Lecture Notes in Computer Science 4670

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

Verónica Dahl Ilkka Niemelä (Eds.)

Logic Programming

23rd International Conference, ICLP 2007
Porto, Portugal, September 8-13, 2007
Proceedings

Volume Editors

Verónica Dahl
Simon Fraser University, School of Computing Science
Burnaby, BC V5A 1S6, Canada
E-mail: veronica@cs.sfu.ca

Ilkka Niemelä
Helsinki University of Technology
Department of Computer Science and Engineering
P.O. Box 5400, 02015 TKK, Finland
E-mail: ilkka.niemela@tkk.fi

Library of Congress Control Number: 2007933827

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

LNCS Sublibrary: SL 2 – Programming and Software Engineering

ISSN 0302-9743
ISBN-10 3-540-74608-0 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-74608-9 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 12115815 06/3180 5 4 3 2 1 0

Preface

This volume contains the proceedings of the 23rd International Conference on Logic Programming, ICLP 2007, held in the World Heritage City of Porto, Portugal, September 8–13, 2007. The conference was colocated with seven pre and post-conference workshops:

- The 8th Workshop on Computational Logic in Multi-Agent Systems (CLIMA-VIII)
- The 2nd International Workshop on Applications of Logic Programming to the Web, Semantic Web and Semantic Web Services (ALPSWS 2007)
- Answer Set Programming: Advances in Theory and Implementation (ASP 2007)
- The 4th Workshop on Constraint Handling Rules (CHR 2007)
- The 7th International Colloquium on Implementation of Constraint and Logic Programming Systems (CICLOPS 2007)
- Workshop on Constraint Based Methods for Bioinformatics (WCB 2007)
- The 17th Workshop on Logic-Based Methods in Programming Environments (WLPE 2007)

Since the first conference held in Marseilles in 1982, ICLP has been the premiere international conference for disseminating research results in logic programming. The present edition of the conference received 74 submissions from 23 countries: Argentina (1), Australia (4), Belgium (6), Brazil (1), Canada (2), Finland (2), France (4), Germany (3), Greece (1), Hungary (2), India (1), Israel (1), Italy (13), The Netherlands (1), Poland (1), Portugal (10), Russia (2), Singapore (2), South Korea (1), Spain (6), the United Arab Emirates (1), the USA (7), and the UK (2). The Program Committee selected 22 technical papers for presentation and inclusion in the proceedings. In addition, the program also included 15 poster presentations.

As in the past, the ICLP Program Committee selected the best paper and the best student paper. The Best Paper Award went to Sabrina Baselice, Piero Bonatti and Giovanni Criscuolo, for their paper "On Finitely Recursive Programs," while Matti Järvisalo and Emilia Oikarinen won the Best Student Paper Award for their paper "Extended ASP Tableaux and Rule Redundancy in Normal Logic Programs."

The highlights of ICLP 2007 included invited talks by Gerhard Brewka titled "Preferences, Contexts and Answer Sets" and by Chitta Baral, titled "Towards Overcoming the Knowledge Acquisition Bottleneck in Answer Set Prolog Applications: Embracing Natural Language Inputs." The program also featured four invited tutorials: "Answer Set Programming for the Semantic Web" by Thomas Eiter, "Coinductive Logic Programming and Its Applications" by Gopal Gupta, "Multi-Paradigm Declarative Languages" by Michael Hanus, and "Logic Programming for Knowledge Representation" by Mirosław Truszczyński.

ICLP 2007 was organized by the Association for Logic Programming (ALP), in collaboration with the Organizing Committees of the collocated workshops and the Computer Science Department at the Faculty of Sciences of the University of Porto. It was sponsored by the Association for Logic Programming, the Portuguese Association for Artificial Intelligence (APPIA), and the University of Porto. We greatly appreciate their generous support.

Many people contributed to the success of the conference, to whom we hereby extend our gratitude and thanks. The General Chair, Fernando Silva, worked hard to coordinate the overall arrangements of the conference, from the conference site and sponsoring to budgeting and registration. PC members and several other external referees provided timely and in-depth reviews of the submitted papers, and worked hard to select the best papers and posters for the conference program. The Workshop Chair, Agostino Dovier, and the Doctoral Consortium Chairs, Enrico Pontelli and Inês Dutra, significantly contributed to the conference's rich programme. Salvador Abreu, the Publicity Chair, worked hard to spread all news regarding the conference, and Ricardo Rocha, the Local Chair, Michel Ferreira and Pedro Ribeiro orchestrated the website and the myriads of practical arrangements needed at the conference site. Bart Demoen, indefatigable after thirteen years of successfully running the Logic Programming Contest, ran yet another challenging and interesting contest. It goes without saying that the broad logic programming community contributed the most by submitting excellent technical and application papers and posters. Last but not least, we thank the developers of the EasyChair conference management system, which made our job definitely easier.

September 2007

Verónica Dahl
Ilkka Niemelä
Program Committee Co-Chairs
ICLP 2007

Organization

ICLP 2007 was organized by the Association for Logic Programming (ALP), in collaboration with the Computer Science Department at the Faculty of Sciences of the University of Porto.

Organizing Committee

General Chair	Fernando Silva (University of Porto, Portugal)
Program Co-chairs	Verónica Dahl (Simon Fraser University, Canada)
	Ilkka Niemelä (Helsinki University of Technology, Finland)
Workshop Chair	Agostino Dovier (University of Udine, Italy)
Doctoral Consortium Chairs	Enrico Pontelli (New Mexico State University, USA)
	Inês Dutra (University of Porto, Portugal)
Publicity Chair	Salvador Abreu (University of Évora, Portugal)
Local Chair	Ricardo Rocha (University of Porto, Portugal)

Program Committee

Maurice Bruynooghe (Katholieke Universiteit Leuven, Belgium)
Keith Clark (Imperial College of London, UK)
Verónica Dahl, Co-chair (Simon Fraser University, Canada)
Marina De Vos (University of Bath, UK)
Yannis Dimopoulos (University of Cyprus, Cyprus)
Inês Dutra (University of Porto, Portugal)
Esra Erdem (Sabanci University, Turkey)
Maurizio Gabbrielli (University of Bologna, Italy)
Patricia M Hill (University of Leeds, UK)
Katsumi Inoue (National Institute of Informatics, Japan)
Tomi Janhunen (Helsinki University of Technology, Finland)
Anthony Kusalik (University of Saskatchewan, Canada)
Nicola Leone (University of Calabria, Italy)
Vladimir Lifschitz (University of Texas at Austin, USA)
Ilkka Niemelä, Co-chair (Helsinki University of Technology, Finland)
Germán Puebla (Technical University of Madrid, Spain)
Francesca Rossi (University of Padua, Italy)
Konstantinos Sagonas (Uppsala University, Sweden)
Peter Schachte (University of Melbourne, Australia)

Torsten Schaub (University of Potsdam, Germany)
Fernando Silva (University of Porto, Portugal)
Guillermo R. Simari (Universidad Nacional del Sur, Argentina)
Tran Cao Son (New Mexico State University, USA)
Paul Tarau (University of North Texas, USA)
Francesca Toni (Imperial College of London, UK)
Eric Villemonte de la Clergerie (INRIA, France)
David S. Warren (State University of New York at Stony Brook, USA)
Stefan Woltran (Vienna University of Technology, Austria)

Additional Referees

Jesús Almendros
Ofer Arieli
Marcello Balduccini
Ralph Becket
Leopoldo Bertossi
Angela Bonifati
Martin Brain
Daniel Cabeza
Martin Caminada
Álvaro Cortés-Calabuig
Susanna Cozza
Danny De Schreye
Bart Demoen
Cinzia Di Giusto
Antonio J. Fernandez
Wolfgang Faber
Michel Ferreira
Michael Fink
Nuno A. Fonseca
Alfredo Gabaldon
Dorian Gaertner
Lorenzo Gallucci
Martin Gebser
Samir Genaim
Ping Hou
Giovambattista Ianni
Zeynep Kiziltan
Joohyung Lee
Ho-fung Leung
Yuliya Lierler
Francisco López-Fraguas
Lunjin Lu
Thomas Lukasiewicz
Marco Maratea
Francis McCabe
Maria Chiara Meo
Loizos Michael
Pavlos Moraitis
José Francisco Morales
Maxime Morge
Lee Naish
Jonathan Needham
Johannes Oetsch
Magdalena Ortiz
Simona Perri
Andrea Pescetti
Pawel Pietrzak
Inna Pivkina
Axel Polleres
Enrico Pontelli
Jörg Pührer
Oliver Ray
Francesco Ricca
Jussi Rintanen
Peter Robinson
Ricardo Rocha
Fariba Sadri
Chiaki Sakama
Vítor Santos Costa
Ken Satoh
Tom Schrijvers
Fernando Silva
Mantas Simkus
Jon Sneyers
Zoltan Somogyi
Harald Sondergaard
John Sowa
Fausto Spoto
Peter Stuckey
Paolo Tacchella
Ferhan Ture
Peter Van Weert
Luca Viganò
Enea Zaffanella
Damiano Zanardini

Table of Contents

Invited Talks

Invited Tutorials

Regular Talks

Answer Set Programming

Applications

Constraint Logic Programming

Semantics

Program Analysis

Special Interest Paper

Implementation

Poster Presentations

Doctoral Consortium Presentations

Towards Overcoming the Knowledge Acquisition Bottleneck in Answer Set Prolog Applications: Embracing Natural Language Inputs

Chitta Baral, Juraj Dzifcak, and Luis Tari

School of Computing and Informatics
Arizona State University
Tempe, AZ 85287-8809

Abstract. Answer set Prolog, or AnsProlog in short, is one of the leading knowledge representation (KR) languages with a large body of theoretical and building block results, several implementations and reasoning and declarative problem solving applications. But it shares the problem associated with knowledge acquisition with all other KR languages; most knowledge is entered manually by people and that is a bottleneck. Recent advances in natural language processing have led to some systems that convert natural language sentences to a logical form. Although these systems are in their infancy, they suggest a direction to overcome the above mentioned knowledge acquisition bottleneck. In this paper we discuss some recent work by us on developing applications that process logical forms of natural language text and use the processed result together with AnsProlog rules to do reasoning and problem solving.

1 Introduction

Answer Set Prolog, or AnsProlog in short, is the logic programming language whose semantics is defined using the answer set semantics [GL88, GL91]. A knowledge base in this language consists of rules of the following form:

$$l_0 \leftarrow l_1, \ldots, l_m, \ not\ l_{m+1}, \ldots, \ not\ l_n.$$

where l_i's are literals. Given an AnsProlog knowledge base KB and a query Q, we say KB entails Q, denoted by $KB \models Q$, if Q is true in all answer sets of KB. A related term, Answer Set Programming (or ASP) often focuses only on encoding a given problem as an AnsProlog program so that each of its answer sets encodes a solution of that problem. For knowledge representation and reasoning both query entailment and problem solving by finding answer sets are important.

Over the last 20 years a large body of work has been done on this language. These include theoretical building block results, several implementations and reasoning and declarative problem solving applications. While the book [Bar03] is a compilation of many of these results, there have been a lot of further developments since this book was written. Some highlights include the use of SAT in finding answer sets [LZ02, Lie05], the extension of AnsProlog to allow probabilities [BGR04], the recent system CLASP [GKNS07] and some recent work

V. Dahl and I. Niemelä (Eds.): ICLP 2007, LNCS 4670, pp. 1–21, 2007.

on applications involving natural language processing and question answering. In this paper we focus on the last aspect.

While we seem to be getting closer to an adequate language for creating knowledge bases, the obstacle of manual knowledge acquisition still haunts the development of a large body of sizeable knowledge bases. (There are other issues that have also hampered such development. They include lack of appropriate tools for facilitating building of knowledge bases such as GUIs, debuggers and the ability to easily reuse previously coded knowledge modules. Some research addressing these issues have been taken up recently [BDT06, dVS07].) Project Halo [FAW+04], an ambitious attempt to encode knowledge of a high school chemistry book chapter realized the bottleneck of knowledge acquisition in its phase one and has focused on it in its second phase.

