure 2 is:

$$\mathsf{mg}(p,t,r,b) = [t \wedge (b \wedge r \rightarrow p) \wedge (r \wedge p \rightarrow b) \wedge (p \wedge b \rightarrow r)] \ \vee \ [\exists t_1, p_1 : (t_1 \leftrightarrow t) \wedge ((p_1 \wedge p) \rightarrow r) \wedge ((p \wedge r) \rightarrow p_1) \wedge ((r \wedge p_1) \rightarrow p) \wedge \mathsf{mg}(p_1, t_1, r, b)]$$

A closed form for $\mathsf{mg}(p,t,r,b)$ is $t \wedge ((p \wedge r) \rightarrow b)$. This expresses that any successful query will bind `T` to a unique value. Moreover, if `P` and `R` are given unique values in a query, then `B` will be determined as a result. Again, this result is less precise than we might have hoped for. If one queries `mg` with fixed values for all of the parameters, except for `P`, then `P` will be determined as a result. This information is not expressed in the result of the Pos analysis.

The problem here is that Pos cannot identify linear dependencies between constraints in a conjunction. For example, both of the systems of linear constraints: $(x - y - z = 0) \wedge (x - y = 3)$ and $(x - y - z = 0) \wedge (x + y = 3)$ are best described by $(x \leftrightarrow y) \wedge (x \rightarrow z)$. However, the former determines z while the latter does not. In fact, the uniqueness information in the two systems is best described by $(x \leftrightarrow y) \wedge z$ and $(x \leftrightarrow y) \wedge (x \leftrightarrow z)$, respectively.

[1] The interest is fixed at 5% to avoid the presence of non-linear constraints.

3 Size Relations

An interesting solution to the multiplicity problem for Pos is obtained by applying a size relations analysis [3,23], more commonly known in the context of termination analysis [5,16]. The interpretation of this type of analysis is parametric to a given size function on terms, called a symbolic norm [16]. Symbolic norms ($|\cdot|$) are similar to linear norms except that variables are mapped to variables. In this approach, linear equalities and inequalities express constraints on the sizes of the terms bound to variables. For example, $z = x + y$ describes those substitutions σ, for which every instance $\sigma\tau$ satisfies $|z\sigma\tau| = |x\sigma\tau| + |y\sigma\tau|$.

Size relation analysis has many applications, and different norms support different uses. Here we are interested in an application for groundness information, so adopt a norm which assigns any ground term to the size zero, and a non-ground term to the sum of the sizes of its variables. We refer to this size function as the var-norm.

Definition 1 (Var-norm). *Given Herbrand term t, we define its* var-norm *by:*

$$|t|_{vn} = \begin{cases} 0 & \text{if } t \text{ is a constant} \\ t & \text{if } t \text{ is a variable} \\ |t_1|_{vn} + \cdots + |t_n|_{vn} & \text{if } t \text{ has form } f(t_1, \ldots, t_n) \end{cases}$$

A size relation analysis for a logic program P can be formalised in terms of the semantics of a corresponding CLP($\mathbb{N}$) program on the sizes of the terms in P. CLP($\mathbb{N}$) stands for constraint logic programs over a domain consisting of the non-negative integers. The semantic objects are sets of CLP($\mathbb{N}$) facts modulo an equivalence relation defined in terms of constraint equivalence. The meaning of a CLP($\mathbb{N}$) program is a potentially infinite set of CLP($\mathbb{N}$) facts. In practice, analyses apply a convex hull operation as an upper bound operator to improve efficiency, and a widening operator [11] (which detects stable edges in corresponding polyhedral representations of the constraints on the term sizes) to guarantee the termination of the analysis.

There is a pleasing similarity between a size relations analysis and a groundness dependency analysis. Both fit into the framework of Giacobazzi et al. [15]. We can translate a (possibly recursive) definition of a predicate into a (possibly recursive) definition of a constraint, expressing the term size relations for the predicate. A term equation $t = s$ is translated into $|t| = |s|$. For the var-norm, the bfq program in Figure 1 is translated to:

$$\begin{aligned} \mathsf{bf}(u, v) = \quad & \exists qt : \mathsf{bfq}(0, u + qt, qt, v) \\ \mathsf{bfq}(w, x, y, z) = \quad & [w = 0 \wedge x = y \wedge z = 0] \\ & \vee\ [\exists n, qh : w = n \ \wedge\ x = qh \ \wedge\ \mathsf{bfq}(n, qh, y, z)] \\ & \vee\ [\exists n, l, r, e, qh, qt, es : \\ & \qquad w = n \ \wedge\ x = l + r + e + qh \ \wedge \\ & \qquad y = l + r + qt \ \wedge\ z = e + es \ \wedge \mathsf{bfq}(n, qh, qt, es)] \end{aligned}$$

It is easy to verify that $w = 0 \wedge x = y + z$ is a fixed point for bfq/4.

Size relations analyses also provide groundness dependencies. Let us make this intuition precise, without delving in the details. Consider a substitution σ which is described by the conjunction of primitive CLP($\mathbb{N}$) constraints each of the form: $\Sigma_j a_j x_j \leq b + \Sigma_k b_k x_k$ where the coefficients a_j and b_k are positive. The groundness information contained in such a constraint can be expressed in Pos by: $\wedge\{x_k | k \in K\} \rightarrow \quad \wedge \{x_j | j \in J\}$. For example, if σ satisfies the size relation $x \leq y + z$ and variables y and z become ground, then x must be ground as a result. Otherwise it would not be correct that *any instance* of σ would map x to a term smaller than $|y|_{\mathsf{vn}} + |z|_{\mathsf{vn}}$.

The groundness information we obtain through size analysis for bfq/4 in Figure 1 is identical to that obtained using Pos, namely $w \wedge (x \leftrightarrow y \wedge z)$. However, the size analysis of bf/2 is illuminating. It shows how Size may be more precise than Pos for groundness dependencies. The CLP($\mathbb{N}$) constraint is: $\mathsf{bf}(u, v) = \exists qt : \mathsf{bfq}(0, u + qt, qt, v) = \exists qt : 0 = 0 \wedge u + qt = qt + v = u = v$. That is, the groundness dependency $u \leftrightarrow v$ can be inferred. This is more precise than what was obtained with Pos. The Size solution $u = v$ was obtained because qt had the same coefficient on both sides of the equation $u + qt = qt + v$. With Pos we cannot distinguish this situation from $u + qt = qt + qt + v$, for example, and so the Pos result has to cover also the possible Size results $u \leq v$ and $u \geq v$.

Formalising the abstract interpretation between Size and Pos using the adjoint framework [9] for abstract interpretation is not obvious, since the translation described above does not give a best approximation. Consider for example the conjunction $(x + y + y = z) \ \wedge \ (x + y = z)$. The Pos information obtained from these constraints is $(x \wedge y) \leftrightarrow z$, while that obtained from the normalised system $x = z \wedge y = 0$ is the more precise $(x \leftrightarrow z) \wedge y$. So we need to define a normalisation procedure in order to use the adjoint framework. Alternatively we can use a more general framework, such as the relational framework [17].

However, Size is not the panacea for groundness analysis that it might appear to be. As an abstract domain it has infinite ascending chains, and hence we need to apply a widening operator in an analysis. Consider the program

```
listof(X,Xs) :- Xs = [X].
listof(X,Xs) :- Xs = [X|Ys], listof(X,Ys).
```

An analysis using Pos finds the groundness dependency $x \leftrightarrow xs$. An analysis using Size constructs the equation:

$$\mathsf{listof}(x, xs) \quad = \quad xs = x \vee [\exists ys : xs = x + ys \wedge \mathsf{listof}(x, ys)]$$

Interpreting $\vee$ as the convex hull operation, Kleene iteration yields:

$$\begin{array}{lcl}
\mathsf{listof}^1(x, xs) & = & (xs = x) \\
\mathsf{listof}^2(x, xs) & = & (xs = x) \vee [\exists ys : xs = x + ys \wedge ys = x)] \ = \ (x \leq xs \leq 2x) \\
\mathsf{listof}^3(x, xs) & = & (x \leq xs \leq 3x) \\
\mathsf{listof}^4(x, xs) & = & (x \leq xs \leq 4x) \\
\vdots & &
\end{array}$$

To avoid traversing the infinite chain, it is customary to apply a widening operator which retains only the stable part of the constraint, and the result $x \leq xs$ is obtained as a fixed point. We have lost the groundness information $x \rightarrow xs$.

4 A More Precise Domain

Inspired by the Size and LSign domains, we introduce a new abstract domain, LPos, that solves the problems discussed in Sections 2 and 3. The idea is to represent linear constraints by their signs (abstract coefficients), but unlike LSign these abstract coefficients are introduced only by widening, and not at abstraction time. Basically, this enables us to preserve the Pos information which might be lost when applying the widening operation of the Size relations domain. For example, recall the analysis of the listof/2 program in Size. The size relations analysis resulted in an infinite chain: $x \leq xs \leq x$, $x \leq xs \leq 2x$, $x \leq xs \leq 3x$, ... and so a widening was applied to get $x \leq xs$. With LPos the analysis gives $x \leq xs \leq \oplus x$, where $\oplus$ stands for some positive coefficient. This ensures the termination of the analysis and preserves the groundness dependencies otherwise lost by widening with Size.

The LPos domain: We define a primitive LPos constraint c to be of the form: $\Sigma_i\ a_i x_i\ op\ b$ where $a_i, b \in \mathbb{R} \cup \{\oplus, \ominus, \top\}$ and $op \in \{\leq, =\}$. Note that the coefficients in an LPos constraint can be concrete (rationals) or abstract (signs). The abstract coefficients represent sets of concrete coefficients under the obvious mapping: $\gamma_{\text{Sign}}(\top) = \mathbb{R}$, $\gamma_{\text{Sign}}(\oplus) = \{a \mid a \in \mathbb{R}, a > 0\}$, $\gamma_{\text{Sign}}(\ominus) = \{a \mid a \in \mathbb{R}, a < 0\}$ and $\gamma_{\text{Sign}}(a) = \{a\}$ otherwise. An LPos constraint is a conjunction of primitive LPos constraints.

The elements of LPos are (equivalence classes of) sets (for disjunction) of LPos constraints. The concretization function for LPos, γ_{LPos} maps the elements of LPos to sets of concrete constraints.

$$\gamma_{\text{LPos}}(\sum_{i=1}^{n} a_i x_i\ op\ b) = \{\sum_{i=1}^{n} c_i x_i\ op\ d \mid c_i \in \gamma_{\text{Sign}}(a_i), d \in \gamma_{\text{Sign}}(b)\}$$

$$\gamma_{\text{LPos}}(c_1 \wedge \cdots \wedge c_m) = \{d_1 \wedge \cdots \wedge d_m \mid d_i \in \gamma_{\text{LPos}}(c_i), 1 \leq i \leq m\}$$

$$\gamma_{\text{LPos}}(D) = \bigcup_{C \in D} \gamma_{\text{LPos}}(C)$$

We consider concrete constraints modulo logical (constraint) equivalence. The concretization induces a partial order on LPos descriptions as follows:

$$D_1 \preceq_{\text{LPos}} D_2 \quad \Leftrightarrow \quad \gamma_{\text{LPos}}(D_1) \subseteq \gamma_{\text{LPos}}(D_2)$$

The *join* and *meet* operations are defined as follows:

$$D_1 \sqcup_{\text{LPos}} D_2 = D_1 \cup D_2 \quad \textit{and} \quad D_1 \sqcap_{\text{LPos}} D_2 = \{C_1 \wedge C_2 \mid C_1 \in D_1, C_2 \in D_2\}$$

Example 1. The LPos description $\{x \leq 0, 2x + \oplus y \leq 4 \wedge -x \leq 0\}$ represents the set of constraints $\{x \leq 0\} \cup \{2x + ay \leq 4 \wedge -x \leq 0 \mid a > 0\}$. Hence $2x + y \leq 4 \wedge -x \leq 0$ is represented by the LPos description, and, since it is logically equivalent, so is $2x + y \leq 4 \wedge -x \leq 0 \wedge y \leq 5 \wedge -x \leq 3$.

Abstract operations: The required operations on LPos to support groundness and uniqueness analyses are: *conjunction*, *disjunction* and *projection*. The first two are $\sqcap_{\text{LPos}}$, and $\sqcup_{\text{LPos}}$ respectively. Projection is the only complex operation in this domain. It is defined as an extension of Fourier elimination, enhanced

to handle the abstract coefficients. Before defining the projection algorithm, we introduce arithmetic over $\mathbb{R} \cup \{\oplus, \ominus, \top\}$, and other operations needed for extracting the sign of the eliminated (by projection) variable's coefficient.

To eliminate variables we need to know their signs. Hence we define a mapping that maps coefficients to their signs:

Definition 2 (Sign abstraction, pos, neg).

$$\alpha_{\text{Sign}}(a) = \begin{cases} 0 & \textit{if } a = 0 \\ \oplus & \textit{if } a > 0 \textit{ or } a = \oplus \\ \ominus & \textit{if } a < 0 \textit{ or } a = \ominus \\ \top & \textit{if } a = \top \end{cases}$$

$$pos(\sum_{i=1}^{n} a_i x_i \ op\ b) = \{x_i \mid \alpha_{\text{Sign}}(a_i) = \oplus\}$$

$$neg(\sum_{i=1}^{n} a_i x_i \ op\ b) = \{x_i \mid \alpha_{\text{Sign}}(a_i) = \ominus\}$$

Definition 3 (Arithmetic extension). *Let* $a_1, a_2 \in \mathbb{R} \cup \{\oplus, \ominus, \top\}$, $op \in \{\times, +\}$. *The results of the arithmetic operation* a_1 *op* a_2 *and of* $-a_1$ *(unary minus) are calculated using standard arithmetic if* a_1 *and* a_2 *are numbers, otherwise they are given as* $\alpha_{\text{Sign}}(a_1)$ *op* $\alpha_{\text{Sign}}(a_2)$ *and* $-\alpha_{\text{Sign}}(a_1)$ *as specified on the right:*

$+$	0	$\oplus$	$\ominus$	$\top$
0	0	$\oplus$	$\ominus$	$\top$
$\oplus$	$\oplus$	$\oplus$	$\top$	$\top$
$\ominus$	$\ominus$	$\top$	$\ominus$	$\top$
$\top$	$\top$	$\top$	$\top$	$\top$

$\times$	0	$\oplus$	$\ominus$	$\top$
0	0	0	0	0
$\oplus$	0	$\oplus$	$\ominus$	$\top$
$\ominus$	0	$\ominus$	$\oplus$	$\top$
$\top$	0	$\top$	$\top$	$\top$

$-$	
0	0
$\oplus$	$\ominus$
$\ominus$	$\oplus$
$\top$	$\top$

To obtain an accurate elimination procedure we need to know the sign of the coefficient of the variable we are eliminating. In case the coefficient is abstract and its sign is unknown (that is, $\top$), we *split* the constraint to obtain a disjunction of constraints in which that coefficient is known.

Definition 4 (Split). *The splitting function on a coefficient* j *is defined by*

$$\mathsf{split}(a) = \begin{cases} \{\oplus, \ominus, 0\} & \textit{if } a = \top \\ \{a\} & \textit{otherwise} \end{cases}$$

$$\mathsf{split}(\sum_{i=1}^{n} a_i x_i \ op\ b, j) = \{a x_j + \sum_{i=1, i \neq j}^{n} a_i x_i \ op\ b \mid a \in \mathsf{split}(a_j)\}$$

$$\mathsf{split}(c_1 \wedge \cdots \wedge c_m, j) = \{\bigwedge_{k=1}^{m} c'_k \mid c'_k \in \mathsf{split}(c_k, j)\}$$

$$\mathsf{split}(D, j) = \bigcup_{C \in D} \mathsf{split}(C, j)$$

Example 2. Given the LPos constraint C (with the constraint involving the coefficient $\top$ highlighted in a box):

$$(x_1 + 2x_2 \leq 2) \ \wedge\ (x_1 + \oplus x_2 \leq \ominus) \ \wedge\ \boxed{(\oplus x_1 + \top x_2 \leq 5)} \ \wedge\ (x_1 \leq 5)$$

the result (the split constraints are indicated in the boxes) of $\mathsf{split}(C, 2)$ is:

$$\left\{ \begin{array}{l} C_{1,1}: \ (x_1 + 2x_2 \leq 2) \ \wedge\ (x_1 + \oplus x_2 \leq \ominus) \ \wedge\ \boxed{(\oplus x_1 + \oplus x_2 \leq 5)} \ \wedge\ (x_1 \leq 5) \\ C_{1,2}: \ (x_1 + 2x_2 \leq 2) \ \wedge\ (x_1 + \oplus x_2 \leq \ominus) \ \wedge\ \boxed{(\oplus x_1 + \ominus x_2 \leq 5)} \ \wedge\ (x_1 \leq 5) \\ C_{1,3}: \ (x_1 + 2x_2 \leq 2) \ \wedge\ (x_1 + \oplus x_2 \leq \ominus) \ \wedge\ \boxed{(\oplus x_1 + 0 x_2 \leq 5)} \ \wedge\ (x_1 \leq 5) \end{array} \right\}$$

Clearly split does not change the interpretation of an abstract constraint:

Proposition 1. *For all* j: $\gamma_{\mathsf{LPos}}(D) = \gamma_{\mathsf{LPos}}(\mathsf{split}(D, j))$.

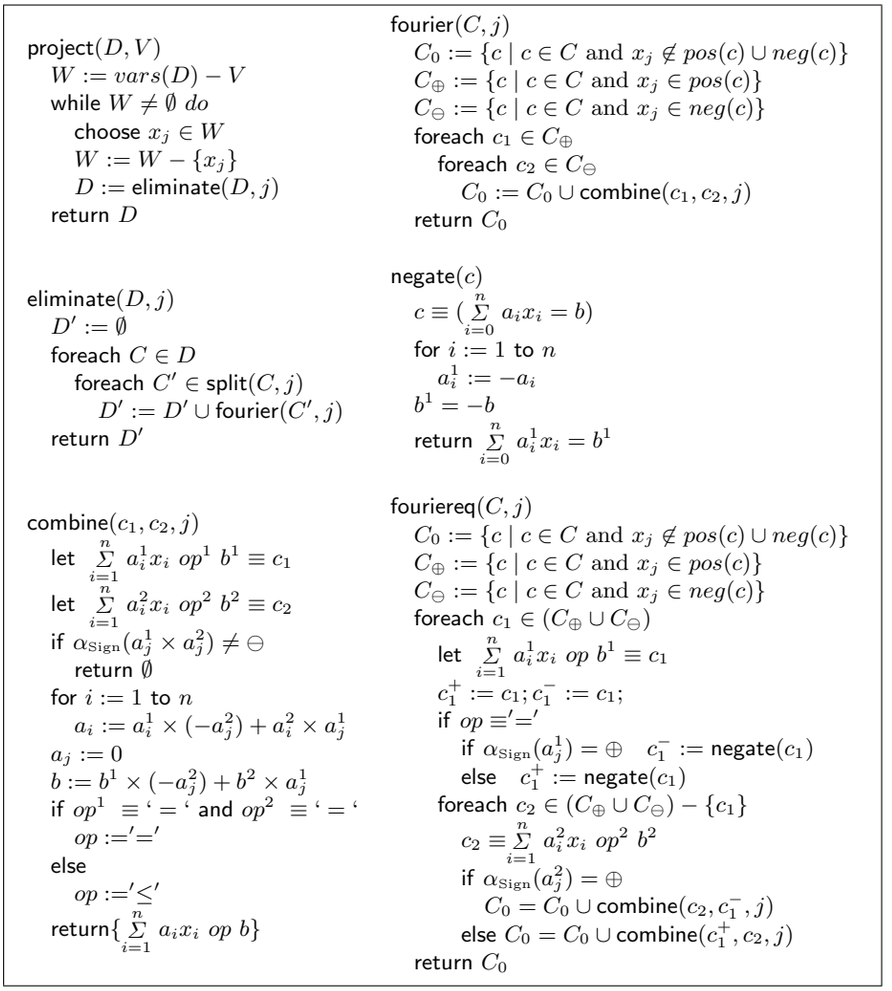


Fig. 3. Projection algorithm for LPos

We can now define *projection*. project(D, V) projects an LPos description D, involving only inequalities, onto a set of variables V. Its definition is given in Figure 3. Note that we can always represent an LPos description using only inequalities by replacing an equality by two inequalities. Equalities can be handled directly by replacing fourier(C, j) by fouriereq(C, j).

Example 3. Consider the LPos description D:

$$\left\{ \begin{array}{ll} C_1: & x_1 + 2x_2 \leq 2 \ \wedge \ x_1 + \oplus x_2 \leq \ominus \ \wedge \ \oplus x_1 + \top x_2 \leq 5 \ \wedge \ x_1 \leq 5 \\ C_2: & x_1 \leq 0 \end{array} \right\}$$

The operation project$(D, \{x_1\})$ proceeds by eliminating the variable x_2 as follows: First we apply eliminate$(D, 2)$ which chooses C_1. Splitting C_1 gives the

constraints described in Example 2. The call to $\mathsf{fourier}(C_{1,1}, 2)$ breaks up $C_{1,1}$ according to the sign of x_2, into

$$\left| C_{\oplus} = \left\{ \begin{array}{l} x_1 + 2x_2 \leq 2,\ x_1 + \oplus x_2 \leq \ominus, \\ \oplus x_1 + \oplus x_2 \leq 5 \end{array} \right\} \right| C_{\ominus} = \emptyset \left| C_0 = \left\{ x_1 \leq 5 \right\} \right|$$

and yields $\{x_1 \leq 5\}$. $\mathsf{fourier}(C_{1,2}, 2)$ breaks up $C_{1,2}$ by the sign of x_2, into

$$\left| C_{\oplus} = \left\{ \begin{array}{l} x_1 + 2x_2 \leq 2 \\ x_1 + \oplus x_2 \leq \ominus \end{array} \right\} \right| C_{\ominus} = \left\{ \oplus x_1 + \ominus x_2 \leq 5 \right\} \left| C_0 = \left\{ x_1 \leq 5 \right\} \right|$$

and returns $\{x_1 \leq 5 \wedge \oplus x_1 \leq \oplus \wedge \oplus x_1 \leq \top\}$. Now, the call to $\mathsf{fourier}(C_{1,3}, 2)$ breaks up $C_{1,3}$ by the sign of x_2, into

$$\left| C_{\oplus} = \left\{ \begin{array}{l} x_1 + 2x_2 \leq 2, \\ x_1 + \oplus x_2 \leq \ominus \end{array} \right\} \right| C_{\ominus} = \emptyset \left| C_0 = \left\{ x_1 \leq 5, \oplus x_1 + 0x_2 \leq 5 \right\} \right|$$

and it yields $\{x_1 \leq 5 \wedge \oplus x_1 \leq 5\}$. The returned values from fourier, together with $x_1 \leq 0$ (the result for C_2) make up the result of eliminate:

$$\left\{ \begin{array}{ll} x_1 \leq 5, & x_1 \leq 5 \wedge \oplus x_1 \leq \oplus \wedge \oplus x_1 \leq \top \\ x_1 \leq 5 \wedge \oplus x_1 \leq 5, & x_1 \leq 0 \end{array} \right\}$$

Widening: The LPos domain contains infinite ascending chains, so we need to use widening. The idea behind the widening is to join abstract primitive constraints in such as way that no groundness or uniqueness dependency is lost.

Definition 5 (Sign equivalence). *Given two abstract primitive* LPos *constraints:*

$$c_1 \equiv \sum_{i=0}^{n} a_i^1 x_i \ op_1 \ b_1 \qquad c_2 \equiv \sum_{i=0}^{n} a_i^2 x_i \ op_2 \ b_2$$

we say that c_1 and c_2 are sign equivalent, denoted $c_1 \sim c_2$ if and only if: $op_1 \equiv op_2$, $\alpha_{\text{Sign}}(a_i^1) \equiv \alpha_{\text{Sign}}(a_i^2)$ $(0 \leq i \leq n)$, and $\alpha_{\text{Sign}}(b_1) \equiv \alpha_{\text{Sign}}(b_2)$. Two LPos *constraints (conjunctions) C_1 and C_2 are sign equivalent, if for each $c_1 \in C_1$ there is a $c_2 \in C_2$ such that $c_1 \sim c_2$, and vice versa.*

Example 4. Consider the following two abstract LPos constraints:

$C_1 \equiv \overbrace{x + \oplus y \leq 10}^{c_1} \wedge \overbrace{x - y \leq 5}^{c_2}$ and $C_2 \equiv \overbrace{x + 4y \leq 4}^{c_3} \wedge \overbrace{x + \ominus y \leq 1}^{c_4}$

c_1, c_3 and c_2, c_4 are sign equivalent, so C_1 and C_2 are sign equivalent. If we add $c_5 \equiv x \leq \oplus$ to C_1, then C_1 and C_2 are no longer sign equivalent, because c_5 is not sign equivalent with c_3 or c_4.

Proposition 2. *Given a finite set of variables $\mathcal{V}$, there are only finitely many non-sign equivalent abstract* LPos *constraints.*

The "widening" that we apply is based on the notion of sign equivalence. This operation is applied when two constraints are sign equivalent, so we do not lose the sign information. This is important for groundness and uniqueness analysis.

Definition 6 (Widening coefficients). *Let* $a_1, a_2 \in \mathbb{R} \cup \{\oplus, \ominus, \top\}$. *Define* $a_1 \sqcup a_2 \stackrel{def}{=} a_1$ *if* $a_1 = a_2$, *otherwise* $a_1 \sqcup a_2 \stackrel{def}{=} \alpha_{\text{Sign}}(a_1)$ *if* $\alpha_{\text{Sign}}(a_1) = \alpha_{\text{Sign}}(a_2)$, *otherwise* $a_1 \sqcup a_2 = \top$. *Extend this to sign equivalent constraints:*

$$\left[\sum_{i=1}^{n} a_i^1 x_i \ op\ b_1\right] \ \sqcup \ \left[\sum_{i=1}^{n} a_i^2 x_i \ op\ b_2\right] = \sum_{i=1}^{n} (a_i^1 \sqcup a_i^2) x_i \ op\ (b_1 \sqcup b_2)$$
$$(c_1^1 \wedge \cdots c_n^1) \sqcup (c_1^2 \wedge \cdots c_m^2) = \bigwedge \{c_i^1 \sqcup c_j \mid c_i^1 \sim c_j^2\}$$

Finally we obtain a widening operation, by widening sign equivalent constraints in the descriptions.

$$\begin{aligned} \mathsf{widen}(D_1, D_2) = \quad & (D_1 \setminus \{C_1 \mid C_1 \in D_1, \exists C_2 \in D_2, C_1 \sim C_2\}) \\ & \cup (D_2 \setminus \{C_2 \mid C_2 \in D_2, \exists C_1 \in D_1, C_1 \sim C_2\}) \\ & \cup \{C_1 \sqcup C_2 \mid C_1 \in D_1, C_2 \in D_2, C_1 \sim C_2\} \end{aligned}$$

Example 5. Let LPos constraints C_1 and C_2 be as in Example 4. Then

$$\mathsf{widen}(\{C_1\}, \{C_2\}) = \{x + \oplus y \leq \oplus \wedge x + \ominus y \leq \oplus\}.$$

5 LPos instead of Pos

In this section we describe the use of LPos for groundness and uniqueness analysis, and show that it is uniformly more accurate than the corresponding Pos analyses. Before going into details we define a splitting function on all variables, which is based on the split given in Definition 4. We use this to map the LPos elements to their Pos description, in both groundness and uniqueness analysis. The split function effectively replaces all $\top$ coefficients by the three choices they represent $\{\oplus, \ominus, 0\}$.

Definition 7 (Splitting on all variables). *Given an* LPos *conjunction* C, *we define the splitting of* C *on all its variables as follows:*

$$\mathsf{splitall}(c_1 \wedge \cdots \wedge c_m) = \{c_1' \wedge \cdots \wedge c_m' \mid c_i' \in \mathsf{splitall}(c_i)\}$$
$$\mathsf{splitall}(\sum_{i=1}^{n} a_i x_i \ op\ b) = \mathsf{splitall}^*(\sum_{i=1}^{n} a_i x_i \ op\ b, n)$$
$$\mathsf{splitall}^*(c, n) = \begin{cases} \{c\} & \text{if } \ n = 0 \\ \bigcup_{c' \in \mathsf{split}(c,n)} \mathsf{splitall}^*(c', n-1) & \text{otherwise} \end{cases}$$

Example 6. Consider the LPos constraint $\oplus x + \top y \leq \oplus \wedge \top x + \ominus y = \ominus$. Splitting on all variables, that is, on x and y, gives a set of nine abstract constraints:

$$\mathsf{splitall}(C) =$$
$$\left\{\begin{array}{ll} \oplus x + 0y \leq \oplus \ \wedge 0x + \ominus y = \ominus, & \oplus x + \oplus y \leq \oplus \wedge 0x + \ominus y = \ominus \\ \oplus x + \ominus y \leq \oplus \wedge 0x + \ominus y = \ominus, & \oplus x + 0y \leq \oplus \ \wedge \oplus x + \ominus y = \ominus \\ \oplus x + \oplus y \leq \oplus \wedge \oplus x + \ominus y = \ominus, & \oplus x + \ominus y \leq \oplus \wedge \oplus x + \ominus y = \ominus \\ \oplus x + 0y \leq \oplus \ \wedge \ominus x + \ominus y = \ominus, & \oplus x + \oplus y \leq \oplus \wedge \ominus x + \ominus y = \ominus \\ \oplus x + \ominus y \leq \oplus \wedge \ominus x + \ominus y = \ominus & \end{array}\right\}$$

By Proposition 1 we can eliminate $\top$ coefficients using splitall. ¿From here on we therefore assume that no LPos description involves $\top$.

Groundness analysis of logic programs: Given a logic program P, the meaning of P over the LPos domain is obtained in two stages: the first is abstracting the Herbrand terms to abstract LPos constraints using the var-norm function, and the second is computing the meaning of the program. For simplicity, we will use a Kleene iteration combined with *widening*, in order to compute an approximation of the least fixed point of the equation generated by the abstraction function.

The abstract LPos version P' of a normalised logic program P is obtained as follows: For each clause $C \equiv p(\overline{x}) \leftarrow b_1, \ldots, b_n$ we generate an equation:

$$p(\overline{x}) = \overline{\exists}\overline{x} : b'_1 \wedge \ldots \wedge b'_n \wedge \bigwedge\{v \in vars(C) \mid (v \geq 0)\}$$

where each b'_i is a translation of b_i according to this rule: $b'_i \equiv \mathsf{var\text{-}norm}(s) \leq \mathsf{var\text{-}norm}(t) \wedge \mathsf{var\text{-}norm}(t) \leq \mathsf{var\text{-}norm}(s)$ if $b_i \equiv (s = t)$, otherwise $b'_i = b_i$. The operations $\wedge$ and $\overline{\exists}$ are $\sqcap_{\mathsf{LPos}}$ and project respectively. The *join* operation will be denoted by $\vee$. The Kleene iteration is performed in the standard way, but after each step (or finitely many steps) we apply a widening.

Similar to Size, LPos objects describe groundness and can be translated to Pos descriptions according to the description function defined below.

Definition 8 (Description function $\alpha_{\mathcal{G}}$). *We can extract groundness information from an* LPos *object as follows:*

$$\alpha_{\mathcal{G}}(c) = \begin{cases} \bigvee_{c' \in \mathsf{splitall}(c)} (\bigwedge neg(c') \rightarrow \bigwedge pos(c')) & \text{if } c \text{ is an inequality} \\ \bigvee_{c' \in \mathsf{splitall}(c)} (\bigwedge neg(c') \leftrightarrow \bigwedge pos(c')) & \text{if } c \text{ is an equation} \end{cases}$$

$$\alpha_{\mathcal{G}}(C) = \bigwedge_{i=1}^{n} \alpha_{\mathcal{G}}(c_i), \quad \text{where } C \equiv c_1 \wedge \cdots \wedge c_n$$

$$\alpha_{\mathcal{G}}(D) = \bigvee_{C \in D} \alpha_{\mathcal{G}}(C)$$

Example 7. Let us (re)analyse the listof program, this time using LPos, and show how LPos gives better information than Size. First we use the var-norm function to abstract the program to the following recursive equations:

$$\begin{aligned} \mathsf{listof}(x, xs) = \quad & -x \leq 0 \wedge -xs \leq 0 \wedge xs = x \\ & \vee [\exists ys : -x \leq 0 \wedge -xs \leq 0 \wedge -ys \leq 0 \wedge xs = x + ys \wedge \mathsf{listof}(x, ys)] \end{aligned}$$

and two steps of Kleene iteration yield:

$$\begin{aligned} \mathsf{listof}^1(x, xs) &= [-x \leq 0 \wedge -xs \leq 0 \wedge xs = x] \\ \mathsf{listof}^2(x, xs) &= [-x \leq 0 \wedge -xs \leq 0 \wedge xs = x] \\ &\quad \vee [\exists ys : -x \leq 0 \wedge -xs \leq 0 \wedge -ys \leq 0 \wedge xs = x + ys \wedge ys = x)] \\ &= [-x \leq 0 \wedge -xs \leq 0 \wedge xs = x] \vee [-x \leq 0 \wedge -xs \leq 0 \wedge xs = x + x] \end{aligned}$$

Applying a widening on $\mathsf{listof}^1(x, xs)$ and $\mathsf{listof}^2(x, xs)$ gives

$$C = -x \leq 0 \wedge -xs \leq 0 \wedge xs = \oplus x$$

which is a fixed point, and its Pos description is $\alpha_{\mathcal{G}}(C) = xs \leftrightarrow x$.

Lemma 1. *Let D_1, D_2 be elements of* LPos *and $\mathcal{V}$ a set of variables. Then:* **(a)** $\alpha_{\mathcal{G}}(D_1 \sqcup_{\mathsf{LPos}} D_2) \Leftrightarrow \alpha_{\mathcal{G}}(D_1) \vee \alpha_{\mathcal{G}}(D_2)$ *and* $\alpha_{\mathcal{G}}(D_1 \sqcap_{\mathsf{LPos}} D_2) \Leftrightarrow \alpha_{\mathcal{G}}(D_1) \wedge \alpha_{\mathcal{G}}(D_2)$*;* **(b)** $\alpha_{\mathcal{G}}(\mathsf{project}(D_1, \mathcal{V})) \Rightarrow \overline{\exists}\mathcal{V} : \alpha_{\mathcal{G}}(D_1)$*; and* **(c)** $\alpha_{\mathcal{G}}(D_1 \sqcup_{\mathsf{LPos}} D_2) = \alpha_{\mathcal{G}}(\mathsf{widen}(D_1, D_2))$.

Theorem 1. (a) *$\alpha_{\mathcal{G}}$ correctly (and accurately) extracts groundness information from* LPos *descriptions; and* **(b)** LPos *is uniformly at least as accurate as* Pos *for groundness analysis of logic programs.*

Uniqueness analysis: With LPos, the analysis of a constraint program has two stages, just like groundness analysis has. Only the abstraction function differs. Given a normalised linear arithmetic constraint program P, an abstract LPos version P' is obtained as follows: For each clause $p(\overline{x}) \leftarrow b_1, \ldots, b_n$ generate the equation: $p(\overline{x}) = \overline{\exists}\overline{x} : b'_1 \wedge \ldots \wedge b'_n$ where b'_i is a translation of b_i: if b_i is an equation or a call, $b'_i = b_i$, otherwise $b'_i = true$. The operations $\wedge$, $\vee$ and $\overline{\exists}$ are the same as in groundness analysis, And Kleene iteration is performed similarly. It is clear that the LPos object obtained in this analysis, is an approximation of the concrete object of P, so a uniqueness analysis is obtained by using the following description function. below.

Definition 9 (Description function $\alpha_{\mathcal{U}}$). *We can extract uniqueness information from an* LPos *objects as follows:*

$$\alpha_{\mathcal{U}}(c) = \begin{cases} true & \text{if C is an inequality} \\ \bigvee_{c \in \mathsf{splitall}(C)} \nabla(vars(c)) & \text{if C is an equation} \end{cases}$$

$$\alpha_{\mathcal{U}}(C) = \bigwedge_{i=1}^{n} \alpha_{\mathcal{U}}(c_i), \quad \text{where } C \equiv c_1 \wedge \cdots \wedge c_n$$

$$\alpha_{\mathcal{U}}(D) = \bigvee_{C \in D} \alpha_{\mathcal{U}}(C)$$

Example 8. Using LPos, the abstract version of Figure 2 is

$$\begin{aligned} \mathsf{mg}(p,t,r,b) = \quad & [(t=1) \wedge (b = cp - r)] \\ & \vee [\exists t_1, p_1 : (t_1 = t-1) \wedge (p_1 = cp - r) \wedge \ \mathsf{mg}(p_1, t_1, r, b)] \end{aligned}$$

Kleene iteration proceeds as follows:

$$\begin{aligned} \mathsf{mg}^1(p,t,r,b) = \quad & [(t=1) \wedge (b = cp - r)] \\ \mathsf{mg}^2(p,t,r,b) = \quad & [(t=1) \wedge (b = cp - r)] \\ & \vee [\exists t_1, p_1 : (t_1 = t-1) \wedge (p_1 = cp - r) \wedge \ (t_1 = 1) \wedge (b = cp_1 - r)] \\ = \quad & [(t=1) \wedge (b = cp - r)] \\ & \vee [(t=2) \wedge (b + (1+c)r - c^2 p) = 0] \end{aligned}$$

and a widening on $\mathsf{mg}^1(p,t,r,b)$ and $\mathsf{mg}^2(p,t,r,b)$ yields:

$$\mathsf{mg}^3(p,t,r,b) \ = \ (t = \oplus) \wedge (b + \oplus r + \ominus p = 0)$$

which is a fixed point. Its description in Pos is

$$t \wedge ((p \wedge r) \rightarrow b) \wedge ((b \wedge r) \rightarrow p) \wedge ((b \wedge p) \rightarrow r)$$

which is more precise than the one we obtained using Pos, namely $t \wedge ((p \wedge r) \rightarrow b)$.

Lemma 2. *Let D_1 and D_2 be elements of* LPos *and $\mathcal{V}$ a set of variables. Then:* **(a)** $\alpha_{\mathcal{U}}(D_1 \sqcup_{\mathsf{LPos}} D_2) \Rightarrow \alpha_{\mathcal{U}}(D_1) \vee \alpha_{\mathcal{U}}(D_2)$ *and* $\alpha_{\mathcal{U}}(D_1 \sqcap_{\mathsf{LPos}} D_2) \Rightarrow \alpha_{\mathcal{U}}(D_1) \wedge \alpha_{\mathcal{U}}(D_2)$*;* **(b)** $\alpha_{\mathcal{U}}(\mathsf{project}(D_1, \mathcal{V})) \Rightarrow \overline{\exists}\mathcal{V} : \alpha_{\mathcal{U}}(D_1)$*; and* **(c)** $\alpha_{\mathcal{U}}(D_1 \sqcup_{\mathsf{LPos}} D_2) = \alpha_{\mathcal{U}}(\mathsf{widen}(D_1, D_2))$.

Theorem 2. **(a)** *$\alpha_{\mathcal{U}}$ correctly (and accurately) extracts uniqueness information from* LPos *descriptions; and* **(b)** LPos *is uniformly at least as accurate as* Pos *for uniqueness analysis of* $\mathsf{CLP}(\mathbb{R}_{\mathsf{Lin}})$ *programs.*

6 Conclusion

We have shown how the well-known Pos domain occasionally is less than adequate for groundness and uniqueness analysis. For example, a Pos-based analysis of the celebrated "mortgage" program exhibits a significant loss of precision.

A partial solution can be obtained by using instead a domain Size of term size constraints. However, we have shown that this solution is not entirely satisfactory. In fact even the reduced product [10] of Pos and Size has shortcomings for uniqueness analysis. For this reason, inspired by the LSign domain, we proposed LPos as an improved Pos for groundness and uniqueness analysis.

A different attempt to improve the precision of Pos-based analysis was made by Filé and Ranzato [13] who considered the use of the powerset of Pos, $\mathcal{P}(\mathsf{Pos})$. However, while this increases the expressiveness of the abstract domain, it can be argued that groundness analysis cannot capitalise on the improved precision, since the Pos content of the result of a $\mathcal{P}(\mathsf{Pos})$ analysis is exactly that of a Pos analysis [14]. (This is interesting because it differs from the situation where a smaller domain is chosen as starting point, for example, the result of a Def analysis [1] may be a less precise than the Def content of a $\mathcal{P}(\mathsf{Def})$ analysis).

Marriott and Stuckey originally introduced the LSign domain [20] to derive information about the satisfiability of linear arithmetic constraints. LSign could be used for groundness and uniqueness analysis also, but it is less precise than what we can obtain in LPos. LPos extends the original LSign in the same direction as the LInt interval improvement of LSign [21], by using both concrete coefficients (size 0 intervals) and abstract coefficients (semi-infinite intervals) and sets of abstract constraints. Unlike LSign and LInt, LPos does not use abstract Gauss-Jordan variable elimination. Interestingly, using this elimination algorithm, the accuracy theorems *fail* to hold. The Fourier elimination creates redundant constraints, and these are anathema to the LSign goal of detecting satisfiability, but they do not reduce the accuracy of the resulting LPos descriptions.

References

1. T. Armstrong, K. Marriott, P. Schachte, and H. Søndergaard. Two classes of Boolean functions for dependency analysis. *Science of Computer Programming*, 31(1):3–45, 1998.
2. N. Baker and H. Søndergaard. Definiteness analysis for CLP($\mathcal{R}$). In G. Gupta *et al.*, editor, *Proc. 16th Australian Computer Science Conf.*, pp. 321–332, 1993.

3. F. Benoy and A. King. Inferring argument size relationships with CLP($\mathcal{R}$). In J. Gallagher, editor, *Logic Program Synthesis and Transformation*, *LNCS*, vol. 1207, pp. 204–223. Springer, 1997.
4. M. Codish and B. Demoen. Deriving polymorphic type dependencies for logic programs using multiple incarnations of Prop. In B. Le Charlier, editor, *Static Analysis: Proc. First Int. Symp.*, *LNCS*,vol. 864, pp. 281–296. Springer, 1994.
5. M. Codish and C. Taboch. A semantic basis for the termination analysis of logic programs. *J. Logic Programming*, 41(1):103–123, 1999.
6. P. Codognet and G. Filé. Computations, abstractions and constraints in logic programs. In *Proc. 4th IEEE Int. Conf. Computer Languages*, pp. 155–164. IEEE Comp. Soc. Press, 1992.
7. A. Cortesi, G. Filé, and W. Winsborough. *Prop* revisited: Propositional formula as abstract domain for groundness analysis. In *Proc. 6th Annual IEEE Symp. Logic in Computer Science*, pp. 322–327. IEEE Computer Society Press, 1991.
8. A. Cortesi, B. Le Charlier, and P. Van Hentenryck. Evaluation of the domain Prop. *Journal of Logic Programming*, 23(3):237–278, 1995.
9. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. 4th POPL*, pp. 238–252. ACM Press, 1977.
10. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proc. 6th POPL*, pp. 269–282. ACM Press, 1979.
11. P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In M. Bruynooghe and M. Wirsing, editors, *PLILP*, *LNCS*, vol. 631, pp. 269–295. Springer, 1992.
12. C. Fecht. *Abstrakte Interpretation logischer Programme: Theorie, Implementierung, Generierung*. PhD thesis, Universität des Saarlandes, Germany, 1997.
13. G. Filé and F. Ranzato. Improving abstract interpretations by systematic lifting to the powerset. In M. Bruynooghe, editor, *Logic Programming: Proc. 1994 Int. Symp.*, pp. 655–669. MIT Press, 1994.
14. G. Filé and F. Ranzato. The powerset operator on abstract interpretations. *Theoretical Computer Science*, 222:77–111, 1999.
15. R. Giacobazzi, S. Debray, and G. Levi. Generalized semantics and abstract interpretation for constraint logic programs. *J. Logic Programming*, 25(3):191–247.
16. N. Lindenstrauss and Y. Sagiv. Automatic termination analysis of logic programs. In L. Naish, editor, *Logic Programming: Proc. 14th Int. Conf. Logic Programming*, pp. 63–77. MIT Press, 1997.
17. K. Marriott. Frameworks for abstract interpretation. *Acta Informatica*, 30(2):103–129, 1993.
18. K. Marriott and H. Søndergaard. Notes for a tutorial on abstract interpretation of logic programs. 1989 North American Conf. Logic Programming.
19. K. Marriott and H. Søndergaard. Precise and efficient groundness analysis for logic programs. *ACM Letters on Programming Languages and Systems*, 2(1–4):181–196.
20. K. Marriott and P. J. Stuckey. Approximating interaction between linear arithmetic constraints. In M. Bruynooghe, editor, *Logic Programming: Proc. 1994 Int. Symp.*, pp. 571–585. MIT Press, 1994.
21. V. Ramachandran, P. Van Hentenryck, and A. Cortesi. Abstract domains for reordering CLP($\Re_{Lin}$) programs. *J. Logic Programming*, 42(3):217–256, 2000.
22. Peter Schachte. *Precise and Efficient Static Analysis of Logic Programs*. PhD thesis, The University of Melbourne, Australia, 1999.
23. A. Van Gelder. Deriving constraints among argument sizes in logic programs. In *Proc. 9th ACM Conf. Principles of Database Systems*, pp. 47–60. ACM Press, 1990.

Speculative Beats Conservative Justification*

Hai-Feng Guo, C.R. Ramakrishnan, and I.V. Ramakrishnan

State University of New York at Stony Brook
Stony Brook, NY 11794, USA
{haifeng,cram,ram}@cs.sunysb.edu

Abstract. Justifying the truth value of a goal resulting from query evaluation of a logic program corresponds to providing evidence, in terms of a proof, for this truth. In an earlier work we introduced the notion of justification [8] and gave an algorithm for justifying tabled logic programs by *post-processing* the memo tables created during evaluation. A *conservatve justifier* such as the one described in that work proceeds in two separate stages: evaluate the truth of literals (that can possibly contribute to the evidence) in the first stage and construct the justification in the next stage. Justifications built in this fashion seldom fail. Whereas for tabled predicates evaluation amounts to a simple table look-up during justification, for non-tabled predicates this amounts to Prolog-style re-execution. In a conservative justifier a non-tabled literal can be re-executed causing unacceptable performance overheads for programs with significant non-tabled components: justification time for a single non-tabled literal can become *quadratic* in its evaluation time!
In this paper we introduce the concept of a *speculative justifier*. In such a justifier we evaluate the truths of literals in tandem with justification. Specifically, we select literals that can possibly provide evidence for the goal's truth, assume that their truth values correspond to the goal's and proceed to build a justification for each of them. Since these truths are not computed before hand, justfications produced in this fashion may fail often. On the other hand non-tabled literals are re-executed less often than conservative justifiers. We discuss the subtle efficiency issues that arise in the construction of speculative justifiers. We show how to judiciously balance the different efficiency concerns and engineer a speculative justifier that addresses the performance problem associated with conservative justifiers. We provide experimental evidence of its efficiency and scalability in justifying the results of our XMC model checker.

1 Introduction

Query evaluation of a goal with respect to a logic program establishes the truth or falsehood of the goal. However the underlying evaluation engine typically provides little or no information as to why the conclusion was reached. This problem broadly falls under the purview of debugging. Usually logic programs

* Research partially supported by NSF awards EIA-9705998, CCR-9876242, IIS-0072927, and EIA-9901602.

P. Codognet (Ed.): ICLP 2001, LNCS 2237, pp. 150–165, 2001.

are debugged using trace-based debuggers (e.g. Prolog's four-port debugger) that operate by tracing through the entire proof search. Such traces are aided through several navigation mechanisms (e.g. setting breakpoints or spy points, skips, leaps, etc.) provided by the debugger.

There are several reasons why trace-based debuggers are cumbersome to use. Firstly, they give the entire search sequence including all the failure paths, which is essentially irrelevant if the user is only interested in comprehending the essential aspects of how the answer was derived. Secondly, the proof search strategy of Prolog, with its forward and backward evaluation, already makes tracing a Prolog execution considerably harder than tracing through procedural programs. The problem is considerably exacerbated for tabled logic programs since the complex scheduling and fixed-point computing strategies of tabled resolution makes it very difficult to comprehend the sequence produced by a tracer. Finally, from our own experience with the XMC model checker [1] (which is an application of the XSB tabled logic programming system [11]) trace-based debuggers provide no support for translating the results of the trace (which is at the logic program evaluation level) to the problem space (e.g. CCS expressions and modal-mu calculus formulas in XMC).

In [8] we proposed the concept of a *justifier* for giving evidence, in terms of a proof, for the truth value of the result generated by query evalaution of a logic program. The essence of justification is to succinctly convey to the user only those parts of the proof search which are relevant to the proof/disproof of the goal. For example, if a query is evaluated to true, the justifier will present the details of a successful computation path, completely ignoring any unsuccessful paths traversed. Similarly, when a query is evaluated to false, it will only show a false literal in each of its computation paths, completely ignoring the true literals. Figure 1 is an illustration of justification, where the predicate `reach/2` (Figure 1a) is tabled. Evaluation of the query `reach(a,d)` generates a forest of search trees (Figure 1b), (See [12] for an overview of tabled evaluation.)

Although justification is a general concept, the focus of our earlier work in [8] was on justifying tabled logic programs. Towards that end we presented an algorithm for justifying such programs by *post-processing* the memo tables created during query evaluation. To justify the answer to a query some "footprints" need to be stored during query evaluation. The justifier uses these footprints to extract evidence supporting the result. The naturalness of using a tabled LP system for justification is that the answer tables created during query evaluation serve as the footprints. Indeed during query evaluation the internally created tables implicitly represent the lemmas that are proved during evaluation. By using these lemmas stored in the tables, the justifier presents only relevant parts of the derivation to the user. In other words the additional information needed for doing justification comes for "free". Thus justification using tabled logic programming system is "non-intrusive" in the sense that it is completely decoupled from query evaluation process and is done only after the latter is completed. More importantly, justification is done without compromising the performance of query evaluation.

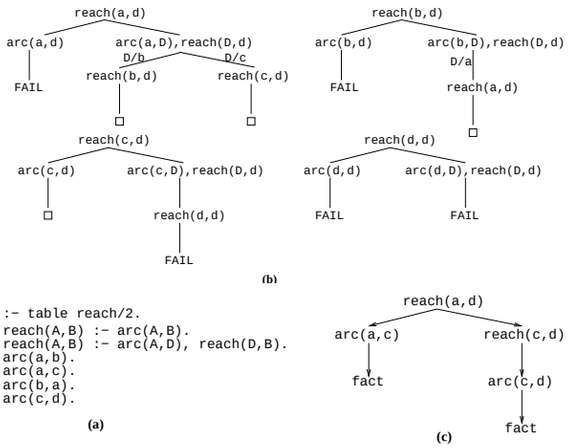


Fig. 1. Justifying reach(a,d): (a) Logic Program (b) Forest of Search Trees (c) Justification

Justifying the truth value of a given literal which we will denote as the goal, amounts to providing a proof that usually will involve searching for other literals relevant to the proof, knowing their truth values, justifying each such truth value and putting them all together to produce a justification of the goal's truth. For some of them we may fail to produce justifications relevant for justifying the goal. In Example 1 below the clause `p :- r` is irrelevant for justifying `p` is *true* since the failure of `r` is not the correct evidence for `p`'s truth. Had we selected this clause and proceeded to build a justification for `r` we would have eventually discovered that it is irrelevant. Thus avoiding irrelevant justifications is an important parameter in the design of justification algorithms.

Example 1 *Consider the following logic program:*

```
p :- r.        p :- t.
r :- ..., fail.
t.
```

The justification algorithm in [8] yields a *conservative justifier* in the sense that by design it is geared towards limiting such wasteful justifications. It does so by evaluating the truth of literals (that can possibly provide supporting evidence for the goal's truth) in the first stage. Armed with the needed truths, in a separate second stage it proceeds to construct their justifications. By evaluating the truth of `r` before hand upon selecting the clause `p :- r` in Example 1, we can avoid building the justfication of `r` to support the truth of `p` and fail eventually.

The algorithm in [8] implicitly assumed that all the predicates in the program are tabled. But real-life logic programs consist of both tabled and non-tabled predicates. How does it handle such programs? Whereas for tabled predicates evaluation is a simple table look-up during justification, for non-tabled predicates this amounts to Prolog-style re-execution. In a conservative justifier, justification of a non-tabled literal can trigger repeated evaluations of other non-tabled literals

on its proof path, causing unacceptable performance overheads for programs with significant non-tabled components. Specifically the time taken to justify the truth of a single non-tabled literal can become *quadratic* over its evaluation time! In fact on large model checking problems our XMC model checker took a few minutes to produce the results whereas the justifier failed to produce a justification even after sevaral hours!

In this paper we explore the concept of a *speculative justifier* to address the above performance problem associated with a conservative justifer. The idea underlying such a justfier is this: When we select a literal as a possible candidate for inclusion in the justificaton of the goal's truth we speculate that it will be relevent and proceed to build its justification. Since we do not know its truth value before hand we may discover eventually that we are unable to produce a justification for it that is relevant for justifying the goal's truth (such as the justification of `r` in Example 1). On the other hand if we never encounter any such literal then for a non-tabled literal we have built its justification without having to repeatedly traverse its proof path. But doing speculative justification naively can result in failing more often and thus offset any gains accrued by avoiding repeated re-executions of non-tabled literals. In this paper we discuss these subtle efficiency issues that arise in the design and implementation of speculative justifiers. We show how to judiciously balance the different efficiency concerns and engineer a speculative justifier that addresses the performance problem associated with conservative justifiers. The rest of the paper is organized as follows. In Section 2 we review the concept of justification. Section 3 reviews conservative justifier. In section 4 we present the design of a speculative justifier. In Section 5 we discuss its implementation and practical impact on real-world applications drawn from model checking. Discussion appears in Section 6. The technical machinery developed in this paper assumes *definite clause logic program*. Extensions are also disussed in Section 6.

Related Work. A number of proposals to explain the results of query evaluation of logic programs have been put forth in the past. These include algorithmic debugging techniques [10], declarative debugging techniques [4,6], assertion based debugging techniques [7], and explanation techniques [5]. A more detailed comparison between justification and these aproaches appears in our earlier work [8]. Suffice it is say here that although justification is similar in spirit to the above approaches in terms of their objectives it differs considerably from all them. It is done as a post-processing step after query evaluation, and not along with the query evaluation (as in algorithmic and assertion-based debugging) or before query evaluation (as in declarative and assertion-based debugging). Unlike declarative debugging justification does not demand any creative input from the user regarding the intended model of the program which can be very hard or even impossible to to do as will be the case in model checking. But beyond all that this paper examines effciency issues that arise in justifying logic programs consisting of both tabled and non-tabled predicates – a topic that has not been explored in the literature.

2 Justification

In this section we will recall the formalisms developed in [8] for justification. We generalize them here in order to deal with mixed programs containing both tabled and non-tabled predicates.

Notational Conventions We use P to denote logic programs; $HB(P)$, $M(P)$ to denote the Herbrand Base and least Herbrand model respectively; `A` and `B` to denote atoms or literals; α to denote a set of atoms or literals; β to denote a conjunction of atoms (a goal is a conjunction of atoms) or literals; θ to denote substitutions; '$\succeq$' to denote atom subsumption ($A \succeq B$ for A subsumes B); and C to denote a clause in a program. For a binary relation R, we denote its (reflexive) transitive closure by R^*.

Definition 1 (Truth Assignment) *The truth assignment of an atom A with respect to program P, denoted by $\tau_{(P)}(A)$, is:*

$$\tau_{(P)}(A) = \begin{cases} true & \forall\theta\ A\theta \in M(P) \\ false & \forall\theta\ A\theta \notin M(P) \end{cases}$$

We drop the parameter P and write the truth assignment as $\tau(A)$ whenever the program is obvious from the context. Let A be an answer to some query in program P, i.e., $\tau(A) = true$. We can complete one step in explaining this answer by finding a clause C such that (i) A unifies with the head of C, and (ii) each literal B in the body of C has $\tau(B) = true$. If A is not an answer to any query, i.e., $\tau(A) = false$, we can explain this failure by showing that for each clause C whose head unifies with A, there is at least one literal B in C such that $\tau(B) = false$. We call such one-step explainations as a *locally consistent explanations (lce's)*, defined formally as follows:

Definition 2 (locally consistent explanation (lce)) *Locally consistent explanation for an atom A with respect to program P, denoted by $\xi_{(P)}(A)$, is a collection of sets of atoms such that:*

1. If $\tau(A) = true$:
   $\xi_{(P)}(A) = \{\alpha_1, \alpha_2, \ldots, \alpha_m\}$, with each α_i being a set of atoms $\{B_1, B_2, \ldots, B_n\}$ such that:
   (a) $\forall\ 1 \leq j \leq n\ \ \tau(B_j) = true$, and
   (b) $\exists\ C \equiv A' :- \beta$ and a substitution θ such that $A'\theta = A$ and $\beta\theta \equiv (B_1, B_2, \ldots, B_n)\theta$.
2. If $\tau(A) = false$:
   $\xi_{(P)}(A) = \{L\}$, a singleton collection where $L = \{B_1, B_2, \ldots, B_n\}$ is the smallest set such that:
   (a) $\forall 1 \leq j \leq n\ \ \tau(B_j) = false$, and
   (b) $\forall$ substitutions θ and $C \equiv A' :- (B'_1, B'_2, \ldots, B'_l)$, $A'\theta = A\theta \implies \exists 1 \leq k \leq l$ such that $B'_k\theta \in L$ and $\forall\ 1 \leq i < k\ \ \tau(B'_i\theta) = true$.

$\xi(reach(a,d)) = \{\{arc(a,c), reach(c,d)\}\}$
$\xi(reach(c,d)) = \{\{arc(c,d)\}\}$
$\xi(arc(c,d)) = \{\{\}\}$
$\xi(reach(a,c)) = \{\{arc(a,c)\},\ \{arc(a,b), reach(b,c)\}\}$
(a) lce's for true literals

$\xi(reach(a,e)) = \{\{arc(a,e), reach(b,e), reach(c,e)\}\}$
$\xi(reach(b,e)) = \{\{arc(b,e), reach(a,e)\}\}$
$\xi(arc(a,e)) = \{\{\}\}$
(b) lce's for false literals

Fig. 2. A fragment of lce's for the example in Figure 1

We write $\xi_{(P)}(A)$ as $\xi(A)$ whenever the program P is clear from the context.

Observe that, for an atom A, the different sets in the collection $\xi(A)$ represent different consistent explanations for the truth or falsehood of A. An answer A can be explained in terms of answers $\{B_1, B_2, \ldots, B_k\}$ in $\xi(A)$ and then (recursively) explaining each B_i. e.g. $\xi(reach(a,d))$ in Figure 2 has a set with elements $arc(a,c)$ and $reach(c,d)$, meaning that the truth value (*true*) of $reach(a,d)$ can be explained using the explanations of $arc(a,c)$ and $reach(c,d)$. Such explanations can be captured by a graph as shown in Figure 1(c). The edges denote locally consistent explanations. We do not use cyclic explanations to justify a true literal. In contrast, cyclic explanations describe infinite proof paths and can be used to justify a false literal. Instead of explicitly representing these cycles, however, we choose to keep the justification as an acyclic graph, breaking each cycle by redirecting at least one edge to a special node marked as `ancestor`. Formally:

Definition 3 (Justification) *A justification for an atom A with respect to program P, denoted by $\mathcal{J}(A,P)$, is a directed acyclic graph $G = (V,E)$ with vertex labels chosen from $HB(P) \cup \{$`fact`,`fail`, `ancestor`$\}$ such that:*

1. G `is rooted at` A`, and is connected`
2. $(B_1, \texttt{fact}) \in E \iff \{\} \in \xi(B_1) \wedge \tau(B_1) = \mathit{true}$
3. $(B_1, \texttt{fail}) \in E \iff \{\} \in \xi(B_1) \wedge \tau(B_1) = \mathit{false}$
4. $(B_1, \texttt{ancestor}) \in E \iff \tau(B_1) = \mathit{false} \wedge \xi(B_1) = \{L\}$
   $\wedge\ \exists\ B_2 \in L$ `s.t.` $(B_2, B_1) \in E^* \ \vee\ B_2 = B_1$
5. $(B_1, B_2) \in E \ \wedge\ \tau(B_1) = \mathit{false} \iff$
   $\xi(B_1) = \{L\} \ \wedge\ B_2 \in L \ \wedge\ (B_2, B_1) \notin E^* \ \wedge\ B_2 \neq B_1$
6. $(B_1, B_2) \in E \ \wedge\ \tau(B_1) = \mathit{true} \implies$
   $\exists L \in \xi(B_1)$ `s.t.` $B_2 \in L \ \wedge\ \{\forall B' \in L\ (B', B_1) \notin E^* \ \wedge\ B' \neq B_1\}$
7. $B_1 \in V \ \wedge\ \tau(B_1) = \mathit{true} \implies \exists\ \mathit{unique}\ L \in \xi(B_1)$ `s.t.`
   $\forall B_2 \in L\ (B_1, B_2) \in E \ \wedge\ (B_2, B_1) \notin E^* \ \wedge\ B_2 \neq B_1$

Rule 1 ensures that A is the root of justification. Rules 2 and 3 are the conditions for adding leaf nodes based on facts. Rules 4 and 5 specifies conditions for justifying false literals, while Rules 6 and 7 deal with true literals.

We will denote the justification graph built for a true (false) literal as *true-justification (false-justification)*.

e.g.. the true-justification in Figure 1(c) is built as follows: $reach(a, d)$ is the root (by rule 1). Now consider the lce $\{arc(a, c), reach(c, d)\}$ in $\xi(reach(a, d))$. Since every element in this lce does not form a cyclic explanation, and is different from $reach(a, d)$, both edges $(reach(a, d),\ arc(a, c))$ and $(reach(a, d), reach(c, d))$ are added to the justification (by Rule 6). Rule 7 guantees that one and only one lce is added into the justification. Next we construct true-justifications for $arc(a, c)$ and $reach(c, d)$ recursively.

3 Conservative Justifier

We review our algorithm in [8] to construct the justification graph. Its high-level aspects are skecthed in Figure 3. V denotes the vertices (labelled by literals in the ξ's) and E denotes the edges in this graph.

Given a literal A the algorithm builds the graph recursively, traversing it depth-first even as it is constructed. At any point, V is the set of "visited" vertices, and *Done* is the set of vertices whose descendents have been completely explored. $V - Done$ contains exactly those vertices that are ancestors to the current vertex A.

```
algorithm Justify(A : atom)
    (* Global: P : program, (V, E): Justification, Done ⊆ V *)
  if (A ∉ V) then (* A has not yet been justified *)
    set V := V ∪ {A}
    if (τ(A) = true) then (* true-justification *)          (1)
      let α_A ∈ ξ(A) such that (α_A ∩ V) ⊆ Done             (2)
      if (α_A = {}) then
        set E := E ∪ (A, fact)
      else
        for each B ∈ α_A do
          set E := E ∪ (A, Justify(B))
    else (* false-justification *)
      let {α_A} = ξ(A)                                      (3)
      if (α_A = {}) then
        set E := E ∪ (A, fail)
      else
        if ((α_A ∩ V) ⊈ Done) then
          set E := E ∪ (A, ancestor)
        for each B ∈ (α_A − (V − Done)) do
          set E := E ∪ (A, Justify(B))
    set Done := Done ∪ {A}
```

Fig. 3. Justification Algorithm

The algorithm is structured as follows: it takes the literal A whose truth is to be justified as the input parameter. It will determine a locally consistent explanation for either a true-justification in case $\tau(A) = true$ (line 2) or a false-justification otherwise (line 3). Finally it justifies the literals in the explaination

set recursively. The selection of the justification is done by backtracking through **let**. Correctness of *Justify* appears in [8].

Algorithm *Justify* in [8] had assumes that all the predicates in the program are atbled. Let us analyse its behavior on "mixed" programs containing both tabled and non-tabled predicates. Observe that the algorithm computes the explanation set for A prior to building the justification graph rooted at A. Computing the explanation set corresponds to evaluating the truth values of literals in the set. Observe that this evaluation is done prior to justifying the truths of the literals in α_A. This ensures that the justifications of the truths of literals in α_A do not fail. In fact the only time a justification gets discarded is when there is a cycle in a true-justification. Algorithm *Justify* is the basis of a *conservative justifier*.

3.1 Efficiency Issues in Conservative Justification

Using the XSB tabled LP system we implemented *Justify* as a post-processing step following query evaluation. The advantage of using a tabled system for justification is that the answers in the tables can be directly used for computing the ξ's (lines 2 and 3). In particular if all the predicates are tabled then the truth value of all the literals are stored in the tables. Hence selecting a $\xi(A)$ amounts to a simple table lookup. In fact we can show:

Proposition 1 *For a logic program consisting of tabled predicates only, the running time of Justify is proportional to the time taken by initial query evaluation.*

Let us examine the behavior of *Justify* on a program containing both tabled and non-tabled predicates. In a tabled LP system there is no provision for storing the truth value of non-tabled literals. Consequently computing ξ's can become expensive since non-tabled predicates must be re-executed (a-la Prolog style) to ascertain their truth values. In fact, as is shown below, the time for justifying a single non-tabled literal can become quadratic its original evaluation time..

Example 2 *Consider the following non-tabled factorial logic program:*

```
fac(0, 1).
fac(N, S) :- N > 0, N1 is N - 1, fac(N1, S1), S is S1 * N.
```

Assume that `fac(N,S)` is evaluted for some fixed n. It is easy to see that evaluation time is $O(n)$. The call to *Justify*`(fac(n,n!))` will first compute ξ`(fac(n,n!))`. This set will include `fac(n-1,(n-1)!)`. Algorithm *Justify* takes $O(n-1)$ steps to compute ξ`(fac(n,n!))` since evaluting the truth value of `fac(n-1,(n-1)!)` requires that many steps. Next *Justify*`(fac(n-1,(n-1)!))` is invoked and the above process is repeated. It is easy to see that *Justify*`(n,n!)` will require $O(n^2)$ time.

One can however table all the predicates in a program. In such a case the truths of `fac(n,n!), fac(n-1,(n-1)!),..., q(0,1)` are all stored in an answer table upon completion of query evaluation. Justifiation will require $O(n)$ time since evaluating the truths of each of the *fac*'s can be done in $O(1)$ time.

But for time and space efficiency predicates are selectively tabled in practice [3]. The interesting question now is this: Can we design an efficient justifier for mixed programs without having to suffer the overheads of repeated re-execution of non-tabled predicates? Indeed our interest in this question was mainly motivated by our expereince with our XMC justifer for model checking [1]. On large model checking problems the XMC model checker took a few minutes to produce the results whereas the justifier failed to produce the justification even after sevaral hours! In the next section we will present an answer to this question.

4 Speculative Justifier

The idea behind a speculative justifer is as follows: Suppose we wish to justify the truth of p and further suppose there is a clause p :- $q_1, q_2, ..., q_n$ in the program. Further suppose we wish to build a true-justification for p. If $\{q_1, q_2, \ldots, q_n\} \in \xi(p)$ then one can build a justification for p by building true-justifications for each of the q_i's, $(1 \leq i \leq n)$. Without evaluating their truths *apriori* we speculate that $\{q_1, q_2, \ldots, q_n\} \in \xi(p)$ and attempt to build a true-justification for all of them. If $\{q_1, q_2, \ldots, q_n\} \in \xi(p)$ then all these justifications will succeed and result in a true-justification for p. If $\{q_1, q_2, \ldots, q_n\} \notin \xi(p)$ then there must exist at least one q_i for which the attempt at building a true-justification for it will fail. Hence this clause cannot provide any evidence as to why p is *true* and we proceed to find another candidate clause. Now suppose we wish to build a false-justification for p. We speculate again that there must exist at least one q_i that is *false*. So we attempt building a false-justification for each of the q_i's in sequence. If we fail to build a false-justification for any of the q_i's then we can conclude that a false-justification for p does not exist. On the other hand if we do succeed then we repeat this process on the next clause that unifies with p. Recall from definition of justification that to justify that p is *false* there must exist a false literal in each of these clauses.

The main advantage of speculative justifiers can be seen when justifying non-tabled predicates. Recall non-tabled literals are re-executed during justification. Speculative justifiers re-execute less often than their conservative counterparts. Consider $\{q_i$:- $q_{i-1}.|1 \leq i \leq n\}$, n is a constant and q_0 is a fact. To build a true-justification for q_n the speculative justifier will attempt to build a true-justification for q_{n-1} which in turn build a true-justification for q_{n-2}, and so on. All of these justifications succeed without ever having to repeat re-execution of any of the q_i's in q_n's proof.

4.1 Efficiency Issues

But speculative justifiers can suffer from inefficiencies. For example, the gains accrued by re-executing non-tabled literals less often can be easily offset by wasted justifications. We discuss these problems below:

- **The Problem of Wasteful Justifications:**
  Naive implementation of speculative justifiers can result in building wasteful justifications that are eventually discarded. For example, suppose we wish to build a true-justification for p using the clause pick $p :- q, r$. Suppose q is *true* and r is *false*. We will succeed in building a true-justificaton for q but fail to do so for r. So using this clause we will fail to build a true-justification for p. But the true-justification built for q is wasted.
- **The Problem of Rebuilding Justifications:**
  In the above example justification of q was discarded as being irrelevant for justifying p. Now suppose later on we encounter the literal q again during justification. If we do not save the justifcation of q then we will have rebuild its justification all over again.

We now propose solutions to these two main sources of inefficiency in a speculative justifier.

Lazy Justification To avoid wasteful justifications we justify tabled literals *lazily*. The idea is this: Let us suppose we select the clause $p :- q_1, q_2, ..., q_n$ for justifying p. Assume we wish to build a true-justification for p. Suppose the literal currently on hand, say q_i, is tabled. Then we do a simple table-lookup to verify that $\tau(q_i)$ is *true*. If this is the case we defer building its justification and move on to the next literal in the sequence. If q_i is non-tabled then we build true-justification for it. We proceed to build justifications for the tabled literals in the clause only after we succeed building true-justifications for all of its non-tabled literals. This idea carries over for false-justifications also.

Sharing Justfications The solution to re-building justifications is to save all of them after they are built the first time. We save the justifications of both tabled and non-tabled literals. But this can result in space ineffciencies especially if sharing is infrequent and irrelevant justifications outweigh relevant ones. A practical compromise between never re-building and always re-building is to share the justifications of tabled literals only. But note that justification of a tabled literal might involve other tabled literals. So we will have to avoid copying the entire justification. Instead we save a "skeleton" of the justification from which we can reproduce the complete justification. We call this skeleton *partial justification*. Intuitively the leaf nodes of a partial justification are either labelled `fail, fact, ancestor` or by a tabled literal. All of the interior nodes except the root are labelled by non-tabled literals. Formally:

Definition 4 (Partial Justification) *A partial justification for an atom A with respect to a program P and table T, denoted by $\mathcal{P}_{(P,T)}(A)$, is a directed acyclic graph $G = (V, E)$ with vertex labels chosen from $HB(P) \cup \{$`fact`, `fail`, `ancestor`$\}$ and the edges from $\{(B_1, B_2) | B_1 = A \vee B_1 \notin T\}$. The conditions for selecting the edges are the same as those used in defining justification (def. 3).*

We drop the parameter P and T and write the partial justification as $\mathcal{P}(A)$ whenever the program and the table are obvious from the context.

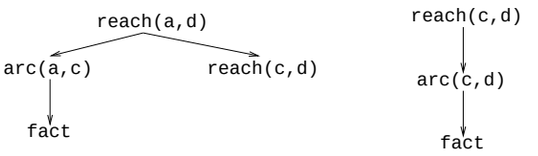


Fig. 4. Partial Justification of $reach(a, d)$ and $reach(c, d)$ in Figure 1

Figure 4 denotes the partial justifications of $reach(a, d)$ and $reach(c, d)$ for the example in Figure 1.

We can compose partial justifications together to yield a complete justification for a literal. Informally composition amounts to "stringing" together the partial justifications of tabled literals at the leaf nodes labelled by those literals. For example in Figure 4, by attaching the partial justification of $reach(c, d)$ to the leaf node labelled $reach(c, d)$ in the partial justification of $reach(a, d)$ yields its complete justification. However care must be exercised when composing partial justifications. In particular compositions that produce cycles in true-justifications must be discarded.

4.2 Algorithmic Aspects of Speculative Justification

The speculative justifier builds a justifcation by composing several partial justifications. The algorithm for partial justification is shown in Figure 5. It takes the following parameters as its input: (i) A which is the literal to be justified, (ii) A's truth value *Tval* and (iii) *Anc* which is a list of tabled literals that are ancestors of A in the justification. The algorithm builds a true(false)-justification if *Tval* is *true (false)*. It returns in J the partial justification of A and D those tabled calls which appears in the leaf nodes of J. We use `clause(A,B)` to pick a clause that unifies with A and `findall` for aggregation. T denotes the tabled literals and their answers.

Recall that to build the complete justification of A we need to know the partial justifications of all the tabled literals that the justification of A depends upon (e.g. *reach(a,d)* depends on *reach(c,d)* in Figure 4). Let $D_A = \{P | P$ is a tabled literal that appears as the label of a leaf node in $\mathcal{P}(A)$ $\}$. We refer it to as the *dependent set*. We will drop the subscript from the notation for the dependent set if the literal that it is associated with is clear from the context.

4.3 Properties of a Speculative Justifier

We will suppose that a speculative justifier is based on algorithm $partial\text{-}justify$ and that the complete justification for any literal is obtained by composing all the partial justifications of tabled literals it depends on. We state below some of its important properties.

Proposition 2 *On purely tabled logic programs, speculative justifier coincides with conservative justifier.*

The above is based on the observation that to justify `A` the speculative justifier generates a partial justification which includes its dependent set and fact nodes. They correspond to a `lce` for `A`.

Proposition 3 *On purely non-tabled logic programs, justification time required by a speculative justifier is proportional to query evaluation time.*

This proposition is based on the observation that when a program has no tabled predicates then the partial justification for A corresponds to complete justification and that evaluation proceeds in Prolog-style.

Theorem 4 *The time taken by a speculative justifier for justification is no more than the time taken by a conservative justifier*

We sketch only the main observation for establishing the above propoerty. Note that a conservative justifier computes a lce for A by re-executing non-tabled

```
algorithm Partial-Justify(A : atom, Tval : truth value, Anc : Ancestors)
    (* Local: J : Justification (V, E); D : Dependent Set *)
  set (J, D) := (({A}, {}), {})
  if ( Tval = true ) then (* build true-justification *)
    clause(A, B)
    if ( B = true ) then(* the selected clause is a fact *)
      set J := ({A, fact}, {(A, fact)})
    else
      for each G ∈ B then
        if ( G ∈ T ) then
          if ( τ(G) = true ) then
            if (G ∈ Anc) then
              fail
            else
              set E := E ∪ {(A, G)}
              set D := D ∪ {G} (* add G to the dependent set *)
          else (* τ(G) ≠ true *)
            fail
        else (* G is a non-tabled call *)
          set E := E ∪ {(A, G)}
          set (J, D) := (J, D) ∪ partial-justify(G, Tval, Anc)
  else (* build false justification *)
    findall(B, clause(A, B), BL)
    if (BL = {}) then (* no clause unifies with A *)
      set J := ({A, fail}, {< A, fail >})
    else
      for each B ∈ BL do
        let G ∈ B (* G is choosen from B sequentially *)
        if ( G ∈ T ) then
          if ( τ(G) = false ) then
            if (G ∈ Anc) then
              set E := E ∪ {(A, ancestor)}
            else
              set E := E ∪ {(A, G)}
              set D := D ∪ {G} (* add G to the dependent set *)
          else (* τ(G) ≠ Tval *)
            fail
        else (* G is a non-tabled call *)
          set E := E ∪ {(A, G)}
          set (J, D) := (J, D) ∪ partial-justify(G, Tval, Anc)
  return (J, D)
```

Fig. 5. Speculative Justification

literals and consulting the answer table for tabled literals. This coresponds to computing the partial justification of A by a speculative justifier. Besides the search paths for computing lce's in a conservative justifier and partial justifications in a speculative justifiier also correspond.

While the above theorem only says that the time taken is proportional, speculative justifiers can do better. Consider the non-tabled factorail program in Example 2. By avoiding repeated re-executions the speculative justifier will build a true justification for `(fac(n,n!))`in $O(n)$ steps whereas it took $O(n^2)$ steps for the conservative justifier.

5 Experimental Results

In [8] we reported on the performance of a conservative justifier based on *Justify* (in Section 3) and implemented using the XSB tabled LP system. It was developed for our XMC model checking environment Model checking in XMC corresponds to evaluating a top-level query that denotes the temporal property of interest. The query succeeds whenever the system being verifed satisfies the property. To succinctly explain the success or failure of the query we use the XMC justifier. We have now implmented the speculative justifier based on *Partial-Justify* (see Section 4). This impelmentation also uses the XSB system. Both the impelmentations only share the justifications of tabled literals.

We compare the performance of both the justfiers on the model checking application using our XMC system. Figure 6(a) compares their running times while Figure 6(b) shows their space usage. The model checking examples used in these experiments ((`i-Protocol`, `ABP`,`Leader`,`Sieve`) were taken from the XMC collection. `i-Protocol` is a sliding window protocol in the GNU UUCP stack, `ABP` is the alternating protocol, `Leader` and `Sieve` are taken from the SPIN [2] example suite.

Observe that the running times of the speculative justifier is significntly better, sometimes by several orders of magnitude. Because of its significant speedups the speculative justfier is able to scale up to large problem sizes. For example, on `i-Protocol`(window size 1, no livelock) and `Leader`(size 6), which are instances of large model checking examples, the speculative justifier took a few minutes whereas the conservative justifier did not finish even after several hours!

Also observe that the space usage of the speculative justifier appears comparable to its conservative counterpart.

Figure 7(a) compares justification time of the speculative justifier to query evauation time while Figure 7(b) compares their space usage. Observe that the running times and space usage of the speculative justifier seems to suggest that they are both nearly proportional to those of query evaluation.

6 Discussion

We introduced the concept of a sepeculative justifier, presented an algoritihm for it and provdided experimental evidence of its efficiency and scalabity. The

justification algorithm in this paper assumed definite clause logic programs. In [8] we show how to extend the justification algorithm in a conservative justifier to normal logic programs evaluated under well-founded semantics. The same extensions carry over to the justifcation algorithms used in the speculative justifier.

In this paper our primary focus was on improving the running time of justification so as to scale to large problem sizes that we encountered in our model checking application. The justifier described in this paper can be used with any other tabled LP system. As far as space usage is concerned it is possible to improve it further. One possibility is to control the size of partial justification. Recall that partial justification can include justification of non-tabled literals. There are several reasons for controlling the justification of non-tabled literals

Benchmark	**Leader (ae_leader)**					**Leader (one_leader)**				
Size	**2**	**3**	**4**	**5**	**6**	**2**	**3**	**4**	**5**	**6**
Conservative	0.18	1.51	10.86	130.3	n/a	0.19	1.41	11.01	136.6	2252.7
Speculative	0.05	0.24	1.21	6.80	35.2	0.06	0.22	1.17	6.04	33.2

Benchmarks	**Sieve (ae_finish)**									
Size	**(3,4)**	**(3,5)**	**(4,5)**	**(4,6)**	**(5,6)**	**(5,7)**	**(6,7)**	**(6,8)**	**(6,9)**	**(6,10)**
Conservative	1.12	1.24	3.65	4.60	11.92	15.71	46.83	51.5	62.8	78.29
Speculative	0.16	0.18	0.42	0.52	1.17	1.45	3.38	3.69	4.13	4.80

Benchmarks	**Meta-lock (mutex)**						**ABP**	**Iproto (bug)**
Size	**(1,2)**	**(1,3)**	**(1,4)**	**(2,1)**	**(3,1)**	**(2,2)**		fix(1)
Conservative	2.11	21.95	310.1	4.77	239.0	488.4	1.81	n/a
Speculative	0.18	0.97	4.98	0.32	4.09	6.16	0.20	193.2

(a) Running Time (in Seconds.)

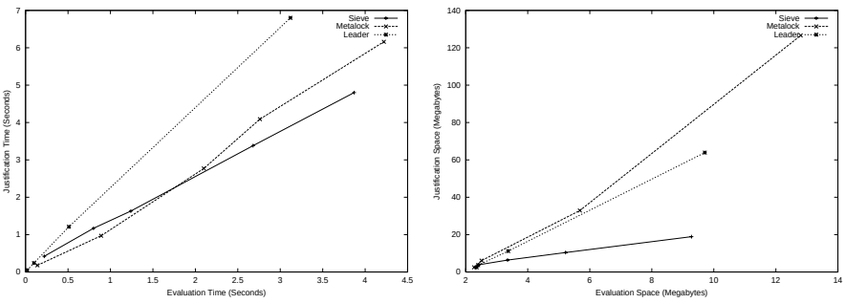
Benchmark	**Leader (ae_leader)**				**Leader (one_leader)**			
Size	**2**	**3**	**4**	**5**	**2**	**3**	**4**	**5**
Conservative	2.35	4.96	17.6	81.1	2.40	2.62	8.25	43.7
Speculative	2.48	3.68	10.4	63.7	2.48	3.68	10.5	63.9

Benchmarks	**Sieve (ae_finish)**									
Size	**(3,4)**	**(3,5)**	**(4,5)**	**(4,6)**	**(5,6)**	**(5,7)**	**(6,7)**	**(6,8)**	**(6, 9)**	**(6,10)**
Conservative	5.03	4.86	9.26	9.16	17.6	33.6	66.4	66.7	67.1	67.6
Speculative	2.63	2.67	3.87	4.04	6.57	10.1	18.9	19.3	19.9	20.8

Benchmarks	**Meta-lock (mutex)**						**ABP**
Size	**(1,2)**	**(1,3)**	**(1,4)**	**(2,1)**	**(3,1)**	**(2,2)**	
Conservative	2.45	6.50	21.3	2.32	14.3	25.4	2.57
Speculative	2.53	6.11	32.9	3.60	19.6	34.3	2.54

(b) Space Usage (in MBs)

Fig. 6. Time and Space Comparison between Conservative and Speculative



(a) Time Usage (b) Space Usage

Fig. 7. Time and Space Comparison between Evaluation and Justification

and thereby control the size of partial justificaion. Firstly, justification of non-tabled literals can be arbitrarily big. Secondly, users may not be interested in justifying non-tabled calls. Thirdly users may prefer to use the familiar 4-port debugger for non-tabled literals over a justifier. Users can specify the non-tabled literals that they are not interested in justifying. The justifier will simply evaluate away such literals without explicitly building a justification for them. Improving space efficiency using such techniques is a topic that deserves further exploration.

References

1. C.R Ramakrishnan, I.V. Ramakrishnan, S.A. Smolka, Y. Dong, X. Du, A. Roychoudhury, and V.N. Venkatakrishnan. Xmc: A logic programming based verification toolset. In *Computer Aided Verification*, 2000.
2. G.J. Holzmann. The model checker SPIN. In *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.
3. S. Dawson, C.R. Ramkrishnan, I.V. Ramakrishnan and T. Swift. Optimizing Clause Resolution: Beyond Unification Factoring. In *International Logic Programming Symposium*, 1995.
4. J.W. Lloyd. Declarative error diagnosis. In *New Generation Computing*, 5(2):133–154, 1987.
5. S. Mallet and M. Ducasse. Generating deductive database explanations. *Proceedings of ICLP*, 154–168, 1999
6. L. Naish, P.W. Dart, and J. Zobel. The NU-prolog debugging environment. In *ICLP*, pages 521–536, 1989.
7. G. Puebla, F. Bueno, and M. Hermenegildo. A framework for assertion-based debugging in constraint logic programming. In *Pre-proceedings of LOPSTR*, 1999.
8. Abhik Roychoudhury, C. R. Ramakrishnan, and I. V. Ramakrishnan. Justifying proofs using memo tables. In *PPDP 2000*, pages 178–189.
9. Y.S. Ramakrishna, C.R. Ramakrishnan, I.V. Ramakrishnan, S.A. Smolka, T. Swift, and D.S. Warren. Efficient model checking using tabled resolution. In *CAV, LNCS 1254*, 1997.
10. Ehud Y. Shapiro. Algorithmic program diagnosis. In *POPL*, 1982.

11. XSB. The XSB logic programming system v2.3, 2001. Available by anonymous ftp from `www.cs.sunysb.edu/~sbprolog`.
12. Prasad Rao, C.R. Ramakrishnan, and I.V. Ramakrishnan. A Thread in Time Saves Tabling Time. In *IICSLP*, 1996.

Local and Symbolic Bisimulation Using Tabled Constraint Logic Programming*

Samik Basu[1], Madhavan Mukund[2], C.R. Ramakrishnan[1], I.V. Ramakrishnan[1], and Rakesh Verma[3]

[1] Department of Computer Science, State University of New York at Stony Brook
Stony Brook, New York, U.S.A.
{bsamik,cram,ram}@cs.sunysb.edu
[2] Chennai Mathematical Institute, Chennai, India.
madhavan@cmi.ac.in
[3] Department of Computer Science, University of Houston, Texas.
rmverma@cs.uh.edu

Abstract. *Bisimulation* is a fundamental notion that characterizes behavioral equivalence of concurrent systems. In this paper, we study the problem of encoding efficient bisimulation checkers for finite- as well as infinite-state systems as logic programs. We begin with a straightforward and short (less than 10 lines) encoding of finite-state bisimulation checker as a tabled logic program. In a goal-directed system like XSB, this encoding yields a *local* bisimulation checker: one where state space exploration is done only until a dissimilarity is revealed. More importantly, the logic programming formulation of local bisimulation can be extended to do *symbolic bisimulation* for checking the equivalence of infinite-state concurrent systems represented by *symbolic transition systems*. We show how the two variants of symbolic bisimulation (late and early equivalences) can be formulated as a tabled constraint logic program in a way that precisely brings out their differences. Finally, we show that our symbolic bisimulation checker actually outperforms non-symbolic checkers even for relatively small finite-state systems.

1 Introduction

A tabled logic programming system offers an attractive platform for encoding computational problems in the specification and verification of systems. The XMC system [12] casts the problem of *model checking*— verifying whether a given concurrent system is in the model of a temporal logic formula— as query evaluation over an "equivalent" logic program [11]. This formulation is based on the connection between models of logic programs and models of temporal logics. In this paper, we consider the related problem of *bisimulation checking* which checks for equivalence of two system descriptions.

* This research was partially supported by NSF Grants EIA-9805735, EIA-9705998, CCR-9876242, CCR9732186 and IIS-0072927.

P. Codognet (Ed.): ICLP 2001, LNCS 2237, pp. 166–180, 2001.

Bisimulation checking is a problem of fundamental importance in verification. Many verification systems such as the Concurrency Workbench of the New Century (CWB-NC) [3] and CADP [1] incorporate bisimulation checkers in their tool sets. Informally, a pair of automata M, M' are said to be bisimilar if for every transition in M there exists a corresponding transition in M' and vice versa. There has been a lot of research on efficient algorithms for bisimulation checking. But the focus of this vast body of work has been on finite-state systems, i.e., one assumes that M and M' are both finite state. Hennessy and Lin were the first to consider the problem of bisimilarity checking of infinite-state systems in the setting of value-passing languages [4]. This initial work has been recently expanded [7,6]. Nevertheless research on this problem remains in a state of infancy.

In this paper, we explore the use of logic programming for the above problem. We begin with a direct formulation of strong- and weak-bisimulation checking for finite-state systems (see Section 2). We show that, using query evaluation with a tabled logic programming system, this encoding yields a *local* bisimulation checker: one where the state space of the concurrent systems is explored only until the first evidence of non-bisimilarity is found. Note that when the systems are indeed bisimilar, the local checker explores the entire (reachable) state space. Even in this case, our bisimulation checker encoded in XSB logic programming system [13] shows performance comparable to the global bisimulation checker in CWB-NC. For systems that are non-bisimilar, the local checker outperforms the global checker by several orders of magnitude.

We introduce *symbolic transition systems* (STSs) which can finitely represent infinite-state systems (see Section 3). STSs are more general than Symbolic Transition Graphs (STGs) and STGs with Assignments (STGAs) used in [4] and [7] respectively. We formulate symbolic bisimulation algorithms for checking two kinds of equivalences widely studied in the literature— *late*- and *early*-equivalences— as tabled constraint logic programs (see Section 4). Similar to the finite-state case, our formulation is a direct encoding of the definition of the bisimulation relations themselves. We describe how the programs can be evaluated using a constraint meta-interpreter implemented in XSB. Our experimental results show that symbolic bisimulation is practical for realistic systems. Surprisingly, our results show how even for relatively small finite-state systems, it may be better to perform symbolic bisimulation on its infinite-state counterparts. We conclude in Section 5 with a short discussion of the implications of this work.