Automatic knowledge acquisition through learning has focused mostly on learning from observations. However, some recent developments in natural language processing and natural language resource compilations has brought us closer to the possibility of acquiring knowledge from natural language text. Amongst them is the resource Wordnet [MBF+90], which is much more than an on-line dictionary; and systems such as LCC [MHG+02, MPHS03] and C&C [CCt] that can translate natural language text to a logical form. Although these systems are in their infancy, they suggest a direction to overcome the above mentioned knowledge acquisition bottleneck. In this paper we discuss some recent work by us on developing applications that process logical forms of natural language text and use the processed result together with AnsProlog rules to do reasoning and problem solving. We will particularly focus on our attempt to build a system that can solve puzzles described in a natural language. We will also briefly mention our other attempts[1], some reported or to be reported in detail elsewhere, in reasoning about cardinality of dynamic sets, reasoning about travel, question answering and textual entailment; all dealing with natural language input.

2 Processing Natural Language Using C&C Tools and Boxer

In this section we discuss in detail how we use the C&C tools and some postprocessing to process natural language and obtain (some) knowledge in AnsProlog syntax which together with additional rules allows us to do reasoning and problem solving.

We start with the C&C [CCt, BCS+04] tools that are used to translate natural language text into a logic form.

2.1 C&C Tools

The C&C tools consists of a set of tools including a robust wide-coverage CCG parser [CC07], several maximum entropy taggers(for example the POS tagger

[1] Many of these works were done in collaboration with Michael Gelfond and Marcello Balduccini of Texas Tech University and Richard Scherl of Monmouth University.

and Named Entity Recognizer) and a semantic analysis tool Boxer [BCS+04]. Most of them can be used individually, but are designed to be used together. The system is very robust and the tools have been used in various scenarios, such as the RTE challenge [RTE], where the problem is to recognize textual entailment; i.e., to decide if a given hypothesis is entailed by a given text or not. Many of the tools use statistics and a set of trained models on which the statistical methods are applied.

When using this system, the extraction of data from the English text is performed as follows. First, the CCG parser [CCt], together with taggers and other tools, is run on the data to obtain the first basic logic form of the data. Then the Boxer [CCt] system is used on the parser output to obtain a first order logic representation of the English text. This is performed sentence by sentence for the whole text. The Boxer output is, as said, in first order logic and thus contains variables as well as predicates and quantifiers. We then translate the Boxer first order logic output to a set of AnsProlog facts.

We now look more closely at each of the steps presented above.

2.2 CCG Parser

The first important step is to obtain the CCG parser output. Note that when doing this, many other tools are used in the process such as the tagger. Nevertheless, the parser is the most important part of this step. The CCG parser is a statistical parser with the focus on wide-coverage and high efficiency, producing a single parse of the text. It is based on the combinatorial categorial grammar(CCG), which is based on the the classical categorial grammar [Woo93]. The CCG parser works on a sentence by sentence basis. That means it parses and analyzes a single sentence and then moves to another and starts anew without using any information obtained from the previous sentence. Thus by itself, it is unable to perform and detect things like anaphoras. However, Boxer does employ a form of anaphora resolution.

The lexical entries of CCG include semantic interpretation and syntactic category. The category can not only be a basic category such as noun or verb phrase and gender, but also a complex category that is obtained recursively from the basic categories. For example transitive verbs can be given as $(S\backslash NP)/NP$, where 'S' is a sentence and NP stands for noun phrase. The delimiters '\' and '/' represent the first and the second argument of the verb respectively.

Let us now look at 2 different examples which illustrate the output of the parser.

Example 1. Consider the following sentences:

"The first puzzle category is person."

The output of the CCG parser, together with all the taggers, is as follows:

```
sem(1,
 bapp('S[dcl]',
  fapp('NP[nb]',
   leaf(1,1,'NP[nb]/N'),
```

```
   fapp('N',
    leaf(1,2,'N/N'),
    fapp('N',
     leaf(1,3,'N/N'),
     leaf(1,4,'N')))),
  fapp('S[dcl]\NP',
   leaf(1,5,'(S[dcl]\NP)/(S[ng]\NP)'),
   leaf(1,6,'S[ng]\NP')))).

word(1, 1, 'The', 'the', 'DT', 'I-NP', 'O', 'NP[nb]/N').
word(1, 2, 'first', 'first', 'JJ', 'I-NP', 'O', 'N/N').
word(1, 3, 'puzzle', 'puzzle', 'NN', 'I-NP', 'O', 'N/N').
word(1, 4, 'category', 'category', 'NN', 'I-NP', 'O', 'N').
word(1, 5, 'is', 'be', 'VBZ', 'I-VP', 'O', '(S[dcl]\NP)/(S[ng]\NP)').
word(1, 6, 'person.', 'person.', 'VBG', 'I-VP', 'O', 'S[ng]\NP').
```

The output consist of 2 parts, first one is a specific parse tree representing the sentence structure which is used by Boxer for semantic interpretation, with leafs representing the words of the sentence and the non-leaf nodes are the applied combinatorial rules. Next we have one line for each word in the sentence. The structure of the data is as follows. First we have the index of the translated sentence, followed by the index corresponding to the position within the tree, the used word, 'base' form of the word, followed by it's types given by various taggers and finally the category.

Now let us look at a bit more complex sentence.

"Jim Kirby and his partner, asked to name the first man on the moon, nominated Luis Armstrong."

The parser outputs the following:

```
sem(1,
 bapp('S[dcl]',
  bapp('NP[nb]',
   lex('N','NP',
    fapp('N',
     leaf(1,1,'N/N'),
     leaf(1,2,'N'))),
   conj('conj','NP[nb]','NP[nb]\NP[nb]',
    leaf(1,3,'conj'),
    fapp('NP[nb]',
     leaf(1,4,'NP[nb]/N'),
     leaf(1,5,'N')))),
  fapp('S[dcl]\NP',
   leaf(1,6,'(S[dcl]\NP)/(S[to]\NP)'),
   fapp('S[to]\NP',
    leaf(1,7,'(S[to]\NP)/(S[b]\NP)'),
    fapp('S[b]\NP',
     leaf(1,8,'(S[b]\NP)/NP'),
```

```
bapp('NP',
 fapp('NP[nb]',
  leaf(1,9,'NP[nb]/N'),
  fapp('N',
   leaf(1,10,'N/N'),
   leaf(1,11,'N'))),
 fapp('NP\NP',
  leaf(1,12,'(NP\NP)/NP'),
  bapp('NP',
   fapp('NP[nb]',
    leaf(1,13,'NP[nb]/N'),
    leaf(1,14,'N')),
   lex('S[dcl]\NP','NP\NP',
    fapp('S[dcl]\NP',
     leaf(1,15,'(S[dcl]\NP)/NP'),
     lex('N','NP',
      fapp('N',
       leaf(1,16,'N/N'),
       leaf(1,17,'N'))))))))))))).
```

```
word(1, 1, 'Jim', 'Jim', 'NNP', 'I-NP', 'I-PERSON', 'N/N').
word(1, 2, 'Kirby', 'Kirby', 'NNP', 'I-NP', 'I-PERSON', 'N').
word(1, 3, 'and', 'and', 'CC', 'O', 'O', 'conj').
word(1, 4, 'his', 'his', 'PRP$', 'I-NP', 'O', 'NP[nb]/N').
word(1, 5, 'partner,', 'partner,', 'NN', 'I-NP', 'O', 'N').
word(1, 6, 'asked', 'ask', 'VBD', 'I-VP', 'O', '(S[dcl]\NP)/(S[to]\NP)').
word(1, 7, 'to', 'to', 'TO', 'B-VP', 'O', '(S[to]\NP)/(S[b]\NP)').
word(1, 8, 'name', 'name', 'VB', 'I-VP', 'O', '(S[b]\NP)/NP').
word(1, 9, 'the', 'the', 'DT', 'I-NP', 'O', 'NP[nb]/N').
word(1, 10, 'first', 'first', 'JJ', 'I-NP', 'O', 'N/N').
word(1, 11, 'man', 'man', 'NN', 'I-NP', 'O', 'N').
word(1, 12, 'on', 'on', 'IN', 'I-PP', 'O', '(NP\NP)/NP').
word(1, 13, 'the', 'the', 'DT', 'I-NP', 'O', 'NP[nb]/N').
word(1, 14, 'moon,', 'moon,', 'NN', 'I-NP', 'O', 'N').
word(1, 15, 'nominated', 'nominate', 'VBD', 'I-VP', 'O', '(S[dcl]\NP)/NP').
word(1, 16, 'Luis', 'Luis', 'NNP', 'I-NP', 'I-PERSON', 'N/N').
word(1, 17, 'Armstrong.', 'Armstrong.', 'NNP', 'I-NP', 'O', 'N').
```

In this case, we obtain a similar output, although more complex. The output is formatted and contains the same information as in the case of the simple sentence. Let us look at the structure of this sentence. Note that the tree is divided into several big parts properly representing the sentence. Note that the first part represents noun phrase 'Jim Kirby and his partner' and the verb phrase 'asked to name the first man on the moon, nominated Luis Armstrong.'. Each

of them is then further divided into specific parts. For example the noun phrase is divided again into 'Jim Kirby', 'and' and 'his partner'.Thus one can easily observe the structure of the sentence just by looking at the given tree. Another point to note is that the some categories contain additional information, which is given in the square brackets. These are called features and provide additional information about the category. For example 'S[dcl]' means that the sentence is declarative.

The output is then ready to be used by the next step, Boxer.

2.3 Boxer

The next step is to use the Boxer tool on the CCG parser output to obtain a logic form from it. As said before, Boxer performs semantic analysis of the given text. The basic output of Boxer is the Discourse Representation Structures (DRSs) and the box representations of Discourse Representation Theory (DRT). However, in addition, one of the Boxer outputs can be a first order logic form representing the semantics of the sentence. This is very useful as it can be then be transformed into AnsProlog programs. Boxer does take care of some natural language issues such as anaphora resolution, which is not covered by the parser. Let us now take a closer look at the given example.

Example 2. Recall the example sentences presented before:

"The first puzzle category is person."

"Jim Kirby and his partner, asked to name the first man on the moon, nominated Luis Armstrong."

The Boxer first order logic output(after running it on the output given by the CCG parser) of the sentences is:

```
some(_G5981, some(_G6002, and(and(n_puzzle_1(_G5981),
and(a_first_1(_G6002), and(n_category_1(_G6002),
r_nn_1(_G5981, _G6002)))), some(_G6124,
and(v_person_1(_G6124), and(n_event_1(_G6124),
r_agent_1(_G6124, _G6002)))))))
```

and

```
some(_G3117, some(_G3287, some(_G3314, some(_G3252,
some(_G3345, some(_G3469, some(_G3499, some(_G3688,
some(_G3730, some(_G3647, some(_G3803, and(and(p_kirby_1(_G3117),
and(p_jim_1(_G3117), and(n_moon_1(_G3287),
and(p_armstrong_1(_G3314), and(p_luis_1(_G3314),
and(a_first_1(_G3252), and(n_man_1(_G3252),
and(v_nominate_1(_G3345), and(n_event_1(_G3345),
and(r_agent_1(_G3345, _G3287), and(r_patient_1(_G3345, _G3314),
and(r_on_1(_G3252, _G3287), and(a_male_1(_G3469),
and(n_partner_1(_G3499), and(r_of_1(_G3499, _G3469),
and(n_moon_1(_G3688), and(p_armstrong_1(_G3730),
```

```
and(p_luis_1(_G3730), and(a_first_1(_G3647), and(n_man_1(_G3647),
and(v_nominate_1(_G3803), and(n_event_1(_G3803),
and(r_agent_1(_G3803, _G3688), and(r_patient_1(_G3803, _G3730),
r_on_1(_G3647, _G3688))))))))))))))))))))))))), some(_G3154,
some(_G3166, some(_G3536, some(_G3548, and(v_ask_1(_G3154),
and(n_proposition_1(_G3166), and(n_event_1(_G3154),
and(r_agent_1(_G3154, _G3117), and(r_theme_1(_G3154, _G3166),
and(some(_G3412, and(v_name_1(_G3412), and(n_event_1(_G3412),
and(r_agent_1(_G3412, _G3117), r_patient_1(_G3412, _G3252))))),
and(v_ask_1(_G3536), and(n_proposition_1(_G3548), and(n_event_1(_G3536),
and(r_agent_1(_G3536, _G3499), and(r_theme_1(_G3536, _G3548),
some(_G3960, and(v_name_1(_G3960), and(n_event_1(_G3960),
and(r_agent_1(_G3960, _G3499), r_patient_1(_G3960, _G3647)
))))))))))))))))))))))))))))))
```

respectively. This can be better visualized as

```
exists X Y(
          n_puzzle_1(X) &
          a_first_1(Y) &
          n_category_1(Y) &
          r_nn_1(X, Y) &
          exists Z(
                 v_person_1(Z) &
                 n_event_1(Z) &
                 r_agent_1(Z, Y)
          )
)
```

and

```
exists X Y Z A B C D E F G H(
       p_kirby_1(X) & p_jim_1(X) &
       n_moon_1(Y) & p_armstrong_1(Z) &
       p_luis_1(Z) & a_first_1(A) &
       n_man_1(A) & v_nominate_1(B) &
       n_event_1(B) &
       r_agent_1(B, Y) &
       r_patient_1(B, Z) &
       r_on_1(A, Y) &
       a_male_1(C) & n_partner_1(D) &
       r_of_1(D, C) &
       n_moon_1(E) & p_armstrong_1(F) &
       p_luis_1(F) & a_first_1(G) &
       n_man_1(G) & v_nominate_1(H) &
       n_event_1(H)&
       r_agent_1(H, E) &
```

```
        r_patient_1(H, F) &
        r_on_1(G, E) &
        exists I J K L(
               v_ask_1(I) & n_proposition_1(J) &
               n_event_1(I) &
               r_agent_1(I, X) &
               r_theme_1(I, J) &
               exists M(
                      v_name_1(M) &
                      n_event_1(M) &
                      r_agent_1(M, X) &
                      r_patient_1(M, A)
               ) &
               v_ask_1(K) & n_proposition_1(_G3548)
               n_event_1(K) &
               r_agent_1(K, D) &
               r_theme_1(K, L) &
               exists N(
                       v_name_1(N) &
                       n_event_1(N) &
                       r_agent_1(N, D) &
                       r_patient_1(N, G)
               )
        )
)
```

respectively.

Let us now look more closely at the given output. Let us first check the format. The predicates are of the form $x\_name\_number(var)$, where 'x' is the specifier, 'name' is the actual word, 'number' is the wordnet meaning and 'var' is the variable assigned to it. The specifier gives the basic category of the word. Some of them are as follows:

- 'n' : noun
- 'p' : person
- 'o' : organization
- 'l' : location
- 'a' : adjective
- 'v' : verb
- 'r' : relation

When looking at the output, one can easily observe that the relations specify connections between the nouns. Boxer detects many types of relations including but not limited to: agent, patient, in, at, as, of and with.

Also note that the links between the words are given using both relations and variables, as the same variable can be used in place of multiple words (for example, 'X' is used for both 'Jim' and 'Kirby' thus connecting them), indicating

they are the same with respect to semantics. This information can then be used to effectively extract information from the sentence.

Another important thing to notice when looking at this example is that it seems like some parts of it are repeating itself, only with different variables. An example of it is the second existential quantifier, 'exists I J K L', where 'I' and 'J' are used in the exact same way and connect the exact same words as 'K' and 'L'. Also note that this 'doubling' was not in the parser output. The problem here seems to be the current version of Boxer. As we will discuss later, the translation is not perfect and particular words such as 'his' (which is present in the sentence) cause trouble. However, one must add that the overall quality of the obtained logic form is still good.

2.4 Obtaining the Answer Set Programs

Once we obtain the output form from Boxer, the next step is to translate the output given as a first order logic form into a set of AnsProlog facts. This is done in multiple steps, the main parts of the transformation are as follows.

Translating Boxer first order logic output to AnsProlog facts:

- Remove all quantifiers from the output
- Modify each predicate of the form $x\_name\_number(var)$, where 'x' is the specifier('n' for noun, 'v' for verb, etc.), 'name' is the actual word, 'number' is the wordnet meaning and 'var' is the variable assigned to the word. Change any such predicate into 'x(name, number, var).'. Note that this change is required to be able to reason about specific words and meanings in AnsProlog.
- In case of negation, replace the statement of the form $not(p_1, p_2, ..., p_n)$, where p_i is a predicate for $i = 1, 2, ..., n$, with $-p_1., -p_2., ..., -p_n.$. This is done to preserve negation and to be able to use it in reasoning.

Example 3. Let us recall the 2 examples.

- The first puzzle category is person
- Jim Kirby and his partner, asked to name the first man on the moon, nominated Luis Armstrong.

In this case, the given Boxer output will be translated to:

```
n(puzzle,1,g5981). a(first,1,g6002).
n(category,1,g6002).
r(nn,1,g5981, g6002).
v(person,1,g6124). n(event,1,g6124).
r(agent,1,g6124, g6002).

object(g5981).
object(g6002).
object(g6124).
```

and respectively into:

```
p(kirby,1,g3117). p(jim,1,g3117).
n(moon,1,g3287).
p(armstrong,1,g3314). p(luis,1,g3314).
a(first,1,g3252). n(man,1,g3252).
v(nominate,1,g3345). n(event,1,g3345).
r(agent,1,g3345, g3287).
r(patient,1,g3345, g3314).
r(on,1,g3252, g3287).
a(male,1,g3469). n(partner,1,g3499).
r(of,1,g3499, g3469).
n(moon,1,g3688). p(armstrong,1,g3730).
p(luis,1,g3730). a(first,1,g3647).
n(man,1,g3647).

object(g3117).
object(g3287).
object(g3314).
object(g3252).
object(g3345).
object(g3287).
object(g3469).
object(g3499).
object(g3688).
object(g3730).
object(g3647).
```

Notice that by performing this transformation, we keep all the information contained in the predicates and variables, but lose other information contained in the first order logic form given by the CCG parser as well as Boxer. Although this does not cause a problem with the applications that we are working on, in future one may also need to use the other knowledge contained in the logic form.

The translation also includes some more specific parts with regards to answer set solvers, such as Smodels [NS97] reserved words translation (for example Boxer output 'eq' is translated into 'equal', since 'eq' is a reserved word in Smodels), changing the case of some letters and adding domain predicates for each variable.

Once we obtain the AnsProlog facts from the text, we can combine them with the rest of the knowledge base.

3 An Application: Solving Puzzles

We now present an application of the given method. We will use the specified method to solve a set of puzzles obtained from [blp07]. These puzzles each happen to have 4 domains, each containing 5 elements. To solve any of these puzzles, one

needs to form tuples of 4 elements, where for each tuple it contains 1 element from each domain, while all the tuples are mutually exclusive(e.g. any element can only be in a single tuple). In addition, the conditions regarding the tuples are given in natural language and a solution must satisfy all of these conditions.

Let us now take a look at a particular puzzle, puzzle number 37 from [blp07].

Example 4 (Quiz game). The domain of the puzzle contains information about a person, his partner, the place they are from and question they have been asked.

- The persons are: Dick Eames, Helen Ingle, Jim Kirby, Joanna Long and Paul Riggs.
- The partners are: Ben Cope, Laura Mayes, Nick O'Hara, Ray Shaw and Tina Urquart
- The towns are: Blunderbury, Errordon, Messford, Slipwood and Wrongham.
- The questions are to name: Babylon wonder, First man on the moon, Mona Lisa artist, Six-day war length and Trafalgar commander.

There is a total of 7 clues given as follows:

1. The two ladies from Blunderbury were asked which of the Seven Wonders of the world was in Babylon and got it half right when they replied the hanging baskets.
2. Helen Ingle and her team-mate were from Errordon.
3. Jim Kirby and his partner, asked to name the first man on the Moon, nominated Louis Armstrong.
4. Laura Mayes, who was half of the team from Slipwood, and Tina Urquart were partnered with persons sharing a first name initial.
5. Ray Shaw and his partner both knew really that the British commander at Trafalgar wasn't Nelson Eddy but their minds went blank just at the wrong moment.
6. Ben Cope and Dick Eames, who isn't from Wrongham, were not partners in the game.
7. Paul Riggs and his partner Nick O'Hara were not the pair who, asked who painted Mona Lisa, replied Leonardo Da... um...Caprio.

Naturally, the goal of the puzzle is to figure out each pair of people, where they are from and what question they had been asked. In this case, the solution is:

(Joanna Long, Tina Urquart, Blunderbury, Babylon wonder)
(Paul Riggs, Nick O'Hara, Wrongham, Six-day war length)
(Jim Kirby, Laura Mayes, Slipwood, First man on the moon)
(Helen Ingle, Ben Cope, Errordon, Mona Lisa artist)
(Dick Eames, Ray Shaw, Messford, Trafalgar commander)

From this one can easily observe that to solve the puzzles in general, one can do the following. Extract the facts and clues from the text, enumerate all the possibilities for the extracted data and use clues to derive rules to impose

restrictions on the possible assignments. Thus in this case, one needs to extract both the puzzles facts and their respective clues into answer set programming facts and rules. Let us first look at the fact extraction.

However, looking back at the example one also notices that he needs to know which names corresponds to men and woman and which have the same initial. This additional knowledge is assumed to be included in the knowledge base.

3.1 Facts Extraction

The facts are extracted directly from specific input sentences. As said before, all the puzzles happen to have 4 domains, each of which contains 5 elements. The problem is that in many cases, some domain elements are not mentioned in the clues at all. Furthermore, many domains include elements similar to the other domain, for example first names, making it impossible to have a completely automatic extraction. Thus, we use the following type of English sentences to define the domains and their elements. (This is similar to the specification of domains in the Constraint Lingo system [FMT04].) For the given puzzle we have (assuming we take into account only first names and represent the questions by single word, we can as they are all different) the following:

```
The first puzzle category is person.
The second puzzle category is partner.
The third puzzle category is town.
The fourth puzzle category is question.
The persons are Dick, Helen,  Jim, Paul and Joanna.
The partners are Ben, Laura, Nick, Ray and Tina.
The towns are Blunderbury, Errordon, Messford, Slipwood and Wrongham.
The questions are wonder, moon, mona, war and Trafalgar.
```

The sentence always start with 'The', followed by one of the following: 'first', 'second', 'third', 'fourth'. This specifies which of the 4 domains the element it is in, for example 'first' means 1st domain, 'second' means 2nd etc. We use this to properly connect each puzzle category to one and only one domain. Next we have the specifier 'puzzle category is' followed by the respective category. In the first case, it is 'person' but it can be any word. Then we have a next sentence of the format 'The *category* are', where *category* can be any of the above categories followed by the elements of that particular category. Note that in the last case, we simplified the questions to one word. This was done in order to improve the performance of the system, in particular the clue extraction part. However, one might use the whole questions if necessary. These facts are automatically extracted and there is a set of specific rules that perform this extraction. Next a set of rules is used to enumerate all possible tuples of 4 elements. These rules are all contained in the knowledge base about puzzles. In this particular case, we use 2 predicates h and at, where the first specifies the first triple of a tuple, while the second predicate includes the first and last element of the tuple. For example $h(joanna, tina, blunderbury), at(joanna, wonder)$ represents a tuple $(joanna, tina, blunderbury, wonder)$. This tuple represents the

knowledge that Joanna and Tina were the pair from Blunderbury and were asked the question to name the Babylon wonder. The reason for splitting the tuples of 4 elements into smaller ones is to improve performance and also to allow us to have a simpler set of clues extraction rules.