2 Bisimilarity of Finite-State Systems

Labeled transition systems (LTSs) are widely used to capture the operational behavior of concurrent systems. An LTS is denoted by $L = (S, Act, \longrightarrow)$, where S is a finite set of states, Act is a finite set of *actions* (transition labels), and $\longrightarrow \subseteq S \times Act \times S$ is a transition relation. Transition from state s to t on an action a is represented by $s \xrightarrow{a} t$. Example LTSs are given in Figure 1.

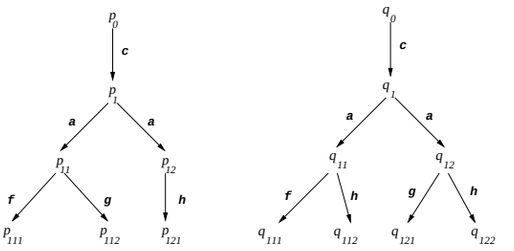


Fig. 1. Example non-symbolic LTS

An LTS $L = (S, Act, \longrightarrow)$ is encoded as a set of facts in a logic program P such that whenever $s \xrightarrow{a} t$, then `trans(`s`, `a`, `t`)` is in P. Note that since $s, t \in S$ as well as $a \in Act$ are from a finite set, they can be represented in a logic program by ground terms. Actions on transitions in LTS are of two types: actions that may be effected by external entity, environment, are called *observable actions* and actions that are the result of synchronization of subsystems are called *internal actions* or τ *actions*. Based on this notion of observability there are two variations of bisimilarity, *strong* and *weak*, described below.

2.1 Strong Bisimulation

Strong bisimulation does not differentiate between internal and observable actions.

Definition 1 (Bisimilarity Relation) *Given an LTS $L = (S, Act, \longrightarrow)$, $\mathcal{R}$ is a bisimilarity relation over L if*

$$\forall s_1, s_2 \in S.\ s_1 \mathcal{R} s_2 \Rightarrow (\forall (s_1 \xrightarrow{a} t_1).\ (\exists (s_2 \xrightarrow{a} t_2).\ t_1 \mathcal{R} t_2) \wedge s_2 \mathcal{R} s_1)$$

Two states in a system are equivalent with respect to bisimulation if they are related by *largest* bisimilarity relation $\mathcal{R}$. Two LTSs can be compared for bisimilarity by computing bisimulation of their disjoint union. For instance, consider the LTSs in Figure 1. States p_{11} and q_{11} are not bisimilar as there exists a transition from p_{11} with action g for which there is no matching transition from q_{11}. As such, states p_1 and q_1 are not bisimilar and also the states p_0 and q_0 are not bisimilar.

Encoding Strong Bisimulation. Using the dual of Definition 1, we can say that two states in a system are not equivalent with respect to bisimulation if they are related by the least relation $\overline{\mathcal{R}}$ defined as follows:

$$\forall s_1, s_2 \in S.\ s_1 \overline{\mathcal{R}} s_2 \Leftarrow (\exists (s_1 \xrightarrow{a} t_1).\ (\forall (s_2 \xrightarrow{a} t_2)\ \Rightarrow t_1 \overline{\mathcal{R}} t_2) \vee s_2 \overline{\mathcal{R}} s_1) \qquad (1)$$

Note that $\overline{\mathcal{R}}$ can be encoded as a logic program by exploiting the least model computation of logic program as follows:

```
bisim(S1, S2)  :- tnot(nbisim(S1, S2)).
nbisim(S1, S2) :- trans(S1, A, T1),
                  no_matching_trans(S2, A, T1).
nbisim(S1, S2) :- nbisim(S2, S1).
```

In the above encoding, `nbisim/2` stands for $\overline{\mathcal{R}}$ defined by Equation 1. The goal `no_matching_trans(S2, A, T1)` stands for $\forall(s_2 \xrightarrow{a} t_2) \Rightarrow t_1 \overline{\mathcal{R}} t_2$ and is in turn defined as:

```
no_matching_trans(S2, A, T1) :-
  forall(trans(S2, A, T2), nbisim(T1, T2)).
        % T1 is not bisimilar to any T2
```

Note that since the terms `S2`, `A` are ground and `T2` is free, `forall/2` can be encoded without considering any free variables as follows:

```
forall(P, Q) :- findall(Q, P, L), all(L).
all([]).
all([Q|Qs]) :- Q, all(Qs).
```

2.2 Local Bisimulation

Evaluating `bisim(`s_1`,` s_2`)` using tabled resolution, we can prove or disprove bisimilarity of states s_1 and s_2. Note that goal directed computation with tabling makes the bisimulation checker "local": state space exploration is done only until the proof for bisimilarity or non-bisimilarity is obtained. However, if the given states are actually bisimilar, then we explore all the states reachable from s_1 and s_2. Another important aspect of our encoding is that it can be directly extended to symbolic bisimulation checking for infinite-state systems.

Given any two states in an LTS $(S, Act, \longrightarrow)$, the worst case time complexity of our bisimulation checker is $O(|S| \times |\longrightarrow|)$ assuming unit-time table lookups. The quadratic factor in our encoding comes from checking for bisimulation between (potentially) every pair of states. Table lookups may add $|S|^2$ factor to the complexity if tables are organized as a list, or $|\log(S)|$ factor if binary tree data structures are used. It should be noted that there are faster bisimulation checking algorithms: the Kanellakis-Smolka algorithm [5] runs in $O(|S| \times |\longrightarrow|)$; Paige and Tarjan's algorithm [9], implemented in CWB-NC, runs in $O(|\longrightarrow| \times \log|S|)$. These algorithms, unlike our implementation, compute equivalence classes of bisimilar states bottom up and are thus global.

2.3 Weak Bisimulation

In practical settings two systems are considered to be equivalent when they are identical with respect to the observable actions. *Weak bisimulation* or *observational equivalence* formalizes this notion. It is defined on the basis of weak transition relation.

Definition 2 (Weak Transition Relation) *Given a LTS $L = (S, Act, \longrightarrow)$, weak transition relation, $\longrightarrow_w \subseteq S \times Act \times S$, is the smallest relation such that*

1. $a \neq \tau$ *and* $s_1(\xrightarrow{\tau})^* s_2 \xrightarrow{a} s_3(\xrightarrow{\tau})^* t_1 \Rightarrow s_1 \xrightarrow{a}_w t_1$
2. $s_1(\xrightarrow{\tau})^* t_1 \Rightarrow s_1 \xrightarrow{\tau}_w t_1$

Note that (2) implies that for every state s, $s \xrightarrow{\tau}_w s$.

Definition 3 (Weak Bisimilarity) *Given an LTS* $L = (S, Act, \longrightarrow)$, $\mathcal{R}_W$ *is a weak bisimilarity relation over* L, *if*

$$\forall s_1, s_2 \in S.\ s_1 \mathcal{R}_W s_2 \Rightarrow (\forall (s_1 \xrightarrow{a} t_1).\ (\exists (s_2 \xrightarrow{a}_w t_2).\ t_1 \mathcal{R}_W t_2) \wedge s_2 \mathcal{R}_W s_1)$$

Encoding Weak Bisimulation. We begin with the encoding the weak transition relation. Note that $(\xrightarrow{\tau})^*$ is the transitive closure of $\xrightarrow{\tau}$ and can be encoded as follows:

```
taustar(S1, S1).
taustar(S1, S2) :-  taustar(S1, T), trans(T, tau, S2).
```

Using `taustar/2` weak transition relation can be directly encoded as follows:

```
weak_trans(S1, tau, T1) :- taustar(S1, T1).
weak_trans(S1, A, T1)   :- taustar(S1, S2), trans(S2, A, S3),
                           A \= tau, taustar(S3, T1).
```

Note that the only difference between Definitions 3 and 1 lies in the selection of matching transition, *i.e.*, $\exists (s_2 \xrightarrow{a}_w t_2).t_1 \mathcal{R}_W t_2$. Definition 1 uses strong transition $\xrightarrow{a}$, whereas Definition 3 uses weak transition $\xrightarrow{a}_w$. Thus the previous encoding of strong bisimilarity can be changed as follows to compute weak bisimilarity:

```
weak_bisim(S1, S2)  :- tnot(nweak_bisim(S1, S2)).
nweak_bisim(S1, S2) :- trans(S1, A, T1), no_matching_trans(S2, A, T1).
nweak_bisim(S1, S2) :- nweak_bisim(S2, S1).
no_matching_trans(S2, A, T1) :-
                  forall(weak_trans(S2, A, T2), nweak_bisim(T1, T2)).
```

2.4 Experimental Results

Below, we compare the performance of our local bisimulation checker with the one based on partition refinement algorithm [9] implemented in CWB-NC. Example systems selected are families of $stack(b, d)$, $queue(b, d)$ and $multi\_link\_buffer(b, d)$ where d denotes the data domain size and b denotes buffer lengths in the first two systems and number of buffers in the third system. All measurements were made on a Sun 4U sparc Ultra Enterprise with 2G memory running Solaris 5.2.6, using XSB v2.3 and CWB-NC v1.11.

A $stack(b, d)$ is defined with a buffer of fixed size b, where insert and delete actions respectively add and delete data to and from the top of the buffer. Whereas, in case of $queue(b, d)$ insert action adds data to the bottom of the buffer and delete action removes data from the top. A $multi\_link\_buffer(b, d)$ is a chain of b buffers (each of length 1), where each buffer can insert and delete data. However, visible actions include only insertion to the first buffer and deletion

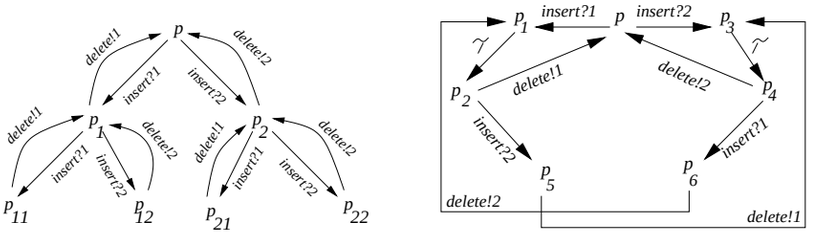


Fig. 2. a. $stack(2,2)$ b.$multi\_link\_buffer(2,2)$

from the last buffer. All other insert and delete actions are synchronized actions; delete from the ith buffer synchronized with the insert action to the $i+1$th buffer ($1 \leq i < b$). These actions are represented as τ transitions. Domain of each element in the buffer ranges over d distinct values.

Figure 2(a) shows the transition system of a $stack(b,d)$ with $b=2$ and $d=2$, where $insert?1$ and $delete!1$ represents input and output actions that insert and delete data value 1 to and from the stack. Similarly, Figure 2(b) shows the transition system of a $multi\_link\_buffer(b,d)$ with $b=2$ and $d=2$.

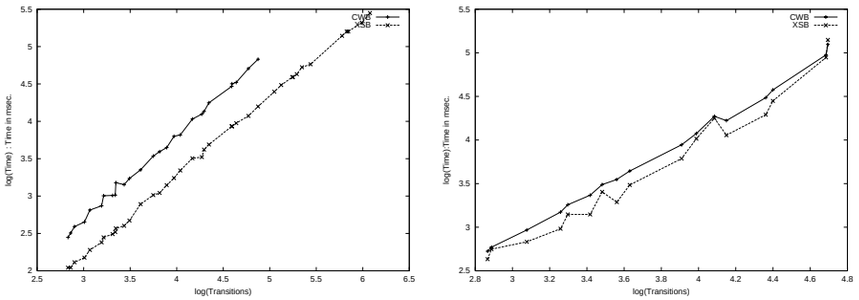


Fig. 3. a. Bisimulation of two identical $stack(b,d)$s in XSB and CWB-NC b.Weak Bisimulation of $queue(b,d)$s and $multi\_link\_buffer(b,d)$ in XSB and CWB-NC

Figure 3(a) shows the time taken (on logarithmic scale) to check for strong bisimilarity of two identical $stack(b,d)$s for different combinations of b and d. The number of transitions in the system is $O(d^b)$. Since the systems are bisimilar, both our encoding and the CWB-NC explore the entire state space. As shown in Figure 3(a), XSB implementation is roughly 3 times faster than CWB-NC implementation and hence comparable. Similar results are obtained when we perform weak bisimilarity checking of $queue(b,d)$ and $multi\_link\_buffer(b,d)$ (Figure 3(b)).

We now compare the time taken to check strong bisimilarity of $stack(b,d)$ and $queue(b,d)$ for different combinations of b and d. These systems are not bisimilar

when $b \geq 2$ and $d \geq 2$. In this case local bisimulation checker implemented in XSB outperforms global checking algorithm implemented in CWB-NC. The XSB implementation can check for bisimilarity of $stack(b, d)$ and $queue(b, d)$ for $b = 5000$, $d \geq 2$ in 41.70 secs. and with CWB-NC the largest system we can check is of $b = 7$, $d = 4$ which takes up about 40.50 secs. We also performed weak bisimulation checking between $stack(b, d)$ and $multi\_link\_buffer(b, d)$. In this case the time taken for XSB implementation to detect non-bisimilarity between two systems with $b = 100$ and $d \geq 2$ is 20.91 secs. while the largest system CWB-NC can check for non-bisimilarity is of $b = 7$ and $d = 3$ in about 79.45 secs. It is worth mentioning here that local bisimulation checking based on XSB in both the cases is independent of d. An inspection of query evaluation in XSB reveals that only two elements from the data domain d are sufficient to prove non-bisimilarity of the given systems. However the same is not the case for CWB-NC as it uses global bisimilarity checking algorithm.

3 Infinite-State Systems

We introduce the notion of *Symbolic Transition Systems* (STSs) as a way to represent large or infinite-state systems. An STS can be viewed as an LTS augmented with state variables, guards on transitions, and nonground terms as action labels. Infinite-state systems can be represented finitely by STSs.

3.1 Symbolic Transition Systems

We assume the standard notion of terms, substitutions and unifiers. We use $\mathcal{V}$ to denote an enumerable set of variables, $\mathcal{F}$ to denote a set of function symbols, $\mathcal{P}$ to denote a set of predicate symbols, and $\mathcal{B}$ to denote $\{true, false\}$. Function and predicate symbols have fixed arity; function symbols of arity 0 are called constants. *Expressions*, denoted by $\mathcal{E}$ are terms over $\mathcal{F} \cup \mathcal{V}$ and *guards*, denoted by γ, are terms over $\mathcal{P} \cup \mathcal{F} \cup \mathcal{V}$ where predicate symbols appear at (and only at) the root. The set of variables in a term t is denoted by $vars(t)$. Substitutions, denoted by θ and σ (possibly primed or subscripted), are mappings from $\mathcal{V}$ to $\mathcal{E}$. A substitution that maps value v to variable x is written as $[v/x]$. A term t under substitution σ is denoted by $t\sigma$; the composition of two substitutions σ_1, σ_2 is denoted simply by $\sigma_1\sigma_2$.

A guard γ of arity n is interpreted as a mapping from $\mathcal{E}^n$ to $\mathcal{B}$. Alternatively, γ can be viewed as a set of substitutions such that $\sigma \in \gamma$ iff $\gamma\sigma = true$. An *action* is a term in one of the following forms:

- *Input Action:* Represented as $c?x$, where c is a constant and x is a variable.
- *Output Action:* Represented as $c!e$, where c is a constant and e is an expression.
- *Internal Action:* Represented by τ, a constant.

Output actions without the expression parameter, as in $c!$, are also known signals, and are simply represented as c.

Definition 4 (Symbolic Transition System) *A symbolic transition system is a finite labeled directed graph $(S, \longrightarrow)$, where S is a set of terms, called* locations*, which form vertices of the graph, and $\longrightarrow$ is the edge relation where each edge $s \xrightarrow{\gamma,\alpha,\rho} t$ is labeled with:*

- *an action α such that*
  - *$vars(\alpha) \subseteq vars(s)$ if α is not an input action.*
  - *$vars(\alpha) \cap vars(s) = \{\}$ if α is an input action.*
- *a guard γ such that $vars(\gamma) \subseteq vars(s)$, and*
- *a transfer relation ρ that relates $vars(s)$ to $vars(t)$.*

If a guard is *true* it is omitted. The transfer relation is used to model updates to variables. The transfer relation is omitted whenever it is the identity mapping over the source and target variables. The edge relation of two STSs representing a stack and a queue that stores arbitrary data values, with maximum buffer length of 2, are shown in Figure 4 (a) and (b) respectively.

$$
\begin{aligned}
s_0 &\xrightarrow{insert?x} s_1(x) \\
s_1(x) &\xrightarrow{delete!x} s_0 \\
s_1(x) &\xrightarrow{insert?y} s_2(x,y) \\
s_2(x,y) &\xrightarrow{delete!y} s_1(x)
\end{aligned}
\qquad\qquad
\begin{aligned}
q_0 &\xrightarrow{insert?x} q_1(x) \\
q_1(x) &\xrightarrow{delete!x} q_0 \\
q_1(x) &\xrightarrow{insert?y} q_2(x,y) \\
q_2(x,y) &\xrightarrow{delete!x} q_1(y)
\end{aligned}
$$

Fig. 4. Example STS representing 2-length stack (a) and 2-length queue (b) over arbitrary data domain

Note that the definition of STSs is general enough to capture Symbolic Transition Graphs (STGs) [4] and STGs with Assignments (STGAs) [7]. For instance, STGs are STSs where each edge $s \xrightarrow{\gamma,\alpha,\rho} t$ is such that $vars(t) \subseteq (vars(s) \cup vars(\alpha))$.

3.2 Semantics of STS

Semantics of an STS $\mathcal{S}$ is given in terms of a transition relation, denoted by $\mathcal{T}(\mathcal{S})$, which is generated by interpreting $\mathcal{S}$ with respect to substitutions. Given an STS $\mathcal{S}$, each state in $\mathcal{T}(\mathcal{S})$ is a location s paired with a substitution σ on $vars(s)$. There are different variants of semantics depending on how variables are interpreted. In the following, we describe *late* and *early* semantics which are the most widely studied to date.

Late semantics is a natural interpretation of the symbolic transition systems, by "reading off" transitions from a state $s\sigma$ by applying the substitution on all components of edges of the form $s \xrightarrow{\gamma,\alpha,\rho} t$ from location s. This is captured formally by the following definition.

Definition 5 (Late Transition Relation) *Let σ be a substitution such that $s \xrightarrow{\gamma,\alpha,\rho} t \in \mathcal{S}$, $\sigma \Rightarrow \gamma$, and σ satisfies ρ. Then $\mathcal{T}(\mathcal{S})$ contains $s\sigma \xrightarrow{\alpha\sigma}_l t\sigma$.*

One interesting aspect of late semantics is that we only capture substitutions on variables in the target state of a transition if they are related by ρ to those in the start state. For instance, consider an input transition of the form $s \xrightarrow{c?x} t$. From definition of STS, $x \notin vars(s)$. If t contains x, then x does not immediately pick up a value due to this transition. The variable x is left to be bound by a guard or transfer relation in a subsequent state. *Early* semantics interprets the new variables introduced on input actions by immediately assigning values to them.

Definition 6 (Early Transition Relation) *Let σ be a substitution such that $s \xrightarrow{\gamma,\alpha,\rho} t \in \mathcal{S}$, $\sigma \Rightarrow \gamma$, and σ satisfies ρ. Then $\mathcal{T}(\mathcal{S})$ contains*

$$s\sigma \xrightarrow{\alpha\sigma}_e t\sigma \qquad \text{if } \alpha \text{ is not an input action and further for all possible ground term } v$$

$$s\sigma \xrightarrow{c?v}_e t\sigma[v/x] \text{ if } \alpha = c?x, \text{ and } v \text{ is a ground term}$$

The two semantics naturally yield two variants of the bisimulation relation, as described in Section 4. Below, we describe how an STS can be encoded as a logic program so that the late semantics can be computed directly by resolution.

Encoding STS as a Constraint Logic Program: The edge relation of an STS $\mathcal{S}$ can be encoded as a constraint logic program P such that for each $s \xrightarrow{\gamma,\alpha,\rho} t \in \mathcal{S}$ `sts_edge(`s`, `α`, `γ`, `ρ`, `t`)` is a fact in P. We can encode each guard γ as a predicate in P so that whether $\sigma \Rightarrow \gamma$ can be checked using the query $\gamma\sigma$. We can also encode the transfer relation ρ as a predicate in P.

The late transition relation (Definition 5) can be computed from this set of facts using the following rule:

```
late_trans(S, A, T) :-  sts_edge(S, A, Gamma, Rho, T),
                        Gamma,   % The guard is satisfied
                        Rho.     % and so is the transfer relation
```

Early transition relation cannot be so directly encoded due to the universal quantifier over values in its definition (see Definition 6).

4 Symbolic Bisimulation

Late bisimulation and early bisimulation, which we describe in detail below, differ in the way input actions are treated. Consider checking the bisimilarity of locations p_1 and q_1 in the STSs given in Figure 5. Clearly, locations $p_{111}, p_{112}, p_{121}$ are all bisimilar to locations $q_{111}, q_{112}, q_{121}, q_{122}$ (all are deadlocked). Furthermore, location p_{11} is bisimilar to q_{11} when $x = 0$, and is bisimilar to q_{12} if $x \neq 0$; location p_{12} is bisimilar to q_{11} when $x \neq 0$; and is bisimilar to q_{12} when $x = 0$. However, are p_1 and q_1 bisimilar?

When q_1 makes a transition, say $q_1 \xrightarrow{c?x} q_{11}$, what is the matching transition from p_1? According to the bisimilarity sets we have computed so far, the

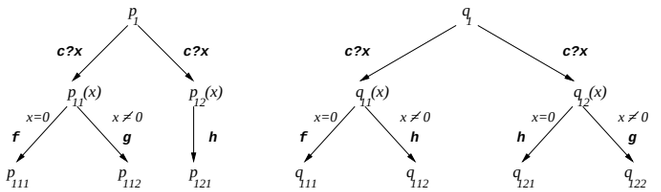


Fig. 5. Example symbolic transition systems

matching transition is the one to p_{11} when $x = 0$ and the one to p_{12} when $x \neq 0$. These two transitions together cover the transition from q_1 to q_{11}. However, note that the action on this transition is an input: $c?x$. Do we know enough about the value of x to make the choice between p_{11} and p_{12}? According to early semantics, the value of x is known when a transition is taken. However, according to late semantics, the value of x is determined only by later guards, and hence unknown when the transition is taken. Hence, p_1 and q_1 are *early-bisimilar* but not *late-bisimilar*.

Before a formal presentation of the bisimulation relations over STSs, we motivate their definitions by starting from the basic bisimulation relation for the finite-state case. In Definition 1, we had

$$\forall s_1, s_2 \in S \; s_1 \mathcal{R} s_2 \Rightarrow (\forall s_1 \xrightarrow{a} t_1 \; (\boxed{\exists s_2 \xrightarrow{a} t_2} \; t_1 \mathcal{R} t_2) \wedge s_2 \mathcal{R} s_1)$$

The question we have now is, having picked a transition $s_1 \xrightarrow{a} t_1$, how do we pick the matching $s_2 \xrightarrow{a} t_2$; and since in the symbolic case the action labels may bind variables, under what substitution. In the late bisimulation case, recall that the variable in an input label is bound only afterward, and hence the matching transition $s_2 \xrightarrow{a} t_2$ should be such that t_1 and t_2 are bisimilar under all substitutions to the input variable. In summary, the matching transition must be picked before considering substitutions. This intuition is captured by the following formal definition of late bisimulation.

Definition 7 (Late Bisimulation) *Given an STS $(S, \longrightarrow)$, the late bisimulation relation with respect to substitution θ, denoted by $\mathcal{R}_l^\theta$, is a subset of $S \times S$ such that*

$$s_1 \mathcal{R}_l^\theta s_2 \Rightarrow (\forall s_1\theta \xrightarrow{\alpha_1}_l t_1\theta \; \boxed{\exists s_2\theta \xrightarrow{\alpha_2}_l t_2\theta} \; \boxed{\forall \sigma} (\alpha_1[\theta\sigma] = \alpha_2[\theta\sigma]) \wedge t_1 \mathcal{R}_l^{\theta\sigma} t_2) \wedge s_2 \mathcal{R}_l^\theta s_1$$

In the early bisimulation case, recall that the variable in an input action is bound at the transition itself, there is no choice to make in terms of substitutions. This is captured by the following formal definition of early bisimulation.

Definition 8 (Early Bisimulation (using $\longrightarrow_e$)) *Given an STS $(S, \longrightarrow)$, the early bisimulation relation with respect to substitution θ, denoted by $\mathcal{R}_e^\theta$, is a subset of $S \times S$ such that*

$$s_1 \mathcal{R}_e^\theta s_2 \Rightarrow (\forall s_1\theta \xrightarrow{\alpha_1}_e t_1\theta \; \boxed{\exists s_2\theta \xrightarrow{\boxed{\alpha_1}}_e t_2\theta} \; (t_1 \mathcal{R}_e^\theta t_2)) \wedge \; s_2 \mathcal{R}_e^\theta s_1$$

The above definition relies on the definition of the early transition relation. It turns out, however, that we can define early bisimulation completely in terms of the late transition relation [10], as follows:

Definition 9 (Early Bisimulation (using $\longrightarrow_l$)) *Given an STS $(S, \longrightarrow)$, the early bisimulation relation with respect to substitution θ, denoted by $\mathcal{R}_e^\theta$, is a subset of $S \times S$ such that*

$$s_1 \mathcal{R}_e^\theta s_2 \Rightarrow (\forall s_1\theta \xrightarrow{\alpha_1}_l t_1\theta \ \boxed{\forall\sigma}\ \boxed{\exists s_2\theta \xrightarrow{\alpha_2}_l t_2\theta}(\alpha_1[\theta\sigma]{=}\alpha_2[\theta\sigma]) \wedge t_1 \mathcal{R}_e^{\theta\sigma} t_2) \wedge\ s_2 \mathcal{R}_e^\theta s_1$$

This alternative definition of early bisimulation is especially important to a logic-programming-based encoding since early transition relations are hard to encode as logic programs.

4.1 Encoding Bisimulation Checkers as Logic Programs

We can encode checkers for equivalence with respect to late as well as early bisimulation following the encoding of bisimulation checkers for the finite-state case.

Early Bisimulation: Consider the complement of early bisimulation relation $\mathcal{R}_e$, written as $\overline{\mathcal{R}_e}$:

$$s_1 \overline{\mathcal{R}_e^\theta} s_2 \ \Leftarrow\ (\exists s_1\theta \xrightarrow{\alpha_1}_l t_1\theta\ \ \exists\sigma \forall s_2\theta \xrightarrow{\alpha_2}_l t_2\theta (\alpha_1[\theta\sigma] = \alpha_2[\theta\sigma]) \Rightarrow t_1 \overline{\mathcal{R}_e^{\theta\sigma}} t_2) \vee s_2 \overline{\mathcal{R}_e^\theta} s_1 \tag{2}$$

Since bisimulation is the largest such relation, the complement is naturally the least relation that satisfies the above equation. This relation can be encoded as a constraint logic program as follows:

```
nbisim(S1,S2) :- late_trans(S1,A1,T1), no_matching_trans(S1,A1,T1,S2).
nbisim(S1,S2) :- nbisim(S2,S1).
no_matching_trans(S1,A1,T1,S2) :-
    forall((A2,T2), late_trans(S2,A2,T2), nsimulate(A1,T1,A2,T2)).
nsimulate(A1,T1,A2,T2) :-   similar_act(A1,A2), nbisim(T1,T2)
                          ; not_similar_act(A1,A2).
```

Several differences are apparent between the finite-state bisimulation checker in Section 2 and the one given above. The first and most obvious difference is the use of `late_trans` for `trans`. The second is the use of a ternary `forall` predicate in order to explicitly differentiate between the bound and free variables. Note that in the finite-state case, there were no free variables in the universally quantified formula, and hence we could vastly simplify the way `forall` was encoded. In the infinite-state case we need to find consistent values for all the free variables used in the universally quantified formula ($\forall s_2\theta \xrightarrow{\alpha_2}_l t_2\theta \ldots$ in Equation 2).

The third difference is the use of `similar_act` (and `not_similar_act`) to check for $(\alpha_1[\theta\sigma] = \alpha_2[\theta\sigma])$ in Equation 2 (and its negation). Although `similar_act(A1,A2)` is `A1=A2`, `not_similar_act(A1,A2)` is not simply the negation of `A1=A2`, for the following reason. Two output actions $c!x$ and $c!y$ can be dissimilar as long as x and y can be bound to different values. Note that,

in contrast, since an input action creates a new bound variable, $c?x$ and $c?y$ are always similar. Hence, we have the following encoding for `similar_act` and `not_similar_act`:

```
similar_act(A1, A2) :- A1 = A2.
not_similar_act(A1, A2) :- A1 \= A2,
                           (   (A1 = in(C,_), (A2 = in(D,_), C\=D
                                               ; A2 = out(_,_)
                                               ; A2 = tau)
                               ; A1 = out(_,_)
                               ; A1 = tau)).
```

Late bisimulation: Let us now consider $\overline{\mathcal{R}_l}$, the complement of late bisimilarity relation:

$$s_1 \overline{\mathcal{R}_l^{\theta}} s_2 \Leftarrow (\exists s_1\theta \xrightarrow{\alpha_1}_l t_1\theta \quad \forall s_2\theta \xrightarrow{\alpha_2}_l t_2\theta \; \exists\sigma(\alpha_1[\theta\sigma] = \alpha_2[\theta\sigma]) \Rightarrow t_1 \mathcal{R}_l^{\theta\sigma} t_2) \vee s_2 \overline{\mathcal{R}_l^{\theta}} s_1 \tag{3}$$

The essence of this equation is that in order to show non-bisimilarity, for every transition from s_2 to t_2 we should find a *local* σ such that either the actions do not match, or t_1 and t_2 are non-bisimilar. This condition can be tested by simply ensuring that the different transitions from s_2 are standardized apart before checking for matching contexts. Standardization can be done via `copy_term/2` which generates a copy of a term with fresh variables. Late bisimulation can thus be derived from the encoding of early bisimulation by modifying `nsimulate/5` as follows:

```
nsimulate(A1, T1, A2, T2) :-
      ( similar_act(A1,A2),
        change_environments(A1, (T1,T2), (U1,U2)),
        nbisim(U1, U2))
    ; not_similar_act(A1,A2).
change_environments(in(_,_), E1, E2) :- copy_term(E1, E2).
change_environments(out(_,_), E1, E1).
change_environments(tau, E1, E1).
```

The predicate `change_environments/3` ensures that each transition on input action from s_2 is evaluated in a separate environment, as required by late bisimulation.

Discussion: Observe that the nested call to `nbisim/2` in the definition of `nsimulate` inherits a new set of constraints from the guards on the two selected transitions as well as the values under which the actions are similar. In our encoding, the current context in which `nbisim/2` is evaluated is maintained implicitly. This is a useful simplification as compared to the original algorithm of Hennessy and Lin [4], where the context of the bisimulation is maintained explicitly. The Hennessy-Lin algorithm returns the most general context under which the two processes are bisimilar. In a similar vein, when our encoding detects that two processes are not bisimilar, we can retrieve the context which witnesses the non-bisimilarity of the two processes.

The complexity of the evaluation is $O(|S| \times | \longrightarrow |)$ assuming unit-time table look up and constraint manipulation, which is same as Hennessy and Lin's

Table 1. Time for symbolic bisimulation checking

	Buffer Length (n)							
	10	20	30	40	50	60	70	80
$stack(n)$ vs. $queue(n)$	0.42	2.15	6.28	14.22	27.99	49.69	80.38	124.42
$stack(n)$ vs. $stack(n)$	0.16	0.54	1.19	2.22	3.68	5.60	8.14	11.37

procedural algorithm [4]. Furthermore, the encoding clearly separates the logical aspects of bisimulation from its representational aspects.

Implementation: Section 4.1 reveals that the bisimulation checker requires both equality and disequality constraints. We have implemented a constraint meta-interpreter that handles tabled logic programs over equality and disequality constraints in XSB. The meta-interpreter maintains the constraint store and simplifies the constraints as they are propagated, thus simulating a tabled CLP environment. We use the traditional trick of trading off the costs associated with maintaining constraint stores always in canonical form against the cost of extra resolution steps due to undetected inconsistencies in non-canonical constraint stores. We have also implemented `forall` to correctly handle free variables in quantified formula over equality domain. Finally, our symbolic bisimulation algorithm is sound but proof of completeness depends on the constraint domain; whereas our bisimulation algorithm for finite-state systems is both sound and complete. Details are available at `http://www.cs.sunysb.edu/~lmc/bisim`.

4.2 Experimental Results

We measured the performance of our symbolic bisimulation checker on an infinite-state version of stack and queue. Stacks and queues, denoted by $stack(n)$ and $queue(n)$, of different buffer lengths (n) but with unspecified domain, were defined as STSs (e.g., see Figure 4) and encoded as `sts_edge` facts. Note that even for fixed values of n, $stack(n)$ and $queue(n)$ are infinite-state systems since each element in them can store arbitrary data values.

Table 1 shows the time performance for checking *early* bisimulation comparing $stack(n)$ with $queue(n)$ and $stack(n)$ with $stack(n)$. Times for *late* bisimilarity checking are about 2% more than their early bisimulation counterpart due to the extra `copy_term` overhead. At first sight, this table displays an anomaly: time taken to check bisimilarity when two systems are not bisimilar (Table 1(row 1)) is more than that for systems which are bisimilar (Table 1(row 2)) for the same buffer length. This appears to contradict our previous observation that local bisimulation explores less state space and takes less time when the systems under consideration are not bisimilar as compared to the case when the systems are bisimilar. However closer inspection reveals that the symbolic state space explored in checking for bisimilarity between two stacks is much less than symbolic state space explored when checking for bisimilarity between a stack and a queue. In fact, the symbolic (global) state space explored for checking

bisimilarity between two stacks is linear in the buffer length. In contrast, the proof for non-bisimilarity of stack and queue depends both on buffer length and data domain size. It is worth mentioning that the state space explored for local bisimilarity checking depends greatly on the way the two transition systems are encoded. In case of checking for bisimilarity between a queue and a stack, if we first explore transitions with output (*delete*) actions before exploring those with input (*insert*) actions the states needed to be explored before the first non-similarity is detected is independent of the buffer length.

It should also be noted that the symbolic state space of these systems may, in fact, be smaller than the ground state space even for data domain sizes as low as 2. For instance, consider the symbolic state space of a 2-element queue in Figure 4(b). Its symbolic state space has 3 states, since the states $q_1(x)$ and $q_1(y)$ are simply variants of each other (i.e., identical modulo names of variables), and hence identified as a single symbolic state. In contrast, even for a 2-element data domain, say $\{1, 2\}$, observe that $q_1(1)$ is a state distinct from $q_1(2)$. Moreover, $q_2(x, y)$ and $q_2(y, x)$ represent the same symbolic state, while $q_2(1, 2)$ and $q_2(2, 1)$ behave differently. This is one of the key reasons why we can do symbolic bisimulation checking comparing two stacks with buffer length as high as 80 in around 11 seconds, whereas in the non-symbolic case, even comparing two stacks with buffer length of 18 each and with data domain size of just 2 explores over over $250K$ states, taking over 270 seconds in that process.

Finally, the meta-interpretation of constraints in the symbolic bisimulation imposes a heavy performance overhead. Using the symbolic bisimulation checker for LTSs (i.e., ground transition systems) is nearly 60 times slower than using the finite-state bisimulation checker. It is expected that a cleverer encoding of the constraint solver, together with the use of attributed variables [2] to integrate the solver with the LP engine, will significantly lower these overheads.

5 Conclusion

In this paper we demonstrated how the power and versatility of tabled logic programming can be used for checking bisimilarity of infinite-state systems in a natural way. Our implementation is goal-directed, i.e., we explore only states needed to prove or disprove the bisimilarity of the given states, and it can handle both early and late versions of strong as well as weak bisimilarity. Furthermore, the complexity of this implementation matches that of Hennessy and Lin's algorithm modulo table-lookup time. Our experimental results show that the symbolic bisimulation checker over an infinite-state system can be considerably more efficient to use compared to regular bisimulation checking over LTSs generated by finite instances of these systems, even for relatively small domain sizes. Applying the symbolic checker to real-life verification problems thus appears feasible despite significant overheads imposed by the constraint solver.

A recent paper [8] explored the use of constraint logic programs for checking bisimilarity of timed systems, where timed systems are encoded by their corresponding transition relation. The encoding used in that work can be seen as

specializing the `nbisim` predicate with respect to the transition relation. It is therefore expected that our encoding can be used for checking bisimilarity of timed systems. However, the performance of the checker will crucially hinge on the performance of a constraint solver for linear constraints needed to evaluate queries over timed systems.

References

1. CADP. Caesar/Aldebaran Development Package c1.112, 2001. Available from `http://www.inrialpes.fr/vasy/cadp.html`.
2. B. Cui and D.S. Warren. A system for tabled constraint logic programming. In *Computational Logic*, pages 478–492, 2000.
3. CWB-NC. The Concurrency Workbench of New Century v1.1.1, 2001. Available from `www.cs.sunysb.edu/~cwb`.
4. M. Hennessy and H. Lin. Symbolic bisimulations. *Theoretical Computer Science*, 138:353–389, 1995.
5. P.C. Kanellakis and S.A. Smolka. CCS expressions, finite state processes, and three problems of equivalence. *Information and Computation*, 86(1):43–68, May 90.
6. Z. Li and H. Chen. Computing strong/weak bisimulation equivalences and observation congruence for value-passing processes. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 300–314, 1999.
7. H. Lin. Symbolic transition graphs with assignments. In *Concurrency Theory (CONCUR)*, pages 50–65, 1996.
8. S. Mukhopadhyay and A. Podelski. Constraint database models characterizing timed bisimilarity. In *Practical Applications of Declarative Language*, 2001.
9. R. Paige and R. E. Tarjan. Three partition refinement algorithms. *SIAM Journal of Computing*, 16(6):973–989, December 1987.
10. J. Parrow. An introduction to π-calculus. In J.A. Bergstra, A. Ponse, and S.A. Smolka, editors, *Handbook of Process Algebra*, chapter 8, pages 479–544. North-Holland, 2001.
11. Y. S. Ramakrishna, C. R. Ramakrishnan, I. V. Ramakrishnan, S. A. Smolka, T. Swift, and D. S. Warren. Efficient model checking using tabled resolution. In *Proceedings of the 9th International Conference on Computer-Aided Verification (CAV '97)*, Haifa, Israel, July 1997. Springer-Verlag.
12. C.R. Ramakrishnan, I.V. Ramakrishnan, S.A. Smolka, Y. Dong, X. Du, A. Roychoudhury, and V.N. Venkatakrishnan. XMC: A logic-programming-based verification toolset. In *Computer Aided Verification (CAV)*, 2000.
13. XSB. The XSB logic programming system v2.3, 2001. Available from `http://www.cs.sunysb.edu/~sbprolog`.

A Simple Scheme for Implementing Tabled Logic Programming Systems Based on Dynamic Reordering of Alternatives

Hai-Feng Guo[1] and Gopal Gupta[2]

[1] Computer Science Dept, SUNY at Stony Brook, NY 11794;
haifeng@cs.sunysb.edu.
[2] Computer Science Dept., Univ. of Texas at Dallas, TX 75083;
gupta@utdallas.edu.

Abstract. Tabled logic programming (LP) systems have been applied to elegantly and quickly solving very complex problems (e.g., *model checking*). However, techniques currently employed for incorporating tabling in an existing LP system are quite complex and require considerable change to the LP system. We present a simple technique for incorporating tabling in existing LP systems based on *dynamically reordering* clauses containing *variant* calls at run-time. Our simple technique allows tabled evaluation to be performed with a *single* SLD tree and without the use of complex operations such as *freezing* of stacks and heap. It can be incorporated in an existing logic programming system with a small amount of effort. Our scheme also facilitates exploitation of parallelism from tabled LP systems. Results of incorporating our scheme in the commercial ALS Prolog system are reported.

1 Introduction

Traditional logic programming systems (e.g., Prolog) use SLD resolution with the following *computation strategy* [11]: subgoals of a resolvent are tried from left to right and clauses that match a subgoal are tried in the textual order they appear in the program. It is well known that SLD resolution may lead to non-termination for certain programs, even though an answer may exist via the declarative semantics. In fact, this is true of any "static" computation strategy that is adopted. That is, given any static computation strategy, one can always produce a program that will not be able to find the answers due to non-termination even though finite solutions may exist. In case of Prolog, programs containing certain types of left-recursive clauses are examples of such programs.

To get around this problem, researchers have suggested computation strategies that are *dynamic* in nature coupled with recording solutions in a *memo table*. A dynamic strategy means that the decision regarding which clause to use next for resolution is taken based on run-time properties, e.g., the nature and type of goals in the current resolvent. OLDT [18] is one such computation strategy, in which solutions to certain subgoals are recorded in a *memo table* (heretofore

P. Codognet (Ed.): ICLP 2001, LNCS 2237, pp. 181–196, 2001.

referred to simply as a *table*). Such a call that has been recorded in the table is referred to as a *tabled call*. In OLDT resolution, when a tabled call is encountered, computation is started to try the alternative branches of the original call and to compute solutions, which are then recorded in the table. These solutions are called *tabled solutions* for the call. When a call to a subgoal that is identical to a previous call is encountered while computing a tabled call—such a call is called a variant call and may possibly lead to non-termination if SLD resolution is used—the OLDT resolution strategy will not expand it as SLD resolution will, rather the solutions to the variant call will only be obtained by matching it with tabled solutions. If any solutions are found in the table, they are *consumed* one by one just as a list of fact clauses by the variant call, each producing a solution for the variant call. Next, the computation of the variant subgoal is suspended until some new solutions appear in the table. This consumption and suspension continues, until all the solutions for the tabled call have been generated and a *fixpoint* reached. Tabled LP systems have been put to many innovative uses. A tabled LP system can be thought of as an engine for efficiently computing fixpoints. Efficient fixpoint computation is critical for many applications, e.g., model checking [14], program analysis [13], non-monotonic reasoning [3].

In this paper, we present a novel, simple scheme for incorporating tabling in a standard logic programming system. Our scheme, which is based on *dynamic reordering of alternatives* (DRA) that contain variant calls, allows one to incorporate tabling in an existing LP system with very little effort. Using DRA we were able to incorporate tabling in the commercial ALS Prolog system [1] in a few months of work. The time efficiency of our tabled ALS (TALS) system is comparable to that of the XSB system [2,17,5,7,23] and B-Prolog [21], the two tabled LP systems currently available[1]. The space efficiency of TALS is comparable to that of B-Prolog and XSB with local scheduling and better than that of XSB with batch scheduling (XSB's current default scheduling strategy). Unlike traditional implementations of tabling [2], DRA works with a single SLD tree without requiring suspension of goals and freezing of stacks. Additionally, no extra overhead is incurred for non-tabled programs. Intuitively, DRA builds the search tree as in normal Prolog execution based on SLD, however, when a variant tabled call is encountered, the branch that lead to that variant call is "moved" to the right of the tree. Essentially, branches of the search tree are reordered during execution to avoid exploring potentially non-terminating branches. The principal advantage of DRA is that because of its simplicity it can be incorporated very easily and efficiently in existing Prolog systems.

In our dynamic alternative reordering strategy, not only are the solutions to variant calls tabled, the alternatives leading to variant calls are also memorized in the table (these alternatives, or clauses, containing variant calls are called *looping alternatives* in the rest of the paper). A tabled call first tries its non-looping alternatives (tabling any looping alternatives that are encountered along the way). Finally, the tabled call repeatedly tries its looping alternatives until it reaches a fixpoint. This has the same effect as shifting branches with variant

[1] YAP [16] is another system with tabling; its implementation mimics XSB.

calls to the right in the search tree. The simplicity of our scheme guarantees that execution is not inordinately slowed down (e.g., in the B-Prolog tabled system [21], a tabled call may have to be re-executed several times to ensure that all solutions are found), nor considerable amount of memory used (e.g., in the XSB tabled system [2] a large number of stacks/heaps may be frozen at any given time), rather, the raw speed of the Prolog engine is available to execute even those programs that contain variant calls.

An additional advantage of our technique for implementing tabling is that parallelism can be naturally exploited. In traditional tabled systems such as XSB, the ideas for parallelism have to be reworked and a new model of parallelism derived [6,16]. In contrast, in a tabled logic programming system based on dynamic reordering, the traditional forms of parallelism found in logic programming (or-parallelism and and-parallelism) can still be exploited. Work is in progress to augment the or-parallel ALS system [1,8] (currently being developed by us [20,9]) with tabling [9].

A disadvantage of our approach is that certain *non-tabled* goals occurring in looping alternatives may be computed more than once. However, this recomputation can be eliminated by the use of tabling, automatic program transformation, or more sophisticated reordering techniques (see later).

2 SLD and OLDT Resolution

Prolog was initially designed to be a declarative language [11], i.e., a logic program with a correct declarative semantics should also get the same results via its procedural semantics. However, the operational semantics of standard Prolog systems that adopt SLD resolution (leftmost-first selection rule and a depth-first search rule) is not close to their declarative semantics. The completeness of SLD resolution ensures that given a query, the solutions implied by the program, if they exist, can be obtained through computation paths in the SLD tree [11]. However, standard Prolog systems with a pre-fixed computation rule may only compute a subset of these solutions due to problems with non-termination.

Example 1. Consider the following program:

```
r(X, Y) :- r(X, Z), r(Z, Y).        (1)
r(X, Y) :- p(X, Y), q(Y).           (2)
p(a, b).   p(a, d).   p(b, c).
q(b).      q(c).
:- table r/2.
:- r(a, Y).
```

Following the declarative semantics of logic programs (e.g., employing bottom-up computation), the example program 1 above should produce two answers $Y{=}b$ and $Y{=}c$. However, standard Prolog system will go into an infinite loop for this program. It is Prolog's computation rule that causes the inconsistency between its declarative semantics and procedural semantics. With the leftmost-first selection rule and depth-first search rule, Prolog systems are trapped in an

infinite loop in the SLD-tree even though computation paths may exist to the solutions. It seems that breadth-first search strategy may solve the problem of infinite-looping, and it does help in finding the first solution. However, if the system is required to find all the solutions and terminate, breadth-first search is not enough, since the SLD tree may contain branches of infinite length.

To get around this problem, a tabled evaluation strategy called OLDT is used in tabled logic programming systems such as XSB. In the most widely available tabled Prolog systems, XSB, OLDT is implemented in the following way[2]. When a call to a tabled predicate is encountered for the first time, the current computation is suspended and a new SLD tree is built to compute the answers to this tabled call. The new tree is called a *generator*, while the old tree (which led to the tabled call) is called a *consumer* w.r.t. the new tabled call. When a call that is a variant of a previous call—and that may potentially cause infinite loop under SLD—is encountered in the generator SLD tree, XSB first consumes the tabled solutions of that call (i.e., solutions that have already been computed by the previous call). If all the tabled solutions have been exhausted, the current call is suspended until some new answers are available in the table. Finally, the solutions produced by the generator SLD tree are consumed by the consumer SLD tree after its execution is resumed. In XSB, the suspension of the consumer SLD tree is realized by freezing the stacks and heap. An implementation based on suspension and freezing of stacks may be quite complex to realize as well as it can incur substantial overhead in terms of time and space. Considerable effort is needed to make such a system very efficient. In this paper, we present a simple scheme for incorporating tabling in a Prolog system in a small fraction of this time. Additionally, our system is comparable in efficiency to existing systems w.r.t. time and space.

The OLDT resolution forest for example 1 following XSB style execution is shown in figure 1. (The figure also shows the memo-table used for recording solutions; the numbers on the edges of the tree indicate the order in which XSB will generate those edges). Compared to SLD, OLDT has several advantages: (i) A tabled Prolog system avoids redundant computation by memoing the com-

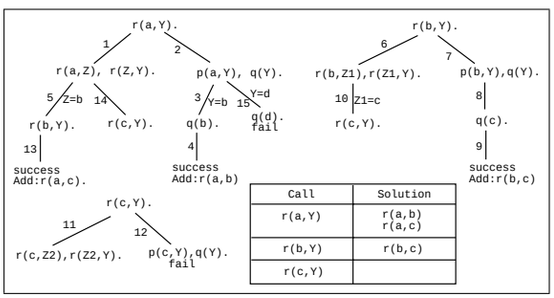


Fig. 1. An OLDT Tree

[2] The current XSB system uses SLG resolution (OLDT augmented with negation).

puted results; in some cases, it can reduce the time complexity of a problem from exponential to polynomial. (ii) A tabled Prolog system terminates for all queries posed to bounded term-sized programs that have a finite least fixpoint. (iii) Tabled Prolog keeps the declarative and procedural semantics of definite Prolog programs consistent.

3 Dynamic Reordering of Alternatives (DRA)

We present a simple technique for implementing tabling that is based on *dynamic reordering of looping alternatives* at run-time, where a *looping alternative* refers to a clause that matches a tabled call containing a recursive variant call. Intuitively, our scheme works by reordering the branches in SLD trees. Branches containing variant calls are moved to the right in the SLD tree for the query. In our scheme, a tabled call can be in one of three possible states: *normal state*, *looping state*, or *complete state*. The state transition graph is shown in figure 2.

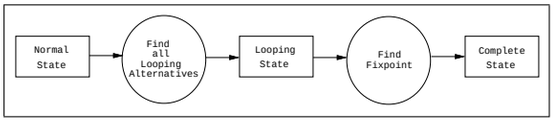


Fig. 2. State Transition Graph

Consider any tabled call $\mathcal{C}$, normal state is initially entered when $\mathcal{C}$ is first encountered during the computation. This first occurrence of $\mathcal{C}$ is allowed to explore the matched clauses as in a standard Prolog system (normal state). In normal state, while exploring the matching clauses, the system tables all the solutions generated for the call $\mathcal{C}$ in this state and also checks for variants of $\mathcal{C}$. If a variant is found, the current clause that matches the original call to $\mathcal{C}$ will be memorized, i.e., recorded in the table, as a *looping alternative*. This call will not be expanded at the moment because it can potentially lead to an infinite loop. Rather it will be solved by consuming the solutions from the table that have been computed by other alternatives. To achieve this, the alternative corresponding to this call will be reordered and placed at the end of the alternative list in the choice-point. A failure will be simulated and the alternative containing the variant will be backtracked over. After exploring all the matched clauses (some of which were possibly tabled as looping alternative), $\mathcal{C}$ goes into its looping state. From this point, tabled call $\mathcal{C}$ keeps trying its tabled looping alternatives repeatedly (by putting the alternative again at the end of the alternative list after it has been tried) until $\mathcal{C}$ is completely evaluated. If no new solution is added to $\mathcal{C}$'s tabled solution set in any one cycle of trying its tabled looping alternatives, then we can say that $\mathcal{C}$ has reached its fixpoint.

$\mathcal{C}$ enters its *complete state* after it reaches its fixpoint, i.e., after all solutions to $\mathcal{C}$ have been found. In the complete state, if the call $\mathcal{C}$ is encountered again

later in the computation, the system will simply use the tabled solutions recorded in the table to solve it. In other words, $\mathcal{C}$ will be solved simply by consuming its tabled solution set one after another as if trying a list of facts.

Considerable research has been devoted to evaluating recursive queries in the field of deductive databases [4]. Intuitively, the DRA scheme can be thought roughly equivalent to the following deductive query evaluation scheme for computing fixpoints of recursive programs: (i) first find all solutions to the query using only *non-recursive clauses* in a top-down fashion, (ii) use this initial solution set as a starting point and compute (semi-naively) the fixpoint using the recursive clauses in a bottom up fashion. By using the initial set obtained from top-down execution of the query using non-recursive clauses, only the answers to the query are included in the final fixpoint. Redundant evaluations are thus avoided as in Magic set evaluation. The proof of correctness of DRA is based on formally showing its equivalence to this evaluation scheme [9,10].

Example 2. Consider resolving the following program's evaluation using DRA:

```
r(X, Y) :- r(X, Z), p(Z, Y).     (1)
r(X, Y) :- p(X, Y).              (2)
r(X, Y) :- r(X, Z), q(Z, Y).     (3)
p(a, b).   p(b, c).   q(c, d).
:- table r/2.
:- r(a, Y).
```

Figure 3 gives the computation tree produced by DRA for example 2 (note that the labels on the branch refer to the clause used for creating that branch). Both clause (1) and clause (3) need to be tabled as looping alternatives for the tabled call `r(a, Y)` (this is accomplished by operations `a_add:(1)` and `a_add:(3)` shown in Figure 3). The second alternative is a non-looping alternative that produces a solution for the call `r(a,Y)` which is recorded in the table (via the operation `s_add` shown in the Figure). The query call `r(a, Y)` is a master tabled call (since it is the first call), while all the occurrences of `r(a, Z)` are slave tabled calls (since they are calls to variant of `r(a,Y)`). When the call `r(a, Y)` enters its looping state, it keeps trying the looping alternatives repeatedly until the solution set does not change any more, i.e., until `r(a, Y)` is completely evaluated (this is accomplished by trying a looping alternative, and then moving it to the end of the alternatives list). Note that if we added two more facts: `p(d,e)` and `q(e,f)`, then we'll have to go through the two looping alternatives one more time to produce the solutions `r(a,e)` and `r(a,f)`.

An important problem that needs to be addressed in any tabled system is *detecting completion*. When there are multiple tabled calls occurring simultaneously during the computation, and results produced by one tabled call may depend on another's, then knowing when the computation of a tabled call is complete (i.e., all solutions have been computed) is quite hard. Completion detection based on finding *strongly connected components* (SCC) has been implemented in the TALS system (details are omitted due to lack of space and can be found elsewhere [9,10]). Completion detection is very similar to the procedure employed in XSB and the issues are illustrated in the next two examples.

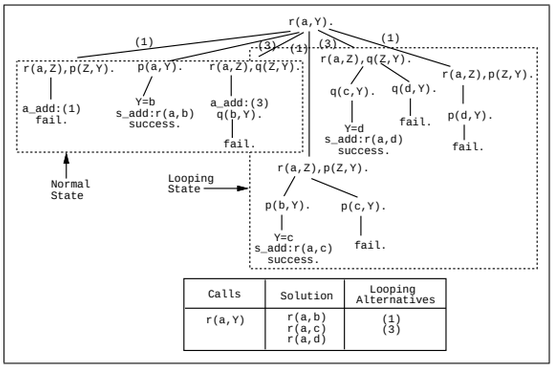


Fig. 3. DRA for Example 2

Example 3. Consider resolving the following program with DRA:

```
r(X, Y) :- r(X, Z), r(Z, Y).      (1)
r(X, Y) :- p(X, Y), q(Y).         (2)
p(a, b).   p(a, d).   p(b, c).
q(b).      q(c).
:- table r/2.
:- r(a, Y).
```

As shown in the computation tree of Figure 4, the tabled call `r(b, Y)` is completely evaluated only if its dependent call `r(c, Y)` is completely evaluated, and `r(a, Y)` is completely evaluated only if its dependent calls, `r(b, Y)` and `r(c, Y)`, are completely evaluated. Due to the depth-first search used in TALS, `r(c, Y)` always enters its complete state ahead of `r(b, Y)`, and `r(b, Y)` ahead of `r(a, Y)`. The depth-first strategy with alternative reordering guarantees for such dependency graphs (i.e., graphs with no cycles) that dependencies can be

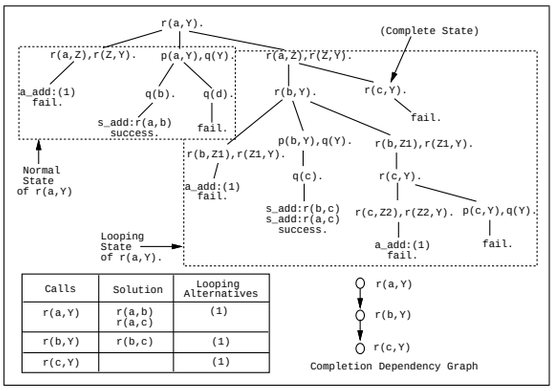


Fig. 4. DRA for Example 3

satisfied without special processing during computation. However, these dependencies can be cyclic as in the following example.

Example 4. Consider resolving the following program with DRA:

```
r(X, Y) :- p(X, Z), r(Z, Y).     (1)
r(X, Y) :- p(X, Y).              (2)
p(a, b).   p(b, a).
:- table r/2.
:- r(a, Y).
```

Figure 5 shows the complete computation tree of example 4. In this example, two tabled calls, `r(a, Y)` and `r(b, Y)`, are dependent on each other, forming a SCC in the completion dependency graph. It is not clear which tabled call is completely evaluated first. A proper semantics can be given to the program only if all tabled calls in a SCC reach their complete state simultaneously. According to depth-first computation strategy, the least deep tabled call of each SCC should be the last tabled call to reach its fixpoint in its SCC. To detect completion correctly, the table is extended to record the least deep tabled call of each SCC, so that the remaining calls in the SCC can tell whether they are in the complete state by checking the state of the least deep call. The state of a tabled call can be set to "complete" only after its corresponding least deep call is in a complete state. In this example, there are two occurrences of `r(b, Y)` during the computation. In its first occurrence, `r(b, Y)` can not be set to "complete" even though it reaches a *temporary* fixpoint after exploring its looping alternative, because it depends on the tabled call `r(a, Y)`, which is not completely evaluated yet. If the call `r(b, Y)` is set to "complete" state at this point, a solution `r(b, b)` will be lost. Only after the tabled call `r(a, Y)` is completely evaluated during its looping state, can the tabled call `r(b, Y)` (within the same SCC with `r(a, Y)`) be set to complete state.

4 Implementation

The DRA scheme can be easily implemented on top of an existing Prolog system. TALS is an implementation of DRA on top of the commercial ALS Prolog system. In the TALS system, tabled predicates are explicitly declared. Tabled solutions are consumed incrementally to mimic semi-naive evaluation [4]. Memory management and execution environment can be kept the same as in a regular Prolog engine. Two main data structures, *table* and *tabled choice-point stack*, are added to the TALS engine. The table data structure is used to keep information regarding tabled calls such as the list of tabled solutions and the list of looping alternatives for each tabled call, while tabled choice-point stack is used to record the properties of tabled call, such as whether it is a master call (the very first call) or a slave call (call to the variant in a looping alternative). The master tabled call is responsible for exploring the matched clauses, manipulating execution states, and repeatedly trying the looping alternatives and solutions for the corresponding tabled call, while slave tabled calls only consume tabled solutions.

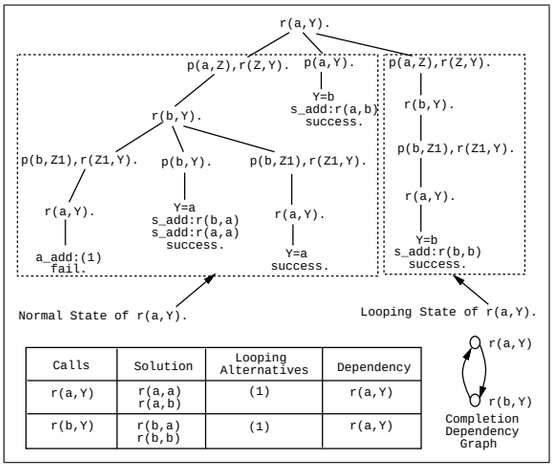


Fig. 5. Example 4

The allocation and reclaiming of master and slave choice-points is similar to regular choice-points, except that the former have a few extra fields to manage the execution of tabled calls.

Very few changes are required to the WAM engine of a Prolog system to implement the DRA scheme (more implementation details can be found elsewhere [9]). We introduce six new WAM instructions, needed for tabled predicates: `table_try_me_else`, `table_retry_me_else`, `table_trust_me`, `table_loop`, `table_consume`, and `table_save`. We differentiate between tabled calls and non-tabled calls at compile-time, and generate appropriate type of WAM `try` instructions. For regular calls, WAM `try_me_else`, `retry_me_else`, and `trust_me_else`, instructions are generated to manage the choice-points, while for tabled calls, these are respectively modified to `table_try_me_else`, `table_retry_me_else`, and `table_trust_me_else` instructions. Every time `table_try_me_else` is invoked, we have to check if the call is a variant of a previous call. If the call is a variant, the address of the WAM code corresponding to this clause is recorded in the table as a looping alternative. The variant call is treated as a *slave* tabled call, which will only consume tabled solutions if there are any in the table, and will not explore any matched clauses. The next-alternative-field of the slave choice-point is changed to `table_consume` so that it repeatedly consumes the next available tabled solutions. If the call is a new tabled call, it will be added into the table by recording the starting code address and its arguments information. This new tabled call is treated as a *master* tabled call, which will explore the matched clauses and generate new solutions. The continuation instruction of a master tabled call is changed to a new WAM instruction `table_save`, which checks if generated solution is new. If so, the new solution is tabled, and execution continues with the sub-goal after the tabled call as in normal Prolog execution. When the last matched alternative of the master choice-point is tried

by first executing the `table_trust_me` instruction, the next-alternative-field of the master choice-point is set to the instruction `table_loop`, so that after finishing the last matched alternative, upon backtracking, the system will enter the looping state to try the looping alternatives. After a fixpoint is reached, and all the solutions have been computed, this instruction is changed to the WAM `trust_me_fail` instruction, which de-allocates the choice-point and simulates a failure, as in normal Prolog execution.

5 Recomputation Issues in DRA

The main idea in the TALS system is to compute the base solution set for a tabled call using clauses not containing variants, then repeatedly applying the clauses with variants (looping alternatives) on this base solution set until the fixpoint of the tabled call is reached. Due to the looping alternatives being repeatedly executed, certain *non-tabled* goals occurring in these clauses may be unnecessarily re-executed. This recomputation can affect the overall efficiency of the TALS system. Non-tabled calls may be redundantly re-executed in the following three situations.

First, while trying a looping alternative, the whole execution environment has to be built again until a slave tabled choice-point is created. Consider the looping alternative:

```
p(X, Y) :- q(X), p(X, Z), r(Z, Y).
...
:- table p/2.
:- p(a, X).
```

Suppose `p/2` is a tabled predicate, while `q/1` and `r/2` are not. Then each time this alternative is tried, `q(X)` has to be computed since it is not a tabled call. That is, the part between the master tabled call and slave tabled call has to be recomputed when this alternative is tried again.

Second, false looping alternatives may occur and may require recomputation. Consider the program below:

```
p(1).
p(2).
:- table p/1.
:- p(X), p(Y).
```

After the first goal `p(X)` gets the solution `p(1)`, a variant call of `p(X)`, namely, `p(Y)`, is met. According to the DRA scheme, the explored clause is then tabled as a looping alternative. However, all the matched clauses `p(1)` and `p(2)` are determinate facts, which will not cause any looping problem. The reason we falsely think that there is a looping alternative is because it is difficult to tell whether `p(Y)` is a descendant of `p(X)` or not. Even worse, the false looping alternatives will generate the solutions in a different order (order is "`X=1`, `Y=1`", "`X=2`, `Y=1`", "`X=2`, `Y=2`", "`X=1`, `Y=2`" versus "`X=1`, `Y=1`", "`X=1`, `Y=2`", "`X=2`, `Y=1`",

"X=2, Y=2" for standard Prolog). This problem of false looping alternative is also present in XSB and B-Prolog.

Third, a looping alternative may have multiple clause definitions for its non-tabled subgoals. Each time a looping alternative is re-tried, all the matching clauses of its non-tabled subgoals have to be computed. For example:

```
p(a, b).
p(X, Y) :- p(X, Z), q(Z, Y).   (1)
p(X, Y) :- t(X, Y).            (2)
t(X, Y) :- p(X, Z), s(Z, Y).   (3)
t(X, Y) :- s(X, Y).            (4)
...
:- table p/2.
:- p(a, X).
```

For the query `p(a, X)`, clause `(1)` and clause `(2)` are two looping alternatives. Consider the second looping alternative. The predicate `p(X, Y)` is reduced to the predicate `t(X, Y)`, which has two matching clauses. The first matching clause of `t(X, Y)`, clause `(3)`, leads to a variant call of `p(X, Y)`, while the second matching clause, clause `(4)`, is a determinate clause. However, each time the looping alternative, clause `(2)`, is re-tried, both matching clauses for the predicate `t(X, Y)` are tried. However, because clause (4) does not lead to any variant of the tabled call, this recomputation is wasted.

In the first case, fortunately, recomputation can be avoided by explicitly tabling the predicate `q/1`, so that `q(X)` can consume the tabled solutions of `q` instead of recomputing them. XSB does not have this problem with recomputation, because XSB freezes the whole execution environment, including the computation state of `q(X)`, when the variant call `p(X,Z)` is reached. This freezing of the computation state of `q(X)` amounts to implicitly tabling it.

The second case can be solved by finding the scope of the master call. If we know that `p(Y)` is out of the scope of `p(X)`, we can compute `p(X)` first, then let the variant call `p(Y)` only consume the tabled solutions. However, one assumption is that the tabled call `p(X)` has a finite fixpoint and thus can be completely evaluated.

The final case can be handled in several ways. One option is to table the specific computation paths leading to the variants of a previous tabled call instead of the whole looping alternative. However, tabling the computation paths will incur substantial overhead. Second option is to table the non-tabled predicates, such as `t(X, Y)`, so that the determinate branches of `t(X, Y)` will not be re-tried. A third option is to unfold the call to `t(X,Y)` in the clause (2) of predicate `p` so that the intermediate predicate `t(X,Y)` is eliminated.

Thus, all cases where non-tabled goals may be redundantly executed can be eliminated. Note that tabling of goals `q(X)` in case (i) and of goal `t(X,Y)` in case (iii) can be done automatically. The unfolding in case (iii) can also be done automatically. At present, in TALS these transformations have to be done manually.

6 Related Work

The most mature implementation of tabling is the XSB [2,19] system from SUNY Stony Brook. As discussed earlier, the XSB system implements OLDT by developing a forest of SLD trees, suspension of execution via freezing of corresponding stacks/heap, and resumption of execution via their unfreezing. Recently, improvements of XSB, called CAT and CHAT [5], that reduce the amount of storage locked up by freezing, have been proposed. Of these, the CHAT system seems to achieve a good balance between time and space overhead since it only freezes the heap, the state of the other stacks is captured and saved in a special memory area (called CHAT area).

Because of considerable investment of effort in design and optimization of the XSB system [2,17,5,23], XSB has turned out to be an extremely efficient system. The modified WAMs that have been designed [23,17], the research done in scheduling strategies [7] for reducing the number of suspensions and reducing space usage [5] are crucial to the efficiency of the XSB system. Ease of implementation and space efficiency are the main advantages of DRA. The scheme based on DRA is quite simple to implement on an existing WAM engine, and produces performance that is comparable to XSB.

Recently, another implementation of a tabled Prolog system—based on *SLDT* and done on top of an existing Prolog system called B-Prolog—has been reported [21]. The main idea behind SLDT is as follows: when a variant is recursively reached from a tabled call, the active choice-point of the original call is transferred to the call to the variant (the variant *steals* the choice-point of the original call, using the terminology in [21]). Suspension is thus avoided (in XSB, the variant call will be suspended and the original call will produce solutions via backtracking) and the computation pattern is closer to SLD. However, because the variant call avoids trying the same alternatives as the previous call, the computation may be incomplete. Thus, repeated *recomputation* [21] of *tabled calls* is required to make up for the solutions lost and to make sure that the fixpoint is reached. Additionally, if there are multiple clauses containing recursive variant calls, the variant calls may be encountered several times in one computation path. Since each variant call executes from the backtracking point of a former variant call, a number of solutions may be lost. These lost solutions have to be found by recomputation. This recomputing may have to be performed several times to ensure that a fixpoint is reached, compromising performance.

Observe that the DRA (used in TALS) is not an improvement of SLDT (used in B-Prolog) rather a completely new way of implementing a tabled LP system (both were conceived independently). The techniques used for evaluating tabled calls and for completion detection in the TALS system are quite different (even though implementations of both DRA and SLDT seem to be manipulating choice-points). B-Prolog is efficient only for certain types of very restricted programs (referred to as *directly recursive* in [21]; i.e., programs with one recursive rule containing one variant call). For programs with multiple recursive rules or with multiple variant calls, the cost of computing the fixpoint in B-Prolog can be substantial.

7 Performance Results

The TALS system has been implemented on top of the WAM engine of the commercial ALS Prolog system. It took us less than two man-months to research and implement the dynamic reordering of alternatives (DRA) scheme (with semi-naive evaluation) along with full support for complex terms on top of commercial ALS Prolog system (tabled negation is not yet supported, work is under way). Our performance data indicates that in terms of time and space efficiency, our scheme is comparable to XSB and B-Prolog. The main advantage of the DRA scheme is that it can be incorporated relatively easily in existing Prolog systems. Note that the most recent releases of XSB (version 2.3) and B-Prolog (version 5.0) were used for performance comparison. All systems were run on a machine with 700MHz Pentium processor and 256MB of main memory. Note that comparing systems is a tricky issue since all three systems employ a different underlying Prolog engine. Table 1 shows the performance of the three systems on regular Prolog programs (i.e., no predicates are tabled) and gives some idea regarding the relative speed of the engines employed by the 3 systems (arithmetic on the TALS system seems to be slow, which is the primary reason for its poor performance on the 10-Queens and Knight benchmarks compared to other systems). Note that `Sg` is the "cousin of the same generation" program, `10-Queen` is the instance of N-Queen problem, `Knight` is the Knight's tour, `Color` is the map-coloring problem, and `Hamilton` is the problem of finding Hamiltonian cycles in a graph. Note that all figures for all the systems are for all solution queries.

In general, the time performance of TALS on most of the CHAT benchmarks is worse than that of XSB, however, it is not clear how much of it is due to the differences in base engine speed, and how much is due to TALS' recomputation of non-tabled goals leading up to looping alternatives (the fix for this described in section 5 could not be used, as the CHAT benchmarks are automatically generated from some pre-processor and are unreadable by humans). However, except for `read` the performance is comparable, (i.e., it is not an order of magnitude worse). With respect to B-Prolog the time-performance is mixed. For programs with multiple looping alternatives TALS performs better than B-Prolog. Table 2 compares the time efficiency among XSB, B-Prolog, and TALS system. These benchmarks are taken from the CHAT suite of benchmarks distributed with XSB and B-Prolog.[3] Most of these benchmarks table multiple predicates many of whom use structures. For XSB, timings for both batch scheduling (XSB-b) and local scheduling (XSB-l) are reported (Note that batch scheduling is currently the default scheduling strategy in XSB since local scheduling assumes an all-solutions query).

Tables 3 and 4 compare the space used by TALS, XSB (both batch and local scheduling), and B-Prolog systems. Table 3 shows the total space used by the

[3] Note that benchmarks used in Table 1 will not benefit much from tabling, except for sg, so a different set of benchmarks is used; most of the benchmarks used in Table 2 cannot be executed under normal Prolog.

Table 1. Running Time (Seconds) on Non-tabled Programs

Benchmarks	**10-Queen**	**Sg**	**Knight**	**Color**	**Hamilton**
XSB	0.441	0.301	2.63	0.08	1.18
B-Prolog	0.666	0.083	3.15	0.233	2.667
TALS	2.46	0.19	11.26	0.38	1.48

Table 2. Running Time (Seconds) on Tabled Programs

Benchmark	**cs_o**	**cs_r**	**disj**	**gabriel**	**kalah**	**peep**	**pg**	**read**	**sg**
TALS	0.16	0.37	0.26	0.72	0.42	0.52	0.29	5.94	0.04
XSB-b	0.081	0.16	0.05	0.06	0.05	0.18	0.05	0.23	0.06
XSB-l	0.071	0.13	0.041	0.05	0.04	0.131	0.041	0.18	0.05
B-Prolog	0.416	0.917	0.233	0.366	0.284	1.417	0.250	0.883	0.084

Table 3. Total Space Usage in Bytes (Excluding Table Space)

Benchmark	**cs_o**	**cs_r**	**disj**	**gabriel**	**kalah**	**peep**	**pg**	**read**	**sg**
TALS	8360	8438	12193	17062	23520	6800	20084	20426	2226
XSB-b	11040	13820	10012	30356	43628	1148296	436012	1600948	3096
XSB-l	6992	8584	6876	23156	9564	19448	16324	125342	3540
B-Prolog	21040	38592	16484	37596	61288	96884	64232	72916	1664

Table 4. Space Overhead for Tabling in Bytes

Benchmark	**cs_o**	**cs_r**	**disj**	**gabriel**	**kalah**	**peep**	**pg**	**read**	**sg**
TALS	672	750	213	190	376	976	420	2666	342
XSB-b	2544	4016	2568	16172	16784	1132596	363872	1356672	0
XSB-l	696	1392	1632	10848	1612	7732	7768	63720	0
B-Prolog	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a

system. This space includes total stack and heap space used as well as *space overhead* to support tabling (but excluding space used for storing the table). The *space overhead* to support tabling in case of TALS includes the extra space needed to record looping alternatives and extra fields used in master and slave choice-points. In case of both XSB-l and XSB-b, the figure includes the CHAT space used. For B-Prolog it is difficult to separate this overhead from the actual heap + stack usage. *Space overhead* incurred is separately reported in Table 4. As can be noticed from Table 3, the space performance of TALS is significantly better than that of XSB-b (for some benchmarks, e.g., peep, pg and read, it is orders of magnitude better). It is also better than the space performance of B-Prolog (perhaps due to the extra space used during recomputation in B-Prolog)

Table 5. Table Space Usage in Bytes

Benchmark	cs_o	cs_r	disj	gabriel	kalah	peep	pg	read	sg
TALS	21056	21400	6488	7244	13496	17256	3852	15404	25128
XSB-b	26572	27072	22768	199948	35784	22688	15876	48032	47568
XSB-l	25356	25858	21592	19076	34160	21920	15108	45944	42448
B-Prolog	20308	20396	20104	16492	26884	15260	13860	38388	69740

and is comparable in performance to XSB-l. For completeness sake, we also report the space used in storing the table for each of the 4 systems in Table 5.

8 Conclusion and Future Work

The advantages of DRA can be listed as follows: (i) It can be easily implemented on top of an existing Prolog system without modifying the kernel of WAM engine in any major way; (ii) It works with a single SLD tree without suspension of goals and freezing of stacks resulting in less space usage; (iii) Unlike SLDT, it avoids blindly recomputing subgoals (to ensure completion) by remembering looping alternatives; (iv) Unlike XSB with local scheduling it produces solutions for tabled goals incrementally while maintaining good space and time performance (v) Parallelism can be easily incorporated in the DRA scheme.

Our alternative reordering strategy can be thought of as a dual [12] of the Andorra-principle [22]. In the Andorra model of execution, goals in a clause are reordered (on the basis of run-time properties, e.g., determinacy) leading to a considerable reduction in search space and better termination behavior. Likewise, our tabling scheme based on reordering alternatives (which correspond to clauses) also reduces the size of the computation (since solutions for tabled call once computed are remembered) and results in better termination behavior.

Our scheme is quite simple to implement. We were able to implement it on top of an existing Prolog engine (ALS Prolog) in a few weeks of work. Performance evaluation shows that our implementation is comparable in performance to highly-engineered tabled systems such as XSB. Work is in progress to add support for tabled negation and or-parallelism, so that large and complex applications (e.g., model-checking) can be tried.

Acknowledgments

This research is partially supported by NSF grants EIA 01-30847, CCR 99-00320, CCR 98-20852, CDA 97-29848, EIA 97-29848, and INT 99-04063. Hai-Feng Guo is also supported by a Post-doctoral fellowship from NSF. Thanks to D. Warren, I.V. Ramakrishnan and C. R. Ramakrishnan of SUNY Stony Brook for comments/discussions; to K. Bowen and C. Houpt of ALS, Inc., for providing us with the source code of the ALS Prolog system and explaining its intricacies, to N-F Zhou, B. Demoen, and K. Sagonas for help with benchmarks, and to the anonymous referees.

References

1. K. Bowen, C. Houpt, et al. ALS Prolog System. www.als.com.
2. W. Chen and D. S. Warren. Tabled Evaluation with Delaying for General Logic Programs. *JACM*, 43(1):20-74, January 1996.
3. W. Chen, T. Swift, and D. Warren. Efficient top-down computation of queries under the well-founded semantics. J. Logic Programming, 24(3):161-199.
4. S. Das. Deductive Databases and Logic Programming. Addison Wesley. 1992.
5. B. Demoen and K. Sagonas. CHAT: the Copy-Hybrid Approach to Tabling. PADL '99, Springer Verlag LNCS 1551, pages 106-121.
6. J. Freire, R. Hu, T. Swift, D. Warren. Exploiting Parallelism in Tabled Evaluations. PLILP 1995: 115-132. LNCS 982.
7. J. Freire, T. Swift, D. S. Warren. Beyond Depth-First Strategies: Improving Tabled Logic Programs through Alternative Scheduling. JFLP 1998(3). MIT Press.
8. G. Gupta, E. Pontelli. Stack-splitting: A Simple Technique for Implementing Or-parallelism in Logic Programming Systems. In *Proc. ICLP*, 1999. pp. 290-305.
9. Hai-Feng Guo. *High Performance Logic Programming*. New Mexico State University. Ph.D. thesis. Oct. 2000.
10. H-F Guo, G. Gupta. *A Simple Scheme for Implementing Tabled LP Systems*. UT Dallas Tech. Rep 02-01. 2001.
11. J.W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1987.
12. E. Pontelli, G. Gupta. On the Duality between Or-parallelism and And-parallelism. In *EuroPar '95*, Springer LNCS 966, pp. 43-54.
13. C. R. Ramakrishnan, S. Dawson, and D. Warren. Practical Program Analysis Using General Purpose Logic Programming Systems. In *Proc. ACM PLDI*. 1996.
14. Efficient Model Checking using Tabled Resolution. Y.S. Ramakrishnan, et al. In *Proceedings of Computer Aided Verification (CAV'97)*. 1997.
15. P. Rao, I. V. Ramakrishnan, et al. Efficient table access mechanisms for logic programs. Journal of Logic Programming, 38(1):31-54, Jan. 1999.
16. Ricardo Rocha, Fernando M. A. Silva, Vítor Santos Costa. Or-Parallelism within Tabling. PADL 1999: 137-151. Springer LNCS 1551.
17. K. Sagonas and T. Swift. An abstract machine for tabled execution of fixed-order stratified logic programs. ACM TOPLAS, 20(3):586 - 635, May 1998.
18. Hisao Tamaki, T. Sato. OLD Resolution with Tabulation. In *ICLP* 1986. pp. 84-98.
19. T. Swift and David S. Warren. An Abstract Machine for SLG Resolution: Definite Programs. *SLP* 1994: 633-652.
20. K. Villaverde, H-F. Guo, E. Pontelli, G. Gupta. Incremental Stack Splitting and Scheduling in the Or-parallel PALS System. Int'l Conf. on Par. Proc. 2001.
21. Neng-Fa Zhou, et al. Implementation of a Linear Tabling Mechanism. *PADL* 2000: 109-123. Springer LNCS 1753.
22. V. Santos Costa et al. Andorra-I: A Parallel Prolog system that transparently exploits both And- and Or-Parallelism. *Proc. ACM PPoPP*, Apr. '91, pp. 83-93.
23. D. S. Warren. The XWAM: A machine that integrates Prolog and deductive database query evaluation. TR 89/25, SUNY at Stony Brook, 1989.

Fixed-Parameter Complexity of Semantics for Logic Programs

Zbigniew Lonc* and Mirosław Truszczyński**

Department of Computer Science, University of Kentucky
Lexington KY 40506-0046, USA
{lonc, mirek}@cs.engr.uky.edu

Abstract. In the paper we establish the fixed-parameter complexity for several parameterized decision problems involving models, supported models and stable models of logic programs. We also establish the fixed-parameter complexity for variants of these problems resulting from restricting attention to Horn programs and to purely negative programs. Most of the problems considered in the paper have high fixed-parameter complexity. Thus, it is unlikely that fixing bounds on models (supported models, stable models) will lead to fast algorithms to decide the existence of such models.

1 Introduction

In this paper we study the complexity of parameterized decision problems concerning models, supported models and stable models of logic programs. In our investigations, we use the framework of the *fixed-parameter complexity* introduced by Downey and Fellows [DF97]. This framework was previously used to study the problem of the existence of stable models of logic programs in [Tru01]. Our present work extends results obtained there. First, in addition to the class of all finite propositional logic programs, we consider its two important subclasses: the class of Horn programs and the class of purely negative programs. Second, in addition to stable models of logic programs, we also study supported models and arbitrary models.

A decision problem is *parameterized* if its inputs are *pairs* of items. The second item in a pair is referred to as a *parameter*. The problems to decide, given a logic program P and an integer k, whether P has a model, supported model or a stable model, respectively, with *at most* k atoms are examples of parameterized decision problems. These problems are NP-complete. However, fixing k (that is, k is no longer regarded as a part of input) makes each of the problems simpler. They become solvable in polynomial time. The following straightforward algorithm works: for every subset $M \subseteq At(P)$ of cardinality at most k, check whether M is a model, supported model or stable model, respectively, of P. The check can

* On leave from Faculty of Mathematics and Information Science, Warsaw University of Technology.

** The second author was partially supported by the NSF grants CDA-9502645, IRI-9619233 and EPS-9874764.

P. Codognet (Ed.): ICLP 2001, LNCS 2237, pp. 197–211, 2001.

be implemented to run in linear time in the size of the program. Since there are $O(n^k)$ sets to be tested, the overall running time of this algorithm is $O(mn^k)$, where m is the size of the input program P and n is the number of atoms in P.

The problem is that algorithms with running times given by $O(mn^k)$ are not practical even for quite small values of k. The question then arises whether better algorithms can be found, for instance, algorithms whose running-time estimate would be given by a polynomial of the order that *does not depend on* k. Such algorithms, if they existed, could be practical for a wide range of values of k and could find applications in computing stable models of logic programs.

This question is the subject of our work. We also consider similar questions concerning related problems of deciding the existence of models, supported models and stable models of cardinality *exactly* k and *at least* k. We refer to all these problems as *small-bound* problems since k, when fixed, can be regarded as "small". In addition, we study problems of existence of models, supported models and stable models of cardinality at most $|At(P)| - k$, exactly $|At(P)| - k$ and at least $|At(P)| - k$. We refer to these problems as *large-bound* problems, since $|At(P)| - k$, for a fixed k, can be informally thought of as "large".

We address these questions using the framework of fixed-parameter complexity [DF97]. Most of our results are negative. They provide strong evidence that for many parameterized problems considered in the paper there are no algorithms whose running time could be estimated by a polynomial of order independent of k.

Formally, a *parameterized* decision problem is a set $L \subseteq \Sigma^* \times \Sigma^*$, where Σ is a fixed alphabet. By selecting a concrete value $\alpha \in \Sigma^*$ of the parameter, a parameterized decision problem L gives rise to an associated *fixed-parameter* problem $L_\alpha = \{x : (x, \alpha) \in L\}$.

A parameterized problem $L \subseteq \Sigma^* \times \Sigma^*$ is *fixed-parameter tractable* if there exist a constant t, an integer function f and an algorithm A such that A determines whether $(x, y) \in L$ in time $f(|y|)|x|^t$ ($|z|$ stands for the length of a string $z \in \Sigma^*$). We denote the class of fixed-parameter tractable problems by FPT. Clearly, if a parameterized problem L is in FPT, then each of the associated fixed-parameter problems L_y is solvable in polynomial time by an algorithm whose exponent does not depend on the value of the parameter y. Parameterized problems that are not fixed-parameter tractable are called *fixed-parameter intractable*.

To study and compare the complexity of parameterized problems Downey and Fellows proposed the following notion of *fixed-parameter reducibility* (or, simply, *reducibility*).

Definition 1. *A parameterized problem L can be* reduced *to a parameterized problem L' if there exist a constant p, an integer function q, and an algorithm A such that:*

1. *A assigns to each instance (x, y) of L an instance (x', y') of L',*
2. *A runs in time $O(q(|y|)|x|^p)$,*
3. *x' depends upon x and y, and y' depends upon y only,*
4. *$(x, y) \in L$ if and only if $(x', y') \in L'$.*

We will use this notion of reducibility throughout the paper. If for two parameterized problems L_1 and L_2, L_1 can be reduced to L_2 and conversely, we say that L_1 and L_2 are *fixed-parameter equivalent* or, simply, *equivalent*.

Downey and Fellows [DF97] defined a hierarchy of complexity classes called the *W hierarchy*:

$$\mathrm{FPT} \subseteq \mathrm{W}[1] \subseteq \mathrm{W}[2] \subseteq \mathrm{W}[3] \subseteq \ldots \ . \tag{1}$$

The classes W[t] can be described in terms of problems that are complete for them (a problem D is *complete* for a complexity class $\mathcal{E}$ if $D \in \mathcal{E}$ and every problem in this class can be reduced to D). Let us call a Boolean formula t-*normalized* if it is a conjunction-of-disjunctions-of-conjunctions ... of literals, with t being the number of conjunctions-of, disjunctions-of expressions in this definition. For example, 2-normalized formulas are conjunctions of disjunctions of literals. Thus, the class of 2-normalized formulas is precisely the class of CNF formulas. We define the *weighted t-normalized satisfiability problem* as:

WS(t)**:** Given a t-normalized formula Φ and a non-negative integer k, decide whether there is a model of Φ with exactly k atoms (or, alternatively, decide whether there is a satisfying valuation for Φ which assigns the logical value **true** to exactly k atoms).

Downey and Fellows show that for every $t \geq 2$, the problem *WS*(t) is complete for the class W[t]. They also show that a restricted version of the problem *WS*(2):

WS$_2$(2)**:** Given a 2-normalized formula Φ with each clause consisting of at most two literals, and an integer k, decide whether there is a model of Φ with exactly k atoms

is complete for the class W[1]. There is strong evidence suggesting that all the implications in (1) are proper. Thus, proving that a parameterized problem is complete for a class W[t], $t \geq 1$, is a strong indication that the problem is not fixed-parameter tractable.

As we stated earlier, in the paper we study the complexity of parameterized problems related to logic programming. All these problems ask whether an input program P has a model, supported model or a stable model satisfying some cardinality constraints involving another input parameter, an integer k. They can be categorized into two general families: *small-bound* problems and *large-bound* problems. In the formal definitions given below, $\mathcal{C}$ denotes a class of logic programs, $\mathcal{D}$ represents a class of models of interest and Δ stands for one of the three arithmetic relations: "$\leq$", "$=$" and "$\geq$".

$\mathcal{D}_\Delta(\mathcal{C})$**:** Given a logic program P from class $\mathcal{C}$ and an integer k, decide whether P has a model M from class $\mathcal{D}$ such that $|M| \ \Delta \ k$.

$\mathcal{D}'_\Delta(\mathcal{C})$**:** Given a logic program P from class $\mathcal{C}$ and an integer k, decide whether P has a model M from class $\mathcal{D}$ such that $(|At(P)| - k) \ \Delta \ |M|$.

In the paper, we consider three classes of programs: the class of Horn programs $\mathcal{H}$, the class of purely negative programs $\mathcal{N}$, and the class of all programs

$\mathcal{A}$. We also consider three classes of models: the class of all models $\mathcal{M}$, the class of supported models $\mathcal{SP}$ and the class of stable models $\mathcal{ST}$.

Thus, for example, the problem $\mathcal{SP}_{\leq}(\mathcal{N})$ asks whether a purely negative logic program P has a supported model M with no more than k atoms ($|M| \leq k$). The problem $\mathcal{ST}'_{\leq}(\mathcal{A})$ asks whether a logic program P (with no syntactic restrictions) has a stable model M in which at most k atoms are false ($|At(P)| - k \leq |M|$). Similarly, the problem $\mathcal{M}'_{\geq}(\mathcal{H})$ asks whether a Horn program P has a model M in which at least k atoms are false ($|At(P)| - k \geq |M|$).

In the three examples given above and, in general, for all problems $\mathcal{D}_{\Delta}(\mathcal{C})$ and $\mathcal{D}'_{\Delta}(\mathcal{C})$, the input instance consists of a logic program P from the class $\mathcal{C}$ and of an integer k. We will regard these problems as parameterized with k. Fixing k (that is, k is no longer a part of input but an element of the problem description) leads to the fixed-parameter versions of these problems. We will denote them $\mathcal{D}_{\Delta}(\mathcal{C}, k)$ and $\mathcal{D}'_{\Delta}(\mathcal{C}, k)$, respectively.