3.2 Clues Extraction

The more interesting part of the natural language processing of the puzzle sentences is the clues extraction. Our initial approach is as follows. We have a set of templates of rules which specify the conditions of the puzzle; i.e., they specify which tuple assignments are correct. We then have an extraction system that 'supplies' these rules. We will illustrate the way it works on an example. Consider the following rule template[2]:

```
sat(O1) :- h(X,Y,Z), at(X,T), c1x(X,O1), c1y(Y,O1), c1z(Z,O1),
           c1t(T), cond(O1).
```

The head of the rule is the satisfaction predicate; a rule later on will then require this to be true thus making sure the condition holds for all answer sets. The 'h' and 'at' predicates are used to connect the domains of the puzzles. The template then allows us to specify 4 variables, e.g. 4 domains, first one, second one, third one and fourth one using predicates 'c1x', 'c1y', 'c1z' and 'c1t' respectively. Then the extraction system 'feeds' these predicates based on clues, making this a real rule. The last predicate says it's a positive rule and also specifies the rule identifier 'O1' used later to ensure all the rules created from templates are connected to the sentences they were extracted from, all of the extracted rules are satisfied and that we get only correct answer sets.

For example if we have the clue of the form 'Jim Kirby and his partner, asked to name the first man on the moon, nominated Luis Armstrong.', we can obtain(assuming we have the proper domains) that the first domain element is Jim and that the fourth is moon(again assuming simplified representations for questions). Then the extraction system would extract c1x(jim), c1y(Y), assuming Y is a variable representing the second domain, c1z(Z), again assuming Z represents the third domains, and c1t(moon). Then the above template rule would be equivalent to:

```
sat(O1) :- h(jim,Y,Z), at(jim,moon), cond(O1).
```

Note that this is done for all the clues. Clearly, the effectiveness of this system depends on the accurate extraction from the logic forms. Also, note that during extraction, we always pick a unique identifier, given by the $O1$ variable. This then ensures that the clue is only used for the cases we extracted.

The main idea when extracting clues is as follows. We try to connect each clue to a sentence of a particular type. In general, one might assume a sentence

[2] An alternative template where h is a binary predicate would work fine too. Also, our use of four domains each with five elements can be easily generalized where the numbers are not predetermined but are part of the puzzle.

has a subject, an object and a verb. Now depending on the verb, and the connection between subject and object, we will select which clue will be used for the sentence. To do this, we look for relation(s) which connects the subject and the object, and if they are present in the facts representing the sentence. For example for the sentence 'Jim Kirby and his partner, asked to name the first man on the moon, nominated Luis Armstrong.' and the given clue template, we use the following set of extraction rules:

```
extract_template_1(X,Y,Z,T,O1) :- n(man,1,O6), r(on,1,O6,O7),
  l(T,1,O7), p(X,1,O4), r(agent,1,O5,O4), v(nominate,1,O5),
  v(ask,1,O1),v(name,1,O2), r(agent,1,O1,O3), r(agent,1,O2,O3).
c1x(X, O1) :- extract_template_1(X,Y,Z,T,O1).
c1y(Y, O1) :- extract_template_1(X,Y,Z,T,O1).
c1z(Z, O1) :- extract_template_1(X,Y,Z,T,O1).
c1t(T, O1) :- extract_template_1(X,Y,Z,T,O1).
cond(O1) :- extract_template_1(X,Y,Z,T,O1).
```

Here assume X, Y, Z, T are variables representing the four puzzle domains respectively and Oi, where i is a positive integer, represents any variable given by boxer. In this case, we use 'O1' as the identifier of the sentence. Note that it does not matter which one of the sentence variables we use, since boxer uses exclusive variables for each sentence. This kind of extraction is performed for all the clues.

To ensure this condition is satisfied later, we have the following rule:

```
:- not sat(O1), cond(O1).
```

This rule guarantees all of the extracted conditions are satisfied.

So far we have discussed the extraction of facts and clues. Now let us look at the enumeration part. This is done using the following simple set of rules:

```
nh(X,Y,Z) :-  h(X,Y1,Z1), Z != Z1.
nh(X,Y,Z) :-  h(X,Y1,Z1), Y != Y1.
nh(X,Y,Z) :-  h(X1,Y1,Z), X != X1.
nh(X,Y,Z) :-  h(X1,Y1,Z), Y != Y1.
nh(X,Y,Z) :-  h(X1,Y,Z1), Z != Z1.
nh(X,Y,Z) :-  h(X1,Y,Z1), X != X1.

h(X,Y,Z) :- d1(X), d2(Y),d3(Z), not nh(X,Y,Z).

n_at(X,T) :- at(X1,T), X != X1.
n_at(X,T) :- at(X,T1), T != T1.
at(X,T) :- d1(X), d4(T), not n_at(X,T).
```

These rules enumerate all the possible combinations. The predicates $d1, d2, d3$ and $d4$ represent the four respective domains of puzzle elements. The 'h' predicate describes the first three domains; the 'at' predicate domains 1 and 4; and domain 1 elements are used as the identifiers.

Let us now provide an idea on how to use the system. With both facts extraction and puzzle specification already present, the system works in a simple way. The system takes English sentences for facts and the English sentences representing the clues of a particular puzzle. It transforms the puzzle domain information and clues into first order logic using CCG parser and Boxer, then uses a script to make them answer set programming facts. Then the general knowledge and extraction rules are added and the answer sets are computed. The resulting answer sets represent the solutions to the puzzles.

Thus, combining the extracted facts, the interface which 'feeds' the rule templates, and the general module about puzzles that includes enumeration, rule templates and some additional information (for example for the given puzzle we need information about names first initial and men/woman names) we obtain an AnsProlog program with the following answer set:

```
h(joanna,tina,blunderbury)
h(paul,nick,wrongham)
h(jim,laura,slipwood)
h(helen,ben,errordon)
h(dick,ray,messford)

at(paul,war)
at(joanna,wonder)
at(jim,moon)
at(helen,mona)
at(dick,trafalgar)
```

One can easily verify that it corresponds to the solution of the puzzle.

Let us now present some of the current limitations of the system. First, since the system uses a set of external tools, it automatically inherits the issues that come with them. At the moment, the most problematic part seems to be current version of the C&C parser and Boxer. In case of the parser, it has problems recognizing most types of quotes. Also, many words not found in its dictionary are not dealt with properly. In case of Boxer, since it uses the output of the parser, the same problems persist. Also, in case of some sentences, particularly longer ones, they seem to be translated incorrectly by Boxer despite the fact that the output of the parser seems correct. We hope at least some of these issues will be fixed by the next release of C&C tools.

4 Other Applications

In this section we briefly discuss two other applications involving Answer Set Prolog and natural language.

4.1 Reasoning About Travel

In this subsection, we use examples where there are AnsProlog rules encoding background knowledge about travel and certain narrative about a person's travel

given in English and a question asked in English. Consider the following story about John's travel.

1. John took a flight from Paris to Baghdad in the morning of June 15.
2. John would take his laptop on the plane with him.
3. A few hours later the plane stopped in Rome, where John was arrested.
4. The police confiscate John's possession.

By using a natural language understanding system for the above story, we extracted the following facts.

– Facts extracted from sentence 1:

```
h(timepoint(morning),1).  object(flight).
h(origin(flight,paris),1).
city(paris). o(take(john,flight),1).  person(john).
h(dest(flight,baghdad),1). city(baghdad).
h(date(june,15),1).
day(15).  month(june).
```

– Facts extracted from sentence 2:

```
o(take(john,laptop),2). person(john). object(laptop).
h(is_in(john,plane),2). vehicle(plane).
h(owned_by(laptop,john),2).
```

– Facts extracted from sentence 3:

```
o(stop_in(plane,rome),3). o(arrest_in(john,rome),4).
h(arrested_in(john,rome),5). h(timepoint(afternoon),3).
vehicle(plane). person(john). city(rome).
```

– Facts extracted from sentence 4:

```
o(confiscate_pos_of(police,john),5). person(john).
person(police).
```

The predicates h and o indicate *holds* and *occurs* for the above facts. For instance, $o(take(john, flight), 1)$ means that the action about John taking a flight occurs at timepoint 1.

With the above story about John, let's suppose we are interested in asking the following questions: (1) Where is John on June 15 evening? (2) Who has the laptop? The questions are automatically translated into the following AnsProlog rules:

– AnsProlog rule for question 1:

```
answer_Q1(C) :- city(C), h(is_in(P,C),T), h(date(june,15),T),
                h(timepoint(evening),T), P=john.
```

– AnsProlog rule for question 2:

```
answer_Q2(P) :- person(P), h(owned_by(laptop,P),n).
```

In the AnsProlog rule for question 2, n is a constant that is assumed to be the last time point of the travel story. To answer the first question, we need to be able to reason that normally a person P is in city $C2$ if the destination of his trip X is $C2$. There can be exceptions to this rule if for some reason, his trip is aborted at a certain timepoint. One of the possible reasons for P's trip to be aborted is that P is arrested in another city $C1$.

```
h(is_in(P,C2),n) :- person(P), city(C1;C2), object(X),
                    h(origin(X,C1),T), h(dest(X,C2),T),
                    not h(aborted(P,X),T1), T < T1.
h(aborted(P,X),T+1) :- person(P), city(C), object(X),
                       o(arrest_in(P,C),T),
                       h(dest(X,C1),T1), C != C1.
```

By using the above rules, the fact $h(is\_in(john, baghdad), n)$ cannot be inferred, as $h(aborted(john, trip), 5)$ becomes true due to the extracted fact $o(arrest\_in(john, rome), 4)$. With the fact $h(arrested\_in(john, rome), 5)$, we can infer $h(is\_in(john, rome, 5)$ using the following rule:

```
h(is_in(P,C),T) :- person(P), city(C), h(arrested_in(P,C),T).
```

Using the inertia axiom, the facts $h(is\_in(john, rome), n)$ and $h(date(june, 15), n)$ are inferred. We assume that the last timepoint n corresponds to evening when the previous to last timepoint $n-1$ is afternoon and no timepoint is assigned for evening by the following rule:

```
h(timepoint(evening),n) :- h(timepoint(afternoon),n-1),
                           not h(timepoint(evening),T).
```

With the facts $h(is\_in(john, rome), n)$, $h(date(june, 15), n)$ and $h(timepoint(evening), n)$, we are able to infer the final answer $answer\_Q1(rome)$ for the first question.

To answer the second question, the system needs to reason with the fact that an ownership of an object OBJ is transferred from person $P1$ to P if P confiscates $P1$'s possession. This is captured by the following two AnsProlog rules:

```
h(owned_by(OBJ,P),T+1) :- person(P;P1), object(OBJ),
       o(confiscate_pos_of(P,P1),T), h(owned_by(OBJ,P1),T).
-h(owned_by(OBJ,P1),T+1) :- person(P;P1), object(OBJ),
       o(confiscate_pos_of(P,P1),T), h(owned_by(OBJ,P1),T).
```

With the above rules, the facts $h(owned\_by(laptop, police), 6)$ and $-h(owned\_by(laptop, john), 6)$ are inferred so that we can infer the answer $answer\_Q2(police)$ for the second question.

4.2 RTE Examples

RTE is an acronym for "Recognizing Textual Entailment." In an RTE task, given a hypothesis-text pair, *h-t*, the goal is to be able to answer "yes" if the system verifies that h can be entailed from t. The module returns "no" if it realizes that h cannot be entailed from t, while it returns "unknown" if a conclusion cannot be reached. Natural language processing is used to extract AnsProlog facts $AF(h)$ and $AF(t)$ corresponding both h and t. Query rules $R(h)$ are generated from $AF(h)$ using a rule generator, and rules $Th(h,t)$ relevant to h and t are used for reasoning. To determine if h entails t, we use the answer set solver Smodels [NS97] with respect to $Th(h,t)$, $R(h)$ and $AF(t)$. We now illustrate this with respect to two *h-t* pairs from RTE [DGM06, BHDD$^+$06].

Example 5 (Yoko Ono Example).

- T: Yoko Ono unveiled a bronze statue of her late husband, John Lennon, to complete the official renaming of England's Liverpool Airport as Liverpool John Lennon Airport.
- H: Yoko Ono is John Lennon's widow.

After using NLP, the following facts $AF(h)$, $AF(t)$ are extracted from both h, t:

$AF(h) = \{h(widow\_of(x3, x6), 1), name(x6, john), name(x3, yoko),$
$name(x6, lennon), name(x3, ono)\}$
$AF(t) = \{h(husband\_of(x1, x), 0), o(pass\_away(x1), 0)), name(x1, john),$
$name(x1, lennon), name(x, yoko), name(x, ono)\}$.

The rule generator of the RTE module takes $AF(h)$ and translates it into the following AnsProlog rules $R(h)$:

```
answer(yes) :- h(widow_of(X3,X6),1),name(X6,john),
               name(X3,yoko),name(X6,lennon),name(X3,ono).
answer(no) :- -h(widow_of(X3,X6),1),name(X6,john),
               name(X3,yoko),name(X6,lennon),name(X3,ono).
answer(unknown) :- not answer(yes), not answer(no).
```

Using the following rule, $Th(h,t)$, that encodes that X is a widow if her husband $X1$ passed away together with the facts $AF(t)$ and query rules $R(h)$, the system returns the fact $answer(yes)$ to indicate that h entails t.

```
h(widow_of(X,X1),T+1) :- o(pass_away(X1),T),
                         h(husband_of(X1,X),T).
```

Now let us consider another RTE example.

Example 6 (Linnaean Society).

- T: Robinson became a fellow of the Linnaean Society at the age of 29 and a successful gardening correspondent of The Times.
- H: Robinson was a member of the Linnaean Society.

After natural language processing, the following facts $AF(h)$, $AF(t)$ are extracted from both h, t:

$AF(h) = \{\ name(x1, robinson),\ name(x5, linnaean),\ name(x5, society),\ h(is(x1, member(x5)), 1)\ \}$
$AF(t) = \{\ name(x1, fellow),\ name(x1, robinson),\ name(x5, linnaean),\ name(x5, society),\ o(become(x1, fellow(x5)), 0)\ \}$

The rule generator of the RTE module takes $AF(h)$ and translates it into the following AnsProlog rules $R(h)$:

```
answer(yes) :- h(is(X1,member(X5)),1),name(X1,robinson),
               name(X5,linnaean),name(X5,society).
answer(no) :- -h(is(X1,member(X5)),1),name(X1,robinson),
                 name(X5,linnaean),name(X5,society).
answer(unknown) :- not answer(yes), not answer(no).
```

The following static and dynamic causal rules, $Th(h, t)$, are used to describe person $X1$ is a member/fellow of society $X2$ if $X1$ becomes a member/fellow of $X2$, while $X1$ is a member of $X2$ if $X1$ is a fellow of $X2$.

```
h(is(X1,member(X2)),T+1) :- o(become(X1,member(X2)),T).

h(is(X1,fellow(X2)),T+1) :- o(become(X1,fellow(X2)),T).

h(is(X1,member(X2)),T) :- h(is(X1,fellow(X2)),T).

-h(is(X1,fellow(X2)),T) :- -h(is(X1,member(X2)),T).
```

With the rules $Th(h, t)$, the facts $AF(t)$ and the query rules $R(h)$, the system then returns the fact $answer(yes)$ to indicate that h indeed entails t.

5 Conclusions

In this paper we discussed initial steps taken towards automatically extracting knowledge from natural language text and using them together with other hand coded knowledge to do reasoning and problem solving. Research in this direction has the potential to alleviate the knowledge acquisition bottleneck that one faces when building knowledge bases.

We briefly discussed three examples: solving puzzles, answering questions about travel and recognizing textual entailment. We reiterate that these are only initial steps and a lot remains to be done. For example, although our formulation could solve several puzzles, it needs to be further generalized to further broaden the class of puzzles it can solve. In many puzzles there are relationship between the elements of the domain and the clues use those relations. Some of these issues are also mentioned in [FMT04]. Thus the formulation in this paper needs to be further improved to solve a broader class of puzzles.

In these initial works we have focused on obtaining facts from the natural langauge processing part. However, the NLP systems that we use produce logic forms and a next step would be to extract more general knowledge (such as simple rules), in the AnsProlog syntax, from the logic forms and use them.

An interesting dimension of working on natural language based applications is that we have come across interesting knowledge representation and reasoning issues such as reasoning about intentions [BG05] and reasoning about the cardinality of dynamic sets. An example of the later is to find out the cardinality of a particular class based on English statements about elements joining and leaving that class or a subclass. We were told that among the 28 questions that could not be answered by any of the TREC QA systems in 2006, 6 of them involved such reasoning about cardinality of sets. The lesson we learned from these attempts is that reasoning with respect to natural language text and queries not only involves using natural language processing but also sometimes requires developing interesting knowledge representation rules, such as rules to reason about membership and cardinality of sets in presence of incomplete knowledge.

Acknowledgements

We thank Yulia Lierler, Steve Maiorano, Jean Michel Pomarede, Michael Gelfond, Richard Scherl, Marcello Balduccini and Vladimir Lifschitz for discussions on the topic of this paper. We thank DTO for its support of this research.

References

[Bar03] Baral, C.: Knowledge representation, reasoning and declarative problem solving. Cambridge University Press, Cambridge (2003)

[BCS+04] Bos, J., Clark, S., Steedman, M., Curran, J.R., Hockenmaier, J.: Wide-coverage semantic representations from a ccg parser. In: Proceedings of the 20th International Conference on Computational Linguistics (COLING '04), Geneva, Switzerland (2004)

[BDT06] Baral, C., Dzifcak, J., Takahashi, H.: Macros, macro calls and use of ensembles in modular answer set programming. In: Etalle, S., Truszczyński, M. (eds.) ICLP 2006. LNCS, vol. 4079, pp. 376–390. Springer, Heidelberg (2006)

[BG05] Baral, C., Gelfond, M.: Reasoning about intended actions. In: Proceedings of AAAI 05, pp. 689–694 (2005)

[BGR04] Baral, C., Gelfond, M., Rushton, N.: Probabilistic reasoning with answer sets. In: Lifschitz, V., Niemelä, I. (eds.) Logic Programming and Nonmonotonic Reasoning. LNCS (LNAI), vol. 2923, pp. 21–33. Springer, Heidelberg (2003)

[BHDD+06] Bar-Haim, R., Dagan, I., Dolan, B., Ferro, L., Giampiccolo, D., Magnini, B., Szpektor, I.: The second pascal recognising textual entailment challenge. In: Proceedings of the Second PASCAL Challenges Workshop on Recognising Textual Entailment, Springer, Heidelberg (2006)

[blp07] England's best logic problems. Penny Press (Spring, 2007)

[CC07] Clark, S., Curran, J.R.: Wide-coverage efficient statistical parsing with ccg and log-linear models. Computational Linguistics (to appear, 2007)

[CCt] http://svn.ask.it.usyd.edu.au/trac/candc/wiki/

[DGM06] Dagan, I., Glickman, O., Magnini, B.: The pascal recognising textual entailment challenge. In: Quiñonero-Candela, J., Dagan, I., Magnini, B., d'Alché-Buc, F. (eds.) MLCW 2005. LNCS (LNAI), vol. 3944, pp. 177–190. Springer, Heidelberg (2006)

[dVS07] de Vos, M., Schaub, T. (eds.): Proceedings of LPNMR'07 Workshop on Software engineering for answer set programming, SEA'07 (2007)

[FAW+04] Friedland, N., Allen, P., Witbrock, M., Angele, J., Staab, S., Israel, D., Chaudhri, V., Porter, B., Barker, K., Clark, P.: Towards a quantitative, platformindependent analysis of knowledge systems. In: Proceedings of the Ninth International Conference on the Principles of Knowledge Representation and Reasoning (KR 2004), pp. 507–515 (2004)

[FMT04] Finkel, R., Marek, M., Truszczyn'ski, M.: Constraint lingo: towards high-level constraint programming. Software—Practice & Experience 34(15), 1481–1504 (2004)

[GKNS07] Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Clasp: A conflict-driven answer set solver. In: Baral, C., Brewka, G., Schlipf, J. (eds.) LPNMR 2007. LNCS (LNAI), vol. 4483, pp. 260–265. Springer, Heidelberg (2007)

[GL88] Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: Kowalski, R., Bowen, K. (eds.) Logic Programming: Proc. of the Fifth Int'l Conf. and Symp., pp. 1070–1080. MIT Press, Cambridge (1988)

[GL91] Gelfond, M., Lifschitz, V.: Classical negation in logic programs and disjunctive databases. New Generation Computing 9, 365–387 (1991)

[Lie05] Lierler, Y.: Cmodels - sat-based disjunctive answer set solver. In: LPNMR, pp. 447–451 (2005)

[LZ02] Lin, F., Zhao, Y.: Assat: Computing answer sets of a logic program by sat solvers. In: Proceedings of AAAI-02 (2002)

[MBF+90] Miller, G., Beckwith, R., Fellbaum, C., Gross, D., Miller, K.: Introduction to wordnet: An on-line lexical database. International Journal of Lexicography (special issue) 3(4), 235–312 (1990)

[MHG+02] Moldovan, D., Harabagiu, S., Girju, R., Morarescu, P., Novischi, A., Lacatusu, F., Badulescu, A., Bolohan, O.: Lcc tools for question answering. In: Voorhees, E., Buckland, L., (eds) Proceedings of TREC 2002 (2002)

[MPHS03] Moldovan, D., Pasca, M., Harabagiu, S., Surdeanu, M.: Performance issues and error analysis in an open-domain question answering system. ACM Transaction on Information Systems 21(2), 133–154 (2003)

[NS97] Niemalä, I., Simons, P.: Smodels - an implementation of the stable model and well-founded semantics for normal logic programs. In: proceedings of the 4th International Conference of Logic Programming and Nonmonotonic Reasoning, pp. 420–429 (1997)

[RTE] http://www.pascal-network.org/challenges/rte/

[Woo93] Mary McGee Wood. Categorial grammars. Routledge (1993)

Preferences, Contexts and Answer Sets

Gerhard Brewka

Universität Leipzig, Institut für Informatik
Postfach 10 09 20, D-04009 Leipzig, Germany
brewka@informatik.uni-leipzig.de

Abstract. Answer set programming (ASP) is a declarative programming paradigm based on logic programs under stable model semantics, respectively its generalization to answer set semantics. Besides the availability of rather efficient answer set solvers, one of the major reasons for the success of ASP in recent years was the shift from a theorem proving to a constraint programming view: problems are represented such that stable models, respectively answer sets, rather than theorems correspond to solutions.

It is obvious that preferences play an important role in everyday decision making - and in many AI applications. For this reason a number of approaches combining answer set programming with explicit representations of preferences have been developed over the last years. The approaches can be roughly categorized according to the preference representation (quantitative vs. qualitative) the type of preferences they allow (static vs. dynamic) and the objects of prioritization (rules vs. atoms/formulas).

We will focus on qualitative dynamic formula preferences, give an account of existing approaches and show that by adding adequate optimization constructs one obtains interesting solutions to problems in belief merging, consistency handling, game theory and social choice.

Explicit representations of contexts also have quite a tradition in AI, going back to foundational work of John McCarthy. A context, intuitively, is a particular view of a state of affairs. Contexts can also be used as representations of beliefs of multiple agents.

We show how multi-context systems based on bridge rules, as developed by Fausto Giunchiglia and colleagues in Trento, can be extended to nonmonotonic context systems. We first discuss multi-context logic programming systems, and then generalize the ideas underlying these systems to a general framework for integrating arbitrary logics, monotonic or nonmonotonic. Techniques from answer set programming are at the heart of the framework.