In the paper, for all but three problems $\mathcal{D}_{\Delta}(\mathcal{C})$ and $\mathcal{D}'_{\Delta}(\mathcal{C})$, we establish their fixed-parameter complexities. Our results are summarized in Tables 1 - 3.

In Table 1, we list the complexities of all problems in which $\Delta =$ "$\geq$". Small-bound problems of this type ask about the existence of models of a program P that contain at least k atoms. Large-bound problems in this group are concerned with the existence of models that contain at most $|At(P)|-k$ atoms (the number of false atoms in these models is at least k). From the point of view of the fixed-parameter complexity, these problems are not very interesting. Several of them remain NP-complete even when k is fixed. In other words, fixing k does not simplify them enough to make them tractable. For this reason, all the entries in Table 1, listing the complexity as NP-complete (denoted by NP-c in the table), refer to fixed-parameter versions $\mathcal{D}_{\geq}(\mathcal{C}, k)$ and $\mathcal{D}'_{\geq}(\mathcal{C}, k)$ of problems $\mathcal{D}_{\geq}(\mathcal{C})$ and $\mathcal{D}'_{\geq}(\mathcal{C})$. The problem $\mathcal{M}'_{\geq}(\mathcal{A}, k)$ is NP-complete for every fixed $k \geq 1$. All other fixed-parameter problems in Table 1 that are marked NP-complete are NP-complete for every value $k \geq 0$.

On the other hand, many problems $\mathcal{D}_{\geq}(\mathcal{C})$ and $\mathcal{D}'_{\geq}(\mathcal{C})$ are "easy". They are fixed-parameter tractable in a strong sense. They can be solved in polynomial time even *without* fixing k. This is indicated by marking the corresponding entries in Table 1 with P (for the class P) rather than with FPT. There is only one exception, the problem $\mathcal{M}'_{\geq}(\mathcal{N})$, which is W[1]-complete.

Table 1. The complexities of the problems $\mathcal{D}_{\geq}(\mathcal{C})$ and $\mathcal{D}'_{\geq}(\mathcal{C})$.

	$\mathcal{H}$	$\mathcal{N}$	$\mathcal{A}$
$\mathcal{M}$	P	P	P
$\mathcal{M}'$	P	W[1]-c	NP-c
$\mathcal{SP}$	P	NP-c	NP-c
$\mathcal{SP}'$	P	NP-c	NP-c
$\mathcal{ST}$	P	NP-c	NP-c
$\mathcal{ST}'$	P	NP-c	NP-c

Small-bound problems for the cases when $\Delta =$ "$=$" or "$\leq$" can be viewed as problems of deciding the existence of "small" models (that is, models containing exactly k or at most k atoms). The fixed-parameter complexities of these problems are summarized in Table 2.

The problems involving the class of all purely negative programs and the class of all programs are W[2]-complete. This is a strong indication that they are fixed-parameter intractable. All problems of the form $\mathcal{D}_{\leq}(\mathcal{H})$ are fixed-parameter tractable. In fact, they are solvable in polynomial time even without fixing the parameter k. We indicate this by marking the corresponding entries with P. Similarly, the problem $\mathcal{ST}_{=}(\mathcal{H})$ of deciding whether a Horn logic program P has a stable model of size exactly k is in P. However, perhaps somewhat surprisingly, the remaining two problems involving Horn logic programs and $\Delta =$ "$=$" are harder. We proved that the problem $\mathcal{M}_{=}(\mathcal{H})$ is W[1]-complete and that the problem $\mathcal{SP}_{=}(\mathcal{H})$ is W[1]-hard. Thus, they most likely are not fixed-parameter tractable. We also showed that the problem $\mathcal{SP}_{=}(\mathcal{H})$ is in the class W[2]. The exact fixed-parameter complexity of $\mathcal{SP}_{=}(\mathcal{H})$ remains unresolved.

Large-bound problems for the cases when $\Delta =$ "$=$" or "$\leq$" can be viewed as problems of deciding the existence of "large" models, that is, models with a small number of false atoms — equal to k or less than or equal to k. The fixed-parameter complexities of these problems are summarized in Table 3.

The problems specified by $\Delta =$ "$\leq$" and concerning the existence of models are in P. Similarly, the problems specified by $\Delta =$ "$\leq$" and involving Horn programs are solvable in polynomial time. Lastly, the problem $\mathcal{ST}'_{=}(\mathcal{H})$ is in P, as well. These problems are in P even without fixing k and eliminating it from input. All other problems in this group have higher complexity and, in all likelihood, are fixed-parameter intractable. One of the problems, $\mathcal{M}'_{=}(\mathcal{N})$, is

Table 2. The complexities of the problem of computing small models (small-bound problems, the cases of $\Delta =$ "$=$" and "$\leq$").

	$\mathcal{H}_{\leq}$	$\mathcal{H}_{=}$	$\mathcal{N}_{\leq}$	$\mathcal{N}_{=}$	$\mathcal{A}_{\leq}$	$\mathcal{A}_{=}$
$\mathcal{M}$	P	W[1]-c	W[2]-c	W[2]-c	W[2]-c	W[2]-c
$\mathcal{SP}$	P	W[1]-h, in W[2]	W[2]-c	W[2]-c	W[2]-c	W[2]-c
$\mathcal{ST}$	P	P	W[2]-c	W[2]-c	W[2]-c	W[2]-c

Table 3. The complexities of the problems of computing large models (large-bound problems, the cases of $\Delta =$ "$=$" and "$\leq$").

	$\mathcal{H}_{\leq}$	$\mathcal{H}_{=}$	$\mathcal{N}_{\leq}$	$\mathcal{N}_{=}$	$\mathcal{A}_{\leq}$	$\mathcal{A}_{=}$
$\mathcal{M}'$	P	W[2]-c	P	W[1]-c	P	W[2]-c
$\mathcal{SP}'$	P	W[3]-c,	W[2]-c	W[2]-c	W[3]-c	W[3]-c
$\mathcal{ST}'$	P	P	W[2]-c	W[2]-c	W[3]-h	W[3]-h

W[1]-complete. Most of the remaining problems are W[2]-complete. Surprisingly, some problems are even harder. Three problems concerning supported models are W[3]-complete. For two problems involving stable models, $\mathcal{ST}'_{=}(\mathcal{A})$ and $\mathcal{ST}'_{\leq}(\mathcal{A})$, we could only prove that they are W[3]-hard. For these two problems we did not succeed in establishing any upper bound on their fixed-parameter complexities.

The study of fixed-parameter tractability of problems occurring in the area of nonmonotonic reasoning is a relatively new research topic. The only two other papers we are aware of are [Tru01] and [GSS99]. The first of these two papers provided a direct motivation for our work here (we discussed it earlier). In the second one, the authors focused on parameters describing *structural* properties of programs. They showed that under some choices of the parameters decision problems for nonmonotonic reasoning become fixed-parameter tractable.

Our results concerning computing stable and supported models for logic programs are mostly negative. Parameterizing basic decision problems by constraining the size of models of interest does not lead (in most cases) to fixed-parameter tractability.

There are, however, several interesting aspects to our work. First, we identified some problems that are W[3]-complete or W[3]-hard. Relatively few problems from these classes were known up to now [DF97]. Second, in the context of the polynomial hierarchy, there is no distinction between the problem of existence of models of specified sizes of clausal propositional theories and similar problems concerning models, supported models and stable models of logic programs. All these problems are NP-complete. However, when we look at the complexity of these problems in a more detailed way, from the perspective of fixed-parameter complexity, the equivalence is lost. Some problems are W[3]-hard, while problems concerning existence of models of 2-normalized formulas are W[2]-complete or easier. Third, our results show that in the context of fixed-parameter tractability, several problems involving models and supported models are hard even for the class of Horn programs. Finally, our work leaves three problems unresolved. While we obtained some bounds for the problems $\mathcal{SP}_{=}(\mathcal{H})$, $\mathcal{ST}'_{\leq}(\mathcal{A})$ and $\mathcal{ST}'_{=}(\mathcal{A})$, we did not succeed in establishing their precise fixed-parameter complexities.

The rest of our paper is organized as follows. In the next section, we review relevant concepts in logic programming. After that, we present several useful fixed-parameter complexity results for problems of the existence of models for propositional theories of certain special types. In the last section we give proofs of some of our complexity results.

2 Preliminaries

In the paper, we consider only the propositional case. A logic program *clause* (or *rule*) is any expression r of the form

$$r = \quad p \leftarrow q_1, \ldots, q_m, \mathbf{not}(s_1), \ldots, \mathbf{not}(s_n), \tag{2}$$

where p, q_i and s_i are propositional atoms. We call the atom p the *head* of r and we denote it by $h(r)$. Further, we call the set of atoms $\{q_1, \ldots, q_m, s_1, \ldots, s_n\}$

the *body* of r and we denote it by $b(r)$. We distinguish the *positive body* of r, $\{q_1, \ldots, q_m\}$ ($b^+(r)$, in symbols), and the *negative body* of r, $\{s_1, \ldots, s_n\}$ ($b^-(r)$, in symbols).

A *logic program* is a collection of clauses. For a logic program P, by $At(P)$ we denote the set of atoms that appear in P. If every clause in a logic program P has an empty negative body, we call P a *Horn* program. If every clause in P has an empty positive body, we call P a *purely negative* program.

A clause r, given by (2), has a *propositional interpretation* as an implication

$$pr(r) = \quad q_1 \wedge \ldots \wedge q_m \wedge \neg s_1 \wedge \ldots \wedge \neg s_n \Rightarrow p.$$

Given a logic program P, by a *propositional interpretation* of P we mean the propositional formula

$$pr(P) = \bigwedge \{pr(r) \colon r \in P\}.$$

We say that a set of atoms M is a *model* of a clause (2) if M is a (propositional) model of the clause $pr(r)$. As usual, atoms in M are interpreted as true, all other atoms are interpreted as false. A set of atoms $M \subseteq At(P)$ is a *model* of a program P if it is a model of the formula $pr(P)$. We emphasize the requirement $M \subseteq At(P)$. In this paper, given a program P, we are interested only in the truth values of atoms that actually occur in P.

It is well known that every Horn program P has a least model (with respect to set inclusion). We will denote this model by $lm(P)$.

Let P be a logic program. Following [Cla78], for every atom $p \in At(P)$ we define a propositional formula $comp(p)$ by

$$comp(p) = \quad p \Leftrightarrow \bigvee \{c(r) \colon r \in P,\ h(r) = p\},$$

where

$$c(r) = \bigwedge \{q \colon q \in b^+(r)\} \wedge \bigwedge \{\neg s \colon s \in b^-(r)\}.$$

If for an atom $p \in At(P)$ there are no rules with p in the head, we get an empty disjunction in the definition of $comp(p)$, which we interpret as a contradiction.

We define the *program completion* [Cla78] of P as the propositional theory

$$comp(P) = \{comp(p) \colon p \in At(P)\}.$$

A set of atoms $M \subseteq At(P)$ is a *supported model* of P if it is a model of the completion of P. It is easy to see that if p does not appear as the head of a rule in P, p is false in every supported model of P. It is also easy to see that each supported model of a program P is a model of P (the converse is not true in general).

Given a logic program P and a set of atoms M, we define the *reduct* of P with respect to M (P^M, in symbols) to be the logic program obtained from P by

1. removing from P each clause r such that $M \cap b^-(r) \neq \emptyset$ (we call such clauses *blocked by* M),

2. removing all negated atoms from the bodies of all the rules that remain (that is, those rules that are not blocked by M).

The reduct P^M is a Horn program. Thus, it has a least model. We say that M is a *stable model* of P if $M = lm(P^M)$. Both the notion of the reduct and that of a stable model were introduced in [GL88].

It is known that every stable model of a program P is a supported model of P. The converse does not hold in general. However, if a program P is purely negative, then stable and supported models of P coincide [Fag94].

In our arguments we use fixed-parameter complexity results on problems to decide the existence of models of prescribed sizes for propositional formulas from some special classes. To describe these problems we introduce additional terminology. First, given a propositional theory Φ, by $At(\Phi)$ we denote the set of atoms occurring in Φ. As in the case of logic programming, we consider as models of a propositional theory Φ only those sets of atoms that are subsets of $At(\Phi)$. Next, we define the following classes of formulas:

tN: the class of t-normalized formulas (if $t = 2$, these are simply CNF formulas)
2N$_3$: the class of all 2-normalized formulas whose every clause is a disjunction of at most three literals (clearly, *2N*$_3$ is a subclass of the class *2N*)
tNM: the class of *monotone* t-normalized formulas, that is, t-normalized formulas in which there are no occurrences of the negation operator
tNA: the class of *antimonotone* t-normalized formulas, that is, t-normalized formulas in which every atom is directly preceded by the negation operator.

Finally, we extend the notation $\mathcal{M}_\Delta(\mathcal{C})$ and $\mathcal{M}'_\Delta(\mathcal{C})$, to the case when $\mathcal{C}$ stands for a class of propositional formulas. In this terminology, $\mathcal{M}'_=(3NM)$ denotes the problem to decide whether a monotone 3-normalized formula Φ has a model in which exactly k atoms are false. Similarly, $\mathcal{M}_=(tN)$ is simply another notation for the problem $WS[t]$ that we discussed above. The following theorem establishes several complexity results that we will use later in the paper.

Theorem 1. *(i) The problems $\mathcal{M}_=(2N)$ and $\mathcal{M}_=(2NM)$ are* W[2]*-complete.*
(ii) The problems $\mathcal{M}_=(2N_3)$ and $\mathcal{M}_=(2NA)$ are W[1]*-complete.*
(iii) The problem $\mathcal{M}'_\leq(3N)$ is W[3]*-complete.*

Proof: The statements (i) and (ii) are proved in [DF97]. To prove the statement (iii), we use the fact that the problem $\mathcal{M}_\leq(3N)$ is $W[3]$-complete [DF97]. We reduce $\mathcal{M}_\leq(3N)$ to $\mathcal{M}'_\leq(3N)$ and conversely. Let us consider a 3-normalized formula $\Phi = \bigwedge_{i=1}^{m} \bigvee_{j=1}^{m_i} \bigwedge_{\ell=1}^{m_{ij}} x[i,j,\ell]$, where $x[i,j,\ell]$ are literals. We observe that Φ has a model of cardinality at most k if and only if a related formula $\bar{\Phi} = \bigwedge_{i=1}^{m} \bigvee_{j=1}^{m_i} \bigwedge_{\ell=1}^{m_{ij}} \bar{x}[i,j,\ell]$, obtained from Φ by replacing every negative literal $\neg x$ by a new atom $\bar{x}$ and every positive literal x by a negated atom $\neg\bar{x}$, has a model of cardinality at least $|At(\bar{\Phi})| - k$. This construction defines a reduction of $\mathcal{M}_\leq(3N)$ to $\mathcal{M}'_\leq(3N)$. It is easy to see that this reduction satisfies all the requirements of the definition of fixed-parameter reducibility.

A reduction of $\mathcal{M}'_\leq(3N)$ to $\mathcal{M}_\leq(3N)$ can be constructed in a similar way. Since the problem $\mathcal{M}_\leq(3N)$ is W[3]-complete, so is the problem $\mathcal{M}'_\leq(3N)$. □

In the proof of part (iii) of Theorem 1, we observed that the reduction we described there satisfies all the requirements specified in Definition 1 of fixed-parameter reducibility. Throughout the paper we prove our complexity results by constructing reductions from one problem to another. In most cases, we only verify the condition (4) of the definition which, usually, is the only non-trivial part of the proof. Checking that the remaining conditions hold is straightforward and we leave these details out.

3 Some Proofs

In this section we present some typical proofs of fixed-parameter complexity results for problems involving existence of models, supported models and stable models of logic programs. Our goal is to introduce key proof techniques that we used when proving the results discussed in the introduction.

Theorem 2. *The problems $\mathcal{M}'_{=}(\mathcal{H})$ and $\mathcal{M}'_{=}(\mathcal{A})$ are* W[2]*-complete.*

Proof: Both problems are clearly in W[2] (models of a logic program P are models of the corresponding 2-normalized formula $pr(P)$). Since $\mathcal{H} \subseteq \mathcal{A}$, to complete the proof it is enough to show that the problem $\mathcal{M}'_{=}(\mathcal{H})$ is W[2]-hard. To this end, we will reduce the problem $\mathcal{M}_{=}(2NM)$ to $\mathcal{M}'_{=}(\mathcal{H})$.

Let Φ be a monotone 2-normalized formula and let $k \geq 0$. Let $\{x_1, \ldots, x_n\}$ be the set of atoms of Φ. We define a Horn program P_Φ corresponding to Φ as follows. We choose an atom a not occurring in Φ and include in P_Φ all rules of the form $x_i \leftarrow a$, $i = 1, 2, \ldots, n$. Next, for each clause $C = x_{i_1} \vee \ldots \vee x_{i_p}$ of Φ we include in P_Φ the rule

$$r_C = \quad a \leftarrow x_{i_1}, \ldots, x_{i_p}.$$

We will show that Φ has a model of size k if and only if P_Φ has a model of size $|At(P_\Phi)| - (k+1) = (n+1) - (k+1) = n - k$.

Let M be a model of Φ of size k. We define $M' = \{x_1, \ldots, x_n\} \setminus M$. The set M' has $n - k$ elements. Let us consider any clause $r_C \in P_\Phi$ of the form given above. Since M satisfies C, there is j, $1 \leq j \leq p$, such that $x_{i_j} \notin M'$. Thus, M' is a model of r_C. Since $a \notin M'$, M' satisfies all clauses $x_i \leftarrow a$. Hence, M' is a model of P_Φ.

Conversely, let M' be a model of P_Φ of size exactly $n - k$. If $a \in M'$ then $x_i \in M'$, for every i, $1 \leq i \leq n$. Thus, $|M'| = n + 1 > n - k$, a contradiction. Consequently, we obtain that $a \notin M'$. Let $M = \{x_1, \ldots, x_n\} \setminus M'$. Since $a \notin M'$, $|M| = k$. Moreover, M satisfies all clauses in Φ. Indeed, let us assume that there is a clause C such that no atom of C is in M. Then, all atoms of C are in M'. Since M' satisfies r_C, $a \in M'$, a contradiction. Now, the assertion follows by Theorem 1. □

Theorem 3. *The problem $\mathcal{M}_{=}(\mathcal{H})$ is* W[1]*-complete.*

Proof: We will first prove the hardness part. To this end, we will reduce the problem $\mathcal{M}_=(2NA)$ to the problem $\mathcal{M}_=(\mathcal{H})$. Let Φ be an antimonotone 2-normalized formula and let k be a non-negative integer. Let $a_0, \ldots, a_k$ be $k+1$ different atoms not occurring in Φ. For each clause $C = \neg x_1 \vee \ldots \vee \neg x_p$ of Φ we define a logic program rule r_C by

$$r_C = \quad a_0 \leftarrow x_1, \ldots, x_p.$$

We then define P_Φ by

$$P_\Phi = \{r_C : C \in \Phi\} \cup \{a_i \leftarrow a_j : i, j = 0, 1, \ldots, k,\ i \neq j\}.$$

Let us assume that M is a model of size k of the program P_Φ. If for some i, $0 \leq i \leq k$, $a_i \in M$ then $\{a_0, \ldots, a_k\} \subseteq M$ and, consequently, $|M| > k$, a contradiction. Thus, M does not contain any of the atoms a_i. Since M satisfies all rules r_C and since it consists of atoms of Φ only, M is a model of Φ (indeed, the body of each rule r_C must be false so, consequently, each clause C must be true). Similarly, one can show that if M is a model of Φ then it is a model of P_Φ. Thus, W[1]-hardness follows by Theorem 1.

To prove that the problem $\mathcal{M}_=(\mathcal{H})$ is in the class W[1], we will reduce it to the problem $\mathcal{M}_=(2N_3)$. To this end, for every Horn program P we will describe a 2-normalized formula Φ_P, with each clause consisting of no more than three literals, and such that P has a model of size k if and only if Φ_P has a model of size $(k+1)2^k + k$. Moreover, we will show that Φ_P can be constructed in time bounded by a polynomial in the size of P (with the degree not depending on k).

First, let us observe that without loss of generality we may restrict our attention to Horn programs whose rules do not contain multiple occurrences of the same atom in the body. Such occurrences can be eliminated in time linear in the size of the program. Next, let us note that under this restriction, a Horn program P has a model of size k if and only if the program P', obtained from P by removing all clauses with bodies consisting of more than k atoms, has a model of size k. The program P' can be constructed in time linear in the size of P and k.

Thus, we will describe the construction of the formula Φ_P only for Horn programs P in which the body of every rule consists of no more than k atoms. Let P be such a program. We define

$$\mathcal{B} = \{B:\ B \subseteq b(r),\ \text{for some } r \in P\}.$$

For every set $B \in \mathcal{B}$ we introduce a new variable $u[B]$. Further, for every atom x in P we introduce 2^k new atoms $x[i]$, $i = 1, \ldots, 2^k$.

We will now define several families of formulas. First, for every $x \in At(P)$ and $i = 1, \ldots, 2^k$ we define

$$D(x, i) = \quad x \Leftrightarrow x[i] \quad (\text{or } (\neg x \vee x[i]) \wedge (x \vee \neg x[i])),$$

and, for each set $B \in \mathcal{B}$ and for each $x \in B$, we define

$$E(B, x) = \quad x \wedge u[B \setminus \{x\}] \Rightarrow u[B] \quad (\text{or } \neg x \vee \neg u[B \setminus \{x\}] \vee u[B]).$$

Next, for each set $B \in \mathcal{B}$ and for each $x \in B$ we define

$$F(B, x) = \quad u[B] \Rightarrow x \quad (\text{or } \neg u[B] \vee x).$$

Finally, for each rule r in P we introduce a formula

$$G(r) = \quad u[b(r)] \Rightarrow h(r) \quad (\text{or } \neg u[b(r)] \vee h(r)).$$

We define Φ_P to be the conjunction of all these formulas (more precisely, of their 2-normalized representations given in the parentheses) and of the formula $u[\emptyset]$. Clearly, Φ_P is a formula from the class $2N_3$. Further, since the body of each rule in P has at most k elements, the set $\mathcal{B}$ has no more than $|P|2^k$ elements, each of them of size at most k ($|P|$ denotes the cardinality of P, that is, the number of rules in P). Thus, Φ_P can be constructed in time bounded by a polynomial in the size of P, whose degree does not depend on k.

Let us consider a model M of P such that $|M| = k$. We define

$$M' = M \cup \{x[i] \colon x \in M, i = 1, \ldots, 2^k\} \cup \{u[B] \colon B \subseteq M\}.$$

The set M' satisfies all formulas $D(x, i)$, $x \in At(P)$, $i = 1, \ldots, 2^k$. In addition, the formula $u[\emptyset]$ is also satisfied by M' ($\emptyset \subseteq M$ and so, $u[\emptyset] \in M'$).

Let us consider a formula $E(B, x)$, for some $B \in \mathcal{B}$ and $x \in B$. Let us assume that $x \wedge u[B \setminus \{x\}]$ is true in M'. Then, $x \in M'$ and, since $x \in At(P)$, $x \in M$. Moreover, since $u[B \setminus \{x\}] \in M'$, $B \setminus \{x\} \subseteq M$. It follows that $B \subseteq M$ and, consequently, that $u[B] \in M'$. Thus, M' satisfies all "E-formulas" in Φ_P.

Next, let us consider a formula $F(B, x)$, where $B \in \mathcal{B}$ and $x \in B$, and let us assume that M' satisfies $u[B]$. It follows that $B \subseteq M$. Consequently, $x \in M$. Since $M \subseteq M'$, M' satisfies x and so, M' satisfies $F(B, x)$.

Lastly, let us look at a formula $G(r)$, where $r \in P$. Let us assume that $u[b(r)] \in M'$. Then, $b(r) \subseteq M$. Since r is a Horn clause and since M is a model of P, it follows that $h(r) \in M$. Consequently, $h(r) \in M'$. Thus, M' is a model of $G(r)$.

We proved that M' is a model of Φ_P. Moreover, it is easy to see that $|M'| = k + k2^k + 2^k = (k + 1)2^k + k$.

Conversely, let us assume that M' is a model of Φ_P and that $|M'| = (k + 1)2^k + k$. We set $M = M' \cap At(P)$. First, we will show that M is a model of P.

Let us consider an arbitrary clause $r \in P$, say

$$r = h \leftarrow b_1, \ldots, b_p,$$

where h and b_i, $1 \leq i \leq p$, are atoms. Let us assume that $\{b_1, \ldots, b_p\} \subseteq M$. We need to show that $h \in M$.

Since $\{b_1, \ldots, b_p\} = b(r)$, the set $\{b_1, \ldots, b_p\}$ and all its subsets belong to $\mathcal{B}$. Thus, Φ_P contains formulas

$$E(\{b_1, \ldots, b_{i-1}\}, b_i) = b_i \wedge u[\{b_1, \ldots, b_{i-1}\}] \Rightarrow u[\{b_1, \ldots, b_{i-1}, b_i\}],$$

where $i = 1, \ldots, p$. All these formulas are satisfied by M'. We also have $u[\emptyset] \in \Phi_P$. Consequently, $u[\emptyset]$ is satisfied by M', as well. Since all atoms b_i, $1 \leq i \leq p$, are

also satisfied by M' (since $M \subseteq M'$), it follows that $u[\{b_1, \ldots, b_p\}]$ is satisfied by M'.

The formula $G(r) = u[\{b_1, \ldots, b_p\}] \Rightarrow h$ belongs to Φ_P. Thus, it is satisfied by M'. It follows that $h \in M'$. Since $h \in At(P)$, $h \in M$. Thus, M is a model of r and, consequently, of the program P.

To complete the proof we have to show that $|M| = k$. Since M' is a model of Φ_P, for every $x \in M$, M' contains all atoms $x[i]$, $1 \leq i \leq 2^k$. Hence, if $|M| > k$ then $|M'| \geq |M| + |M| \times 2^k \geq (k+1)(1+2^k) > (k+1)2^k + k$, a contradiction.

So, we will assume that $|M| < k$. Let us consider an atom $u[B]$, where $B \in \mathcal{B}$, such that $u[B] \in M'$. For every $x \in B$, Φ_P contains the rule $F(B, x)$. The set M' is a model of $F(B, x)$. Thus, $x \in M'$ and, since $x \in At(P)$, we have that $x \in M$. It follows that $B \subseteq M$. It is now easy to see that the number of atoms of the form $u[B]$ that are true in M' is smaller than 2^k. Thus, $|M'| < |M| + |M| \times 2^k + 2^k \leq (k-1)(1+2^k)+2^k < (k+1)2^k+k$, again a contradiction. Consequently, $|M| = k$.

It follows that the problem $\mathcal{M}_=(\mathcal{H})$ can be reduced to the problem $\mathcal{M}_=(2N_3)$. Thus, by Theorem 1, the problem $\mathcal{M}_=(\mathcal{H})$ is in the class W[1]. This completes our argument. □

Theorem 4. *The problem $\mathcal{SP}_=(\mathcal{A})$ is in $W[2]$.*

Proof: We will show a reduction of $\mathcal{SP}_=(\mathcal{A})$ to $\mathcal{M}_=(2N)$, which is in $W[2]$ by Theorem 1. Let P be a logic program with atoms $x_1, \ldots, x_n$. We can identify supported models of P with models of its completion $comp(P)$. The completion is of the form $comp(P) = \Phi_1 \wedge \ldots \wedge \Phi_n$, where

$$\Phi_i = x_i \Leftrightarrow \bigvee_{j=1}^{m_i} \bigwedge_{\ell=1}^{m_{ij}} x[i, j, \ell],$$

$i = 1, \ldots, n$, and $x[i, j, \ell]$ are literals. It can be constructed in linear time in the size of the program P.

We will use $comp(P)$ to define a formula Φ_P. The atoms of Φ_P are $x_1, \ldots, x_n$ and $u[i, j]$, $i = 1, \ldots, n$, $j = 1, \ldots, m_i$. For $i = 1, \ldots, n$, let

$$G_i = x_i \Rightarrow \bigvee_{j=1}^{m_i} u[i, j], \qquad (\text{or } \neg x_i \vee \bigvee_{j=1}^{m_i} u[i, j]),$$

$$G'_i = \bigvee_{j=1}^{m_i} u[i, j] \Rightarrow x_i, \qquad (\text{or } \bigwedge_{j=1}^{m_i} (x_i \vee \neg u[i, j])),$$

$$H_i = \bigwedge_{j=1}^{m_i - 1} \bigwedge_{j'=j+1}^{m_i} (\neg u[i, j] \vee \neg u[i, j']), \quad \text{for every } i \text{ such that } m_i \geq 2,$$

$$I_i = \bigwedge_{j=1}^{m_i} (u[i, j] \Rightarrow \bigwedge_{\ell=1}^{m_{ij}} x[i, j, \ell]) \qquad (\text{or } \bigwedge_{j=1}^{m_i} \bigwedge_{\ell=1}^{m_{ij}} (\neg u[i, j] \vee x[i, j, \ell])),$$

$$J_i = \bigvee_{j=1}^{m_i} \bigwedge_{\ell=1}^{m_{ij}} x[i,j,\ell] \Rightarrow x_i, \qquad (\text{or } \bigwedge_{j=1}^{m_i} (x_i \vee \bigvee_{\ell=1}^{m_{ij}} \neg x[i,j,\ell])).$$

The formula Φ_P is a conjunction of the formulas written above (of the formulas given in the parentheses, to be precise). Clearly, Φ_P is a 2-normalized formula. We will show that $comp(P)$ has a model of size k (or equivalently, that P has a supported model of size k) if and only if Φ_P has a model of size $2k$.

Let $M = \{x_{p_1}, \ldots, x_{p_k}\}$ be a model of $comp(P)$. Then, for each $i = p_1, \ldots, p_k$, there is j, $1 \leq j \leq m_i$, such that M is a model of $\bigwedge_{\ell=1}^{m_{ij}} x[i,j,\ell]$ (this is because M is a model of every formula Φ_i). We denote one such j (an arbitrary one) by j_i. We claim that

$$M' = M \cup \{u[i,j_i] : \ i = p_1, \ldots, p_k\}$$

is a model of Φ_P. Clearly, G_i is true in M' for every i, $1 \leq i \leq n$. If $x_i \notin M$ then $u[i,j] \notin M'$ for all $j = 1, \ldots, m_i$. Thus, G'_i is satisfied by M'. Since for each i, $1 \leq i \leq n$, there is at most one j such that $u[i,j] \in M'$, it follows that every formula H_i is true in M'. By the definition of j_i, if $u[i,j] \in M'$ then $j = j_i$ and M' is a model of $\bigwedge_{\ell=1}^{m_{ij}} x[i,j,\ell]$. Hence, I_i is satisfied by M'. Finally, all formulas J_i, $1 \leq i \leq n$, are clearly true in M'. Thus, M' is a model of Φ_P of size $2k$.

Conversely, let M' be a model of Φ_P such that $|M'| = 2k$. Let us assume that M' contains exactly s atoms $u[i,j]$. The clauses H_i ensure that for each i, M' contains at most one atom $u[i,j]$. Therefore, the set $M' \cap \{u[i,j]: i = 1, \ldots, n\ j = 1, \ldots, m_i\}$ is of the form $\{u[p_1,j_{p_1}], \ldots, u[p_s,j_{p_s}]\}$ where $p_1 < \ldots < p_s$.

Since the conjunction of G_i and G'_i is equivalent to $x_i \Leftrightarrow \bigvee_{j=1}^{m_i} u[i,j]$, it follows that exactly s atoms x_i belong to M'. Thus, $|M'| = 2s = 2k$ and $s = k$. It is now easy to see that M' is of the form $\{x_{p_1}, \ldots, x_{p_k}, u[p_1,j_{p_1}], \ldots, u[p_k,j_{p_k}]\}$.

We will now prove that for every i, $1 \leq i \leq n$, the implication

$$x_i \Rightarrow \bigvee_{j=1}^{m_i} \bigwedge_{\ell=1}^{m_{ij}} x[i,j,\ell]$$

is true in M'. To this end, let us assume that x_i is true in M'. Then, there is j, $1 \leq j \leq m_i$, such that $u[i,j] \in M'$ (in fact, $i = p_t$ and $j = j_{p_t}$, for some t, $1 \leq t \leq k$). Since the formula I_i is true in M', the formula $\bigwedge_{\ell=1}^{m_{ij}} x[i,j,\ell]$ is true in M'. Thus, the formula $\bigvee_{j=1}^{m_i} \bigwedge_{\ell=1}^{m_{ij}} x[i,j,\ell]$ is true in M', too.

Since for every i, $1 \leq i \leq n$, the formula J_i is true in M', it follows that all formulas Φ_i are true in M'. Since the only atoms of M' that appear in the formulas Φ_i are the atoms $x_{p_1} \ldots, x_{p_k}$, it follows that $M = \{x_{p_1} \ldots, x_{p_k}\}$ is a model of $comp(P) = \Phi_1 \wedge \ldots \wedge \Phi_n$.

Thus, the problem $\mathcal{SP}_=(\mathcal{A})$ can be reduced to the problem $\mathcal{M}_=(2N)$, which completes the proof. □

For the problem $\mathcal{SP}_=(\mathcal{A})$ we also established the hardness result — we proved that it is W[2]-hard (we omit the proof due to space restrictions). Thus, we found the exact location of this problem in the W-hierarchy. For the problem $\mathcal{ST}'_{\leq}(\mathcal{A})$, that we are about to consider now, we only succeeded in establishing the lower

bound on its complexity. We proved it to be W[3]-hard. We did not succeed in obtaining any non-trivial upper estimate on its complexity.

Theorem 5. *The problem $\mathcal{ST}'_{\leq}(\mathcal{A})$ is* W[3]*-hard.*

Proof: We will reduce the problem $\mathcal{M}'_{\leq}(3N)$ to the problem $\mathcal{ST}'_{\leq}(\mathcal{A})$. Let

$$\Phi = \bigwedge_{i=1}^{m} \bigvee_{j=1}^{m_i} \bigwedge_{\ell=1}^{m_{ij}} x[i,j,\ell]$$

be a 3-normalized formula, where $x[i,j,\ell]$ are literals. Let $u[1],\ldots,u[m], v[1],\ldots, v[2k+1]$ be new atoms not occurring in Φ. For each atom $x \in At(\Phi)$, we introduce new atoms $x[s]$, $s = 1,\ldots,k$.

Let P_Φ be a logic program with the following rules:

$$A(x,y,s) = x[s] \leftarrow \mathbf{not}(y[s]), \quad x,y \in At(\Phi),\ x \neq y,\ s = 1,\ldots,k,$$

$$B(x) = x \leftarrow x[1], x[2],\ldots,x[k], \quad x \in At(\Phi),$$

$$C(i,j) = u[i] \leftarrow x'[i,j,1], x'[i,j,2],\ldots,x'[i,j,m_{ij}], \quad i = 1,\ldots,m,\ j = 1,\ldots,m_i,$$

where

$$x'[i,j,\ell] = \begin{cases} x & \text{if } x[i,j,\ell] = x \\ \mathbf{not}(x) & \text{if } x[i,j,\ell] = \neg x, \end{cases}$$

$$D(q) = v[q] \leftarrow u[1], u[2],\ldots,u[m], \quad q = 1,\ldots,2k+1.$$

Clearly, $|At(P_\Phi)| = nk + n + m + 2k + 1$, where $n = |At(\Phi)|$. We will show that Φ has a model of cardinality at least $n - k$ if and only if P_Φ has a stable model of cardinality at least $|At(P_\Phi)| - 2k = n(k+1) + m + 1$.

Let $M = At(\Phi) \setminus \{x_1,\ldots,x_k\}$ be a model of Φ, where $x_1,\ldots,x_k$ are some atoms from $At(\Phi)$ that are not necessarily distinct. We claim that $M' = At(P_\Phi) \setminus \{x_1,\ldots,x_k, x_1[1],\ldots,x_k[k]\}$ is a stable model of P_Φ.

Let us notice that a rule $A(x,y,s)$ is not blocked by M' if and only if $y = x_s$. Hence, the program $P_\Phi^{M'}$ consists of the rules:

$$x[1] \leftarrow \quad , \text{ for } x \neq x_1,$$

$$x[2] \leftarrow \quad , \text{ for } x \neq x_2$$

$$\ldots$$

$$x[k] \leftarrow \quad , \text{ for } x \neq x_k$$

$$x \leftarrow x[1], x[2],\ldots,x[k], \quad x \in At(\Phi)$$

$$v[q] \leftarrow u[1], u[2],\ldots,u[m], \quad q = 1,\ldots,2k+1,$$

and of some of the rules with heads $u[i]$. Let us suppose that every rule of P_Φ with head $u[i]$ contains either a negated atom $x \in M$ or a non-negated atom $x \notin M$. Then, for every $j = 1,\ldots,m_i$ there exists ℓ, $1 \leq \ell \leq m_{ij}$ such that either $x[i,j,\ell] = \neg x$ and $x \in M$, or $x[i,j,\ell] = x$ and $x \notin M$. Thus, M is not

a model of the formula $\bigvee_{j=1}^{m_i} \bigwedge_{\ell=1}^{m_{ij}} x[i,j,\ell]$ and, consequently, M is not a model of Φ, a contradiction. Hence, for every $i = 1, \ldots, m$, there is a rule with head $u[i]$ containing neither a negated atom $x \in M$ nor a non-negated atom $x \notin M$. These rules also contribute to the reduct $P_\Phi^{M'}$.

All atoms $x[s] \neq x_1[1], x_2[2], \ldots, x_k[k]$ are facts in $P_\Phi^{M'}$. Thus, they belong to $lm(P_\Phi^{M'})$. Conversely, if $x[s] \in lm(P_\Phi^{M'})$ then $x[s] \neq x_1[1], x_2[2], \ldots, x_k[k]$. Moreover, it is evident by rules $B(x)$ that $x \in lm(P_\Phi^{M'})$ if and only if $x \neq x_1, x_2, \ldots, x_k$. Hence, by the observations in the previous paragraph, $u[i] \in lm(P_\Phi^{M'})$, for each $i = 1, \ldots, m$. Finally, $v[q] \in lm(P_\Phi^{M'})$, $q = 1, \ldots 2k+1$, because the rules $D(q)$ belong to the reduct $P_\Phi^{M'}$. Hence, $M' = lm(P_\Phi^{M'})$ so M' is a stable model of P_Φ and its cardinality is at least $n(k+1) + m + 1$.

Conversely, let M' be a stable model of P_Φ of size at least $|At(P_\Phi)| - 2k$. Clearly all atoms $v[q]$, $q = 1, \ldots, 2k+1$, must be members of M' and, consequently, $u[i] \in M'$, for $i = 1, \ldots, m$. Hence, for each $i = 1, \ldots, m$, there is a rule in P_Φ

$$u[i] \leftarrow x'[i,j,1], x'[i,j,2], \ldots, x'[i,j,m_{ij}]$$

such that $x'[i,j,\ell] \in M'$ if $x'[i,j,\ell] = x$, and $x'[i,j,\ell] \notin M'$ if $x'[i,j,\ell] = \neg x$. Thus, M' is a model of the formula $\bigvee_{j=1}^{m_i} \bigwedge_{\ell=1}^{m_{ij}} x[i,j,\ell]$, for each $i = 1, \ldots, m$. Therefore $M = M' \cap At(\Phi)$ is a model of Φ.

It is a routine task to check that rules $A(x,y,s)$ and $B(x)$ imply that all stable models of P_Φ are of the form

$$At(P_\Phi) \setminus \{x_1, x_2, \ldots, x_k, x_1[1], x_2[2], \ldots, x_k[k]\}$$

($x_1, x_2, \ldots, x_k$ are not necessarily distinct). Hence, $|M| = |M' \cap At(\Phi)| \geq n - k$. We have reduced the problem $\mathcal{M}'_{\leq}(3N)$ to the problem $\mathcal{ST}'_{\leq}(\mathcal{A})$. Thus, the assertion follows by Theorem 1. □

References

Cla78. K.L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and data bases*, pages 293–322. Plenum Press, New York-London, 1978.

DF97. R. G. Downey and M. R. Fellows. *Parameterized Complexity*. Springer-Verlag, 1997.

Fag94. F. Fages. Consistency of Clark's completion and existence of stable models. *Journal of Methods of Logic in Computer Science*, 1:51–60, 1994.

GL88. M. Gelfond and V. Lifschitz. The stable semantics for logic programs. In R. Kowalski and K. Bowen, editors, *Proceedings of the 5th International Conference on Logic Programming*, pages 1070–1080. MIT Press, 1988.

GSS99. G. Gottlob, F. Scarcello, and M. Sideri. Fixed parameter complexity in AI and nonmonotonic reasoning. In M. Gelfond, N. Leone, and G. Pfeifer, editors, *Logic Programming and Nonmonotonic Reasoning, Proceedings of the 5th International Conference, LPNMR99*, volume 1730 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.

Tru01. M. Truszczyński. Computing large and small stable models. *Theory and Practice of Logic Programming*, 2001. To appear.

Ultimate Well-Founded and Stable Semantics for Logic Programs with Aggregates

Marc Denecker, Nikolay Pelov, and Maurice Bruynooghe

Dept. of Computer Science, K.U.Leuven
Celestijnenlaan 200A, B-3001 Heverlee, Belgium
{marcd,pelov,maurice}@cs.kuleuven.ac.be

Abstract. In this paper, we propose an extension of the well-founded and stable model semantics for logic programs with aggregates. Our approach uses Approximation Theory, a fixpoint theory of stable and well-founded fixpoints of non-monotone operators in a complete lattice. We define the syntax of logic programs with aggregates and define the immediate consequence operator of such programs. We investigate the well-founded and stable semantics generated by Approximation Theory. We show that our approach extends logic programs with stratified aggregation and that it correctly deals with well-known benchmark problems such as the shortest path program and the company control problem.

1 Introduction

Programs with aggregates have been investigated in the context of database applications [1,14,17,20]. One important approach to programs with aggregates is *Monotone aggregation* [17,20]. This approach concentrates on programs where the 2-valued immediate consequence operator is monotone with respect to some given lattice order. In such cases, one can take the least fixpoint as the intended model of the program. Another approach is by using stratification. *Aggregate stratified programs* [1,14] can be split up in a sequence of different *strata* such that each aggregate expression only refers to predicates defined in previous strata. A disadvantage of both approaches is that they impose serious syntactic restrictions on the programs. Therefore for many problems, one has to carefully tune the program in such a way that it satisfies the syntactic restrictions. For this reason, a number of attempts have been made for extending the well-founded semantics [21] and the stable semantics [10] for unstratified programs with aggregates.

Well-founded semantics is a natural semantics for deductive databases. As shown in [2,4] it captures a general principle of non-monotone inductive definition. For this reason, [3] used the well-founded semantics as the semantic principle of ID-logic, a logic for knowledge representation which integrates classical logic assertions and inductive definitions. ID-logic can be seen as an extension and provides an epistemological foundation for Abductive logic Programming [11]. Also the stable semantics has been shown to be important for knowledge representation, in particular for nonmonotonic reasoning. Since recently, it is used as the foundation of an emerging paradigm of *stable logic programming* [13]

P. Codognet (Ed.): ICLP 2001, LNCS 2237, pp. 212–226, 2001.

where problems are solved by computing stable models of a logic program. Since recently, aggregates are attracting increasing interest in the context of these extensions. Experiments in solving combinatorial search problems showed that many of these problems cannot be adequately expressed without aggregation. Recently, one of the prominent systems for stable logic programming **smodels** [15] has been extended successfully with a limited form of aggregate constraints [16]. [22] describes an extension of an abductive system with aggregates and some experiments with this system in the context of scheduling and puzzles.

As will be shown in section 5, current extensions of well-founded semantics [12,18,19] and stable model semantics [12,16] of programs with aggregates are still weak in some ways. In this paper, we propose a stronger extension of the well-founded and stable model semantics for logic programs with aggregates. Our approach uses Approximation Theory developed in [5]. With each nonmonotone operator, this theory associates a Kripke-Kleene, a Well-founded and a set of stable fixpoints. In the case of the immediate consequence operator of a logic program, these fixpoints identify exactly the models in the corresponding types of Logic Programming semantics. We define the syntax of logic programs with aggregates and define an immediate consequence operator of such programs. We investigate the well-founded and stable semantics generated by Approximation Theory, and show this approach extends logic programs with stratified aggregation and that it correctly deals with recursive benchmark problems such as the shortest path program and the company control problem. An important property is that if the 2-valued immediate consequence operator is monotone then its ultimate well-founded model coincides with the least fixpoint of this operator.

2 Aggregate Logic Programs: Syntax and T_P

A sorted signature Σ is a set of sort symbols and sorted function, predicate and variable symbols. The arity of a function symbol is a tuple $(s_1, .., s_n) : s$ where $s_1, .., s_n, s$ are sort symbols; the arity of a predicate symbol is a tuple $(s_1, .., s_n)$. We introduce a set $Agg \subseteq \Sigma$ of sorted second order predicate symbols. These will be called the aggregate symbols. The arity of a symbol F of Agg is specified as a tuple $(\tau_1, .., \tau_n)$ where τ_i is either a tuple $(s_1, .., s_m)$ specifying the sort of a relation argument, a tuple $(s_1, .., s_m) : s$ specifying the sort of a function argument or a first order sort symbol s. Sort symbols nat, int and $real$ denote the natural, respectively integer and real numbers.

Example 2.1. The following aggregate symbols are used below:

- $Card_s$: has arity $((s), nat)$ denotes the cardinality relation of sets of sort s.
- Min_s, Max_s: have arity $((s), s)$ and denote the minimality, respectively maximality relation of sets of the sort s representing a partial order.
- $Sum_{(s_1,..,s_n)}$: has arity $((s_1, .., s_n), (s_1, .., s_n) : nat, nat)$ and denotes the sum function that summates a function over a set. I.e. for a relation R, function F and integer n, $Sum(R, F, n)$ holds iff $n = \sum_{(x_1,..,x_n)\in R} F(x_1, .., x_n)$ and R is finite.

All these aggregate symbols are indexed by the sorts over which they range. In particular, Agg may contain many cardinality, minimality, maximality or summation aggregates, each ranging over other sorts. However, in the sequel, we will drop the annotations with sorts and will write $Card, Min, Max, Sum, ...$

A signature Σ is interpreted over a *Σ-structure* I; this is an assignment of:

- a set D_s to each sort symbol s,
- a function $f_I : D_{s_1} \times .. \times D_{s_n} \to D_s$ to each function symbol f with arity $(s_1, .., s_n) : s$,
- a relation $p_I \subseteq D_{s_1} \times .. \times D_{s_n}$ to each predicate symbol p with arity $(s_1, .., s_n)$.
- an appropriately typed second order relation F_I to each aggregate symbol $\mathrm{F} \in Agg$.

Next we define the syntax of aggregate formulas and aggregate programs based on Σ as a restricted subset of second order formulas. We define the notions of *lambda expression* and of *set expression, aggregate atom* and *aggregate formula* by simultaneous induction:

- A *lambda expression* is of the form $\lambda(X_1, .., X_n)t$ where $X_1, .., X_n$ are different variables and t is a first order term. Variables appearing in t and not among $X_1, .., X_n$ are called the free variables of the lambda expression.
- A *set expression* is of the form $\{(X_1, .., X_n)|\phi\}$ where again $X_1, .., X_n$ are different variables and ϕ an aggregate formula. Variables appearing in ϕ and not among $X_1, .., X_n$ are called the free variables of the set expression.
- An *aggregate atom* is a well-typed second order atom where each function argument contains a lambda expression, each relation argument contains a set expression and each other argument contains a term.
- An *aggregate formula* is either a (first order) atom, an aggregate atom or is a composed formula obtained by applying the normal composition rules for $\wedge, \vee, \leftarrow, \neg, \forall, \exists$.

Variables of aggregate atoms are first order, hence quantification is never over second order variables. Also, a set expression which is part of an aggregate atom in turn contains an aggregate formula, hence, nested aggregation is allowed.

Example 2.2. The formula

$$\forall T.Sum\left(\begin{array}{c}\{(U,C)|bucket(U) \wedge \neg leaking(U) \wedge capacity(U,C)\},\\ \lambda(U,C)C,\\ T\end{array}\right) \to T \geq 1000$$

expresses that the sum of the capacity of non-leaking buckets is at least 1000. The first argument denotes the set of pairs of a non-leaking bucket and its capacity while the second argument represents the projection function. The argument U (the *bucket*) in the set and lambda expression is necessary as a capacity has to be counted as many times as there are non-leaking buckets with it. Using a function *capac* mapping buckets on their capacity, an alternative formulation is:

$$\forall T.Sum\left(\begin{array}{c}\{(U)|bucket(U) \wedge \neg leaking(U)\},\\ \lambda(U)capac(U),\\ T\end{array}\right) \to T \geq 1000$$

Its meaning is: $\forall T.(T = \sum_{U \in \{U \mid bucket(U) \wedge \neg leaking(U)\}} capac(U)) \rightarrow T \geq 1000$

Aggregate expressions have been denoted in different ways. The most common (e.g [12]) is of the form $group\_by(p(\overline{X}, \overline{Y}), [\overline{X}], C = F(t[\overline{X}, \overline{Y}]))$ where $\overline{X}$ are the grouping variables and C is the result of computing the aggregate function F. It is equivalent in our notation to: $\text{F}(\{\overline{Y} \mid p(\overline{X}, \overline{Y})\}, \lambda(\overline{Y})t[\overline{X}, \overline{Y}], C)$. For example, the atom $group\_by(d(A, B, M), [A], N = Sum(M))$ is written in the syntax defined here as $Sum(\{(B, M) \mid d(A, B, M)\}, \lambda(B, M)M, N)$ and denotes that $N = \sum_{(B,M) \in \{(B,M) \mid d(A,B,M)\}} M$.

Next we define aggregate programs. Let Σ be a sorted signature of symbols and $\Sigma' = \Sigma \cup \Sigma_d$ an extension of Σ with a set Σ_d of sorted predicate symbols, called the *defined predicate symbols*. An aggregate program based on Σ' is a set of rules of the form $p(t_1, .., t_n) \leftarrow B$ where p is a defined predicate, $t_1, .., t_n$ are appropriately typed first order terms and B is an aggregate formula based on Σ'. The resulting syntax extends the standard logic programming syntax by allowing aggregate expressions and general formulas in the body of rules.

2.1 Examples

Example 2.3 (Shortest path). Given are two sorts *edge* and *weight* and an interpreted predicate symbol *edge* of arity $(edge, edge, weight)$ representing a directed weighted graph. The defined predicate *sp* has the same sort; $sp(X, Y, Z)$ means that the shortest path from X to Y has total weight W. It is defined as:

$$sp(X, Y, W) \leftarrow Min\left(\left\{ W \;\middle|\; \begin{array}{c} edge(X, Y, W) \vee \\ \exists Z, W_1, W_2.(sp(X, Z, W_1) \wedge \\ edge(Z, Y, W_2) \wedge W = W_1 + W_2) \end{array} \right\}, W\right).$$

Example 2.4 (Company control). A sort *comp* represents companies and a sort *shares* interpreted over the real interval [0..1] represents shares. The interpreted predicate *owns_stock* has arity $(comp, comp, shares)$ and $owns\_stock(X, Y, S)$ means that a company X owns a fraction S of the stock of a company Y. The defined predicate $Controls$ has arity $(comp, comp)$; $Controls(X, Y)$ denotes that company X has a controlling interest in a company Y. This is the case if X owns (by its own shares in Y augmented with the shares of other companies Z controlled by X) more than 50% of the stock of Y. It is defined as:

$$Controls(X, Y) \leftarrow \begin{array}{l} Sum\left(\left\{ (Z, S) \;\middle|\; \begin{array}{c} owns\_stock(X, Y, S) \wedge Z = X \vee \\ controls(X, Z) \wedge owns\_stock(Z, Y, S) \end{array} \right\}, \right. \\ \qquad \left. \begin{array}{c} \lambda(Z, S)S, \\ T \end{array} \right) \\ \wedge T > 0.5 \end{array}$$

2.2 Immediate Consequence Operator

First, we give a standard definition of truth assignment of aggregate formulas. Given is a signature Σ and a Σ-structure I. A variable assignment σ is a (well-typed) mapping from the variables of Σ to domain elements. For a given variable assignment σ, a tuple of variables $\overline{X} = (X_1, .., X_n)$ and a tuple of domain elements $\overline{x} = (x_1, .., x_n)$, we denote by $\sigma[\overline{X}/\overline{x}]$ the variable assignment σ' such that $\sigma'(X) = \sigma(X)$ for each variable X not in $\overline{X}$, and $\sigma'(X_i) = x_i$. We define satisfiability $\models$ of a formula in a structure and the evaluation operator $eval_{I,\sigma}$ which maps first order terms to domain elements, lambda expressions to functions and set expressions to relations. The definition is by simultaneous induction:

- $eval_{I,\sigma}(X) = \sigma(X)$
- $eval_{I,\sigma}(f(t_1, .., t_n)) = f_I(eval_{I,\sigma}(t_1), .., eval_{I,\sigma}(t_n))$
- $eval_{I,\sigma}(\lambda(\overline{X})t) = \{(\overline{x}, y) \mid (\overline{x}, y) \in D_{s_1} \times .. \times D_{s_n} \times D_s$ and $y = eval_{I,\sigma}[\overline{X}/\overline{x}](t)\}$ where $(s_1, .., s_n) : s$ is the arity of the lambda expression.
- $eval_{I,\sigma}(\{\overline{X}|\phi\}) = \{\overline{x}|\ \overline{x} \in D_{s_1} \times .. \times D_{s_n}$ and $I, \sigma[\overline{X}/\overline{x}] \models \phi\}$ where $(s_1, .., s_n)$ is the arity of the set expression.
- for each first order or aggregate atom $\mathrm{F}(t_1, .., t_n)$, $I, \sigma \models \mathrm{F}(t_1, .., t_n)$ iff $(eval_{I,\sigma}(t_1), .., eval_{I,\sigma}(t_n)) \in \mathrm{F}_I$
- the standard truth recursion for $\wedge, \vee, \neg, \rightarrow, ...$

Given is a signature $\Sigma' = \Sigma \cup \Sigma_d$ (symbols of Σ (Σ_d) are called the interpreted symbols (respectively the defined predicate symbols)), we assume the existence of a Σ-structure I_o. It may interpret some of the sorts from Σ over the integers or reals. If Σ contains the standard arithmetic function and predicate symbols such as $+, -, \leq$ they are interpreted in the standard way. Other sorts represent user-defined sorts and are interpreted appropriately. Aggregate symbols are also part of the signature Σ and are interpreted by some fixed domain independent relation (e.g. the minimum relation over a poset, the cardinality of a set, sum, product, etc.). An analogy with database terminology can be drawn here: defined predicates correspond to intensional database predicates (IDB) while interpreted predicates correspond to extensional database predicates (EDB).

Let P be a Σ'-aggregate program and $\mathcal{I}_{I_o}$ be the set of all Σ'-structures extending I_o and interpreting the defined symbols. With these definitions, the immediate consequence operator of P can be defined in a straightforward way.

Definition 2.1 (2-valued T_P operator for aggregate programs). *We define the immediate consequence operator $T_P : \mathcal{I}_{I_o} \rightarrow \mathcal{I}_{I_o}$ of P as the operator mapping an interpretation I to $T_P(I) = I'$ such that for each defined symbol p, $p_{I'} = \{(x_1, .., x_n)|$ there is a rule $p(t_1, .., t_n) \leftarrow B$ in P and a variable assignment σ such that $eval_{I,\sigma}(t_1, .., t_n) = (x_1, .., x_n)$ and $I, \sigma \models B\}$.*

Typically, the intended interpretation of a logic program P is given by one or some subset of fixpoints of the T_P operator. The semantics defined in the next sections is a fixpoint semantics. For each pair $I_1, I_2 \in \mathcal{I}_{I_o}$, define $I_1 \sqsubseteq I_2$ iff for each defined symbol p, $p_{I_1} \subseteq p_{I_2}$. A well-known phenomenon is that aggregates lead to T_P operators that are nonmonotone with respect to this order, even when the program p doesn't contain a negation symbol.

Example 2.5. Consider the program: $p(a) \leftarrow Card(\{X \mid p(X)\}, N) \wedge N < 2$.

The T_P of this program is non-monotone as it maps any interpretation of p with at least two elements to an interpretation assigning the empty set to p. The unique fixpoint of T_P interprets p by $\{a\}$. This seems a natural solution.

For aggregate programs with a $\sqsubseteq$-monotone T_P operator, T_P has a least fixpoint. As argued in [12], it is a natural solution for the semantics of P[1].

Example 2.6. Reconsider the company control program from Example 2.4. In the natural situation where the sort *shares* is interpreted with positive numbers, the T_P is monotone. Indeed, if we add new *Controls* facts to the structure, then the result of summing the shares of controlling companies can only increase and hence more *Controls* facts can be derived. It is well-known that the least fixpoint of this operator is the intended solution of this problem.

The examples above illustrate cases where the intended interpretation corresponds to a minimal fixpoint of the immediate consequence operator. In general, nonmonotone operators may have no or multiple (minimal) fixpoints. For pure logic programs, not all minimal fixpoints are intended interpretations. This can also happen with aggregate programs:

Example 2.7. Consider the program:

$$p(0) \leftarrow Card(\{X \mid p(X)\}, 1) \qquad\qquad q \leftarrow \neg p(0)$$

This program has two fixpoints. One in which p is interpreted by the singleton 0, and q is false, and another in which p is empty and q is true. Both are minimal, but clearly only the second is intended. The first fixpoint is not acceptable because $p(0)$ only depends positively on itself, hence, the model is not well-founded.

Approximation theory [5,6] is a fixpoint theory that assigns fixpoints to any nonmonotone operator in a complete lattice and was used to describe the semantics of logic programs, default logic and autoepistemic logic. The next section recalls its basics and uses it to define a semantics for aggregate programs.

3 Approximation Theory

Let $\langle L, \leq \rangle$ be a complete[2] lattice with least element $\bot$ and largest element $\top$. The bilattice of $\langle L, \leq \rangle$ is the structure $\langle L^2, \leq, \leq_i \rangle$ where we define $(x, y) \leq (x_1, y_1)$ iff $x \leq x_1$ and $y \leq y_1$, and $(x, y) \leq_i (x_1, y_1)$ iff $x \leq x_1$ and $y \geq y_1$. Both $\leq$ and $\leq_i$ are complete lattice orders in L^2. The order $\leq$ has least element $(\bot, \bot)$ and

[1] Note that the approach of monotone aggregation is far more general than the case of programs with a $\sqsubseteq$-monotone T_P operator. Indeed, in monotone aggregation, programs are considered for which T_P is monotone with respect to some user defined order. Such operators are not necessarily $\sqsubseteq$-monotone.

[2] Each subset has a least upper bound and a greatest lower bound.

largest element $(\top, \top)$ whereas $\leq_i$ has least element $(\bot, \top)$ and largest element $(\top, \bot)$.

In approximation theory, the intuition underlying bilattice elements (x, y) is that they are *approximations* of lattice elements. In particular (x, y) approximates a lattice element z if $x \leq z \leq y$. The elements x and y constitute respectively an underestimate and overestimate of z. The set of lattice elements approximated by (x, y) is denoted $[x, y]$. Note that this interpretation is only possible for tuples (x, y) such that $x \leq y$. We call such tuples *consistent*. Inconsistent tuples do not approximate any lattice element ($[x, y]$ is empty). The order $\leq_i$ is called the information order; $(x, y) \leq_i (x_1, y_1)$ implies that (x_1, y_1) is a more precise approximation than (x, y) (in particular, $[x, y] \supseteq [x_1, y_1]$). Note that for tuples (x, x), $[x, x] = \{x\}$. Therefore, the tuples of identical elements constitute the natural embedding of L in L^2.

We define a framework for studying the fixpoints of any operator on a complete lattice by investigating the fixpoints of approximations.

Definition 3.1. *Let $O : L \to L$ be an operator on L. An* approximation $A : L^2 \to L^2$ *of O is a $\leq_i$-monotone operator such that:*

- *$A(x, x) = (O(x), O(x))$, that is, A coincides with O on the embedding of L in L^2.*
- *A is symmetric, that is, if $A(x, y) = (x_1, y_1)$ then $A(y, x) = (y_1, x_1)$.*

Every operator $A : L^2 \to L^2$ can be defined by a unique pair A_1, A_2 of functions of type $L^2 \to L$ such that for each (x, y), $A(x, y) = (A^1(x, y), A^2(x, y))$. It is straightforward to see that A is an approximation of some operator iff (1) for each pair (x, y), $A^1(x, y) = A^2(y, x)$ and (2) A^1 is monotone in its first argument and antimonotone in its second argument. Vice versa, each operator $A^1 : L^2 \to L$ which is monotone in its first and anti-monotone in its second argument can be used to construct an approximation A. A maps a tuple (x, y) to the tuple $(A^1(x, y), A^1(y, x))$ and approximates the operator $O : L \to L$ where for each x, $O(x) = A^1(x, x)$.

Let A be an approximation of O. Then A has a least fixpoint $\mathrm{KK}(A)$ in the information order $\leq_i$, called the Kripke-Kleene fixpoint of A. The Kripke-Kleene fixpoint approximates each fixpoint of O. This is not good enough if we want to approximate minimal fixpoints of O (or some subset of them).

In order to obtain better approximations, we define a *stable operator* $ST_A : L \to L$. Observe that with each lattice element x, we can associate an $L \to L$ operator, denoted $A^1(., x)$ which maps elements z to $A^1(z, x)$. The operator $A^1(., x)$ is monotone and has a least fixpoint. Define $ST_A(x) = lfp(A^1(., x))$. ST_A is an anti-monotone operator. Fixpoints of ST_A are called *stable fixpoints* of A. It can be shown that they are minimal fixpoints of O. The *extended stable operator* $\mathcal{ST}_A : L^2 \to L^2$ maps (x, y) to $(ST_A(y), ST_A(x))$ and is $\leq_i$-monotone. It has a $\leq_i$-least fixpoint $WF(A)$, called the well-founded fixpoint of A.

An operator O may have many different approximations. It was investigated in [6] how the fixpoints of these different approximations relate and how approximations can be constructed from O. Let L^c be the restriction of L^2 to the

consistent pairs. We say that an approximation A_1 is *less precise* than approximation A_2 and denote by $A_1 \leq_i A_2$ if for each $(x, y) \in L^c$, $A_1(x, y) \leq_i A_2(x, y)$. The following properties have been proven:

Proposition 3.1. *[6] Let $A_1 \leq_i A_2$. Then: (1) A stable fixpoint of A_1 is a stable fixpoint of A_2. (2) The Kripke-Kleene and well-founded fixpoint of A_1 are less precise (w.r.t $\leq_i$) than those of A_2.*

Two important observations can be derived. First, two approximations coinciding on L^c have the same stable, Kripke-Kleene and well-founded fixpoints. The behaviour of the approximation on the inconsistent elements is not relevant! Hence, it suffices to define an approximation only on the consistent elements.

Definition 3.2 (Partial Approximation). *An operator $A : L^c \to L^c$ is a* partial approximation *of an operator $O : L \to L$ if A is $\leq_i$-monotone and $A(x, x) = (O(x), O(x))$ for each lattice element x.*

It can be proven that any partial approximation can be extended to an approximation defined on L^2. Hence, the sequel focuses on partial approximations.

Improving an approximation yields increasing sets of stable fixpoints and more precise Kripke-Kleene and well-founded fixpoints. There must be a limit to this process. One can construct a most precise partial approximation:

Definition 3.3 (Ultimate Approximation). *Let $O([x, y]) = \{O(z) | z \in [x, y]\}$. The* ultimate partial approximation *$Ult(O) : L^c \to L^c$ of $O : L \to L$ is:*

$$Ult(O)(x, y) = (glb(O([x, y])), lub(O([x, y])))$$

It is easy to show that $Ult(O)$ is $\leq_i$ monotone, coincides with O on the pairs (x, x), and is more precise than any other partial approximation of O. Moreover, its fixpoints are determined only by O. Thus, for each operator O on $\langle L, \leq \rangle$, we define the *ultimate stable* (respectively *ultimate Kripke-Kleene* and *ultimate well-founded*) fixpoints of O as the stable (respectively Kripke-Kleene and well-founded) fixpoints of its ultimate partial approximation $Ult(O)$.

3.1 Approximations of Logic Programs

Approximation theory was used in [7] for describing the semantics of Logic Programming, Default logic and Autoepistemic logic. In the case of Logic Programming, the underlying lattice is the lattice of Herbrand interpretations. The bilattice corresponds to 4-valued interpretations. In particular, any pair (I, J) of 2-valued interpretations defines the following truth function:

- p is true in (I, J) iff $p \in I \cap J$; i.e. under- and overestimate agree on the truth of p.
- p is false in (I, J) iff $p \notin I \cup J$; i.e. under- and overestimate agree on the falsity of p.

- p is undefined in (I, J) iff $p \in J \setminus I$; i.e. underestimate of p is false and overestimate is true.
- p is inconsistent in (I, J) iff $p \in I \setminus J$; i.e. underestimate of p is true and overestimate is false.

The consistent bilattice elements (I, J) correspond exactly to the 3-valued interpretations (since $I \setminus J$ is empty).

In terms of the bilattice representation of 4-valued interpretations, the four-valued immediate consequence operator $\mathcal{T}_P$ of a program P [9] can be defined as the operator that maps any pair (I, J) to $(\mathcal{T}_P^1(I, J), \mathcal{T}_P^1(J, I))$. Here, the operator $\mathcal{T}_P^1(I, J)$ is defined as the 2-valued interpretation I' such that a ground atom p is true in I' iff there exists a ground instance of a ground rule $p \leftarrow B$ such that each positive literal of B is true in I and each negative literal is true in J.

The main result about the relationship between approximation theory and LP-semantics is that the 4-valued immediate consequence operator $\mathcal{T}_P$ of a program P is an approximation of the 2-valued immediate consequence operator T_P, that the stable operator of P is the stable operator of $\mathcal{T}_P$ and hence that the Kripke-Kleene, stable and well-founded models of P are the Kripke-Kleene, stable and well-founded fixpoints of $\mathcal{T}_P$.

In general, $\mathcal{T}_P$ is not the ultimate approximation[3] of T_P, and consequently, the ultimate fixpoints of T_P are not the Kripke-Kleene, stable and well-founded models of P. Below are shown some important relationships between the standard models and the ultimate versions. Let P be a logic program.

Proposition 3.2. *(a) The well-founded (resp. Kripke Kleene) model of P is less precise than the ultimate well-founded (resp. Kripke Kleene) model of P. (b) If the well-founded model of P is 2-valued then it is the ultimate well-founded model of P. (c) If P_1, P_2 are two different programs such that their 2-valued immediate consequence operators coincide, then their ultimate Kripke-Kleene, ultimate well-founded and ultimate stable model coincide.*

Example 3.1. A simple example of a logic program for which the standard and ultimate semantics differ is the following:

$$P_1 = \{\, p \leftarrow p \qquad p \leftarrow \neg p \,\}$$

Its Kripke-Kleene and well-founded fixpoint is $(\{\}, \{p\})$ (i.e. p is undefined) and it has no stable models. On the other hand, note that the 2-valued immediate consequence operators of this program is monotone and that it coincides with the immediate consequence operator of the program

$$P_2 = \{\, p \leftarrow \,\}$$

[3] It can be seen easily that the 3-valued ultimate approximation is obtained by evaluating rule bodies, not according to the standard 3-valued truth function but using the technique of *supervaluations*. In this scheme, a formula is true(false) in a 3-valued interpretation I if it is true(false) in each 2-valued interpretation approximated by I. Otherwise, the formula is undefined.

Hence, they have the same ultimate models. Since the Kripke-Kleene model of P_2 is the 2-valued interpretation $\{p\}$, this is also the ultimate Kripke-Kleene, ultimate well-founded and the unique ultimate stable model of P_1.

3.2 Ultimate Semantics for Programs with Aggregates

Given is a signature $\Sigma' = \Sigma \cup \Sigma_d$, a Σ-structure I_o for the interpreted symbols Σ, and an aggregate program with defined predicates Σ_d. The structure $\langle \mathcal{I}_{I_o}, \sqsubseteq \rangle$ of Σ'-structures extending I_o is a complete lattice. The operator T_P is an operator in this lattice. Hence, we can apply ultimate approximation theory.

Definition 3.4. *The ultimate Kripke-Kleene, ultimate well-founded and ultimate stable models of P are the ultimate Kripke-Kleene, ultimate well-founded and ultimate stable fixpoints of T_P.*

4 Analysis and Results

In this section, we investigate the ultimate semantics of aggregate programs. Everywhere in this section, we assume the existence of a given signature Σ, the Σ-structure I_o and an aggregate program P based on $\Sigma' = \Sigma \cup \Sigma_d$.

The first result shows that if the immediate consequence operator is monotone, then the ultimate well-founded model is the least fixpoint of this operator.

Theorem 4.1. *[6] If T_P is $\sqsubseteq$-monotone then its ultimate well-founded and unique stable fixpoint are equal to the least fixpoint of T_P.*

Note that this property is not satisfied by the well-founded semantics. Example 3.1 illustrates this. The 2-valued T_P operator of the program P_1 is constant and hence monotone, yet its least fixpoint is not the well-founded model of P_1.

Aggregate stratified programs are studied in [1,14]. P is an aggregate stratified program iff the body B of any rule is a conjunction of atoms, negative literals and aggregate atoms. Moreover, for each predicate symbol p there is a level number i_p such that for each rule $p \leftarrow B$, if predicate symbol q occurs in an atom of B then $i_q \leq i_p$, if q appears in a negative literal in B or in a set expression in an aggregate atom in B, then $i_q < i_p$. In other words, predicate symbols appearing negatively or in an aggregate atom in the body of a rule defining p must be defined in lower levels.

The next theorem expresses that the ultimate well-founded (and stable) semantics of aggregate programs generalises the perfect model semantics of aggregate stratified programs.

Theorem 4.2. *If P is a aggregate stratified program then its ultimate well-founded model is its perfect model [1,14].*

Another theorem shows that each aggregate program has a natural translation into a ground, infinitary logic program, such that the ultimate well-founded model of the aggregate program is the well-founded model of its translation.

Let $\mathcal{B}_o$ be the set $\{p(x_1,..,x_n)|p \in \Sigma_d$ and $x_1,..,x_n \in D_{s_1} \times .. \times D_{s_n}$ (with $(s_1,..,s_n)$ the arity of $p)\}$. For any 2-valued interpretation I extending I_o and any set B of positive or negative literals of $\mathcal{B}_o$, define that $I \models B$ iff $(x_1,..,x_n) \in p_I$ if $p(x_1,..,x_n) \in B$ and $(x_1,..,x_n) \notin p_I$ if $\neg p(x_1,..,x_n) \in B$.

Definition 4.1. *Define $lp(P)$ as the (infinitary) ground logic program based on $\mathcal{B}_o$ and consisting of all ground clauses $p(x_1,..,x_n) \leftarrow B$ for which there exists a variable assignment σ and a rule $p(t_1,..,t_n) \leftarrow F$ in P such that:*

- *$p(x_1,..,x_n) = p(eval_{I,\sigma}(t_1),..,eval_{I,\sigma}(t_n))$ and*
- *B is any set of literals of $\mathcal{B}_o$ such that for each 2-valued Σ-interpretation I extending I_o: if $I \models B$ then $I, \sigma \models F$.*

Theorem 4.3. *The ultimate well-founded model of P is the well-founded model of $lp(P)$.*

Finally, we return to the examples of Section 2.1. Since the immediate consequence operator of company control program is monotone (if shares are positive), the following proposition follows.

Proposition 4.1. *For any finite structure I_o interpreting owns_stock and interpreting the sort shares by positive real numbers, the ultimate well-founded model is 2-valued and is the unique stable model. Moreover, a fact $Controls(c_1, c_2)$ belongs to this model iff a company c_1 controls a company c_2.*

The ultimate well-founded model of the shortest path problem correctly represents shortest path distances.

Proposition 4.2. *Let I_o be a structure for the shortest path program P_{sp} defining a weighted directed graph with positive weights. Then the ultimate well-founded model of P_{sp} is 2-valued and contains a fact $sp(x, y, w)$ iff there exists a path between x and y with total weight w and the total weight of all other paths between x and y is greater or equal to w.*

5 Related Work

The following toy-problems illustrate some issues in the relationship between ultimate well-founded semantics and monotone aggregation [17].

In the example below, $p/1$ is a predicate interpreted over the domain $\{0, 1, 2\}$.

$$p(X) \leftarrow Min(\{Y|Y = 1 \vee p(Y)\}, X)$$

Intuitively, the intended meaning is $p_I = \{p(1)\}$. However, note that $p_{I'} = \{p(0)\}$ is a fixpoint too. Note that the immediate consequence operator of this program is not monotone with respect to the standard order $\sqsubseteq$ on interpretations. However, it is monotone with respect to other orders. In approaches of monotone aggregation, one must find such an appropriate order. In monotone aggregation,

only structures that assign a singleton set to p are considered. In this collection of structures, one can define the following order: $I \leq J$ if it holds that if $p_I = \{x\}$ and $p_J = \{y\}$, then $x \leq y$. T_P is monotone with respect to this order. However, its minimal fixpoint assigns $p_I = \{0\}$. T_P is also monotone with respect to the inverse order $\geq$, and its least fixpoint in this order is the intended solution $p_I = \{1\}$. This shows that the selection of the right order is important in monotone aggregation. Consider now the following extension:

$$p(X) \leftarrow Min(\{Y|Y = 1 \vee p(Y)\}, X)$$
$$r(X) \leftarrow Min(\{Y|Y = 1 \vee r(Y)\}, Z) \wedge X = 2 - Z$$

The immediate consequence operator is not longer monotone with respect to $\geq$. The above program has the same T_P operator as the following program (obtained by removing redundant rules from lp(P)):

$$p(0) \leftarrow p(0) \qquad r(2) \leftarrow r(0)$$
$$p(1) \leftarrow \neg p(0) \qquad r(1) \leftarrow \neg r(0)$$

This program is stratified and has a 2-valued well-founded model $\{p(1), r(1)\}$. It is the ultimate well-founded model of the original program.

The above examples suggest that the approach developed in this paper might be both more general and simpler than monotone aggregation. A formal investigation of the relationship between ultimate semantics and monotone aggregation needs to be conducted.

Kemp and Stuckey [12] define a well-founded semantics of logic programs with aggregates. However, their definition doesn't deal well with positive recursion through aggregation. Hence, their semantics is strictly weaker than the ultimate well-founded semantics defined in this paper. For example, in the well-founded model of [12] of the following program

$$p(0) \leftarrow Card(\{X|p(X)\}, 1)$$

the atom p is undefined while it is false in the ultimate well-founded model.

In [18] was proposed another extension of the well-founded and valid semantics for programs with aggregates. The extension is more in the spirit of the ultimate well-founded semantics. The authors rely on having aggregate functions defined for 3-valued multisets and satisfying certain properties of monotonicity. However, we have shown in this paper (the definition of ultimate approximations) that for defining 3-valued (or even 4-valued) semantics of programs with aggregates one can still work with aggregate functions defined on standard sets. Further investigation is needed about the precise relation of these two approaches.

Dix and Osorio [8] have also observed that the standard well-founded semantics is not strong enough for programs with aggregation. They have pointed out that the WFS$^+$ (introduced independently by Dix and Schlipf) is well-suited, however its complexity is on the first level of the polynomial hierarchy. In their paper [8] they propose two weaker extensions of the well-founded semantics WFS^1 and WFS^2 which are polynomially computable. The common property

for these two semantics is that a is true for a program containing the single rule $a \leftarrow \neg a$. For this program, a is undefined in the ultimate well-founded semantics. Whether the ultimate well-founded semantics is weaker in general than WFS[1] and WFS[2] is an open question.

[12] also defines a stable semantics of programs with aggregates where the aggregate atoms are treated like negative literals during the stability transformation. As the authors have pointed out, this has the effect that a program may have non minimal stable models. In the ultimate well-founded and stable semantics for aggregate programs, such problems do not appear.

Another extension of the stable model semantics for aggregate programs [16] was done in the context of the **smodels** system [15]. The authors consider weight constraints of the form $L \leq \{l_1 = w_1, \ldots, l_n = w_n\} \leq U$ where l_i are literals and w_i are real numbers representing weights. Such constraint is true in an interpretation if the sum of the weights of the literals satisfied by the interpretation is between L and U. For details about how stable models of programs with weight constraints are computed we refer to [16].

To show the relationship with our semantics we can translate a weight constraint to an aggregate formula of the form

$$\exists S.Sum(\{I|\ l_1 \wedge I = 1 \vee \ldots \vee l_n \wedge I = n\}, \lambda(I)w(I), S) \wedge L \leq S \wedge S \leq U$$

where $w(I)$ is a new interpreted function which corresponds to the weights w_i. For programs with weight constraints only in the body we can show that stable models are ultimate stable models of the aggregate program obtained by applying the above transformation. In most cases the inverse will also hold but not always, as Example 3.1 shows. The advantage of using a special form of aggregate expressions, like weight constraints, is that more efficient algorithms can be developed. In the case of the **smodels** system, the complexity of computing stable models remains in the same class as for logic programs without aggregates [16]. The reason for this is the use of a less precise approximation underlying **smodels**.

In [3], ID-logic is defined as a logic extending first order logic with *(general nonmonotone inductive) definitions*. Each definition can be thought of as one logic program defining a subset of predicates. Using the ideas presented here, one can extend ID-logic with aggregates. A theory in this extended logic consists of *aggregate formulas* and *aggregate programs* as defined here. A model of such a theory is a 2-valued interpretation I that satisfies all aggregate formulas and is *an* ultimate well-founded model of each of its aggregate programs.[4]

6 Conclusion and Future Work

In this paper we have used the Approximation Theory to develop a well-founded and stable model semantics for programs with aggregates. We do this by defining a 2-valued T_P operator for programs with aggregates in an intuitive way.

[4] Note that an aggregate program may have many ultimate well-founded models if the interpretation I_o is not fixed but may vary.

Approximation theory then shows how to define a most precise approximation of this operator, called an ultimate approximation. We argued that the well-founded and stable fixpoints of this ultimate approximation are well suited for programs with aggregation. In particular, if the T_P operator of a programs is monotone then the ultimate well-founded model is 2-valued and coincides with the least fixpoint of the T_P operator.

Another important topic for further research is to investigate the complexity of computing the ultimate well-founded semantics for programs with aggregates. We expect that if we restrict to Datalog programs, we will obtain exponential complexity in terms of the number of ground atoms. This is because a one step computation of the ultimate approximation on a 3-valued interpretation (I, J) requires to apply the T_P operator to all possible 2-valued interpretations I' such that $I \sqsubseteq I' \sqsubseteq J$. If the size of the Herbrand base is n then for the first step, there are 2^n possible interpretations. However, Approximation Theory is an interesting setting for reducing the complexity. Indeed, note that the ultimate well-founded model is approximated by the well-founded model of any approximating operator. Less precise alternative approximation operators with lower complexity can be searched for such that their well-founded fixpoints coincide with the ultimate well-founded model for certain collections of well-behaved programs. An example where this is possible is the class of aggregate stratified programs.

References

1. M. P. Consens and A. O. Mendelzon. Low-complexity aggregation in GraphLog and Datalog. *Theoretical Computer Science*, 116(1):95–116, 1993.
2. M. Denecker. The well-founded semantics is the principle of inductive definition. In J. Dix, L. F. del Cerro, and U. Furbach, editors, *Logics in Artificial Intelligence, European Workshop*, volume 1489 of *Lecture Notes in Computer Science*, pages 1–16, Dagstuhl, Germany, 1998. Springer.
3. M. Denecker. Extending classical logic with inductive definitions. In J. Lloyd et al., editors, *1st International Conference on Computational Logic*, volume 1861 of *Lecture Notes in Artificial Intelligence*, pages 703–717, London, U.K., July 2000. Springer.
4. M. Denecker, M. Bruynooghe, and V. W. Marek. Logic programming revisited: logic programs as inductive definitions. *ACM Transactions on Computational Logic*, 2001. accepted.
5. M. Denecker, V. Marek, and M. Truszczyński. Approximating operators, stable operators, well-founded fixpoints and applications in non-monotonic reasoning. In J. Minker, editor, *Logic-based Artificial Intelligence*, pages 127–144. Kluwer Academic Publishers, 2000.
6. M. Denecker, V. Marek, and M. Truszczyński. Ultimate approximations. Report CW 320, Department of Computer Science, K.U.Leuven, Belgium, Sept. 2001.
7. M. Denecker, V. W. Marek, and M. Truszczynski. Uniform semantic treatment of default and autoepistemic logics. In A. Cohn, F. Giunchiglia, and B. Selman, editors, *Proceedings of the 7th International Conference on Principles of Knowledge Representation and Reasoning*, pages 74–84, 2000.

8. J. Dix and M. Osorio. On well-behaved semantics suitable for aggregation. Technical Report TR 11/97, University of Koblenz, Department of Computer Science, Rheinau 1, Apr. 1997. Accepted at ILPS '97 as poster paper.
9. M. Fitting. Bilattices and the semantics of logic programming. *The Journal of Logic Programming*, 11(1,2):91–116, 1991.
10. M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In R. A. Kowalski and K. A. Bowen, editors, *Logic Programming, Proceedings of the 5th International Conference and Symposium*, pages 1070–1080, Seattle, Washington, 1988. MIT Press.
11. A. C. Kakas, R. Kowalski, and F. Toni. Abductive logic programming. *Journal of Logic and Computation*, 2(6):719–770, Dec. 1992.
12. D. B. Kemp and P. J. Stuckey. Semantics of logic programs with aggregates. In V. A. Saraswat and K. Ueda, editors, *Proceedings of the International Logic Programming Symposium*, pages 387–401. MIT Press, 1991.
13. V. W. Marek and M. Truszczyński. Stable models and an alternative logic programming paradigm. In K. R. Apt, V. W. Marek, M. Truszczyński, and D. S. Warren, editors, *The Logic Programming Paradigm: A 25-Year Perespective*, pages 375–398. Springer, 1999.
14. I. S. Mumick, H. Pirahesh, and R. Ramakrishnan. The magic of duplicates and aggregates. In D. McLeod, R. Sacks-Davis, and H.-J. Schek, editors, *16th International Conference on Very Large Data Bases*, pages 264–277. Morgan Kaufmann, 1990.
15. I. Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25(3,4):241–273, 1999.
16. I. Niemelä, P. Simons, and T. Soininen. Stable model semantics of weight constraint rules. In M. Gelfond, N. Leone, and G. Pfeifer, editors, *Proceedings of the Fifth International Conference on Logic Programming and Nonmonotonic Reasoning*, volume 1730 of *Lecture Nortes in Computer Science*, pages 317–331. Springer-Verlag, 1999.
17. K. A. Ross and Y. Sagiv. Monotonic aggregation in deductive databases. *Journal of Computer and System Sciences*, 54(1):79–97, 1997.
18. S. Sudarshan, D. Srivastava, R. Ramakrishnan, and C. Beeri. Extending the well-founded and valid semantics for aggregation. In D. Miller, editor, *Proceedings of the International Logic Programming Symposium*, pages 590–608. MIT Press, 1993.
19. A. Van Gelder. The well-founded semantics of aggregation. In *Proceedings of the Eleventh ACM Symposium on Principles of Database Systems*, pages 127–138. ACM Press, 1992.
20. A. Van Gelder. Foundations of aggregation in deductive databases. In S. Ceri, K. Tanaka, and S. Tsur, editors, *Third International Conference on Deductive and Object-Oriented Databases*, volume 760 of *Lecture Notes in Computer Science*, pages 13–34. Springer-Verlag, 1993.
21. A. Van Gelder, K. A. Ross, and J. S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, 1991.
22. B. Van Nuffelen and M. Denecker. Problem solving in ID-logic with aggregates: some experiments. In A. K. Mark Denecker, editor, *Eight International Workshop on Nonmonotonic Reasoning, special track on Abductive Reasoning*, Breckenridge, Colorado, USA, 2000.

Alternating Fixed Points in Boolean Equation Systems as Preferred Stable Models*

K. Narayan Kumar[1,2], C.R. Ramakrishnan[1], and S.A. Smolka[1]

[1] Department of Computer Science,
State University of New York at Stony Brook
Stony Brook, New York, U.S.A.
{kumar,cram,sas}@cs.sunysb.edu
[2] Chennai Mathematical Institute, Chennai, India.
kumar@smi.ernet.in

Abstract. We formally characterize alternating fixed points of boolean equation systems as models of (propositional) normal logic programs. To this end, we introduce the notion of a *preferred stable model* of a logic program, and define a mapping that associates a normal logic program with a boolean equation system such that the solution to the equation system can be "read off" the preferred stable model of the logic program. We also show that the preferred model cannot be calculated a-posteriori (i.e. compute stable models and choose the preferred one) but rather must be computed in an intertwined fashion with the stable model itself. The mapping reveals a natural relationship between the evaluation of alternating fixed points in boolean equation systems and the Gelfond-Lifschitz transformation used in stable-model computation.
For alternation-free boolean equation systems, we show that the logic programs we derive are stratified, while for formulas with alternation, the corresponding programs are non-stratified. Consequently, our mapping of boolean equation systems to logic programs preserves the computational complexity of evaluating the solutions of special classes of equation systems (e.g., linear-time for the alternation-free systems, exponential for systems with alternating fixed points).

1 Introduction

Model checking [2,15,3] is a verification technique aimed at determining whether a system specification possesses a property expressed as a temporal logic formula. Model checking has enjoyed wide success in verifying, or finding design errors in, real-life systems. An interesting account of a number of these success stories can be found in [4].

Model checking has spurred interest in evaluating *alternating fixed points* as these are needed to express system properties of practical import, such as those

* Research supported in part by NSF grants EIA-9705998, CCR-9876242, EIA-9805735, CCR-9988155, and IIS-0072927, and ARO grants ARO DAAD190110003 and DAAD190110019.

P. Codognet (Ed.): ICLP 2001, LNCS 2237, pp. 227–241, 2001.

involving subtle fairness constraints. Probably, the most canonical temporal logic for expressing alternating fixed points is the modal mu-calculus [14,9], which makes explicit use of the dual fixed-point operators μ (least fixed point) and ν (greatest fixed point). A variety of temporal logics can be encoded in the mu-calculus, including Linear Temporal Logic (LTL), Computation Tree Logic (CTL) and its derivative CTL*.

Fixed-point operators may be nested in mu-calculus formulas and different fixed-point formulas may be mutually dependent on each other. Alternating fixed-point formulas are those having a least fixed point that is mutually dependent on a greatest fixed point.

Recently, it has been demonstrated that logic programming (LP) can be successfully applied to the construction of practical and versatile model checkers [16,7]. Central to this approach is the connection between models of temporal logics and models of logic programs. For example, the XMC model checker [17] verifies an alternation-free modal mu-calculus formula by evaluating the perfect model of an equivalent stratified logic program. While the relationship between models of alternating modal mu-calculus formulae and stable models of logic programs has been conjectured [11], there has been no formal characterization of this connection. Establishing this relationship is the focus of this paper.

The model-checking problem for the modal mu-calculus can be formulated in terms of solving Boolean Equation Systems (BESs): see [19,12] and Appendix A of this paper. A BES is a system of mutually dependent equations over boolean-valued variables, where each equation is designated as a greatest or least fixed point. To characterize the solutions of BESs in terms of models of logic programs, we introduce the notion of *preferred stable models* of normal logic programs, and describe a mapping from BESs to propositional normal logic programs such that the solution to a BES can be obtained from the preferred stable model of the corresponding logic program. The mapping also ensures that alternation-free BESs are mapped to stratified logic programs, and is thus a conservative extension of the mapping used by the XMC system. This preserves the linear-time complexity of model checking alternation-free formulas.

Preferred answer sets [18,1] have been defined as a way to capture priority and preference in knowledge representation. They are defined for a more general class of logic programs including disjunctive logic programs, possibly with different flavors of negation (efault, classical, etc.). When restricted to normal logic programs, preferred answer sets differ from our definition of preferred stable models. For example, in contrast to the notion of preference used in [18], the solution to a BES cannot be found by simply imposing a selection criterion on the stable models of an equivalent logic program (see Example 2.8). Nevertheless, the exact relationship between the solution of a BES and the different preferred answer set semantics proposed in the literature remains to be fully explored.