We finally give a brief outlook on how the two main topics of the talk, preferences and contexts, can be combined fruitfully.

Several of the presented results were obtained in cooperation with Thomas Eiter, Ilkka Niemelä and Mirek Truszczyński.

V. Dahl and I. Niemelä (Eds.): ICLP 2007, LNCS 4670, p. 22, 2007.

Answer Set Programming for the Semantic Web (Tutorial)*,**

Thomas Eiter

Institut für Informationssysteme, Technische Universität Wien
Favoritenstraße 9-11, A-1040 Vienna, Austria
eiter@kr.tuwien.ac.at

Abstract. The *Semantic Web* [1, 2, 3] aims at extending the current Web by standards and technologies that help machines to understand the information on the Web so that they can support richer discovery, data integration, navigation, and automation of tasks. Its development proceeds in layers, and the Ontology layer is the highest one that has currently reached a sufficient maturity, in the form of the *OWL Web Ontology Language (OWL)* [4, 5], which is based on Description Logics. Current efforts are focused on realizing the Rules layer, which should complement the Ontology layer and offer sophisticated representation and reasoning capabilities. This raises, in particular, the issue of interlinking rules and ontologies. Excellent surveys that classify many proposals for combining rules and ontologies are [6, 7]; general issues that arise in this are discussed e.g. in [8, 9, 10]. Notably, the World Wide Web Consortium (W3C) has installed The Rule Interchange Format (RIF) Working Group on order to produce a core rule language plus extensions which together allow rules to be translated between rule languages and thus transferred between rule systems; a first working draft has been released recently.

Answer Set Programming (ASP) [11, 12, 13, 14], also called A-Prolog [15, 16, 17], is a well-known declarative programming paradigm which has its roots in Logic Programming and Non-monotonic Reasoning [18]. Thanks to its many extensions [19], ASP is well-suited for modeling and solving problems which involve common sense reasoning, and has been fruitfully applied to a range of applications including data integration, configuration, diagnosis, text mining, reasoning about actions and change, etc.; see [16, 17, 20].

Within the context of the Semantic Web, the usage of ASP and related formalisms has been explored in different directions:

– On the one hand, they have been exploited as a tool to encode reasoning tasks in Description Logics, like [16, 22, 23, 24, 25, 26, 27].

* This tutorial is based on material and results which has been obtained in joint work with Giovambattista Ianni (Università della Calabria), Thomas Krennwallner (TU Wien), Thomas Lukasiewicz (Università di Roma "La Sapienza"), Axel Polleres (DERI Galway), Roman Schindlauer (TU Wien), and Hans Tompits (TU Wien).

** The work has been partially supported by the EC NoE REWERSE (IST 506779) and the Austrian Science Fund (FWF) project P17212-N04.

V. Dahl and I. Niemelä (Eds.): ICLP 2007, LNCS 4670, pp. 23–26, 2007.

– On the other hand, they have been used as a basis for giving a semantics to a combination of rules and ontologies. Here, increasing levels of integration have been considered:
  - *loose couplings*, where rule and ontology predicates are separated, and the interaction is via a safe semantic interface like an inference relation e.g. [28, 29, 30, 31, 32]
  - *tight couplings*, where rule and ontology predicates are separated, and the interaction is at the level of models, e.g. [33, 34, 35, 36, 37, 38, 39, 10, 40]; and
  - *full integration*, where no distinction between rule and ontology predicates is made, e.g., [41, 42, 43, 44].

In this tutorial, we will first briefly review ASP and ontology formalisms. We then will recall some of the issues that come up with the integration of rules and ontologies. After that, we will consider approaches to combine rules and ontologies under ASP, where particular attention well be devoted to non-monotonic description logic programs [45] and its derivatives [28, 46] as a representative of loose couplings. However, also other approaches will be discussed. We further discuss the potential of such combinations, some applications, and finally some open issues.

References

1. Berners-Lee, T.: Weaving the Web. Harper, San Francisco, CA (1999)
2. Berners-Lee, T., Hendler, J., Lassila, O.: The Semantic Web. Scientific American 284, 34–43 (2001)
3. Fensel, D., Wahlster, W., Lieberman, H., Hendler, J. (eds.): Spinning the Semantic Web: Bringing the World Wide Web to Its Full Potential. MIT Press, Cambridge (2002)
4. W3C: OWL Web ontology language overview (2004), Available at `http://www.w3.org/TR/2004/~REC-owl-features-20040210/`
5. Horrocks, I., Patel-Schneider, P.F., van Harmelen, F.: From $\mathcal{SHIQ}$ and RDF to OWL: The making of a Web ontology language. J. Web Sem. 1, 7–26 (2003)
6. Antoniou, G., Damásio, C.V., Grosof, B., Horrocks, I., Kifer, M., Maluszynski, J., Patel-Schneider, P.F.: Combining rules and ontologies: A survey. Technical Report IST506779/Linköping/I3-D3/D/PU/a1, Linköping University (2005)
7. Pan, J.Z., Franconi, E., Tessaris, S., Stamou, G., Tzouvaras, V., Serafini, L., Horrocks, I., Glimm, B.: Specification of coordination of rule and ontology languages. Project Deliverable D2.5.1, KnowledgeWeb NoE (2004)
8. de Bruijn, J, Eiter, T., Polleres, A., Tompits, H.: On representational issues about combinations of classical theories with nonmonotonic rules. In: Lang, J., Lin, F., Wang, J. (eds.) KSEM 2006. LNCS (LNAI), vol. 4092, pp. 1–22. Springer, Heidelberg (2006)
9. Eiter, T., Ianni, G., Polleres, A., Schindlauer, R., Tompits, H.: Reasoning with rules and ontologies. In: Barahona, P., Bry, F., Franconi, E., Henze, N., Sattler, U. (eds.) Reasoning Web. LNCS, vol. 4126, pp. 93–127. Springer, Heidelberg (2006)
10. Rosati, R.: Integrating ontologies and rules: Semantic and computational issues. In: Barahona, P., Bry, F., Franconi, E., Henze, N., Sattler, U. (eds.) Reasoning Web. LNCS, vol. 4126, pp. 128–151. Springer, Heidelberg (2006)

11. Provetti, A., Cao, S.T. (eds.): Proc. AAAI 2001 Spring Symposium on Answer Set Programming: Towards Efficient and Scalable Knowledge Representation and Reasoning. AAAI Press, Stanford (2001)
12. Lifschitz, V.: Answer Set Programming and Plan Generation. Artificial Intelligence 138, 39–54 (2002) (Seminal paper at ICLP'99 (invited talk), coining the term Answer Set Programming)
13. Marek, V.W., Truszczyński, M.: Stable Models and an Alternative Logic Programming Paradigm. In: Apt, K., Marek, V.W., Truszczyński, M., Warren, D.S. (eds.) The Logic Programming Paradigm – A 25-Year Perspective, pp. 375–398. Springer, Heidelberg (1999)
14. Niemelä, I.: Logic Programming with Stable Model Semantics as Constraint Programming Paradigm. Ann. Math. and Artif. Int. 25, 241–273 (1999)
15. Balduccini, M., Gelfond, M.: Diagnostic reasoning with A-Prolog. Theory and Practice of Logic Programming 3, 425–461 (2003)
16. Baral, C.: Knowledge Representation, Reasoning, and Declarative Problem Solving. Cambridge University Press, Cambridge (2002)
17. Gelfond, M.: Representing knowledge in A-Prolog. In: Kakas, A.C., Sadri, F. (eds.) Computational Logic: Logic Programming and Beyond. LNCS(LNAI), vol. 2408, pp. 413–451. Springer, Heidelberg (2002)
18. Gelfond, M., Lifschitz, V.: Classical negation in logic programs and disjunctive databases. New Generation Computing 9, 365–385 (1991)
19. Niemelä, I. (ed.): Language extensions and software engineering for ASP. Tech. Rep. WP3, Working Group on Answer Set Programming (WASP, IST-FET-2001-37004) (2005)
`http://www.tcs.hut.fi/Research/Logic/wasp/wp3/wasp-wp3-web/`
20. Woltran, S.: Answer set programming: Model applications and proofs-of-concept. Tech. Rep. WP5, Working Group on Answer Set Programming (WASP, IST-FET-2001-37004) (2005) `http://www.kr.tuwien.ac.at/projects/WASP/report.html`
21. Bertino, E., Provetti, A., Salvetti, F.: Local closed-world assumptions for reasoning about semantic web data. In: Buccafurri, F. (ed.) Proc. APPIA-GULP-PRODE, pp. 314–323 (2003)
22. Van Belleghem, K., Denecker, M., De Schreye, D.: A strong correspondence between description logics and open logic programming. In: Proc. ICLP-1997, pp. 346–360. MIT Press, Cambridge (1997)
23. Alsaç, G., Baral, C.: Reasoning in description logics using declarative logic programming. Tech. Rep. CS&E Dept, Arizona State University (2001)
24. Swift, T.: Deduction in ontologies via ASP. In: Lifschitz, V., Niemelä, I. (eds.) Logic Programming and Nonmonotonic Reasoning. LNCS (LNAI), vol. 2923, pp. 275–288. Springer, Heidelberg (2003)
25. Hustadt, U., Motik, B., Sattler, U.: Reducing SHIQ-description logic to disjunctive datalog programs. In: Proc. KR-2004, pp. 152–162. AAAI Press, Stanford (2004)
26. Heymans, S., Vermeir, D.: Integrating ontology languages and answer set programming. In: Mařík, V., Štěpánková, O., Retschitzegger, W. (eds.) DEXA 2003. LNCS, vol. 2736, pp. 584–588. Springer, Heidelberg (2003)
27. Heymans, S., Vermeir, D.: Integrating semantic web reasoning and answer set programming. In: Proc. ASP- 2003, pp. 194–208 (2003)
28. Eiter, T., Ianni, G., Schindlauer, R., Tompits, H.: A uniform integration of higher-order reasoning and external evaluations in answer-set programming. In: Proc. IJCAI-2005, Professional Book Center, pp. 90–96 (2005)
29. Lukasiewicz, T.: Probabilistic description logic programs. In: Godo, L. (ed.) ECSQARU 2005. LNCS (LNAI), vol. 3571, pp. 737–749. Springer, Heidelberg (2005)

30. Lukasiewicz, T.: Fuzzy description logic programs under the answer set semantics for the Semantic Web. In: Proc. RuleML-2006, pp. 89–96. IEEE Computer Society Press, Los Alamitos (2006)
31. Wang, K., Antoniou, G., Topor, R.W., Sattar, A.: Merging and aligning ontologies in dl-programs. In: Adi, A., Stoutenburg, S., Tabet, S. (eds.) RuleML 2005. LNCS, vol. 3791, pp. 160–171. Springer, Heidelberg (2005)
32. Yang, F., Chen, X., Wang, Z.: p-dl-programs: Combining dl-programs with preference for Semantic Web. Manuscript (2006)
33. Grosof, B.N., Horrocks, I., Volz, R., Decker, S.: Description logic programs: Combining logic programs with description logics. In: Proc. WWW-2003, pp. 48–57. ACM Press, New York (2003)
34. Horrocks, I., Patel-Schneider, P.F., Boley, H., Tabet, S., Grosof, B., Dean, M.: SWRL: A Semantic Web rule language combining OWL and RuleML, W3C Member Submission. (2004), http://www.w3.org/Submission/SWRL/
35. Donini, F.M., Lenzerini, M., Nardi, D., Schaerf, A.: $\mathcal{AL}$-log: Integrating datalog and description logics. J. Intell. Inf. Syst. 10, 227–252 (1998)
36. Levy, A.Y., Rousset, M.C.: Combining Horn rules and description logics in CARIN. Artificial Intelligence 104, 165–209 (1998)
37. Rosati, R.: Towards expressive KR systems integrating datalog and description logics: Preliminary report. In: Proc. DL- 1999, pp. 160–164 (1999)
38. Rosati, R.: On the decidability and complexity of integrating ontologies and rules. J. Web Sem. 3, 61–73 (2005)
39. Rosati, R.: $\mathcal{DL}$+*log*: Tight integration of description logics and disjunctive datalog. In: Proc. KR 2006, pp. 68–78. AAAI Press, Stanford (2006)
40. Yang, F., Chen, X.: DLclog: A hybrid system integrating rules and description logics with circumscription. In: Proc. DL 07 (2007), http://www.inf.unibz.it/krdb/events/dl-2007/
41. Motik, B., Horrocks, I., Rosati, R., Sattler, U.: Can OWL and logic programming live together happily ever after? In: Cruz, I., Decker, S., Allemang, D., Preist, C., Schwabe, D., Mika, P., Uschold, M., Aroyo, L. (eds.) ISWC 2006. LNCS, vol. 4273, pp. 501–514. Springer, Heidelberg (2006)
42. de Bruijn, J., Pearce, D., Polleres, A., Valverde, A.: A logic for hybrid rules. In: Proc. RuleML 2006, IEEE Computer Society Press, Washington (2006), http://2006.ruleml.org/online-proceedings/rule-integ.pdf
43. de Bruijn, J., Pearce, D., Polleres, A., Valverde, A.: Quantified equilibrium logic and hybrid rules. In: Marchiori, M., Pan, J.Z., de Sainte Marie, C. (eds.) RR 2007. LNCS, vol. 4524, Springer, Heidelberg (2007)
44. Motik, B., Rosati, R.: A faithful integration of description logics with logic programming. In: Proc. IJCAI 2007, pp. 477–482. AAAI Press, Stanford (2007)
45. Eiter, T., Lukasiewicz, T., Schindlauer, R., Tompits, H.: Combining Answer Set Programming with Description Logics for the Semantic Web. In: Proc. KR 04, pp. 141–151 (2004)
46. Eiter, T., Ianni, G., Krennwallner, T., Schindlauer, R.: Exploiting conjunctive queries in description logic programs. In: Proc. DL 07 (2007), http://www.inf.unibz.it/krdb/events/dl-2007/

Coinductive Logic Programming and Its Applications

Gopal Gupta, Ajay Bansal, Richard Min, Luke Simon, and Ajay Mallya

Department of Computer Science,
University of Texas at Dallas,
Richardson, TX 75080

Abstract. Coinduction has recently been introduced as a powerful technique for reasoning about unfounded sets, unbounded structures, and interactive computations. Where induction corresponds to least fixed point semantics, coinduction corresponds to greatest fixed point semantics. In this paper we discuss the introduction of coinduction into logic programming. We discuss applications of coinductive logic programming to verification and model checking, lazy evaluation, concurrent logic programming and non-monotonic reasoning.

1 Introduction

Recently *coinduction* has been introduced as a technique for reasoning about unfounded sets [10], behavioral properties of programs [4,7], and proving liveness properties in model checking [13]. Coinduction also serves as the foundation for lazy evaluation [8] and type inference [16] in functional programming as well as for interactive computing [6,25].

Coinduction is the dual of induction. Induction corresponds to well-founded structures that start from a basis which serve as the foundation for building more complex structures. For example, natural numbers are inductively defined via the base element zero and the successor function. Inductive definitions have 3 components: initiality, iteration and minimality [6]. Thus, the inductive definition of list of numbers is as follows: (i) `[ ]` (empty list) is a list (initiality); (ii) `[H|T]` is as a list if `T` is a list and `H` is some number (iteration); and, (iii) nothing else is a list (minimality). Minimality implies that infinite-length lists of numbers are not members of the inductively defined set of lists of numbers. Inductive definitions correspond to least fixed point interpretations of recursive definitions.

Coinduction eliminates the initiality condition and replaces the minimality condition with maximality. Thus, the coinductive definition of a list of numbers is: (i) `[H|T]` is as a list if `T` is a list and `H` is some number (iteration); and, (ii) the set of lists is the maximal set of such lists. There is no base case in coinductive definitions, and while this may appear circular, the definition is well formed since coinduction corresponds to the greatest fixed point interpretation of recursive definitions (recursive definitions for which gfp interpretation is intended

V. Dahl and I. Niemelä (Eds.): ICLP 2007, LNCS 4670, pp. 27–44, 2007.

are termed corecursive definitions). Thus, the set of lists under coinduction is the set of all infinite lists of numbers (no finite lists are contained in this set). Note, however, that if we have a recursive definition with a base case, then under coinductive interpretation, the set defined will contain both finite and infinite-sized elements, since in this case the gfp will also contain the lfp. In the context of logic programming, in the presence of coinduction, proofs may be of infinite length. A coinductive proof essentially is an infinite-length proof.

2 Coinduction and Logic Programming

Coinduction has been incorporated in logic programming in a systematic way only recently [22,21], where an operational semantics—similar to SLD—is given for computing the greatest fixed point of a logic program. This operational semantics called co-SLD relies on a *coinductive hypothesis rule* and systematically computes elements of the gfp of a program via backtracking. The semantics is limited to only *regular proofs*, i.e., those cases where the infinite behavior is obtained by infinite repetition of a finite number of finite behaviors.

Consider the list example above. The normal logic programming definition of a stream (list) of numbers is given as program P1 below:

```
stream([]).
stream([H|T]) :- number(H), stream(T).
```

Under SLD resolution, the query `?- stream(X)` will systematically produce all finite streams one by one starting from the `[]` stream. Suppose now we remove the base case and obtain the program P2:

```
stream([H|T]) :- number(H), stream(T).
```

In the program P2, the meaning of the query `?- stream(X)` is semantically null under standard logic programming. The problems are two-fold. The Herbrand universe does not allow for infinite terms such as `X` and the least Herbrand model does not allow for infinite proofs, such as the proof of `stream(X)` in program P2; yet these concepts are commonplace in computer science, and a sound mathematical foundation exists for them in the field of hyperset theory [4]. Coinductive LP extends the traditional declarative and operational semantics of LP to allow reasoning over infinite and cyclic structures and properties [22,23,21]. In the coinductive LP paradigm the declarative semantics of the predicate `stream/1` above is given in terms of *infinitary Herbrand (or co-Herbrand) universe, infinitary (or co-Herbrand) Herbrand base [12], and maximal models (computed using greatest fixed-points).*

Thus, under coinductive interpretation of P2, the query `?- stream(X)` produces all infinite sized stream as answers, e.g., `X = [1, 1, 1, ... ]`, `X = [1, 2, 1, 2, ... ]`, etc., thus, P2 is not semantically null (but proofs may be of infinite-length).

If we take a coinductive interpretation of program P1, then we get all finite and infinite stream as answers to the query `?- stream(X)`. Coinductive logic programming allows programmers to manipulate infinite structures. As a result,

unification has to be necessarily extended and "occurs check" removed. Thus, unification equations such as `X = [1 | X]` are allowed in coinductive logic programming; in fact, such equations will be used to represent infinite (regular) structures in a finite manner.

The operational semantics of coinductive logic programming is given in terms of the *coinductive hypothesis rule* which states that during execution, if the current resolvent R contains a call C' that unifies with a call C encountered earlier, then the call C' succeeds; the new resolvent is $R'\theta$ where $\theta = mgu(C, C')$ and R' is obtained by deleting C' from R. With this extension, a clause such as

```
p([1|T]) :- p(T)
```

and the query `?- p(Y)` will produce an infinite answer `Y = [1|Y]`.

Thus, given a call during execution of a logic program, where, earlier, the candidate clauses were tried one by one via backtracking, under coinductive logic programming the trying of candidate clauses is extended with yet more alternatives: applying the coinductive hypothesis rule to check if the current call will unify with any of the earlier calls. The coinductive hypothesis rule will work for only those infinite proofs that are *regular* in nature, i.e., infinite behavior is obtained by a finite number of finite behaviors interleaved infinite number of times (such as a circular linked list). More general implementations of coinduction are possible, but they are beyond the scope of this paper [21].

Even with regular proofs, there are many applications of coinductive logic programming, some of which are discussed next. These include model checking, concurrent logic programming, real-time systems, non-monotonic reasoning, etc. We will not focus on the implementation of coinductive LP (implementation atop YAP is available from the authors) except to note that to implement coinductive LP, one needs to remember in a memo-table (memoize) all the calls made to coinductive predicates.

Finally note that one has to be careful when using both inductive and coinductive predicates together, since careless use can result in interleaving of least fixed point and greatest fixed point computations. Such programs cannot be given meaning easily. Consider the following program where the predicate `p` is coinductive and `q` is inductive.

```
p :- q.
q :- p.
```

For computing the result of goal `?- q.`, we will use lfp semantics, which will produce null, implying that `q` should fail. Given the goal `?- p.` now, it should also fail, since `p` calls `q`. However, if we use gfp semantics (and the coinductive hypothesis computation rule), the goal `p` should succeed, which, in turn, implies that `q` should succeed. Thus, naively mixing coinduction and induction leads to contradictions. This contradiction is resolved by disallowing such cyclical nesting of inductive and coinductive predicates, i.e., *stratifying* inductive and coinductive predicates in a program. An inductive predicate in a given strata cannot call a coinductive predicate in a higher strata and vice versa [23,21].

3 Examples

Next, we illustrate coinductive Logic Programming via more examples.

Infinite Streams: The following example involves a combination of an inductive predicate and a coinductive predicate. By default, predicates are inductive, unless indicated otherwise. Consider the execution of the following program, which defines a predicate that recognizes infinite streams of natural numbers. Note that only the `stream/1` predicate is coinductive, while the `number/1` predicate is inductive.

```
:- coinductive stream/1.
stream([ H | T ]) :- number(H), stream(T).
number(0).
number(s(N)) :- number(N).
| ?- stream([ 0, s(0), s(s(0)) | T ]).
```

The following is an execution trace, for the above query, of the memoization of calls by the operational semantics. Note that calls of `number/1` are not memo'ed because `number/1` is inductive.

```
MEMO: stream([ 0, s(0), s(s(0)) | T ])
MEMO: stream([ s(0), s(s(0)) | T ])
MEMO: stream([ s(s(0)) | T ])
```

The next goal call is `stream(T)`, which unifies with the first memo'ed ancestor, and therefore immediately succeeds. Hence the original query succeeds with the infinite solution:

```
T = [ 0, s(0), s(s(0)) | T ]
```

The user could force a failure here, which would cause the goal to be unified with the next two matching memo'ed ancestor producing `T = [s(0),s(s(0))|T]` and `T = [s(s(0))|T]` respectively. If no remaining memo'ed elements exist, the goal is memo'ed, and expanded using the coinductively defined clauses, and the process repeats—generating additional results, and effectively enumerating the set of (rational) infinite lists of natural numbers that begin with the prefix `[0,s(0),s(s(0))]`.

The goal `stream(T)` is true whenever `T` is some infinite list of natural numbers. If `number/1` was also coinductive, then `stream(T)` would be true whenever `T` is a list containing either natural numbers or ω, i.e., infinity, which is represented as an infinite application of successor `s(s(s(...)))`. Such a term has a finite representation as `X = s(X)`.

Note that excluding the occurs check is necessary as such structures have a greatest fixed-point interpretation and are in the co-Herbrand Universe. This is in fact one of the benefits of coinductive LP. Unification without occurs check is typically more efficient than unification with occurs check, and now it is even possible to define non-trivial predicates on the infinite terms that result from such unification, which are not definable in LP with rational trees. Traditional logic programming's least Herbrand model semantics requires SLD resolution to

unify with occurs check (or lack soundness), which adversely affects performance in the common case. Coinductive LP, on the other hand, has a declarative semantics that allows unification without doing occurs check, and it also allows for non-trivial predicates to be defined on infinite terms resulting from such unification.