The rest of this paper develops along the following lines. Section 2 presents an informal introduction to BESs and their relationship to logic programs; these ideas are formalized in Section 3. Section 4 introduces the notion of a preferred stable model of a normal logic program, while Section 6 formally establishes

the relationship between solutions to BESs and preferred stable models. Our concluding remarks are offered in Section 7. Due to space limitations, proofs are omitted or sketched. Full proofs can be found in [10]. The paper also contains an appendix (Appendix A) reviewing the standard connection between model checking in the modal mu-calculus and solving BESs.

2 Overview

2.1 Boolean Equation Systems

A Boolean Equation System (BES) is a sequence of fixed-point equations over boolean variables, with an associated sequence of signs (sign map) that specifies the polarity of the fixed points. The i-th equation is of the form $X_i = \alpha_i$ where α_i is a positive boolean formula over variables $\{X_1, X_2, \ldots\}$ and constants 0 and 1. The i-th sign, σ_i, is μ if the i-th equation is a least fixed-point equation and ν if it is a greatest fixed-point equation. We use $\langle X_1 = \alpha_1, X_2 = \alpha_2, \ldots, X_n = \alpha_n \rangle$ to denote the sequence of equations in a BES of size n, and $\langle \sigma_1, \sigma_2, \ldots, \sigma_n \rangle$ to denote its associated sign map. In a BES of size n, X_1 is called the *innermost* variable (and the equation $X_1 = \alpha_1$ the innermost fixed point), and X_n is called the *outermost* variable. A BES of size n is said to be *closed* if all variables occurring in α_i for all $1 \leq i \leq n$ are drawn from $\{X_1, X_2, \ldots, X_n\}$.

In the following, we use ϕ to range over BESs, and $\mathcal{E}$ (possibly subscripted) for specific BESs. Let ϕ be a BES $\langle X_1 = \alpha_1, X_2 = \alpha_2, \ldots, X_n = \alpha_n \rangle$. We use ϕ_i to denote the subsystem $\langle X_1 = \alpha_1, X_2 = \alpha_2, \ldots, X_i = \alpha_i \rangle$. Thus $\phi = \phi_n$, and ϕ_0 denotes the empty BES $\langle\ \rangle$.

In the following, we use a series of examples to informally describe the semantics of a BES and the relationship between BESs and logic programs; these ideas are formalized in subsequent sections of the paper.

A solution of a BES is a truth assignment to the variables $\{X_1, X_2, \ldots\}$ satisfying the fixed-point equations such that outer equations have higher priority over inner equations. More precisely, a solution of a BES $\langle X_1 = \alpha_1, X_2 = \alpha_2, \ldots, X_n = \alpha_n \rangle$ is a valuation that is a fixed point for the outermost variable X_n and is "minimal" (closest to 0 if $\sigma_n = \mu$, and closest to 1 if $\sigma_n = \nu$) among all solutions for the subsystem $\langle X_1 = \alpha_1, X_2 = \alpha_2, \ldots, X_{n-1} = \alpha_{n-1} \rangle$.

Example 2.1. Consider the BES $\mathcal{E}_1 = \langle X_1 = X_1 \wedge X_2, X_2 = X_1 \vee X_2 \rangle$ with sign map $\langle \mu, \mu \rangle$. We first consider all solutions to the subsystem $\langle X_1 = X_1 \wedge X_2 \rangle$. The fixed points for the subsystem are $(X_1 = 0, X_2 = 0)$, $(X_1 = 0, X_2 = 1)$ and $(X_1 = 1, X_2 = 1)$. X_2 is free in the subsystem so we need to consider solutions to the subsystem independently of the value of X_2. For $X_2 = 0$, there is only one fixed point and hence $(X_1 = 0, X_2 = 0)$ is a solution. Among the two fixed points corresponding to $X_2 = 1$, the least (since $\sigma_1 = \mu$) is $(X_1 = 0, X_2 = 1)$. Hence the solutions for the subsystem are $(X_1 = 0, X_2 = 0)$ and $(X_1 = 0, X_2 = 1)$. Both of these valuations are fixed points for $X_2 = X_1 \vee X_2$, but $(X_1 = 0, X_2 = 0)$ is the least fixed point and (since $\sigma_2 = \mu$) the solution to $\mathcal{E}_1$. □

Example 2.2. The signs of both equations in $\mathcal{E}_1$ were identical; for a more complex example, consider the BES $\mathcal{E}_2 = \langle X_1 = X_1 \wedge X_2, X_2 = X_1 \vee X_2, X_3 = X_3 \wedge X_2 \rangle$ with sign map $\langle \mu, \mu, \nu \rangle$. Following the evaluation of $\mathcal{E}_1$, it is easy to see that the solutions of the subsystem $\langle X_1 = X_1 \wedge X_2, X_2 = X_1 \vee X_2 \rangle$ are $(X_1 = 0, X_2 = 0, X_3 = 0)$ and $(X_1 = 0, X_2 = 0, X_3 = 1)$. Of these, only $(X_1 = 0, X_2 = 0, X_3 = 0)$ is a fixed point for $X_3 = X_3 \wedge X_2$ and hence is the solution for $\mathcal{E}_2$. □

In $\mathcal{E}_2$, the inner subsystem's solutions were independent of the values assigned to the outer variable X_3. This property does not hold in general, as shown by the following example.

Example 2.3. Consider the BES $\mathcal{E}_3 = \langle X_1 = X_1 \wedge X_2, X_2 = X_1 \vee X_2 \rangle$ with sign map $\langle \nu, \mu \rangle$. We first consider all solutions for the subsystem $\langle X_1 = X_1 \wedge X_2 \rangle$. As in $\mathcal{E}_1$, the fixed points of the subsystem are $(X_1 = 0, X_2 = 0)$, $(X_1 = 0, X_2 = 1)$ and $(X_1 = 1, X_2 = 1)$. The only fixed point having $X_2 = 0$ is $(X_1 = 0, X_2 = 0)$ and is a solution. Among the two fixed points corresponding to $X_2 = 1$, the greatest (since $\sigma_1 = \nu$) is $(X_1 = 1, X_2 = 1)$. Hence the solutions for the subsystem are $(X_1 = 0, X_2 = 0)$ and $(X_1 = 1, X_2 = 1)$. Both valuations are fixed points for $X_2 = X_1 \vee X_2$, but $(X_1 = 0, X_2 = 0)$ is the least fixed point and (since $\sigma_2 = \mu$) the solution to $\mathcal{E}_3$. □

Nesting and Alternation in BES: We say that X_i *depends on* X_j if α_i contains a reference to X_j, or to X_k such that X_k depends on X_j. A BES is said to be *nested* if there are two variables X_i and X_j such that X_i depends on X_j and $\sigma_i \neq \sigma_j$. We say that X_i and X_j are *mutually dependent* if X_i depends on X_j and vice versa. A BES is *alternation free* if whenever X_i and X_j are mutually dependent, $\sigma_i = \sigma_j$. Otherwise, the BES is said to contain *alternating fixed points*. For instance, the BES $\mathcal{E}_1$ has no nested fixed points, $\mathcal{E}_2$ has nested fixed points but is alternation free, while $\mathcal{E}_3$ has alternating fixed points. Note that every BES that has alternating fixed points is also nested.

The order of equations in a nested BES is important, as the following example shows.

Example 2.4. Consider the BES $\mathcal{E}_4 = \langle X_1 = X_2 \vee X_1, X_2 = X_2 \wedge X_1 \rangle$ with sign map $\langle \mu, \nu \rangle$. Note that $\mathcal{E}_4$ differs from $\mathcal{E}_3$ only in the order in which the equations are defined (and the corresponding change variable names). Valuations $(X_1 = 0, X_2 = 0)$ and $(X_1 = 1, X_2 = 1)$ are solutions for the subsystem $X_1 = X_2 \vee X_1$; among these only $(X_1 = 1, X_2 = 1)$ is a fixed point for $X_2 = X_2 \wedge X_1$, and hence is the solution for $\mathcal{E}_4$. □

2.2 Boolean Equation Systems as Logic Programs

A *normal logic program* over a set of propositions $\mathcal{A}$ is a set of clauses of the form $\gamma \leftarrow \beta$ where $\gamma \in \mathcal{A}$ and β is a boolean formula in negation normal form over $\mathcal{A} \cup \{0, 1\}$. We use 1 and 0 to denote *true* and *false*, respectively. In a clause of the form $\gamma \leftarrow \beta$, γ is called the head of the clause and β its body. We use

p and q (possibly subscripted) to denote propositions and P to denote normal logic programs. A *definite logic program* is a program where every clause body is a positive boolean formula. We say that a proposition p *uses* another proposition q in a program if q appears in the body of a clause with p as the head; and p *negatively uses* q if q appears in the scope of a negation. A program is said to be *stratified* if no cycle in the transitive closure of the *uses* relation contains two literals p and q such that p negatively uses q.

We use stable models as the semantics of normal logic programs [8]. Stable models coincide with the standard least-model semantics for definite logic programs and the perfect-model semantics for stratified logic programs.

A BES consisting only of least fixed points (and hence not nested) can readily be seen as equivalent to a definite propositional logic program. We can thus use logic-program evaluation techniques to find the solution to such a BES.

Example 2.5. Consider the propositional program $P_1 = \{p_1 \leftarrow p_1 \wedge p_2,\ p_2 \leftarrow p_1 \vee p_2\}$. This program is equivalent to BES $\mathcal{E}_1$ where p_1 represents X_1 and p_2 represents X_2. The least model for P_1 is $\{\}$, from which we can derive $(X_1 = 0, X_2 = 0)$ as the solution for $\mathcal{E}_1$. □

For a nested but non-alternating BES we can construct a stratified propositional logic program such that the solution of the BES can be obtained from the perfect model of the logic program. This approach requires that greatest fixed-point equations be converted to least fixed-point equations using the equivalence $\nu X.\phi \equiv \neg\mu Z.\neg\phi[\neg Z/X]$ (see e.g. [12]). In fact, this is the strategy deployed by the XMC model checker.

Example 2.6. Consider the propositional program $P_2 = \{p_1 \leftarrow p_1 \wedge p_2.\ p_2 \leftarrow p_1 \vee p_2.\ p_3 \leftarrow \neg q_3.\ q_3 \leftarrow q_3 \vee \neg p_2.\}$. This program is equivalent to the BES $\mathcal{E}_2$ letting p_i represent X_i. The perfect model for P_2 is $\{q_3\}$ from which we can derive $(X_1 = 0, X_2 = 0, X_3 = 0)$ as the solution for $\mathcal{E}_2$. □

However, for a BES with alternating fixed points, this translation yields a non-stratified logic program which may not have a unique stable model.

Example 2.7. Consider the propositional program $P_3 = \{p_1 \leftarrow \neg q_1.\ q_1 \leftarrow q_1 \vee \neg p_2.\ p_2 \leftarrow p_1 \vee p_2.\}$. This program is equivalent to the BES $\mathcal{E}_3$ where p_i represents X_i. There are two stable models for P_3: $\{p_1, p_2\}$, and $\{q_1\}$ which yields two candidates $(X_1 = 1, X_2 = 1)$ and $(X_1 = 0, X_2 = 0)$ as solutions to $\mathcal{E}_3$. □

The problem with the translation from BESs to normal logic programs informally described by the above examples lies in the fact that a BES defines an ordered set (sequence) of equations, and the ordering is lost in the corresponding normal logic program. In fact it is easy to see that the program P_3 also corresponds to the BES $\mathcal{E}_4$ where p_2 represents X_1 and p_1 represents X_2: one of P_3's stable models corresponds to the solution of $\mathcal{E}_3$ and the other to the solution of $\mathcal{E}_4$.

At first sight, it appears that one can simply "select" the appropriate stable model by applying the order information *after* the stable-model computation. The following example illustrates why such an *a posteriori* selection will not work.

Example 2.8. Consider BES $\mathcal{E}_5 = \langle X_1 = X_2 \wedge X_3, X_2 = X_1 \wedge X_3, X_3 = X_2 \wedge X_3 \rangle$ with sign map $\langle \nu, \mu, \nu \rangle$. The corresponding logic program is $\{p_1 \leftarrow \neg q_1, q_1 \leftarrow \neg p_2 \vee q_3, p_2 \leftarrow \neg q_1 \wedge \neg q_3, p_3 \leftarrow \neg q_3, q_3 \leftarrow \neg p_2 \vee q_3\}$. The stable models for this program are $\{p_1, p_2, p_3\}$ and $\{q_1, q_3\}$, which correspond to solutions $v_1 = (X_1 = 1, X_2 = 1, X_3 = 1)$ and $v_2 = (X_1 = 0, X_2 = 0, X_3 = 0)$, respectively. Of these, v_1 assigns the value 1 to X_3 (the outermost variable) and appears to be minimal since $\sigma_3 = \nu$. However, the solution to BES $\mathcal{E}_5$ is v_2 since v_1 is not a solution to the subsystem $\langle X_1 = X_2 \wedge X_3, X_2 = X_1 \wedge X_3 \rangle$. □

Hence the "minimal" stable model may not correspond to the solution of a BES. Rather one must take into account the order information in the BES that is lost in the translation to logic programs. In Section 4 we define the notion of *preferred stable models* where information on ordering of literals is taken into account in the definition of the model itself.

3 Solutions to Boolean Equation Systems

Let $\chi = \{X_1, X_2, \ldots\}$ be the set of variables. The set of *positive boolean formulas* over χ is given by the following grammar:

$$\alpha \quad := \quad X_i \in \chi \;\mid\; \alpha_1 \wedge \alpha_2 \;\mid\; \alpha_1 \vee \alpha_2$$

A *valuation* v is a map $v : \chi \to \{0, 1\}$ with 0 standing for `false` and 1 for `true`. Let $\mathcal{V}$ denote the set of valuations. Given a positive boolean formula α and a valuation v, $\alpha[v]$ denotes the boolean value obtained by evaluating α using valuation v. Given a valuation v and a boolean value a, the valuation $v[a/X_i]$ is the valuation that returns the same value as v for all X_j other than X_i and returns a for X_i.

The solution of a boolean equation system ϕ, denoted by $[\![\phi]\!]$, is defined as a function that maps valuations to valuations. The mapping is such that $[\![\phi]\!](v)$ depends on v only for the free variables of ϕ. Thus, for a closed system $[\![\ ]\!]$ defines a constant function.

We first consider finding solutions to ϕ_i, where $\sigma_i = \mu$. Consider a function f parameterized by a valuation v defined as $f(v) = \lambda x.\alpha_i[v[x/X_i]]$. Since evaluating a formula w.r.t. a valuation results in a boolean value, $f(v)$ maps booleans to booleans. Now, the least fixed point of $f(v)$ (taken w.r.t. the natural partial order on $\{0, 1\}$ with 0 less than 1), denoted by $\mathtt{lfp}(f(v))$, gives the smallest value for X_i such that the fixed-point equation $\mu : X_i = \alpha_i$ holds for the given valuation v. Note that the value assigned by v to X_i is immaterial since $f(v)$ considers only $v[x/X_i]$. Let valuation v' be such that for all inner variables X_j, $j \leq i$, $v'(X_j)$ are the fixed points of the equations in ϕ_{i+1} when the outer variables X_k, $k > i$, are substituted by $v'(X_k)$. Then $\mathtt{lfp}(f(v'))$ is the fixed-point value of X_i corresponding to the values of the outer variables as specified by v'. Thus, $[\![\phi_i]\!](v)(X_i)$, is identical to $\mathtt{lfp}(f([\![\phi_{i+1}]\!](v)))$. The semantics of greatest fixed-point equations can be explained similarly.

The solution of a system ϕ is defined by induction on the size i of the system as follows:

$$\begin{array}{rcl} [\![\phi_0]\!](v) & = & v \\ [\![\phi_{i+1}]\!](v) & = & \begin{cases} [\![\phi_i]\!](v[\mathtt{lfp}(\xi_{i+1})/X_{i+1}]) \text{ if } \sigma_{i+1} = \mu \\ [\![\phi_i]\!](v[\mathtt{gfp}(\xi_{i+1})/X_{i+1}]) \text{ if } \sigma_{i+1} = \nu \end{cases} \quad i \geq 0 \\ & & \text{where } \xi_{i+1} = \lambda x.\alpha_{i+1}[[\![\phi_i]\!](v[x/X_{i+1}])] \end{array}$$

3.1 Solutions as Preferred Fixed Points

It is useful to treat the solution to a BES as the "minimum valuation" that satisfies the equations in the BES. We now formalize this notion. We define a family of partial orders $\sqsubseteq_i$, $i \geq 1$, on valuations that captures our intuition that least fixed-point variables take values as close to 0 as possible and greatest fixed-point variables take values as close to 1 as possible. Further, it also captures the idea that outer variables (i.e. variables with a higher index) have higher priority then inner variables (i.e. variables with a lower index).

We say that a valuation v is a *fixed point* of the system $\langle X_1 = \alpha_1, \ldots, X_n = \alpha_n \rangle$ if $v(X_i) = \alpha_i[v]$ for $1 \leq i \leq n$.

Definition 3.1 (Fixed Points of a BES). *The set of fixed points of a BES ϕ with respect to a valuation v, denoted by $FP(v)(\phi)$, is such that*

$$\begin{array}{rl} FP(v)(\phi_0) = v & \\ FP(v)(\phi_{i+1}) = \{u \mid u \in FP(v)(\phi_i) \text{ and } u(X_{i+1}) = \alpha_{i+1}[u]\} & \text{if } i \geq 0 \end{array}$$

Note that in the above definition, we ignore the signs of the equations. We now define a partial order on valuations based on the signs which is then used to select the preferred fixed point.

For a given sign map σ, we define the following partial orders: Let the partial order $\leq_i$ over $\{0, 1\}$ be defined as $0 \leq_i 1$ iff $\sigma_i = \mu$ and $1 \leq_i 0$ iff $\sigma_i = \nu$. The partial order $\sqsubseteq_i$ over valuations is defined by recursion over i as follows:

$$\begin{array}{rcl} u \sqsubseteq_1 v & \iff & u(X_1) \leq_1 v(X_1) \\ u \sqsubseteq_{i+1} v & \iff & u(X_{i+1}) <_{i+1} v(X_{i+1}) \text{ or } u(X_{i+1}) = v(X_{i+1}) \ \wedge \ u \sqsubseteq_i v \end{array}$$

We say that $u \sqsubset_i v$ if $u \sqsubseteq_i v$ and $u \neq v$. It is easy to see that $\sqsubseteq_i$ is a partial order on any set of valuations that agree at X_j for all $j > i$.

Definition 3.2 (Preferred Fixed Points). *The preferred fixed points of a BES ϕ with respect to a valuation v, denoted by $PFP(v)(\phi)$, is the set of valuation such that:*

$$\begin{array}{rcll} PFP(v)(\phi_0) & = & v & \\ PFP(v)(\phi_{i+1}) & = & & \\ \multicolumn{4}{c}{\min_{\sqsubseteq_{i+1}}(\{u \mid u \in PFP(v[u(X_{i+1})/X_{i+1}])(\phi_i) \cap FP(v)(\phi_{i+1})\}) \qquad \text{if } i \geq 0} \end{array}$$

Observe from the above definitions that $PFP(v)(\phi) \subseteq FP(v)(\phi)$. Moreover, $u(X_j) = v(X_j)$ for all $j > (i+1)$ for any $u \in FP(v)(\phi_i)$. Thus, $\sqsubseteq_{i+1}$ is a linear order (as it is a lexicographic order) on the set of valuations in $FP(v)(\phi_{i+1})$. This leads us to the following proposition:

Proposition 3.3. *For every boolean equation system ϕ and valuation v, there is a unique preferred fixed point, i.e., $|PFP(v)(\phi)| = 1$. In particular, when the system is closed, there is exactly one preferred fixed point.*

For the preferred fixed point to capture the solution of a given BES, the preference $\min_{\sqsubseteq}$ must be applied to a set of preferred fixed points of the inner equation. To see this, consider the following formula:

$$X_1 = X_2 \wedge X_3$$
$$X_2 = X_3 \wedge X_1$$
$$X_3 = X_2 \wedge X_3$$

with $\sigma_1 = \nu$, $\sigma_2 = \mu$ and $\sigma_3 = \nu$. It is easy to verify that the valuation v that assigns 1 to X_1, X_2 and X_3 is a fixed point, and is minimal w.r.t. $\sqsubseteq_3$. However, it is also easy to check that the solution to the BES assigns 0 to X_1, X_2 and X_3, and this is the preferred fixed point according to the definition given above.

Theorem 3.4. *Let ϕ be a BES of size n. Then, for all $i \leq n$, $[\![\phi_i]\!](v)$ is the preferred fixed point of ϕ_i w.r.t. v.*

The proof follows by an easy induction on i.

4 Preferred Stable Models of Normal Logic Programs

Let $P = \{p_1 \leftarrow \beta_1, p_2 \leftarrow \beta_2, \ldots, p_n \leftarrow \beta_n\}$ be a logic program. A proposition $p \notin \{p_1, p_2, \ldots, p_n\}$ is said to be *free* in P if there is some β such that p occurs in β_i.

We represent a model of a logic program by a substitution that maps propositions to truth values $\{0, 1\}$. We use w_0 to denote the substitution that maps all propositions to 0. Given a substitution w over propositions, we extend it to literals such that $w(\neg p) = \neg w(p)$ for every proposition p, where $\neg 0 = 1$ and $\neg 1 = 0$. Finally, given a program P and a substitution w, the program $P[w]$ is the one obtained by substituting all *free* propositions p in P by $w(p)$.

Definition 4.1 (Least Models for Definite Logic Programs). *The least model of a definite logic program P w.r.t. a substitution w on the free propositions of P is defined by the following equations:*

$$\begin{array}{rcl} M(\{\})(w) & = & w \\ M(\{p_i \leftarrow \beta_i\} \cup P')(w) & = & M(P')(w[b_i/p_i]) \\ & & \textit{where } b_i = \mathtt{lfp}(\lambda x.\beta_i[M(P')(w[x/p_i])]) \end{array}$$

The traditional least model of P under the closed world assumption is simply $M(P)(w_0)$.

We now recall the definition of stable model semantics for normal logic programs.

Definition 4.2 (Gelfond-Lifschitz Transformation [8]). *The Gelfond-Lifschitz transform of a propositional normal logic program P with respect to substitution w is a program $P \wr w$ obtained by replacing every negative literal of the form $\neg p$ in P by $\neg w(p)$.*

Note that for all P and w, $P \downarrow w$ is a definite logic program. A substitution w is a *stable model* of a program P iff it is the least model of $P \downarrow w$.

Definition 4.3 (Stable Models). *The set of stable models of a normal logic program P w.r.t. a substitution w on the free propositions of P, denoted by $SM(P)(w)$, is defined as:*

$$SM(P)(w) = \{u \mid u = M(P[w] \downarrow u)(w)\}$$

4.1 Preferred Stable Models

We now define stable models w.r.t. a *preference sequence*: a sequence $S = \langle l_1, l_2, \ldots, l_m \rangle$ of literals such that no proposition appears both positively and negatively in S. As usual, we represent by S_i the initial subsequence of S of length i. Given a substitution w mapping propositions to truth values, we extend w to literals with the usual interpretation that $w(\neg p) = \neg w(p)$ for some proposition p where $\neg 0 = 1$ and $\neg 1 = 0$.

Definition 4.4 (Preference Order $\sqsubseteq_S$). *Given two substitutions w_1 and w_2 and a preference sequence $S = \langle l_1, l_2, \ldots, l_m \rangle$, we say that w_2 is preferred over w_1 (written as $w_1 \sqsubseteq_S w_2$) if $w_1(l_m) < w_2(l_m)$, or $w_1(l_m) = w_2(l_m)$ and $w_1 \sqsubseteq_{S_{m-1}} w_2$. For an empty preference sequence $S = \langle\ \rangle$, $w_1 \sqsubseteq_S w_2$ for any pair of substitutions w_1 and w_2.*

Note that $\sqsubseteq_S$ defines a lexicographic order on substitutions, and hence is reflexive and transitive. Moreover, for any pair of substitutions w_1, w_2 that agree at all literals not in S, we have $w_1 \sqsubseteq_S w_2 \wedge w_2 \sqsubseteq_S w_1 \Rightarrow w_1 = w_2$. This means that for every set of substitutions that agree on all literals not in S, there is a unique minimum element w.r.t. $\sqsubseteq_S$. We denote this element by $\min_{\sqsubseteq_S}$.

Definition 4.5 (Preferred Stable Models). *The preferred stable model of a normal logic program P w.r.t. to a substitution w and a preference sequence S, denoted by $PSM_S(P)(w)$, is defined inductively on the size of P as follows:*

$$\begin{aligned} PSM_S(\{\})(w) &= w \\ PSM_S(\{p_i \leftarrow \beta_i\} \cup P')(w) &= \\ \min_{\sqsubseteq_S}(\{u \mid u \in PSM_S(P')(w[u(p_i)/p_i]) \cap SM(\{p_i \leftarrow \beta_i\} \cup P')(w)\}) & \end{aligned}$$

By $PSM_S(P)$ we denote the set of all preferred stable models w.r.t. arbitrary substitutions.

It is easy to show that the above definition is well-defined in the sense that the value of PSM_S is independent of the clause $\{p_i \leftarrow \beta_i\}$ selected for use in the recursive case.

A preference sequence S is said to be *complete* w.r.t. program P if every proposition in P appears (positively or negatively) in S. Every program that has at least one stable model w.r.t. to a substitution w has exactly one preferred stable model w.r.t. a complete preference sequence and w. Formally,

Proposition 4.6 (Uniqueness of Preferred Stable Models). *Let P be a normal logic program, w be a valuation, and S be a preference sequence that is complete w.r.t. P. Then $|PSM_S(P)(w)| \leq 1$. Moreover $|PSM_S(P)(w)| = 0$ iff $SM(P)(w) = \{\}$. In particular, for closed programs, $|PSM_S(P)| \leq 1$ and is 0 iff $SM(P) = \{\}$.*

5 Mapping Boolean Equation Systems to Propositional Logic Programs

In order to map BESs to logic programs, we first consider the mapping between the variables in a given BES ϕ and propositions in the corresponding logic program P. The logic program P we derive is over propositions $\{p_1, p_2, \ldots, q_1, q_2, \ldots\}$ such that each variable X_i in ϕ corresponds to literal p_i if $\sigma_i = \mu$ and to $\neg q_i$ if $\sigma_i = \nu$. The idea behind the mapping is to translate the equations into clauses in a normal logic program, considering greatest fixed points in terms of their dual least fixed points. The salient aspect of the mapping we define is that negation is used only where absolutely necessary: negation will be used only when variables of differing (fixed-point) signs are related. The following function $\mathcal{M}$ translates variables of a BES to propositions in the translated logic program:

$$\mathcal{M}(X_i) = \begin{cases} p_i & \text{if} \quad \sigma_i = \mu \\ \neg q_i & \text{if} \quad \sigma_i = \nu \end{cases}$$

We lift $\mathcal{M}$ to boolean expressions by replacing the variables in a given expression using the above definition. In order to translate greatest fixed-point equations, we need to find the dual of the equation. This is done by constructing $\overline{\alpha}$, which is $\neg\alpha$ to negation normal form. When we apply $\mathcal{M}$ to boolean expressions with negation, we also reduce expressions of the form $\neg\neg p$ to p.

Definition 5.1. *The translation function $\mathcal{P}$ maps boolean equation systems to normal logic programs such that $\mathcal{P}(\phi)$ for a boolean equation system ϕ is given by*

$$\mathcal{P}(\phi_0) = \{\}$$
$$\mathcal{P}(\phi_{i+1}) = \begin{cases} \{p_{i+1} \leftarrow \mathcal{M}(\alpha_{i+1})\} \cup \mathcal{P}(\phi_i) & \textit{if} \quad \sigma_{i+1} = \mu \\ \{q_{i+1} \leftarrow \mathcal{M}(\overline{\alpha_{i+1}})\} \cup \mathcal{P}(\phi_i) & \textit{if} \quad \sigma_{i+1} = \nu \end{cases} \qquad \textit{for} \quad i \geq 0$$

Note that for each X_i in ϕ there is exactly one proposition in $\mathcal{P}(\phi)$. Observe that if X_k appears in α_i then the literal corresponding to X_k appears in negated form in the program clause corresponding to α_i if and only if $\sigma_i \neq \sigma_k$. Thus, negative dependencies are introduced in $\mathcal{P}(\phi)$ only if the corresponding variables in ϕ differ in sign. In an alternation-free BES, the dependency between opposite-signed variables is cycle-free. Hence we have the following proposition:

Proposition 5.2. *If ϕ is an alternation-free boolean equation system then $\mathcal{P}(\phi)$ is a stratified logic program.*

Stratified logic programs have unique stable models which can be evaluated in polynomial time. Thus, the logic programs generated from alternation-free BESs can be efficiently evaluated.

We complete the translation by defining a mapping between sign maps of BESs and preference sequences for the corresponding logic programs.

Definition 5.3. *The translation function $\mathcal{P}$ maps the sign map σ of a BES of size n to the preference sequence $\langle l_1, l_2, \ldots, l_n \rangle$ such that for all $1 \leq i \leq n$:*

$$l_i = \begin{cases} p_i & \text{if} \quad \sigma_i = \mu \\ \neg q_i & \text{if} \quad \sigma_i = \nu \end{cases}$$

We have overloaded the symbol $\mathcal{P}$ to denote the translation functions that map different aspects of the BES to logic programs with preferences. Note that $\mathcal{P}(\sigma)$ is a complete preference sequence whenever σ is a sign map of a closed BES.

6 Solutions to Boolean Equation Systems Are Preferred Stable Models

We now show that given a closed BES ϕ of size n, its solution can be obtained from the preferred stable model of $\mathcal{P}(\phi)$. This is done by showing that

(i) the preference order among fixed points in a BES corresponds to the order imposed by preference sequences,
(ii) every stable model of $\mathcal{P}(\phi)$ is a fixed point of ϕ, and
(iii) the preferred fixed point of ϕ is a stable model of $\mathcal{P}(\phi)$.

We first formalize the relationship between the valuations of a BES and the models of the corresponding logic program. Given a valuation v over $\chi = \{X_1, X_2, \ldots\}$, let $\mathcal{P}(v)$ be a substitution over $\mathcal{A} = \{p_1, p_2, \ldots, q_1, q_2, \ldots\}$ such that for all $i \geq 0$, $\mathcal{P}(v)(p_i) = v(X_i)$ and $\mathcal{P}(v)(q_i) = \neg v(X_i)$. Similarly, for any substitution w over $\mathcal{A}$ satisfying $w(p_i) = \neg w(q_i)$ for every i, we write $\mathcal{P}^{-1}(w)$ to denote the valuation over χ such that $\mathcal{P}^{-1}(w)(X_k) = w(p_k)$.

The correspondence between valuations and substitutions, and between sign maps and preference sequences, is formalized in the following theorem:

Theorem 6.1. *Let ϕ be a closed BES of size n with sign map σ and let $S = \mathcal{P}(\sigma)$ be a preference sequence. For any pair of valuations v_1, v_2 over $\{X_1, X_2, \ldots, X_n\}$, $v_1 \sqsubseteq_n v_2$ iff $\mathcal{P}(v_1) \sqsubseteq_S \mathcal{P}(v_2)$.*

The following theorem formally states the second step needed for establishing the correspondence between preferred fixed points and preferred stable models.

Theorem 6.2. *Let ϕ be a BES of size n and v be a valuation. If u is a stable model of $\mathcal{P}(\phi_n)[\mathcal{P}(v)]$ then $\mathcal{P}^{-1}(u)$ is a fixed point of ϕ w.r.t. the valuation v.*

The proof follows from the definition of stable models and the translation mapping $\mathcal{P}$.

The third step would be trivial if the converse of Theorem 6.2 were true. It turns out, however, that not all fixed points of a BES correspond to stable models of the translated program. This mismatch arises because the definition of fixed points ignores the signs of the equations as well as the order of nesting, while the translation from BES to logic programs ignores only the order of nesting.

Thus even for non-nested BESs the set of fixed points may be larger than the set of stable models of the corresponding program. For example, consider the BES ϕ with equations $\langle X_1 = X_2, X_2 = X_1\rangle$ and sign map $\langle \mu, \mu\rangle$. The BES has two fixed points: $v_1 = (X_1 = 0, X_2 = 0)$ and $v_2 = (X_1 = 1, X_2 = 1)$, with v_1 as the solution. The program $\mathcal{P}(\phi)$ is $\{p_1 \leftarrow p_2, p_2 \leftarrow p_1\}$, and has only one stable model $\{\}$. Similarly the system $\mathcal{E}_3$ in Example 2.3 (Section 2) has three fixed points but of these only two ($(X_1 = 0, X_2 = 0)$ and $(X_1 = 1, X_2 = 1)$) correspond to stable models ($\{q_1\}$ and $\{p_1, p_2\}$, respectively) of the translated program.

Thus, we need to show that we have not "lost" the solution to a BES in the translation, as formally stated by the following theorem.

Theorem 6.3. *Let ϕ be a closed BES of size n and v be a valuation. Then, for all n, $\mathcal{P}([\![\phi_n]\!](v))$ is a stable model of $\mathcal{P}(\phi_n)[\mathcal{P}(v)]$.*

The proof is by induction on the size of the BES ϕ. The proof of the base case relies on the fact that, when $\sigma_1 = \mu$, p_1 does not appear in negative literals in $\mathcal{M}(\alpha_1)$, and hence $\mathcal{P}(\phi_1)[\mathcal{P}(v)] \wr \mathcal{P}([\![\phi_1]\!](v)) = \{(p_1 \leftarrow \mathcal{M}(\alpha_1))\}[\mathcal{P}(v)]$; symmetrically, when $\sigma_1 = \nu$, q_1 does not appear in negative literals in $\mathcal{M}(\overline{\alpha_1})$, and hence $\mathcal{P}(\phi_1)[\mathcal{P}(v)] \wr \mathcal{P}([\![\phi_1]\!](v)) = \{(q_1 \leftarrow \mathcal{M}(\overline{\alpha_1}))[\mathcal{P}(v)]\}$. The difficulty in the induction step arises from the following: the definition of the stable model allows for substitution of values only for the negative literals. Consequently, the syntax of the resulting program is such that one cannot directly apply the induction hypothesis. To get around this, we "approximate" this program (from "below" in the μ case and from "above" in the ν case) by one where the hypothesis is applicable. The details are available in [10].

Thus, the stable models of $\mathcal{P}(\phi)$ correspond (via the translation function $\mathcal{P}$ over valuations) to a subset of the set of fixed points of ϕ that contains the preferred fixed point of ϕ. Since preference orders over valuations and substitutions coincide (from Theorem 6.1) it is easy to establish from Definition 4.5 that the preferred stable model of $\mathcal{P}(\phi)$ corresponds to the preferred fixed point of ϕ.

Corollary 6.4. *Let ϕ be a closed BES with sign map σ and let $S = \mathcal{P}(\sigma)$. Then $PFP(\phi) = \{v\}$ and $PSM_S(\mathcal{P}(\phi)) = \{w\}$ such that $v = \mathcal{P}^{-1}(w)$.*

7 Conclusions

We have shown how to compute alternating fixed points of boolean equation systems by translating a given equation system ϕ into a propositional normal logic program $\mathcal{P}(\phi)$, and computing the preferred stable model of $\mathcal{P}(\phi)$.

Our results provide the basis for extending the XMC model checker, which currently handles only the alternation-free fragment of the modal mu-calculus, to the full mu-calculus. XMC casts model checking as a query-evaluation problem over logic programs with stratified negation and uses the XSB logic-programming system [20] for the actual evaluation. For formulas with alternating fixed points, the resulting logic program may be non-stratified. XSB, while computing the

well-founded model for such a program, produces a residual program that effectively summarizes the cycles with negation in the original program. We can then evaluate the preferred stable model of the residual program using the stable models generator of [13] or the DLV system [6] as the core engine. The answer to the original model-checking question can be directly obtained from the model so computed. Experimental results have shown that the residual programs so derived are typically small and the preferred stable model can be calculated efficiently [11].

We have also shown that for alternation-free boolean equation systems, the logic programs we derive are stratified. Consequently, our mapping of boolean equation systems to logic programs preserves the linear-time complexity of evaluating solutions of such equation systems established in [5]. We moreover conjecture that for BESs with alternating fixed points, time complexity exponential in the "alternation depth" of the equation system can be attained, again matching the best upper bound known to date. This result would depend critically on the use of the Gelfond-Lifschitz transformation to steer the computation of the preferred stable models of the non-stratified logic programs that our translation produces in the case of alternating fixed points.

Acknowledgements

We would like to thank the anonymous referees for their valuable comments and suggestions.

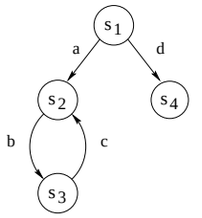
References

1. G. Brewka and T. Eiter. Preferred answer sets for extended logic programs. *Artificial Intelligence*, 109(1–2):297–356.
2. E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In D. Kozen, editor, *Proceedings of the Workshop on Logic of Programs,* Yorktown Heights, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer Verlag, 1981.
3. E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM TOPLAS*, 8(2), 1986.
4. E. M. Clarke and J. M. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28(4), December 1996.
5. R. Cleaveland and B. U. Steffen. A linear-time model checking algorithm for the alternation-free modal mu-calculus. *Formal Methods in System Design*, 2:121–147, 1993.
6. DLV The DLV project – a disjunctive datalog system (and more). Available at: `http://www.dbai.tuwien.ac.at/proj/dlv/` since 1996.
7. G. Delzanno and A. Podelski. Model checking in CLP. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 223–239, 1999.
8. M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *International Conference on Logic Programming*, pages 1070–1080, 1988.

9. D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, 1983.
10. K. Narayan Kumar, C.R. Ramakrishnan, and S. A. Smolka. Alternating fixed points in boolean equation systems as preferred stable models. Technical report, Department of Computer Science, SUNY at Stony Brook, Stony Brook, New York, 2001. Available from: `http://www.cs.sunysb.edu/~cram/papers`.
11. Xinxin Liu, C. R. Ramakrishnan, and Scott A. Smolka. Fully local and efficient model checking of alternating fixed points. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1384 of *Lecture Notes in Computer Science*, pages 56–70. Springer Verlag, 1998.
12. A. Mader. *Verification of Modal Properties using Boolean Equations*. PhD thesis, Technical University of Munich, 1997.
13. I. Niemela and P. Simons. Efficient implementation of the well-founded and stable model semantics. In *Joint International Conference and Symposium on Logic Programming*, pages 289–303, 1996.
14. V. R. Pratt. A decidable mu-calculus. In *Proceedings of the 22nd IEEE Ann. Symp. on Foundations of Computer Science,* Nashville, Tennessee, pages 421–427, 1981.
15. J. P. Queille and J. Sifakis. Specification and verification of concurrent systems in Cesar. In *Proceedings of the International Symposium in Programming*, volume 137 of *Lecture Notes in Computer Science*, Berlin, 1982. Springer-Verlag.
16. Y. S. Ramakrishna, C. R. Ramakrishnan, I. V. Ramakrishnan, S. A. Smolka, T. L. Swift, and D. S. Warren. Efficient model checking using tabled resolution. In *Proceedings of the 9th International Conference on Computer-Aided Verification (CAV '97)*, Haifa, Israel, July 1997. Springer-Verlag.
17. C.R. Ramakrishnan, I.V. Ramakrishnan, S.A. Smolka, Y. Dong, X. Du, A. Roychoudhury, and V.N. Venkatakrishnan. XMC: A logic-programming-based verification toolset. In *Computer Aided Verification (CAV)*, 2000.
18. C. Sakama and K. Inoue. Representing priorities in logic programs. In *Joint International Conference/Symposium on Logic Programming*, pages 82–96, 1996.
19. B. Vergauwen and J. Lewi. Efficient local correctness checking for single and alternating boolean equation systems. In *Proceedings of ICALP'94*, pages 304–315. LNCS 820, 1994.
20. XSB. The XSB tabled logic programming system. Available from `http://xsb.sourceforge.net`.

A Modal Mu-Calculus and Boolean Equation Systems

Formulas in the modal mu-calculus are constructed from existential (denoted by $\langle\cdot\rangle$) and universal (denoted by $[\cdot]$) modalities; explicit greatest and least fixed-point operators (denoted by ν and μ, respectively); formula variables that index the fixed points; and the traditional conjunction/disjunction operators and constants *true* and *false* from classical logic. Models of mu-calculus formulas are given in terms of sets of vertices (called states) of an edge-labeled graph called a labeled transition system (LTS). For example, the formula $\nu X.\langle - \rangle true \wedge [-]X$ characterizes deadlock freedom: a state s that models X is such that it is possible to make a transition from s (i.e., the meaning of $\langle - \rangle true$) and every destination state reached models X (the meaning of $[-]X$).



(a)

$\nu\, X_1 = X_2 \wedge X_4$
$\nu\, X_2 = X_3$
$\nu\, X_3 = X_2$
$\nu\, X_4 = \mathit{false}$

(c)

$\nu X.\langle - \rangle \mathit{true} \wedge [-]X$

(b)

$X_1 = \mathit{false}$
$X_2 = \mathit{true}$
$X_3 = \mathit{true}$
$X_4 = \mathit{false}$

(d)

Fig. 1. Example Labeled Transition System (a), mu-calculus formula for deadlock freedom (b), corresponding Boolean Equation System (c), and its solution (d).

Fixed points in a mu-calculus formula may be nested: i.e., a fixed-point formula ϕ_1 may occur in the scope of another fixed point formula ϕ_2. We then say that the outer formula ϕ_2 depends on the inner formula ϕ_1. Moreover, the inner fixed-point formula ϕ_1 may refer to the variable indexing the outer fixed point ϕ_2, thereby making ϕ_1 and ϕ_2 mutually dependent. Formulas where the mutually dependent fixed points have different fixed-point operators (i.e., μ and ν) are called *alternating* fixed-point formulas. An example of an alternating fixed-point formula is:

$$\nu X.\mu X'.[a]X \wedge [-a]X'$$

which expresses the property that transitions labeled a occur infinitely often along every infinite path of the system.

The problem of determining the model of a modal mu-calculus formula w.r.t. a given LTS can be reduced to solving boolean equation systems (BESs). For example, consider the LTS of Figure 1(a) and the formula for deadlock freedom. To determine the formula's model, it suffices to solve the BES of Figure 1(c): each variable X_i indicates whether or not state s_i of the LTS is in the model. The solution, given in Figure 1(d), reflects the fact that states s_2 and s_3 are in the model while s_1 and s_4 are not.

Fages' Theorem for Programs with Nested Expressions

Esra Erdem and Vladimir Lifschitz

Department of Computer Sciences
University of Texas at Austin
Austin, TX 78712, USA
{esra,vl}@cs.utexas.edu

Abstract. We extend a theorem by François Fages about the relationship between the completion semantics and the answer set semantics of logic programs to a class of programs with nested expressions permitted in the bodies of rules. Fages' theorem is important from the perspective of answer set programming: whenever the two semantics are equivalent, answer sets can be computed by propositional solvers, such as SATO, instead of answer set solvers, such as SMODELS. The need to extend Fages' theorem to programs with nested expressions is related to the use of choice rules in the input language of SMODELS.

1 Introduction

This note continues the line of research on the relationship between two theories of negation as failure—one based on program completion [4], the other based on stable models, or answer sets [9]. A syntactic condition that guarantees the equivalence of these two concepts was given by Fages [7]. Babovich, Erdem and Lifschitz [2] generalized Fages' theorem and showed that results of this type can be applied to computing answer sets for a logic program: whenever the two semantics are equivalent, answer sets can be computed by propositional solvers, such as SATO [16] or RELSAT [3], instead of answer set solvers, such as SMODELS [13] and DLV [6].

This possibility is important from the perspective of answer set programming. The idea of this programming method is to reduce a given computational problem to computing an answer set for a logic program. Examples and references can be found in [11].

Lloyd and Topor [12] generalized the completion semantics to programs containing nested expressions (formulas) in the bodies of rules. A similar generalization of the answer set semantics was proposed in [10]. In this note we show how Fages' theorem on the relationship between completion and answer sets can be extended to these more general programs. We argue that this extension of Fages' theorem is important in connection with some of the syntactic constructs available in the input language of system SMODELS.

P. Codognet (Ed.): ICLP 2001, LNCS 2237, pp. 242–254, 2001.

Consider a simple example. Program

$$\begin{array}{l} p \leftarrow \mathit{not}\ \mathit{not}\ p, \\ p \leftarrow p, q \end{array} \tag{1}$$

contains nested occurrences of negation as failure in the body of the first rule.[1] It belongs to the syntactic class for which our theorem guarantees the equivalence of the answer set semantics to the completion semantics. This program has two answer sets $\emptyset$, $\{p\}$; they are identical to the models of the completion of (1):

$$\begin{array}{l} p \equiv \neg\neg p \vee (p \wedge q), \\ q \equiv \bot. \end{array} \tag{2}$$

In the syntax of SMODELS, the first of rules (1) can be written as a "choice rule" $\{p\}$. The relationship between rules like this and nested expressions is systematically studied in [8].

In the next section, we discuss a more interesting example of this kind—a formalization of the 8-queens problem in the language of SMODELS. Section 3 is a review of answer sets and completion for programs with nested expressions. After extending Fages' syntactic condition—"tightness"—to programs of this kind, we introduce our generalization of Fages' theorem (Section 4). The proof is expressed in terms of a generalization of the model-theoretic counterpart of completion introduced in [1], called supportedness (Sections 5, 6).

2 The 8-Queens Problem

In the 8-queens problem, the goal is to find a configuration of 8 queens on an 8×8 chessboard such that no queen can be taken by any other queen. In other words, no two queens may be on the same row, on the same column, or on the same diagonal.

The 8-queens problem can be presented to SMODELS as in Figure 1. The choice rule that follows the definitions of *row* and *column* instructs SMODELS to select atoms of the form $\mathit{occupied}(R, C)$ for including in an answer set in such a way that, for every column C, exactly one atom $\mathit{occupied}(R, C)$ be selected. The program has 92 answer sets, corresponding to all possible arrangements of 8 queens. Given this input file, SMODELS produces one of the solutions:

```
Stable Model: occupied(4,1) occupied(2,2) occupied(7,3)
occupied(5,4) occupied(1,5) occupied(8,6) occupied(6,7)
occupied(3,8)
```

[1] The double negation in the first rule of (1) is redundant from the point of view of the completion semantics, but it does affect the program's answer sets. On the other hand, the second rule is redundant from the point of view of the answer set semantics, but, generally, dropping a rule like this can change a program's completion in an essential way.

```
row(1..8).
column(1..8).

1{occupied(R,C):row(R)}1 :- column(C).

:- occupied(R,C), occupied(R,C1),
   row(R), column(C), column(C1), C < C1.

:- occupied(R,C), occupied(R1,C1),
   row(R), column(C), row(R1), column(C1),
   C < C1, abs(R - R1) == abs(C - C1).
```

Fig. 1. The 8-queens problem presented to SMODELS

Consider the grounded version of the program in Figure 1, with the domain predicates *row* and *column* dropped from the rules. It consists of the rules

$$1\{\mathit{occupied}(1,C),\dots,\mathit{occupied}(8,C)\}1 \tag{3}$$

for all C in $\{1,\dots,8\}$,

$$\leftarrow \mathit{occupied}(R,C),\mathit{occupied}(R,C1) \tag{4}$$

for all $R,C,C1$ in $\{1,\dots,8\}$ such that $C<C1$, and

$$\leftarrow \mathit{occupied}(R,C),\mathit{occupied}(R1,C1) \tag{5}$$

for all $R,R1,C,C1$ in $\{1,\dots,8\}$ such that $C<C1$ and $|R-R1|=|C-C1|$. Rewritten in terms of nested expressions, as described in [8, Section 6], rule (3) becomes

$$\begin{array}{l}\mathit{occupied}(R,C) \leftarrow \mathit{not\ not\ occupied}(R,C)\\ \leftarrow \mathit{occupied}(R,C),\mathit{occupied}(R1,C) \qquad (R<R1)\\ \leftarrow \mathit{not\ occupied}(1,C),\dots,\mathit{not\ occupied}(8,C).\end{array} \tag{6}$$

The first of these rules allows each of the atoms $\mathit{occupied}(R,C)$ to be included or not included in an answer set arbitrarily. The second rule prohibits the selections that include more than one atom $\mathit{occupied}(R,C)$ with the same value of C. The third rule prohibits the selections that include no such atoms for some value of C.

To sum up, the program from Figure 1 can be rewritten as the union of programs (6), (4) and (5).

The results of this paper show that the answer sets for the program consisting of these rules can be equivalently described as the models of the program's completion. Consequently, an answer set for this program can be found by the Causal Calculator (CCALC)[2] — a system that can compute the completion of a program, even if the program contains nested expressions, and then can call a propositional solver to find a model of the completion.

[2] `http://www.cs.utexas.edu/users/tag/cc/` .

We present the union of programs (6), (4) and (5) to CCALC as shown in Figure 2.[3] CCALC produces the following output, using SATO to find a model:

```
Satisfying Interpretation:  occupied(1,5)  occupied(2,8)
occupied(3,4) occupied(4,1)  occupied(5,3)  occupied(6,6)
occupied(7,2)  occupied(8,7)
```

```
:- sorts
        row; column.

:- variables
        R,R1 :: row;
        C,C1 :: column.

:- constants
        1..8 :: row;
        1..8 :: column;
        occupied(row,column) :: cwAtomicFormula.

occupied(R,C) :- not (not occupied(R,C)).

:- occupied(R,C),occupied(R1,C), R < R1.

:- (/\R: -occupied(R,C)).

:- occupied(R,C), occupied(R,C1), C < C1.

:- occupied(R,C), occupied(R1,C1),
   C < C1, abs(R-R1) =:= abs(C-C1).
```

Fig. 2. The 8-queens problem presented to CCALC

In our experiments, SMODELS took 0.06 seconds to find a solution to the 8-queens problem, and SATO took 0.01 seconds. For 20 queens, the run time of SMODELS was 200 seconds, and the run time of SATO was 0.08 seconds. For 30 queens, SMODELS did not terminate after many hours of search; the run time of SATO was only 0.29 seconds.[4] This example shows that, given a representation of a computational problem in the input language of SMODELS, it is sometimes

[3] In CCALC, sorts of variables correspond to domain predicates of SMODELS. The symbol `cwAtomicFormula` in the declaration of `occupied` means "an atomic formula satisfying the closed world assumption".

[4] We used SMODELS 2.26 with LPARSE 0.99.59, and CCALC 1.9 with SATO 3.2. The times do not include the preprocessing done by LPARSE for SMODELS and by CCALC for SATO. The constraint logic programming system CLP [15] is computationally even more efficient in application to the n-queens problem: it takes 0.01 seconds to find a solution for 20 queens. According to [14], this problem can be also solved quickly using the abductive logic programming system SLDNFAC [5].

faster to find a solution by running a propositional solver on the completion of the corresponding program with nested expressions than by using SMODELS itself.

3 Programs

The words *atom* and *literal* are understood here as in propositional logic; we call the sign $\neg$ in a negative literal $\neg A$ *classical negation* to distinguish from the symbol for negation as failure (*not*). *Elementary formulas* are literals and the 0-place connectives $\bot$ and $\top$. *Formulas* are built from elementary formulas using the unary connective *not* and the binary connectives , (conjunction) and ; (disjunction). A *(nondisjunctive) rule* is an expression of the form

$$Head \leftarrow Body \tag{7}$$

where *Head* is a literal or $\bot$, and *Body* is a formula.[5] If $Head = \bot$, we will sometimes drop $\bot$ from the head; a rule with $Head = \bot$ is called a *constraint*.

A *(nondisjunctive) program* is a set of rules.

We define when a consistent set X of literals *satisfies* a formula F (symbolically, $X \models F$) recursively, as follows:

- for elementary F, $X \models F$ if $F \in X$ or $F = \top$,
- $X \models not\ F$ if $X \not\models F$,
- $X \models (F, G)$ if $X \models F$ and $X \models G$,
- $X \models (F; G)$ if $X \models F$ or $X \models G$.

A consistent set X of literals is *closed under* Π if, for every rule (7) in Π, $Head \in X$ whenever $X \models Body$.

Let Π be a program without negation as failure. We say that X is an *answer set* for Π if X is minimal among the consistent sets of literals closed under Π. It is easy to see that there can be at most one such set. For instance, the answer set for the program

$$\begin{array}{l} p \leftarrow \top \\ p \leftarrow p, q \end{array} \tag{8}$$

is $\{p\}$.

The *reduct* Π^X of a program Π relative to a set X of literals is obtained from Π by replacing every maximal occurrence of a formula of the form *not* F in Π (that is, every occurrence of *not* F that is not in the range of another *not*) with $\bot$ if $X \models F$, and with $\top$ otherwise.[6] A consistent set X of literals is an *answer set* for Π if it is the answer set for the reduct Π^X. For instance, $\{p\}$ is an answer set for (1) since it is an answer set for the reduct (8) of (1) relative to $\{p\}$.

[5] In [10], the syntax of rules is more general: the head may be an arbitrary formula, in particular a disjunction.

[6] This definition is equivalent to the recursive definition of the reduct given in [10].

We say that a formula (or a program) is *normal* if it does not contain classical negation. A normal formula F can be identified with the propositional formula obtained from F by replacing every comma with $\wedge$, every semicolon with $\vee$, and *not* with $\neg$.

Consider a finite normal program Π. The completion of Π is defined as follows. If A is an atom or the symbol $\bot$, by $Comp(\Pi, A)$ we denote the propositional formula

$$A \equiv (Body_1 \vee \ldots \vee Body_k) \tag{9}$$

where the disjunction extends over all rules

$$A \leftarrow Body_i \tag{10}$$

in Π with the head A.[7] The *completion* of Π is the set of formulas $Comp(\Pi, A)$ for all A. For instance, the bodies of rules (1), written as propositional formulas are $\neg\neg p$ and $p \wedge q$; for this program Π, formulas (2) are $Comp(\Pi, p)$ and $Comp(\Pi, q)$. In addition to these two formulas, the completion of this program includes also $Comp(\Pi, \bot)$, which is the tautology $\bot \equiv \bot$.

As in the case of normal programs without nesting, an answer set for any normal program Π satisfies the completion of Π. In the next section we will see under what conditions the converse is true.

4 A Generalization of Fages' Theorem

Although the theorem below applies to normal programs only, it is convenient to define tightness for programs that may contain classical negation.

A formula is called *negative* if every occurrence of an atom in this formula is in the scope of a negation as failure.

We consider nondisjunctive programs consisting of rules of the form

$$Head \leftarrow P, N \tag{11}$$

where P is an arbitrary formula, and N is a negative formula. In applications, P will be usually a formula that does not contain negation as failure. Any nondisjunctive program can be turned into a program of this kind using the "equivalent transformations" discussed in [10, Section 4]. For instance, the rule

$$p \leftarrow q; not\ r$$

does not have the form (11), but it can be equivalently replaced by the pair of rules

$$\begin{array}{l} p \leftarrow q, \top \\ p \leftarrow \top, not\ r. \end{array} \tag{12}$$

In fact, some disjunctive programs can be converted to this special form as well; for instance,

$$p; not\ q \leftarrow r$$

[7] This is essentially the definition from [12] restricted to the propositional case.

can be equivalently replaced by

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

([10], Proposition 6(iii)).

An occurrence of a formula F in a formula is *singular* if the symbol before this occurrence is $\neg$; otherwise, the occurrence is *regular*. It is clear that the occurrence of F can be singular only if F is an atom. For any formula G, by $\mathit{lit}(G)$ we denote the set of all literals having regular occurrences in G. For instance, $\mathit{lit}(p; \mathit{not}\ \neg r) = \{p, \neg r\}$.

Now we are ready to show how Fages' syntactic condition is extended to programs with nested expressions.

A program Π whose rules have the form (11) is *tight* on a set X of literals if there exists a function λ from X to ordinals satisfying the following condition:

(*) for every rule (11) in Π, if $\mathit{Head} \in X$ and $X \models (P, N)$ then, for all $L \in X \cap \mathit{lit}(P)$, $\lambda(L) < \lambda(\mathit{Head})$.

For instance, program (1) written as

$$p \leftarrow \top, \mathit{not}\ \mathit{not}\ p,$$
$$p \leftarrow (p, q), \top$$

is tight on $\{p\}$: take $\lambda(p) = 0$. Program (12) is tight on $\{p, q, r\}$: take $\lambda(p) = 1$, $\lambda(q) = \lambda(r) = 0$.

If the program is finite then, without loss of generality, the values of λ can be assumed to be finite.

For programs whose rules are of the form

$$\mathit{Head} \leftarrow L_1, \ldots, L_m, \mathit{not}\ L_{m+1}, \ldots, \mathit{not}\ L_n \tag{13}$$

where each L_i is a literal, the definition of tightness above is slightly more general than the definition given in [2]. If we take P to be $L_1, \ldots, L_m$ and N to be $\mathit{not}\ L_{m+1}, \ldots, \mathit{not}\ L_n$ then the condition $X \models (P, N)$ can be written as

$$L_1, \ldots, L_m \in X,\ L_{m+1}, \ldots, L_n \notin X,$$

and it implies that $X \cap \mathit{lit}(P) = \{L_1, \ldots, L_m\}$. In [2], the restriction $L_{m+1}, \ldots, L_n \notin X$ is not included in the definition of tightness.[8] For instance, the program

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

is tight on $\{p, q\}$ in the sense of (*), but it is not tight in the sense of [2]. Note that $\{p, q\}$ is both a model of the completion of this program and its answer set.

Our generalization of Fages' theorem is stated as follows:

[8] The possibility of including this restriction was suggested to us by Hudson Turner on May 2, 2001.

Theorem 1 *For any finite normal program Π whose rules have the form (11) and any set X of atoms such that Π is tight on X, X is an answer set for Π iff X satisfies the completion of Π.*

For instance, program (1) is tight on the models $\emptyset$, $\{p\}$ of its completion (2). By the theorem above, it follows that $\emptyset$ and $\{p\}$ are the answer sets for (1).

Theorem 1 is more general than Fages' original theorem [7] in three ways. First, we defined tightness relative to a set X; Fages' definition corresponds to the special case where X is the set of all atoms. Second, Fages' definition is limited to rules of form (13). Third, it does not include the improvement mentioned in footnote ([8]).

In the modification of Theorem 1 stated below, the rules of Π are not required to have the form (11), and tightness is replaced by a simpler condition. We say that a literal L is a *positive body element* of a nondisjunctive program Π if Π contains a rule $Head \leftarrow Body$, with $Head \neq \bot$, such that $Body$ contains a regular occurrence of L that is not in the scope of negation as failure. We denote by $pos(\Pi)$ the set of positive body elements of Π. For instance, if Π is (1) then $pos(\Pi) = \{p, q\}$; if Π is (4)–(6) then $pos(\Pi) = \emptyset$.

Theorem 2 *For any finite normal program Π and any set X of atoms disjoint from $pos(\Pi)$, X is an answer set for Π iff X satisfies the completion of Π.*

For instance, this theorem shows that all models of the completion of the 8-queens program (4)–(6) are answer sets.

Theorems 1 and 2 are proved in the next two sections.

5 Supported Sets

The proof of Theorem 1 below is expressed in terms of a model-theoretic counterpart of completion—supportedness.

We say that a set X of literals is *supported by* a program Π if, for every literal $L \in X$, there exists a rule (7) in Π such that $Head = L$ and $X \models Body$. In application to finite normal programs, the combination of closure and supportedness exactly corresponds to the program's completion:

Proposition 1 *For any finite nondisjunctive normal program Π, a set of atoms satisfies the completion of Π iff it is closed under and supported by Π.*

Proof. Let Π be a finite nondisjunctive normal program. Recall that the completion of Π consists of equivalences (9) where A is an atom or the symbol $\bot$. It is clear that a set X satisfies the completion of Π iff, for each A,

(a) for every rule (10) in Π with the head A, if $X \models Body_i$ then $A \in X$, and
(b) if $A \in X$ then $X \models Body_i$ for some rule (10) in Π with the head A.

(Given the convention about writing normal formulas in the syntax of propositional logic introduced in Section 3, the relation $X \models F$ restricted to normal formulas F is equivalent to the satisfaction relation of classical logic.) Condition (a) expresses that X is closed under Π, and condition (b) expresses that X is supported by Π.

An answer set for a program Π is closed under and supported by Π:

Proposition 2 *For any nondisjunctive program Π and any consistent set X of literals, if X is an answer set for Π then X is closed under and supported by Π.*

Under the tightness condition, the converse holds as well—the sets closed under and supported by Π are answer sets for Π:

Proposition 3 *For any program Π whose rules have the form (11) and any consistent set X of literals such that Π is tight on X, if X is closed under and supported by Π then X is an answer set for Π.*

Propositions 2 and 3 generalize Theorem 1 to programs that may contain classical negation and may consist of infinitely many rules. The proofs of these propositions are given in the next section. Theorem 1 is an immediate consequence of Propositions 1–3.

6 Proofs

Lemma 1 *Given a formula F without negation as failure and two sets Z, Z' of literals such that $Z' \subseteq Z$, if $Z' \models F$ then $Z \models F$.*

Proof. Immediate by structural induction.

The following lemma is the special case of Proposition 2 in which Π is assumed to be a program without negation as failure.

Lemma 2 *For any nondisjunctive program Π without negation as failure and any consistent set X of literals, if X is an answer set for Π, then X is closed under and supported by Π.*

Proof. Let Π be a nondisjunctive program without negation as failure and X be an answer set for Π. By the definition of an answer set for programs without negation as failure, X is closed under Π. To prove supportedness, take any literal L in X. Since X is minimal among the sets closed under Π, $X \setminus \{L\}$ is not closed under Π. This means that Π contains a rule (7) such that $X \setminus \{L\} \models \mathit{Body}$ but $\mathit{Head} \notin X \setminus \{L\}$. By Lemma 1, $X \models \mathit{Body}$. Since X is closed under Π, it follows that $\mathit{Head} \in X$, so that $\mathit{Head} = L$.

The definition of the reduct F^X of a formula F is similar to the definition of the reduct of a program given in Section 3.

Lemma 3 *For any formula F, any nondisjunctive program Π, and any consistent set X of literals,*

(i) $X \models F$ iff $X \models F^X$;
(ii) X is closed under Π iff X is closed under Π^X;
(iii) X is supported by Π iff X is supported by Π^X.

Proof. Part (i) is immediate by structural induction; parts (ii) and (iii) follow from (i).

Proof of Proposition 2. Consider a nondisjunctive program Π and an answer set X for Π. By the definition of an answer set, X is an answer set for Π^X. Then, by Lemma 2, X is closed under and supported by Π^X. By Lemma 3(ii,iii), it follows that X is closed under and supported by Π.

Lemma 4 *For any formula F, and any set X of literals, $X \models F$ iff $X \cap lit(F) \models F$.*

Proof. Immediate by structural induction.

The following lemma is the special case of Proposition 3 in which Π is assumed to be a program without negation as failure.

Lemma 5 *Let Π be a program without negation as failure whose rules have the form (11). For any consistent set X of literals such that Π is tight on X, if X is closed under and supported by Π, then X is an answer set for Π.*

Proof. By the definition of an answer set for programs without negation as failure, we need to show that no proper subset of X is closed under Π. Let Y be a proper subset of X, and let λ be a function from X to ordinals satisfying condition (*) from the definition of tightness (Section 4). Take a literal $L \in X \setminus Y$ such that $\lambda(L)$ is minimal. Since X is supported by Π, there is a rule

$$L \leftarrow P, N$$

in Π such that

$$X \models (P, N). \tag{14}$$

By the choice of λ, for all $L' \in X \cap lit(P)$,

$$\lambda(L') < \lambda(L).$$

By the choice of L, no literal L' satisfying this inequality may belong to $X \setminus Y$, so that $X \cap lit(P)$ is disjoint from $X \setminus Y$. Consequently, $X \cap lit(P) \subseteq Y$. Since Π does not contain negation as failure, and N is a negative formula, $lit(N) = \emptyset$. Consequently, $lit(P, N) = lit(P)$, so that

$$X \cap lit(P, N) \subseteq Y. \tag{15}$$

By Lemma 4, we can conclude from (14) that

$$X \cap lit(P, N) \models (P, N).$$

In view of (15), it follows by Lemma 1 that $Y \models (P, N)$. Consequently, Y is not closed under Π.

Lemma 6 *For any program Π whose rules have the form (11) and any consistent set X of literals, if Π is tight on X then so is Π^X.*

Proof. Let Π be a program whose rules have the form (11) and X be a consistent set of literals. Suppose that Π is tight on X. Then there exists a function λ from X to ordinals such that, for every rule (11) in Π, if $Head \in X$ and $X \models (P, N)$ then, for all $L \in X \cap lit(P)$,

$$\lambda(L) < \lambda(Head).$$

By Lemma 3(i) and the definition of the reduct, $X \models (P, N)$ iff $X \models (P^X, N^X)$. By the definition of the reduct, $lit(P^X) \subseteq lit(P)$. Therefore, for every rule (11) in Π, if $Head \in X$ and $X \models (P^X, N^X)$ then, for all $L \in X \cap lit(P^X)$, the inequality above is satisfied. Since every rule of Π^X has the form

$$Head \leftarrow P^X, N^X$$

for some rule (11) in Π, it follows that Π^X is tight on X.

Proof of Proposition 3. Consider a program Π whose rules have the form (11) and a consistent set X of literals such that Π is tight on X. Assume that X is closed under and supported by Π. Then, by Lemma 3(ii,iii), X is closed under and supported by Π^X. By Lemma 6, since Π is tight on X, so is Π^X. Hence, by Lemma 5, X is an answer set for Π^X, and consequently an answer set for Π.

Proof of Theorem 2. Let Π be a finite normal program and let X be a set of literals disjoint from $lit(\Pi)$. As discussed in Section 4, the transformations from [10] allow us to turn Π into a program Π' with the same answer sets such that every rule of Π' has the form (11) where P does not contain negation as failure. The examination of these transformations shows that the completion of Π' is equivalent to the completion of Π. Furthermore, these transformations do not change the set of positive body elements, so that $pos(\Pi') = pos(\Pi)$. Consequently, X is disjoint from $pos(\Pi')$, which implies that X is disjoint from $lit(P)$ for every rule (11) of Π'. It follows that condition (*) from the definition of tightness (Section 4) applied to Π' holds trivially for any choice of λ. Then, by Theorem 1, X is an answer set for Π' iff X satisfies the completion of Π'. Consequently, this condition holds for Π as well.

7 Conclusion

We extended the theorem by François Fages that describes the relationship between the completion semantics and the answer set semantics of logic programs to programs with nested expressions permitted in the bodies of rules. The study of this relationship is important from the perspective of answer set programming, and nested expressions are interesting because of their relation to some syntactic constructs that play an important role in the input language of SMODELS. Experiments show that, given a representation of a computational problem in that

language, it is sometimes faster to find a solution by running a propositional solver on the completion of the corresponding program with nested expressions than by using SMODELS.

Acknowledgments

We are grateful to Joohyung Lee, Hudson Turner and the anonymous referees for comments on a draft of this paper. This work was partially supported by National Science Foundation under grant IIS-9732744. The first author was also supported by a NATO science fellowship.

References

1. Krzysztof Apt, Howard Blair, and Adrian Walker. Towards a theory of declarative knowledge. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 89–148. Morgan Kaufmann, San Mateo, CA, 1988.
2. Yuliya Babovich, Esra Erdem, and Vladimir Lifschitz. Fages' theorem and answer set programming.[9] In *Proc. NMR-2000*, 2000.
3. Roberto Bayardo and Robert Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proc. IJCAI-97*, pages 203–208, 1997.
4. Keith Clark. Negation as failure. In Herve Gallaire and Jack Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, New York, 1978.
5. Marc Denecker and Bert Van Nuffelen. Experiments for integration CLP and abduction.[10] In Krysztof R. Apt, Antonios C. Kakas, Eric Monfroy, and Francesca Rossi, editors, *Proceedings of the 1999 ERCIM/COMPULOG workshop on Constraints*, pages 1–15, 1999.
6. Thomas Eiter, Nicola Leone, Christinel Mateis, Gerald Pfeifer, and Francesco Scarcello. A deductive system for non-monotonic reasoning. In *Proceedings of the 4th International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 363–374. Springer-Verlag, 1997.
7. François Fages. Consistency of Clark's completion and existence of stable models. *Journal of Methods of Logic in Computer Science*, 1:51–60, 1994.
8. Paolo Ferraris and Vladimir Lifschitz. Weight constraints as nested expressions.[11] Unpublished draft, 2001.
9. Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In Robert Kowalski and Kenneth Bowen, editors, *Logic Programming: Proc. Fifth Int'l Conf. and Symp.*, pages 1070–1080, 1988.
10. Vladimir Lifschitz, Lappoon R. Tang, and Hudson Turner. Nested expressions in logic programs. *Annals of Mathematics and Artificial Intelligence*, 25:369–389, 1999.
11. Vladimir Lifschitz. Answer set planning. In *Proc. ICLP-99*, pages 23–37, 1999.
12. John Lloyd and Rodney Topor. Making Prolog more expressive. *Journal of Logic Programming*, 3:225–240, 1984.
13. Ilkka Niemelä and Patrik Simons. Efficient implementation of the well-founded and stable model semantics. In *Proc. Joint Int'l Conf. and Symp. on Logic Programming*, pages 289–303, 1996.

[9] `http://arxiv.org/abs/cs.ai/0003042 .`
[10] `http://www.cs.kuleuven.ac.be/cgi-bin-dtai/publ_info.pl?id=20013 .`
[11] `http://www.cs.utexas.edu/users/vl/papers/weight.ps .`

14. Nikolay Pelov, Emmanuel De Mot, and Marc Denecker. Logic programming approaches for representing and solving constraint satisfaction problems : a comparison.[12] In Parigot M. and Voronkov A., editors, *Proceedings of the 7th International Conference on Logic for Programming and Automated Reasoning*, volume 1955 of *Lecture Notes in Artificial Intelligence*, pages 225–239. Springer, 2000.
15. Pascal van Hentenryck. *Constraint Satisfaction in Logic Programming*. MIT Press, 1989.
16. Hantao Zhang. SATO: An efficient propositional prover. In *Proc. CADE-97*, 1997.

[12] `http://www.cs.kuleuven.ac.be/cgi-bin-dtai/publ_info.pl?id=32258` .

Semantics of Normal Logic Programs with Embedded Implications

Fernando Orejas[2], Edelmira Pasarella[1,2], and Elvira Pino[2]

[1] Dpto de Computación y Tecnología de la Información, Universidad Simón Bolívar, Aptdo Postal 89000, Caracas 1089, Venezuela.
edelmira@lsi.upc.es

[2] Dpto de L.S.I., Universidad Politècnica de Catalunya, Campus Nord, Mòdul C6, Jordi Girona 1-3, 08034 Barcelona, Spain.
{pino,orejas}@lsi.upc.es

Abstract. The aim of our work is the definition of a model-theoretic semantics of normal logic programs with embedded implications. We first propose a quite simple operational semantics for this class of programs whose negation mechanism is the constructive negation. This semantics is used to prove the adequacy of the model-theoretic semantics. Then we define a declarative semantics for this class of programs in terms of Beth models and show that in the model class associated to every program there is a least model that can be seen as the semantics of the program, which may be built upwards as the least fixpoint of a continuous immediate consequence operator. Finally, it is proved that the operational semantics is sound and complete with respect to the least fixpoint semantics.

1 Introduction

There are two main approaches to decompose large logic programs into manageable units [2]. On the one hand, different kinds of modular units, similar to the module notions existing in other programming paradigms, have been proposed and studied together with the corresponding composition operations. This kind of modularization can be considered "external" to the logic programming paradigm, since it is based on the use of constructions which are, to a certain extent, independent of any programming formalism [10]. Conversely, the second approach provides a structuring mechanism in terms of a logical connective. In particular, this approach originated in the work of Miller [9]. The idea is that intuitionistic implication may be used to structure a logic program into blocks, as it is done in imperative programming languages. For more details on the use of this connective the reader may look, for instance, at [2].

Providing semantics to normal logic programming units is, in general, a difficult task, due to the non-monotonic nature of negation that hinders the definition of a compositional semantics. Nevertheless, a certain amount of work has been done for defining the semantics of normal logic program modules according to the first approach mentioned above (see, e.g., [6,4]). This is not quite true

P. Codognet (Ed.): ICLP 2001, LNCS 2237, pp. 255–268, 2001.

for the second approach. To our knowledge, only [1] and [7] consider this case. The reason for this lack of work is, probably, the apparent difficulty in mixing two semantically very different connectives such as negation and intuitionistic implication. Intuitionistic connectives seem to ask for intuitionistic models, like Kripke or Beth models, while negation seems to ask for some kind of 3-valued models. The semantics presented in [1] and [7] is defined in terms of some sort of Kripke models. The results presented in [1] are very restrictive. Only the case of negation as failure is considered, programs must be stratified and signatures may only contain predicate symbols, i.e. function symbols are not allowed. The work presented in [7] also deals with negation as failure, but the other restrictions are not present. In addition, we found the model-theoretic semantics quite ad-hoc. The kind of Kripke models used are not really intuitionistic: first, the interpretation associated to a given world is a three-valued structure and, second and more importantly, the order relation between worlds is not monotonic, in contrast with the intuition underlying intuitionistic Kripke models. This means that if an atom is satisfied by the interpretation associated to a given world, then it may not be satisfied by the interpretation associated to a greater world.

In [8] we defined a new declarative compositional semantics for a general class of normal logic program units, in terms of a class of models that we called ranked. As we pointed out in that paper, ranked models are, intuitively, quite close to Beth models. This lead us to think that both connectives could have a natural and reasonably simple semantics in terms of intuitionistic (Beth) models. Moreover, this semantics would make more explicit the intuitionistic nature of negation in logic programming already pointed out by other authors (e.g. [12]). In this paper we follow these ideas. We first propose a quite simple operational semantics for this class of programs whose negation mechanism is the constructive negation [3,5,13]. This semantics is used to prove the adequacy of the model-theoretic semantics. Then we define a declarative semantics for this class of programs in terms of Beth models and show that in the model class associated to every program there is a least model that can be seen as the semantics of the program, which may be built upwards as the least fixpoint of a continuous immediate consequence operator. Finally, it is proved that the operational semantics is sound and complete with respect to the least fixpoint semantics.

This paper is organized as follows. In the following section we present some basic concepts and notation used in the paper. Section 3 introduces the operational semantics of our programs. Section 4 defines the Beth models used in the paper and their associated forcing relation. The next section introduces the immediate consequence operator showing that it is monotonic and continuous. Section 6, defines an ordering relation on the class of models of a program and shows that the least model coincides with the least fixpoint of the immediate consequence operator. Finally, in Section 7, the operational semantics is proved to be sound and complete with respect to the least model semantics.

2 Preliminaries

A *countable signature* Σ consists of a pair of sets (FS_Σ, PS_Σ) of function and predicate symbols with some associated arity. Terms, atoms or first order formulas built by using functions and/or predicates from Σ and, also, variables from a fixed countable set X of variable symbols are called Σ-terms, Σ-atoms and Σ-formulas, respectively. The Herbrand universe, denoted by H_Σ, is the set of all ground Σ-terms constructed by using Σ-functions. Terms are denoted by $t, s, \ldots$, predicate symbols by $p, q, \ldots$ and function symbols by $f, g, \ldots$. Letters a, b denote atoms and the character ℓ is used for literals. A formula whose subterms are variables is called a *flat* formula. $\varphi^\forall$ and $\varphi^\exists$ are the universal and existential closure of φ, respectively. The logical constants are denoted by $\underline{\mathtt{t}}$ and $\underline{\mathtt{f}}$. Programs are denoted by using the letters P and Q. In general, subscripts and superscripts will be used if needed and a bar is used to denote (finite) sequences of objects. *Normal logic programs with embedded implications* over a signature Σ are finite sets of clauses $a :- G_1, \ldots, G_k$, where a is a Σ-atom and $\forall i \in \{1, \ldots, k\}$, G_i is an *intuitionistic Σ-literal*, that is, either a Σ-literal, b or $\neg b$, where b is a Σ-atom; or an *intuitionistic Σ-expression*, $P_i \supset G'_i$, where P_i is a Σ-program and G'_i is an intuitionistic Σ-literal. The idea behind this kind of goals is that, when evaluating G'_i, one may use the definitions in P_i as auxiliary local definitions (in addition to other global definitions in the given program). Clauses whose head is empty correspond to goals of this class of language, called *intuitionistic Σ-goals*.

Free variables in a clause are assumed to be implicitly quantified universally. This means that the scope of a variable is the clause where it is defined. In particular, given the goal $\{p(x).\} \supset p(f(x))$, x in $p(x)$ is not considered to be bound to x in $p(f(x))$ and, as a consequence, the goal should succeed.

We consider that clauses are written following the structure of constraint normal clauses with flat head. That is, $p(t_1, \ldots, t_n) :- G_1, \ldots, G_k$ is written as the constrained clause $p(x_1, \ldots, x_n) :- G_1, \ldots, G_k \Box x_1 = t_1, \ldots, x_n = t_n$

Moreover, we suppose that the identical tuple $x_1, \ldots, x_n$ of *fresh variables* occurs in all clauses (in a program) with predicate p in its heads. Also, just to simplify, clauses of the form $a :- \Box\underline{\mathtt{t}}$ are written as a.

The *set of definitions of a predicate* p in a program P is defined as usual:

$$Def(P, p) \equiv \{p(\overline{x}) :- \overline{G}^k \Box c^k \in P\}$$

Constraints occurring in programs are *equality Σ-constraints*, that is, arbitrary first order Σ-formulas in which the only relational symbol occurring in atoms is the equality (formulas composing equality atoms with the connectives $\neg, \wedge, \vee, \rightarrow$, and the quantifiers $\forall, \exists$). Constraints are denoted by using the letters c and d (possibly with sub or super-scripts). We will handle constraints in a logical way, using logical consequence of the *free equality theory*, FET_Σ (see, e.g., [12]).

A constraint c is *satisfiable* (resp. unsatisfiable) if, and only if, $FET_\Sigma \models c^\exists$ (resp. $FET_\Sigma \models \neg(c^\exists)$); a constraint d is *less general* than c if, and only if, $FET_\Sigma \models (d \rightarrow c)^\forall$. A ground substitution $\overline{x} = \overline{t}$ (where t_i are closed terms) is called a *solution of a constraint* c if, and only if, $FET_\Sigma \models (\overline{x} = \overline{t} \rightarrow c)^\forall$.

A constrained Σ-atom is a pair $p(\overline{x})\Box c(\overline{x})$, where $p \in PS_\Sigma$ and $c(\overline{x})$ is a satisfiable Σ-constraint. The set of all the constrained Σ-atoms is denoted $\mathcal{L}_\Sigma(X)$.

3 Operational Semantics

In this section we introduce an operational semantics for the class of normal logic programs with embedded implication. This semantics is presented in terms of a derivation relation over sequents of the form $P \vdash \overline{G}\Box c$, where P is a Σ-program and $:- \overline{G}\Box c$ is a Σ-goal. It may be noted that our semantics is very simple, but not very useful for practical purposes, since it is too non-deterministic to be directly implemented. Its main aim is to show the adequacy of the model-theoretic semantics defined below. In particular, our treatment of negation can be seen as a more abstract and simple variation of [3,5,13]. Instead, we could have introduced an operational semantics closer to implementation. For instance, a variation of the BCN semantics [11]. However, the proofs for the main results of this paper would have been slightly more complex.

The following mutually recursive definitions establish our semantics.

Definition 1. *Let P be a Σ-program and $:- \overline{G}\Box c$ a Σ-goal. $P \vdash \overline{G}\Box c$ can be proved with computed answer c' if and only if there exists a finite derivation of $P \vdash \overline{G}\Box c$, that is, $P \vdash \overline{G}\Box c \stackrel{n}{\Rightarrow} P \vdash \Box c'$, $n \geq 0$, $FET_\Sigma \models c'^{\exists}$ where $\stackrel{n}{\Rightarrow}$ corresponds to n applications (derivation steps) of the relation $\Rightarrow$ over sequents.*

Definition 2. *The derivation relation $\Rightarrow$ over sequents is defined as follows:*

1. *$P \vdash \overline{G}_1, p(\overline{x}), \overline{G}_2\Box c \Rightarrow P \vdash \overline{G}_1, \overline{G}, \overline{G}_2\Box c \wedge d$ if there exists a (renamed apart) clause $p(\overline{x}) :- \overline{G}\Box d \in Def(P, p)$ and $FET_\Sigma \models (c \wedge d)^{\exists}$*
2. *$P \vdash \overline{G}_1, \neg p(\overline{x}), \overline{G}_2\Box c \Rightarrow P \vdash \overline{G}_1, \overline{G}_2\Box c'$ if for every (renamed apart) clause $p(\overline{x}) :- G_1, \ldots, G_m\Box d$ there exists $J \subseteq \{1, \ldots, m\}$ such that $\forall j \in J : P \vdash \neg G_j\Box d$ can be proved with computed answer d_j and $FET_\Sigma \models (c' \rightarrow \neg d \vee \bigvee_{j \in J} d_j)^{\forall}$.*
3. *$P \vdash \overline{G}_1, Q \supset G, \overline{G}_2\Box c \Rightarrow P \vdash \overline{G}_1, \overline{G}_2\Box c'$ if $P \cup Q \vdash G\Box c$ can be proved with computed answer c'.*
4. *$P \vdash \overline{G}_1, \neg(Q \supset G), \overline{G}_2\Box c \Rightarrow P \vdash \overline{G}_1, \overline{G}_2\Box c'$ if $P \cup Q \vdash \neg G\Box c$ can be proved with computed answer c'.*

We assume that whenever a constrained Σ-atom $\neg\neg a\Box c$ occurs in the right part of a sequent, it denotes $a\Box c$.

Example 3. Given the programs $P = \{p(x) :- p(x)\Box x = a, q(x) :- \Box x = a\}$ and $Q = \{r :- p(x), \neg q(x)\}$

$P \vdash Q \supset \neg r \Rightarrow P \vdash \Box\underline{\mathtt{t}}$ because (def. 2.3):
$P \cup Q \vdash \neg r \Rightarrow P \cup Q \vdash \Box\underline{\mathtt{t}}$ (def. 2.2)
$P \cup Q \vdash \neg p(x) \Rightarrow P \cup Q \vdash \Box x \neq a$ (def. 2.2) and
$P \cup Q \vdash q(x) \Rightarrow P \cup Q \vdash \Box x = a$ (def. 2.1)
and $FET_\Sigma \models (\underline{\mathtt{t}} \rightarrow x \neq a \vee x = a)^{\forall}$

4 Model Theory Semantics

In this section, we introduce a class of Beth models, and an associated forcing relation, to define the semantics of our programs. Beth models and Kripke models are based on a similar intuition [14]. Both kinds of models are defined as a family of logical structures, where each structure, (corresponding to a *world*) denotes the amount of knowledge one has at a certain moment. Worlds are (partially) ordered, where the ordering relation denotes the increase of knowledge. In these models a *forcing* relation plays the role of satisfaction. Forcing is defined for each world and defines what one can expect to be true in the given world. The key difference between Kripke and Beth models is in the definition of the forcing relation. In Kripke models an atomic formula is forced in a world w if it is satisfied by the associated structure. In Beth models, an atomic formula is forced in w if we may be sure that it will be satisfied in the future. This is formalized saying that the formula is satisfied by all the structures in a bar for that world. Where a bar for w is a set of worlds such that any increasing sequence of worlds starting in w would contain a world in the bar.

In our case, worlds are pairs (P, L), where P is a program and L is a set of constrained atoms. The structure associated to a world is also represented (as a variation of Herbrand structures) as a set of constrained atoms. The intuition is that for a given world (P, L), assuming that the given program includes all the clauses in P, we know that the atoms in L are false and the ones in the associated structure are true. Worlds can be seen as stages in computation, where additional computation provides additional knowledge. The idea is that the atoms in L, for a world (P, L), should be supported by the given program and by the knowledge in the previous worlds. In this context, forcing should be defined like for Beth models: an atomic formula is forced in a world if it will hold in the future.

Example 4. To illustrate the ideas above, given $P = \{p :- \neg q, q :- \neg r, s :- \neg p\}$, a model for P may include, for instance, the worlds: $(\emptyset, \emptyset), (\emptyset, \{r\}), (\emptyset, \{p, r\})$ with the associated structures $(\emptyset), (\{q\}), (\{q, s\})$ In this model, the world $(\emptyset, \{r\})$ together with its associated structure $\{q\}$ represent that, at certain stage, we may know that q is true but r is not. The fact that the program in these worlds is empty means that we may have this knowledge without assuming that we have additional clauses (others than the ones in P). However, we may consider that the atom s is forced in this world, because it holds in the following (larger) world.

Definition 5. *A Σ-world w is a pair (P_w, L_w) where P_w is a Σ-program up to variable renaming and $L_w \subseteq \mathcal{L}_\Sigma(X)$. The set of all Σ-worlds is denoted W_Σ. A Σ-structure is a 3-tuple $\mathcal{B} = (W, \preceq, I)$, where $W \subseteq W_\Sigma$ and*

1. *For every Σ-program P, $(P, \emptyset) \in W$*
2. *$\preceq$ is a partial order on W, such that $\forall v, w \in W : v \preceq w$ if, and only if, $P_v = P_w$ and $L_v \subseteq L_w$. The strict order associated to $\preceq$ is denoted $\prec$.*
3. *The interpretation function $I : W \to 2^{\mathcal{L}_\Sigma(X)}$ satisfies the following properties:*

(a) $\forall v \in W,\ \underline{\mathtt{t}}\Box c \in I(v)$ *if* $FET_\Sigma \models c^\exists$
(b) (Monotonicity) $\forall v, w \in W$*, if* $v \preceq w$ *then* $I(v) \subseteq I(w)$

In addition, we consider that, $\forall w \in W$, the sets of constrained atoms L_w and $I(w)$ are closed under renaming of variables, disjunction of constraints and less general constraints. Also, $\mathcal{B}(P) = \langle W(P), \preceq, I(P)\rangle$ corresponds to the ordered structure associated to P occurring in $\mathcal{B}$ such that $W(P) = \{w \in W \,|\, (P, \emptyset) \preceq w\}$ and $I(P) = \{I(w) \mid w \in W(P)\}$.

For simplicity, we will assume the following notational conventions: $\ell\Box c \in (I(w), L_w)$ means $a\Box c \in I(w)$ if $\ell = a$, and $a\Box c \in L_w$ if $\ell = \neg a$. Conversely, we write $\neg\ell\Box c \in (I(w), L_w)$ to denote $a\Box c \in L_w$ if $\ell = a$, and $a\Box c \in I(w)$ if $\ell = \neg a$.

Definition 6 (forcing). *Let* $\mathcal{B} = (W, \preceq, I)$ *be a* Σ*-structure. We say that* $B \subseteq W$ *is a bar w.r.t. a world* $v \in W$ *iff for each* $\preceq$*-increasing chain of worlds* $v_0, v_1, \ldots$ *in* W *such that* $v_0 = v$*, there exists* $k \geq 0$ *and* $w \in B$ *such that* $v_k \preceq w \preceq v_{k+1}$*. The bar* B *is strict iff* $\forall v, w \in B$*,* $v \not\preceq w$ *and* $w \not\preceq v$*. Then, the forcing relation* $\Vdash$*, on a* Σ*-structure* $\mathcal{B}$ *is inductively defined for every world as follows. Let* $v \in W$*:*

1. *For all satisfiable constraints* c *and* d*:* $v, \mathcal{B} \Vdash c\Box d$*, iff* $I(v) \models (c \wedge d)^\exists$*.*
2. $v, \mathcal{B} \Vdash \ell\Box c$*, iff there exists a bar* $B \subseteq W$ *with respect to* v *such that for all* $w \in B : \ell\Box c \in (I_{\mathcal{B}}(w), L_w)$*.*
3. $v, \mathcal{B} \Vdash \overline{G}_1, \overline{G}_2\Box c$ *iff* $v, \mathcal{B} \Vdash \overline{G}_1\Box c$*,* $v, \mathcal{B} \Vdash \overline{G}_2\Box c$*.*
4. $v, \mathcal{B} \Vdash \neg(G_1, \ldots, G_m)\Box c$ *iff* $v, \mathcal{B} \Vdash \neg G_j\Box c_j$ *for some* $j \in J \subseteq \{1, \ldots, m\}$ *and* $FET_\Sigma \models (c \to \bigvee_{j\in J} c_j)^\forall$*.*
5. $v, \mathcal{B} \Vdash P \supset G\Box c$ *iff there exists a satisfiable constraint* c'*,* $FET_\Sigma \models (c' \to c)^\forall$ *such that* $(P_v \cup P, \emptyset), \mathcal{B} \Vdash G\Box c'$
6. $v, \mathcal{B} \Vdash \neg(P \supset G\Box c)$ *iff there exists a satisfiable constraint* c'*,* $FET_\Sigma \models (c' \to c)^\forall$ *such that* $(P_v \cup P, \emptyset), \mathcal{B} \Vdash \neg G\Box c'$
7. $v, \mathcal{B} \Vdash p(\overline{x}) :- \overline{G}\Box d$ *iff* $\forall w \succeq v$ *if* $w, \mathcal{B} \Vdash \overline{G}\Box d$ *then* $w, \mathcal{B} \Vdash p(\overline{x})\Box d$*.*

A program P can be seen as an intuitionistic theory. As a consequence, one could just define the class of models defined by P as the subclass of all the structures such that P is forced by the world $(\emptyset, \emptyset)$ (or, perhaps, by the world $(P, \emptyset)$). However, this is not satisfactory for our purposes. Many models in that class would not agree with the computational interpretation of our models discussed above. According to the definition below, a structure is a model of a program P if two conditions are satisfied. The first one is that the structure associated to a world (P', L) should satisfy all the consequences that could be computed from the clauses in P and in P' and the negative information in L. The second condition states that the negative information, L, in a world (P', L) must be supported by the clauses in P and in P' and the information included in previous worlds.

Now, in order to formalize these intuitions we will define a notion of *local forcing*, which can be seen as a kind of local satisfaction on a given world. There are two key ideas in this definition. The first one is to consider that a positive literal ℓ is locally forced in a world w if ℓ is in the interpretation of w, and a

negative literal ℓ is locally forced in $w = (P, L)$ if ℓ is in L. The second idea is to consider that a formula $P' \supset \ell$ is locally forced in a world $w = (P, L)$ if ℓ is locally forced in a bar for the world $(P \cup P', \emptyset)$ consisting of worlds $(P \cup P', L')$ where L' is included in L. This means that in L we have enough negative information to compute ℓ. The extension to other kind of formulas is the obvious one.

Definition 7 (Local forcing). *The local forcing relation, $\Vdash_l$, on a Σ-structure $\mathcal{B} = (W, \preceq, I)$ is inductively defined for every world as follows. Let $v \in W$, then:*

1. $v, \mathcal{B} \Vdash_l \ell \Box c$ *iff* $\ell \Box c \in (I(v), L_v)$.
2. $v, \mathcal{B} \Vdash_l P^1 \supset \ldots \supset P^n \supset \ell \Box c$ *iff there exists a satisfiable constraint c', $FET_\Sigma \models (c' \rightarrow c)^\forall$ and there exists a bar B w.r.t. $(P_v \cup P^1 \cup \ldots \cup P^n, \emptyset)$ such that $\forall v' \in B$, $L_{v'} \subseteq L_v$ and $\ell \Box c' \in (I(v'), L_{v'})$.*
3. $v, \mathcal{B} \Vdash_l \neg(P^1 \supset \ldots \supset P^n \supset \ell) \Box c$ *iff* $v, \mathcal{B} \Vdash_l P^1 \supset \ldots \supset P^n \supset \neg\ell \Box c$.
4. $v, \mathcal{B} \Vdash_l \overline{G}_1, \overline{G}_2 \Box c$ *iff* $v, \mathcal{B} \Vdash_l \overline{G}_1 \Box c$, $v, \mathcal{B} \Vdash_l \overline{G}_2 \Box c$.
5. $v, \mathcal{B} \Vdash_l p(\overline{x}) :- \overline{G} \Box d$ *iff* $\forall w \succeq v$ *if* $w, \mathcal{B} \Vdash_l \overline{G} \Box d$ *then* $w, \mathcal{B} \Vdash_l p(\overline{x}) \Box d$.

The relation $\Vdash_l$ is included in $\Vdash$, i.e. if $v, \mathcal{B} \Vdash_l \overline{G} \Box c$ then $v, \mathcal{B} \Vdash \overline{G} \Box c$.

Definition 8 (Models). *$\mathcal{B} \in Struct(\Sigma)$ is a model of P, written $\mathcal{B} \models_d P$, if, and only if, $\forall w \in W_\mathcal{B}$, the following two conditions hold:*

1. $\forall p(\overline{x}) :- \overline{G} \Box d \in P \cup P_w : w, \mathcal{B} \Vdash_l p(\overline{x}) :- \overline{G} \Box d$
2. *Supported worlds: if $p \Box c \in L_w$ then $\forall p(\overline{x}) :- G_1, \ldots, G_m \Box d \in Def(P \cup P_w, p)$, there exist satisfiable constraints $\{d_j\}_{j \in J}$, $J \subseteq \{1, \ldots, m\}$ such that $\forall j \in J$ $\exists v \in W_\mathcal{B}$, $v \prec w$ with $v, \mathcal{B} \Vdash_l \neg G_j \Box d_j$ and $FET_\Sigma \models (c \rightarrow \neg d \vee \bigvee_{j \in J} d_j)^\forall$.*

For every program P, we define $Mod(P)$ as the class of all its models.

Example 9. Consider the program $P = \{r :- \{p :- \neg q\} \supset s,\ s :- p\}$. A model of P could include any of the structures $\mathcal{B}_1$ or $\mathcal{B}_2$ described below, where the sets at the right hand side of the worlds in $\mathcal{B}_1$ or $\mathcal{B}_2$ denote their interpretation.

$$\mathcal{B}_1 = \left\{ \begin{array}{ccc} (\emptyset, \{p, q, s\})\{r\} & & \\ | & & \\ (\emptyset, \{p, q\})\{r\} & (\{p :- \neg q\}, \{q\})\{p, r, s\} & \\ | & | & \\ (\emptyset, \emptyset)\emptyset & (\{p :- \neg q\}, \emptyset)\emptyset & \end{array} \right\} \quad \mathcal{B}_2 = \left\{ \begin{array}{cc} (\emptyset, \{q\})\{p, s, r\} & \\ | & \\ (\emptyset, \emptyset)\{p, s, r\} & (\{p :- \neg q\}, \emptyset)\{p, q, r, s\} \end{array} \right\}$$

5 Least Fixpoint Semantics

In this section, we define an immediate consequence operator T_P that can be used, as usual, to build (bottom-up) a least fixpoint of the operator, which is shown to be a model of the given program P. This fixpoint will be shown to be the least model in $Mod(P)$, with respect to an ordering that will be defined in the following section. Moreover, the operational semantics defined above will be shown to be sound and complete with respect to that model. However, T_P is not defined on all Σ-structures as one would expect, since the

class $Struct(\Sigma)$ includes some structures that are not constructive. Instead, T_P is defined on the subclass of *Noetherian* Σ-structures, which is enough for our purposes. In particular, we show that our operator is monotonic and continuous for this subclass.

Although the definition of T_P may look complex, the intuition is quite simple. Given a Σ-structure $\mathcal{A}$, $T_P(\mathcal{A})$, on one hand, for every world (P', L) in $\mathcal{A}$, T_P builds a new world, and the corresponding interpretation, where all the immediate negative consequences (w.r.t. the information we have at this point and the clauses in P and P') are added to L and all the immediate positive consequences are added to its interpretation. On the other hand, additional worlds can be added including the negative information that is supported by the existing worlds in $\mathcal{A}$. The interpretation of these additional worlds just includes the positive information that we have at this point.

Definition 10. *For all $\mathcal{A} \in Struct(\Sigma)$, $T_P(\mathcal{A}) = \langle W_{T_P(\mathcal{A})}, \preceq, I_{T_P(\mathcal{A})} \rangle$ is the structure in $Struct(\Sigma)$ defined as follows for every Σ-program P':*

1. *$W_{T_P(\mathcal{A})}(P') = \{t_{P'}(v) \,|\, v \in W_{\mathcal{A}}(P')\} \cup \{Succ_{P'}(v) \,|\, v$ is maximal in $W_{\mathcal{A}}(P')\}$ where $t_{P'}(v) = (P', L_{t_{P'}(v)})$ and $Succ_{P'}(v) = (P', L_{Succ_{P'}(v)})$ are defined as follows for every $v \in W_{\mathcal{A}}$,*

$$L_{t_{P'}(v)} = \{p \square c \mid \text{for all } p(\overline{x}) :- G_1, \ldots, G_m \square d \in P \cup P', \text{ there exist satisfiable constraints } \{d_j\}_{j \in J}, \text{ and for every } j \in J \subseteq \{1, \ldots, m\}, \exists v' \in W_{\mathcal{A}}(P') \text{ with } v' \prec v \text{ such that } v', \mathcal{A} \Vdash_l \neg G_j \square d_j, \text{ and } FET_\Sigma \models (c \rightarrow \neg d \vee \bigvee_{j \in J} d_j)^\forall\}$$

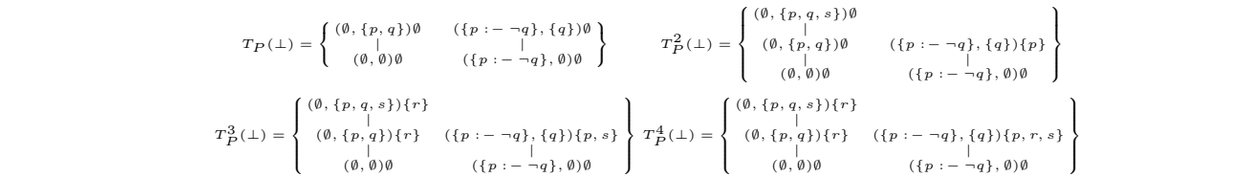
$$L_{Succ_{P'}(v)} = \{p \square c \mid \text{for all } p(\overline{x}) :- G_1, \ldots, G_m \square d \in P \cup P', \text{ there exist satisfiable constraints } \{d_j\}_{j \in J}, \text{ and for every } j \in J \subseteq \{1, \ldots, m\}, v, \mathcal{A} \Vdash_l \neg G_j \square d_j, \text{ and } FET_\Sigma \models (c \rightarrow \neg d \vee \bigvee_{j \in J} d_j)^\forall\}$$

2. *$I_{T_P(\mathcal{A})}$ is defined as follows. If $w \in t_{P'}(W_{\mathcal{A}}(P'))$:*

$$I_{T_P(\mathcal{A})}(w) = \{p \square c \mid \text{there is } \{p(\overline{x}) :- \overline{G}^k \square d^k \mid 1 \leq k \leq n\} \subseteq P \cup P', \text{ and } \forall k, 1 \leq k \leq n, \exists v \in W_{\mathcal{A}}(P'), w = t_{P'}(v) \text{ such that } v, \mathcal{A} \Vdash_l \overline{G}^k \square d^k \text{ and } FET_\Sigma \models (c \rightarrow \bigvee_{k=1}^n d^k)^\forall\}$$

If $w \in Succ_{P'}(W_{\mathcal{A}}(P'))$, $I_{T_P(\mathcal{A})}(w) = \{p \square c \in I_{T_P(\mathcal{A})}(w') \mid w' \preceq w\}$.

Example 11. Let us see the construction of the least fixpoint of the program P given in example 9. The bottom Σ-structure is just a structure where, for every program P', we just have a world $(P', \emptyset)$ whose interpretation is the empty set of atoms. Then, the least fixpoint is $T_P^4(\bot)$:



Σ-structures can be ordered according to the amount of information they contain. In particular, given two structures $\mathcal{A}$ and $\mathcal{B}$, we consider that $\mathcal{A} \preceq_F \mathcal{B}$ if there is some mapping from the worlds in $\mathcal{A}$ to the worlds in $\mathcal{B}$ such that the positive and negative information in a world w in $\mathcal{A}$ is included in the positive and negative information of the corresponding worlds in $\mathcal{B}$. In general, this mapping may associate to each w in $\mathcal{A}$, not just a world in $\mathcal{B}$, but a set of worlds. In addition, we ask this mapping to be downward surjective, which means that the worlds in $\mathcal{A}$ are surjectively mapped into a prefix of the set of worlds in $\mathcal{B}$.

Definition 12. *For all $\mathcal{A}$ and $\mathcal{B} \in Struct(\Sigma)$, $\mathcal{A} \preceq_F \mathcal{B}$ if, and only if, for every Σ-program P, there exists a map $f_P : W_{\mathcal{A}}(P) \to 2^{W_{\mathcal{B}}(P)}$ which is:*

i) Monotonic: $\forall w \in W_{\mathcal{A}}(P) \forall v \in f_P(w), w \preceq v$ and $I_{\mathcal{A}}(w) \subseteq I_{\mathcal{B}}(v)$, and
ii) Downward surjective: $\forall w \in W_{\mathcal{A}}(P) \forall w' \in W_{\mathcal{B}}(P)$ such that $\forall v \in f_P(w), w' \preceq v$ then $\exists w'' \in W_{\mathcal{A}}(P)$ with $w'' \preceq w$ and $w' \in f_P(w'')$.

Remark 13. For every Σ-program P, if T_P is applied to a Σ-structure $\mathcal{A}$ such that for every P', $W_{\mathcal{A}}(P')$ is a totally ordered structure, then $t_{P'} : W_{\mathcal{A}}(P') \to W_{T_P(\mathcal{A})}(P')$ in Definition 10, is a downward surjective embedding. In addition, all $t_{P'}$ defined by the powers of T_P on $\bot$, $\{T_P^k(\bot) \mid 0 \preceq k\}$, are monotonic.

The previous relation is not an ordering when defined over the class of all Σ-structures. The problem is that antisymmetry may fail when considering non-constructive structures where there are infinite descending sequences of worlds. However, the following theorem shows that this relation is indeed an ordering when restricted to the subclass of *Noetherian* Σ-structures i.e. structures that do not include infinite descending sequences of worlds.

Theorem 14. *$\preceq_F$ is a complete partial order on Noetherian Σ-structures.*

Proof. $\preceq_F$ is trivially reflexive. It is transitive because *i*) and *ii*) are preserved under composition. To show that it is antisymmetric we use parallel induction over the Noetherian order, $\preceq$, of worlds of Σ-structures, to prove that, if there exist monotonic and downward surjective maps $f_P : W_{\mathcal{A}}(P) \to 2^{W_{\mathcal{B}}(P)}$ and $g_P : W_{\mathcal{B}}(P) \to 2^{W_{\mathcal{A}}(P)}$ then their compositions are the corresponding identities.

The bottom is $\bot = \langle W_\bot, \preceq_\bot, I_\bot \rangle$, where $W_\bot = \{(P, \emptyset) | P \text{ is a } \Sigma - program\}$, $\preceq_\bot = \{(w, w) \mid w \in W_\bot\}$ and $\forall w \in W_\bot$, $I_\bot(w) = \{\underline{\mathtt{t}} \Box c \mid FET_\Sigma \models c^\exists\}$. Note that this bottom is trivially Noetherian.

Finally, the least upper bound for every increasing chain of structures $\mathcal{A}_1 \preceq_F \mathcal{A}_2 \preceq_F \ldots$, can be described as a "world-by-world union" where the correspondence between them is established by the maps $W_{\mathcal{A}_1}(P) \to W_{\mathcal{A}_2}(P) \to \ldots$. ■

Theorem 15. *T_P, when restricted to the class of Noetherian Σ-structures, is monotonic and continuous with respect to $\preceq_F$ so, it has a least fixpoint $T_P \uparrow \omega$.*

Proof. First of all, monotonicity is proved by showing that $\forall \mathcal{A}, \mathcal{B} \in Struct(\Sigma)$ such that $\mathcal{A} \preceq_F \mathcal{B}$ and, $\forall f_{P'} : W_{\mathcal{A}}(P') \to 2^{W_{\mathcal{B}}(P')}$ satisfying Definition 12, the map $g_{P'} : W_{T_P(\mathcal{A})}(P') \to 2^{W_{T_P(\mathcal{B})}(P')}$ defined as follows, also satisfies definition 12:

- $\forall w \in W_{T_P(\mathcal{A})}(P')$, if $w = t_{P'}(v)$ then $g_{P'}(t_{P'}(v)) = t_{P'}(f_{P'}(v))$
- $\forall w \in W_{T_P(\mathcal{A})}(P')$, if $w = Succ_{P'}(v)$ then $g_{P'}(Succ_{P'}(v)) = Succ_{P'}(f_{P'}(v))$

Then, since T_P is monotonic, to prove continuity it is enough to prove that T_P is finitary, that is for any infinite chain of Σ-structures $\mathcal{A}_1 \preceq_F \mathcal{A}_2 \preceq_F \ldots$, we have that $T_P(\sqcup \mathcal{A}_i) \preceq_F \sqcup T_P(\mathcal{A}_i)$. The proof proceeds in the standard way. First, one can show that any immediate consequence in $T_P(\sqcup \mathcal{A}_i)$ is obtained using a finite set of literals $\ell_j \Box d_j$ from a finite set of worlds in $\sqcup \mathcal{A}_i$. Then, there will be a least upper bound $\mathcal{A}_n$, in the chain $\{\mathcal{A}_i\}$, including all these literals. ■

6 Least Model Semantics

In this section we will prove that the least fixpoint of the immediate consequence operator $T_P \uparrow \omega$ is the least model in $Mod(P)$ with respect to a proper notion of ordering. The key issue here is to define an ordering relation in $Mod(P)$, which we will denote by $\sqsubseteq$, such that it adequately captures the intuition that the "best model" is the least one. The definition of this ordering is based, first, on the definition of an ordering between ordered structures associated to a given program P. Then this ordering is extended to compare Σ-structures by comparing the ordered structures included.

One may notice that, in an ordered structure associated to P, (if this structure is part of a model of a program P') the negative information associated to a given world will contain, at most, the negative information supported by the worlds below. Similarly, the positive information associated to a given world will contain, at least, all the consequences that can be computed from the clauses in P and in P' and the negative information in the world. In this sense, one may consider that the "best ordered structure" is one in which the negative and positive information associated to each world is, respectively, the maximum and the minimum amount of possible information. This means that the ordering between ordered structures should be based on an extension of the, so-called, standard ordering of 3-valued structures. However, given two ordered structures $\mathcal{B}_1(P)$ and $\mathcal{B}_2(P)$, we should not try to compare pairwise all the worlds in one structure with the associated worlds in the other. For instance, let us suppose that P consists of the clause $r :- \neg q$ and P' is empty. If q is not included in the interpretation of the world $(P, \emptyset)$ in $\mathcal{B}_1(P)$ then the world above may be $(P, \{q\})$ and its interpretation would include r. However, if q is included in the interpretation of $(P, \emptyset)$ in $\mathcal{B}_2(P)$ then the world above can be $(P, \{r\})$. $\mathcal{B}_1(P)$

should be considered better than $\mathcal{B}_2(P)$. This can be done by defining this ordering among ordered structures as some kind of lexicographic extension of the standard ordering.

However, the fact that ordered structures may be not linear poses some small additional difficulty: two worlds may be incomparable but, at the same time, be defined over the same set of worlds. Nevertheless, with the intuition discussed above, to compare $\mathcal{B}_1(P)$ and $\mathcal{B}_2(P)$ we proceed as follows. First, we look for two bars $B1$ and $B2$ in $\mathcal{B}_1(P)$ and $\mathcal{B}_2(P)$, respectively, in each structure, such that all the worlds and interpretations below coincide and such that all the worlds and/or interpretations in both bars are different. Then, if all the worlds and interpretations in $B1$ are smaller (w.r.t. the standard ordering) than all the worlds and interpretations in $B2$, then we consider $\mathcal{B}_1(P)$ smaller than $\mathcal{B}_2(P)$. The following definitions capture these intuitions:

Definition 16. *Given a Σ-structure $\mathcal{B} = (W, \preceq, I)$, a Σ-program P and a strict bar $B \subseteq W$ w.r.t. $(P, \emptyset)$, we define $B{\downarrow} = \langle W_{B\downarrow}, \preceq, I_{B\downarrow}\rangle$ such that $W_{B\downarrow} = \{v \in W(P) \mid \exists w \in B$ and $v \prec w\}$ and $I_{B\downarrow} = \{I(w) \mid w \in W_{B\downarrow}\}$.*

Let $\mathcal{B}_1$ and $\mathcal{B}_2$ be Σ-structures, P a Σ-program, $\mathcal{B}_1(P)$ and $\mathcal{B}_2(P)$ the ordered structures associated to P in $\mathcal{B}_1$ and $\mathcal{B}_2$, respectively. Let $B_i \subseteq W_i(P)$, $i \in \{1, 2\}$ be strict bars w.r.t. $(P, \emptyset)$. Then, B_1 and B_2 are separator bars w.r.t. $\mathcal{B}_1(P)$ and $\mathcal{B}_2(P)$ if, and only if, $B_1{\downarrow} = B_2{\downarrow}$ and for any other strict bars $B'_i \subseteq W_i(P)$ w.r.t. $(P, \emptyset)$ such that $\forall v_i \in B_i\ \exists v'_i \in B'_i$: $v_i \preceq v'_i$, $i \in \{1, 2\}$, $B'_1{\downarrow} \neq B'_2{\downarrow}$.

Separator bars are the bars, discussed above, that define where the differences in two ordered structures start. Separator bars are uniquely determined:

Lemma 17. *Let $\mathcal{B}_1 = (W_1, \preceq, I_1)$ and $\mathcal{B}_2 = (W_2, \preceq, I_2)$ be Σ-structures, P a Σ-program and $\mathcal{B}_1(P)$, $\mathcal{B}_2(P)$ the ordered structures associated to P in $\mathcal{B}_1$ and $\mathcal{B}_2$, and B_1, B_2 separator bars w.r.t. $\mathcal{B}_1(P)$ and $\mathcal{B}_2(P)$. Then B_1 and B_2 are unique.*

Notice that given a Σ-structure $\mathcal{B}$, each bar B w.r.t. $(P, \emptyset)$ in $\mathcal{B}(P)$ induces a substructure $\langle W_{B\downarrow} \cup B, \preceq, I_{B\downarrow} \cup \{I(w) \mid w \in B\}\rangle$ which corresponds to an initial segment of the ordered structure $\mathcal{B}(P)$. For simplicity, in what follows, we will refer to this substructure as $B{\downarrow} \cup B$.

Definition 18. *Let $\mathcal{B}_1$ and $\mathcal{B}_2$ be Σ-structures, P a Σ-program and $\mathcal{B}_1(P)$ and $\mathcal{B}_2(P)$ the ordered structures associated to P in $\mathcal{B}_1$ and $\mathcal{B}_2$, respectively.*

1. *Let $B_i \subseteq W_i(P)$ be strict bars w.r.t $(P, \emptyset)$, $i \in \{1, 2\}$, $B_1 \equiv B_2$ if, and only if, $B_1 = B_2$ and $\forall w \in B_1 : I_1(w) = I_2(w)$*
2. *Given two strict bars $B_i \subseteq W_i(P)$ w.r.t $(P, \emptyset)$, $i \in \{1, 2\}$, $B_1 \sqsubseteq_b B_2$ if, and only if, $B_1 \equiv B_2$ or one of the following conditions holds:*
   (a) $\forall w_1 \in B_1 \forall w_2 \in B_2 : L_{w_2} \subset L_{w_1}$
   (b) $B_1 = B_2$ and $\forall w \in B_1 : I_1(w) \subset I_2(w)$.
   The strict order associated to this definition is denoted $\sqsubset_b$.
3. *$\mathcal{B}_1(P) \sqsubseteq_s \mathcal{B}_2(P)$ if, and only if, there exist separator bars $B_i \subseteq W_i(P)$, $i \in \{1, 2\}$ w.r.t. $\mathcal{B}_1(P)$ and $\mathcal{B}_2(P)$ such that (a) or (b) holds:*

(a) $B_1 \equiv B_2$ *and* $\mathcal{B}_2(P) = B_1{\downarrow} \cup B_1$
(b) $B_1 \sqsubset_b B_2$
The strict order associated to this definition is denoted $\sqsubset_s$.

Theorem 19. *The relation* $\sqsubseteq_b$ *over strict bars in ordered structures associated to a* Σ*-program in a* Σ*-structure is a partial order.*

Proof. Reflexivity is straightforward and transitivity is a quite direct consequence of transitivity of $\subseteq$. To prove antisymmetry assume that $B_i \subseteq W_i(P)$, $i \in \{1, 2\}$ are strict bars w.r.t $(P, \emptyset)$, such that $B_1 \sqsubseteq_b B_2$ and $B_2 \sqsubseteq_b B_1$ but they do not satisfy $B_2 \equiv B_1$. Then, B_1 and B_2 only can be comparable by 18.2a. Then, $\forall w_1 \in B_1\ \forall w_2 \in B_2 :\ L_{w_2} \subset L_{w_1}$ and $L_{w_1} \subset L_{w_2}$. This is a contradiction. ■

Theorem 20. *The relation* $\sqsubseteq_s$ *over ordered structures associated to a* Σ*-program* P *in a* Σ*-structures is a partial order.*

Proof. Reflexivity holds trivially. To prove antisymmetry suppose $\mathcal{B}_1(P) \sqsubseteq_s \mathcal{B}_2(P)$ and $\mathcal{B}_2(P) \sqsubseteq_s \mathcal{B}_1(P)$. By lemma 17, both relationships are established by the same separator bars B_1 and B_2 and $B_1 \sqsubseteq_b B_2$ and $B_2 \sqsubseteq_b B_1$. So, $B_1 \equiv B_2$ and, $\mathcal{B}_1(P) = \mathcal{B}_2(P)$ by 18.3a ($\mathcal{B}_1(P) = B_1{\downarrow} \cup B_1$ and $\mathcal{B}_2(P) = B_2{\downarrow} \cup B_2$). Transitivity is a consequence of lemma 17 and of transitivity of $\sqsubseteq_b$. ■

Example 21. In Example 9, we can see that $\mathcal{B}_1(\emptyset) \sqsubseteq_s \mathcal{B}_2(\emptyset)$. In this case, the separator bars are $B_1 = \{(\emptyset, \{p, q\})\}$ and $B_2 = \{(\emptyset, \{q\})\}$, and $B_1 \sqsubseteq_b B_2$ because $\{q\} \subset \{p, q\}$ Also, $\mathcal{B}_1(\{p :- \neg q\}) \sqsubseteq \mathcal{B}_2(\{p :- \neg q\})$. Here, the separator bars are $B'_1 = B'_2 = \{(\{p :- \neg q\}, \emptyset)\}$ and $B'_1 \sqsubseteq_b B'_2$ because $I_1((\{p :- \neg q\}, \emptyset)) = \emptyset \subset I_2((\{p :- \neg q\}, \emptyset)) = \{p, q, r, s\}$.

Now, we may define the order relation between Σ-structures. One obvious possible definition for such an ordering would consist in saying that $\mathcal{B}_1$ is smaller than $\mathcal{B}_2$ if for every program P, $\mathcal{B}_1(P)$ is smaller than $\mathcal{B}_2(P)$. However, this definition would not be adequate. The problem is that, to decide what there should be in a given world for a program P we may need to look what information is included in the worlds associated to a different program P'. The reason is that a certain clause in P may include an intuitionistic implication. This means that before comparing the ordered structures associated to P one should compare the ordered structures associated to P'. Again, this means that the ordering over Σ-structures should be a kind of lexicographic extension of the ordering on the structures associated to programs.

Definition 22. *Given* $\mathcal{B}_i \in Struct(\Sigma)$, $i \in \{1, 2\}$, $\mathcal{B}_1 \sqsubseteq \mathcal{B}_2$ *if, and only if, for every* Σ*-program* P *one of the following conditions holds:*

1. $\mathcal{B}_1(P) \sqsubseteq_s \mathcal{B}_2(P)$
2. *There exists a program* P', $P \subseteq P' \subseteq flat(P)$ *such that* $\mathcal{B}_1(P') \sqsubseteq_s \mathcal{B}_2(P')$

where $flat(P)$ denotes the program consisting of all the clauses in P and in all the programs $flat(Q)$, where Q occurs in the left-hand-side of a clause in P.

Theorem 23. *The relation $\sqsubseteq$ is a partial order in $Struct(\Sigma)$.*

Proof. Reflexivity is a consequence of reflexivity of $\sqsubseteq_s$. To prove antisymmetry we can consider the four cases resulting of combining 22.1 and 22.2. Combining just 22.1 directly leads to the equality of structures, and, it is not difficult to see that any other case leads to a contradiction. Again we can consider the four cases to prove transitivity. Now, combining just 22.2 let to a contradiction, and, the other cases hold by transitivity of $\sqsubseteq_s$. ■

Theorem 24. *For any Σ-program P, $T_P{\uparrow}\omega$ is the $\sqsubseteq$-least model in $Mod(P)$.*

Proof. The proof uses double induction on the iterations of T_P to prove that for every $\mathcal{B} \in Mod(P)$ and for every $k \in I\!N$, $T_P^k(\bot) \sqsubseteq \mathcal{B}$; and, then, on the $\subseteq$-chain of programs (ordered structures) in the Σ-structures $T_P^k(\bot)$ and $\mathcal{B}$ to prove that conditions 22.1 and 22.2 hold. The base case is quite simple. The proof of $T_P^{k+1}(\bot) \sqsubseteq \mathcal{B}$ from $T_P^k(\bot) \sqsubseteq \mathcal{B}$ is a little long due to technicalities but the essential idea is quite simple and direct: It is enough to consider that the definition of T_P for worlds and interpretations are, respectively, if-and-only-if versions of the notions of supported models and local forcing of clauses in Definition 7. ■

7 Soundness and Completeness

In this section we will prove the equivalence between the operational and the model-theoretic semantics defined above. In particular this means showing the soundness and completeness of the operational semantics of a program P with respect to the least fixpoint of the immediate consequence operator T_P.

Theorem 25 (Soundness). *If $P \vdash \overline{G}\Box c$ can be proved with computed answer c', then $(\emptyset, \emptyset), T_P{\uparrow}\omega \Vdash \overline{G}\Box c'$.*

Proof. We prove that for every Σ-program Q, if $P \cup Q \vdash \overline{G}\Box c$ can be proved with computed answer c', then $(Q, \emptyset), T_P{\uparrow}\omega \Vdash \overline{G}\Box c'$. The proof proceeds by induction on the number of derivation (plus subderivations) steps. The base step, when the Σ-expression in the goal is of the form $\Box c$, is trivial. To prove the inductive step there are four cases to consider depending if the goal is negative and if it has embbeded implication. The proofs for each of these cases are very similar and standard. Using the corresponding operational rule, the inductive hypothesis and the definition of T_P we conclude that $(Q, \emptyset), T_P{\uparrow}\omega \Vdash \overline{G}\Box c'$. ■

Theorem 26 (Completeness). *If $(\emptyset, \emptyset), T_P{\uparrow}\omega \Vdash \overline{G}\Box c$, then $P \vdash \overline{G}\Box c$ can be proved with computed answers $c_1, \ldots, c_n$ such that $FET_\Sigma \models (c \to \bigvee_{i=1}^n c_i)^\forall$.*

Proof. As in the previous theorem, we prove that for every Σ-program Q if $(Q, \emptyset), T_P{\uparrow}\omega \Vdash \overline{G}\Box c$, then $P \cup Q \vdash \overline{G}\Box c$ can be proved with computed answers $c_1, \ldots, c_n$ such that $FET_\Sigma \models (c \rightarrow \bigvee_{i=1}^{n} c_i)^\forall$. Again the proof is very standard and uses induction on the iterations of T_P. The base step, $k = 0$, is trivial. For proving the inductive step (in the same four cases) it is enough to use the corresponding operational rule, the inductive hypothesis and the definition of T_P. ■

Acknowledgements

This work has been partially supported by the CICYT project HEMOSS (ref. TIC98-0949-C02-01) and CIRIT GRC 1999SGR-150.

References

1. A. J. Bonner and L. T. McCarty. Adding negation-as-failure to intuitionistic logic programming. In *Proc. of the North American Conf. on Logic Programming*, 681–703, 1990.
2. M. Bugliesi, E. Lamma, and Mello Paola. Modularity in logic programming. *Journal of Logic Programming*, 19,20:443–502, 1994.
3. W. Drabent. What is a failure? An approach to constructive negation. *Acta Informática*, 32:27–59, 1995.
4. S. Etalle and F. Teusink. A compositional semantics for normal open programs. In *Proc. Int. Conf. and Symp. on Logic Programming'96*. The MIT Press, 1996.
5. F. Fages. Constructive negation by pruning. *J. of Logic Programming*, 32:85–118, 1997.
6. G. Ferrand and A. Lallouet. A compositional proof method of partial correctness for normal logic programs. In *Int. Logic Programming Symp.*, pages 209–223. J. Lloyd, ed., 1995.
7. L. Giordano and N. Olivi. Combining negation as failure and embedded implication in logic programs. *Journal of Logic Programming*, (36):91–147, 1998.
8. P. Lucio, F. Orejas, and E. Pino. An algebraic framework for the definition of compositional semantics of normal logic programs. *Journal of Logic Programming*, 40:89–123, 1999.
9. D. Miller. A logical analysis of modules in logic programming. *Journal of Logic Programming*, 6:79–108, 1989.
10. F. Orejas, E. Pino, and H. Ehrig. Institutions for logic programming. *Theoretical Computer Science*, 173:485–511, 1997.
11. E. Pasarella, E. Pino, and F. Orejas. Constructive negation without subsidiary trees. In Alpuente M., editor, *Proceedings of the 9th International Workshop on Functional and Logic Programming, WFLP'2000.*, pages 195–209, 2000.
12. J.C. Shepherson. Negation in logic programming. In J. Minker, editor, *Foundations on Deductive Databases and Logic Programs*, pages 19–88. Morgan Kaufmann, 1988.
13. P. J. Stuckey. Negation and constraint logic programmming. *Information and Computation*, 118:12–23, 1995.
14. D. van Dalen. Intuitionistic logic. In D. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic – Vol III*, 1986.

A Multi-adjoint Logic Approach to Abductive Reasoning

Jesús Medina[1], Manuel Ojeda-Aciego[1], and Peter Vojtáš[2]

[1] Dept. Matemática Aplicada. Universidad de Málaga.*
{jmedina,aciego}@ctima.uma.es
[2] Dept. Mathematical Informatics. P.J. Šafárik University.**
vojtas@kosice.upjs.sk

Abstract. Multi-adjoint logic programs has been recently introduced [9, 10] as a generalization of monotonic logic programs [2, 3], in that simultaneous use of several implications in the rules and rather general connectives in the bodies are allowed.
This paper discusses abductive reasoning—that is, reasoning in which explanatory hypotheses are formed and evaluated. To model uncertainty in human cognition and real world applications; we use multi-adjoint logic programming to introduce and study a model of abduction problem.

1 Introduction

Broadly speaking, abduction aims at finding explanations for, or causes of, observed phenomena or facts; it is inference to the best explanation, a pattern of reasoning that occurs in such diverse places as medical diagnosis, scientific theory formation, accident investigation, language understanding, and jury deliberation. More formally, abduction is an inference mechanism where given a knowledge base and some observations, the reasoner tries to find hypotheses which together with the knowledge base explain the observations. Reasoning based on such an inference mechanism is referred to as abductive reasoning.

Abductive reasoning has been recognized as an important form of reasoning with incomplete information that is appropriate for many problems in Artificial Intelligence. These problems include updates in databases, belief revision, planning, diagnosis, natural language understanding, default reasoning, user modelling and, in general, problems requiring reasoning with incomplete information.

The purpose of this work is to provide a theoretical framework for abduction in multi-adjoint logic programming [9]. The special feature of multi-adjoint logic programs is that it is possible to use a number of different implications in the rules of our programs. Specifically, the language and semantics of monotonic logic programs are generalized in order to encompass more complex rules. For

* Partially supported by Spanish DGI project BFM2000-1054-C02-02 and Junta de Andalucía project TIC-115.
** and partially supported by Slovak project VEGA 1/7557/20

P. Codognet (Ed.): ICLP 2001, LNCS 2237, pp. 269–283, 2001.

simplicity in the presentation, only the propositional (ground) case will be considered. A whole class of abduction problems with uncertainty expressed within the language of multiadjoint programs can be solved by our method.

A general theory of logic programming which allows the simultaneous use of different implications in the rules and rather general connectives in the bodies is presented in [9], where models of these programs are proved to be post-fixpoints of the immediate consequences operator, which turns out to be monotonic under very general hypotheses. In addition, the continuity of the immediate consequences operator is studied, and some sufficient conditions for its continuity are obtained. A procedural semantics, under these conditions, for multi-adjoint logic programs, together its completeness result was given in [10].

The structure of the paper is as follows: In Section 2, the preliminary definitions are introduced; later, in Section 3, the syntax and semantics of multi-adjoint logic programs are given, and the results about the continuity of the immediate consequences operator are presented. In Section 4, the procedural semantics of multi-adjoint logic programs is defined and the completeness results are stated. Section 5 is the main part of this paper. We give definitions of the abduction problem and of correct and computed explanations. We prove soundness and completeness of our abduction semantics. The computation of the cheapest explanation wrt a price function can be implemented, in determined lattices, by a logic programming computation followed by a linear programming optimization. The paper finishes with some conclusions and pointers to future work.

2 Preliminary Definitions

In order to make this paper as self-contained as possible, the preliminary definitions required to formally define multi-adjoint logic programs are given in this section, which contains the approach given in [2,3,9].

We assume the reader is familiar to constructions and terminology of universal algebra such as graded set, Ω-algebra and subalgebra of an Ω-algebra, which are used to define formally the syntax and the semantics of the languages we will deal with.

The main concept we use in this section is that of *adjoint pair*, firstly introduced in a logical context by Pavelka [12], who interpreted the poset structure of the set of truth-values as a category, and the relation between the connectives of implication and conjunction as functors in this category. The result turned out to be another example of the well-known concept of adjunction, introduced by Kan in the general setting of category theory in 1950 (see also the notion of a relative pseudo-complement in lattice theory e.g. in Rasiowa and Sikorski's 'Mathematics of matamathematics' (1968)).

Definition 1 (Adjoint pair). *Let $\langle P, \preceq\rangle$ be a partially ordered set and $(\leftarrow, \&)$ a pair of binary operations in P such that:*

(a1) Operation $\&$ is increasing in both arguments, i.e. if $x_1, x_2, y \in P$ such that $x_1 \preceq x_2$ then $(x_1 \,\&\, y) \preceq (x_2 \,\&\, y)$ and $(y \,\&\, x_1) \preceq (y \,\&\, x_2)$;

(a2) *Operation* $\leftarrow$ *is increasing in the first argument (the consequent) and decreasing in the second argument (the antecedent), i.e. if* $x_1, x_2, y \in P$ *such that* $x_1 \preceq x_2$ *then* $(x_1 \leftarrow y) \preceq (x_2 \leftarrow y)$ *and* $(y \leftarrow x_2) \preceq (y \leftarrow x_1)$*;*
(a3) *For any* $x, y, z \in P$*, we have that* $x \preceq (y \leftarrow z)$ *holds if and only if* $(x \,\&\, z) \preceq y$ *holds.*

Then we say that $(\leftarrow, \&)$ *forms an* adjoint pair *in* $\langle P, \preceq \rangle$.

The property (a3) corresponds to the categorical adjointness; and can be adequately interpreted in terms of multiple-valued inference as both the assertion that the truth-value of $y \leftarrow z$ is the maximal x satisfying $x \,\&\, z \preceq_P y$, and the validity of a generalized modus ponens rule [5].

Extending the results in [2,3,13,14] to a more general setting, in which different implications (Łukasiewicz, Gödel, product) and thus, several modus ponens-like inference rules are used, naturally leads to considering several *adjoint pairs* in the lattice. More formally,

Definition 2 (Multi-Adjoint Semilattice). *Let* $\langle L, \preceq \rangle$ *be a cus-lattice. A* multi-adjoint semilattice $\mathcal{L}$ *is a tuple* $(L, \preceq, \leftarrow_1, \&_1, \ldots, \leftarrow_n, \&_n)$ *satisfying the following items:*

(l1) $\langle L, \preceq \rangle$ *is bounded, i.e. it has bottom* $(\bot)$ *and top* $(\top)$ *elements;*
(l2) $(\leftarrow_i, \&_i)$ *is an adjoint pair in* $\langle L, \preceq \rangle$ *for* $i = 1, \ldots, n$*;*
(l3) $\top \,\&_i\, \vartheta = \vartheta \,\&_i\, \top = \vartheta$ *for all* $\vartheta \in L$ *for* $i = 1, \ldots, n$.

Remark 1. Note that residuated lattices are a special case of multi-adjoint semilattice, in which the underlying poset has a cus-lattice structure, has monoidal structure wrt $\otimes$ and $\top$, and only one adjoint pair is present.

From the point of view of expressiveness, it is interesting to allow extra operators to be involved with the operators in the multi-adjoint semilattice. The structure which captures this possibility is that of a multi-adjoint algebra.

Definition 3 (Multi-Adjoint Ω-Algebra). *Let* Ω *be a graded set containing operators* $\leftarrow_i$ *and* $\&_i$ *for* $i = 1, \ldots, n$ *and possibly some extra operators, and let* $\mathfrak{L} = (L, I)$ *be an* Ω*-algebra whose carrier set* L *is a cus-lattice under* $\preceq$.

We say that $\mathfrak{L}$ *is a* multi-adjoint Ω-algebra *with respect to the pairs* $(\leftarrow_i, \&_i)$ *for* $i = 1, \ldots, n$ *if* $\mathcal{L} = (L, \preceq, I(\leftarrow_1), I(\&_1), \ldots, I(\leftarrow_n), I(\&_n))$ *is a multi-adjoint semilattice.*

In practice,the extra operators will be assumed to be either conjunctors or disjunctors or aggregators.

Example 1. Consider $\Omega = \{\leftarrow_P, \&_P, \leftarrow_G, \&_G, \wedge_L, @\}$, the real unit interval $U = [0, 1]$ with its lattice structure, and the interpretation function I defined as:

$$I(\leftarrow_P)(x, y) = \min(1, x/y) \qquad I(\&_P)(x, y) = x \cdot y$$

$$I(\leftarrow_G)(x, y) = \begin{cases} 1 & \text{if } y \leq x \\ x & \text{otherwise} \end{cases} \qquad I(\&_G)(x, y) = \min(x, y)$$

$$I(@)(x, y, z) = \tfrac{1}{6}(x + 2y + 3z) \qquad I(\wedge_L)(x, y) = \max(0, x + y - 1)$$

that is, connectives are interpreted as product and Gödel connectives, a weighted sum and Łukasiewicz conjunction; then $\langle U, I\rangle$ is a multi-adjoint Ω-algebra with one aggregator and one additional conjunctor (denoted $\wedge_L$ to make explicit that its adjoint implicator is not in the language). □

The syntax of the propositional languages we will work with will be defined by using the concept of Ω-algebra. To begin with, the concept of alphabet of the language is introduced below.

Definition 4 (Alphabet). *Let Ω be a graded set, and Π a countably infinite set. The* alphabet *$A_{\Omega,\Pi}$ associated to Ω and Π is defined to be the disjoint union $\Omega \cup \Pi \cup S$, where S is the set of auxiliary symbols "(", ")" and ",".*

In the following, we will use only A_Ω to designate an alphabet, for deleting the reference to Π cannot lead to confusion.

Definition 5 (Expressions). *Given a graded set Ω and alphabet A_Ω. The Ω-algebra $\mathfrak{E} = \langle {A_\Omega}^*, I\rangle$ of* expressions *is defined as follows:*

1. *The carrier ${A_\Omega}^*$ is the set of strings over A_Ω.*
2. *The interpretation function I satisfies the following conditions for strings $a_1, \ldots, a_n$ in ${A_\Omega}^*$:*
   - *$c_{\mathfrak{E}} = c$, where c is a constant operation ($c \in \Omega_0$).*
   - *$\omega_{\mathfrak{E}}(a_1) = \omega\, a_1$, where ω is an unary operation ($\omega \in \Omega_1$).*
   - *$\omega_{\mathfrak{E}}(a_1, a_2) = (a_1 \omega\, a_2)$, where ω is a binary operation ($\omega \in \Omega_2$).*
   - *$\omega_{\mathfrak{E}}(a_1, \ldots, a_n) = \omega(a_1, \ldots, a_n)$, where ω is a n-ary operation ($\omega \in \Omega_n$) and $n > 2$.*

Note that a expression is only a string of letters of the alphabet, that is, it needn't be a well-formed formula. Actually, the well-formed formulas is the subset of the set of expressions defined as follows:

Definition 6 (Well-formed formulas). *Let Ω be a graded set, Π a countable set of propositional symbols and $\mathfrak{E}$ the algebra of expressions corresponding to the alphabet $A_{\Omega,\Pi}$. The* well-formed formulas *(in short, formulas) generated by Ω over Π is the least subalgebra $\mathfrak{F}$ of the algebra of expressions $\mathfrak{E}$ containing Π.*

The set of formulas, that is the carrier of $\mathfrak{F}$, will be denoted F_Ω. It is well-known that least subalgebras can be defined as an inductive closure, and it is not difficult to check that it is freely generated, therefore it satisfies the unique homomorphic extension theorem.

3 Syntax and Semantics of Multi-adjoint Logic Programs

Multi-adjoint logic programs are constructed from the abstract syntax induced by a multi-adjoint algebra on a set of propositional symbols. Specifically, we will consider a multi-adjoint Ω-algebra $\mathfrak{L}$ whose extra operators are either conjunctors, denoted $\wedge_1, \ldots, \wedge_k$, or disjunctors, denoted $\vee_1, \ldots, \vee_l$, or aggregators,

denoted $@_1, \ldots, @_m$. (This algebra will host the manipulation the truth-values of the formulas in our programs.)

In addition, let Π be a set of propositional symbols and the corresponding algebra of formulas $\mathfrak{F}$ freely generated from Π by the operators in Ω. (This algebra will be used to define the syntax of a propositional language.)

Remark 2. As we are working with two Ω-algebras, and to discharge the notation, we introduce a special notation to clarify which algebra an operator belongs to, instead of continuously using either $\omega_{\mathfrak{L}}$ or $\omega_{\mathfrak{F}}$. Let ω be an operator symbol in Ω, its interpretation under $\mathfrak{L}$ is denoted $\dot{\omega}$ (a dot on the operator), whereas ω itself will denote $\omega_{\mathfrak{F}}$ when there is no risk of confusion.

The definition of multi-adjoint logic program is given, as usual, as a set of rules and facts. The particular syntax of these rules and facts is given below:

Definition 7 (Multi-Adjoint Logic Programs). *A* multi-adjoint logic program *is a set $\mathbb{P}$ of rules of the form $\langle (A \leftarrow_i \mathcal{B}), \vartheta \rangle$ such that:*

1. *The* rule *$(A \leftarrow_i \mathcal{B})$ is a formula of $\mathfrak{F}$;*
2. *The* confidence factor *ϑ is an element (a truth-value) of L;*
3. *The* head *of the rule A is a propositional symbol of Π.*
4. *The* body *formula $\mathcal{B}$ is a formula of $\mathfrak{F}$ built from propositional symbols $B_1, \ldots, B_n$ ($n \geq 0$) by the use of conjunctors $\&_1, \ldots, \&_n$ and $\wedge_1, \ldots, \wedge_k$, disjunctors $\vee_1, \ldots, \vee_l$ and aggregators $@_1, \ldots, @_m$.*
5. Facts *are rules with body $\top$.*
6. *A* query *(or* goal*) is a propositional symbol intended as a question ?A prompting the system.*

Note that an arbitrary composition of conjunctors, disjunctors and aggregators is also an aggregator.

Sometimes, we will represent the above pair as $A \xleftarrow{\vartheta}_i @[B_1, \ldots, B_n]$, where[1] $B_1, \ldots, B_n$ are the propositional variables occurring in the body and $@$ is the aggregator obtained as a composition.

Example 2. The following program $\mathbb{P}$, where the subscripts G, L, P on the connectives mean Gödel, Łukasiewicz and product connectives, is an example of a [0,1]-valued multi-adjoint logic program consisting of five rules and three facts.

$$\texttt{high\_fuel\_consumption} \xleftarrow{0.8}_G \texttt{rich\_mixture} \wedge_L \texttt{low\_oil} \tag{1}$$

$$\texttt{overheating} \xleftarrow{0.5}_P \texttt{low\_oil} \tag{2}$$

$$\texttt{noisy\_behaviour} \xleftarrow{0.8}_P \texttt{rich\_mixture} \tag{3}$$

$$\texttt{overheating} \xleftarrow{0.9}_L \texttt{low\_water} \tag{4}$$

$$\texttt{noisy\_behaviour} \xleftarrow{1}_P \texttt{low\_oil} \tag{5}$$

$$\texttt{low\_oil} \xleftarrow{0.2}_P \tag{6}$$

$$\texttt{low\_water} \xleftarrow{0.2}_P \tag{7}$$

$$\texttt{rich\_mixture} \xleftarrow{0.5}_P \tag{8}$$

[1] Note the use of square brackets.

This program is intended to represent some general knowledge about the behaviour of a car.

Definition 8 (Interpretation). *An* interpretation *is a mapping $I\colon \Pi \to L$. The set of all interpretations of the formulas defined by the Ω-algebra $\mathfrak{F}$ in the Ω-algebra $\mathfrak{L}$ is denoted $\mathcal{I}_{\mathfrak{L}}$.*

Note that by the unique homomorphic extension theorem, each of these interpretations can be uniquely extended to the whole set of formulas F_Ω.

The ordering $\preceq$ of the truth-values L can be easily extended to the set of interpretations as usual:

Definition 9 (Semilattice of interpretations). *Consider two interpretations $I_1, I_2 \in \mathcal{I}_{\mathfrak{L}}$. Then, $\langle \mathcal{I}_{\mathfrak{L}}, \sqsubseteq \rangle$ is a cus-lattice where $I_1 \sqsubseteq I_2$ iff $I_1(p) \preceq I_2(p)$ for all $p \in \Pi$. The least interpretation $\triangle$ maps every propositional symbol to the least element $\bot$ of L.*

A rule of a multi-adjoint logic program is satisfied whenever the truth-value of the rule is greater or equal than the confidence factor associated with the rule. Formally:

Definition 10 (Satisfaction, Model). *Given an interpretation $I \in \mathcal{I}_{\mathfrak{L}}$, a weighted rule $\langle A \leftarrow_i \mathcal{B}, \vartheta \rangle$ is* satisfied *by I iff $\vartheta \preceq \hat{I}(A \leftarrow_i \mathcal{B})$. An interpretation $I \in \mathcal{I}_{\mathfrak{L}}$ is a* model *of a multi-adjoint logic program $\mathbb{P}$ iff all weighted rules in $\mathbb{P}$ are satisfied by I.*

Note the following equalities

$$\hat{I}(A \leftarrow_i \mathcal{B}) = \hat{I}(A) \mathbin{\dot{\leftarrow}_i} \hat{I}(\mathcal{B}) = I(A) \mathbin{\dot{\leftarrow}_i} \hat{I}(\mathcal{B})$$

and the evaluation of $\hat{I}(\mathcal{B})$ proceeds inductively as usual, till all propositional symbols in $\mathcal{B}$ are reached and evaluated under I. For the particular case of a fact (a rule with $\top$ in the body) satisfaction of $\langle A \leftarrow_i \top, \vartheta \rangle$ means

$$\vartheta \preceq \hat{I}(A \leftarrow_i \top) = I(A) \mathbin{\dot{\leftarrow}_i} \top$$

by property *(a3)* of adjoint pairs this is equivalent to $\vartheta \mathbin{\dot{\&}_i} \top \preceq I(A)$ and this by assumption *(l3)* of multi-adjoint semilattices gives $\vartheta \preceq I(A)$.

Definition 11. *An element $\lambda \in L$ is a* correct answer *for a program $\mathbb{P}$ and a query $?A$ if for an arbitrary interpretation $I\colon \Pi \to L$ which is a model of $\mathbb{P}$ we have $\lambda \preceq I(A)$.*

Example 3. The interpretation I defined by

$$\begin{aligned} I(\texttt{low\_oil}) &= 0.25 \\ I(\texttt{low\_water}) &= 0.35 \\ I(\texttt{overheating}) &= 0.45 \\ I(\texttt{rich\_mixture}) &= 0.90 \\ I(\texttt{noisy\_behaviour}) &= 0.75 \\ I(\texttt{high\_fuel\_consumption}) &= 0.55 \end{aligned}$$

is a model of the program given in Example 2.

If we add the query ?`overheating` to the program, then 0.1 is a correct answer. Actually, it is the greatest correct answer for the query.

The immediate consequences operator, given by van Emden and Kowalski in [15], can be generalised to the framework of multi-adjoint logic programs as follows:

Definition 12. *Let $\mathbb{P}$ be a multi-adjoint logic program. The* immediate consequences operator $T_{\mathbb{P}}^{\mathfrak{L}}: \mathcal{I}_{\mathfrak{L}} \to \mathcal{I}_{\mathfrak{L}}$*, mapping interpretations to interpretations, is defined by considering*

$$T_{\mathbb{P}}^{\mathfrak{L}}(I)(A) = \sup \left\{ \vartheta \,\dot{\&}_i \, \hat{I}(\mathcal{B}) \mid A \xleftarrow{\vartheta}_i \mathcal{B} \in \mathbb{P} \right\}$$

Note that all the suprema involved in the definition do exist because L is assumed to be a cus-lattice.

As it is usual in the logic programming framework, the semantics of a multi-adjoint logic program is characterized by the post-fixpoints of $T_{\mathbb{P}}^{\mathfrak{L}}$.

Theorem 1 ([9]). *An interpretation I of $\mathcal{I}_{\mathfrak{L}}$ is a model of a multi-adjoint logic program $\mathbb{P}$ iff $T_{\mathbb{P}}^{\mathfrak{L}}(I) \sqsubseteq I$.*

Note that the fixpoint theorem works even without any further assumptions on conjunctors (definitely they need not be commutative and associative).

The monotonicity of the operator $T_{\mathbb{P}}^{\mathfrak{L}}$, for the case of only one adjoint pair, has been shown in [2]. The proof for the general case is similar.

Theorem 2 ([9]). *The operator $T_{\mathbb{P}}^{\mathfrak{L}}$ is monotonic.*

Due to the monotonicity of the immediate consequences operator, the semantics of $\mathbb{P}$ is given by its least model which, as shown by Knaster-Tarski's theorem, is exactly the least fixpoint of $T_{\mathbb{P}}^{\mathfrak{L}}$, which can be obtained by transfinitely iterating $T_{\mathbb{P}}^{\mathfrak{L}}$ from the least interpretation $\triangle$.

It is worth to investigate conditions which make the $T_{\mathbb{P}}^{\mathfrak{L}}$ operator to be continuous, in [9] it was proved that whenever every operator in Ω turns out to be continuous in the lattice, then $T_{\mathbb{P}}$ is also continuous and, consequently, its least fixpoint can be obtained by a countably infinite iteration from the least interpretation. Formally,

Theorem 3 ([9]). *If all the operators occurring in the bodies of the rules of a program $\mathbb{P}$ are continuous, and the adjoint conjunctions are continuous in their second argument, then $T_{\mathbb{P}}^{\mathfrak{L}}$ is continuous.*

4 Procedural Semantics of Multi-adjoint Logic Programs

Once shown that the $T_{\mathbb{P}}^{\mathfrak{L}}$ operator can be continuous under very general hypotheses, then the least model can be reached in at most countably many iterations. Therefore, it is worth to define a procedural semantics which allow us to actually construct the answer to a query against a given program.

In the following, we work in a hybrid Ω-algebra made up from the elements of the lattice, and the same alphabet of the language but the adjoint implicators.

For the formal description of the computational model, we will consider an extended the language $\mathfrak{F}'$ defined on the same graded set, but whose carrier is the disjoint union $\Pi \cup L$; this way we can work simultaneously with propositional symbols and with the truth-values they represent.

Definition 13. *Let $\mathbb{P}$ be a multi-adjoint logic program on a multi-adjoint Ω-algebra $\mathfrak{L}$ with carrier L and V the set of truth values of the rules in $\mathbb{P}$. The* extended language $\mathfrak{F}'$ *is the corresponding Ω-algebra of formulas freely generated from the disjoint union of Π and V.*

The formulas in the language $\mathfrak{F}'$ will be referred as *extended formulas.* An operator symbol ω interpreted under $\mathfrak{F}'$ will be denoted as $\bar{\omega}$.

Our computational model will take a query (an atom), and will provide a lower bound of the value of A under any model of the program. Intuitively, the computation proceeds by, somehow, substituting propositional symbols by lower bounds of their truth-value until, eventually, an extended formula with no propositional symbol is obtained, which will be interpreted in the multi-adjoint semilattice to get the computed answer.

Given a program $\mathbb{P}$, we define the following admissible rules for transforming any extended formula.

Definition 14. Admissible rules *are defined as follows:*

1. *Substitute an atom A in an extended formula by $(\vartheta \bar{\&}_i \mathcal{B})$ whenever there exists a rule $\langle A \leftarrow_i \mathcal{B}, \vartheta \rangle$ in $\mathbb{P}$.*
2. *Substitute an atom A in an extended formula by $\bot$.*
3. *Substitute an atom A in an extended formula by ϑ whenever there exists a fact $\langle A \leftarrow_i \top, \vartheta \rangle$ in $\mathbb{P}$.*

Note that if an extended formula turns out to have no propositional symbols, then it can be directly interpreted in the multi-adjoint Ω-algebra $\mathfrak{L}$. This justifies the following definition of *computed answer.*

Definition 15. *Let $\mathbb{P}$ be a program in a multi-adjoint language interpreted on a multi-adjoint semilattice $\mathcal{L}$ and let $?A$ be a goal. An element $\dot{@}[r_1, \ldots, r_m]$, with $r_i \in L$, for all $i \in \{1, \ldots, m\}$ is said to be a* computed answer *if there is a sequence $G_0, \ldots, G_{n+1}$ such that*

1. *$G_0 = A$ and $G_{n+1} = \bar{@}[r_1, \ldots, r_m]$ where $r_i \in L$ for all $i = 1, \ldots n$.*
2. *Every G_i, for $i = 1, \ldots, n$, is a formula in $\mathfrak{F}'$.*
3. *Every G_{i+1} is inferred from G_i by one of the admissible rules.*

The idea of the computation is to consecutively apply admissible rules until an extended formula with no propositional symbols $\bar{@}[r_1, \ldots, r_m]$ is obtained, which can be interpreted as the element $\dot{@}[r_1, \ldots, r_m]$ in the lattice $\mathcal{L}$.

An alternative formalism of Generalized Annotated Logic Programs (GALP) was introduced in [7]. The procedural semantics of GALP uses a CLP-like procedure to solve a set of lattice inequalities to find a computed answer. Our procedural semantics replaces constraints in the form of inequalities by equalities building a final formula for $\dot{@}$ which is the best answer.

It might be the case that for some lattices it is not possible to get the correct answer, simply consider L to be the powerset of a two-element set $\{a, b\}$ ordered by inclusion. The requirement of the reductant property stated below will allow us to avoid these cases, see [7,11] for details.

Definition 16 (Reductant, reductant property). *Let $\mathbb{P}$ be a program on a multi-adjoint Ω-algebra $\mathfrak{F}$ with values in a multi-adjoint semilattice $\mathfrak{L}$; assume that all the rules in $\mathbb{P}$ with head A are $A \xleftarrow{\vartheta_i}_i \mathcal{B}_i$ for $i = 1, \ldots, k$. A* reductant for *A is a rule $A \xleftarrow{\vartheta} @(\mathcal{B}_1, \ldots, \mathcal{B}_n)$ such that for any $b_1, \ldots, b_k$ we have*

$$\sup\{\vartheta_i \,\dot{\&}_i\, b_i \mid i = 1, \ldots, n\} = \vartheta \,\dot{\&}\, \dot{@}(b_1, \ldots, b_k)$$

A program $\mathbb{P}$ is said to have the reductant property *if there exist reductants for any atom A occurring in the head of some rule.*

Note that ϑ and $@$ should depend only on the (multi-)set of $\&_i$.

Certainly, it will be interesting to consider only programs which contain all its reductants, but this might be a too heavy condition on our programs; the following proposition shows that we can assume that our programs contain all the reductants, because the set of models is preserved.

Proposition 1 ([10]). *Any reductant $A \xleftarrow{\vartheta} \mathcal{B}$ of $\mathbb{P}$ is satisfied by any model of $\mathbb{P}$. In short, $\mathbb{P} \models A \xleftarrow{\vartheta} \mathcal{B}$.*

As a consequence of the proposition above, we can assume that a program contains all its reductants, since its set of models is not modified.

Definition 17. *Given a program $\mathbb{P}$ with the reductant property and a query $?A$, the* greatest computed answer *is a computed answer in which calculation admissible rules 1 and 3 are applied only with rules (and facts) reductants in $\mathbb{P}$, and the admissible rule 2 is applied if and only if no rule/fact exists for a given atom in the extended formula.*

Theorem 4. *Given a program $\mathbb{P}$, a query $?A$ and a computed answer λ'. If λ is the greatest computed answer, then $\lambda' \leq \lambda$.*

The theorem above shows that computed answers as in the previous definition are actually the greatest.

Theorem 5. *Given a program $\mathbb{P}$ with the reductant property, for all atom A let λ_A be the greatest computed answer for $\mathbb{P}$ and query $?A$, then $\lambda_A \preceq T_{\mathbb{P}}^{\omega}(\triangle)(A)$.*

It was proved in [10] that, given a program $\mathbb{P}$, then $T_{\mathbb{P}}^{n}(\triangle)(A)$ is a computed answer for all n and for all query $?A$. Now, in conjunction with the result above we straightforwardly obtain the to following corollaries which will be used later.

Corollary 1. *$\lambda \in L$ is the greatest computed answer for program $\mathbb{P}$ and query $?A$ if and only if $\lambda = T_{\mathbb{P}}^{\omega}(\triangle)(A)$.*

To finish the section, simply recall the following result from [10].

Corollary 2. *$\lambda \in L$ is a correct answer for program $\mathbb{P}$ and query $?A$ if and only if $\lambda \preceq T_{\mathbb{P}}^{\omega}(\triangle)(A)$.*

5 Abduction in Multi-adjoint Logic Programs

To state the intuition behind an abduction problem, if will use the program in Example 2, but without the three facts; that is, we have no information about variables $\texttt{rich\_mixture}, \texttt{low\_oil}, \texttt{low\_water}$, which will turn out to be hypotheses to explain the behaviour of our car.

If we notice it is noisy, overheated and has a high fuel consumption, we would like to know why it is so. Let us call these assertions *observation variables* and let us denote it by

$$OV = \{\texttt{noisy\_behaviour}, \texttt{overheated}, \texttt{high\_fuel\_consumption}\}.$$

As noisiness can be subjective and the height of fuel consumption and temperature of the engine can take different (high) values, our observations are estimated (by an expert) by confidence factors. So the second parameter of our abduction problem are observations (sometimes called manifestations, symptoms, effects) represented by a theory, i.e. a partial mapping $OBS{:}\, OV \to L$ consisting of facts, which can be thought of as observation variables equipped with confidence factors

$$\texttt{high\_fuel\_consumption} \overset{0.25}{\leftarrow_P}, \quad \texttt{overheating} \overset{0.25}{\leftarrow_P}, \quad \texttt{noisy\_behaviour} \overset{0.5}{\leftarrow_P}$$

Note that it is not necessary to assume a specific type of implication for observations - any will do. The full strength of multi-adjointness is needed for the logic programming part where specific implication describes a specific action of the truth value of the rule.

We would like to find explanations (causes) for given observations (symptoms) by means of semantical consequence. Namely, explanations are assertions which together with domain knowledge forces every model of them to be also a model of observations. That is, whenever in a real world situation represented by an interpretation I, both domain knowledge and explanations are true, then all observations are true in I.

Possible explanations will be L-fuzzy subsets of the set of hypotheses

$$H = \{\texttt{rich\_mixture}, \texttt{low\_oil}, \texttt{low\_water}\}$$

This makes sense, because in the realm of a theory and of observations suffering from uncertainty, we can expect that certain level of confidence of hypotheses can be (under the presence of the theory) an explanation of these (uncertain) observations. The formal definitions of abduction problem, explanation, etc. are given below.

Definition 18. *An* abduction problem *consists of a tuple* $\mathcal{A} = \langle \mathbb{P}, OBS, H \rangle$, *where*

1. $\mathbb{P}$ *is a multi-adjoint logic program.*
2. $H \subseteq \Pi$ *is the set of hypotheses.*
3. $OBS{:}\, OV \to L$ *is the* L*-fuzzy theory of observations (where* OV *is a set of observation variables such that* $OV \cap H = \varnothing$*)*

The intended meaning of $OV \cap H = \varnothing$ is that observation variables should not be explained by themselves.

A theory is a mapping assigning formulas a truth value. For two theories T and S, let $T \cup S$ denote the union of them as a theory defined by

$$(T \cup S)(A) = \max\{T(A), S(A)\}$$

Definition 19. *An L-fuzzy theory* $E\colon H \to L$ *is a* correct explanation *to an abduction problem* $\mathcal{A} = \langle \mathbb{P}, OBS, H \rangle$ *if*

1. $\mathbb{P} \cup E$ *is satisfiable.*
2. $\mathbb{P} \cup E$ *semantically implies* OBS*, that is every model of* $\mathbb{P} \cup E$ *is also a model of* OBS*.*

It will be useful to represent explanations as a subset of L^n, especially when $L = [0, 1]$. If $H = \{h_1, \ldots, h_n\}$ is the set of hypotheses and $E\colon H \to L$ is an explanation, it is uniquely determined by its values

$$(E(h_1), \ldots, E(h_n)) \in L^n$$

so, an element $\varepsilon = (\varepsilon_1, \ldots, \varepsilon_n) \in L^n$ represents the mapping $E_\varepsilon(h_i) = \varepsilon_i$.

The set of correct explanations for $\mathcal{A}$ will be denoted by $SOL_d(\mathcal{A})$, where the subscript d resembles the declarative character of the explanations.

Using our motivating example we are illustrating a possible solution of a whole class of problems which are formulated within our formalism.

Example 4. Having our motor vehicle example and our multi-adjoint program $\mathbb{P}$ and the query `?high_fuel_consumption` by multi-adjoint logic program computation, using the first program rule we get

$$\min(0.8, \max(0, \texttt{rich\_mixture} + \texttt{low\_oil} - 1))$$

Now similarly as in the two valued logic abductive logic programming [6], our procedure instead of failing in a proof when a selected subgoal fails to unify with the head of any rule, the subgoal is viewed as a hypothesis, that is, if we know confidence factor for rich mixture and low oil (from an explanation) we have the computed answer for high fuel consumption. To fulfil

$$OBS(\texttt{high\_fuel\_consumption}) \geq 0.25$$

it should be

$$\min(0.8, \max(0, \texttt{rich\_mixture} + \texttt{low\_oil} - 1)) \geq 0.25$$

hence

$$\texttt{rich\_mixture} + \texttt{low\_oil} \geq 1.25$$

Note that if $OBS(\texttt{high\_fuel\_consumption})$ would be greater than 0.8, there is no explanation for this. To overcome this we could think of the possibility of calculating the truth value of the metamathematical assertion "E is an explanation for $\mathcal{A}$". Here we consider the case, when this truth value is 1, that is, E is an explanation with full confidence. This is why we are calculating the truth value of the following implication: Whenever I is a model of $\mathbb{P} \cup E$ then I is a model of OBS. In particular the truth value of the implication

$$\text{“If} \quad I(\texttt{rich\_mixture}) + I(\texttt{low\_oil}) > 1.25$$
$$\text{then} \quad I(\texttt{high\_fuel\_consumption}) \geq 0.25\text{”}$$

And again, the truth value of $x \leq y$ can be calculated as $\overset{\bullet}{\leftarrow}(y, x)$.

So our multi-adjoint logic programming abduction should run as a usual logic program with two exceptions

- it successfully ends without resolving variables which are in the set of hypotheses
- it is prompted by a query with threshold, which can serve as a cut (as we will see later).

Definition 20 (Procedural semantics for abduction). *Let us have an abduction problem $\mathcal{A} = \langle \mathbb{P}, OBS, H \rangle$ and consider $m \in OV$. A* successful abduction *for $\mathcal{A}$ and m is a sequence $\mathcal{G} = (G_0, G_1, \ldots, G_l)$ of extended formulas in the language of multi-adjoint logic program computations such that:*

1. *$G_0 = m$,*
2. *G_l contains only variables from H, and*
3. *(a) For all $i < l$, G_{i+1} is inferred from G_i by one of admissible rules, and*
   (b) For the constant interpretation $I_\top \colon \Pi \to \{\top\}$ the inequality $I_1(G_{i+1}) \geq OBS(m)$ holds.

The last condition is to be understood as a cut, because it allows to estimate the best possible computation of remaining propositional variables.

This definition allows the explanation of a single observation variable. Merging of several observation variables is in the Definition 21 bellow. Solutions are obtained as a combination (intersection) of the above set through all m's in observation variables. Moreover, the set of all solutions is the union through all possible combinations of all possible computational branches for all observation variables.

For $H = \{h_1, \ldots h_n\}$, the expression G_l can be understood as a function of n variables from L^n to L and that is why can denote it by $G_l = \mathcal{G}_m(h_1, \ldots, h_n)$.

Theorem 6 (Existence of solutions). *Let $\mathcal{A} = \langle \mathbb{P}, OBS, H \rangle$ be an abduction problem and assume that for each $m \in OV$ there is a successful abduction for $\mathcal{A}$ and m. Then $SOL_d(\mathcal{A}) \neq \varnothing$.*

Our definition of abduction gives us the possibility to define computed explanations for abduction problems $\mathcal{A}$.

Definition 21. *A tuple $\varepsilon = (\varepsilon_1, \ldots, \varepsilon_n)$ is* a computed explanation *for an abduction problem $\mathcal{A} = \langle \mathbb{P}, OBS, H \rangle$ if for every $m \in OV$ there is an abduction $\mathcal{G}_m$ for $\mathcal{A}$ and m such that*

$$\mathcal{G}_m(\varepsilon_1, \ldots, \varepsilon_n) \geq OBS(m)$$

The set of all computed explanations will be denoted by $SOL_p(\mathcal{A})$, where the subscript p resembles the procedural character of this definition.

Example 5. In our motor vehicle example we calculated that first rule of $\mathbb{P}$ gives $\texttt{rich\_mixture} + \texttt{low\_oil} \geq 1.25$, similarly second rule gives $\texttt{low\_oil} \geq 0.5$ and the third gives $\texttt{rich\_mixture} \geq 0.625$, hence the set (coordinates are ordered as $(\texttt{rich\_mixture}, \texttt{low\_oil}, \texttt{low\_water})$)

$$\{(\varepsilon_1, \varepsilon_2, \varepsilon_3) \in [0,1]^3 : \varepsilon_1 + \varepsilon_2 \geq 1.25 \text{ and } \varepsilon_1 \geq 0.625 \text{ and } \varepsilon_2 \geq 0.5\}$$

is a subset of $SOL_p(\mathcal{A})$.

This example shows also that the area got as an intersection of some computational branches for m's has the shape of a convex body. Moreover, the set of all solutions is the union of such areas. It seems reasonable that, to get the cheapest answer, we have just to run a linear programming optimization separately on each of these areas.

Theorem 7 (Soundness). *Assume $\mathcal{A}$ is a definite abduction problem, then $SOL_p(\mathcal{A}) \subseteq SOL_d(\mathcal{A})$ (that is, every computed explanation for $\mathcal{A}$ is also a correct explanation).*

In the completeness theorem below we need the assumption that our logical program has a finite computational tree, according to abductions. This is very often the case in practical applications of abduction, because e.g. observations should not be explained by themselves, and most of logic programs for abduction are layered. Moreover if the conjunctors are archimedean (also very often) then the abduction ends below the observation value threshold, and hence is cut.

Theorem 8 (Completeness). *Assume $\mathcal{A}$ is an abduction problem and the logical program has a finite computational tree according to abductions, then $SOL_d(\mathcal{A}) \subseteq SOL_p(\mathcal{A})$ (that is, every correct explanation for A is also a computed explanation).*

From now on we do not have to distinguish between two sets of solutions and we simply denote it $SOL(\mathcal{A})$.

We comment here very briefly the possibility of using linear programming in some cases to obtain the cheapest explanation to an abduction problem wrt a given cost function.

Example 6. Continuing the example of motor vehicles assume that checking $(\varepsilon_1, \varepsilon_2, \varepsilon_3) \in SOL(\mathcal{A})$ costs $2\varepsilon_1 + \varepsilon_2 + 0.1\varepsilon_3$. The space of solutions $SOL(\mathcal{A})$ is bounded by linear surfaces in $[0,1]^3$ and is the union of four convex bodies, obtained from the combinations of rules in the program for each observation variable.

Using the first three rules of the program $\mathbb{P}$ we get one of these four convex bodies, namely the set

$$S_1 = \{(\varepsilon_1, \varepsilon_2, \varepsilon_3) \in [0,1]^3 \mid \varepsilon_1 + \varepsilon_2 \geq 1.25 \text{ and } \varepsilon_1 \geq 0.625 \text{ and } \varepsilon_2 \geq 0.5\}$$

Applying a linear programming method for S_1 wrt our cost function we get in this convex body a minimal solution $(0.625, 0.625, 0)$ at cost of 1.875.

Actually, the cheapest solution of our running abduction problem $\mathcal{A}$ is $\varepsilon_{min} = (0.25, 1, 0.35)$ at cost of 1.535, obtained from the convex body of all solutions to the first, fourth and fifth program rule.

Eiter and Gottlob showed in [4] that deciding $SOL_d(\mathcal{A}) \neq \varnothing$ for two valued logic is complete for complexity classes at the second level of polynomial hierarchy, while the use of priorisation raises the complexity to the third level in certain cases (for arbitrary propositional theories).

One can ask how difficult is to decide, in our framework, whether $\varepsilon \in SOL(\mathcal{A})$ or not. Now we see that it substantially depends on the complexity of logic programming computation and the complexity of functions evaluating the truth values for connectives; thus, from a computational point of view, it makes sense to use simple connectives (e.g. linear in each coordinate, as product is, or even partly constant). So assuming connectives are easy this problem is in NP.

Regarding the linear programming approach to the cheapest explanations, as linear programming is polynomial and prolog is in NP, to find minimal solutions for an abduction problem (assuming connectives are coordinatewise linear) does not increase the complexity and remains in NP.

6 Conclusions and Future Work

In this work a theoretical framework for abduction in multi-adjoint logic programming is introduced; a sound and complete procedural semantics has been defined, and the possibility of obtaining the cheapest possible explanation to an abduction problem wrt a cost function by means of a logic programming computation followed by a linear programming optimization has been shown.

Future work on this area will be concerned with showing the embedding of different approaches to abduction in our framework, as well as the study of complexity issues in lattices more general than the unit interval.

Acknowledgements

We thank C. Damásio and L. Moniz Pereira for communicating the existence of the Technical Report [2], on which this research began.

References

1. C. V. Damásio and L. M. Pereira. Monotonic and residuated logic programs. Technical report, Dept. Computer Science. Univ. Nova de Lisboa, 2000. Available at `http://centria.di.fct.unl.pt/~cd`.
2. C. V. Damásio and L. M. Pereira. A theory of logic programming. Technical report, Dept. Computer Science. Univ. Nova de Lisboa, 2000.
3. C.V. Damásio and L. Moniz Pereira. Hybrid probabilistic logic programs as residuated logic programs. In *Logics in Artificial Intelligence*, pages 57–73. Lect. Notes in AI, 1919, Springer-Verlag, 2000.
4. T. Eiter and G. Gottlob. The complexity of logic based abduction. *Journal of the ACM*, 42:3–42, 1995.
5. P. Hájek. *Metamathematics of Fuzzy Logic*. Trends in Logic. Studia Logica Library. Kluwer Academic Publishers, 1998.

6. A.C. Kakas, R.A. Kowalski, and F. Toni. The role of abduction in logic programming. In *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 5, pages 235–324. Oxford Univ. Press, 1998.
7. M. Kifer and V. S. Subrahmanian. Theory of generalized annotated logic programming and its applications. *J. of Logic Programming*, 12:335–367, 1992.
8. J.W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, second, extended edition, 1987.
9. J. Medina, M. Ojeda-Aciego, and P. Vojtáš. Multi-adjoint logic programming with continuous semantics. Submitted for publication. Manuscript available at `http://www.satd.uma.es/aciego/TR/malp-tr.pdf`.
10. J. Medina, M. Ojeda-Aciego, and P. Vojtáš. A procedural semantics for multi-adjoint logic programming. Submitted for publication. Manuscript available at `http://www.satd.uma.es/aciego/TR/procsem-tr.pdf`.
11. S. Morishita. A unified approach to semantics of multi-valued logic programs. Technical Report RT 5006, IBM Tokyo, 1990.
12. J. Pavelka. On fuzzy logic I, II, III. *Zeitschr. f. Math. Logik und Grundl. der Math.*, 25, 1979.
13. P. Vojtáš. Fuzzy logic programming. *Fuzzy sets and systems*, 2001. Accepted.
14. P. Vojtáš and L. Paulík. Soundness and completeness of non-classical extended SLD-resolution. In *Proc. Extensions of Logic Programming*, pages 289–301. Lect. Notes in Comp. Sci. 1050, Springer-Verlag, 1996.
15. M. van Emden and R. Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM*, 23(4):733–742, 1976.

Proving Correctness and Completeness of Normal Programs – A Declarative Approach

Włodzimierz Drabent[1] and Mirosława Miłkowska[2,⋆]

[1] Institute of Computer Science, Polish Academy of Sciences, Warszawa, Poland
and IDA, Linköpings universitet, Linköping, Sweden
[2] Institute of Informatics, Warsaw University, Warszawa, Poland
wdr@ida.liu.se, M.Milkowska@mimuw.edu.pl

Abstract. We advocate a declarative approach to proving properties of logic programs. Total correctness can be separated into correctness, completeness and clean termination; the latter includes non-floundering. Only clean termination depends on the operational semantics, in particular on the selection rule. We show how to deal with correctness and completeness in a declarative way, treating programs only from the logical point of view. Specifications used in this approach are interpretations (or theories). We point out that specifications for correctness may differ from those for completeness, as usually there are answers which are neither considered erroneous nor required to be computed.
We present proof methods for correctness and completeness for definite programs and generalize them to normal programs. The considered semantics of normal programs is the standard one, given by the program completion in 3-valued logic.
The method of proving correctness of definite programs is not new and can be traced back to the work of Clark in 1979. However a more complicated approach using operational semantics was proposed by some authors. We show that it is not stronger than the declarative one, as far as properties of program answers are concerned.

1 Introduction

This paper discusses reasoning about logic programs in terms of their declarative semantics. We view total correctness of programs as consisting of correctness, completeness and clean termination. *Correctness* means that any answer obtained from the program satisfies its specification. As logic programming is nondeterministic, one is interested in *completeness*, i.e. that all the results required by the specification are computed. Programs should also (cleanly) *terminate* — computations should be finite and without run-time errors, like floundering and arithmetical exceptions. Obviously, clean termination depends on the operational semantics, in particular on the selection rule. However correctness and completeness do not. It is desirable that they could be dealt with in a declarative way, abstracting from any operational semantics and treating programs

⋆ This research was partly supported by a KBN grant no. 8 T11C 015 15.

P. Codognet (Ed.): ICLP 2001, LNCS 2237, pp. 284–299, 2001.

and their answers only from the logical point of view. This makes it possible to separate reasoning about "logic" and "control".

In this paper we show how to prove correctness and completeness declaratively. We discuss a known method of proving correctness of definite programs and introduce a method for proving completeness. Then we generalize both methods to programs with negation.

The proof method for definite program correctness [Cla79,Hog81,Der93] is simple and straightforward. It is declarative: it abstracts from any operational semantics. It should be well known. However its usefulness is often not appreciated. Instead a more complicated approach using operational semantics was proposed by some authors [BC89,Apt97,PR99]. That approach takes into account the form of atoms selected under LD-resolution. We show that the operational approach is not stronger than the declarative one, as far as properties of program answers are concerned.

The following observation is important for our approach: one should not require using the same specification for both correctness and completeness. This is natural, as there usually are answers which are neither considered erroneous nor required to be computed. Using the same specification requires making decisions like "is $append([\,], 7, 7)$ correct?"; this brings substantial and unnecessary complications. So there is some 3-valued flavour even in logic programming without negation [Nai00].

The paper consists of two main chapters. The first is devoted to definite programs, the second to normal programs. In each case we first discuss proving correctness, then completeness. A comparison with the operational approach to proving correctness follows the section on definite programs correctness. An additional section contains example proofs for normal programs. The paper is concluded by a section on related work. Unabridged, preliminary version of this paper can be found in [DM01].

2 Preliminaries

For basic definitions we refer the reader to [Llo87] and to [Apt97,Doe94]. We consider the declarative semantics given by 3-valued logical consequence of program completion [Kun87] (however our proof methods use only 2-valued logic). This is a standard semantics for normal programs with finite failure [Doe94]. It is a generalization of the classical semantics for definite programs. SLDNF-resolution is sound for it and important completeness results exist.

By a computed (resp. correct) answer we mean an instance $Q\theta$ of a query Q, where θ is a computed (correct) answer substitution for Q and the given program (and Q is a sequence of literals). Notice that, by soundness and completeness of SLD-resolution, the sets of computed and of correct answers for a given program (and arbitrary queries) are equal. So in the case of definite programs we often do not distinguish between these two kinds of answers.

We represent interpretations as sets: Herbrand interpretations as sets of ground atoms, non Herbrand ones as sets of constructs of the form $p(\ldots)$, where p is a predicate symbol (cf. [Llo87, p. 12], [Doe94, p. 124]).

3 Reasoning about Definite Programs

First we discuss correctness. We show a way of proving program correctness w.r.t. specifications. In the next section we compare it with an approach related to operational semantics and show that it is not weaker. Then we show a way of proving completeness.

3.1 Correctness of Definite Programs

We begin with a brief discussion on specifications. As a standard example let us take the program APPEND:

$$\begin{array}{l} app(\,[\,],L,L\,) \leftarrow \\ app(\,[H|K],L,[H|M]\,) \leftarrow app(\,K,L,M\,) \end{array}$$

We want to prove that it indeed appends lists. We need a precise statement (a specification) of this property. A slight complication is that the program does not actually define the relation of list concatenation, but its superset; the least Herbrand model contains atoms like $app([\,],1,1)$. This is a common phenomenon in logic programming, the least model contains "ill-typed" atoms which are irrelevant for the correctness of the program.

So we want to prove that:

For any answer $app(k,l,m)$, if k and l are lists[1] then m is a list and $k * l = m$.

(By a list we mean a term $[t_1,\ldots,t_n]$ (in Prolog notation), where $n \geq 0$ and $t_1,\ldots,t_n$ are possibly non ground terms. Symbol $*$ denotes the list concatenation.) This property could be equivalently expressed as

$$spec \models app(k,l,m) \tag{1}$$

for any answer $app(k,l,m)$, where *spec* is the Herbrand interpretation:

$$spec = \{\, app(k,l,m) \in \mathcal{H} \mid \text{if } k \text{ and } l \text{ are lists then } m \text{ is a list and } k * l = m \,\}$$

($\mathcal{H}$ is the Herbrand base; we assume a fixed infinite set of function symbols). Obviously, (1) holds iff all the ground instances of $app(k,l,m)$ are in *spec*.

Notice that we do not need to refer to the notion of a query in the specification. Assume that $app(k,l,m) = app(k',l',m')\theta$ is a computed answer for a

[1] Actually, the requirement on k is unnecessary. Our intention however is to follow the corresponding example from [Apt97]. A full specification of APPEND may be: if l is a list or m is a list then k,l,m are lists and $k * l = m$.

query $app(k', l', m')$. If k', l' are lists then (k, l are lists and) the specification implies that m is a list and $k * l = m$.

Such specifications, referring to program answers, will be called *declarative*. A declarative specification can be an interpretation (possibly a non Herbrand one) or a theory.[2] In this paper we will use specifications of the first kind, but most of our results also apply to specifications of the second kind.

Definition 3.1. A definite program is **correct** w.r.t. a declarative specification *spec* iff $spec \models Q$ for any answer Q of the program.

Notice that if a program is correct w.r.t. a Herbrand interpretation *spec* then its least Herbrand model is a subset of *spec*.

To prove correctness (of a logic program w.r.t. a declarative specification) we use an obvious approach, discussed among others by Clark [Cla79], Hogger [Hog81, p. 378–9] and Deransart [Der93, Section 3].[3] We will call it the *natural* proof method. It consists of showing that $spec \models C$ for each clause C of the considered program. The soundness of the natural method follows from the following simple property:

Proposition 3.2 (Correctness, definite programs). Let P be a program and *spec* be an interpretation. If

$$spec \models P$$

then P is correct w.r.t. specification *spec*.

Proof. By soundness of SLD-resolution, $P \models Q$ for any answer Q. Now $spec \models P$ and $P \models Q$ imply $spec \models Q$. (This also holds for *spec* being a theory.) □

The method is also complete [Der93] in the following sense. If a program P is correct w.r.t. a declarative specification *spec* then there exists a stronger specification $spec' \subseteq spec$ such that $spec' \models P$, and thus the method is applicable to $spec'$. (The least model of P over the given domain can be taken as $spec'$.)

In our example the correctness proof of APPEND is simple. We present here its less trivial part with details. Consider the second clause. To show that

$$spec \models app([H|K], L, [H|M]) \leftarrow app(K, L, M)$$

take ground terms h, k, l, m[4] such that $spec \models app(k, l, m)$ (in other words $app(k, l, m) \in spec$). We have to show that $spec \models app([h|k], l, [h|m])$. Assume that $[h|k]$ and l are lists (hence k is a list). Then m is a list and $k * l = m$, as $spec \models app(k, l, m)$. Thus $[h|m]$ is a list and $[h|k] * l = [h|m]$, hence $app([h|k], l, [h|m]) \in spec$. This concludes the proof.

[2] A specification equivalent to our example specification *spec* may consist of an axiom $app(k, l, m) \leftrightarrow (list(k), list(l) \rightarrow list(m), k{*}l{=}m)$ together with axioms describing predicates $=$, *list* and function $*$, and an induction schema for lists.

[3] where it is called "inductive proof method".

[4] and valuation $\{ H/h, K/k, L/l, M/m \}$

Programs dealing with accumulators or difference lists are sometimes considered difficult to reason about. The following example shows that this is not the case when the natural method is used. Consider the standard REVERSE program:

$$\begin{array}{l} reverse(X,\, Y) \leftarrow rev(X,\, Y,\, [\,]) \\ rev([\,], X, X) \leftarrow \\ rev([H|L], X, Y) \leftarrow rev(L, X, [H|Y]) \end{array}$$

Formally, a difference list representing a list $[t_1, \ldots, t_n]$ is any pair $([t_1, \ldots, t_n|t],\, t)$ of terms. The declarative reading of the program is simple: the first argument of *rev* is a list, its reverse is represented as a difference list of the second and the third argument. This can be expressed by a formal specification

$$\begin{array}{rl} spec = & \{\, reverse([t_1, \ldots, t_n], [t_n, \ldots, t_1]) \mid n \geq 0,\ t_1, \ldots, t_n \in \mathcal{T} \,\} \\ & \cup \{\, rev([t_1, \ldots, t_n], [t_n, \ldots, t_1|t], t) \mid n \geq 0,\ t_1, \ldots, t_n, t \in \mathcal{T} \,\} \end{array}$$

where $\mathcal{T}$ is the set of ground terms. The reader can check that $spec \models$ REVERSE, thus the program is correct w.r.t. *spec*.

Notice that the natural method refers only to the declarative semantics of programs. A specification is an interpretation (alternatively a theory). Correctness is expressed as truth (of the program's answers) in the interpretation. Program clauses are treated as logic formulae, their truth in the interpretation is to be shown. We abstract from any operational semantics, in particular from the form of queries appearing during computation. The reasoning is obviously independent from the selection rule. Still we can use declarative specifications to reason about queries and corresponding answers, using the fact that an answer is an instance of the query.

3.2 Call-Success Specifications and the Operational Approach

Some authors propose another approach to proving correctness of definite programs [BC89], [Apt97, Chapter 8], [PR99].[5] We will call it an *operational* proof method. In this section we show that it is not stronger than the natural method from the previous section (as far as properties of program answers are concerned).

A specification in the operational approach consists of two parts. The *precondition* specifies the procedure calls that appear during the computation (more precisely, the atoms that are selected in the LD-resolution). The *postcondition* specifies the procedure successes (the computed instances of procedure calls). We will call such specifications *call-success specifications*. Formally, pre- and postconditions are sets of atoms, closed under substitutions.

A program is correct if every procedure call and every success satisfy the pre- or postcondition respectively, provided that the initial query satisfies a certain condition. Notice that this is not a declarative property. It considers not only computed answers, but whole computations (LD-trees), and it depends on the selection rule used.

[5] Whenever these approaches differ, we follow that of [Apt97].

The operational proof method was proposed by Bossi and Cocco [BC89] and is an instance of the method of Drabent and Małuszyński [DM88]. It is based on the following verification condition. For each clause C of the program, show that for each (possibly non ground) instance $H \leftarrow B_1, \ldots, B_n$ $(n \geq 0)$ of C:

if $H \in pre,\ B_1, \ldots, B_k \in post$ then $B_{k+1} \in pre$ (for $k = 0, \ldots, n-1$),
if $H \in pre,\ B_1, \ldots, B_n \in post$ then $H \in post$.

The condition on the initial query is that, for any instance $B_1, \ldots, B_n$ $(n > 0)$ of the query, if $B_1, \ldots, B_k \in post$ then $B_{k+1} \in pre$ (for $k = 0, \ldots, n-1$).

Let us come back to our APPEND example. We refer here to its treatment in [Apt97, p. 214]. The precondition and postcondition are, respectively,

$$pre = \{\, app(k,l,m) \mid k \text{ and } l \text{ are lists} \,\},$$
$$post = \{\, app(k,l,m) \mid k,l,m \text{ are lists and } k * l = m \,\}.$$

(Here k, l, m are terms, possibly non ground.) The verification conditions to be proved consist of one implication for the first clause of APPEND and two implications for the second one. The details of the proof can be found in [Apt97].

Notice that the operational method requires proving one implication per atom occurring in the program or in the initial query. In contrast, the natural method from the previous section requires proving one implication per program clause. The natural method is independent of the operational semantics and of the ordering of the body atoms of program clauses.

The natural method is an instance of the operational one. (For a given declarative specification *spec* from the natural method, take the set of all atoms as the precondition and $post = \{A \mid spec \models A\}$ as the postcondition). We show however that both methods are equivalent, as far as properties of answers are of interest. Consider a call-success specification $\langle pre, post \rangle$. A corresponding declarative specification could be seen, speaking informally, as implication $pre \rightarrow post$.

Definition 3.3. Let *pre* and *post* be sets of atoms closed under substitution. The *declarative specification corresponding* to the call-success specification $\langle pre, post \rangle$ is the Herbrand interpretation

$$pre{\rightarrow}post \ := \ \{\, A \in \mathcal{H} \mid \text{if } A \in pre \text{ then } A \in post \,\}.$$

In other words, $pre{\rightarrow}post = (\mathcal{H} \setminus pre) \cup (\mathcal{H} \cap post)$. If P is correct w.r.t. $pre{\rightarrow}post$ and $A\theta$ is an answer to a query $A \in pre$ then $A\theta \in post$.

The following proposition compares corresponding declarative and call-success specifications. Proposition 3.5 compares both proof methods, showing that the natural method is not weaker.

Proposition 3.4. If a program P is correct w.r.t. the call-success specification $\langle pre, post \rangle$ then P is correct w.r.t. declarative specification $pre{\rightarrow}post$.

Proposition 3.5. If P and $\langle pre, post \rangle$ satisfy the verification condition of the operational method then $pre{\rightarrow}post \models P$.

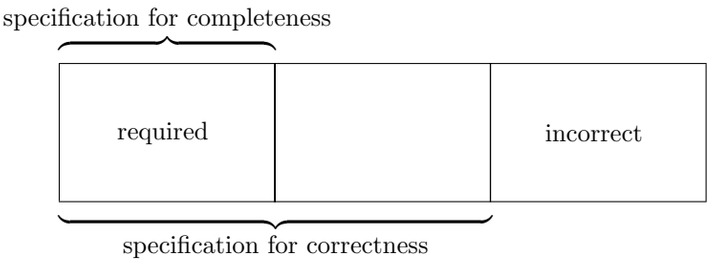
In other words, assume that by the operational method it can be shown that a program P is correct w.r.t. $\langle pre, post\rangle$. Then it can be shown that P is correct w.r.t. $pre{\rightarrow}post$, using the natural method.

For proofs of both propositions see [Dra99] (and [CD88] for the second one). The first property is mentioned also in [dB+97,PR99]. The reverse of the two propositions does not hold. For a counterexample consider reordering the body atoms in a correct program. For further comparisons see [Dra99].

Notice the difference in the treatment of "ill-typed" atoms (like $app([\,], 1, 1)$ for APPEND) by the specifications in both methods. Declarative specifications *include* all such atoms, call-success specifications *exclude* them.

3.3 Completeness of Definite Programs

Let us begin from an observation that for a given program a specification for completeness is in general different from that for correctness. For the purposes of correctness we describe a superset of the set of answers of a program. For the purposes of completeness we describe its subset, as a program satisfying a completeness requirement may compute something more than required. Often when a specification for correctness is of the form $pre{\rightarrow}post$ then a specification for completeness is $post$.



For instance, it makes no sense to require that APPEND program were complete w.r.t. the specification *spec* from the beginning of Section 3.1. This would mean computing all the "ill-typed" answers, like $app(a, b, c)$. Our specification for completeness of APPEND is the Herbrand interpretation

$$specC = \{app(k, l, m) \in \mathcal{H} \mid k, l, m \text{ are lists, } k * l = m\}.$$

Notice that it properly expresses our intentions: APPEND should compute all the cases of list concatenation. The difference $spec - specC$ contains only "ill-typed" atoms, with the first or second argument not being a list. We are not interested whether they are answers of APPEND.

As previously, we consider specifications which are (possibly non Herbrand) interpretations.

Definition 3.6. A program P is **complete** for a query Q w.r.t. a specification $specC$ if $specC \models Q\theta$ implies that $Q\theta$ is an answer for the program, for any instance $Q\theta$ of Q.

Notice that an answer $Q\theta$ is an instance of some computed answer for Q.

Below we refer to theory ONLY-IF(P) [Apt90] that is usually used while defining Clark completion comp(P) of a program P. Informally, ONLY-IF(P) is P with implications reversed. For each predicate symbol p, if the clauses of P beginning with p are $p(\vec{t_1})\leftarrow B_1, \ldots, p(\vec{t_k})\leftarrow B_k$ then ONLY-IF(P) contains

$$p(\vec{x}) \;\rightarrow\; \bigvee_{i=1}^{k} \exists_{-\vec{x}}\; \vec{x} = \vec{t_i} \;\wedge\; B_i,$$

where $\vec{x}$ are distinct new variables and the quantification is over the variables occurring in the clauses. For $k = 0$ the implication is equivalent to $\neg p(\vec{x})$. In our example, ONLY-IF(APPEND) is (equivalent to)

$$app(x, y, z) \;\rightarrow\; x = [\,], y = z \;\vee\; \exists\, h, k, m\, (x = [h|k], z = [h|m], app(k, y, m)).$$

We will also need a specification for equality: $spec_{=} = \{t{=}t \mid t \text{ is a ground term}\}$.

The following property can be used to prove completeness of a program. It is a special case of Theorem 4.6 for normal programs.

Proposition 3.7 (Completeness, definite programs). Let P be a program and Q a query. Assume that

(i) $specC \cup spec_{=} \models$ ONLY-IF(P) and
(ii) P terminates for Q, i.e. there exists a finite SLD-tree for Q and P.

Then P is complete for Q.

For instance, consider program APPEND and the specification $specC$ given above. It is easy to show that $specC \cup spec_{=} \models$ ONLY-IF(APPEND). Consider $Q = app(k, l, m)$, where m is a list. One can show, using any standard method, that Q terminates under Prolog selection rule. Thus APPEND is complete for Q.

Now assume that k, l are distinct variables. Taking k', l' being lists such that $k' * l' = m$ we have $specC \models app(k', l', m)$ and from the proposition we get $P \models app(k', l', m)$. So by completeness of SLD-resolution, $app(k', l', m)$ (or a more general atom) is a computed answer for Q. Summarizing, Q succeeds and produces all the required divisions of m into two lists.

We believe that the proposition above is a formalization of a common way of informal reasoning about completeness, which consists of checking that any tuple of argument values to be defined by the predicate is "covered" by some of its clauses.

The method proposed here proves program completeness for queries that terminate. This should not be seen as disadvantage, since termination of a program has to be established anyway. Notice that the proposition without the termination requirement does not hold. Program $\{\, app(X, Y, Z) \leftarrow app(X, Y, Z)\,\}$ is a counterexample.

4 Reasoning about Normal Programs

In this chapter, unless stated otherwise, the considered programs (queries) are normal programs (queries). As the operational semantics we consider SLDNF-resolution [AD94]. We usually skip "finitely" in phrases like "finitely fails".

In order to introduce specifications for normal programs let us first consider definite programs with queries which may contain negative literals. Assume that we have a definite program P complete w.r.t. a Herbrand specification $specC$ and correct w.r.t. a $specS$ (C as completeness, S as soundness). If an atomic query A fails then $specC \models \neg A$. So for P and atomic queries, finite failure is correct w.r.t. the specification for completeness. Now consider a query $Q = p(\vec{t}), \neg q(\vec{u})$. If it succeeds with an answer $Q\theta$ then $spec' \models Q\theta$ for an interpretation $spec'$ that interprets p as $specS$ and q as $specC$. If Q fails then $spec'' \models \neg Q$ for an interpretation $spec''$ that interprets p as $specC$ and q as $specS$. In order to deal with this phenomenon, we will use certain renamings of predicate symbols.

Definition 4.1. Let $\mathcal{L}$ be a first order language. Let $\mathcal{Q}$ be a formula or a set of formulae (e.g. a query or a program) of $\mathcal{L}$. Let us extend $\mathcal{L}$ by adding, for any predicate symbol p, a new predicate symbol p'.

$\mathcal{Q}'$ is $\mathcal{Q}$ with p replaced by p' in every negative literal of $\mathcal{Q}$ (for any predicate symbol p, except for $=$). Similarly, $\mathcal{Q}''$ is $\mathcal{Q}$ with p replaced by p' in every positive literal.

If I is an interpretation for $\mathcal{L}$ then I' is the interpretation obtained from I by replacing each p by p'.

For normal programs, a specification for correctness should describe two possibly overlapping sets of ground atoms, those allowed to succeed and those allowed to fail. Similarly, a specification for completeness should describe two disjoint sets, of the ground atoms required to fail and of those required to succeed. It is natural to allow to succeed any atom not required to fail, and allow to fail any atom not required to succeed. Hence the two sets needed to specify completeness can be the complements of the two sets used to specify correctness.

Definition 4.2. A **specification** for a normal program is a pair *(specS, specC)*, where $specC$ and $specS$ are interpretations such that $specC \subseteq specS$.

For an informal explanation, assume that a program P is correct w.r.t. a Herbrand specification $spec = (specS, specC)$. If a ground atomic query A succeeds then $A \in specS$. If A fails then $A \notin specC$. If P is complete w.r.t. $spec$ then any $A \in specC$ succeeds and any $A \notin specS$ fails.

4.1 Correctness of Normal Programs

Definition 4.3. We say that a program P is **correct** with respect to a specification $spec = (specS, specC)$ iff for any query Q

(i) every computed answer Q satisfies: $specS \cup specC' \models Q'$,
(ii) if Q finitely fails then $specS \cup specC' \models \neg Q''$.

In particular, if P is correct with respect to $spec = (specS, specC)$, then every computed answer Q satisfies the following. For each positive literal A in Q, $specS \models A$, and for each negative literal $\neg A$ in Q, $specC \models \neg A$.

Theorem 4.4 (Correctness, normal programs). Let P be a program, Q a query and $spec = (specS, specC)$ a specification, such that

(a) $specS \cup specC' \models P'$
(b) $specS \cup specC' \cup spec_{=} \models$ ONLY-IF(P'')

then

(i) if $comp(P) \models_3 Q$, then $specS \cup specC' \models Q'$
(ii) if $comp(P) \models_3 \neg Q$, then $specS \cup specC' \models \neg Q''$
and P is correct w.r.t. *spec*.

Proof (outline). The proof [DM01] is based on (1) similarity between $P' \cup$ ONLY-IF$(P'') \cup CET$ and Stärk's partial completion $pcomp(P)$ [Stä96], (2) equivalence of 3-valued consequences of $comp(P)$ and classical consequences of $pcomp(P)$ (modulo a certain syntactic transformation) [Stä96], and (3) soundness of SLDNF-resolution w.r.t. 3-valued completion semantics [Doe94]. □

The Theorem 4.4 is valid for any operational semantics, which is sound w.r.t. 3-valued completion semantics. This includes constructive negation (cf. [Dra95] and the references therein) and extensions of SLDNF-resolution allowing selecting non ground negative literals under certain conditions [Llo87,Stä96].

4.2 Completeness of Normal Programs

To discuss completeness we need to refer to the notion of SLDNF-tree. We will follow the definition of Apt and Doets [AD94]. We outline it below, for the details the reader is referred to [AD94] or [Doe94].

An SLDNF-tree for query Q and program P is a set of trees, with one of them distinguished as the main tree. The nodes of the trees are queries and the trees are, roughly speaking, SLDNF-trees of [Llo87]. Q is the root of the main tree. Any node with a non ground negative literal selected is a leaf of a tree, such a node is marked *floundered*. Whenever a ground negative literal $\neg A$ is selected in a node N then there exists a subsidiary tree with the root A. The whole SLDNF-tree may be viewed as a tree of trees, in which the tree with the node N is the parent of the subsidiary tree with the root A.

The leaves of each tree can be marked *failed* or *success*, with the expected meaning. So if a leaf N is neither marked *failed* nor *success* then a negative literal $\neg A$ is selected in N, moreover A is non ground or the subsidiary tree for A neither succeeds nor finitely fails. A tree succeeds if it has a *success* leaf. A tree finitely fails if it is finite and all its leaves are marked *failed*.

The SLDNF-tree succeeds (finitely fails) if the main tree does. To each success leaf of the main tree there corresponds a computed answer substitution θ for Q (and a computed answer $Q\theta$), defined as expected.

Definition 4.5. We say that a program P is **complete** for a query Q w.r.t. a specification $spec = (specS, specC)$ iff

(i) $specS \cup specC' \models \neg Q'$ implies that some SLDNF-tree for Q finitely fails,
(ii) $specS \cup specC' \models Q''\sigma$ implies that some SLDNF-tree for Q succeeds with an answer $Q\theta$ more general than $Q\sigma$.

Theorem 4.6 (Completeness, normal programs). Assume that the set of function symbols is infinite. Let P be a program, Q a query and $spec = (specS, specC)$ a specification such that

1. (a) $specS \cup specC' \models P'$, (b) $specS \cup specC' \cup spec_{=} \models \text{ONLY-IF}(P'')$,
2. there exists an SLDNF-tree for Q such that its main tree is finite and all the leaves of the main tree are marked *failed* or *success*.

Then P is complete for Q w.r.t. $spec = (specS, specC)$.

Conditions (a),(b) are the same as in Theorem 4.4. Condition 2 implies that each $\neg A$ selected in the main tree is ground and the subsidiary tree for A succeeds or fails. Notice that the SLDNF-tree may be infinite or contain floundering nodes. However the "important part" of it is finite and without floundering and can be computed under some search strategy in a finite number of steps. (When a success is obtained in a subsidiary tree, traversing this tree can be abandoned.)

Proof (outline). By Theorem 4.4 program P is correct w.r.t. *spec*. The existence of SLDNF-trees satisfying the thesis is proved by contradiction [DM01]. □

4.3 Examples

In this section we illustrate our method of proving correctness and completeness of normal programs by two examples. The first one is a small program defining the subset relation. We have chosen this example (from [Stä96]) because it is short and it has nested negations. Completeness of normal programs can however be better illustrated by the second example defining the subset relation with an additional requirement that a subset must be a list without repetitions.

Example 4.1. Let P be the following program:

$$
\begin{array}{l}
subset(L, M) \leftarrow \neg notsubset(L, M) \\
notsubset(L, M) \leftarrow member(X, L), \neg member(X, M) \\
member(X, [X|L]) \leftarrow \\
member(X, [Y|L]) \leftarrow member(X, L)
\end{array}
$$

Consider Herbrand specification $spec = (specS,\ specC)$, where

$$
\begin{array}{l}
specS = sS_m \cup sS_n \cup sS_s, \qquad specC = sC_m \cup sC_n \cup sC_s \\
sS_m = \{member(x, l) \mid l \text{ is a list} \rightarrow x \in l\} \\
sC_m = \{member(x, l) \mid l \text{ is a list} \wedge x \in l\} \\
sS_n = \{notsubset(l, m) \mid l \text{ and } m \text{ are lists} \rightarrow l \not\subseteq m\} \\
sC_n = \{notsubset(l, m) \mid l \text{ and } m \text{ are lists} \wedge l \not\subseteq m\} \\
sS_s = \{subset(l, m) \mid l \text{ and } m \text{ are lists} \rightarrow l \subseteq m\} \\
sC_s = \{subset(l, m)| \mid l \text{ and } m \text{ are lists} \wedge l \subseteq m\}
\end{array}
$$

We would like to prove that our program is correct and complete with respect to the above specification *spec*. We show that conditions (a) and (b) of Theorem 4.4 are satisfied. This implies that whenever $subset(l, m)$ is a computed answer of P then $sS_s \models subset(l, m)$, and if l, m are lists then $l \subseteq m$. Also, whenever a query $subset(l, m)$ fails then $sC_s \models \neg subset(l, m)$. (Notice that l, m are ground, as otherwise the query flounders.) Hence l or m is not a list or $l \not\subseteq m$. From Theorem 4.6 it follows that, for ground lists l, m, if $l \subseteq m$ then $subset(l, m)$ succeeds, otherwise it fails, since under Prolog selection rule any ground query $subset(l, m)$ terminates without floundering.

Let $spec' = specS \cup specC'$. In order to prove condition (a), for each clause C of P one has to show that $spec' \models C'$. In order to prove condition (b) one has to show, for each predicate p of P, that the implication of ONLY-IF(P'') beginning from p is true in the interpretation $spec' \cup spec_{=}$.

Let us consider the second clause of program P. For condition (a) we have to prove that:

$$spec' \models notsubset(L, M) \leftarrow member(X, L) \wedge \neg member'(X, M).$$

Let l, m, x be any elements of the universe such that $spec' \models member(x, l) \wedge \neg member'(x, m)$. That means that $member(x, l) \in sS_m$ and $member(x, m) \notin sC_m$. We would like to prove that $notsubset(l, m) \in sS_n$. So assume that l and m are lists. From $member(x, l) \in sS_m$ we obtain that $x \in l$, and from $member(x, m) \notin sC_m$ — $x \notin m$. Hence $l \not\subseteq m$.

For condition (b) and predicate *notsubset* we have to show that

$$spec' \models notsubset'(L, M) \rightarrow \exists X(member'(X, L) \wedge \neg member(X, M))$$

Let l, m be any elements of the universe such that $spec' \models notsubset'(l, m)$. So l and m are lists and $l \not\subseteq m$. So there exists an element, say a, such that $a \in l$ and $a \notin m$. Thus $member(a, l) \in sC_m$ and $member(a, m) \notin sS_m$. Hence $spec' \models member'(a, l) \wedge \neg member(a, m)$ and $spec' \models \exists X(member'(X, L) \wedge \neg member(X, M))$, so the implication above is true in $spec'$.

Let C denote the first clause of P. Notice that $subset(L, M) \leftrightarrow \neg notsubset(L, M)$ is true both in $sS_s \cup sC_n$ and in $sC_s \cup sS_n$. After replacing *notsubset* by *notsubset'*, this implies $sS_s \cup sC'_n \models C'$, and hence (a) for the first clause. After replacing *subset* by *subset'*, this implies $sS_n \cup sC'_s \models subset'(L, M) \rightarrow \neg notsubset(L, M)$, hence (b) for predicate *subset*.

The proof for predicate *member* boils down to a correctness and completeness proof of a definite program. □

Example 4.2. Let P be the following program:

$subs([\,], L) \leftarrow$
$subs([H|T], LH) \leftarrow select(H, LH, L), subs(T, L), \neg member(H, T)$
$select(H, [H|L], L) \leftarrow$
$select(H, [X|L], [X|LH]) \leftarrow select(H, L, LH)$

The definition and specification of *member* are the same as in Example 4.1. A Herbrand specification for P is $spec = (specS,\ specC)$, where

$$specS = sS_m \cup sS_{sel} \cup sS_{subs}, \quad specC = sC_m \cup sC_{sel} \cup sC_{subs}$$

$$\begin{aligned}
sS_{sel} &= \{select(e,l,m) \mid l \text{ is a list } \rightarrow e \in l \wedge m \text{ is a list} \wedge l \approx [e|m]\} \\
sC_{sel} &= \{select(e,l,m) \mid l \text{ and } m \text{ are lists such that} \\
&\quad\ l = [e_1,\ldots,e_i,e,e_{i+1},\ldots e_k],\ m = [e_1,\ldots,e_i,e_{i+1},\ldots e_k], 0 \le i \le k\} \\
sS_{subs} &= \{subs(l,m) \mid m \text{ is a list} \rightarrow listd(l) \wedge l \subseteq m\} \\
sC_{subs} &= \{subs(l,m) \mid m \text{ is a list} \wedge listd(l) \wedge l \subseteq m\}
\end{aligned}$$

Here $l \approx m$ means that lists l and m contain the same elements and $listd(l)$ means that l is a list with distinct elements.

Let $spec' = specS \cup specC'$. To prove condition (a) for the second clause of predicate $subs/2$, assume that

$$spec' \models select(h,lh,l), subs(t,l), \neg member'(h,t). \qquad (A)$$

We show that $subs([h|t],lh) \in sS_{subs}$. So let lh be a list. From (A) it follows that:
(1) $select(h,lh,l) \in sS_{sel}$ hence $h \in lh$ and l is a list such that $lh \approx [h|l]$ (since lh is a list);
(2) $subs(t,l) \in sS_{subs}$ hence $listd(t)$ and $t \subseteq l$, thus $[h|t] \subseteq lh$, by (1);
(3) $member(h,t) \notin sC_m$ hence $h \notin t$ (since t is a list), thus $listd([h|t])$, by (2).
We obtained $[h|t] \subseteq lh$ and $listd([h|t])$, this completes the proof of condition (a) for the most complex clause of P.

The remaining part of the proof of condition (a) and proof of condition (b) is skipped here. It follows that P is correct w.r.t. *spec*.

Consider a query $Q = subs(L,M)$, where L is a variable and M a ground list. Once it is shown that for such queries P terminates without floundering (under some selection rule and search strategy), it follows that P is complete for such queries. This means that for a given list all its subsets (and all their permutations) will be computed.

Assume that we do not have a termination proof and request all answers to a query Q from an interpreter with run-time checks for floundering. Then if the execution terminates, we know that all the answers for Q required by the specification have been produced. This happens in the case of P and Prolog. □

Every (logic) programmer should have, at least in her mind, intended interpretations of all used relations. Specification *spec* is a formalization of such interpretations. We believe that the methods advocated in this paper are a formalization of informal reasoning performed by a competent programmer to convince herself about correctness of a program.

5 Related Work

In this section we present a brief overview of related work. A more detailed comparison with the work on definite programs can be found in [Dra99].

Due to our approach to specifications, we do not need any explicit notion of precondition, type information, or domain of a procedure. Such notions are used in most other approaches [BC89,Apt97,PR99,Dev90] to deal with "ill-typed" atoms, for which the behaviour of the program is of no interest. Our approach is related to a 3-valued approach to definite programs [Nai00]. We however avoid any use of 3-valued logic. [Nai96] advocated declarative view for a class of program properties.

A completeness theorem for definite programs is given in [DM93]. It is stronger than ours, as its premises do not refer to termination. It however requires checking some other conditions, that checking is similar to proving termination. So whenever termination has to be shown anyway, our approach is simpler.

Comparison with the operational method [BC89,Apt97,PR99] for correctness of definite programs has been given in Section 3.2. We showed that the natural method of Section 3.1 is not weaker. Some inconveniences of the operational approach are discussed in [Dra99]. The operational method can be generalized to correctness of normal programs [Apt93,PR99]. Here the comparison is similar. The operational approach refers to LDNF-resolution, while the declarative method of Section 4.1 is independent from the operational semantics. So it covers arbitrary selection rules (e.g. delays used to avoid floundering) and various generalizations of SLDNF-resolution. We expect that reasoning in Section 3.2 can be generalized to this case thus showing that the operational method is not stronger (as far as properties of program answers are concerned).

Our approach to normal programs considers their 3-valued semantics, which is more precise than 2-valued, used in [Dev90,Apt93,PR99]. Further comparisons are needed with [Dev90] and with completeness realated reasoning in [PR99].

An important approach to proving properties of normal programs is proposed by Stärk [Stä97]. It deals with normal programs, executed under Prolog selection rule. A tool to mechanically verify the proofs exists. Success, failure and termination are described by an inductive theory. The theory can be seen as a further development of the notion of program completion. The program's properties of interest are expressed as formulae and one has to prove that they are consequences of the theory. This is opposite to our approach where properties are expressed as specifications and one has to prove that certain theory obtained from the program is a consequence of the specification.

6 Conclusions

This paper advocates declarative reasoning about logic programs. We show how to prove correctness and completeness of definite and normal logic programs in a declarative way, independently from the operational semantics. This makes it possible to separate reasoning about "logic" from reasoning about "control". The method for proving correctness of definite programs is not new, however its usefulness has not been appreciated. The methods for completeness and for correctness of normal programs are a contribution of this work.

For definite program we refer to two specifications; one for correctness and one for completeness. This makes it possible to specify the program semantics approximately, thus simplifying the specifications and the proofs. In this paper specifications are interpretations, but the approach seems applicable to specifications being theories.

The semantics of programs with negation is 3-valued. We do not however explicitly refer to 3-valued logic. Instead, a pair of specifications plays a role of a 3-valued specification. In this case it turns out that the same pair of specifications can conveniently be used both for correctness and completeness.

We believe that the presented proof methods are simple and natural. We claim that they are a formalization of a style of thinking in which a competent logic programmer reasons (or should reason) about her programs. We believe that these methods, possibly treated informally, are a valuable tool for actual everyday reasoning about real programs. We believe that they should be used in teaching logic programming.

References

AD94. K. R. Apt and K. Doets. A new definition of SLDNF-resolution. *Journal of Logic Programming*, 18(2):177–190, 1994.

Apt90. K. Apt. Logic programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B*, chapter 10, pages 493–574. Elsevier Science Publishers B.V., 1990.

Apt93. K. R. Apt. Declarative programming in Prolog. In D. Miller, editor, *Logic Programming, Proc. ILPS'93*, pages 12–35. The MIT Press, 1993.

Apt97. K. R. Apt. *From Logic Programming to Prolog.* Prentice-Hall, 1997.

BC89. A. Bossi and N. Cocco. Verifying correctness of logic programs. In *Proceedings of TAPSOFT '89, vol. 2*, pages 96–110. Springer-Verlag, 1989. LNCS 352.

CD88. B. Courcelle and P. Deransart. Proofs of partial correctness for attribute grammars with application to recursive procedures and logic programming. *Information and Computation*, 78(1):1–55, 1988.

Cla79. K. L. Clark. Predicate logic as computational formalism. Technical Report Technical Report 79/59, Imperial College, London, December 1979.

dB+97. F.S. de Boer et al. Proving concurrent constraint programs correct. *ACM TOPLAS*, 19(5):685–725, 1997.

Der93. P. Deransart. Proof methods of declarative properties of definite programs. *Theoretical Computer Science*, 118:99–166, 1993.

Dev90. Y. Deville. *Logic Programming: Systematic Program Development.* Addison-Wesley, 1990.

DM88. W. Drabent and J. Małuszyński. Inductive assertion method for logic programs. *Theoretical Computer Science*, 59:133–155, 1988.

DM93. P. Deransart and J. Małuszyński. *A Grammatical View of Logic Programming.* The MIT Press, 1993.

DM01. W. Drabent and M. Miłkowska. Proving correctness and completeness of normal programs — a declarative approach. Preliminary, unabridged version, http://www.ipipan.waw.pl/~drabent/proving.iclp01-prel.ps.gz, 2001.

Doe94. K. Doets. *From Logic to Logic Programming.* The MIT Press, Cambridge, MA, 1994.

Dra95. W. Drabent. What is failure? An approach to constructive negation. *Acta Informatica*, 32(1):27–59, 1995.

Dra99. W. Drabent. It is declarative: On reasoning about logic programs. Poster in Proc. 1999 International Conference on Logic Programming (unpublished report, http://www.ipipan.waw.pl/~drabent/itsdeclarative3.ps.gz), May 1999.

Hog81. C. J. Hogger. Derivation of logic programs. *J. of ACM*, 28(2):372–392, 1981.

Kun87. K. Kunen. Negation in logic programming. *Journal of Logic Programming*, 4(4):289–308, 1987.

Llo87. J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 2nd edition, 1987.

Nai96. L. Naish. A declarative view of modes. In *Proceedings of JICSLP'96*, pages 185–199. MIT Press, 1996.

Nai00. Lee Naish. A three-valued semantics for Horn clause programs. *Australian Computer Science Communications*, 22(1):174–180, January 2000.

PR99. D. Pedreschi and S. Ruggieri. Verification of logic programs. *Journal of Logic Programming*, 39(1-3):125–176, 1999.

Stä96. R. F. Stärk. From logic programs to inductive definitions. In W. A. Hodges et al., editors, *Logic: From Foundations to Applications, European Logic Colloqium'93*, pages 453–481. Clarendon Press, Oxford, 1996.

Stä97. R. F. Stärk. Formal verification of logic programs: Foundations and implementation. In *Logical Foundations of Computer Science LFCS '97 — Logic at Yaroslavl*, pages 354–368. Springer-Verlag, 1997. LNCS 1234.

An Order-Sorted Resolution with Implicitly Negative Sorts

Ken Kaneiwa[1] and Satoshi Tojo[2]

[1] National Institute of Informatics
2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo 101-8430, Japan
kaneiwa@nii.ac.jp
[2] Japan Advanced Institute of Science and Technology
1-1 Asahidai, Tatsunokuchi, Ishikawa 923-1292, Japan
tojo@jaist.ac.jp

Abstract. We usually use natural language vocabulary for sort names in order-sorted logics, and some sort names may contradict other sort names in the sort-hierarchy. These implicit negations, called lexical negations in linguistics, are not explicitly prefixed by the negation connective. In this paper, we propose the notions of structured sorts, sort relations, and the contradiction in the sort-hierarchy. These notions specify the properties of these implicit negations and the classical negation, and thus, we can declare the exclusivity and the totality between two sorts, one of which is affirmative while the other is negative. We regard the negative affix as a strong negation operator, and the negative lexicon as an antonymous sort that is exclusive to its counterpart in the hierarchy. In order to infer from these negations, we integrate a structured sort constraint system into a clausal inference system.

1 Introduction

Order-sorted logics, or many-sorted logics, have been well discussed as tools to represent hierarchical knowledge in the field of artificial intelligence [18,4,3,7,14], [16,19]. Recently, description logics [15,1,2,10] as outlined in [6] have been studied as a theoretical approach to terminological knowledge representation, which represent structured concepts by more primitive concepts, as are similar to those in a sort-hierarchy.

However, a sort-hierarchy may contain sorts with implicitly negative meanings. These negations are called *lexical negations* in linguistics and are distinct from the negative particle *not*. Since every sort name is a mere string or a symbol, these implicitly negative sorts are not interpreted to represent their original meanings. Nevertheless, a knowledge representation system should take account of the fact that the lexical negation '*unhappy*' is opposite in meaning to the positive expression '*happy*', or '*winner*' contradicts '*loser*', in a sort-hierarchy.

In order to realize this, we have to analyze the properties of lexical negations in natural language and then consider dealing with these negations in a sort-hierarchy. In [12], lexical negations (words that implicitly have negative meaning) are classified as follows:

P. Codognet (Ed.): ICLP 2001, LNCS 2237, pp. 300–314, 2001.

(i) Negative affix (in-,un-,non-):
incoherent, inactive, unfix, nonselfish, illogical, impolite, etc.
(ii) Lexicon with negative meaning:
doubt (believe not), deny (approve not), prohibit (permit not), forget (remember not), etc.

First, we introduce a hybrid knowledge representation system of Beierle [3] that distinguishes between taxonomical information (in the sort-hierarchy) and assertional information (in the assertional knowledge base), as an extension of an order-sorted logic. This system can deal with the taxonomical information in an assertional knowledge base in which a sort symbol can be expressed as a unary predicate (called a sort predicate) in clausal forms. Since a sort and a unary predicate have the same expressive power, we can regard a subsort declaration $s_1 \sqsubseteq s_2$ as the following logical implication form:

$$s_1(x) \rightarrow s_2(x)$$

where the unary predicates $s_1(x), s_2(x)$ corresponding to the sorts s_1, s_2 are sort predicates. Let C, C_1, C_2 be clauses, s, s_1, s_2 sorts (or sort predicates), θ a sorted substitution, and t, t_1, t_2 sorted terms. In order to use the information in a sort-hierarchy in a clausal knowledge base, or an assertional knowledge base, the following inference rules:

$$\frac{\neg s_1(t_1) \vee C_1 \quad s_2(t_2) \vee C_2}{\theta(C_1 \vee C_2)} \ (subsort\ resolution)$$

where $s_2 \sqsubseteq_S s_1$ and $\theta(t_1) = \theta(t_2)$, and

$$\frac{\neg s(t) \vee C}{C} \ (elimination)$$

where $Sort(t)$[1] $\sqsubseteq_S s$, are added to his resolution system. This hybrid knowledge representation system provides a useful way to deal with a sort-hierarchy in a clausal knowledge base.

Hereafter, we illustrate the deductions which we would like to realize in a sort-hierarchy with lexical negations. The first example concerns a negative affix: *unhappy*. A sort *unhappy* is not only a negative expression but also a subexpression of *emotional*. Hence, the sort *emotional* can be derived from *unhappy* (like *happy*), whereas it cannot be derived from the classical negation $\neg happy$. In addition, the sort *unhappy* is a stronger negative statement than the classical negation $\neg happy$, so that $\neg happy$ can be derived from *unhappy*, but *unhappy* cannot be derived from $\neg happy$. The fact $\neg emotional(bob)$, that is the person *bob* is not emotional, yields that '$\neg happy(bob) \wedge \neg unhappy(bob)$.' In contrast, no premise can derive '$\neg happy(bob) \wedge \neg\neg happy(bob)$.' This shows the sort *unhappy* has the meaning of *emotional*, but the classical negation $\neg happy$ does not have the meaning of *emotional*.

[1] For any sorted term t, the function $Sort(t)$ assigns its sort to term t.

The next example concerns lexicon with negative meaning: *loser*. Suppose a sort-hierarchy where both of *winner* and *loser* are subsumed by *player*. Needless to say, *loser* is different from the classical negation $\neg winner$ of *winner*, because *loser* means the negative event opposite to an event denoted by *win* but the classical negation $\neg winner$ denies the event denoted by *win*. Therefore, the supersort *player* can be derived from *loser* (or *winner*), but not from $\neg winner$. Furthermore, if the person *tom* is not a player, then the negations $\neg winner$ and $\neg loser$ can be derived. In contrary, if the person *tom* is a player, then *winner* or *loser* holds in the totality (i.e. *tom* must be a winner or a loser) of *winner* and *loser*. By the totality, if the person *tom* is a player but not a loser ($\neg loser$), then *tom* is a winner. If *tom* is neither a winner nor a loser ($\neg winner \wedge \neg loser$), then *tom* is not a player ($\neg player$)

We would like to derive these facts from implicitly negative sorts. However, it is hard to describe implicitly negative sorts in the sort-hierarchy, so that many knowledge bases would lose the property that implicit negations are exclusive to their antonyms and partial to their classical negation. In fact, Beierle's inference system for sort-hierarchy and order-sorted substitutions in clauses do not generate any reasoning mechanism for negative sorts. Description logics and feature logics [16] provide complex sort expressions but not any clausal reasoning mechanism with these expressions. Therefore, these inference systems with sort-hierarchy cannot immediately derive the above results from subsorts, supersorts and classical negation. In the following sections, we will propose a method to describe the properties of lexical negations implicitly included in a sort-hierarchy and develop an inference machinery.

This paper is organized as follows. In Section 2 presents an order-sorted logic that includes the complex sort expressions of implicit negations. We give an account of structured sorts, sort relations, and contradiction in a sort-hierarchy. Section 3 and Section 4 present the formalization of order-sorted logic with structured sorts, and systems of clausal resolution. In Section 5, we give our conclusions and discuss future work.

2 Implicitly Negative Sorts

In order to deal with implicitly negative sorts in a sort-hierarchy, we introduce structured sorts, sort relations, and contradiction in a sort-hierarchy into an order-sorted logic. These notions can be used to declare the properties of implicitly negative sorts in a sort-hierarchy.

2.1 Structured Sorts and Sort Relations

We consider the representation of sorts in a hierarchy whose names are declared as lexical negations (classified as negative affixes or lexicons with negative meaning). In this paper, we call a sort denoted by a word with negative affix *a negative sort* and a sort denoted by a lexicon with negative meaning *an opposite sort*. In general, we call these sorts *implicitly negative sorts*. To represent these negative

sorts, we introduce the notation of structured sorts and relations between sorts whereby a negative sort is defined by the structured sort with strong negation operator [17] and an opposite sort is defined by exclusivity. In particular, we denote an opposite sort as exclusive to its antonymous sort in a hierarchy, so that these two sorts exclude each other but neither sort is negative. In fact, we should not say that an opposite sort is negative, rather we should say that these two sorts are opposite in meaning.

Structured sorts are constructed from atomic sorts, the connectives $\sqcap, \sqcup$, and the negative operators; $\overline{happy}$ is a complement (classical negation) of *happy*, and $\sim happy$ is a negative sort (strong negation) of that.

We now give several relations between structured sorts in order to represent implicitly negative sorts embedded in a sort-hierarchy. '$\sqsubseteq_S$' denotes a subsort relation between structured sorts. With this relation, a set of sorts are partially ordered (i.e. reflexive, anti-symmetric, and transitive). '$=_S$' denotes an equivalence relation between structured sorts. Furthermore, we add an exclusivity relation '$\|$' and a totality relation '$|_{s_i}$' between structured sorts; if $s \parallel s'$ then s and s' are exclusive, and if $s \mid_{s_i} s'$ then s together with s' composes the whole of s_i.

Using these sort relations, we can define the following properties (totality, partiality, and exclusivity) to declare various negations (in particular, lexical negations), as in Table 1.

Table 1. Three negations

Negation type	Expression	Relationship		Property
(1) Complement (classical negation)	$\overline{happy}$	$\overline{happy} \mid_\top happy$ $\overline{happy} \parallel happy$	(in Axioms)	totality exclusivity
(2) Negative sort (strong negation)	$\sim happy$	$\sim happy \parallel happy$ $\sim happy \sqsubseteq_S \overline{happy}$	(in Axioms)	exclusivity partiality
(3) Opposite sort (antonym)	*sad*	*sad* $\parallel$ *happy*	(in Declarations)	exclusivity

2.2 A Contradiction in a Sort-Hierarchy

We present a contradiction in a sort-hierarchy containing the three negations (complement, negative sort, and opposite sort) that we have explained.

A deductive system with implicitly negative sorts has to determine a contradiction in a sort-hierarchy in order that it can provide a sound inference mechanism derived from the three negations and their relations to each other.

In classical logic, we can say that a set Δ of formulas is contradictory if a formula A and its classical negation $\neg A$ are simultaneously derivable from Δ. In this case, we can syntactically establish the contradiction, because $\neg A$ indicates the negation of A by the negative operator $\neg$. Given the opposite sorts s and s' (e.g. *winner* and *loser*), we should also say that Δ is contradictory if the two formulas $s(x), s'(x)$ denoted by the sort predicates s and s' are simultaneously derivable from Δ. This indicates that the sort symbols s and s' have a negative relation to each other in our language definition. [2]

Using an exclusivity relation between sorts, we give a definition of contradictions in a sort-hierarchy that supports deduction from the three negations. A set Δ of formulas is said to be contradictory if there exist sorts s, s' such that $s \parallel s'$ and $s(t)$ and $s'(t)$ are derivable from Δ. In section 3, we will redefine the notion of contradiction in a sort-hierarchy that enables our deduction system to ensure the consistency of a knowledge base.

3 An Order-Sorted Logic with Structured Sorts

On the specification we propose in Section 2, we define the syntax and semantics of an order-sorted logic with structured sorts.

3.1 Structured Sort Signature

Given a set $\mathcal{S}$ of sort symbols, every sort $s_i \in \mathcal{S}$ is called an atomic sort. We define the set of structured sorts composed by the atomic sorts, the connectives, and the negative operators as follows.

Definition 1 (Structured sorts). *Given a set $\mathcal{S}$ of atomic sorts, the set $\mathcal{S}^+$ of structured sorts is defined by:*

(1) If $s \in \mathcal{S}$, then $s \in \mathcal{S}^+$,
(2) If $s, s' \in \mathcal{S}^+$, then $(s \sqcap s')$, $(s \sqcup s')$, $(\overline{s})$, $(\sim s) \in \mathcal{S}^+$.

The structured sort $\overline{s}$ is called the classical negation of sort s and the structured sort $\sim s$ is called the strong negation of sort s. For convenience, we can denote $s \sqcap s'$, $s \sqcup s'$, $\overline{s}$ and $\sim s$ without parentheses when there is no confusion.

Example 1. Given the atomic sorts male, student, person and happy, we can give structured sorts as follows.

$$student \sqcap \overline{male},\ person \sqcup \sim happy.$$

The structured sort $student \sqcap \overline{male}$ means "students that are not male," and the structured sort $person \sqcup \sim happy$ means "individuals that are persons or unhappy."

[2] Gabbay and Hunter introduce the notation $\neg_\alpha \beta$ that means 'α negates β,' concerning the contradictory sorts [8].

We define a sorted signature on the set $\mathcal{S}^+$ of structured sorts. $\mathcal{F}_n$ is a set of n-ary function symbols $(f, f_0, f_1, \ldots)$, and $\mathcal{P}_n$ is a set of n-ary predicate symbols $(p, p_0, p_1, \ldots)$. Let $\mathcal{S} = \{s_1, \ldots, s_n\}$ be a set of atomic sorts. We introduce *the sort predicates* $p_{s_1}, \ldots, p_{s_n}$ (discussed in [3]) indexed by the sorts $s_1, \ldots, s_n$ where p_{s_i} is a unary predicate (i.e. $p_{s_i} \in \mathcal{P}_1$) and equivalent to the sort s_i. We simply write s for p_s when this will not cause confusion. For example, instead of the formula $p_s(t)$ where t is a term, we use the notation $s(t)$. We denote by $\mathcal{P}_\mathcal{S}$ the set $\{p_s \in \mathcal{P}_1 \mid s \in \mathcal{S} - \{\top, \bot\}\}$ of the sort predicates indexed by all sorts in $\mathcal{S} - \{\top, \bot\}$. A sorted signature extended to include structured sorts and sort predicates is defined as follows.

Definition 2 (Sorted signature on $\mathcal{S}^+$). *A sorted signature on $\mathcal{S}^+$, which we call a structured sort signature, is an ordered quadruple $\Sigma^+ = (\mathcal{S}^+, \mathcal{F}, \mathcal{P}, \Omega)$ satisfying the following conditions:*

(1) $\mathcal{S}^+$ is the set of all structured sorts constructed by $\mathcal{S}$.
(2) $\mathcal{F}$ is the set $\bigcup_{n\geq 0} \mathcal{F}_n$ of all function symbols.
(3) $\mathcal{P}$ is the set $\bigcup_{n\geq 0} \mathcal{P}_n$ of all predicate symbols.
(4) Ω is a set of sort declarations of functions and predicates such that:
  (i) If $f \in \mathcal{F}_n$, then $f\colon s_1 \times \ldots \times s_n \to s \in \Omega$ where $s_1, \ldots, s_n, s \in \mathcal{S} - \{\bot\}$. In particular, if $c \in \mathcal{F}_0$, then $c\colon \to s \in \Omega$.
  (ii) If $p \in \mathcal{P}_n$, then $p\colon s_1 \times \ldots \times s_n \in \Omega$ where $s_1, \ldots, s_n \in \mathcal{S} - \{\bot\}$. In particular, if $p_s \in \mathcal{P}_\mathcal{S}$, then $p_s\colon \top \in \Omega$.

Note that the sort declarations of functions and predicates are given by atomic sorts. The structured sort signatures do not include subsort declarations.

3.2 Sort-Hierarchy Declaration

We will build a sort-hierarchy over $\mathcal{S}^+$, instead of subsort declarations in sorted signatures of typical order-sorted logics. In our logic, we cannot enumerate all the subsort relations on $\mathcal{S}^+$ because the set of subsort declarations representing a subsort relation may be infinite. Hence, we first give a finite set of subsort declarations, so that the subsort relation should be derived by a sort constraint system. For this purpose, we deal with subsort declarations as subsort formulas but not as static expressions in signatures. Let $\Sigma^+ = (\mathcal{S}^+, \mathcal{F}, \mathcal{P}, \Omega)$ be a structured sort signature. For $s, s' \in \mathcal{S}^+$, $s \sqsubseteq_S s'$ is said to be a subsort declaration over Σ^+ that indicates s is a subsort of s'. For instance,

$$player \sqcap winner \sqsubseteq_S person$$

is a subsort declaration over a structured sort signature. We denote by $D_{\mathcal{S}^+} = \{s \sqsubseteq_S s' | s, s' \in \mathcal{S}^+\}$ the set of all subsort declarations on $\mathcal{S}^+$. In the next definition, the sort-hierarchy is obtained by a finite set of subsort declarations.

Definition 3 (Sort-hierarchy declaration). *A sort-hierarchy declaration is an ordered pair $H = (\mathcal{S}^+, D)$, where*

(1) $\mathcal{S}^+$ *is the set of structured sorts constructed by* $\mathcal{S}$*,*
(2) D *is a finite set* $\{s_1 \sqsubseteq_S s'_1, s_2 \sqsubseteq_S s'_2, \ldots\}$ *of subsort declarations on* $\mathcal{S}^+$*.*

Extended declarations on $\mathcal{S}^+$ are defined by subsort declarations as follows.

Definition 4. *A sort equivalence declaration, an exclusivity declaration and a totality declaration are defined respectively by*

- $s =_S s'$ *iff* $s \sqsubseteq_S s'$ *and* $s' \sqsubseteq_S s$*.*
- $s \parallel s'$ *iff* $(s \sqcap s') =_S \bot$*.*
- $s \mid_{s_i} s'$ *iff* $(s \sqcup s') =_S s_i$*.*

We use the abbreviation $s \mid s'$ to denote $s \mid_\top s'$. The above notations are useful for declaring complicated sort relations in a sort-hierarchy declaration $H = (\mathcal{S}^+, D)$.

Example 2. The sort-hierarchy declaration $H = (\mathcal{S}^+, D)$ consists of the set $\mathcal{S}^+$ of structured sorts constructed by

$$\mathcal{S} = \{person, winner, loser, player, \bot, \top\}$$

and the finite set D of subsort declarations with

$$D = \{winner \sqsubseteq_S player,\ player \sqsubseteq_S person,\\ loser \sqsubseteq_S player,\ winner \mid_{player} loser,\ winner \parallel loser\}.$$

The sorts *winner* and *loser* are subsorts of *player*, and the sort *player* is a subsort of *person*. The totality declaration $winner \mid_{player} loser$ indicates that *winner* and *loser* have the property totality in *player*. The exclusivity declaration $winner \parallel loser$ indicates that *winner* and *loser* are mutually exclusive.

3.3 Structured Sort Constraint System

We develop a constraint system with respect to a subsort relation on $\mathcal{S}^+$.

Definition 5. *Let* s, s', s'' *be structured sorts. The axioms and rules of structured sort constraint system CS consist of:*

Reflexivity $s \sqsubseteq_S s$
Idempotency $s \sqsubseteq_S s \sqcap s, \quad s \sqcup s \sqsubseteq_S s$

Commutativity $s \sqcap s' \sqsubseteq_S s' \sqcap s, \quad s \sqcup s' \sqsubseteq_S s' \sqcup s$
Associativity $(s \sqcap s') \sqcap s'' =_S s \sqcap (s' \sqcap s''), \quad (s \sqcup s') \sqcup s'' =_S s \sqcup (s' \sqcup s'')$
Distributivity $(s \sqcup s') \sqcap s'' =_S (s \sqcap s'') \sqcup (s' \sqcap s''), \quad (s \sqcap s') \sqcup s'' =_S (s \sqcup s'') \sqcap (s' \sqcup s'')$

Least and greatest sorts $\bot \sqsubseteq_S s, \quad s \sqsubseteq_S \top$

Conjunction $s \sqcap s' \sqsubseteq_S s, \quad s \sqsubseteq_S s \sqcap \top$
Disjunction $s \sqsubseteq_S s \sqcup s', \quad s \sqcup \bot \sqsubseteq_S s$
Absorption $(s \sqcap s') \sqcup s \sqsubseteq_S s, \quad s \sqsubseteq_S (s \sqcup s') \sqcap s$

Classical negation $s \parallel \overline{s}, \quad s \mid \overline{s}, \quad \overline{s \sqcap s'} =_S \overline{s} \sqcup \overline{s'}, \quad \overline{s \sqcup s'} =_S \overline{s} \sqcap \overline{s'}$
Strong negation $s \parallel\!\sim s, \quad \sim s \sqsubseteq_S \overline{s}$

Transitivity rule $\dfrac{s \sqsubseteq_S s' \quad s' \sqsubseteq_S s''}{s \sqsubseteq_S s''}$

Introduction rule $\dfrac{s \sqsubseteq_S s'}{s'' \sqcap s \sqsubseteq_S s'' \sqcap s'}$

Elimination rule $\dfrac{s \sqcup s' \sqsubseteq_S s \sqcup s'' \quad s \parallel s' \quad s \parallel s''}{s' \sqsubseteq_S s''}$.

A derivation of an expression (a subsort declaration, or a clause which we will define) from a set of expressions is defined as follows.

Definition 6 (Derivation). *Let Δ a set of expressions. A derivation of F_n in a system X from Δ is a finite sequence $F_1, F_2, \ldots, F_n$ such that*

(i) $F_i \in \Delta$,
(ii) F_i is an axiom of system X, or
(iii) F_i follows from $F_j (j < i)$ by one of the rules of system X.

We write $\Delta \vdash_X F$ if F has a derivation from Δ in the system X. This notion of derivations can be used for the structured sort constraint system CS, and a clausal inference system which we will present.

3.4 Sorted Terms and Formulas with Structured Sort Constraints

An alphabet for an order-sorted first-order language $\mathcal{L}_{\mathcal{S}^+}$ of structured sort signature Σ^+ contains the following: the set $\mathcal{V} = \bigcup_{s \in \mathcal{S} - \{\bot\}} \mathcal{V}_s$ of variables for all atomic sorts in $\mathcal{S} - \{\bot\}$ (where $\mathcal{V}_s$ is a set of variables $x_1 \colon s, x_2 \colon s, \ldots$ for atomic sort s), the connectives $\neg, \wedge, \vee, \rightarrow$, the quantifiers $\forall, \exists$, and the auxiliary parentheses and commas.

We give the expressions *sorted term* and *formula* for our order-sorted first-order language with structured sorts.

Definition 7 (Sorted terms). *Let $\Sigma^+ = (\mathcal{S}^+, \mathcal{F}, \mathcal{P}, \Omega)$ be an structured sort signature and let $H = (\mathcal{S}^+, D)$ be a sort-hierarchy declaration. The set $TERM_{\Sigma^+, s}$ of terms of sort s is defined by:*

(1) A variable $x \colon s$ is a term of sort s.
(2) A constant $c \colon s$ is a term of sort s where $c \in \mathcal{F}_0$ and $c \colon \rightarrow s \in \Omega$.
(3) If $t_1, \ldots, t_n$ are terms of sorts $s_1, \ldots, s_n$, then $f(t_1, \ldots, t_n) \colon s$ is a term of sort s where $f \in \mathcal{F}_n$ and $f \colon s_1 \times \ldots \times s_n \rightarrow s \in \Omega$.
(4) If t is a term of sort s' with $D \vdash_{CS} s' \sqsubseteq_S s$, then t is a term of sort s.

We denote by $TERM_{\Sigma^+}$ the set of all sorted terms $\bigcup_{s \in \mathcal{S} - \{\bot\}} TERM_{\Sigma^+, s}$.

We define a structured sort substitution with respect to a subsort relation derivable in the constraint system CS. That is, the subsort declarations are obtained by an application of the rules from CS so that the substitution is defined via the subsort declarations.

Definition 8 (Structured sort substitution). *A structured sort substitution is a function θ mapping from a finite set of variables to $TERM_{\Sigma^+}$ where $\theta(x\colon s) \neq x\colon s$ and $\theta(x\colon s) \in TERM_{\Sigma^+,s}$*[3].

In the above definition none of the terms of sort $\bot$ can be substituted for variables. If there do not exist subsorts s' of s such that $s' \neq \bot$, then the substitutions correspond to many-sorted substitutions (i.e. not order-sorted substitutions).

Definition 9 (Sorted formulas). *Let $\Sigma^+ = (\mathcal{S}^+, \mathcal{F}, \mathcal{P}, \Omega)$ be a structured sort signature and let $H = (\mathcal{S}^+, D)$ be a sort-hierarchy declaration. The set $FORM_{\Sigma^+}$ of sorted formulas is defined by:*

(1) If $t_1, \ldots, t_n$ are terms of $s_1, \ldots, s_n$, then $p(t_1, \ldots, t_n)$ is an atomic formula (or simply an atom) where $p \in \mathcal{P}_n$ and $p\colon s_1 \times \ldots \times s_n \in \Omega$,
(2) If A and B are formulas, then $(\neg A)$, $(A \wedge B)$, $(A \vee B)$, $(A \to B)$, $(\forall x\colon sA)$, and $(\exists x\colon sA)$ are formulas.

We introduce literals in order to represent formulas in clause form. A positive literal is an atomic formula $p(t_1, \ldots, t_n)$, and a negative literal is the negation $\neg p(t_1, \ldots, t_n)$ of an atomic formula. A literal is a positive or a negative literal.

Definition 10. *Let $L_1, \ldots, L_n$ be literals. The formula $L_1 \vee \ldots \vee L_n (n \geq 0)$ is said to be a clause. We denote by CL_{Σ^+} the set of all clauses.*

3.5 Σ^+-structure

As in the semantics of standard order-sorted logics, we consider a structure that consists of the universe and an interpretation over $\mathcal{S}^+ \cup \mathcal{F} \cup \mathcal{P}$ and satisfies the sort declarations of functions and predicates on $\mathcal{S}$. The interpretation of atomic sorts is defined by subsets of the universe. Hence, the interpretation of structured sorts is constructed by the interpretation of atomic sorts and the operations of set theory.

Definition 11. *Given a structured sort signature $\Sigma^+ = (\mathcal{S}^+, \mathcal{F}, \mathcal{P}, \Omega)$, a Σ^+-structure is an ordered pair $M^+ = (U, I^+)$ such that*

(1) U is a non-empty set.
(2) I^+ is a function on $\mathcal{S}^+ \cup \mathcal{F} \cup \mathcal{P}$ where
- *$I^+(s) \subseteq U$ (in particular, $I^+(\top) = U$ and $I^+(\bot) = \emptyset$),*
  $I^+(s \sqcap s') = I^+(s) \cap I^+(s')$,
  $I^+(s \sqcup s') = I^+(s) \cup I^+(s')$,
  $I^+(\overline{s}) = I^+(\top) - I^+(s)$,
  $I^+(\sim s) \subseteq I^+(\top) - I^+(s)$,
- *$I^+(f)\colon I^+(s_1) \times \ldots \times I^+(s_n) \to I^+(s)$ where $f \in \mathcal{F}_n$ and $f\colon s_1 \times \ldots \times s_n \to s \in \Omega$,*

[3] In order to substitute variables with terms of the subsorts, the set $TERM_{\Sigma^+,s}$ of terms of sort s contain the terms of their subsorts obtained by subsort declarations that are derivable using a sort constraint system.

- $I^+(p) \subseteq I^+(s_1) \times \ldots \times I^+(s_n)$ *where* $p \in \mathcal{P}_n$ *and* $p\colon s_1 \times \ldots \times s_n \in \Omega$ *(in particular,* $I^+(p_s) = I^+(s)$ *where* $p_s \in \mathcal{P}_\mathcal{S}$ *and* $p_s\colon \top \in \Omega$*).*

A variable assignment (or simply an assignment) in a Σ^+-structure $M^+ = (I^+, U)$ is a function $\alpha\colon \mathcal{V} \to U$ where $\alpha(x\colon s) \in I^+(s)$ for all variables $x\colon s \in \mathcal{V}$. Let α be an assignment in a Σ^+-structure $M^+ = (I^+, U)$, let $x\colon s$ be a variable in $\mathcal{V}$, and $d \in I^+(s)$. The assignment $\alpha[d/x\colon s]$ is defined by $\alpha[d/x\colon s] = (\alpha - \{(x\colon s, \alpha(x\colon s))\}) \cup \{(x\colon s, d)\}$.

We now define an interpretation over structured sort signatures Σ^+. If an interpretation $\mathcal{I}^+$ consists of a Σ^+-structure M^+ and an assignment α in M^+, then $\mathcal{I}^+$ is said to be a Σ^+-interpretation.

Definition 12. *Let* $\mathcal{I}^+ = (M^+, \alpha)$ *be a* Σ^+*-interpretation. The denotation* $[\![\ ]\!]_\alpha$ *is defined by*

(1) $[\![x\colon s]\!]_\alpha = \alpha(x\colon s)$,
(2) $[\![c\colon s]\!]_\alpha = I^+(c)$ *with* $I^+(c) \in I^+(s)$,
(3) $[\![f(t_1, \ldots, t_n)\colon s]\!]_\alpha = I^+(f)([\![t_1]\!]_\alpha, \ldots, [\![t_n]\!]_\alpha)$.

We formalize a satisfiability relation indicating that a Σ^+-interpretation satisfies sorted formulas and subsort declarations.

Definition 13. *Let* $\mathcal{I}^+ = (M^+, \alpha)$ *be a* Σ^+*-interpretation and let* F *be a sorted formula or a subsort declaration. We define the satisfiability relation* $\mathcal{I} \models F$ *by the following rules:*

(1) $\mathcal{I}^+ \models p(t_1, \ldots, t_n)$ *iff* $([\![t_1]\!]_\alpha, \ldots, [\![t_n]\!]_\alpha) \in I^+(p)$,
(2) $\mathcal{I}^+ \models (\neg A)$ *iff* $\mathcal{I}^+ \not\models A$,
(3) $\mathcal{I}^+ \models (A \wedge B)$ *iff* $\mathcal{I}^+ \models A$ *and* $\mathcal{I}^+ \models B$,
(4) $\mathcal{I}^+ \models (A \vee B)$ *iff* $\mathcal{I}^+ \models A$ *or* $\mathcal{I}^+ \models B$,
(5) $\mathcal{I}^+ \models (A \to B)$ *iff* $\mathcal{I}^+ \not\models A$ *or* $\mathcal{I}^+ \models B$,
(6) $\mathcal{I}^+ \models (\forall x\colon s)A$ *iff for all* $d \in I^+(s)$, $\mathcal{I}^+[d/x\colon s] \models A$ *holds,*
(7) $\mathcal{I}^+ \models (\exists x\colon s)A$ *iff for some* $d \in I^+(s)$, $\mathcal{I}^+[d/x\colon s] \models A$ *holds,*
(8) $\mathcal{I}^+ \models s \sqsubseteq_S s'$ *iff* $I^+(s) \subseteq I^+(s')$.

Let F be a sorted formula or a subsort declaration and let $\Gamma \subseteq \mathit{FORM}_{\Sigma^+} \cup D_{\mathcal{S}^+}$. If $\mathcal{I}^+ \models F$, then $\mathcal{I}^+$ is said to be a Σ^+-model of F. We denote $\mathcal{I}^+ \models \Gamma$ if $\mathcal{I}^+ \models F$ for every $F \in \Gamma$. If $\mathcal{I}^+ \models \Gamma$, then $\mathcal{I}^+$ is said to be a Σ^+-model of Γ. If Γ has a Σ^+-model , then Γ is Σ^+-satisfiable. If Γ has no Σ^+-model , then Γ is Σ^+-unsatisfiable. If every Σ^+-interpretation $\mathcal{I}^+$ is a Σ^+-model of F, then F is said to be Σ^+-valid. We write $\Gamma \models_{\Sigma^+} F$ (F is a consequence of Γ in the class of Σ^+-structures) if every Σ^+-model of Γ is a Σ^+-model of F ($\in \mathit{FORM}_{\Sigma^+} \cup D_{\mathcal{S}^+}$).

Let $H = (\mathcal{S}^+, D)$ be a sort-hierarchy declaration and Δ a set of clauses. In the clausal inference system we will present in the next section, their rules are applied to clauses in Δ (which expresses an assertional knowledge base), related to a subsort relation derivable from D. If $\mathcal{I}^+$ is a Σ^+-model of both D and Δ, then $\mathcal{I}^+$ is said to be a Σ^+-model of (D, Δ), denoted by $\mathcal{I}^+ \models (D, \Delta)$. We write $(D, \Delta) \models_{\Sigma^+} F$ (F is a consequence of (D, Δ) in the class of Σ^+-structures) if every Σ^+-model of (D, Δ) is a Σ^+-model of F ($\in \mathit{FORM}_{\Sigma^+} \cup D_{\mathcal{S}^+}$).

4 Resolution with Structured Sorts

In addition to structured sort constraint system CS, we design a (clausal) resolution system with structured sorts. We adopt the method (proposed in [3]) of coupling a clausal knowledge base [13,11] and a sort-hierarchy in which every sort can be used to express the sort predicate which is included in clauses. Then, we define a hybrid inference system in order to combine the two systems.

4.1 Clausal Inference System with Sort Predicates

We present a clausal inference system, in which clauses may include sort predicates, e.g., $p(t_1, t_2) \vee s(t)$ where s is a sort predicate.

Definition 14 (Cut rule). *Let L, L' be positive literals and C, C' clauses.*

$$\frac{\neg L \vee C \quad L' \vee C'}{(C \vee C')\theta}$$

where there exists a mgu θ for L and L'.

The cut rule is one of the usual rules included in clausal inference systems. In addition to the cut rule, our clausal inference system have to include inference rules of sort predicates related to subsort declarations. We introduce the inference rules for resolution as follows.

Definition 15 (Resolution rules with sort predicates). *Let s, s', s_i be structured sorts or sort predicates, L, L' positive literals, t, t' sorted terms, and C, C' clauses. Resolution rules with sort predicates are given as follows.*

Subsort rule.

$$\frac{\neg s(t) \vee C \quad s'(t') \vee C' \quad s' \sqsubseteq_S s}{(C \vee C')\theta}$$

where there exists a mgu θ for t and t'.

Sort predicate rule. [4]

$$\frac{\neg s(t) \vee C \quad s' \sqsubseteq_S s}{C}$$

where $t \in TERM_{\Sigma^+, s'}$.

Exclusivity rule.

$$\frac{s(t) \vee C \quad s'(t') \vee C' \quad s \parallel s'}{(C \vee C')\theta}$$

where there exists a mgu θ for t and t'.

[4] Instead of the sort predicate rule, the subsort rule can derive the same results by adding valid atoms with sort predicates.

Totality rule.

$$\frac{s_i(t) \vee C \quad \neg s(t') \vee C' \quad s \mid_{s_i} s'}{(s'(t) \vee C \vee C')\theta}$$

where there exists a mgu θ for t and t'.

In particular, the exclusivity rule and the totality rule are useful for resolutions with respect to implicit negations embedded in a sort-hierarchy. The exclusivity rule will be applied when an opposite sort is declared as $s \parallel s'$. We write *resolution system MS* for the system defined by the cut rule in Definition 14 and the resolution rules in Definition 15.

4.2 Hybrid Inference System with Clauses and Structured Sort Constraints

We define a hybrid inference system obtained by combining a clausal inference system with a sort constraint system. The inference rules in the hybrid system are applied to subsort declarations and clauses including sort predicates, so that they can deal with sort-hierarchy information in an assertional knowledge base.

Definition 16 (Hybrid inference system). *A hybrid inference system is a system obtained by adding the axioms and rules in a constraint system into a clausal inference system. We write $X+Y$ for the hybrid inference system obtained from a clausal inference system X and a constraint system Y.*

The hybrid inference system $X+Y$ can be regarded as an extension of the clausal inference system X. We write $(D, \Delta) \vdash_{X+Y} F$ to denote $D \cup \Delta \vdash_{X+Y} F$.

Lemma 1. *The axioms of the structured sort constraint system CS are Σ^+-valid.*

Lemma 2. *Let $F, F_1, \ldots, F_n$ be subsort declarations. The conclusion F of each rule in the structured sort constraint system CS is a consequence of its premise $\{F_1, \ldots, F_n\}$ in the class of Σ^+-structures. That is, $\{F_1, \ldots, F_n\} \models_{\Sigma^+} F$.*

Proof. Elimination rule: Suppose that $I^+(s) \cup I^+(s') = I^+(s) \cup I^+(s'')$, $I^+(s) \cap I^+(s') = \emptyset$, and $I^+(s) \cap I^+(s'') = \emptyset$. Let $d \in I^+(s')$. Since $I^+(s') \subseteq I^+(s) \cup I^+(s') \subseteq I^+(s) \cup I^+(s'')$, we have $d \in I^+(s) \cup I^+(s'')$. $d \in I(s')$ and $I^+(s) \cap I^+(s') = \emptyset$ imply $d \notin I^+(s)$. Therefore $d \in I^+(s'')$. Similarly, the other rules can be proved. ∎

Lemma 3. *Let $F, F_1, \ldots, F_n$ be clauses or subsort declarations. The conclusion F of each rule in the resolution system RS is a consequence of its premise $\{F_1, \ldots, F_n\}$ in the class of Σ^+-structures. That is, $\{F_1, \ldots, F_n\} \models_{\Sigma^+} F$.*

Proof. For each rule we show $\{F_1, \ldots, F_n\} \models_{\Sigma^+} F$.

1. Exclusivity rule: Suppose that $\mathcal{I}^+ \models s(t) \vee C$, $\mathcal{I}^+ \models s'(t') \vee C'$, and $I^+(s) \cap I^+(s') = \emptyset$. Let θ be a structured sort substitution such that $\theta(t) = \theta(t')$. So $\mathcal{I}^+ \models (s(t) \vee C)\theta$ and $\mathcal{I}^+ \models (s'(t') \vee C')\theta$. By $I^+(s) \cap I^+(s') = \emptyset$, either $\mathcal{I}^+ \models s(t)\theta$ or $\mathcal{I}^+ \models s'(t')\theta$ does not hold. By the hypothesis, $\mathcal{I}^+ \models C\theta$ or $\mathcal{I}^+ \models C'\theta$. Therefore $\mathcal{I}^+ \models C\theta \vee C'\theta$
2. Totality rule: Assume that $\mathcal{I}^+ \models s_i(t) \vee C$, $\mathcal{I}^+ \models \overline{s'}(t') \vee C'$, and $\mathcal{I}^+ \models s \mid_{s_i} s'$, i.e. $I^+(s) \cup I^+(s') = I^+(s_i)$. Let θ be a structured sort substitution such that $\theta(t) = \theta(t')$. If $I^+(s) \cup I^+(s') = I^+(s_i)$, then $\mathcal{I}^+ \models s(t) \vee s'(t') \vee C$. Then, we can obtain $\mathcal{I}^+ \models s'(t) \theta \vee C\theta \vee C'\theta$. Therefore the conclusion is a consequence of its premise. ∎

The next theorem shows the soundness of the structured sort constraint system CS and the resolution system RS.

Theorem 1. *Let $H = (\mathcal{S}^+, D)$ be a sort-hierarchy declaration, Δ a set of clauses, and X a system. If $(D, \Delta) \vdash_X F$, then $(D, \Delta) \models_{\Sigma^+} F$.*

Proof. By Lemma 1, 2, and 3, this is proved. ∎

We give the notion of contradiction in an exclusivity relation from the sort-hierarchy. This notion is defined by deciding whether there is a contradiction between an opposite sort and its antonymous sort.

Definition 17. *Let $H = (\mathcal{S}^+, D)$ be a sort-hierarchy declaration, Δ a set of clauses, and X a system. (D, Δ) is said to be contradictory on an exclusivity relation if there exists sorts s, s' such that $(D, \Delta) \vdash_X s \parallel s'$ and $(D, \Delta) \vdash_X s(t)$ and $(D, \Delta) \vdash_X s'(t)$. (D, Δ) is said to be logically contradictory if $(D, \Delta) \vdash_X A$ and $(D, \Delta) \vdash_X \neg A$.*

The contradiction between A and $\neg A$ (corresponding to "logically contradictory" in the above definition) is defined in the usual manner of logics. We say that (D, Δ) is consistent if (D, Δ) is neither contradictory on an exclusivity relation nor logically contradictory.

Theorem 2. *Let $H = (\mathcal{S}^+, D)$ be a sort-hierarchy declaration and Δ a set of clauses. If (D, Δ) has a Σ^+-model, then (D, Δ) is consistent.*

Proof. Suppose that $\mathcal{I}^+$ is a Σ^+-model of (D, Δ). If (D, Δ) is contradictory on an exclusivity relation, then there exists s, s' such that $(D, \Delta) \vdash_X s \parallel s'$ and $(D, \Delta) \vdash_X s(t)$ and $(D, \Delta) \vdash_X s'(t)$. By Theorem 1, $\mathcal{I}^+ \models s \parallel s'$ and then $\mathcal{I}^+ \models s(t)$ and $\mathcal{I}^+ \models s'(t)$. Then $I^+(s) \cap I^+(s') = \emptyset$ but $[\![t]\!]_\alpha \in I^+(s)$ and $[\![t]\!]_\alpha \in I^+(s')$. If (D, Δ) is logically contradictory, then $\mathcal{I}^+ \models \neg A$ and $\mathcal{I}^+ \models A$. Hence, the both cases are contradiction to the hypothesis. Therefore (D, Δ) is consistent. ∎

A refutation is a derivation of the empty clause (denoted $\Box$) from (D, Δ), written as $(D, \Delta) \vdash_X \Box$. The next corollary guarantees that the hybrid inference system $CS + RS$ is sound.

Corollary 1. *Let $H = (\mathcal{S}^+, D)$ be a sort-hierarchy declaration and Δ a set of clauses. If $(D, \Delta) \vdash_{CS+RS} \Box$, then $(D, \Delta) \models_{\Sigma^+} \Box$.*

Proof. When the empty clause $\square$ is derived, the final rule applied in the refutation must be one of the rules in the resolution system RS. We consider each case as follows:

1. Cut rule: There exists a structured sort substitution θ such that $L\theta = L'\theta$, and $(D, \Delta) \vdash_{CS+RS} \neg L$ and $(D, \Delta) \vdash_{CS+RS} L'$. So, by Theorem 1, we have $(D, \Delta) \models_{\Sigma^+} \neg L$ and $(D, \Delta) \models_{\Sigma^+} L'$. Now assume that $\mathcal{I}^+$ is a Σ^+-model of (D, Δ). Then $\mathcal{I}^+ \models L\theta$ and $\mathcal{I}^+ \not\models L'\theta (= L\theta)$ contradicts our assumption. Since (D, Δ) has no Σ^+-model, $(D, \Delta) \models_{\Sigma^+} \square$ is proved.
2. Resolution rules: Similar to 1. ∎

5 Conclusions

This paper has presented an order-sorted logic that can deal with implicit negations in a sort hierarchy. We have presented a hybrid inference system that consists of a clausal inference system and a structured sort constraint system. This system includes structured sort expressions composed of atomic sorts, connectives, and negative operators, in order to deal with implicitly negative sorts embedded in a sort-hierarchy. To represent these negative sorts, we have proposed the notions of sort relations (subsort relation, equivalence relation, exclusivity relation, and totality relation) on the structured sorts, and we have axiomatized the properties of implicitly negative sorts. Thus, the structured sort constraint system can derive relationships between classical negation, strong negation, and antonyms in a sort-hierarchy. Furthermore, the contradiction in the sort-hierarchy as defined by the exclusivity relation enables us to prove the soundness of our logic with structured sorts.

We need to improve our hybrid inference system in order to tackle implementation issues caused by the complicated sort expressions. As a work which remains theoretical, the complete system must be given by revising the axioms and rules.

References

1. Baader, F., & Hanschke, P. (1991). A scheme for integrating concrete domains into concept languages. *Pages 452–457 of: Twelfth international conference on artificial intelligence.*
2. Baader, F., & Sattker, U. (1999). Expressive number restrictions in description logics. *Journal of logic and computation*, **9**(3), 319–350.
3. Beierle, C., Hedtsuck, U., Pletat, U., Schmitt, P.H., & Siekmann, J. (1992). An order-sorted logic for knowledge representation systems. *Artificial intelligence*, **55**, 149–191.
4. Cohn, A. G. (1987). A more expressive formulation of many sorted logic. *Journal of automated reasoning*, **3**, 113–200.
5. Cohn, A. G. (1989). Taxonomic reasoning with many sorted logics. *Artificial intelligence review*, **3**, 89–128.

6. Donini, F. D., Lenzerini, M., Nardi, D., & Schaerf, A. (1996). Reasoning in description logic. Brewka, G. (ed), *Principles of knowledge representation*. CSLI Publications, FoLLI.
7. Frisch, Alan M. (1991). The substitutional framework for sorted deduction: fundamental results on hybrid reasoning. *Artificial intelligence*, **49**, 161–198.
8. Gabbay, D., & Hunter, A. (1999). Negation and contradiction. Gabbay, D. M., & Wansing, H. (eds), *What is negation?* Kluwer Academic Publishers.
9. Gallier, Jean H. (1986). *Logic for computer science. foundations of automatic theorem proving*. Harper & Row.
10. Horrocks, I. (1999). A description logic with transitive and inverse roles and role hierarchies. *Journal of logic and computation*, **9**(3), 385–410.
11. Lobo, J., Minker, J., & Rajasekar, A. (1992). *Foundations of disjunctive logic programming*. The MIT Press.
12. Ota, A. (1980). *Hitei no imi (in Japanese)*. Taishukan.
13. Richards, T. (1989). *Clausal form logic. an introduction to the logic of computer reasoning*. Addison-Wesley Publishing Company.
14. Schmidt-Schauss, M. (1989). *Computational aspects of an order-sorted logic with term declarations*. Springer-Verlag.
15. Schmidt-Schauss, M., & Smolka, G. (1991). Attributive concept descriptions with complements. *Artificial intelligence*, **48**, 1–26.
16. Smolka, G. (1992). Feature-constraint logics for unification grammars. *Journal of logic programming*, **12**, 51–87.
17. Wagner, G. (1991). Logic programming with strong negation and inexact predicates. *Journal of logic computation*, **1**(6), 835–859.
18. Walther, C. (1987). *A many-sorted calculus based on resolution and paramodulation*. Pitman and Kaufman Publishers.
19. Weibel, T. (1997). An order-sorted resolution in theory and practice. *Theoretical computer science*, **185**(2), 393–410.

Logic Programming in a Fragment of Intuitionistic Temporal Linear Logic

Mutsunori Banbara[1], Kyoung-Sun Kang[2],
Takaharu Hirai[3], and Naoyuki Tamura[4]

[1] Department of Mathematics, Nara National College of Technology,
22 Yata, Yamatokoriyama, 639-1080, Japan
banbara@libe.nara-k.ac.jp
[2] Department of Computer Engineering, Pusan University of Foreign Studies,
55-1, Uam-dong, Nam-gu, Pusan, 608-738, Korea
kskang@taejo.pufs.ac.kr
[3] Graduate School of Science and Technology, Kobe University,
1-1 Rokkodai, Nada, Kobe, 657-8501, Japan
hirai@pascal.cs.kobe-u.ac.jp
[4] Department of Computer and Systems Engineering, Kobe University,
1-1 Rokkodai, Nada, Kobe, 657-8501, Japan
tamura@kobe-u.ac.jp

Abstract. Recent development of logic programming languages based on linear logic suggests a successful direction to extend logic programming to be more expressive and more efficient. The treatment of formulas-as-resources gives us not only powerful expressiveness, but also efficient access to a large set of data. However, in linear logic, whole resources are kept in one context, and there is no straight way to represent complex data structures as resources. For example, in order to represent an ordered list and time-dependent data, we need to put additional indices for each resource formula. This paper describes a logic programming language, called TLLP, based on intuitionistic temporal linear logic. This logic, an extension of linear logic with some features from temporal logics, allows the use of the modal operators '$\bigcirc$'(next-time) and '$\Box$'(always) in addition to the operators used in intuitionistic linear logic. The intuitive meaning of modal operators is as follows: $\bigcirc\, B$ means that B can be used exactly once at the next moment in time; $\Box\, B$ means that B can be used exactly once any time; $!\, B$ means that B can be used arbitrarily many times (including 0 times) at any time. We first give a proof theoretic formulation of the logic of the TLLP language. We then present a series of resource management systems designed to implement not only interpreters but also compilers based on an extension of the standard WAM model.

1 Introduction

Linear logic was introduced by J.-Y.Girard in 1987 [4] as a resource-conscious refinement of classical logic. Since then a number of logic programming languages

P. Codognet (Ed.): ICLP 2001, LNCS 2237, pp. 315–330, 2001.

based on linear logic have been proposed: LO[1], ACL[12], Lolli[3][8][9], Lygon[5], Forum[13], and LLP[2][15].

These languages suggest a direction to extend logic programming to be more expressive and more efficient. The treatment of formulas-as-resources gives us not only powerful expressiveness, but also efficient access to a large set of data. However, in linear logic, whole resources are kept in one context, and there is no straight way to represent complex data structures as resources. For example, in order to represent an ordered list and time-dependent data, we need to put additional indices for each resource formula.

Temporal Linear Logic (TLL) is an extension of linear logic with some features of temporal logic. TLL was first studied by Kanovich and Itoh [11], and a cut-free sequent system has been proposed by Hirai [6]. The semantics model of TLL consists an infinite number of phase spaces linearly ordered by the time clock. Each phase space is the same as that of linear logic.

This paper describes a logic programming language, called TLLP, based on intuitionistic temporal linear logic [6]. We first give a proof theoretic formulation of the logic of the TLLP language. We then present a series of resource management systems designed to implement not only interpreters but also compilers based on an extension of the standard WAM model. Finally, we describe some implementation methods based on our systems.

2 Intuitionistic Temporal Linear Logic

In this paper, we will focus on the sequent system *ITLL* [6] of intuitionistic temporal linear logic developed by Hirai. The expressive power of *ITLL* is shown by a natural encoding of Timed Petri Nets. It is this logic that we shall use to design and implement the logic programming language described below.

ITLL allows the use of the modal operators '$\bigcirc$'(next-time) and '$\Box$'(always) in addition to the operators used in intuitionistic linear logic. Compared with the sequent system *ILL* (see Fig. 1) of intuitionistic linear logic, three rules (L $\Box$), (R $\Box$), and ($\bigcirc$) are added. The entire set of *ITLL* sequent rules is given in Fig. 2. Here, the left-hand side of sequents are multisets of formulas, and the structural rule for exchange need not be explicitly stated. The structural rule for weakening (W!) and contraction (C!) are available only for assumptions marked with the modal operator '!'. This means that, in general, formulas not !-marked can be used exactly once. Limited-use formulas can represent time-dependent resources in *ITLL*. The intuitive meaning of these modal operators is as follows:

- $\bigcirc B$ means that B can be used exactly once at the next moment in time.
- $\Box B$ means that B can be used exactly once any time.
- $!B$ means that B can be used arbitrarily many times (including 0 times) at any time.

By combining these modalities with binary operators in linear logic, several resources can be expressed. For example, $B \,\&\, \bigcirc B$ means that B can be used exactly once either at the present time or at the next moment in time. $\bigcirc(1 \,\&\, B)$ means that B can be used at most once at the next moment in time.

$$\frac{}{B \longrightarrow B}\ (\text{Identity}) \qquad \frac{\Delta_1 \longrightarrow B \quad \Delta_2, B \longrightarrow C}{\Delta_1, \Delta_2 \longrightarrow C}\ (\text{Cut})$$

$$\frac{}{\Delta, 0 \longrightarrow C}\ (\text{L0}) \qquad \frac{}{\Delta \longrightarrow \top}\ (\text{R}\top)$$

$$\frac{\Delta \longrightarrow C}{\Delta, 1 \longrightarrow C}\ (\text{L1}) \qquad \frac{}{\longrightarrow 1}\ (\text{R1})$$

$$\frac{\Delta, B_i \longrightarrow C}{\Delta, B_1 \,\&\, B_2 \longrightarrow C}\ (\text{L}\&_i) \qquad \frac{\Delta \longrightarrow C_1 \quad \Delta \longrightarrow C_2}{\Delta \longrightarrow C_1 \,\&\, C_2}\ (\text{R}\&)$$

$$\frac{\Delta, B_1, B_2 \longrightarrow C}{\Delta, B_1 \otimes B_2 \longrightarrow C}\ (\text{L}\otimes) \qquad \frac{\Delta_1 \longrightarrow C_1 \quad \Delta_2 \longrightarrow C_2}{\Delta_1, \Delta_2 \longrightarrow C_1 \otimes C_2}\ (\text{R}\otimes)$$

$$\frac{\Delta, B_1 \longrightarrow C \quad \Delta, B_2 \longrightarrow C}{\Delta, B_1 \oplus B_2 \longrightarrow C}\ (\text{L}\oplus) \qquad \frac{\Delta \longrightarrow C_i}{\Delta \longrightarrow C_1 \oplus C_2}\ (\text{R}\oplus_i)$$

$$\frac{\Delta_1 \longrightarrow C_1 \quad \Delta_2, B \longrightarrow C_2}{\Delta_1, \Delta_2, C_1 \multimap B \longrightarrow C_2}\ (\text{L}\multimap) \qquad \frac{\Delta, B \longrightarrow C}{\Delta \longrightarrow B \multimap C}\ (\text{R}\multimap)$$

$$\frac{\Delta, B \longrightarrow C}{\Delta, !\,B \longrightarrow C}\ (\text{L}!) \qquad \frac{!\,\Delta \longrightarrow C}{!\,\Delta \longrightarrow !\,C}\ (\text{R}!)$$

$$\frac{\Delta \longrightarrow C}{\Delta, !\,B \longrightarrow C}\ (\text{W}!) \qquad \frac{\Delta, !\,B, !\,B \longrightarrow C}{\Delta, !\,B \longrightarrow C}\ (\text{C}!)$$

$$\frac{\Delta, B[t/x] \longrightarrow C}{\Delta, \forall x.B \longrightarrow C}\ (\text{L}\forall) \qquad \frac{\Delta \longrightarrow C[t/x]}{\Delta \longrightarrow \exists x.C}\ (\text{R}\exists)$$

$$\frac{\Delta, B[y/x] \longrightarrow C}{\Delta, \exists x.B \longrightarrow C}\ (\text{L}\exists) \qquad \frac{\Delta \longrightarrow C[y/x]}{\Delta \longrightarrow \forall x.C}\ (\text{R}\forall)$$

provided, in each case, y does not appear free in the conclusion.

Fig. 1. The proof system *ILL* for intuitionistic linear logic

(Rules of *ILL*)

$$\frac{\Delta, B \longrightarrow C}{\Delta, \Box\, B \longrightarrow C}\ (\text{L}\Box) \qquad \frac{!\,\Gamma, \Box\, \Sigma \longrightarrow C}{!\,\Gamma, \Box\, \Sigma \longrightarrow \Box\, C}\ (\text{R}\Box) \qquad \frac{!\,\Gamma, \Box\, \Sigma, \Delta \longrightarrow C}{!\,\Gamma, \Box\, \Sigma, \bigcirc\, \Delta \longrightarrow \bigcirc\, C}\ (\bigcirc)$$

Fig. 2. The proof system *ITLL* for intuitionistic temporal linear logic

Two formulas B and C are equivalent, denoted $B \equiv C$, if the sequents $B \longrightarrow C$ and $C \longrightarrow B$ are provable in *ITLL*. The notation $\bigcirc^n$ means n multiplicity of $\bigcirc$. We note the following sequents that are provable in *ITLL*.

$$!\,B \equiv !\,!\,B, \qquad \Box\, B \equiv \Box\, \Box\, B, \qquad !\,B \equiv \Box\, !\,B,$$
$$!\,B \longrightarrow \Box\, B \otimes \cdots \otimes \Box\, B, \qquad \Box\, B \longrightarrow \bigcirc^n B \quad (n \geq 0)$$

The main differences from other temporal linear logic systems [11][16] are that *ITLL* includes the modal operator '!', and it satisfies a cut elimination

theorem. Both of these additions are very important for the design of a language based on the notion of *Uniform Proofs*.

3 Language Design

The idea of uniform proofs [14], proposed by Miller et. al, is a simple and powerful notion for designing logic programming languages. Uniform proof search is a cut-free, *goal-directed proof search* in which a sequent $\Gamma \longrightarrow G$ denotes the state of the computation trying to solve the goal G from the program Γ. Goal-directed proof search is characterized operationally by the bottom-up construction of proofs in which right-introduction rules are applied first and left-introduction rules are applied only when the right-hand side is atomic. This means that the operators in the goal G are executed independently from the program Γ, and the program is only considered when its goal is atomic. A logical system is an *abstract logic programming language* if restricting it to uniform proofs retains completeness. The logics of Prolog, λProlog, and Lolli are examples of abstract logic programming language.

Clearly, intuitionistic linear logic (even over the connectives: $\top$, $\&$, $\otimes$, $\multimap$, $!$, and $\forall$) is not an abstract logic programming language. For example, the sequents $a \otimes b \longrightarrow b \otimes a$ and $!\, a \,\&\, b \longrightarrow !\, a$ are both provable in *ILL* but do not have uniform proofs.

Hodas and Miller have designed the linear logic programming language Lolli [7][8] by restricting formulas so that the above counterexamples do not appear, although it retains desirable features of linear logic connectives such as ! and $\otimes$. The Lolli language is based on the following fragment of linear logic:

$$R ::= \top \mid A \mid R_1 \,\&\, R_2 \mid G \multimap R \mid G \Rightarrow R \mid \forall x.R$$
$$G ::= 1 \mid \top \mid A \mid G_1 \otimes G_2 \mid G_1 \,\&\, G_2 \mid G_1 \oplus G_2 \mid R \multimap G \mid R \Rightarrow G \mid !\, G \mid \forall x.G \mid \exists x.G$$

Here, R-formulas and G-formula are called *resource* and *goal formulas* respectively. The connective $\Rightarrow$ is called *intuitionistic implication*, and it is defined as $B \Rightarrow C \equiv (!\, B) \multimap C$.

The sequent of Lolli is of the form $\Gamma; \Delta \longrightarrow G$ where Γ is a set of resource formulas, Δ is a multiset of resource formulas, and G is a goal formula. Γ and Δ are called *intuitionistic* and *linear context* respectively, and they correspond to the *program*. G is called the *goal*. The sequent $\Gamma; \Delta \longrightarrow G$ can be mapped to the linear logic sequent $!\,\Gamma, \Delta \longrightarrow G$. Thus, the right introduction rule for $\multimap$ adds its assumption (called a *linear resource*) to the linear context, in which every formula can be used exactly once. The right introduction rule for $\Rightarrow$ adds its assumption (called an *intuitionistic resource*) to the intuitionistic context, in which every formula can be used arbitrarily many times (including 0 times).

Hodas and Miller developed a series of proof systems $\mathcal{L}$ (see Fig. 3) and $\mathcal{L}'$ in [7]. They proved that $\mathcal{L}$ is sound and complete with respect to the *ILL* rules restricted to the Lolli language. They also proved $\mathcal{L}$ preserves completeness even if probability is restricted to uniform proofs. $\mathcal{L}'$ is the proof system that results from replacing the Identity, L$\multimap$, L$\Rightarrow$, L$\&$, and L$\forall$ rules in $\mathcal{L}$ with a single rule, called *backchaining*.

$$\frac{}{\Gamma; A \longrightarrow A}\ (\text{Identity}) \qquad \frac{\Gamma, B; \Delta, B \longrightarrow C}{\Gamma, B; \Delta \longrightarrow C}\ (\text{absorb})$$

$$\frac{}{\Gamma; \emptyset \longrightarrow 1}\ (\text{R}1) \qquad \frac{}{\Gamma; \Delta \longrightarrow \top}\ (\text{R}\top)$$

$$\frac{\Gamma; \Delta, B_i \longrightarrow C}{\Gamma; \Delta, B_1 \,\&\, B_2 \longrightarrow C}\ (\text{L}\,\&_i) \qquad \frac{\Gamma; \Delta \longrightarrow C_1 \quad \Gamma; \Delta \longrightarrow C_2}{\Gamma; \Delta \longrightarrow C_1 \,\&\, C_2}\ (\text{R}\,\&)$$

$$\frac{\Gamma; \Delta_1 \longrightarrow C_1 \quad \Gamma; \Delta_2, B \longrightarrow C_2}{\Gamma; \Delta_1, \Delta_2, C_1 \multimap B \longrightarrow C_2}\ (\text{L}\multimap) \qquad \frac{\Gamma; \Delta, B \longrightarrow C}{\Gamma; \Delta \longrightarrow B \multimap C}\ (\text{R}\multimap)$$

$$\frac{\Gamma; \emptyset \longrightarrow C_1 \quad \Gamma; \Delta, B \longrightarrow C_2}{\Gamma; \Delta, C_1 \multimap B \longrightarrow C_2}\ (\text{L}\Rightarrow) \qquad \frac{\Gamma, B; \Delta \longrightarrow C}{\Gamma; \Delta \longrightarrow B \Rightarrow C}\ (\text{R}\Rightarrow)$$

$$\frac{\Gamma; \Delta, B[t/x] \longrightarrow C}{\Gamma; \Delta, \forall x.B \longrightarrow C}\ (\text{L}\forall) \qquad \frac{\Gamma; \Delta \longrightarrow C[y/x]}{\Gamma; \Delta \longrightarrow \forall x.C}\ (\text{R}\forall)$$

provided that y is not free in the conclusion.

$$\frac{\Gamma; \emptyset \longrightarrow C}{\Gamma; \emptyset \longrightarrow\, !\, C}\ (\text{R}\,!) \qquad \frac{\Gamma; \Delta_1 \longrightarrow C_1 \quad \Gamma; \Delta_2 \longrightarrow C_2}{\Gamma; \Delta_1, \Delta_2 \longrightarrow C_1 \otimes C_2}\ (\text{R}\otimes)$$

$$\frac{\Gamma; \Delta \longrightarrow C[t/x]}{\Gamma; \Delta \longrightarrow \exists x.C}\ (\text{R}\exists) \qquad \frac{\Gamma; \Delta \longrightarrow C_i}{\Gamma; \Delta \longrightarrow C_1 \oplus C_2}\ (\text{R}\oplus_i)$$

Fig. 3. The proof system $\mathcal{L}$ for the Lolli language

In this paper, we will use a more restrictive definition for resource and goal formulas. Let A be atomic and $m \geq 1$:

$$R ::= S_1 \,\&\, \cdots \,\&\, S_m$$
$$S ::= \top \mid A \mid G \multimap A \mid \forall x.S$$
$$G ::= 1 \mid \top \mid A \mid G_1 \otimes G_2 \mid G_1 \,\&\, G_2 \mid G_1 \oplus G_2 \mid R \multimap G \mid S \Rightarrow G \mid\, !\, G \mid \forall x.G \mid \exists x.G$$

Here, S-formulas are called *resource clauses* in which A and G are called the *head* and the *body* respectively. S-formulas correspond to program clauses. Although this simplification does not change expressiveness of the language, it makes the presentation of *backchaining* simpler, as is discussed below.

Since full intuitionistic linear logic is not an abstract logic programming language, it is obvious that intuitionistic temporal linear logic is not as well. For example, in addition to the counterexamples in *ILL*, the sequents $\Box \bigcirc a \longrightarrow \bigcirc a$, $!\bigcirc a \longrightarrow \bigcirc a$, and $a \,\&\, \bigcirc a \longrightarrow \bigcirc a$ are all provable in *ITLL*, but they do not have uniform proofs.

Fig. 4 presents a proof system $\mathcal{TL}$ for the connectives $\top$, $\&$, $\multimap$, $\Rightarrow$, $\forall$, 1, $!$, $\otimes$, $\oplus$, $\exists$, $\bigcirc$, and $\Box$. Two rules, L$\Box$ and $\bigcirc$, are added in addition to those that arise in $\mathcal{L}$. This system has been designed to support the logic programming language TLLP over the following formulas: If A is atomic and $m \geq 1$,

$$R ::= S_1 \,\&\, \cdots \,\&\, S_m \mid \Box(S_1 \,\&\, \cdots \,\&\, S_m) \mid \bigcirc R$$
$$S ::= \top \mid A \mid G \multimap A \mid \forall x.S$$
$$G ::= 1 \mid \top \mid A \mid G_1 \otimes G_2 \mid G_1 \,\&\, G_2 \mid G_1 \oplus G_2 \mid R \multimap G \mid S \Rightarrow G \mid\, !\, G \mid \forall x.G \mid \exists x.G \mid \bigcirc G$$

(Rules of $\mathcal{L}$)

$$\frac{\Gamma; \Delta, B \longrightarrow C}{\Gamma; \Delta, \Box\, B \longrightarrow C}\ (\mathrm{L}\,\Box) \qquad \frac{\Gamma; \Box\, \Sigma, \Delta \longrightarrow C}{\Gamma; \Box\, \Sigma, \bigcirc\, \Delta \longrightarrow \bigcirc\, C}\ (\bigcirc)$$

Fig. 4. $\mathcal{TL}$: A proof system for the connectives $\top$, $\&$, $\multimap$, $\Rightarrow$, $\forall$, 1, !, $\otimes$, $\oplus$, $\exists$, $\bigcirc$, and $\Box$.

Let D be a $\&$-product of resource clauses $S_1 \,\&\, \cdots \,\&\, S_m$. Compared with Lolli, $\bigcirc^n D$ and $\bigcirc^n \Box\, D$ are added to resource formulas, and $\bigcirc\, G$ is added to goal formulas. The intuitive meaning of these formulas is as follows: $\bigcirc^n D$ means that the resource clause S_i $(1 \leq i \leq m)$ in D can be used exactly once at time n; $\bigcirc^n \Box\, D$ means that the resource clause S_i $(1 \leq i \leq m)$ in D can be used exactly once any time at and after time n; $\bigcirc\, G$ adjusts time one clock ahead and then executes G.

The proofs of propositions in this paper are based on Hodas and Miller's results in [7] for the Lolli language, and we will only give proof outlines.

Proposition 1. Let G be a goal formula, Γ a set of resource clauses, and Δ a multiset of resource formulas. Let D^* be the result of replacing all occurrences of $B \Rightarrow C$ in D with $(!\, B) \multimap C$, and let $\Gamma^* = \{B^* \mid B \in \Gamma\}$. Then the sequent $\Gamma; \Delta \longrightarrow G$ is provable in $\mathcal{TL}$ if and only if $!(\Gamma^*), \Delta^* \longrightarrow G^*$ is provable in *ITLL*.

Proof (sketch). The proof of this proposition can be shown by giving a simple conversion between proofs in the two systems. The cases of $\bigcirc$ and L$\Box$ are also immediate. □

Proposition 2. Let G be a goal formula, Γ a set of resource clauses, and Δ a multiset of resource formulas. Then the sequent $\Gamma; \Delta \longrightarrow G$ has a proof in $\mathcal{TL}$ if and only if it has a uniform proof in $\mathcal{TL}$.

Proof (sketch). The proof in the reverse direction is immediate, since a uniform proof in $\mathcal{TL}$ is a proof in $\mathcal{TL}$. The forward direction can be proved by showing that any proof in $\mathcal{TL}$ can be converted to a uniform proof of the same endsequent by permuting the rules to move occurrences of the left-rule up, though, and above instances of the right-rule. We explicitly show one case, that is when L$\Box$ occurs below R$\&$:

$$\dfrac{\dfrac{\begin{matrix}\Xi_1 \\ \Gamma; \Delta, B \longrightarrow C_1\end{matrix} \quad \begin{matrix}\Xi_2 \\ \Gamma; \Delta, B \longrightarrow C_2\end{matrix}}{\Gamma; \Delta, B \longrightarrow C_1 \,\&\, C_2}\ (\mathrm{R}\,\&)}{\Gamma; \Delta, \Box\, B \longrightarrow C_1 \,\&\, C_2}\ (\mathrm{L}\,\Box)$$

where Ξ_1 and Ξ_2 are uniform proofs of their endsequents respectively. The above proof structure can be converted to the following:

$$\dfrac{\dfrac{\begin{matrix}\Xi_1 \\ \Gamma; \Delta, B \longrightarrow C_1\end{matrix}}{\Gamma; \Delta, \Box\, B \longrightarrow C_1}\ (\mathrm{L}\,\Box) \quad \dfrac{\begin{matrix}\Xi_2 \\ \Gamma; \Delta, B \longrightarrow C_2\end{matrix}}{\Gamma; \Delta, \Box\, B \longrightarrow C_2}\ (\mathrm{L}\,\Box)}{\Gamma; \Delta, \Box\, B \longrightarrow C_1 \,\&\, C_2}\ (\mathrm{R}\,\&)$$

□

As with $\mathcal{L}$ and $\mathcal{L}'$, the left-hand rules can be restricted to a form of backchaining. Let us consider the following definition: Let R be a resource formula. $\|R\|$ is defined as a set of resource clauses (S-formulas):

1. if $R = A$ then $\|R\| = \{A\}$,
2. if $R = G \multimap A$ then $\|R\| = \{G \multimap A\}$,
3. if $R = \forall x.S$ then for all closed terms t, $\|R\| = \|S[t/x]\|$,
4. if $R = S_1 \,\&\, \cdots \,\&\, S_m$ then $\|R\| = \|S_1\| \cup \cdots \cup \|S_m\|$,
5. if $R = \Box\, R'$ then $\|R\| = \|R'\|$,
6. if $R = \bigcirc\, R'$ then $\|R\| = \emptyset$

Let $\mathcal{TL}'$ be a proof system that results from replacing the Identity, absorb, L$\multimap$, L$\Rightarrow$, L&, L$\forall$, and L$\Box$ rules in $\mathcal{TL}$ with the backchaining rules in Fig. 5. These backchaining rules (especially the definition of $\|\cdot\|$) are simpler than the original rule for Lolli because of the restrictive definition of resource formulas. It is noticed that the absorb rule is integrated into (BC!$_1$) and (BC!$_2$).

Proposition 3. Let G be a goal formula, Γ a set of resource clauses, and Δ a multiset of resource formulas. Then the sequent $\Gamma; \Delta \longrightarrow G$ has a proof in $\mathcal{TL}$ if and only if it has a proof in $\mathcal{TL}'$.

Since uniform proofs are complete for $\mathcal{TL}$, this proposition can be proved by showing that there is a uniform proof in $\mathcal{TL}$ if and only if there is a proof in $\mathcal{TL}'$. We do not present the proof here. A similar proof has been given by Hodas and Miller in [7] for the Lolli language.

$$\frac{}{\Gamma; D \longrightarrow A}\ (\mathrm{BC}_1) \qquad \frac{}{\Gamma, D; \emptyset \longrightarrow A}\ (\mathrm{BC!}_1)$$

provided, in each case, A is atomic and $A \in \|D\|$.

$$\frac{\Gamma; \Delta \longrightarrow G}{\Gamma; \Delta, D \longrightarrow A}\ (\mathrm{BC}_2) \qquad \frac{\Gamma, D; \Delta \longrightarrow G}{\Gamma, D; \Delta \longrightarrow A}\ (\mathrm{BC!}_2)$$

provided, in each case, A is atomic and $G \multimap A \in \|D\|$.

Fig. 5. Backchaining for the proof system $\mathcal{TL}'$

3.1 TLLP Example Programs

We will present simple TLLP examples here. For the syntax, we use '`:-`' for the inverse of $\multimap$, '`,`' for $\otimes$, '`-<>`' for $\multimap$, '`=>`' for $\Rightarrow$, '`@`' for $\bigcirc$, and '`#`' for $\Box$.

We first consider a Lolli program that finds a Hamilton path through the complete graph of four vertices. Since each vertex is represented as a linear resource, the constraints such that each vertex must be used exactly can be expressed.

```
p(V,V,[V]) :- v(V).
p(U,V,[U|P]) :- v(U), e(U,W), p(W,V,P).
e(U,V).
goal(P) :- v(a) -<> v(b) -<> v(c) -<> v(d) -<> p(a,d,P).
```

When the goal `goal(P)` is executed, the vertices are added as resources, and the goal `p(a,d,P)` will search a path from `a` to `d` by consuming each vertex exactly once.

In addition to the resource-sensitive features of Lolli, TLLP can describe the time-dependent properties of resources, in particular, the precise order of the moments when some resources are consumed. For example, `#v(a)` denotes the vertex a that can be used exactly once at and after present. `@ #v(c)` denotes the vertex c that can be used exactly once at and after the next moment in time. So, the following TLLP program finds a Hamilton path that satisfies such constraints. It is noticed that time is adjusted one clock ahead every time the path crosses an arc.

```
p(V,V,[V]) :- v(V).
p(U,V,[U|P]) :- v(U), e(U,W), @p(W,V,P).
e(U,V).
goal(P) :- #v(a) -<> @ @v(b) -<> @ #v(c) -<> #v(d) -<> p(a,d,P).
```

Our next example is a simple Timed Petri Nets reachability emulator. This program checks the reachability of a Timed Petri Net in Fig. 6 from the initial marking (one token in p) to the final marking (one token in p and two tokens in r). Each d_i, a non-negative integer, is the delay time for the transition t_i.

```
tpn :- #p -<> (goal :- p, r, r) => tpn(1, 100).
tpn(Dep, Lim) :- Dep =< Lim, fire(Dep).
tpn(Dep, Lim) :- Dep =< Lim, Dep1 is Dep + 1, tpn(Dep1, Lim).
next(D) :- D1 is D - 1, D1 > 0, fire(D1).
fire(D) :- goal.
fire(D) :- p, @ #p -<> @ #q -<> next(D).
fire(D) :- q, q, q, @ #r -<> next(D).
fire(D) :- @next(D).
```

Since the proof search of TLLP is depth-first and is not complete, we use a *iterative deepening* search, a combination of depth-fisrt and breadth-first search. First, the predicate `tpn(Dep, Lim)` checks the reachability at depth 1, and then it increases the depth by one if the check fails.

Besides the examples presented above, the latest TLLP package includes programs for the flow-shop scheduling problem and Conway's Game of Life.

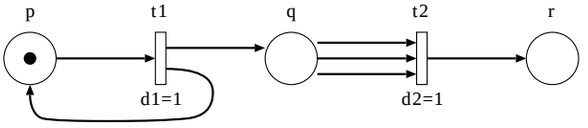


Fig. 6. An example of Timed Petri Nets

4 Resource Management Model

The resource management during a proof search in $\mathcal{TL}'$ is a serious problem for the implementor. Let us consider, for example, the execution of the goal $G_1 \otimes G_2$:

$$\frac{\Gamma; \Delta_1 \longrightarrow G_1 \quad \Gamma; \Delta_2 \longrightarrow G_2}{\Gamma; \underbrace{\Delta_1, \Delta_2}_{\Delta} \longrightarrow G_1 \otimes G_2} \ \mathrm{R}\otimes$$

When the system applies this rule during bottom-up search, the linear context Δ must be divided into Δ_1 and Δ_2. If Δ contains n resource formulas, all 2^n possibilities might need to be tested to find a desirable partition.

For Lolli, Hodas and Miller solved this problem by splitting resources lazily, and they proposed a new execution model called the I/O model [8].

In this model, the sequent $I\,\{G\}\,O$ means that the goal G can be executed given the *input context* I so that the *output context* O remains. The input and output context, together called *IO*-context, are lists of resource formulas, !-marked resource formulas, or the special symbol 1 that denotes a place where a resource formula has been consumed. In the the execution of the goal $G_1 \otimes G_2$:

$$\frac{I\,\{G_1\}\,M \quad M\,\{G_2\}\,O}{I\,\{G_1 \otimes G_2\}\,O} \ (\otimes)$$

First, $I\,\{G_1\}\,M$ tries to execute G_1 given the input context I. If this succeeds, the output context M is forwarded to G_2, and then $M\,\{G_2\}\,O$ is attempted. If this second attempt fails, $I\,\{G_1\}\,M$ retries to find a different, more desirable consumption pattern.

We will extend the I/O model for the TLLP language. The additional problem here is that the bottom-up application of the rule for $\bigcirc$ in $\mathcal{TL}'$ requires manipulating large dynamic data structures.

$$\frac{\Gamma; \Box\,\Sigma, \Delta \longrightarrow G}{\Gamma; \Box\,\Sigma, \bigcirc\,\Delta \longrightarrow \bigcirc\,G} \ (\bigcirc)$$

For example, when the system executes the goal $\bigcirc\,G$ given input context $I = [p, \bigcirc\,q, \bigcirc\,\bigcirc\,r, !\,s]$, we need to reconstruct and create a new input context $I' = [1, q, \bigcirc\,r, !\,s]$ before the execution of the goal G.

We introduce a *time index* to solve this problem. Fig. 7 presents an extension of the I/O model for the TLLP language, called *IOT*. *IOT* makes use of a time

$$\frac{}{I\,\{1\}_T\,I}\ (1) \qquad \frac{\mathit{subcontext}_T(O,I)}{I\,\{\top\}_T\,O}\ (\top)$$

$$\frac{I\,\{G_1\}_T\,M \quad M\,\{G_2\}_T\,O}{I\,\{G_1 \otimes G_2\}_T\,O}\ (\otimes) \qquad \frac{I\,\{G_1\}_T\,O \quad I\,\{G_2\}_T\,O}{I\,\{G_1 \,\&\, G_2\}_T\,O}\ (\&)$$

$$\frac{I\,\{G_i\}_T\,O}{I\,\{G_1 \oplus G_2\}_T\,O}\ (\oplus) \qquad \frac{[\langle !\,S, 0\rangle \mid I]\,\{G\}_T\,[\langle !\,S, 0\rangle \mid O]}{I\,\{S \Rightarrow G\}_T\,O}\ (\Rightarrow)$$

$$\frac{[\langle R, T+n\rangle \mid I]\,\{G\}_T\,[1 \mid O]}{I\,\{\bigcirc^n R \multimap G\}_T\,O}\ (\multimap)$$

provided that R is a formula of the form: $S_1 \,\&\, \cdots \,\&\, S_m$ or $\Box(S_1 \,\&\, \cdots \,\&\, S_m)$.

$$\frac{I\,\{G\}_T\,I}{I\,\{!\,G\}_T\,I}\ (!) \qquad \frac{I\,\{G\}_{T+1}\,O}{I\,\{\bigcirc\,G\}_T\,O}\ (\bigcirc)$$

$$\frac{\mathit{pick}_T(I, O, A)}{I\,\{A\}_T\,O}\ (\mathrm{BC}_1) \qquad \frac{\mathit{pick}_T(I, M, G \multimap A) \quad M\,\{G\}_T\,O}{I\,\{A\}_T\,O}\ (\mathrm{BC}_2)$$

Fig. 7. *IOT*: An I/O model for propositional TLLP

index T. The sequent is of the form $I\,\{G\}_T\,O$. T, non-negative integer, is the *current time*. At a given point in the proof, only resources that can be used at that time may be used. T is also used to set a *consumption time* of newly added resources.

Each element in *IOT*-context is a pair $\langle R, t\rangle$ where R is a resource formula or !-marked resource formula, and t is its consumption time, or the special symbol 1. Linear resources have the form $\langle S_1 \,\&\, \ldots \,\&\, S_m, t\rangle$ or $\langle \Box(S_1 \,\&\, \ldots \,\&\, S_m), t\rangle$, where t is its consumption time calculated from the value of T, and its multiplicity of $\bigcirc$. Intuitionistic resources have the form $\langle !\,S, 0\rangle$, where S is a resource clause. For example, the consumable resources at time T have the following forms in the context: $\langle S_1 \,\&\, \ldots \,\&\, S_m, T\rangle$, $\langle \Box(S_1 \,\&\, \ldots \,\&\, S_m), t\rangle$ where $t \leq T$, and $\langle !\,S, 0\rangle$

The relation $\mathit{pick}_T(I, O, S)$ holds if S occurs in the context I and is consumable at time T, and O results from replacing that occurrence of S in I with 1. The relation also holds if $!S$ occurs in I, and I and O are equal. The relation $\mathit{subcontext}_T(O, I)$ holds if O arises from replacing arbitrarily many (including 0) non-!-marked elements of I that are consumable any time at and after time T with 1.

To prove that *IOT* is logically equivalent to $\mathcal{TL}'$, we need to define the notion of difference $I -_T O$ for two *IOT*-context I and O that satisfy the relation $\mathit{subcontext}_T(O, I)$. $I -_T O$ is a pair $\langle \Gamma, \Delta\rangle$, where Γ is a set of all formulas S such that $\langle !\,S, 0\rangle$ is an element of I (and O), and Δ is a multiset of all formulas $\bigcirc^{\max(0,t-T)} R$ such that $\langle R, t\rangle$ occur in I (If R is of the form $S_1 \,\&\, \cdots \,\&\, S_m$, then $t \geq T$. If R is of the form $\Box(S_1 \,\&\, \cdots \,\&\, S_m)$, then t is arbitrary), and the corresponding place in O is the symbol 1.

Proposition 4. Let T be a non-negative integer. Let I and O be IOT-contexts that satisfy $subcontext_T(O, I)$. Let $I -_T O$ be the pair $\langle \Gamma, \Delta \rangle$ and let G be a goal formula. $I\,\{G\}_T\,O$ is provable in IOT if and only if $\Gamma; \Delta \longrightarrow G$ is provable in $\mathcal{TL}'$.

Proof (sketch). This proposition, in both directions, can be proved by induction on proof structure. □

5 Level-Based Resource Management Model

The I/O model provides an efficient computation model for proof search. The I/O model has been refined several times. Cervesato et. al recently have proposed a refinement designed to eliminate the non-determinism in management of linear context involving & and $\top$ [3]. However, the I/O model and its refinements still require copying and scanning large dynamic data structures to control the consumption of linear resources. Thus, they are more suited to develop interpreters in high-level languages rather than compilers.

We point out two problems here. First, during the execution of $I\,\{G\}\,O$ (especially *pickR*), the context O is reconstructed from the context I by replacing linear consumed resources with 1. This will slow down the execution speed. Secondly, let us consider the execution of the goal $G_1 \,\&\, G_2$:

$$\frac{I\,\{G_1\}\,O \quad I\,\{G_2\}\,O}{I\,\{G_1 \,\&\, G_2\}\,O}\ (\&)$$

This rule means that the goal G_1 and G_2 must use the same resources. In a naive implementation, the system first copies the input context and executes the two conjuncts separately, and then it compares their output contexts. This leads to unnecessary backtracking.

To solve these problems, Tamura et. al have introduced a refinement of the I/O model with *level indices* [10][15], called the *IOL* model [1]. Hodas et. al recently proposed the refinement of *IOL* for the complete treatment of $\top$ in [9].

IOL makes use of two level indices L and U to manage the consumption of resources. The sequent is of the form $\vdash_{L,U} I\,\{G\}\,O$. L, a positive integer, is the *current consumption level*. At a given point in the proof, only linear resources labeled with that consumption level (and intuitionistic resources labeled with 0) can be used. U, a negative integer, is the *current consumption maker*. When a linear resource is consumed, its consumption level is changed to the value of U.

Each element in *IOL*-context is a pair $\langle R, \ell \rangle$, where R is a resource formula, and ℓ is its consumption level. Linear resources have the form $\langle R, \ell \rangle$, where ℓ is the value of L at which the resource can be consumed. Intuitionistic resources have the form $\langle S, 0 \rangle$ where S is a resource clause.

$$\frac{\vdash_{L,U-1} I\,\{G_1\}\,M \quad change_{U-1,L+1}(M, N) \quad \vdash_{L+1,U} N\,\{G_2\}\,O \quad thinable_{L+1}(O)}{\vdash_{L,U} I\,\{G_1 \,\&\, G_2\}\,O}\ (\&)$$

[1] In this paper, we use the notation in [10] to explain the *IOL* model.

$$\frac{}{\vdash^T_{L,U} I\,\{1\}\,I}\ (1) \qquad \frac{\mathit{subcontext}^T_{U,L}(O,I)}{\vdash^T_{L,U} I\,\{\top\}\,O}\ (\top)$$

$$\frac{\vdash^T_{L,U} I\,\{G_1\}\,M \quad \vdash^T_{L,U} M\,\{G_2\}\,O}{\vdash^T_{L,U} I\,\{G_1 \otimes G_2\}\,O}\ (\otimes)$$

$$\frac{\vdash^T_{L,U-1} I\,\{G_1\}\,M \quad \mathit{change}_{U-1,L+1}(M,N) \quad \vdash^T_{L+1,U} N\,\{G_2\}\,O \quad \mathit{thinable}_{L+1}(O)}{\vdash^T_{L,U} I\,\{G_1 \,\&\, G_2\}\,O}\ (\&)$$

$$\frac{\vdash^T_{L,U} I\,\{G_i\}\,O}{\vdash^T_{L,U} I\,\{G_1 \oplus G_2\}\,O}\ (\oplus_i) \qquad \frac{\vdash^T_{L,U} [\langle S,0,0\rangle \mid I]\,\{G\}\,[\langle S,0,0\rangle \mid O]}{\vdash^T_{L,U} I\,\{S \Rightarrow G\}\,O}\ (\Rightarrow)$$

$$\frac{\vdash^T_{L,U} [\langle R,T+n,L\rangle \mid I]\,\{G\}\,[\langle R,T+n,U\rangle \mid O]}{\vdash^T_{L,U} I\,\{\bigcirc^n R \multimap G\}\,O}\ (\multimap)$$

provided that R is a formula of the form: $S_1 \,\&\, \cdots \,\&\, S_m$ or $\Box(S_1 \,\&\, \cdots \,\&\, S_m)$.

$$\frac{\vdash^T_{L+1,U} I\,\{G\}\,O}{\vdash^T_{L,U} I\,\{!\,G\}\,O}\ (!) \qquad \frac{\vdash^{T+1}_{L,U} I\,\{G\}\,O}{\vdash^T_{L,U} I\,\{\bigcirc G\}\,O}\ (\bigcirc)$$

$$\frac{\mathit{pick}^T_{L,U}(I,O,A)}{\vdash^T_{L,U} I\,\{A\}\,O}\ (\mathrm{BC}_1) \qquad \frac{\mathit{pick}^T_{L,U}(I,M,G \multimap A) \quad \vdash^T_{L,U} M\,\{G\}\,O}{\vdash^T_{L,U} I\,\{A\}\,O}\ (\mathrm{BC}_2)$$

Fig. 8. *IOTL*: A level-based I/O model for propositional TLLP

For example, the outline of the execution of the goal $G_1 \,\&\, G_2$ is as follows:

1. $\vdash_{L,U-1} I\,\{G_1\}\,M$ Decrement U so that we know which resources are consumed during the execution of G_1, and then execute G_1.
2. $\mathit{change}_{U-1,L+1}(M,N)$ Change the level of resources that have been consumed in G_1 to $L+1$.
3. $\vdash_{L+1,U} N\,\{G_2\}\,O$ Increment L and U, and then execute G_2.
4. $\mathit{thinable}_{L+1}(O)$ Check whether none of resources in O have $L+1$ as their consumption level.

IOL is logically equivalent to $\mathcal{L}'$. In *IOL*, all resources are kept in a single table, called *resource table*, during execution. The consumption of resources can be achieved easily by changing their consumption level destructively. The idea of this model has already been used as a basis for a compiler system for a useful fragment of first-order Lolli, in which the resource table is implemented as an array, and the speed access to resources is achieved by using a hash table.

For TLLP, we give a refinement of *IOT*, called *IOTL* in Fig. 8, with level indices of *IOL*. The sequent of *IOTL* is of the form $\vdash^T_{L,U} I\,\{G\}\,O$, where T is the current time, L is the current consumption level, and U is the current consumption maker.

Each element in *IOTL*-contexts is a tuple $\langle R,t,\ell\rangle$, where R is a resource formula, t is its consumption time, and ℓ is its consumption level. Linear resources have the form $\langle S_1 \,\&\, \ldots \,\&\, S_m,t,\ell\rangle$ or $\langle \Box(S_1 \,\&\, \ldots \,\&\, S_m),t,\ell\rangle$, where t is calculated

from the value of T and its multiplicity of $\bigcirc$, and ℓ is the value of L at which the resource can be consumed. Intuitionistic resources have the form $\langle S, 0, 0\rangle$, where S is a resource clause.

When the system executes $\vdash^T_{L,U} I\,\{G\}\,O$, the consumable resources in the context I have the following forms: $\langle S_1 \,\&\, \ldots \,\&\, S_m, T, L\rangle$, $\langle \Box(S_1 \,\&\, \ldots \,\&\, S_m), t, L\rangle$ where $t \leq T$, and $\langle S, 0, 0\rangle$.

The relation $pick^T_{L,U}(I, M, S)$ selects a consumable resource clause S from the input context I. The output context M is the same as I, except that the consumption level of the selected clause is changed to the value of U if it is a linear resource. The relation $change_{\ell,\ell'}(M, N)$ modifies the context M so that any resources in M with level ℓ have their level changed to ℓ' in the context N. The relation $thinable_\ell(O)$ checks whether none of resources in O have ℓ as their consumption level. The relation $subcontext^T_{U,L}(O, I)$ then consumes some resources. The output context O is the same as I, except that the consumption levels of some resources are changed to the value of U, if they are linear resources.

6 Implementation Design

In this section, we discuss implementation issues for the TLLP language.

For Lolli, Hodas has developed the I/O model-based interpreters both in Prolog and SML. Tamura et. al have designed an extension of standard WAM model (called LLPAM) and have developed a compiler system [2]. This compiler system supports the first-order Lolli language, except for the goal $\forall x.G$. LLPAM was first designed, based on *IOL* [10][15], and recently refined with the *top flag* in $\mathcal{LRM}$ [9], for the complete treatment of $\top$. LLPAM has also been improved to incorporate the resources compilation technique in [2].

The main differences between LLPAM and WAM is as follows:

- Three new date areas are added: `RES` (the resource table), `SYMBOL` (the symbol table), and `HASH` (the hash table). `HASH` and `SYMBOL` are used for speed access to resources, and the entries in `RES` are hashed on the predicate symbol and the value of the first argument in the current implementation.
- Six new registers are added: `R`, `L`, `U`, `T`, `R1`, and `R2`. `R` is the current top of `RES`. `L` and `U` are the current values of L (current consumption level) and U (current consumption maker) in *IOL* respectively. `T` is the *top flag* in $\mathcal{LRM}$. `R1` and `R2` are used for picking up consumable resources quickly.
- New instructions for newly added connectives are added.

For TLLP, there seem to be at least three approaches to develop efficient implementations: TLLP interpreter, translator from TLLP to Lolli, and TLLP compiler. First, it is easy to implement a TLLP interpreter based on *IOT* and *IOTL* in high-level languages like Prolog, but the resulting systems are slow. Secondly, it is possible to translate TLLP programs into Lolli programs by adding

[2] The latest package (version 0.5) including all source code can be obtained from `http://bach.seg.kobe-u.ac.jp/llp/`.

a new argument for the current time T in IOT to each predicate in Lolli. The drawback of this simple translation is that the goal $\top$ in TLLP can not be correctly translated into that in Lolli. Finally, it is also possible to extended LLPAM to support the TLLP language. We summarize important points that have been improved:

- Two new fields `time` and `box` have been added to each entry in `RES`. The `time` field denotes the consumption time in $IOTL$. The `box` flag is set to false if the newly added resource is not prefixed by $\Box$, otherwise true.
- A new register `TI` has been added. `TI` denotes the current time T in $IOTL$. Choice instructions such as *try* in WAM therefore need to set and restore the value of `TI`. `TI` is used to set the `time` field of newly added resource. `TI` is also used for hash key for speed access to the resources.
- In LLPAM, the instruction `add_res` $\mathtt{A}_i$, $\mathtt{A}_j$ is used to add linear resource clauses, where $\mathtt{A}_i$ is its head, $\mathtt{A}_j$ is its *closure* that consists of the compiled code and a set of bindings for free variables. We replaced this instruction with two new instructions `add_exact_timed_res` $\mathtt{A}_i$, $\mathtt{A}_j$, n and `add_timed_res` $\mathtt{A}_i$, $\mathtt{A}_j$, n. The former is used to add a resource clause S_i $(1 \leq i \leq m)$ in $\bigcirc^n(S_1 \,\&\, \cdots \,\&\, S_m)$, where $\mathtt{A}_i$ is its head, $\mathtt{A}_j$ is its closure, and n is the multiplicity of $\bigcirc$. The latter is used to add a resource clause S_i $(1 \leq i \leq m)$ in $\bigcirc^n \Box(S_1 \,\&\, \cdots \,\&\, S_m)$, $\mathtt{A}_i$ is its head, $\mathtt{A}_j$ is its closure, and n is the multiplicity of $\bigcirc$.
- In LLPAM, the instruction `pickup_resource` p/n, $\mathtt{A}_i$, L finds a consumable resource with predicate symbol p/n by checking its consumption level, and then it sets its index value to $\mathtt{A}_i$. If there are no consumable resources, it jumps to L. We need to improve this instruction so that it checks not only the level condition but also the time condition by comparing the consumption time (the `time` field) of resources with the current time (the current value of `TI`).

The specification of LLPAM have been shown in the papers [9] and [2].

7 Conclusion and Future Work

Recent development of logic programming languages based on linear logic suggests a successful direction to extend logic programming to be more expressive and more efficient. In this paper, we have designed the logic programming language TLLP based on intuitionistic temporal linear logic and have discussed some implementation issues for TLLP. The following points are still remaining:

- TLLP supports a small fragment of intuitionistic temporal linear logic.
- $IOTL$ needs to be refined with the idea of top flag in $\mathcal{LRM}$ [9] for the complete treatment of $\top$.
- The goal $\forall x.G$ is not supported in the current implementation.

Currently, we have developed a prototype of TLLP compiler system based on the extension presented in this paper. The latest TLLP package (version 0.1) including TLLP interpreters, a translator from TLLP into Lolli, and a prototype compiler is available from `http://kaminari.scitec.kobe-u.ac.jp/tllp/`.

References

1. Jean-Marc Andreoli and Remo Pareschi. Linear objects: Logical processes with built-in inheritance. *New Generation Computing*, 9:445–473, 1991.
2. Mutsunori Banbara and Naoyuki Tamura. Compiling Resources in a Linear Logic Programming Language. In Konstantinos Sagonas, editor, *Proceedings of the JICSLP'98 Post Conference Workshop 7 on Implementation Technologies for Programming Languages based on Logic*, pages 32–45, June 1998.
3. Iliano Cervesato, Joshua S. Hodas, and Frank Pfenning. Efficient resource management for linear logic proof search. In R. Dyckhoff, H. Herre, and P. Schroeder-Heister, editors, *Proceedings of the Fifth International Workshop on Extensions of Logic Programming — ELP'96*, pages 67–81, Leipzig, Germany, 28–30 March 1996. Springer-Verlag LNAI 1050.
4. Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.
5. James Harland and David Pym. A uniform proof-theoretic investigation of linear logic programming. *Journal of Logic and Computation*, 4(2):175–207, April 1994.
6. Takaharu Hirai. An Application of Temporal Linear Logic to Timed Petri Nets. In *Proceedings of the Petri Nets'99 Workshop on Applications of Petri Nets to Intelligent System Development*, pages 2–13, June 1999.
7. Joshua S. Hodas. *Logic Programming in Intuitionistic Linear Logic: Theory, Design and Implementation*. PhD thesis, University of Pennsylvania, Department of Computer and Information Science, 1994.
8. Joshua S. Hodas and Dale Miller. Logic programming in a fragment of intuitionistic linear logic. *Information and Computation*, 110(2):327–365, 1994. Extended abstract in the Proceedings of the Sixth Annual Symposium on Logic in Computer Science, Amsterdam, July 15–18, 1991.
9. Joshua S. Hodas, Kevin Watkins, Naoyuki Tamura, and Kyoung-Sun Kang. Efficient Implementation of a Linear Logic Programming Language. In Joxan Jaffar, editor, *Proceedings of the 1998 Joint International Conference and Symposium on Logic Programming*, pages 145–159. The MIT Press, June 1998.
10. Kyoung-Sun Kang, Mutsunori Banbara, and Naoyuki Tamura. Efficient resource management model for linear logic programming languages. *Computer Software*, 18(0):138–154, 2001. (in Japanese).
11. Max I. Kanovich and Takayasu Ito. Temporal linear logic specifications for concurrent processes (extended abstract). In *Proceedings of 12th Annual IEEE Symposium on Logic in Computer Science (LICS'97)*, pages 48–57, 1997.
12. Naoki Kobayashi and Akinori Yonezawa. ACL — A concurrent linear logic programming paradigm. In D. Miller, editor, *Proceedings of the 1993 International Logic Programming Symposium*, pages 279–294, Vancouver, Canada, October 1993. MIT Press.
13. Dale Miller. A multiple-conclusion specification logic. *Theoretical Computer Science*, 165(1):201–232, 1996.
14. Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

15. Naoyuki Tamura and Yukio Kaneda. Extension of WAM for a linear logic programming language. In T. Ida, A. Ohori, and M. Takeichi, editors, *Second Fuji International Workshop on Functional and Logic Programming*, pages 33–50. World Scientific, Nov. 1996.
16. Makoto Tanabe. Timed petri nets and temporal linear logic. In *Lecture Notes in Computer Science 1248: Proceedings of Application and Theory of Petri Nets*, pages 156–174, June 1997.

A Computational Model for Functional Logic Deductive Databases

Jesús M. Almendros-Jiménez[1,⋆], Antonio Becerra-Terón[1], and Jaime Sánchez-Hernández[2,⋆]

Dpto. de Lenguajes y Computación.
Universidad de Almería
Dpto. de Sistemas Informáticos y Programación.
Universidad Complutense de Madrid
{jalmen,abecerra}@ual.es, jaime@sip.ucm.es

Abstract. This paper adds the handling of negative information to a functional-logic deductive database language. By adopting as semantics for negation the so-called *CRWLF*, wherein the negation is intended as 'finite failure' of reduction, we will define Herbrand algebras and models for this semantics and a fix point operator to be used in a new goal-directed bottom-up evaluation mechanism based on magic transformations. This bottom-up evaluation will simulate the top-down one of the original program; in fact, it will carry out a goal-directed lazy evaluation.

1 Introduction

Deductive databases [17] are database management systems whose query language and, usually, storage structure are designed around a logical data model. They offer a rich query language which extends the relational model in many directions (for instance, support for non-first normal form and recursive relations) and they are suited for application in which a large number of data must be accessed and complex queries must be supported.

With respect to the operational semantics, most deductive database systems (for instance, DATALOG [19], CORAL [16], ADITI [20]) use *bottom-up* evaluation instead of *top-down* one like Prolog systems. The reason is that the bottom-up approach allows to use a *set-at-a-time* evaluation, i.e. it processes sets of goals, rather than proceeding one (sub) goal at a time, where operations like relational joins can be made for disk-resident data efficiently. Therefore, when the program is data-intensive, this evaluation method is potentially much more efficient than top-down techniques. The idea of the *goal-directed bottom-up evaluation* is to generate by using the *fix point operator* [3] the subset of the *Herbrand model* of the program relevant to the query solving. With this aim, the bottom-up evaluation in such languages involves a query-program transformation termed *Magic Sets* [5], in such a way that a logic program-query is transformed into a *magic*

⋆ The authors have been partially supported by the Spanish CICYT (project TIC 98-0445-C03-02 TREND)

P. Codognet (Ed.): ICLP 2001, LNCS 2237, pp. 331–347, 2001.

logic program-query whose bottom-up evaluation is devised to simulate the top-down one of the original program and query. The program is evaluated until no new facts are generated or the answer to the query is found. The transformed program adds new predicates, called *magic predicates*, whose role is to pass information (instantiated and partially instantiated arguments in the predicates of the query) to the program in order to consider only those instances of the program rules relevant to the query solving. Several transformation methods have been studied in the past, for instance, *Generalized Magic Sets* [5], *Generalized Supplementary Magic Sets* [5] and *Magic Templates* [15].

The use of negation in deductive databases allows to increase their expressive power as query languages. The introduction of negation in logic programming (see [4] for a survey), and thus the study of semantic models of the so-called *general logic programs*, have been widely studied in the past. Most relevant semantics for handling of negation are the *stable model semantics* [6] and the *well-founded semantics* [21].

However, the incorporation of negation in the deductive languages implies new problems in the magic transformation. In fact, new magic transformations have been proposed pointing out the one presented for *modularly stratified programs* [18] and the *doubled program* approach [10] by adopting both the well-founded semantics. The problem arises from the three valued nature of the magic predicates that result, and the well-founded model of the transformed magic program and the original one may disagree [10]. In [10] classes of *sideways information-passing strategies*, also called *sip*s, which ensure that the magic sets are two-valued, are defined. These sips, named *well-founded sips*, make sure that the well-founded model of a program is preserved w.r.t. the query in the transformed program. Moreover, they subsume the *left-to-right sips* intended for modularly stratified programs [18]. Finally, they present a new magic transformation by using a doubled program technique which preserves the well-founded model w.r.t. the query regardless of the sips to be used. The drawback of this approach is that the bottom-up evaluation of the program must end.

On the other hand, the integration of functional and logic programming has been widely investigated during the last years. It has led to the recent design of modern programming languages such as CURRY [9] and $\mathcal{TOY}$ [13]. The basic ideas in functional-logic programming consist in *lazy narrowing* as operational mechanism, following some class of *narrowing strategy* combined with some kind of *constraint solving* and *higher order* features, see [8] for a survey.

In [1], a framework for goal-directed bottom-up evaluation for functional-logic programs without negation has been proposed. As in the logic paradigm, the bottom-up evaluation is based on a magic transformation for a given program-query into a magic program-query. In the cited paper, the semantics adopted for the programs is the *C*onstructor *B*ased *Re W*riting *L*ogic (*CRWL*) presented in [7]. This bottom-up evaluation is based on the use of a fix point operator over *CRWL*-Herbrand algebras and it simulates the *demand driven strategy* [11] for top-down evaluation of *CRWL*-programs.

Recently, a framework called *C*onstructor *B*ased *Re*Writing *L*ogic with *F*ailure (*CRWLF*) has been presented in [14], extending the semantics *CRWL* in order to handle negative information and wherein the negation is intended as 'finite failure' of reduction. The formulas that are provable in *CRWL* can also be proved in *CRWLF* but, in addition, *CRWLF* provides 'proofs of unprovability' within *CRWL*. *CRWLF* can only give an approximation to failure in *CRWL* that corresponds to the cases in which unprovability refers to 'finite failure' of reduction.

The aim of this paper is to add the handling of negative information to a functional-logic language, and to present a goal-directed bottom-up evaluation mechanism in order to get a computational model for a "functional-logic" deductive language. With this aim, we will replace the semantics *CRWL* used in [1], by the semantics *CRWLF* and we will present Herbrand algebras and models and a fix point operator which computes the Herbrand model of a *CRWLF*-program. Finally, we will propose an extension of our goal-directed bottom-up evaluation, based on a new magic transformation and the use of the defined fix point operator.

As an example, a "functional-logic" deductive database can handle a boss hierarchical line as follows:

(**1**) $\texttt{boss(jesus)} \rightarrow \texttt{jaime}$. (**3**) $\texttt{member(X,[])} \rightarrow \texttt{false}$.

(**2**) $\texttt{boss(jaime)} \rightarrow \texttt{antonio}$. (**4**) $\texttt{member(X,[Y|L])} \rightarrow \texttt{member(X,L)} \Leftarrow \texttt{X} \not\bowtie \texttt{Y}$.

(**5**) $\texttt{member(X,[Y|L])} \rightarrow \texttt{true} \Leftarrow \texttt{X} \bowtie \texttt{Y}$.

(**6**) $\texttt{superboss(P)} \rightarrow \texttt{[boss(P)|superboss(boss(P))]}$.

Goal : $-\texttt{member(jaime, superboss(jesus))} \bowtie \texttt{true}$.

where $\bowtie$ and $\not\bowtie$ refer to a joinability constraint (both sides reduce to the same constructor term) and its logical negation, respectively. Our evaluation method will evaluate the function `superboss`, which defines a possibly infinite data, as far as needed in order to solve the goal. It starts with `superboss(jesus)` as $\bot$, which represents the undefined value, and then `superboss(jesus)` is evaluated up to $[\texttt{jaime}|\bot]$, necessary for the goal solving. Moreover, in our framework, we have to solve lazily the negative constraints. For instance, supposing the query $\texttt{superboss(X)} \not\bowtie \texttt{superboss(Y)}$ w.r.t. the above program and starting with `superboss(X)` and `superboss(Y)` as $\bot$, the evaluation can bind the variables `X` to `jesus` and `Y` to `jaime` evaluating `superboss(jesus)` up to $[\texttt{jaime}|\bot]$ and `superboss(jaime)` up to $[\texttt{antonio}|\bot]$ where $[\texttt{jaime}|\bot]$ conflicts with $[\texttt{antonio}|\bot]$ and therefore obtaining the answer $\texttt{X} = \texttt{jesus}, \texttt{Y} = \texttt{jaime}$.

As theoretic results of this paper, we will establish the soundness and completeness results of our bottom-up evaluation w.r.t. Herbrand models and the proof-semantics *CRWLF*. Moreover, we will establish correspondences among proofs of a given goal in the cited logic and the "facts" computed by means of the bottom-up evaluation showing the optimality of our method.

The rest of the paper will be organized as follows. In section 2, we will introduce *CRWLF*; section 3 will define the fix point operator and the Herbrand models; section 4 will present the magic transformation; section 5 will establish soundness, completeness and optimality results and, finally, section 6 will describe the conclusions and future work. Due to the lack of space, the full proofs of our results can be found in [2].

2 The *CRWLF* Framework

In this section we summarize the *C*onstructor *R*e*W*riting *L*ogic with *F*ailure presented in [14]. Assuming a signature $\Sigma = DC \cup FS$ where $DC = \bigcup_{n \in \mathbb{N}} DC^n$ is a set of *constructor* symbols $c, d, ...$ and $FS = \bigcup_{n \in \mathbb{N}} FS^n$ is a set of *function* symbols $f, g, ...$, all of them with associated arity and such that $DC \cap FS = \emptyset$, and also a countable set $\mathcal{V}$ of *variable* symbols X,Y,... we write *Term* for the set of (total) *terms* e,e',... (also called *expressions*) built up with Σ and $\mathcal{V}$ in the usual way, and we distinguish the subset *CTerm* of (total) constructor terms or (total) *c-terms* $s, t, ...$, built up only with DC and $\mathcal{V}$. Terms intend to represent possibly reducible expressions, whereas c-terms represent data values, not further reducible. We extend the signature Σ by adding two new constants: $\bot$ that plays the role of undefined value and F that will be used as an explicit representation of failure of reduction. The set $Term_\bot$ of *partial* terms and the set $CTerm_\bot$ of *partial* c-terms are defined in a natural way. Partial c-terms represent the result of partially evaluated expressions, and thus they can be considered as approximations to the value of expressions. Moreover, we will consider the sets $Term_{\bot,\mathsf{F}}$ and $CTerm_{\bot,\mathsf{F}}$. A natural *approximation ordering* $\leq$ over $CTerm_{\bot,\mathsf{F}}$ can be defined as the least partial ordering satisfying: $\bot \leq t$, $X \leq X$ and $h(t_1, ..., t_n) \leq h(t'_1, ..., t'_n)$, if $t_i \leq t'_i$ for all $i \in \{1, ..., n\}$, $h \in DC \cup FS$. The intended meaning of $t \leq t'$ is that t is less defined or has less information than t'. Note that the only relations satisfied by F are $\bot \leq \mathsf{F}$ and $\mathsf{F} \leq \mathsf{F}$. In particular, F is maximal. This is reasonable, since F represents 'failure of reduction' and this gives a no further refinable information about the result of the evaluation of an expression.

In the context of *CRWLF*, a program $\mathcal{P}$ is a set of conditional rewrite rules of the form:

$$\underbrace{f(t_1, ..., t_n)}_{head} \rightarrow \underbrace{r}_{body} \Leftarrow \underbrace{C}_{condition}$$

where $f \in FS^n$, and fulfilling the following conditions: $(t_1, ..., t_n)$ is a linear tuple (each variable in it occurs only once) with $t_1, ..., t_n \in CTerm$; $r \in Term$; C is a set of constraints of the form $e' \bowtie e''$ (*joinability*), $e' <> e''$ (*divergence*), $e' \not\bowtie e''$ (*failure of joinability*) or $e' \not<> e''$ (*failure of divergence*) where $e', e'' \in Term$[1]; *extra variables* are not allowed[2], i.e. $var(r) \cup var(C) \subseteq var(\overline{t})$. The reading of the rule is: $f(t_1, ..., t_n)$ reduces to r if the condition C is satisfied. We will need to use *c-instances* of rules, where a c-instance of a program rule R is defined as $[R]_{\bot,\mathsf{F}} = \{R\theta \mid \theta \in CSubst_{\bot,\mathsf{F}}\}$ with $CSubst_{\bot,\mathsf{F}} = \{\theta : \mathcal{V} \rightarrow CTerm_{\bot,\mathsf{F}}\}$.

In our framework and due to we allow non-determinism, in general an expression can be reduced to an infinite set of values, but we will need some finite representation of these sets. For example, we can define the non-deterministic function f as: $f \rightarrow zero$, $f \rightarrow suc(suc(f))$. It is easy to see that f can be reduced

[1] The original *CRWL* framework [7] only considered joinabilities; divergence constraints were incorporated in [12] and failures of both joinabilities and divergences are introduced in [14].

[2] In [7] extra variables are allowed, but the use of function nesting and non-deterministic functions can nicely replace them in many cases.

Table 1. Rules for *CRWLF*-provability

(1) $\dfrac{}{e \lhd \{\bot\}}$	(2) $\dfrac{}{X \lhd \{X\}}$	$X \in \mathcal{V}$
(3) $\dfrac{e_1 \lhd \mathcal{C}_1 \quad \ldots \quad e_n \lhd \mathcal{C}_n}{c(e_1,\ldots,e_n) \lhd \{c(\bar{t}) \mid \bar{t} \in \mathcal{C}_1 \times \ldots \times \mathcal{C}_n\}}$		$c \in DC^n \cup \{\mathsf{F}\}$
(4) $\dfrac{e_1 \lhd \mathcal{C}_1 \quad \ldots \quad e_n \lhd \mathcal{C}_n \quad \ldots \quad f(\bar{t}) \lhd_R \mathcal{C}_{R,\bar{t}} \quad \ldots}{f(e_1,\ldots,e_n) \lhd \bigcup_{R \in \mathcal{P}_f, \bar{t} \in \mathcal{C}_1 \times \ldots \times \mathcal{C}_n} \mathcal{C}_{R,\bar{t}}}$		$f \in FS^n$
(5) $\dfrac{}{f(\bar{t}) \lhd_R \{\bot\}}$	(6) $\dfrac{r \lhd \mathcal{C} \quad C}{f(\bar{t}) \lhd_R \mathcal{C}}$	$(f(\bar{t}) \to r \Leftarrow C) \in [R]_{\bot,\mathsf{F}}$
(7) $\dfrac{e_i \tilde{\diamondsuit} e_i'}{f(\bar{t}) \lhd_R \{\mathsf{F}\}}$	(8) $\dfrac{}{f(t_1,\ldots,t_n) \lhd_R \{\mathsf{F}\}}$	$(f(\bar{t}) \to r \Leftarrow \ldots, e_i \diamondsuit e_i', \ldots) \in [R]_{\bot,\mathsf{F}}$ $R \equiv (f(s_1,\ldots,s_n) \to r \Leftarrow C)$, t_i and s_i have a $DC \cup \{\mathsf{F}\}$-clash for some $i \in \{1,\ldots,n\}$
(9) $\dfrac{e \lhd \mathcal{C} \quad e' \lhd \mathcal{C}'}{e \bowtie e'}$	$\exists t \in \mathcal{C}, t' \in \mathcal{C}'\ t \downarrow t'$	
(10) $\dfrac{e \lhd \mathcal{C} \quad e' \lhd \mathcal{C}'}{e \diamond\!\!> e'}$	$\exists t \in \mathcal{C}, t' \in \mathcal{C}'\ t \uparrow t'$	
(11) $\dfrac{e \lhd \mathcal{C} \quad e' \lhd \mathcal{C}'}{e \not\bowtie e'}$	$\forall t \in \mathcal{C}, t' \in \mathcal{C}'\ t \not\downarrow t'$	
(12) $\dfrac{e \lhd \mathcal{C} \quad e' \lhd \mathcal{C}'}{e \not\diamond\!\!> e'}$	$\forall t \in \mathcal{C}, t' \in \mathcal{C}'\ t \not\gamma t'$	

to the values $zero, suc(suc(zero))$, $suc(suc(suc(suc(zero))))$... We can use the undefined value $\bot$ to express that the possible reductions of f have the form $zero$ or $suc(suc(\bot))$, noted as $f \lhd \{zero, suc(suc(\bot))\}$. This set of values is a *Sufficient Approximation Set* (*SAS*) for f which provides enough information about the values of f to prove that f cannot be reduced to $suc(zero)$. Of course, an expression will have multiple *SAS*'s. Any expression has $\{\bot\}$ as its simplest *SAS* and, for example, the expression f has an infinite number of *SAS*'s: $\{\bot\}$, $\{zero, suc(suc(\bot))\}$, $\{zero, suc(suc(zero)), suc(suc(suc(suc(\bot))))\}$,...
In *CRWLF* five kinds of statements can be deduced (assume $e \in Term_{\bot,\mathsf{F}}$):

- $e \lhd \mathcal{C}$: $\mathcal{C}$ is a *SAS* for e;
- $e \bowtie e'$ (joinability): e and e' can be both reduced to some $t \in CTerm$;
- $e \diamond\!\!> e'$ (divergence): e and e' can be reduced to some (possibly partial) c-terms t and t' having a DC-clash.
- $e \not\bowtie e'$: failure of $e \bowtie e'$;
- $e \not\diamond\!\!> e'$: failure of $e \diamond\!\!> e'$.

where given a set of constructors S, we say that the c-terms t and t' have a *S-clash* if they have different constructors of S at the same position.

We will use the symbol $\diamondsuit$ to refer to any of the constraints $\bowtie, \diamond\!\!>, \not\bowtie, \not\diamond\!\!>$. The constraints $\not\bowtie$ and $\bowtie$ are called the *complementary* of each other; the same holds for $\not\diamond\!\!>$ and $\diamond\!\!>$, and we write $\tilde{\diamondsuit}$ for the complementary of $\diamondsuit$. When proving a constraint $e \diamondsuit e'$, the calculus *CRWLF* will evaluate a *SAS* for the expressions e and e'. These *SAS*'s will consist of c-terms and provability of the constraint $e \diamondsuit e'$ depends on certain syntactic (hence decidable) relations between these ones defined as follows.

Definition 1 (Relations over $CTerm_{\bot,\mathsf{F}}$).

- $t \downarrow t' \Leftrightarrow_{def} t = t', t \in CTerm$

- $t \uparrow t' \Leftrightarrow_{def}$ *t and t' have a DC-clash*
- $t \not\downarrow t' \Leftrightarrow_{def}$ *t or t' contain F as subterm, or they have a DC-clash*
- *$\not\gamma$ is defined as the least symmetric relation over $CTerm_{\perp,\mathsf{F}}$ satisfying:*
  - *i) $X \not\gamma X$, for all $X \in \mathcal{V}$*
  - *ii) $\mathsf{F} \not\gamma t$, for all $t \in CTerm_{\perp,\mathsf{F}}$*
  - *iii) if $t_1 \not\gamma t'_1, ..., t_n \not\gamma t'_n$ then $c(t_1, ..., t_n) \not\gamma c(t'_1, ..., t'_n)$, for $c \in DC^n$*

Table 1 shows the rules for the *CRWLF*-calculus. Rule (1) considers $\{\perp\}$ as the simplest *SAS* for any expression. Rules (2) and (3) consider the case of variables and constructors. In rule (4), when evaluating a function call $f(e_1, ..., e_n)$, we produce *SAS*'s for the arguments. Next, we consider all the possible values t of the cross product of these *SAS*'s and all the rules R for f, denoted by $\mathcal{P}_f$, by generating a *SAS* for each combination: $f(\bar{t}) \lhd_R \mathcal{C}_{R,\bar{t}}$. The notation $\lhd_R$ indicates that only the rule R is used to produce the corresponding *SAS*. By joining the *SAS*'s $\mathcal{C}_{R,\bar{t}}$, we obtain the final *SAS* for $f(e_1, ..., e_n)$. Rules (5) to (8) consider all the possible ways in which a rule R can be used to produce a *SAS* for a call $f(\bar{t})$ where $t_i \in CTerm_{\perp,\mathsf{F}}$. Rule (5) is the trivial case. Rule (6) applies whether there exists a c-instance of R with head $f(\bar{t})$ and the constraints of C of this c-instance are provable. In this case, the *SAS* will be the one produced by the body of the c-instance. Rules (7) and (8) produce the *SAS* $\{\mathsf{F}\}$ due to a failure of one of the constraints (by proving the complementary) and a failure in parameter passing, respectively. Finally, the rules (9) to (12) deal with constraints and are easily defined by using the relations $\downarrow, \uparrow, \not\downarrow, \not\gamma$.

It can be proved that the relations $\lhd, \bowtie, \diamond, \not\bowtie, \not\diamond$ satisfy some desirable properties such as monotonicity or closure under substitutions. On the other hand, if $e \diamond e'$ is provable, then $e \tilde{\diamond} e'$ is not provable. Full details about *CRWLF* can be found in [14].

Finally, a *goal* $\mathcal{G}$ is a set of constraints and a solution $\theta \in CSubst_{\perp,\mathsf{F}}$ of $\mathcal{G}$ w.r.t. a *CRWLF*-program $\mathcal{P}$ holds $\mathcal{P} \vdash_{CRWLF} \mathcal{G}\theta$.

3 Herbrand Models

In this section we present *CRWLF*-Herbrand algebras and a fix point operator for computing the least Herbrand model of a program. We assume the reader has familiarity with basic concepts of model theory on functional and logic programming (see [3,7] for more details), but now we point up some notions.

Given S, a *partially ordered set* (in short, *poset*) with a least element *bottom* $\perp$ (equipped with a partial order $\leq$), the set of all totally defined elements of S will be noted by *Def(S)*. We write $\mathcal{C}(S)$, $\mathcal{I}(S)$ for the sets of cones and ideals of S, respectively. The set $\bar{S} =_{def} \mathcal{I}(S)$ denotes the *ideal completion* of S, which is also a *poset* under the *set-inclusion* ordering $\subseteq$, and there is a natural mapping for each $x \in S$ into the principal ideal generated by x, $< x >=_{def} \{y \in S : y \leq x\} \in \bar{S}$. Furthermore, $\bar{S}$ is a *cpo* (i.e. every directed set $D \subseteq \bar{S}$ has a least upper bound) whose finite elements are the principal ideals $< x >$, $x \in S$.

Definition 2 (Herbrand Algebras). *For any given signature Σ, a Herbrand algebra $\mathcal{H}$ is an algebraic structure of the form $\mathcal{H} = (CTerm_{\perp,\mathsf{F}}, \{f^{\mathcal{H}}\}_{\{f \in FS\}})$ where $CTerm_{\perp,\mathsf{F}}$ is a poset with the approximation ordering $\leq$ and $f^{\mathcal{H}} \in [CTerm^{l}_{\perp,\mathsf{F}} \rightarrow_n CTerm_{\perp,\mathsf{F}}]$ for $f \in FS^l$, where $[D \rightarrow_n E] =_{def} \{f : D \rightarrow \mathcal{C}(E) | \ \forall\ u, u' \in D : (u \leq u' \Rightarrow f(u) \subseteq f(u'))\}$. From the set $\{f^{\mathcal{H}}\}_{\{f \in FS\}}$, we can distinguish the deterministic functions $f \in FS^n$, holding that $f^{\mathcal{H}} \in [CTerm^{n}_{\perp,\mathsf{F}} \rightarrow_d CTerm_{\perp,\mathsf{F}}]$ where $[D \rightarrow_d E] =_{def} \{f \in [D \rightarrow_n E] | \ \forall\ u \in D : f(u) \in \mathcal{I}(E)\}$. Finally, the elements of $Def(\mathcal{H})$ are the elements of $CTerm_{\mathsf{F}}$.*

Definition 3 (Herbrand Denotation). *Let $\mathcal{H}$ be a Herbrand algebra, a valuation over $\mathcal{H}$ is any mapping $\eta : \mathcal{V} \rightarrow CTerm_{\perp,\mathsf{F}}$, and we say that η is totally defined iff $\eta(X) \in Def(\mathcal{H})$ for all $X \in \mathcal{V}$. We denote by $Val(\mathcal{H})$ the set of all valuations, and by $DefVal(\mathcal{H})$ the set of all totally defined valuations. The evaluation of an $e \in Term_{\perp,\mathsf{F}}$ in $\mathcal{H}$ under η yields $[\![e]\!]^{\mathcal{H}}\eta \in \mathcal{C}(CTerm_{\perp,\mathsf{F}})$ which is defined recursively as follows:*

- $[\![\perp]\!]^{\mathcal{H}}\eta =_{def} < \perp >$, $[\![\mathsf{F}]\!]^{\mathcal{H}}\eta =_{def} < \mathsf{F} >$ *and* $[\![X]\!]^{\mathcal{H}}\eta =_{def} < \eta(X) >$, *for* $X \in \mathcal{V}$.
- $[\![c(e_1, \ \ldots\ , e_n)]\!]^{\mathcal{H}}\eta =_{def} < c([\![e_1]\!]^{\mathcal{H}}\eta, \ \ldots\ , [\![e_n]\!]^{\mathcal{H}}\eta) >$ *for all* $c \in DC^n$.
- $[\![f(e_1, \ \ldots\ , e_n)]\!]^{\mathcal{H}}\eta =_{def} f^{\mathcal{H}}([\![e_1]\!]^{\mathcal{H}}\eta, \ \ldots\ , [\![e_n]\!]^{\mathcal{H}}\eta)$, *for all* $f \in FS^n$.

Due to non-determinism, the evaluation of an expression yields a cone rather than an element. It can be proved that for any $e \in Term_{\perp,\mathsf{F}}$ and $\eta \in Val(\mathcal{H})$ then $[\![e]\!]^{\mathcal{H}}\eta \in \mathcal{I}(CTerm_{\perp,\mathsf{F}})$ if $f^{\mathcal{H}}$ is deterministic for every f occurring in e, and $[\![e]\!]^{\mathcal{H}}\eta \in \mathcal{I}(CTerm_{\mathsf{F}})$ if $e \in Term_{\mathsf{F}}$ and $\eta \in DefVal(\mathcal{H})$.

Definition 4 (Poset of Herbrand Algebras). *We can define a partial order over the Herbrand algebras as follows: given $\mathcal{A}$ and $\mathcal{B}$, then $\mathcal{A} \leq \mathcal{B}$ iff $f^{\mathcal{A}}(t_1, \ldots, t_n) \subseteq f^{\mathcal{B}}(t_1, \ldots, t_n)$ for every $f \in FS^n$ and $t_i \in CTerm_{\perp,\mathsf{F}}, 1 \leq i \leq n$. In such a way that the Herbrand algebras with this order are a poset with bottom.*

Moreover, it can be proved that the ideal completion of this poset is a cpo, called $\mathcal{HALG}$, and $[\![\ ]\!]$ is continuous in $\mathcal{HALG}$.

Definition 5 (Herbrand Models). *Given a program $\mathcal{P}$ and a Herbrand algebra $\mathcal{H}$. We define:*

- *$\mathcal{H}$ satisfies a SAS $e \lhd \mathcal{C}$ under a valuation η (in symbols, $(\mathcal{H}, \eta) \models e \lhd \mathcal{C}$) iff $[\![t]\!]^{\mathcal{H}}\eta \in [\![e]\!]^{\mathcal{H}}\eta$ for every $t \in \mathcal{C}$.*
- *$\mathcal{H}$ satisfies a joinability $e \bowtie e'$ under a valuation η (in symbols, $(\mathcal{H}, \eta) \models e \bowtie e'$) iff there exist $t \in [\![e]\!]^{\mathcal{H}}\eta \cap CTerm_{\perp,\mathsf{F}}$ and $t' \in [\![e']\!]^{\mathcal{H}}\eta \cap CTerm_{\perp,\mathsf{F}}$ such that $t \downarrow t'$.*
- *$\mathcal{H}$ satisfies a divergence $e <\!> e'$ under a valuation η (in symbols, $(\mathcal{H}, \eta) \models e <\!> e'$) iff there exist $t \in [\![e]\!]^{\mathcal{H}}\eta \cap CTerm_{\perp,\mathsf{F}}$ and $t' \in [\![e']\!]^{\mathcal{H}}\eta \cap CTerm_{\perp,\mathsf{F}}$ such that $t \uparrow t'$.*
- *$\mathcal{H}$ satisfies a failure of joinability $e \not\bowtie e'$ under a valuation η (in symbols, $(\mathcal{H}, \eta) \models e \not\bowtie e'$) iff for every $t \in [\![e]\!]^{\mathcal{H}}\eta \cap CTerm_{\perp,\mathsf{F}}$ and $t' \in [\![e']\!]^{\mathcal{H}}\eta \cap CTerm_{\perp,\mathsf{F}}$, then $t \not\downarrow t'$ holds.*

- $\mathcal{H}$ *satisfies a failure of divergence* $e \lhd\rhd e'$ *under a valuation* η *(in symbols,* $(\mathcal{H}, \eta) \models e \lhd\rhd e'$*) iff for every* $t \in [\![e]\!]^{\mathcal{H}}\eta \cap CTerm_{\perp,\mathsf{F}}$ *and* $t' \in [\![e']\!]^{\mathcal{H}}\eta \cap CTerm_{\perp,\mathsf{F}}$*, then* $t \not\Upsilon t'$ *holds.*
- $\mathcal{H}$ *satisfies a rule* $f(\bar{t}) \to r \Leftarrow C \in \mathcal{P}$ *iff*
  - *every valuation* η *such that* $(\mathcal{H}, \eta) \models C$ *verifies* $[\![f(\bar{t})]\!]^{\mathcal{H}}\eta \supseteq [\![r]\!]^{\mathcal{H}}\eta$
  - *every valuation* η *such that, for some* $i \in \{1, \ldots, n\}$*,* l_i *and* t_i *have a* $DC \cup \{\mathsf{F}\}$*-clash, where* $l_i \in [\![s_i]\!]^{\mathcal{H}}\eta$*, verifies* $\mathsf{F} \in [\![f(\bar{s})]\!]^{\mathcal{H}}\eta$
  - *every valuation* η *such that there exists* $e_i \Diamond e_i' \in C$ *such that* $(\mathcal{H}, \eta) \models e_i \tilde{\Diamond} e_i'$ *verifies* $\mathsf{F} \in [\![f(\bar{t})]\!]^{\mathcal{H}}\eta$
- $\mathcal{H}$ *is a model of* $\mathcal{P}$ *(in symbols,* $\mathcal{H} \models \mathcal{P}$*) iff* $\mathcal{H}$ *satisfies all the rules in* $\mathcal{P}$*.*

Definition 6 (Fix Point Operator). *Given a Herbrand algebra* $\mathcal{A}$*, and* $f \in FS$*, we define the fix point operator as:*

$$
\begin{aligned}
T_{\mathcal{P}}(\mathcal{A}, f)(\bar{s}) =_{def} \{\ [\![r]\!]^{\mathcal{A}}_{\eta} \mid & \textit{ if there exist } f(\bar{t}) \to r \Leftarrow C \in \mathcal{P}, \textit{ and } \eta \in Val(\mathcal{A}) \\
& \textit{such that } s_i \in [\![t_i]\!]^{\mathcal{A}}_{\eta} \textit{ and } (\mathcal{A}, \eta) \models C\} \\
\cup \{\ \mathsf{F} \mid & \textit{ if there exists } f(\bar{t}) \to r \Leftarrow C \in \mathcal{P}, \textit{ such that for some} \\
& i \in \{1, \ldots, n\}, s_i \textit{ and } t_i \textit{ have a } DC \cup \{\mathsf{F}\} - \textit{clash}\} \\
\cup \{\ \mathsf{F} \mid & \textit{ if there exist } f(\bar{t}) \to r \Leftarrow C \in \mathcal{P}, \textit{ and } \eta \in Val(\mathcal{A}) \textit{ such} \\
& \textit{that } s_i \in [\![t_i]\!]^{\mathcal{A}}_{\eta} \textit{ and } (\mathcal{A}, \eta) \models e \tilde{\Diamond} e' \textit{ for some } e \Diamond e' \in C\} \\
\cup \{\ \perp \mid & \textit{ otherwise}\}
\end{aligned}
$$

Given $\mathcal{A} \in \mathcal{HALG}$, there exists an unique $\mathcal{B} \in \mathcal{HALG}$ denoted by $T_{\mathcal{P}}(\mathcal{A})$ such that $f^{\mathcal{B}}(t_1, \ldots, t_n) = T_P(\mathcal{A}, f)(t_1, \ldots, t_n)$ for every $f \in FS^n$ and $t_i \in CTerm_{\perp,\mathsf{F}}$, $1 \leq i \leq n$. The following result which characterizes the least Herbrand model can be ensured.

Theorem 1. *The fix point operator* $T_{\mathcal{P}}$ *is continuous in* $\mathcal{HALG}$ *and satisfies:*

1. *For every* $\mathcal{A} \in \mathcal{HALG}$*:* $\mathcal{A} \models \mathcal{P}$ *iff* $T_P(\mathcal{A}) \leq \mathcal{A}$*.*
2. $T_{\mathcal{P}}$ *has a least fix point* $\mathcal{M}_{\mathcal{P}} = {\mathcal{H}_{\mathcal{P}}}^{\omega}$ *where* ${\mathcal{H}_{\mathcal{P}}}^{0}$ *is the bottom in* $\mathcal{HALG}$ *and* ${\mathcal{H}_{\mathcal{P}}}^{k+1} = T_{\mathcal{P}}({\mathcal{H}_{\mathcal{P}}}^{k})$
3. $\mathcal{M}_{\mathcal{P}}$ *is the least Herbrand model of* $\mathcal{P}$*.*

Moreover, we can see that satisfaction in $\mathcal{M}_{\mathcal{P}}$ can be characterized in terms of $\vdash_{CRWLF}$ provability: for any constraint φ, $\mathcal{P} \vdash_{CRWLF} \varphi$ iff $(\mathcal{M}_{\mathcal{P}}, \eta) \models \varphi$, for all $\eta \in DefVal(\mathcal{H})$. Therefore, $\vdash_{CRWLF}$ is sound and complete w.r.t. the Herbrand models, and thus the Herbrand model $\mathcal{M}_{\mathcal{P}}$ can be regarded as the intended *(canonical) model of a program* $\mathcal{P}$*.*

4 Magic Transformation for Negative Constraints

In order to present our new magic transformation, first we will show the main ideas of the presented one in [1] by using the following example where the proposed goal has as solution $\mathtt{X} = \mathtt{s(s(Z))}, \mathtt{Y} = \mathtt{s(0)}$.

Original Program			**Goal**
(**1**) $\mathtt{f(s(X_1), X_2)}$	$\to \mathtt{X_2}$	$\Leftarrow \mathtt{h(X_1, X_2)} \bowtie \mathtt{0}, \mathtt{p(X_2)} \bowtie \mathtt{s(0)}.$	$:- \mathtt{f(g(X), Y)} \bowtie \mathtt{s(0)}.$
(**2**) $\mathtt{g(s(X_3))}$	$\to \mathtt{s(g(X_3))}.$		
(**3**) $\mathtt{h(s(X_4), s(0))}$	$\to \mathtt{0}.$		
(**4**) $\mathtt{p(s(0))}$	$\to \mathtt{s(0)}.$		

Transformed Program **Goal**

$: -\mathtt{f}(\mathtt{mg\_g}^{\mathtt{N}}(\mathtt{X}), \mathtt{Y}) \bowtie \mathtt{s}(0).$

Filtered Rules

(1) $\mathtt{f}(\mathtt{s}(\mathtt{X}_1), \mathtt{X}_2) \rightarrow \mathtt{X}_2 \Leftarrow \mathtt{mg\_f}^{\mathtt{P}}(\mathtt{s}(\mathtt{X}_1), \mathtt{X}_2) \bowtie \mathtt{true}, \mathtt{h}(\mathtt{X}_1, \mathtt{X}_2) \bowtie 0, \mathtt{p}(\mathtt{X}_2) \bowtie \mathtt{s}(0).$

(2) $\mathtt{g}(\mathtt{s}(\mathtt{X}_3)) \rightarrow \mathtt{s}(\mathtt{g}(\mathtt{X}_3)) \Leftarrow \mathtt{mg\_g}^{\mathtt{P}}(\mathtt{s}(\mathtt{X}_3)) \bowtie \mathtt{true}.$

(3) $\mathtt{h}(\mathtt{s}(\mathtt{X}_4), \mathtt{s}(0)) \rightarrow 0 \Leftarrow \mathtt{mg\_h}^{\mathtt{P}}(\mathtt{s}(\mathtt{X}_4), \mathtt{s}(0)) \bowtie \mathtt{true}.$

(4) $\mathtt{p}(\mathtt{s}(0)) \rightarrow \mathtt{s}(0) \Leftarrow \mathtt{mg\_p}^{\mathtt{P}}(\mathtt{s}(0)) \bowtie \mathtt{true}.$

Magic Rules

(5) $\mathtt{mg\_f}^{\mathtt{P}}(\mathtt{mg\_g}^{\mathtt{N}}(\mathtt{X}), \mathtt{Y}) \rightarrow \mathtt{true}.$

(6) $\mathtt{mg\_h}^{\mathtt{P}}(\mathtt{X}_5, \mathtt{X}_6) \rightarrow \mathtt{mg\_f}^{\mathtt{P}}(\mathtt{s}(\mathtt{X}_5), \mathtt{X}_6).$

(7) $\mathtt{mg\_p}^{\mathtt{P}}(\mathtt{X}_8) \rightarrow \mathtt{mg\_f}^{\mathtt{P}}(\mathtt{s}(\mathtt{X}_7), \mathtt{X}_8) \Leftarrow \mathtt{h}(\mathtt{X}_7, \mathtt{X}_8) \bowtie 0.$

(8) $\mathtt{mg\_f}^{\mathtt{P}}(\mathtt{s}(\mathtt{mg\_g}^{\mathtt{N}}(\mathtt{X}_9)), \mathtt{X}_{10}) \rightarrow \mathtt{mg\_f}^{\mathtt{P}}(\mathtt{mg\_g}^{\mathtt{N}}(\mathtt{s}(\mathtt{X}_9)), \mathtt{X}_{10}).$

(9) $\mathtt{mg\_h}^{\mathtt{P}}(\mathtt{s}(\mathtt{mg\_g}^{\mathtt{N}}(\mathtt{X}_{11})), \mathtt{X}_{12}) \rightarrow \mathtt{mg\_h}^{\mathtt{P}}(\mathtt{mg\_g}^{\mathtt{N}}(\mathtt{s}(\mathtt{X}_{11})), \mathtt{X}_{12}).$

Goal Solving Rules

(10) $\mathtt{f}(\mathtt{mg\_g}^{\mathtt{N}}(\mathtt{s}(\mathtt{X}_{13})), \mathtt{X}_{14}) \rightarrow \mathtt{f}(\mathtt{s}(\mathtt{mg\_g}^{\mathtt{N}}(\mathtt{X}_{13})), \mathtt{X}_{14}) \Leftarrow \mathtt{mg\_f}^{\mathtt{P}}(\mathtt{mg\_g}^{\mathtt{N}}(\mathtt{s}(\mathtt{X}_{13})), \mathtt{X}_{14}) \bowtie \mathtt{true}.$

(11) $\mathtt{h}(\mathtt{mg\_g}^{\mathtt{N}}(\mathtt{s}(\mathtt{X}_{15})), \mathtt{X}_{16}) \rightarrow \mathtt{h}(\mathtt{s}(\mathtt{mg\_g}^{\mathtt{N}}(\mathtt{X}_{15})), \mathtt{X}_{16}) \Leftarrow \mathtt{mg\_h}^{\mathtt{P}}(\mathtt{mg\_g}^{\mathtt{N}}(\mathtt{s}(\mathtt{X}_{15})), \mathtt{X}_{16}) \bowtie \mathtt{true}.$

The magic transformation transforms every pair ($\mathcal{P}$,$\mathcal{G}$), where $\mathcal{P}$ is a program and $\mathcal{G}$ is a goal, into a pair ($\mathcal{P}^{\mathcal{MG}}$,$\mathcal{G}^{\mathcal{MG}}$) in such a way that the transformed program $\mathcal{P}^{\mathcal{MG}}$, evaluated by means of the fix point operator, computes solutions for $\mathcal{G}^{\mathcal{MG}}$, which are also solutions of $\mathcal{G}$ w.r.t. the program $\mathcal{P}$.

Firstly, the so-called *passing magic (boolean) functions* of the form mg_f^P, like in logic programming, will activate the evaluation of the functions through the fix point operator (see rules **(1)**, **(2)**, **(3)** and **(4)** in the transformed program), whenever there exists a call, passing the arguments from head to body and conditions of every program rule for them (see rules **(6)** and **(7)** in the above program obtained from the rule **(1)** of the original program) by means of left-to-right sips.

Secondly, the transformation process adds magic rules for the *outermost functions* (set denoted by $outer(\mathcal{P}, \mathcal{G})$), which are defined as the leftmost functions occurring either in every side of each constraint of the goal, or in the body and every side of each constraint of every program rule of the outermost functions, or in every side of each constraint of every program rule of functions occurring in the scope of an outermost function. In the above example, `f`, `h` and `p`. The idea is that whenever a function is in the scope of an outermost function, every program rule for the *inner function* generates a rule, called *nesting magic rule*, for the passing magic function of the outermost one (see rule **(8)** generated by the nesting occurring in the goal). The head and body of every program rule of each inner function are "turned around" and introduced as arguments of head and body, respectively, of the magic rule for the outermost function, occurring at the same position where they appear in the goal, and filling the rest of arguments with fresh variables. The inner functions are substituted in the magic rules by the so-called *nesting magic constructors* of the form mg_f^N, given that patterns in program rules must be c-terms. In order to get a lazy evaluation, the introduction of these nesting magic rules is only achieved whether the nested function is *demanded by some of the rules* of the nesting function; that is, the rule pattern is a c-term or it is variable occurring out of scope of a function in the body or in the constraints. Moreover, the same kind of passing magic rules must be introduced for each information passing in the rule. For instance, given that `g` becomes an inner function for `h` due to the information passing from the

first argument of $\mathtt{f}$ to $\mathtt{h}$ in the rule **(1)**, then the nesting magic rule **(9)** is also included.

Thirdly, every constraint $e \bowtie e' \in \mathcal{G}$ is transformed into a new constraint wherein each inner function is replaced by nesting magic constructors. In the above example, $\mathtt{f}(\mathtt{mg\_g}^{\mathtt{N}}(\mathtt{X}), \mathtt{Y}) \bowtie \mathtt{s}(\mathtt{0})$. Moreover, a new rule is generated as seed from the goal of the original program (see rule **(5)**).

Lastly, whenever there exists a nesting, new rules will include, called *goal solving rules*, which will allow us to get the answer of the new transformed goal and they are similar to the nesting magic rules (see rules **(10)** and **(11)**).

In our new magic transformation in order to handle negative information according to *CRWLF*, we have to lazily solve the four kinds of constraints. For instance, $\not\bowtie$ should be lazily solved, that is, as far as needed up to $\not\downarrow$ holds, like in the following example.

Original Program **Goal**

$(\mathbf{1})\ \mathtt{f}(\mathtt{X}_1) \to \mathtt{s}(\mathtt{t}(\mathtt{X}_1))$. $\quad : -\mathtt{f}(\mathtt{X}) \not\bowtie \mathtt{g}(\mathtt{X})$.
$(\mathbf{2})\ \mathtt{g}(\mathtt{X}_2) \to \mathtt{0}$.
$(\mathbf{3})\ \mathtt{t}(\mathtt{X}_3) \to \mathtt{s}(\mathtt{s}(\mathtt{0}))$.

Transformed Program **Goal**

Filtered Rules $\quad : -\mathtt{f}(\mathtt{X}) \not\bowtie \mathtt{g}(\mathtt{X})$.

$(\mathbf{1})\ \mathtt{f}(\mathtt{X}_1) \to \mathtt{s}(\mathtt{t}(\mathtt{X}_1)) \Leftarrow \mathtt{mg\_f}^{\mathtt{P}}(\mathtt{X}_1) \bowtie \mathtt{true}$.
$(\mathbf{2})\ \mathtt{g}(\mathtt{X}_2) \to \mathtt{0} \Leftarrow \mathtt{mg\_g}^{\mathtt{P}}(\mathtt{X}_2) \bowtie \mathtt{true}$.
$(\mathbf{3})\ \mathtt{t}(\mathtt{X}_3) \to \mathtt{s}(\mathtt{s}(\mathtt{0})) \Leftarrow \mathtt{mg\_t}^{\mathtt{P}}(\mathtt{X}_3) \bowtie \mathtt{true}$.

Magic Rules

$(\mathbf{4})\ \mathtt{mg\_} \not\bowtie^{\mathtt{P}} (\mathtt{mg\_f}^{\mathtt{N}}(\mathtt{X}), \mathtt{mg\_g}^{\mathtt{N}}(\mathtt{X})) \to \mathtt{true}$.
$(\mathbf{5})\ \mathtt{mg\_} \not\bowtie^{\mathtt{P}} (\mathtt{s}(\mathtt{mg\_t}^{\mathtt{N}}(\mathtt{X}_4)), \mathtt{X}_5) \to \mathtt{mg\_} \not\bowtie^{\mathtt{P}} (\mathtt{mg\_f}^{\mathtt{N}}(\mathtt{X}_4), \mathtt{X}_5)$.
$(\mathbf{6})\ \mathtt{mg\_} \not\bowtie^{\mathtt{P}} (\mathtt{X}_6, \mathtt{0}) \to \mathtt{mg\_} \not\bowtie^{\mathtt{P}} (\mathtt{X}_6, \mathtt{mg\_g}^{\mathtt{N}}(\mathtt{X}_7))$.
$(\mathbf{7})\ \mathtt{mg\_} \not\bowtie^{\mathtt{P}} (\mathtt{s}(\mathtt{s}(\mathtt{0})), \mathtt{X}_9) \to \mathtt{mg\_} \not\bowtie^{\mathtt{P}} (\mathtt{mg\_t}^{\mathtt{N}}(\mathtt{X}_8), \mathtt{X}_9)$.
$(\mathbf{8})\ \mathtt{mg\_} \not\bowtie^{\mathtt{P}} (\mathtt{X}_{10}, \mathtt{X}_{11}) \to \mathtt{mg\_} \not\bowtie^{\mathtt{P}} (\mathtt{s}(\mathtt{X}_{10}), \mathtt{s}(\mathtt{X}_{11}))$.
$(\mathbf{9})\ \mathtt{mg\_f}^{\mathtt{P}}(\mathtt{X}_{12}) \to \mathtt{mg\_} \not\bowtie^{\mathtt{P}} (\mathtt{mg\_f}^{\mathtt{N}}(\mathtt{X}_{12}), \mathtt{X}_{13})$.
$(\mathbf{10})\ \mathtt{mg\_g}^{\mathtt{P}}(\mathtt{X}_{15}) \to \mathtt{mg\_} \not\bowtie^{\mathtt{P}} (\mathtt{X}_{14}, \mathtt{mg\_g}^{\mathtt{N}}(\mathtt{X}_{15}))$.
$(\mathbf{11})\ \mathtt{mg\_t}^{\mathtt{P}}(\mathtt{X}_{16}) \to \mathtt{mg\_} \not\bowtie^{\mathtt{P}} (\mathtt{mg\_t}^{\mathtt{N}}(\mathtt{X}_{16}), \mathtt{X}_{17})$.

In this example, the function $\mathtt{t}$ does not need to be evaluated since the *SAS*'s $\{\mathtt{s}(\bot)\}$ and $\{\mathtt{0}\}$ for the functions $\mathtt{f}$ and $\mathtt{g}$ respectively, are enough to solve the goal $\mathtt{f}(\mathtt{X}) \not\bowtie \mathtt{g}(\mathtt{X})$ which holds by DC-clash (see rule (11) of *CRWLF*).

With this aim we will consider the operators $\bowtie$, $\lozenge$, $\not\bowtie$ and $\not\lozenge$ as they were outermost symbols, and the functions occurring in the constraints as they were inner symbols [3]. It supposes the introduction of nesting magic rules for the operators and the functions become magic constructors inside of these new nesting magic rules. In the above example, the new seed is the rule **(4)** which nests both expressions occurring in the goal constraint and thus the nesting magic rules **(5)** and **(6)** are generated.

Moreover, the magic transformation analyzes the body of each definition rule of the nested functions $\mathtt{f}$ and $\mathtt{g}$ by detecting, in the case of $\mathtt{f}$, a constructor in the body, and thus it generates the rule **(8)**. This rule indicates that if the evaluation of both hand-sides of the constraint generates the same outermost constructor, then the bottom-up evaluation has to continue. In the general case, magic rules of this kind for a given constraint will be added for every constructor out of the

[3] Remark that it does not mean that the notion of outermost function will be modified in the rest of the paper.

scope of a function either occurring in the constraint or in the body of some rule of the leftmost functions of the constraint. Finally, the rules **(9), (10)** and **(11)** will allow to recover the passing magic functions. In the general case, one of these will be added for each outermost function symbol. The bottom-up evaluation of this program is as follows:

⊙ $\mathcal{H}^0_{\mathcal{PMG}} = \bot$

⊙ $\mathcal{H}^1_{\mathcal{PMG}} = \{\mathtt{mg\_} \not\bowtie^{\mathtt{P}} (\mathtt{mg\_f}^{\mathtt{N}}(\mathtt{X}), \mathtt{mg\_g}^{\mathtt{N}}(\mathtt{X})) \lhd \{\mathtt{true}\}, \ldots\}.$

⊙ $\mathcal{H}^2_{\mathcal{PMG}} = \{\mathtt{mg\_f}^{\mathtt{P}}(\mathtt{X}) \lhd \{\mathtt{true}\}, \mathtt{mg\_} \not\bowtie^{\mathtt{P}} (\mathtt{s}(\mathtt{mg\_t}^{\mathtt{N}}(\mathtt{X})), \mathtt{mg\_g}^{\mathtt{N}}(\mathtt{X})) \lhd \{\mathtt{true}\}, \mathtt{mg\_g}^{\mathtt{P}}(\mathtt{X}) \lhd \{\mathtt{true}\},$
$\mathtt{mg\_} \not\bowtie^{\mathtt{P}} (\mathtt{mg\_f}^{\mathtt{N}}(\mathtt{X}), \mathtt{0}) \lhd \{\mathtt{true}\}, \ldots\}.$

⊙ $\mathcal{H}^3_{\mathcal{PMG}} = \{\mathtt{f}(\mathtt{X}) \lhd \{\mathtt{s}(\bot)\}, \mathtt{g}(\mathtt{X}) \lhd \{\mathtt{0}\}, \mathtt{mg\_} \not\bowtie (\mathtt{s}(\mathtt{mg\_t}^{\mathtt{N}}(\mathtt{X})), \mathtt{0}) \lhd \{\mathtt{true}\}, \ldots\}.$

By following with the new magic transformation, this one is not the only modification w.r.t. [1]. Negative constraints can be satisfied not only by DC-clash but also when a failure value for functions appears. Failures of reduction can appear in the following cases: (a) failure of the condition of a rule, by rule (7) of *CRWLF*, or (b) failure in the parameter passing, by rule (8) of *CRWLF*. In the case (a), for the outermost functions, we have the same problems as logic programming, and the magic transformation must be modified in order to avoid that some magic functions cannot be evaluated to `true` due to the information passing including constraints with undefined functions. Our transformation will ensure that the magic functions are two-valued, that is, their *SAS*'s will include, at least, `true` or `F`, like in the following example:

Original Program — **Goal**: $: -\mathtt{f}(\mathtt{X},\mathtt{Y}) \not\bowtie \mathtt{s}(\mathtt{0}).$

(1) $\mathtt{f}(\mathtt{X}_1,\mathtt{X}_2) \rightarrow \mathtt{X}_2 \Leftarrow \mathtt{h} \bowtie \mathtt{0}, \mathtt{p}(\mathtt{X}_2) \bowtie \mathtt{s}(\mathtt{0}).$
(2) $\mathtt{h} \rightarrow \mathtt{h}.$
(3) $\mathtt{p}(\mathtt{0}) \rightarrow \mathtt{s}(\mathtt{s}(\mathtt{0})).$
(4) $\mathtt{p}(\mathtt{s}(\mathtt{0})) \rightarrow \mathtt{s}(\mathtt{0}).$

Transformed Program — **Goal**: $: -\mathtt{f}(\mathtt{X},\mathtt{Y}) \not\bowtie \mathtt{s}(\mathtt{0}).$

FilteredRules
(1) $\mathtt{f}(\mathtt{X}_1,\mathtt{X}_2) \rightarrow \mathtt{X}_2 \Leftarrow \mathtt{mg\_f}^{\mathtt{P}}(\mathtt{X}_1,\mathtt{X}_2) \bowtie \mathtt{true}, \mathtt{h} \bowtie \mathtt{0}, \mathtt{p}(\mathtt{X}_2) \bowtie \mathtt{s}(\mathtt{0}).$
(2) $\mathtt{h} \rightarrow \mathtt{h} \Leftarrow \mathtt{mg\_h}^{\mathtt{P}} \bowtie \mathtt{true}.$
(3) $\mathtt{p}(\mathtt{0}) \rightarrow \mathtt{s}(\mathtt{s}(\mathtt{0})) \Leftarrow \mathtt{mg\_p}^{\mathtt{P}}(\mathtt{0}) \bowtie \mathtt{true}.$
(4) $\mathtt{p}(\mathtt{s}(\mathtt{0})) \rightarrow \mathtt{s}(\mathtt{0}) \Leftarrow \mathtt{mg\_p}^{\mathtt{P}}(\mathtt{s}(\mathtt{0})) \bowtie \mathtt{true}.$

Magic Rules
(5) $\mathtt{mg\_} \not\bowtie^{\mathtt{P}} (\mathtt{mg\_f}^{\mathtt{N}}(\mathtt{X},\mathtt{Y}), \mathtt{s}(\mathtt{0})) \rightarrow \mathtt{true}.$
(6) $\mathtt{mg\_} \not\bowtie^{\mathtt{P}} (\mathtt{X}_4,\mathtt{X}_5) \rightarrow \mathtt{mg\_} \not\bowtie^{\mathtt{P}} (\mathtt{mg\_f}^{\mathtt{N}}(\mathtt{X}_3,\mathtt{X}_4),\mathtt{X}_5) \Leftarrow \mathtt{h} \bowtie \mathtt{0}, \mathtt{p}(\mathtt{X}_4) \bowtie \mathtt{s}(\mathtt{0}).$
(7) $\mathtt{mg\_} \bowtie^{\mathtt{P}} (\mathtt{mg\_h}^{\mathtt{N}},\mathtt{X}_6) \rightarrow \mathtt{mg\_} \bowtie^{\mathtt{P}} (\mathtt{mg\_h}^{\mathtt{N}},\mathtt{X}_6).$
(8) $\mathtt{mg\_} \bowtie^{\mathtt{P}} (\mathtt{s}(\mathtt{s}(\mathtt{0})),\mathtt{X}_7) \rightarrow \mathtt{mg\_} \bowtie^{\mathtt{P}} (\mathtt{mg\_p}^{\mathtt{N}}(\mathtt{0}),\mathtt{X}_7).$
(9) $\mathtt{mg\_} \bowtie^{\mathtt{P}} (\mathtt{s}(\mathtt{0}),\mathtt{X}_8) \rightarrow \mathtt{mg\_} \bowtie^{\mathtt{P}} (\mathtt{mg\_p}^{\mathtt{N}}(\mathtt{s}(\mathtt{0})),\mathtt{X}_8).$
(10) $\mathtt{mg\_f}^{\mathtt{P}}(\mathtt{X}_9,\mathtt{X}_{10}) \rightarrow \mathtt{mg\_} \not\bowtie^{\mathtt{P}} (\mathtt{mg\_f}^{\mathtt{N}}(\mathtt{X}_9,\mathtt{X}_{10}),\mathtt{X}_{11}).$
(11) $\mathtt{mg\_h}^{\mathtt{P}} \rightarrow \mathtt{mg\_} \bowtie^{\mathtt{P}} (\mathtt{mg\_h}^{\mathtt{N}},\mathtt{X}_{12}).$
(12) $\mathtt{mg\_p}^{\mathtt{P}}(\mathtt{X}_{13}) \rightarrow \mathtt{mg\_} \bowtie^{\mathtt{P}} (\mathtt{mg\_p}^{\mathtt{N}}(\mathtt{X}_{13}),\mathtt{X}_{14}).$
(13) $\mathtt{mg\_} \bowtie^{\mathtt{P}} (\mathtt{mg\_h}^{\mathtt{N}},\mathtt{0}) \rightarrow \mathtt{mg\_f}^{\mathtt{P}}(\mathtt{X}_{15},\mathtt{X}_{16}).$
(14) $\mathtt{mg\_} \bowtie^{\mathtt{P}} (\mathtt{mg\_p}^{\mathtt{N}}(\mathtt{X}_{18}),\mathtt{s}(\mathtt{0})) \rightarrow \mathtt{mg\_f}^{\mathtt{P}}(\mathtt{X}_{17},\mathtt{X}_{18}).$

In the previous example, the proposed goal has as answers $\mathtt{Y} = \mathtt{0}$ and $\mathtt{Y} = \mathtt{s}(\mathtt{s}(\mathtt{Z}))$, due to the failure in the instances $\mathtt{p}(\mathtt{0}) \bowtie \mathtt{s}(\mathtt{0})$ and $\mathtt{p}(\mathtt{s}(\mathtt{s}(\mathtt{Z}))) \bowtie \mathtt{s}(\mathtt{0})$ of the conditions of `f`.

The idea consists in detecting the outermost functions in negative constraints, and for each definition rule of these functions, to avoid the left-to-right sips (see rules **(13)** and **(14)** obtained from the rule **(1)** of the original program).

By considering the left-to-right sips presented in [1], the rule **(14)** would be replaced by the rule $\mathtt{mg\_} \bowtie^{\mathtt{P}} (\mathtt{mg\_p}^{\mathtt{N}}(\mathtt{X}_{18}),\mathtt{s}(\mathtt{0})) \rightarrow \mathtt{mg\_f}^{\mathtt{P}}(\mathtt{X}_{17},\mathtt{X}_{18}) \Leftarrow \mathtt{h} \bowtie \mathtt{0}$ taking into account the constraint $\mathtt{h} \bowtie \mathtt{0}$ for the information passing to $\mathtt{p}(\mathtt{X}_{18})$. Given that `h` has as unique *SAS* $\{\bot\}$, then the *SAS* $\mathtt{mg\_p}^{\mathtt{P}}(\mathtt{0}) \lhd \{\mathtt{true}\}$ cannot be

generated by using the previous rule and the rule **(12)**. Therefore, the bottom-up evaluation could not obtain the *SAS* $\mathtt{p(0)} \lhd \{\mathtt{s(s(0))}\}$ which allows to get the *SAS* $\mathtt{f(X,0)} \lhd \{\mathtt{F}\}$ which can be used to satisfy the goal. In this way, the evaluation of the program could not generate the answer $\mathtt{Y} = \mathtt{0}$, and thus the use of left-to-right sips produces uncompleteness w.r.t. *CRWLF*.

In the case (b), for the outermost functions, the failure of reduction will be computed by means of the fix-point operator (see definition 6). In the above example, the evaluation would compute the *SAS* $\mathtt{p}(\widehat{\mathtt{s(0)}}) \lhd \{\mathtt{F}\}$ [4] by the rule **(4)**, where $\widehat{\mathtt{s(0)}} =_{\mathtt{def}} \{\mathtt{0}, \mathtt{s(s(Z))}, \mathtt{F}\}$ allowing to generate the additional answer $\mathtt{Y} = \mathtt{s(s(Z))}$ by applying the rule **(1)** of the transformed program. The bottom-up evaluation for the above program is as follows:

$\odot\ \mathcal{H}^0_{\mathcal{P}^{\mathcal{MG}}} = \bot$
$\odot\ \mathcal{H}^1_{\mathcal{P}^{\mathcal{MG}}} = \{\mathtt{mg\_} \not\bowtie^{\mathtt{P}} (\mathtt{mg\_f^N(X,Y)}, \mathtt{s(0)}) \lhd \{\mathtt{true}\}, \mathtt{p}(\widehat{\mathtt{0}}) \lhd \{\mathtt{F}\}, \mathtt{p}(\widehat{\mathtt{s(0)}}) \lhd \{\mathtt{F}\}, \mathtt{p(0)} \lhd \{\mathtt{F}, \bot\},$
$\mathtt{p(s(0))} \lhd \{\mathtt{F}, \bot\}, \ldots\}.$
$\odot\ \mathcal{H}^2_{\mathcal{P}^{\mathcal{MG}}} = \{\mathtt{mg\_f^P(X,Y)} \lhd \{\mathtt{true}\}, \mathtt{f(X,s(s(Z)))} \lhd \{\mathtt{F}\}, \ldots\}.$
$\odot\ \mathcal{H}^3_{\mathcal{P}^{\mathcal{MG}}} = \{\mathtt{mg\_} \bowtie^{\mathtt{P}} (\mathtt{mg\_h^N}, \mathtt{0}) \lhd \{\mathtt{true}\}, \mathtt{mg\_} \bowtie^{\mathtt{P}} (\mathtt{mg\_p^N(Y)}, \mathtt{s(0)}) \lhd \{\mathtt{true}\}, \ldots\}.$
$\odot\ \mathcal{H}^4_{\mathcal{P}^{\mathcal{MG}}} = \{\mathtt{mg\_h^P} \lhd \{\mathtt{true}\}, \mathtt{mg\_p^P(Y)} \lhd \{\mathtt{true}\}, \mathtt{mg\_} \bowtie^{\mathtt{P}} (\mathtt{s(s(0))}, \mathtt{s(0)}) \lhd \{\mathtt{true}\},$
$\mathtt{mg\_} \bowtie^{\mathtt{P}} (\mathtt{s(0)}, \mathtt{s(0)}) \lhd \{\mathtt{true}\}, \ldots\}.$
$\odot\ \mathcal{H}^5_{\mathcal{P}^{\mathcal{MG}}} = \{\mathtt{p(0)} \lhd \{\mathtt{F}, \mathtt{s(s(0))}\}, \mathtt{p(s(0))} \lhd \{\mathtt{F}, \mathtt{s(0)}\}, \ldots\}.$
$\odot\ \mathcal{H}^6_{\mathcal{P}^{\mathcal{MG}}} = \{\mathtt{f(X,0)} \lhd \{\mathtt{F}\}, \ldots\}.$

and the instances of the goal $\mathtt{f(X,0)} \not\bowtie \mathtt{s(0)},\ \mathtt{f(X,s(s(Z)))} \not\bowtie \mathtt{s(0)}$ are satisfied in the Herbrand algebra.

However, it is not enough, given that the failures of reduction can also appear at the same cases due to inner functions, that is due to failure (c) in the condition and (d) in parameter passing of a rule, like in the following example.

Original Program **Goal**

(1) $\mathtt{f(X_1)} \to \mathtt{s(X_1)}.$ $: -\mathtt{f(g(X))} \not\bowtie \mathtt{s(0)}.$
(2) $\mathtt{g(s(0))} \to \mathtt{0} \Leftarrow \mathtt{p(0)} \bowtie \mathtt{0}.$
(3) $\mathtt{p(0)} \to \mathtt{s(s(0))}.$

Transformed Program **Goal**

$: -\mathtt{f(mg\_g^N(X))} \not\bowtie \mathtt{s(0)}.$

FilteredRules
(1) $\mathtt{f(X_1)} \to \mathtt{s(X_1)} \Leftarrow \mathtt{mg\_f^P(X_1)} \bowtie \mathtt{true}.$
(2) $\mathtt{g(s(0))} \to \mathtt{0} \Leftarrow \mathtt{mg\_g^P(s(0))} \bowtie \mathtt{true}, \mathtt{p(0)} \bowtie \mathtt{0}.$
(3) $\mathtt{p(0)} \to \mathtt{s(s(0))} \Leftarrow \mathtt{mg\_p^P(0)} \bowtie \mathtt{true}.$
Magic Rules
(4) $\mathtt{mg\_} \not\bowtie^{\mathtt{P}} (\mathtt{mg\_f^N(mg\_g^N(X))}, \mathtt{s(0)}) \to \mathtt{true}.$
(5) $\mathtt{mg\_} \not\bowtie^{\mathtt{P}} (\mathtt{s(X_2)}, \mathtt{X_3}) \to \mathtt{mg\_} \not\bowtie^{\mathtt{P}} (\mathtt{mg\_f^N(X_2)}, \mathtt{X_3}).$
(6) $\mathtt{mg\_} \bowtie^{\mathtt{P}} (\mathtt{s(s(0))}, \mathtt{X_4}) \to \mathtt{mg\_} \bowtie^{\mathtt{P}} (\mathtt{mg\_p^N(0)}, \mathtt{X_4}).$
(7) $\mathtt{mg\_} \bowtie^{\mathtt{P}} (\mathtt{mg\_p^N(0)}, \mathtt{0}) \to \mathtt{mg\_f^P(mg\_g^N(s(0)))}.$
(8) $\mathtt{mg\_} \not\bowtie^{\mathtt{P}} (\mathtt{mg\_f^N(0)}, \mathtt{X_5}) \to \mathtt{mg\_} \not\bowtie^{\mathtt{P}} (\mathtt{mg\_f^N(mg\_g^N(s(0)))}, \mathtt{X_5}) \Leftarrow \mathtt{p(0)} \bowtie \mathtt{0}.$
(9) $\mathtt{mg\_} \not\bowtie^{\mathtt{P}} (\mathtt{mg\_f^N(F)}, \mathtt{X_6}) \to \mathtt{mg\_} \not\bowtie^{\mathtt{P}} (\mathtt{mg\_f^N(mg\_g^N(0)}, \mathtt{X_6}).$
(10) $\mathtt{mg\_} \not\bowtie^{\mathtt{P}} (\mathtt{mg\_f^N(F)}, \mathtt{X_7}) \to \mathtt{mg\_} \not\bowtie^{\mathtt{P}} (\mathtt{mg\_f^N(mg\_g^N(s(s(Z)))}, \mathtt{X_7}).$
(11) $\mathtt{mg\_f^P(X_8)} \to \mathtt{mg\_} \not\bowtie^{\mathtt{P}} (\mathtt{mg\_f^N(X_8)}, \mathtt{X_9}).$
(12) $\mathtt{mg\_p^P(X_{10})} \to \mathtt{mg\_} \bowtie^{\mathtt{P}} (\mathtt{mg\_p^N(X_{10})}, \mathtt{X_{11}}).$
Goal Solving Rules
(13) $\mathtt{f(mg\_g^N(s(0)))} \to \mathtt{f(0)} \Leftarrow \mathtt{mg\_f^P(mg\_g^N(s(0)))} \bowtie \mathtt{true}.$
(14) $\mathtt{f(mg\_g^N(0))} \to \mathtt{f(F)} \Leftarrow \mathtt{mg\_f^P(mg\_g^N(0))} \bowtie \mathtt{true}.$
(15) $\mathtt{f(mg\_g^N(s(s(Z))))} \to \mathtt{f(F)} \Leftarrow \mathtt{mg\_f^P(mg\_g^N(s(s(Z))))} \bowtie \mathtt{true}.$

As you can see, we have the answers $\mathtt{X} = \mathtt{0},\ \mathtt{X} = \mathtt{s(0)}$ and $\mathtt{X} = \mathtt{s(s(Z))}$ for the proposed goal. In this example, the failure of reduction in $\mathtt{g(X)}$ which is nested by $\mathtt{f}$, is due to the reasons (c): failure in the condition of $\mathtt{g}$ ($\mathtt{p(0)} \bowtie \mathtt{0}$) and (d): failure in the parameter passing ($\mathtt{g(0)}$ and $\mathtt{g(s(s(Z)))}$ w.r.t. the head $\mathtt{g(s(0))}$ of the rule

[4] $\hat{t}$ denotes the set $\{t' \in \mathit{CTerm}_{\bot,\mathtt{F}} \mid t \textit{ and } t' \textit{ have a } DC \cup \{\mathtt{F}\}\text{-clash}\}$.

(2)). The case (c) is handled by the magic rules for the outermost functions since the condition of the rules for the inner functions will appear as condition of these magic rules (see rule **(8)**). This failure generates $\mathtt{mg\_} \not\bowtie (\mathtt{mg\_f^N(0), s(0)}) \lhd \{\mathtt{F}\}$ by applying the rule **(8)** which allows to obtain the answer $\mathtt{X = s(0)}$ by applying the rules **(11)**, **(1)** and **(13)**, respectively. The case (d) is handled by introducing rules for representing the failure of parameter passing of the nested functions. For instance the rules **(9)** and **(10)** are introduced by the rule **(2)** in order to obtain $\mathtt{f(mg\_g^N(0))} \lhd \{\mathtt{s(F)}\}$ and $\mathtt{f(mg\_g^N(s(s(Z))))} \lhd \{\mathtt{s(F)}\}$, by using also **(11)** and **(1)** and, finally **(14)** and **(15)**.

Next, we present an algorithm **Magic_Alg** for this transformation. It uses an auxiliary algorithm **Nesting** in order to generate nesting magic and goal solving rules. They are shown in tables 2 and 3, wherein

- $\bar{t}|_i$ represents the subterm of $\bar{t}$ at position i; $\bar{e}[e']_i$ represents $\bar{e}$ replacing the subexpressions at position i by e'; safe(φ) represents the subexpressions of φ out of the scope of a function.
- e^N is defined as $X^N =_{def} X$, $c(\bar{e})^N =_{def} c(\overline{e^N})$ and $f(\bar{e})^N =_{def} f(\overline{e^{mg^N}})$ and $X^{mg^N} =_{def} X$, $c(\bar{e})^{mg^N} =_{def} c(\overline{e^{mg^N}})$ and $f(\bar{e})^{mg^N} =_{def} mg\_f^N(\overline{e^{mg^N}})$; and e^P is defined as $X^P =_{def} X$, $c(\bar{e})^P =_{def} c(\overline{e^P})$ and $f(\bar{e})^P =_{def} mg\_f^P(\overline{e^{mg^N}})$.
- The functions f_{ind} are auxiliary ones added whenever f has nested constructors in the patterns of its rules.
- $\bar{e}$ represents the sequence of expressions to be considered; $h(\bar{t})$ is the head of a program rule; f is a function, representing that $\bar{e}$ is in the scope of f; the boolean *Nested?* is true whenever the parameter f has been input; M_g represents the computed set of magic rules; P_g represents the computed set of program rules; *ind* indicates the position of $\bar{e}$ in the scope of f; G represents a set of triples (f, g, i) whose meaning is that the function g is nested by f at position i; *pos_op* indicates the position of $\bar{e}$ in a constraint, which can be either left ($\bar{e} \diamondsuit e'$) or right ($e' \diamondsuit \bar{e}$) hand-side (it can take two values, 1 and 2); and *op* represents the operator which is being considered.

The algorithm is applied as follows where "_" denotes arguments not needed for the calling:

$Mg := \emptyset$; $Pg := \mathcal{P}$; $G := \emptyset$; $C := \emptyset$;
for every $e \diamondsuit e' \in \mathcal{G}$ **do**
  if $((\diamondsuit = \not\bowtie)$ **or** $(\diamondsuit = \lhd\rhd))$ **then** $M_g := M_g \cup \{\diamondsuit(e, e')^P \rightarrow true\}$;
    else $M_g := M_g \cup \{\diamondsuit(e, e')^P \rightarrow true \Leftarrow C^N\}$;
  endif
  Magic_Alg$(e, \_, \_, false, M_g, P_g, ind, G, 1, \diamondsuit, ((\diamondsuit \equiv \not\bowtie)$ **or** $(\diamondsuit \equiv \lhd\rhd)))$;
  Magic_Alg$(e', \_, \_, false, M_g, P_g, ind, G, 2, \diamondsuit, ((\diamondsuit \equiv \not\bowtie)$ **or** $(\diamondsuit \equiv \lhd\rhd)))$;
  $C := C \cup \{e \diamondsuit e'\}$
endfor

Once the algorithm has been applied, $\mathcal{P}^{\mathcal{MG}}$ consists of $P_g^P \cup M_g$, where $(f(\bar{t}) \rightarrow r \Leftarrow C)^P$ is of the form $f(\bar{t}) \rightarrow r \Leftarrow mg\_f^P(\bar{t}) \bowtie true, C^N$, and $\Sigma^{\mathcal{MG}}$ consists of $DC \cup DC^{\mathcal{MG}}$ and $FS \cup FS^{\mathcal{MG}}$ where $DC^{\mathcal{MG}}$ and $FS^{\mathcal{MG}}$ denote the set of nesting magic constructors and passing magic functions, respectively. $\mathcal{G}^{\mathcal{MG}}$ consists in the set of constraints $e^N \diamondsuit e'^N$ where $e \diamondsuit e' \in \mathcal{G}$.

5 Soundness, Completeness and Optimality Results

Our first result about our evaluation method establishes the equivalence among both the original and the transformed program w.r.t. the given goal.

Theorem 2 (Soundness and Completeness).
$\mathcal{P} \vdash_{CRWLF} \mathcal{G}\theta \Leftrightarrow \mathcal{P}^{\mathcal{MG}} \vdash_{CRWLF} \mathcal{G}^N\theta$.

The second result ensures optimality of our bottom-up evaluation method, in the sense that every computed *SAS* corresponds either with a subproof of some solution for the goal or with a function call from the goal.

We denote by $neg(\mathcal{P}, \mathcal{G})$ the subset of $outer(\mathcal{P}, \mathcal{G})$ containing the functions occurring in negative constraints.

Theorem 3 (Optimality).

- *If* $\mathcal{P}^{\mathcal{MG}} \vdash_{CRWLF} f(\bar{e})^N \lhd \mathcal{C}$, $\mathcal{C} \neq \{\bot\}$, *then* $f \in outer(\mathcal{P}, \mathcal{G})$ *and either:*
  - *there exist* θ *and a proof* $\mathcal{P} \vdash_{CRWLF} \mathcal{G}\theta$ *of minimal size and a subproof of the form* $\mathcal{P} \vdash_{CRWLF} f(\bar{e}) \lhd \mathcal{C}$, *or*
  - *there exist* $\mathcal{P}^{\mathcal{MG}} \vdash_{CRWLF} g(\bar{e'})^N \lhd \mathcal{C}'$, *a program rule instance* $g(\bar{t}) \to r \Leftarrow C, e \diamondsuit e', ... \in [R]_{\bot,\mathsf{F}}$, *and proofs* $\mathcal{P} \vdash_{CRWLF} e'_i \lhd \mathcal{C}_i$ *where* $t_i \in \mathcal{C}_i$, *and either a proof of minimal size* $\mathcal{P} \vdash_{CRWLF} e \lhd \mathcal{C}_e$ *or a proof of minimal size* $\mathcal{P} \vdash_{CRWLF} e' \lhd \mathcal{C}_{e'}$, *containing a subproof* $\mathcal{P} \vdash_{CRWLF} f(\bar{e}) \lhd \mathcal{C}$, *and if* $g \notin neg(\mathcal{P}, \mathcal{G})$ *then there exists a proof* $\mathcal{P} \vdash_{CRWLF} C$.
- *If* $\mathcal{P}^{\mathcal{MG}} \vdash_{CRWLF} f_k(\bar{e})^N \lhd \mathcal{C}$, *then there exists a proof* $\mathcal{P}^{\mathcal{MG}} \vdash_{CRWLF} f(\bar{e'}) \lhd \mathcal{C}$ *containing subproofs* $\mathcal{P}^{\mathcal{MG}} \vdash_{CRWLF} e_i \lhd \mathcal{C}_i$ *for some* $\mathcal{C}_i$.
- *If* $\mathcal{P}^{\mathcal{MG}} \vdash_{CRWLF} \diamondsuit(e, e')^P \lhd \mathcal{C}_0$, *then there exist proofs* $\mathcal{P}^{\mathcal{MG}} \vdash_{CRWLF} e^N \lhd \mathcal{C}$ *for some* $\mathcal{C}$ *and* $\mathcal{P}^{\mathcal{MG}} \vdash_{CRWLF} e'^N \lhd \mathcal{C}'$ *for some* $\mathcal{C}'$.
- *If* $\mathcal{P}^{\mathcal{MG}} \vdash_{CRWLF} f(\bar{e})^P \lhd \mathcal{C}_0$, *then there exists a proof* $\mathcal{P}^{\mathcal{MG}} \vdash_{CRWLF} f(\bar{e})^N \lhd \mathcal{C}$ *for some* $\mathcal{C}$.

6 Conclusions and Future Work

In this paper, we have presented a goal-directed bottom-up evaluation for functional-logic programs with negative information. By adopting the *CRWLF*-semantics [14] for our language, we have defined Herbrand models for this semantics and a fix-point operator which computes the Herbrand model of *CRWLF*-programs. Moreover, we have modified the magic transformation presented in [1] in order to handle negative constraints avoiding the known problems related to the magic transformations. As future work we go toward the incorporation of grouping and aggregation operators in our language, as well as the investigation of an extension of the relational algebra for it. Also, we are starting the implementation of this language, which will be based on the use of traditional indexing techniques.

Table 2. Magic Algorithm

```
Magic_Alg(in ē : tuple(Expression); in h(t̄) : Expression; in f : FunctionSymbol;
in Nested? : Bool; in/out M_g : Program; in/out P_g : Program; in/out ind : Index;
in/out G : set(tuple(FunctionSymbol, FunctionSymbol, Index)); in pos_op : Index;
in op : Operator; in Failure? : Bool)
var C' : Constraint;
if Nested? then
   ind := 0;
   for every e_1, ..., e_n do
   case e_i of
    X, X ∈ V :
      if (t̄|_j ≡ X) then
        for every ((h, k, j) ∈ G) do
          G := G ∪ {(f_ind, k, i)};
          Nesting(k, f, M_g, P_g, ind, i, G, pos_op, op, Failure?);
        endfor;
      endif;
    c(ē'), c ∈ DC^{MG} :
      for every f_ind(t̄) → r ⇐ C ∈ P_g do
        if (t_i ≡ c(t̄') and not (t̄' ≡ X̄ and X̄ ∩ (safe(r) ∪ safe(C)) = ∅)) then
         P_g := P_g ∪ {f_{ind+1}(t̄[t̄']_i) → r ⇐ C^N, f_ind(X̄[c(V̄)]_i)^N → f_{ind+1}(X̄[V̄]_i)^N};
         M_g := M_g ∪ {op(Y_1, Y_2[f_{ind+1}(X̄[V̄]_i)]_pos_op)^P → op(Y_1, Y_2[f_ind(X̄[c(V̄)]_i)]_pos_op)^P};
         M_g := M_g ∪ {f_{ind+1}(Ȳ)^P → op(Z_1, Z_2[f_{ind+1}(Ȳ)]_pos_op)^P};
         endif;
         if (t_i ∈ V and t_i ∈ safe(r) ∪ safe(C)) then
         P_g := P_g ∪ {f_{ind+1}(t̄) → r ⇐ C^N, f_ind(X̄[c(V̄)]_i)^N → f_{ind+1}(X̄[V̄]_i)^N};
         M_g := M_g ∪ {op(Y_1, Y_2[f_{ind+1}(X̄[V̄]_i)]_pos_op)^P → op(Y_1, Y_2[f_ind(X̄[c(V̄)]_i)]_pos_op)^P};
         M_g := M_g ∪ {f_{ind+1}(Ȳ)^P → op(Z_1, Z_2[f_{ind+1}(Ȳ)]_pos_op)^P};
        endif;
      endfor;
      ind := ind + 1;
      Magic_Alg(ē', h(t̄), f, true, M_g, P_g, ind, G, pos_op, op, Failure?);
    k(ē'), k ∈ FS :
      if ((f_ind, k, i) ∉ G) then
        G := G ∪ {(f_ind, k, i)}; Nesting(k, f, M_g, P_g, ind, i, G, pos_op, op, Failure?);
        Magic_Alg(mg_k^N(ē'), h(t̄), f, true, M_g, P_g, ind, G, pos_op, op, Failure?);
      endif;
   endcase;
  endfor;
else
   for every e_1, ..., e_n do
   case e_i of
    c(ē'), c ∈ DC^m :
      M_g := M_g ∪ {op(X_j, Y_j)^P → op(c(X̄), c(Ȳ))^P, 1 ≤ j ≤ m};
      Magic_Alg(ē', h(t̄), −, false, M_g, P_g, ind, G, pos_op, op, Failure?);
    k(ē'), k ∈ FS :
      M_g := M_g ∪ {k(Ȳ)^P → op(X_1, X_2[k(Ȳ)]_pos_op)^P};
      Magic_Alg(ē', h(t̄), k, true, M_g, P_g, ind, G, pos_op, op, Failure?);
      for every k(s̄) → r ⇐ C ∈ P_g do
        M_g := M_g ∪ {op(X_1, X_2[r]_pos_op)^P → op(X_1, X_2[k(s̄)]_pos_op)^P ⇐ C^N};
        Magic_Alg(r, k(s̄), −, false, M_g, P_g, ind, G, pos_op, op, Failure?);
        C' := ∅;
        for every e◇e' ∈ C do
          if Failure? then M_g := M_g ∪ {◇(e, e')^P → k(s̄)^P};
                      else M_g := M_g ∪ {◇(e, e')^P → k(s̄)^P ⇐ C'^N};
          endif
            Magic_Alg(e, k(s̄), −, false, M_g, P_g, ind, G, 1, ◇, ((◇ ≡⋈) or (◇ ≡<>)));
            Magic_Alg(e', k(s̄), −, false, M_g, P_g, ind, G, 2, ◇, ((◇ ≡⋈) or (◇ ≡<>)));
            C' := C' ∪ {e◇e'};
        endfor;
      endfor;
   endcase;
   endfor;
endif;
```

Table 3. Nesting Transformation

Nesting(**in** k : FunctionSymbol; **in** f : FunctionSymbol; **in/out** M_g : Program; **in/out** P_g : Program; **in/out** ind : Index; **in** i : Index; **in/out** G : set(tuple(FunctionSymbol, FunctionSymbol, Index)); **in** pos_op : Index; **in** op : Operator; **in** Failure? : Bool)
var C'' : Constraint;
for every $f_{ind}(\bar{t}) \rightarrow r \Leftarrow C \in P_g$ **do**
 if $(t_i \notin \mathcal{V})$ **or** $((t_i \in \mathcal{V})$ **and** $(t_i \in safe(r) \cup safe(C)))$ **then**
  for every $k(\bar{s}) \rightarrow r' \Leftarrow C' \in P_g$ **do**
   $M_g := M_g \cup \{op(Y_1, Y_2[f_{ind}(\bar{X}[r']_i)]_{pos\_op})^P \rightarrow op(Y_1, Y_2[f_{ind}(\bar{X}[k(\bar{s})]_i)]_{pos\_op})^P \Leftarrow C'\}$;
   $M_g := M_g \cup \{op(Y_1, Y_2[f_{ind}(\bar{X}[F]_i)]_{pos\_op})^P \rightarrow op(Y_1, Y_2[f_{ind}(\bar{X}[k(\bar{s})]_i)]_{pos\_op})^P \Leftarrow C'\}$;
   $P_g := P_g \cup \{f_{ind}(\bar{X}[k(\bar{s})]_i)^N \rightarrow f_{ind}(\bar{X}[r']_i)^N \Leftarrow C'\}$;
   $P_g := P_g \cup \{f_{ind}(\bar{X}[k(\bar{s})]_i)^N \rightarrow f_{ind}(\bar{X}[F]_i)^N \Leftarrow C'\}$;
   Magic_Alg(r', $k(\bar{s})$, true, f, M_g, P_g, ind, G, pos_op, op, Failure?);
   $C'' := \emptyset$;
   for every $e \diamondsuit e' \in C'$ **do**
    if Failure? **then** $M_g := M_g \cup \{\diamondsuit(f_{ind}(\bar{X}[e]_i), f_{ind}(\bar{X}[e']_i))^P \rightarrow f_{ind}(\bar{X}[k(\bar{s})]_i)^P\}$;
     else $M_g := M_g \cup \{\diamondsuit(f_{ind}(\bar{X}[e]_i), f_{ind}(\bar{X}[e']_i))^P \rightarrow f_{ind}(\bar{X}[k(\bar{s})]_i)^P \Leftarrow C''^N\}$;
    endif;
     Magic_Alg(e, $f_{ind}(\bar{X}[k(\bar{s})]_i)$, $-$, false, M_g, P_g, ind, G, 1, $\diamondsuit$, $((\diamondsuit \equiv \not\rhd\lhd)$ **or** $(\diamondsuit \equiv <\!\!\diamond\!\!>)))$;
     Magic_Alg(e', $f_{ind}(\bar{X}[k(\bar{s})]_i)$, $-$, false, M_g, P_g, ind, G, 2, $\diamondsuit$, $((\diamondsuit \equiv \not\rhd\lhd)$ **or** $(\diamondsuit \equiv <\!\!\diamond\!\!>)))$;
     $C'' := C'' \cup \{e \diamondsuit e'\}$;
    endfor;
   endif;
  endfor;
 endif;
endfor;

References

1. J. M. Almendros-Jiménez and A. Becerra-Terón. A Framework for Goal-Directed Bottom-Up Evaluation of Functional Logic Programs. In *Proc. FLOPS'01*, LNCS 2024, pages 153–169.
2. J. M. Almendros-Jiménez, A. Becerra-Terón, and J. Sánchez-Hernández. A Computational Model for Functional Logic Deductive Databases, available in `http://www.ual.es/~jalmen`. Technical report, Universidad de Almería, 2001.
3. K. R. Apt. Logic programming. In *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 10, pages 493–574. MIT Press, 1990.
4. K. R. Apt and R. N. Bol. Logic Programming and Negation: A Survey. *JLP*, 19,20:9–71, 1994.
5. C. Beeri and R. Ramakrishnan. On the Power of Magic. *JLP*, 10(3,4):255–299, 1991.
6. M. Gelfond and V. Lifschitz. The Stable Model Semantics for Logic Programming. In *Proc. ICLP/SLP'88*, pages 1070–1080. MIT Press.
7. J. C. González-Moreno, M. T. Hortalá-González, F. López-Fraguas, and M. Rodríguez-Artalejo. An Approach to Declarative Programming Based on a Rewriting Logic. *JLP*, 1(40):47–87, 1999.
8. M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *JLP*, 19,20:583–628, 1994.
9. M. Hanus. Curry, An Integrated Functional Logic Language, Version 0.7.1. Technical report, University of Kiel, Germany, June 2000.
10. D. B. Kemp, D. Srivastava, and P. J. Stuckey. Bottom-up Evaluation and Query Optimization of Well-Founded Models. *TCS*, 146(1,2):145–184, 1995.

11. R. Loogen, F. J. López-Fraguas, and M. Rodríguez-Artalejo. A Demand Driven Computation Strategy for Lazy Narrowing. In *Proc. PLILP'93*, LNCS 714, pages 184–200.
12. F. J. López-Fraguas and J. Sánchez-Hernández. Disequalities may help to Narrow. In *Proc. APPIA-GULP-PRODE'99*, pages 89–104.
13. F. J. López-Fraguas and J. Sánchez-Hernández. $\mathcal{TOY}$: A Multiparadigm Declarative System. In *Proc. RTA'99*, LNCS 1631, pages 244–247.
14. F. J. López-Fraguas and J. Sánchez-Hernández. Proving Failure in Functional Logic Programs. In *Proc. CL'00*, LNCS 1861, pages 179–193.
15. R. Ramakrishnan. Magic Templates: A Spellbinding Approach to Logic Programs. *JLP*, 11(3,4):189–216, 1991.
16. R. Ramakrishnan, D. Srivastava, S. Sudarshan, and P. Seshadir. The CORAL Deductive Database System. *VLDB*, 3(2):161–210, 1994.
17. R. Ramakrishnan and J. Ullman. A Survey of Deductive Database Systems. *JLP*, 23(2):125–149, 1995.
18. K. Ross. Modular Stratification and Magic Sets for Datalog Programs with Negation. *ACM*, 41(6):1216–1266, 1994.
19. J. D. Ullman. Bottom-up Beats Top-down for Datalog. In *Procs. PODS'89*, pages 140–149. ACM Press.
20. J. Vaghani, K. Ramamohanarao, D. B. Kemp, Z. Somogyi, P. J. Stuckey, T. S. Leask, and J. Harland. The Aditi Deductive Database System. *VLDB*, 3(2):245–288, 1994.
21. A. van Gelder, K. Ross, and J. Schlipf. The Well-Founded Semantics for General Logic Programs. *ACM*, 38(3):620–650, 1991.

A Logic Programming Approach to the Integration, Repairing and Querying of Inconsistent Databases*

Gianluigi Greco, Sergio Greco, and Ester Zumpano

DEIS, Università della Calabria, 87030 Rende, Italy
{ggreco,greco,zumpano}@si.deis.unical.it

Abstract. In this paper we present a logic programming based framework for the integration of possibly inconsistent databases. In particular we consider the problem of 'merging' databases and, since the resulting 'merged' database may be inconsistent (with respect to the constraints defined on the input databases or with respect to additional constraints), we address the problem of managing inconsistent databases. We propose a general logic framework for computing repairs and consistent answers over inconsistent databases. The logic framework and the techniques for computing repairs and consistent answers proposed here are more general than previously proposed techniques. Indeed, our technique is sound and complete for universally quantified constraints whereas previous defined techniques only consider restricted cases.

1 Introduction

The aim of data integration is to provide a uniform integrated access to multiple heterogeneous information sources, which were designed independently for autonomous applications and whose contents are strictly related. The integration of knowledge from multiple sources is an important aspect in several areas such as data warehousing, database integration, automated reasoning systems, active reactive databases and others. However, the database obtained from the merging of different sources could contain inconsistent data. The following example shows a typical case of inconsistency.

Example 1. Consider the database consisting of the single binary relation *Teaches(Course, Professor)* where the attribute *Course* is a key for the relation. Assume there are two different instances for the relations *Teaches*: $D_1 = \{Teaches(c_1, p_1), Teaches(c_2, p_2)\}$ and $D_2 = \{Teaches(c_1, p_1), Teaches(c_2, p_3)\}$.

The two instances satisfy the constraint that *Course* is a key but, from their union we derive a relation which does not satisfy the constraint since there are two distinct tuples with the same value for the attribute *Course*. □

* Work partially supported by MURST grants under the projects "Data-X" and "D2I". The second author is also supported by ISI-CNR.

P. Codognet (Ed.): ICLP 2001, LNCS 2237, pp. 348–364, 2001.

In the integration of two conflicting databases simple solutions could be based on the definition of preference criteria such as a partial order on the source information or majority criteria [18]. However, these solutions are not generally satisfactory and more useful solutions are those based on 1) the computation of 'repairs' for the database, 2) the computation of consistent answers [2].

The computation of repairs is based on the insertion and deletion of tuples so that the resulting database satisfies all constraints, whereas the computation of consistent answers is based on the identification of tuples satisfying integrity constraints and on the selection of tuples matching the goal. For instance, for the integrated database of Example 1, we have two alternative repairs consisting in the deletion of one of the tuples (c_2, p_2) and (c_2, p_3). The consistent answer to a query over the relation $Teaches$ contains the unique tuple (c_1, p_1) so that we don't know exactly which professor teaches course c_2.

In this paper, we focus our attention on the integration of conflicting instances [1,2,7] related to the same concept and possibly coming from different sources. We introduce two operators, called *Merge operator* and *Prioritized Merge operator*, which allow us to combine data coming from different sources. Moreover, since the resulting 'merged' database may be inconsistent (with respect to the constraints defined on the input databases or with respect to additional constraints), we address the problem of managing inconsistent databases.

We propose a general logic framework for computing repairs and consistent answers over inconsistent databases. Our technique is based on the rewriting of the different types of constraints into (prioritized) extended disjunctive rules with two different forms of negation: negation as failure and classical negation. The disjunctive program can be used both to generate 'repairs' for the database, i.e. minimal sets of insert and delete operations which make the database consistent, and to produce consistent answers, i.e. maximal sets of atoms which do not violate the constraints. Our technique is more general than techniques previously proposed, and it is sound and complete as each preferred stable model defines a repair and each repair is derived from a preferred stable model.

Recently there have been several proposals considering the problem of managing inconsistent databases. The problem has been deeply investigated mainly in the areas of databases and artificial intelligence [1,2,4,7,18,19,21]. A technique based on the rewriting of (single head) integrity constraints into logic rules has been proposed in [12], whereas [3] proposed a technique for the rewriting of binary constraints into logic (non disjunctive) rules.

2 Background

We assume familiarity with disjunctive logic program and disjunctive deductive databases [8,9] and recall here non standard definitions used in the paper.

Extended disjunctive databases. *Extended Datalog* programs extend standard Datalog programs with a different form of negation, known as *classical* or *strong negation*, which can also appear in the head of rules [9,11,15]. An ex-

tended atom is either an atom, say A or its negation $\neg A$. An extended Datalog program is a set of rules of the form

$$A_0 \vee ... \vee A_k \leftarrow B_1, ..., B_m, not\ B_{m+1}, ..., not\ B_n \qquad k + n > 0$$

where $A_0, ..., A_k, B_1, ..., B_n$ are extended atoms. A 2-valued interpretation I for an extended program $\mathcal{P}$ is a pair $\langle T, F\rangle$ where T and F define a partition of $\mathcal{B}_\mathcal{P} \cup \neg\mathcal{B}_\mathcal{P}$ and $\neg\mathcal{B}_\mathcal{P} = \{\neg A | A \in \mathcal{B}_\mathcal{P}$ (the Herbrand base of $\mathcal{P})\}$. An interpretation $I = \langle T, F\rangle$ is *consistent* if there is no atom A such that $A \in T$ and $\neg A \in T$. The semantics of an extended program $\mathcal{P}$ is defined by considering each negated predicate symbol, say $\neg p$, as a new symbol syntactically different from p and by adding to the program, for each predicate symbol p with arity n, the constraint $\leftarrow p(X_1, ..., X_n), \neg p(X_1, ..., X_n)$. In the following, for the sake of simplicity, we shall also use rules whose bodies may contain disjunctions.

Queries. Predicate symbols are partitioned into two distinct sets: *base predicates* and *derived predicates*. Base predicates correspond to database relations and they do not appear in the head of any rule, whereas derived predicates are defined by means of rules. Given a database D, a predicate symbol r and a program $\mathcal{P}$, $D(r)$ denotes the set of r-tuples in D whereas $\mathcal{P}_D$ denotes the program derived from the union of $\mathcal{P}$ with the tuples in D, i.e. $\mathcal{P}_D = \mathcal{P} \cup \{r(t) \leftarrow\ \ | \ t \in D(r)\}$. In the following a tuple t of a relation r will also be denoted as a fact $r(t)$. The semantics of $\mathcal{P}_D$ is given by the set of its stable models by considering either their union (*possible semantics* or *brave reasoning*) or their intersection (*certain semantics* or *cautious reasoning*). A *query* Q is a pair $(g, \mathcal{P})$ where g is a predicate symbol, called the *query goal*, and $\mathcal{P}$ is a program. The application of a query Q to a database D will be denoted by $Q(D)$. The answer to a query $Q = (g, \mathcal{P})$ over a database D, under the possible (resp. certain) semantics, is given by $D'(g)$ where $D' = \bigcup_{M \in SM(\mathcal{P}_D)} M$ (resp. $D' = \bigcap_{M \in SM(\mathcal{P}_D)} M$).

Integrity constraints. Database schemata contain the knowledge on the structure of data, i.e. they define constraints on the form the data must have. Integrity constraints express semantic information on data, that is relationships that must hold among data in the theory and they are mainly used to validate database transactions. They are usually defined by first order rules or by means of special notations for particular classes such as keys and functional dependencies.

Definition 1. An *integrity constraint* (or *embedded dependency*) is a formula of the first order predicate calculus of the form: $(\forall X)\ [\ \Phi(X) \supset (\exists Z)\Psi(Y)\ ]$ where Φ and Ψ are two conjunctions of literals, X and Y are the distinct sets of variables appearing in Φ and Ψ respectively, and $Z = X - Y$ is the set of variables existentially quantified. □

In the above definition the conjunction Φ is called the *body* and the conjunction Ψ the *head* of the integrity constraint. Most of the dependencies developed in database theory are restricted cases of some of the above classes. For instance, functional dependencies are positive, full, single-head, unirelational, equality-generating constraints [14].

In the rest of the paper we concentrate on *full* (or *universal*) *single-head* constraints, where Ψ is a literal or a conjunction of built-in literals (i.e. comparison operators). Therefore, an integrity constraint is a formula of the form: $(\forall X)\ [\ B_1 \wedge \cdots \wedge B_n \wedge not\ A_1 \wedge \cdots \wedge not\ A_m \wedge \phi \supset A_0\ ]$ where $A_1, ..., A_m, B_1, ..., B_n$ are base positive literals, ϕ is a conjunction of built-in literals, A_0 is a base positive atom or a conjunction of built-in atoms, X denotes the list of all variables appearing in $B_1, ..., B_n$; moreover the variables appearing in $A_0, ..., A_m$ and ϕ, also appear in $B_1, ..., B_n$. Often we shall write our constraints in a different format by moving literals from the head to the body and vice versa. For instance, the above constraint could be rewritten as $(\forall X)\ [\ B_1 \wedge \cdots \wedge B_n \wedge \phi \supset A_0 \vee A_1 \vee \cdots \vee A_m\ ]$ or in the form of a rule with empty head, called *denial*: $(\forall X)\ [\ B_1 \wedge \cdots \wedge B_n \wedge not\ A_0 \wedge not\ A_1 \wedge \cdots \wedge not\ A_m \wedge \phi \supset\ ]$ which is satisfied only if the body is false.

3 Database Integration

Integrating data from different sources consists of two main steps: the first in which the various relations are merged together and the second in which some tuples are removed (or inserted) from the resulting database in order to ensure some *integrity* constraints. Before formally introducing the database integration problem let us recall some basic definitions and notations.

Let R be a relation name, then we denote by: i) *attr(R)* the set of attributes of R, ii) *key(R)* the set of attributes in the primary key of R, iii) $fd(R)$ the set of functional dependencies of R, and iv) *inst(R)* the instance of R (set of tuples). Given a tuple $t \in inst(R)$, $key(t)$ denotes the values of the key attributes of t whereas, for a given database D, $fd(D)$ denotes the set of functional dependencies of D and *inst(D)* denotes the database instance.

We assume that relations associated with the same concept have been homogenized with respect to a common ontology, so that attributes denoting the same concepts have the same name [23]. We say that two homogenized relations R and S, associated with the same concept, are *overlapping* if $key(R) = key(S)$.

The database integration problem is as follows: given two databases $D_1 = \{R_1, ..., R_k\}$ and $D_2 = \{S_1, ..., S_k\}$ where R_i and S_i refer to the same concept and a set of integrity constraints $\mathcal{IC}$, computes a database $D = \{T_1, ..., T_k\}$, where each T_i, derived from R_i and S_i is such that $inst(D) \models \mathcal{IC}$. It is important to note that $\mathcal{IC}$ denotes a set of constraints which must be satisfied by the integrated database and it may be different and more general than constraints defined on the input databases.

Since the integrated database D does not generally satisfy all integrity constraints in $\mathcal{IC}$, we first compute a database D' by 'merging' the input databases and then compute repairs which make the database consistent. Alternatively, we leave the integrated database inconsistent and compute answers by selecting only tuples satisfying the constraints. Therefore, in the integration of databases we deal with three different problems: 1) merging of the input databases, 2) com-

puting repairs for the possible inconsistent merged database, and 3) computing consistent answers to queries. These problems will next be addressed.

Database merging. The first step in the integration of two databases $D_1 = \{R_1, ..., R_k\}$ and $D_2 = \{S_1, ..., S_k\}$ is the merging of two different data sources; this can be done by means of a merging operator applied to D_1 and D_2. The resulting database $D = \{T_1, ..., T_k\}$ is obtained by merging relations associated with the same concept, that is $T_i = R_i \diamond S_i$ for $1 \leq i \leq k$.

Definition 2. Given two, not necessarily distinct, relations R and S such that $attr(R) \subseteq attr(S)$ and two tuples $t_1 \in inst(R)$ and $t_2 \in inst(S)$, we say that t_1 is *less informative* than t_2 ($t_1 \ll t_2$) if for each attribute a in $attr(R)$, $t_1[A] = t_2[A]$ or $t_1[A] = \perp$, where $\perp$ denotes the null value. Moreover, given two relations R and S, we say that $R \ll S$ if $\forall t_1 \in inst(R)\ \exists t_2 \in inst(S)$ s.t. $t_1 \ll t_2$. □

Definition 3. Let R and S be two relations, a binary operator $\diamond$ such that:

1. $attr(R \diamond S) = attr(R) \cup attr(S)$,
2. $R \bowtie S \ll R \diamond S$.

is called *merge operator*. Moreover, it is said to be

- *lossless*, if for all $inst(R)$ and $inst(S)$, $R \ll (R \diamond S)$ and $S \ll (R \diamond S)$;
- *dependency preserving*, if for all $inst(R)$ and $inst(S)$, is $(R \diamond S) \models (fd(R) \cap fd(S))$. □

Observe that if a merge operator is dependency preserving, the resulting relation is consistent with respect to the functional dependencies defined on the input relations; this happens because the key constraints belong to $fd(R) \cap fd(S)$. For this reason, in the following, we only consider lossless operators and introduce preferences (i.e. a partial order) according to the functional dependencies of the source databases which are preserved in the merged database.

Definition 4. Given two lossless merge operators $\diamond_1$ and $\diamond_2$, we say that i) $\diamond_1$ is *content preferable* to $\diamond_2$ ($\diamond_1 \prec_C \diamond_2$) if, for all R and S, $|R \diamond_1 S| < |R \diamond_2 S|$, and ii) $\diamond_1$ is *dependency preferable* to $\diamond_2$ ($\diamond_1 \prec_{FD} \diamond_2$) if, for all R and S, the number of tuples in $(R \diamond_1 S)$ which violate the $fd(R) \cap fd(S)$ are less than the number of tuples in $(R \diamond_2 S)$ which violate the $fd(R) \cap fd(S)$. □

Repairing inconsistent databases. Once the logical conflicts due to the schema heterogeneity have been resolved, conflicts may arise, during the integration process, among instances provided by different sources. In particular, the same real-world object may correspond to many tuples (possibly residing in different overlapping relations), that may have the same value for the key attributes but different values for some non-key attribute.

Let us first introduce the formal definition of consistent database and repairs.

Definition 5. Given a database D and a set of integrity constraint $\mathcal{IC}$ on D, we say that D is *consistent* if $D \models \mathcal{IC}$, i.e. if all integrity constraints in $\mathcal{IC}$ are satisfied by D, otherwise it is *inconsistent*. □

Definition 6. Given a (possibly inconsistent) database D, a *repair* for D is a pair of sets of atoms (R^+, R^-) such that 1) $R^+ \cap R^- = \emptyset$, 2) $D \cup R^+ - R^- \models \mathcal{IC}$ and 3) there is no pair $(S^+, S^-) \neq (R^+, R^-)$ such that $S^+ \subseteq R^+$, $S^- \subseteq R^-$ and $D \cup S^+ - S^- \models \mathcal{IC}$. The database $D \cup R^+ - R^-$ will be called the *repaired database*. □

Thus, repaired databases are consistent databases which are derived from the source database by means of a minimal set of insertion and deletion of tuples. Given a repair R, R^+ denotes the set of tuples which will be added to the database, whereas R^- denotes the set of tuples of D which will be deleted. In the following, for a given repair R and a database D, $R(D) = D \cup R^+ - R^-$ denotes the application of R to D.

Example 2. Assume we are given a database $D = \{p(a), p(b), q(a), q(c)\}$ with the *inclusion dependency* $(\forall X)\ [\ p(X) \supset q(X)\ ]$. D is inconsistent since $p(b) \supset q(b)$ is not satisfied. The repairs for D are $R_1 = (\{q(b)\}, \emptyset)$ and $R_2 = (\emptyset, \{p(b)\})$ producing, respectively, the repaired databases $R_1(D) = \{p(a), p(b), q(a), q(c), q(b)\}$ and $R_2(D) = \{p(a), q(a), q(c)\}$. □

Querying inconsistent databases. A (relational) query over a database defines a function from the database to a relation. It can be expressed by means of alternative equivalent languages such as relational algebra, 'safe' relational calculus or 'safe' non recursive Datalog [22]. In the following we shall use Datalog. Thus, a query is a pair $(g, \mathcal{P})$ where $\mathcal{P}$ is a safe non-recursive Datalog program and g is a predicate symbol specifying the output (derived) relation. Observe that relational queries define a restricted case of disjunctive queries. The reason for considering relational and disjunctive queries is that, as we shall show next, relational queries over databases with constraints can be rewritten into extended disjunctive queries over databases without constraints.

Definition 7. Given a database D and a set of integrity constraints $\mathcal{IC}$, an atom A is true (resp. false) with respect to $\mathcal{IC}$ if A belongs to all repaired databases (resp. there is no repaired database containing A). The set of atoms which are neither true nor false is undefined. □

Thus, true atoms appear in all repaired databases whereas undefined atoms appear in a proper subset of repaired databases. Given a database D and a set of integrity constraints $\mathcal{IC}$, the application of $\mathcal{IC}$ to D, denoted by $\mathcal{IC}(D)$, defines three distinct sets of atoms: the set of true atoms $\mathcal{IC}(D)^+$, the set of undefined atoms $\mathcal{IC}(D)^u$ and the set of false atoms $\mathcal{IC}(D)^-$.

Definition 8. Given a database D and a query $Q = (g, \mathcal{P})$, the *consistent answer* of the query Q on the database D, denoted as $Q(D, \mathcal{IC})$, gives three sets, denoted as $Q(D, \mathcal{IC})^+$, $Q(D, \mathcal{IC})^-$ and $Q(D, \mathcal{IC})^u$. These contain, respectively, the sets of g-tuples which are *true* (i.e. belonging to $Q(D')$ for all repaired databases D'), *false* (i.e. not belonging to $Q(D')$ for all repaired databases D') and *undefined* (i.e. set of tuples which are neither true nor false). □

4 A Logic Programming Approach

The aim of this section is to describe how the integration process (i.e. database merging, specifying integrity constraints, repairing and querying inconsistent databases) can be modeled by means of a logic program. In particular, we show that the merging of databases can be performed by means of a (stratified) Datalog program and that the computation of repairs and consistent answers can be thought of as a set of rules for integrating some sources; moreover, we also show that every disjunctive program can be carried out by rewriting constraints into logic rules.

4.1 Merging Databases

We start by introducing two merging operators and a function which given two relations R and S, replaces null values appearing in the tuple of R with values of the related tuples in S.

In more detail, given two relations R and S such that $attr(R) \subseteq attr(S)$, the operator Θ is defined as follows:

$$\Theta(R,S) = \{t \in R \mid \nexists t_1 \in S \ s.t. \ key(t) = key(t_1)\ \} \cup \{t \mid \exists t_1 \in R, \exists t_2 \in S \ s.t.$$

$$\forall a \in attr(R)\ (\ t[a] = \begin{cases} t_2[a] & \text{if } (a \in attr(S) \wedge t_1[a] = \bot) \\ t_1[a] & \text{otherwise} \end{cases}\ )\}$$

The Merge Operator. Given two overlapping relations R and S, the *merge operator*, denoted by $\boxtimes$, integrates the information provided by R and S. Let $T = R \boxtimes S$, then the schema of T contains the union of the attributes in R and S, and its instance is obtained by completing the information coming from each input relation with that coming from the other one.

Definition 9. Let R and S be two overlapping relations. The *merge operator* is a binary operator defined as follows:

$$R \boxtimes S = \Theta(R \supset\!\bowtie S, S) \cup \Theta(R \bowtie\!\subset S, R) \qquad \square$$

$R \boxtimes S$ computes the full outer join and extends tuples coming from R (resp. S) with the values of tuples of S (resp. R) having the same key. The extension of a tuple is carried out by the operator Θ which replaces null values appearing in a given tuple of the first relation with values appearing in some correlated tuple of the second relation. Thus, the merge operator applied to two relations R and S 'extends' the content of tuples of both R and S.

Example 3. Consider the relations R and S with schemata $(K, Title, Author)$ and $(K, Title, Author, Year)$ where K is the key of both relations. Assuming that the instances of R and S consist of the following facts

$R(1, Moon, Greg)$	$S(3, Flowers, Smith, 1965)$
$R(2, Money, Jones)$	$S(4, Sea, Taylor, 1971)$
$R(3, Sky, Jones)$	$S(5, Sun, Steven, 1980)$

the relation $T = R \boxtimes S$ is as follows:

$T(1, Moon, Greg, \perp)$ $\quad$ $T(3, Flowers, Smith, 1965)$
$T(2, Money, Jones, \perp)$ $\quad$ $T(4, Sea, Taylor, 1971)$
$T(3, Sky, Jones, 1965)$ $\quad$ $T(5, Sun, Steven, 1980)$ □

The Prioritized Merge Operator. In order to satisfy preference constraints, we introduce an asymmetric merge operator, called *prioritized merge operator*, which gives preference to data coming from the left relation when conflicting tuples are detected.

Definition 10. Let R and S be two overlapping relations and let $S' = S \rtimes (\pi_{key(S)} S - \pi_{key(R)} R)$ be the set of tuples in S not joining with any tuple in R. The *prioritized merge operator* is defined as follows:

$$R \lhd S = \Theta(R \rhd\!\bowtie S, S) \cup (R \bowtie\!\lhd S') \qquad \square$$

The prioritized merge operator includes all tuples of the left relation and only the tuples of the right relation whose key does not identify any tuple in the left relation. Moreover, only tuples 'coming' from the left relation are extended since tuples coming from the right relation, which join tuples coming from the left relation, are not included. Thus, when integrating relations conflicting on the key attributes, the prioritized merge operator gives preference to the tuples of the left side relation and completes them with values taken from the right side relation.

Example 4. Consider the relations R and S of Example 3. The relation $T = R \lhd S$ is as follows:

$T(1, Moon, Greg, \perp)$ $\quad$ $T(4, Sea, Taylor, 1971)$
$T(2, Money, Jones, \perp)$ $\quad$ $T(5, Sun, Steven, 1980)$
$T(3, Sky, Jones, 1965)$ □

The above two merge operators can be easily defined by means of a (non recursive) Datalog program.

Definition 11. Let R and S be two relations and let $K = key(R) = key(S)$, $A = attr(R) \cap attr(S)$, $B = attr(R) - attr(S)$ and $C = attr(S) - attr(R)$ be sets of attributes. The integrated relation $P = R \boxtimes S$ is defined by the following program

$$p(K, A, B, C) \leftarrow r(K, A, B),\ s(K, A, C)$$
$$p(K, A, B, C) \leftarrow r(K, A', B),\ not\, s(K, A', C'),\ s(K, A'', C),\ extend(A', A'', A)$$
$$p(K, A, B, C) \leftarrow s(K, A', C),\ not\, r(K, A', B'),\ r(K, A'', B),\ extend(A', A'', A)$$

where

$$extend([\,], [\,], [\,])$$
$$extend([A|L_1], [B|L_2], [C|L_3]) \leftarrow max(A, B, C),\ extend(L_1, L_2, L_3)$$

$$max(A, B, A) \leftarrow A \neq \perp$$
$$max(A, B, B) \leftarrow A = \perp$$ □

Also the prioritized merge operation, introduced in Definition 10 can be easily expressed by means of a logic program.

Definition 12. Let R and S be two relations and let $K = key(R) = key(S)$, $A = attr(R) \cap attr(S)$, $B = attr(R) - attr(S)$ and $C = attr(S) - attr(R)$ be sets of attributes. The integrated relation $P = R \lhd S$ is defined by the following program

$$p(K, A, B, C) \leftarrow r(K, A, B),\ s(K, A, C)$$
$$p(K, A, B, C) \leftarrow r(K, A', B),\ not\, s(K, A', C'),\ s(K, A'', C),\ extend(A', A'', A)$$
$$p(K, A, \bot, C) \leftarrow s(K, A, C),\ not\, r(K, A, B)$$ □

Thus, in the following we assume that the merging of two databases D_1 and D_2 is done by a stratified Datalog program $\mathcal{MP}$ (merging program) which applied to D_1 and D_2 gives a new database D ($\mathcal{MP}(D_1, D_2) = D$) consisting of a subset of the facts inferred by means of the merging program (D is a subset of $\mathcal{SM}(\mathcal{MP}_{D_1 \cup D_2})$). The use of a (stratified) logic program for integrating databases makes the merging process more flexible with respect to the use of predefined operators such as the ones introduced here and the others defined in the literature [23].

Example 5. Consider the two relations R and S denoting oriented graphs. The following program P 'merges' the two relations by deleting tuples which can be inferred, i.e. it computes the relation $T = R \cup S - \pi_{R.F,\, S.T}(R \bowtie_{R.T\, =\, S.F} S)$.

$$rs(F, T) \leftarrow r(F, T) \qquad t'(F, T) \leftarrow rs(F, I),\ rs(I, T)$$
$$rs(F, T) \leftarrow s(F, T) \qquad t(F, T) \leftarrow rs(F, T),\ not\ t'(F, T)$$

The output database consists of the t-tuples computed by P, that is, the tuples computed by applying P to R and S.

The following program merges the two relations R and S by computing their union and replacing null values with values taken from the closure of the graph:

$$c(F, T) \leftarrow rs(F, T),\ T \neq \bot \qquad tc(F, T) \leftarrow rs(F, T)$$
$$c(F, T) \leftarrow rs(F, \bot),\ tc(F, T) \qquad tc(F, T) \leftarrow rs(F, I),\ tc(I, T)$$ □

Fact 1 *The merging of two databases D_1 and D_2 can be done in polynomial time in the size of D_1 and D_2.* □

4.2 Managing Inconsistent Databases

Since the merged database could be inconsistent, we present a technique which permits us to compute repairs and consistent answers for possibly inconsistent databases. The technique is based on the generation of a disjunctive program $\mathcal{DP}(\mathcal{IC})$ derived from the set of integrity constraints $\mathcal{IC}$. The repairs for the database can be generated from the stable models of $\mathcal{DP}(\mathcal{IC})$ whereas the computation of the consistent answers of a query $(g, \mathcal{P})$ can be derived by considering the stable models of the program $\mathcal{P} \cup \mathcal{DP}(\mathcal{IC})$ over the database D.

Definition 13. Let c be a universally quantified constraint of the form

$$(\forall X)[B_1 \wedge ... \wedge B_n \wedge \varphi \supset A_1 \vee \cdots \vee A_m\ ]$$

where $B_1, ..., B_n, A_1, ..., A_n$ are positive atoms, then, $dj(c)$ denotes the rule

$$\neg B'_1 \vee ... \vee \neg B'_n \vee A'_1 \vee ... \vee A'_m \leftarrow (B_1 \vee B'_1), ..., (B_n \vee B'_n), \varphi, \\ (not\ A_1 \vee \neg A'_1), ..., (not\ A_m \vee \neg A'_m)$$

where $B'_i(A'_i)$ denotes the atom derived from $B_i(A_i)$, by replacing the predicate symbol p with the new symbol p_d. Let $\mathcal{IC}$ be a set of universally quantified integrity constraints, then $\mathcal{DP}(\mathcal{IC}) = \{\ dj(c) \mid c \in \mathcal{IC}\ \}$. □

Thus, $\mathcal{DP}(\mathcal{IC})$ denotes the set of generalized disjunctive rules derived from the rewriting of $\mathcal{IC}$, $\mathcal{DP}(\mathcal{IC})_D$ denotes the program derived from the union of the rules in $\mathcal{DP}(\mathcal{IC})$ with the facts in D and $\mathcal{SM}(\mathcal{DP}(\mathcal{IC})_D)$ (resp. $\mathcal{MM}(\mathcal{DP}(\mathcal{IC})_D)$) denotes the set of stable (resp. minimal) models of $\mathcal{DP}(\mathcal{IC})_D$. Recall that every stable model is consistent, according to the definition of consistent set given in Section 2, since it cannot contain two atoms of the form A and $\neg A$.

Example 6. Consider the integrated relation T of Example 3. The functional dependency $K \rightarrow (Title, Author, Year)$ stating that K is a key for the relation can be rewritten as a first order formula r:

$$\forall(X, Y, Z, W, Y', Z', W')[T(X, Y, Z, W) \wedge T(X, Y', Z', W') \supseteq Y = Y', Z = Z', W = W']$$

The associated disjunctive rule $dj(r)$ is

$$\neg T_d(X, Y, Z, W) \vee \neg T_d(X, Y', Z', W') \leftarrow (T(X, Y, Z, W) \vee T_d(X, Y, Z, W)), \\ (T(X, X', Y', Z') \vee T_d(X, X', Y', Z')), \\ not(Y = Y', Z = Z', W = W')$$

Observe that a (generalized) extended disjunctive Datalog program can be simplified by eliminating from the body rules all literals whose predicate symbols are derived and do not appear in the head of any rule (these literals cannot be true). Thus the rule can be simplified as

$$\neg T_d(X, Y, Z, W) \vee \neg T_d(X, Y', Z', W') \leftarrow T(X, Y, Z, W), T(X, X', Y', Z'), \\ not(Y = Y', Z = Z', W = W')$$

since the predicate T_d does not appear in the head of any rule and for this reason cannot be inferred by such a program. The above program has two stable models $M_1 = D \cup \{\neg T_d(3, Sky, Jones, 1965)\}$ and $M_2 = D \cup \{\neg T_d(3, Flowers, Smith, 1965)\}$. □

Definition 14. A set of integrity constraints $\mathcal{IC}$ is said to be *acyclic* if, by rewriting all constraints in $\mathcal{IC}$ as denials, every base atom appears either positive or negative in all rules. □

4.3 Computing Database Repairs

Every stable model can be used to define a possible repair for the database by interpreting new derived atoms (denoted by the subscript "d") as insertions and deletions of tuples. Thus, if a stable model M contains two atoms $\neg p_d(t)$ (derived atom) and $p(t)$ (base atom) we deduce that the atom $p(t)$ violates some constraint and, therefore, it must be deleted. Analogously, if M contains the derived atoms $p_d(t)$ and does not contain $p(t)$ (i.e. $p(t)$ is not in the database) we deduce that the atom $p(t)$ should be inserted in the database.

Definition 15. Given a database D and a set of integrity constraint $\mathcal{IC}$ over D and letting M be a stable model of $\mathcal{DP}(\mathcal{IC})_D$, then, $\mathcal{R}(M) = (\ \{p(t) \mid p_d(t) \in M \ \wedge\ p(t) \notin D\},\ \{p(t) \mid \neg p_d(t) \in M \ \wedge\ p(t) \in D\}\ )$. □

Theorem 2. *Given a database D and a set of integrity constraints $\mathcal{IC}$ on D, then*

1. *(Soundness) for every stable model M of $\mathcal{DP}(\mathcal{IC})_D$, $\mathcal{R}(M)$ is a repair for D;*
2. *(Completeness) for every database repair S for D there exists a stable model M for $\mathcal{DP}(\mathcal{IC})_D$ such that $S = \mathcal{R}(M)$.* □

Example 7. Consider the database of Example 2. The rewriting of the integrity constraint $(\forall X)[\ p(X) \supset q(X)\ ]$ produces the disjunctive rule

$$r : \neg p_d(X) \vee q_d(X) \leftarrow (p(X) \vee p_d(X)), (not\ q(X) \vee \neg q_d(X))$$

which can be rewritten in the simpler form

$$r' : \neg p_d(X) \vee q_d(X) \leftarrow p(X), not\ q_d(X)$$

since the predicates p_d and $\neg q_d$ do not appear in the head of any rule. The program P_D, where P is the program consisting of the disjunctive rule r' and D is the input database, has two stable models $M_1 = D \cup \{\ \neg p_d(b)\}$ and $M_2 = D \cup \{\ q_d(b)\}$. The derived repairs are $\mathcal{R}(M_1) = (\{q(b)\}, \emptyset)$ and $\mathcal{R}(M_2) = (\emptyset, \{p(b)\})$ corresponding, respectively, to the insertion of $q(b)$ and to the deletion of $p(b)$. □

4.4 Computing Consistent Answers

Let M be a stable model of $\mathcal{DP}(\mathcal{IC})_D$ and $\mathcal{R}(M)$ the associated repair for the database D; then $\mathcal{R}(M, D)$ denotes the repaired database obtained by means of the deletions and insertions specified in M.

The consistent answer for the query $Q = (g, \mathcal{P})$ over the database D under constraints $\mathcal{IC}$ is as follows:

$$Q(D,\mathcal{IC})^+ = \{\ g(t) |\ \forall M \in \mathcal{SM}((\mathcal{DP}(\mathcal{IC}))_D)\ is\ \ g(t) \in \mathcal{MM}((\mathcal{P} \cup \mathcal{DP}(\mathcal{IC}))_{\mathcal{R}(M,D)})\}$$
$$Q(D,\mathcal{IC})^- = \{\ g(t) |\ \nexists M \in \mathcal{SM}((\mathcal{DP}(\mathcal{IC}))_D)\ s.t.\ g(t) \in \mathcal{MM}((\mathcal{P} \cup \mathcal{DP}(\mathcal{IC}))_{\mathcal{R}(M,D)})\}$$
$$Q(D,\mathcal{IC})^u = \{\ g(t) |\ \exists M_1, M_2 \in \mathcal{SM}((\mathcal{DP}(\mathcal{IC}))_D)\ s.t.$$
$$g(t) \in \mathcal{MM}((\mathcal{P} \cup \mathcal{DP}(\mathcal{IC}))_{\mathcal{R}(M_1,D))}),\ g(t) \notin \mathcal{MM}((\mathcal{P} \cup \mathcal{DP}(\mathcal{IC}))_{\mathcal{R}(M_2,D)})\}$$

For instance, in Example 7, the set of true tuples are those belonging to the intersection of the two models, that is $p(a)$, $q(a)$ and $q(c)$, whereas the set of undefined tuples are those belonging to the union of the two models and not belonging to their intersection, that is $p(b)$ and $q(b)$.

Note that for every database D, query $Q = (g, \mathcal{P})$ and repaired database D':

1. Each atom $A \in Q(D, \mathcal{IC})^+$ belongs to the stable model of $\mathcal{P}_{D'}$ (soundness)
2. Each atom $A \in Q(D, \mathcal{IC})^-$ does not belong to the stable model of $\mathcal{P}_{D'}$ (completeness).

Example 8. Consider the database of Example 1. The functional dependency defined by the key of relation $Teaches$ can be defined as

$$(\forall X \, \forall Y \, \forall Z) \; [\; Teaches(X, Y) \wedge Teaches(X, Z) \supset Y = Z \;]$$

The corresponding disjunctive program P consists of the rule

$$\neg Teaches_d(X, Y) \vee \neg Teaches_d(X, Z) \leftarrow Teaches(X, Y), Teaches(X, Z), Y \neq Z$$

The program P_D has two stable models: $M_1 = D \cup \{\neg Teaches_d(c_2, p_2)\}$ and $M_2 = D \cup \{\neg Teaches_d(c_2, p_3)\}$. The answer to the query $(Teaches, \emptyset)$ gives the tuple (c_1, p_1) as true and the tuples (c_2, p_2) and (c_2, p_3) as undefined. □

5 Repair Constraints

In this section we introduce repair constraints which permit us to restrict the number of repairs. These constraints can be defined during the integration phase to give preference to certain data with respect to others.

Definition 16. A *repair constraint* is a denial rule of the form

$$\leftarrow up_1(A_1), ..., up_k(A_k), L_1, ..., L_n$$

where $up_1, ..., up_k \in \{insert, delete\}$, $A_1, ..., A_k$ are atoms and $L_1, ..., L_n$ are standard literals. □

Informally, the semantics of a repair constraint is as follows: if $L_1, ..., L_n$ is true in the repaired database then at least one of the updates $up_i(A_i)$ must be false.

Definition 17. Let D be a database, $\mathcal{IC}$ a set of integrity constraints and $\mathcal{RC}$ a set of repair constraints. We say a repair R for D satisfies $\mathcal{RC}$ (written $(R, D) \models \mathcal{RC}$) if for each $\leftarrow insert(A_1), ..., insert(A_k), delete(B_1), ..., delete(B_h), L_1, ..., L_n$ in $\mathcal{RC}$ then i) there is some A_i false in R^+ or ii) there is some B_i true in R^- or iii) there is some L_i false in $R(D)$. □

Given a database D, a set of integrity constraints $\mathcal{IC}$ and a set of repair constraints $\mathcal{RC}$ over D, a repair R for the database D is said to be *feasible* if all repair constraints in $\mathcal{RC}$ are satisfied. Moreover, a repaired database $R(D)$ is said to be *feasible* if R is feasible. Clearly, feasible repaired databases are consistent.

Example 9. Consider the database $D = \{e(Peter, 30000),\ e(John, 40000),\ e(John, 50000)\}$ containing information about names and salaries of employees, and the integrity constraint $(\forall X, Y, Z)[e(X, Y) \wedge e(X, Z) \supset Y = Z]$. There are two repairs for such a database: $R_1 = (\emptyset, \{e(John, 40000)\})$ and $R_2 = (\emptyset, \{e(John, 50000)\})$ producing, respectively, the repaired databases $D_1 = \{e(Peter, 30000), e(John, 50000)\}$ and $D_2 = \{e(Peter, 30000), e(John, 40000)\}$. The repair constraint

$$\leftarrow delete(e(X, S)),\ e(X, S'),\ S' > S$$

states that if the same employee occurs with more than one salary we cannot delete the tuple with the lowest salary. Thus, the repair R_1 is not feasible since it deletes the tuple $e(John, 40000)$ but the repaired database $R_1(D)$ contains the tuple $e(John, 50000)$. □

Definition 18. Given a database D, a set of integrity constraints $\mathcal{IC}$ and a set of repair constraints $\mathcal{RC}$ over D, an atom A is true (resp. false) with respect to $(D, \mathcal{IC}, \mathcal{RC})$ if A belongs to all feasible repaired databases (resp. there is no feasible repaired database containing A). The set of atoms which are neither true nor false is undefined. □

Clearly, for an empty set of repair constraints Definition 7 and Definition 18 coincide. The formal semantics of databases with both integrity and repair constraints is given by rewriting the repair constraints into (generalized) extended disjunctive rules with empty heads (denials). In particular, the sets of integrity constraints $\mathcal{IC}$ and repair constraints $\mathcal{RC}$ are rewritten into an extended disjunctive program $\mathcal{DP}(\mathcal{IC}, \mathcal{RC})$. Each stable model of $\mathcal{DP}(\mathcal{IC}, \mathcal{RC})$ over a database D can be used to generate a repair for the database, whereas each stable model of the program $P \cup \mathcal{DP}(\mathcal{IC}, \mathcal{RC})$, over the database D, can be used to compute a consistent answer to a query (g, P). Each model defines a set of actions (update operations) on the inconsistent database to achieve a consistent state.

Definition 19. Let r be a repair constraint of the form

$$\leftarrow insert(A_1), ..., insert(A_k), delete(A_{k+1},)..., delete(A_m), B_1, ..., B_l,\ not\ B_{l+1}, ..., not\ B_n, \varphi$$

where $A_1, ..., A_m, B_1, ..., B_n$ are atoms. Then, $dj(r)$ denotes the denial rule

$$\leftarrow A'_1, ..., A'_k,\ \neg A'_{k+1}, ..., \neg A'_m,\ ((B_1, not\ \neg B'_1) \vee B'_1),\ ...,\ ((B_l, not\ \neg B'_l) \vee B'_l),\ ((not\ B_{l+1}, not\ B'_{l+1}) \vee \neg B'_{l+1}),\ ...,\ ((not\ B_n, not\ B'_n) \vee \neg B'_n),\ \varphi$$

where $A'_i(B'_i)$ is derived from $A_i(B_i)$ by replacing the predicate symbol, say p, with p_d and φ is a conjunction of built-in atoms. Let $\mathcal{RC}$ be a set of repair constraints, then $\mathcal{DP}(\mathcal{RC}) = \{\ dj(r) \mid r \in \mathcal{RC}\ \}$. Moreover, $\mathcal{DP}(\mathcal{IC}, \mathcal{RC})$ denotes the set $\mathcal{DP}(\mathcal{IC}) \cup \mathcal{DP}(\mathcal{RC})$. □

Example 10. Consider the database of Example 9. The repair constraint

$$\leftarrow delete(e(X,S)),\ e(X,S'),\ S' > S$$

is rewritten as

$$\leftarrow \neg e_d(X,S),\ (((e(X,S'),\ not\ \neg\, e_d(X,S')) \vee e_d(X,S')),\ S' > S \qquad \square$$

Given a database D, a set of integrity constraints $\mathcal{IC}$ and a set of repair constraints $\mathcal{RC}$ over D, then for each stable model M of $\mathcal{DP}(\mathcal{IC},\mathcal{RC})_D$, $\mathcal{R}(M)$ denotes the following pair: $\mathcal{R}(M) = (\ \{p(t) \mid p_d(t) \in M \wedge p(t) \notin D\},\ \{p(t) \mid \neg p_d(t) \in M\ \wedge\ p(t) \in D\}\ )$.

Lemma 1. *Let $\mathcal{IC}$ be a set of integrity constraints and $\mathcal{RC}$ a set of repair constraints over a database D, then M is a stable model for $\mathcal{DP}(\mathcal{IC},\mathcal{RC})_D$ if and only if it is a stable model for $\mathcal{DP}(\mathcal{IC},\mathcal{RC})_D$ satisfying $\mathcal{DP}(\mathcal{RC})$.* $\square$

The following theorem states the relation between feasible repairs and stable models.

Theorem 3. *Given a database D, a set of integrity constraints $\mathcal{IC}$ and a set of repair constraints $\mathcal{RC}$ over D, then*

1. *(Soundness) for every stable model M of $\mathcal{DP}(\mathcal{IC},\mathcal{RC})_D$, $\mathcal{R}(M)$ is a feasible repair for D;*
2. *(Completeness) for every feasible database repair R for D there exists a stable model M for $\mathcal{DP}(\mathcal{IC},\mathcal{RC})_D$ such that $R = \mathcal{R}(M)$.* $\square$

An *update constraint* is a repair constraint of the form $\leftarrow up(p(t_1,...,t_n))$ where $up \in \{delete, insert\}$ and $t_1,...,t_n$ are terms (constants or variables). Update constraints define the type of update operations allowed on the database.

6 Prioritized Repairs

In this section we extend our framework by considering prioritized updates. These rules can be useful when the database is derived from the integration of different databases since they can be used to take into account the origin of data.

Definition 20. A *prioritized update rule* is a rule of the form $up_1(A) \preceq up_2(B)$ where $up_1, up_2 \in \{insert, delete\}$, and A and B are atoms. $\square$

For any two update atoms a_1 and a_2, if $a_1 \preceq a_2$ then we say that a_2 has higher priority than a_1. Moreover, $a_1 \prec a_2$ if $a_1 \preceq a_2$ and $a_1 \neq a_2$. A priority statement $a_1 \preceq a_2$ means that for each a_1' instance of a_1 and for each a_2' instance of a_2 is $a_1' \preceq a_2'$. Clearly, the sets of ground instantiations of a_1 and a_2 must have empty intersections. In the following we shall denote with $\mathcal{PC}$ the set of prioritized update rules and with $\mathcal{PC}^*$ the reflexive, transitive closure of $\mathcal{PC}$.

The relation $\sqsubseteq$ is defined over the feasible repairs of D as follows. For any repairs R_1, R_2 and R_3 of D,

1. $R_1 \sqsubseteq R_1$,
2. $R_1 \sqsubseteq R_2$ if a) $\exists e_2 \in R_2 - R_1$, $\exists e_1 \in R_1 - R_2$ such that $(e_1 \preceq e_2) \in \mathcal{PC}^*$ and b) $\nexists e_3 \in R_1 - R_2$ such that $(e_2 \preceq e_3) \in \mathcal{PC}^*$,
3. if $R_1 \sqsubseteq R_2$ and $R_2 \sqsubseteq R_3$, then $R_1 \sqsubseteq R_3$.

If $R_1 \sqsubseteq R_2$ we say that R_2 is *preferable* to R_1. Moreover, we write $R_1 \sqsubset R_2$ if $R_1 \sqsubseteq R_2$ and $R_1 \neq R_2$. A set of updates R is a *preferred* repair for D if R is a repair for D and there is no repair R' for D which is preferable to R.

Definition 21. Let D be a database, IC a set of full integrity constraints, $\mathcal{RC}$ a set of repair constraints and $\mathcal{PC}$ a set of prioritized update rules. Then, $\mathcal{DP}(\mathcal{IC}, \mathcal{RC}, \mathcal{PC})$ denotes the prioritized generalized extended disjunctive program $(DP(\mathcal{IC}, \mathcal{RC}), \Phi(\mathcal{PC}))$ where $\Phi(\mathcal{PC}) = \{A'_1 \preceq A'_2 | up_1(A_1) \preceq up_2(A_2)\}$ and A'_i is derived from $up_i(A_i)$ as follows:

1. $A'_i = p_d(t_1, .., t_n)$ if $up_i(A_i) = insert(p(t_1, .., t_n))$,
2. $A'_i = \neg p_d(t_1, .., t_n)$ if $up_i(A_i) = delete(p(t_1, .., t_n))$. □

Thus, the new disjunctive program is obtained by rewriting integrity constraints into generalized extended disjunctive rules, repair constraints into generalized extended denials and then adding prioritized rules obtained from the rewriting of prioritized update rules. In the following, $\mathcal{DP}(\mathcal{IC}, \mathcal{RC}, \mathcal{PC})$ denotes the pair $(\mathcal{DP}(\mathcal{IC}, \mathcal{RC}), \Phi(\mathcal{PC}))$.

Theorem 4. *Let D be a database, IC a set of full integrity constraints, $\mathcal{RC}$ a set of repair constraints and $\mathcal{PC}$ a set of prioritized update rules, then*

1. *(Soundness) for every preferred stable model M of $\mathcal{DP}(\mathcal{IC}, \mathcal{RC}, \mathcal{PC})_D$, $\mathcal{R}(M)$ is a preferred repair for D;*
2. *(Completeness) for every preferred database repair S for D there exists a preferred stable model M for $\mathcal{DP}(\mathcal{IC}, \mathcal{RC}, \mathcal{PC})_D$ such that $S = \mathcal{R}(M)$.* □

Prioritized update rules can be useful in the integration of databases to take into account the origin of data. For instance, when we want to give preference (credit) to the data coming from one source with respect to those coming from another.

Each atom is of the form $p_{[s]}(t_1, ..., t_n)$ where p is a predicate symbol and s is a source database identifier. Moreover, we denote with $*$ a generic database and $p_{[*]}(t_1, ..., t_n)$ is also denoted by $p(t_1, ..., t_n)$ (i.e. the origin of data is not important). Inside repair and prioritized constraints *delete* operations may refer to the origin of data.
In this case each tuple $p(t_1, ..., t_n)$ coming from a source s is stored as $p(t_1, ..., t_n, s)$ and the constraints are rewritten accordingly.

7 Conclusions

The proposed technique is sound and complete for universally quantified constraints whereas previously defined techniques only consider restricted cases. In

the general case, our technique is quite expensive (checking if a fact belongs to the consistent answer of a query Q is bounded in the second level of the polynomial hierarchy (Π_2^P)), but there are significant classes of constraints such as functional and inclusion dependencies which can be computed efficiently; in fact, for these classes of constraints, computing nondeterministically a repair can be done in polynomial time and computing the consistent answer is polynomial if the program appearing in the query is empty [10]. The introduction of repair constraints and prioritized update rules makes possible to restrict the set of feasible repairs; clearly, in the general case, the complexity increases since checking if there exists a preferred repair (resp. for each preferred repair) for a database D, such that the answer to a query Q is not empty, is complete for the third level of the polynomial hierarchy (Σ_3^P-complete and Π_3^P-complete, respectively) [10].

Acknowledgement

The authors thank Thomas Eiter for addressing the work on priorized logic programming.

References

1. Argaval, S., Keller, A. M., Wiederhold, G., Saraswat, K., Flexible Relation: an Approach for Integrating Data from Multiple, Possibly Inconsistent Databases. *Proc. Int. Conf. on Data Engineering*, pages 495-504, 1995.
2. Arenas, M., Bertossi, L., Chomicki, J., Consistent Query Answers in Inconsistent Databases. *Proc. Int. Conf. on Principles of Database Systems*, pages 68-79, 1999.
3. Arenas, M., Bertossi, L., Chomicki, J., Specifying and Querying Database repairs using Logic Programs with Exceptions. *FQAS Conf.*, pages 27-41, 2000.
4. Baral, C., Kraus, S., Minker, J., Combining Multiple Knowledge Bases. *IEEE-TKDE*, Vol. 3, No. 2, pages 208-220, 1991.
5. Baral, C., Kraus, S., Minker, J., Subrahmanian, V. S., Combining Knowledge Bases Consisting of First Order Theories. *Proc. ISMIS Conf.*, pages 92-101, 1991.
6. Bry, F., Query Answering in Information System with Integrity Constraints, *Proc. IFIP 11.5 Working Conf.*, 1997.
7. Dung, P. M., Integrating Data from Possibly Inconsistent Databases. *Proc. CoopIS Conf.*, pages 58-65, 1996.
8. Eiter, T., Gottlob, G., Mannila, H., Disjunctive Datalog, *ACM Transaction on Database Systems*, Vol. 22, No. 3, pages 364–418, 1997.
9. Gelfond, M., Lifschitz, V., Classical Negation in Logic Programs and Disjunctive Databases, *New Generation Computing*, No. 9, pages 365-385, 1991.
10. Greco, G., Greco, S., Zumpano, E., A Logical Framework for Querying and Repairing Inconsistent Databases. *ISI-CNR Technical Report*, 2001.
11. Greco, S., Saccà, D., Negative Logic Programs. *North American Conf. on Logic Progr.*, pages 480-497, 1990.
12. Greco, S., Zumpano E., Querying Inconsistent Databases *Proc. Int. Conf. on Logic for Programming and Automatic Reasoning*, pages 308-325, 2000.
13. Grant, J., Subrahmanian, V. S., Reasoning in Inconsistent Knowledge Bases. *IEEE Transaction of Knowledge and Data Engineering*, Vol. 7, No. 1, pages 177-189, 1995.

14. Kanellakis, P. C., Elements of Relational Database Theory. *Handbook of Theoretical Computer Science*, Vol. 2, J. van Leewen (ed.), North-Holland, 1991.
15. Kowalski, R. A., Sadri, F., Logic Programs with Exceptions. *New Generation Computing*, Vol. 9, No. 3/4, pages 387-400, 1991.
16. Lin, J., A Semantics for Reasoning Consistently in the Presence of Inconsistency. *Artificial Intelligence*, Vol. 86, No. 1, pages 75-95, 1996.
17. Lin, J., Integration of Weighted Knowledge Bases. *Artificial Intelligence*, Vol. 83, No. 2, pages 363-378, 1996.
18. Lin, J., Mendelzon, A. O., Knowledge Base Merging by Majority, in *Dynamic Worlds: From the Frame Problem to Knowledge Management*, R. Pareschi and B. Fronhoefer (eds.), Kluwer, 1999.
19. Pereira, L.M., Alferes, J.J., Aparicio, J. N., The Extended Stable Models of Contradiction Removal Semantics, *5th Portuguese AI Conference,* pages 105-119, 1991.
20. Sakama, C., Inoue, K., Priorized logic programming and its application to commonsense reasoning. *Artificial Intelligence*, No. 123, pages 185-222, 2000.
21. Subrahmanian, V. S., Amalgamating Knowledge Bases. *ACM Transaction on Database Systems*, Vol. 19, No. 2, pages 291-331, 1994.
22. Ullman, J. K., *Principles of Database and Knowledge-Base Systems*, Vol. 1, Computer Science Press, 1988.
23. Yan, L.L., Ozsu, M.T., Conflict Tolerant Queries in Aurora. *Proc. Int. Conf. on Cooperative Information Systems*, pages 279-290, 1999.

Author Index