List Membership: This example illustrates that some predicates are naturally defined inductively, while other predicates are naturally defined coinductively. The `member/2` predicate is an example of an inherently inductive predicate.

```
member(H, [ H | _ ]).
member(H, [ _ | T ]) :- member(H, T).
```

If this predicate was declared to be coinductive, then `member( X, L)` is true whenever `X` is in `L` or whenever `L` is an infinite list, even if `X` is not in `L`! The definition above, whether declared coinductive or not, states that the desired element is the last element of some prefix of the list, as the following equivalent reformulation of `member/2`, called `membera/2` demonstrates, where `drop/3` drops a prefix ending in the desired element and returns the resulting suffix.

```
membera(X, L) :- drop(X, L, _).
drop(H, [ H | T ], T).
drop(H, [ _ | T ], T1) :- drop(H, T, T1).
```

When the predicate is inductive, this prefix must be finite, but when the predicate is declared coinductive, the prefix may be infinite. Since an infinite list has no last element, it is trivially true that the last element unifies with any other term. This explains why the above definition, when declared to be coinductive, is always true for infinite lists regardless of the presence of the desired element.

A mixture of inductive and coinductive predicates can be used to define a variation of `member/2`, called `comember/2`, which is true if and only if the desired element occurs an infinite number of times in the list. Hence it is false when the element does not occur in the list or when the element only occurs a finite number of times in the list. On the other hand, if `comember/2` was declared inductive, then it would always be false. Hence coinduction is a necessary extension.

```
:- coinductive comember/2.
comember(X, L) :- drop(X, L, L1), comember(X, L1).
?- X = [ 1, 2, 3 | X ], comember(2, X).
        Answer: yes.
?- X = [ 1, 2, 3, 1, 2, 3 ], comember(2, X).
        Answer: no.
?- X = [ 1, 2, 3 | X ], comember(Y, X).
        Answer: Y = 1;
                Y = 2;
                Y = 3;
```

Note that `drop/3` will have to be evaluated using OLDT tabling for it not to go into an infinite loop for inputs such as `X = [1,2,3|X]` (if `X` is absent from the list `L`, the lfp of `drop(X,L)` is null).

List Append: Let us now consider the definition of standard `append` predicate.

```
append([], X, X).
append([H|T], Y, [H|Z]) :- append(T, Y, Z).
```

Not only can the above definition append two finite input lists, as well as split a finite list into two lists in the reverse direction, it can also append infinite lists under coinductive execution. It can even split an infinite list into two lists that when appended, equal the original infinite list. For example:

```
| ?- Y = [4, 5, 6, | Y], append([1, 2, 3], Y, Z).
       Answer: Z = [1, 2, 3 | Y], Y = [4, 5, 6, | Y]
```

More generally, the coinductive append has interesting algebraic properties. When the first argument is infinite, it doesn't matter what the value of the second argument is, as the third argument is always equal to the first. However, when the second argument is infinite, the value of the third argument still depends on the value of the first. This is illustrated below:

```
| ?- X = [1, 2, 3, | X], Y = [3, 4 | Y], append(X, Y, Z).
       Answer: Z = [1, 2, 3 | Z], X = [1,2,3|X], Y = [3,4|Y]
```

The coinductive append can also be used to split infinite lists as in:

```
| ?- Z = [1, 2 | Z], append(X, Y, Z).
       Answers: X = [], Y = [1, 2 | Z], Z = [1, 2 | Z];
                X = [1], Y = [2 | Z], Z = [1, 2 | Z];
                X = [1, 2], Y = Z, Z = [1, 2 | Z];
                X = [1, 2 | X], Y = _, Z = [1, 2 | Z];
                X = [1, 2, 1], Y = [2 | Z], Z = [1, 2 | Z];
                X = [1, 2, 1, 2], Y = Z, Z = [1, 2 | Z];
                X = [1, 2, 1, 2 | X], Y = _, Z = [1, 2 | Z];
                ......
```

Note that application of the coinductive hypothesis rule will produce solutions in which `X` gets bound to an infinite list (fourth and seventh solutions above).

Sieve of Eratosthenes: Coinductive LP also allows for lazy evaluation to be elegantly incorporated into Prolog. Lazy evaluation allows for manipulation of, and reasoning about, cyclic and infinite data structures and properties. Lazy evaluation can be put to fruitful use, in situations where only a finite part of the infinite term is of interest. In the presence of coinductive LP, if the infinite terms involved are rational, then given the goal `p(X), q(X)` with coinductive predicates `p/1` and `q/1`, then `p(X)` can coinductively succeed and terminate, and then pass the resulting `X` to `q(X)`. If `X` is bound to an infinite irrational term during the computation, then `p` and `q` must be executed in a coroutined manner to produce answers. That is, one of the goals must be declared the producer of `X` and the other the consumer of `X`, and the consumer goal must not be allowed to bind `X`. Consider the (coinductive) lazy logic program for the sieve of Eratosthenes:

```
:- coinductive sieve/2, filter/3, comember/2.
primes(X) :- generate_infinite_list(I), sieve(I,L), comember(X,L).
```

```
sieve([H|T], [H|R]) :- filter(H,T,F), sieve(F,R).
filter(H,[],[]).
filter(H,[K|T],[K|T1]) :- R is K mod H, R > 0, filter(H,T,T1).
filter(H,[K|T],T1) :- 0 is K mod H, filter(H,T,T1).
```

In the above program `filter/3` removes all multiples of the first element in the list, and then passes the filtered list recursively to `sieve/2`. If the call `generate_infinite_list(I)` binds `I` to an inductive or rational list (e.g., `X = [2, ..., 20]` or `X = [2, .., 20 | X]`), then filter can be `completely` processed in each call to `sieve/2`. However, in contrast, if `I` is bound to an irrational infinite list as in:

```
:- coinductive int/2.
int(X, [X|Y]) :- X1 is X+1, int(X1, Y).
generate_infinite_list(I) :- int(2,I).
```

then in `primes/1` predicate, the calls `generate_infinite_list/1`, `comember/2`, and `sieve/2` should be co-routined, and likewise, in the `sieve/2` predicate, the calls `filter/3` and the recursive call `sieve/2` must be coroutined.

4 Application to Model Checking and Verification

Model checking is a popular technique used for verifying hardware and software systems. It works by constructing a model of the system in terms of a finite state Kripke structure and then determining if the model satisfies various properties specified as temporal logic formulae. The verification is performed by means of systematically searching the state space of the Kripke structure for a counter-example that falsifies the given property. The vast majority of properties that are to be verified can be classified into *safety* properties and *liveness* properties. Intuitively, safety properties are those which assert that 'nothing bad will happen' while liveness properties are those that assert that 'something good will eventually happen.'

An important application of coinductive LP is in directly representing and verifying properties of Kripke structures and ω-automata (automata that accept infinite strings). Just as automata that accept finite strings can be directly programmed using standard LP, automata that accept infinite strings can be directly represented using coinductive LP (one merely has to drop the base case).

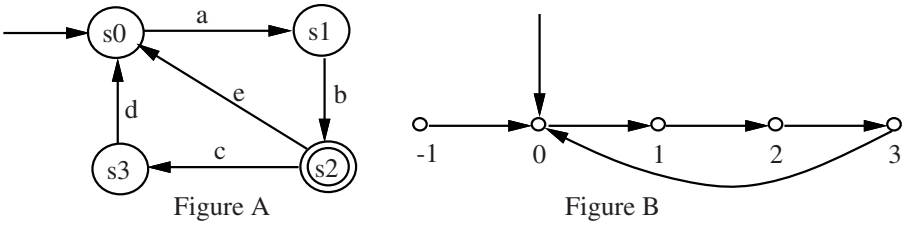


Figure A Figure B

Fig. 1. Example Automata

Consider the automata (over finite strings) shown in Figure 1.A which is represented by the logic program below.

```
automata([X|T], St) :- trans(St, X, NewSt), automata(T, NewSt).
automata([], St) :- final(St).
trans(s0, a, s1).                          trans(s1, b, s2).
trans(s2, c, s3).                          trans(s3, d, s0).
trans(s2, e, s0).                          final(s2).
```

A call to `?- automata(X, s0)` in a standard LP system will generate all finite strings accepted by this automata. Now suppose we want to turn this automata into an ω-automata, i.e., it accepts infinite strings (an infinite string is accepted if states designated as final state are traversed infinite number of times), then the (coinductive) logic program that simulates this automata can be obtained by simply dropping the base case (for the moment, we'll ignore the requirement that final-designated states occur infinitely often; this can be easily checked by `comember/2`).

```
automata([X|T], St) :- trans(St, X, NewSt), automata(T, NewSt).
```

Under coinductive semantics, posing the query `| ?- automata(X, s0).` will yield the solutions:

```
X = [a, b, c, d | X];
X = [a, b, e | X];
```

This feature of coinductive LP can be leveraged to directly and elegantly verify liveness properties in model checking, multi-valued model checking, for modeling and verifying properties of timed ω-automata, checking for bisimilarity, etc.

4.1 Verifying Liveness Properties

It is well known that safety properties can be verified by reachability analysis, i.e, if a counter-example to the property exists, it can be finitely determined by enumerating all the reachable states of the Kripke structure. Verification of safety properties amounts to computing least fixed-points and thus is elegantly handled by standard LP systems extended with tabling [18]. Verification of liveness properties under such tabled LP systems is however problematic. This is because counterexamples to liveness properties take the form of infinite traces, which are semantically expressed as greatest fixed-points. Tabled LP systems [18] work around this problem by transforming the temporal formula denoting the property into a semantically equivalent least fixed-point formula, which can then be executed as a tabled logic program. This transformation is quite complex as it uses a sequence of nested negations.

In contrast, coinductive LP can be directly used to verify liveness properties. Coinductive LP can directly compute counterexamples using greatest fixed-point temporal formulae without requiring any transformation. Intuitively, a state is not live if it can be reached via an infinite loop (cycle). Liveness counterexamples can be found by (coinductively) enumerating all possible states that can be reached via infinite loops and then by determining if any of these states constitutes a valid counterexample. Consider the example of a modulo 4 counter,

adapted from [20] (See Figure 1.B). For correct operation of the counter, we must verify that along every path the state s_{-1} is not reached, i.e., there is at least one infinite trace of the system along which s_{-1} never occurs. This property is naturally specified as a greatest fixed-point formula and can be verified coinductively. A simple coinductive logic program S_P to solve the problem is shown below. We compose the counter program with the negation of the property, i.e., `N1 >= 0`. Note that `sm1` represents the state corresponding to `-1`.

```
:- coinductive s0/2, s1/2, s2/2, s3/2, sm1/2.
sm1(N,[sm1|T]) :- N1 is N+1 mod 4, s0(N1,T), N1>=0.
s0(N,[s0|T]) :- N1 is N+1 mod 4, s1(N1,T), N1>=0.
s1(N,[s1|T]) :- N1 is N+1 mod 4, s2(N1,T), N1>=0.
s2(N,[s2|T]) :- N1 is N+1 mod 4, s3(N1,T), N1>=0.
s3(N,[s3|T]) :- N1 is N+1 mod 4, s0(N1,T), N1>=0.
```

The counter is coded as a cyclic program that loops back to state s_0 via states s_1, s_2 and s_3. State s_{-1} represents a state where the counter has the value -1. The property P to be verified is whether the state s_{-1} is live. The query `:- sm1(-1,X), comember(sm1,X)` where the `comember` predicate coinductively checks that `sm1` occurs in `X` infinitely often, will fail implying inclusion of the property in the model, i.e., the absence of a counterexample to the property. The benefit of our approach is that we do not have to transform the model into a form amenable to safety checking. This transformation is expensive in general and can reportedly increase the time and memory requirements by 6-folds [20].

This direct approach to verifying liveness properties also applies to *multi-valued model checking* of the μ-calculus [13]. Multi-valued model checking is used to model systems, whose specification has varying degrees of inconsistency or incompleteness. Earlier effort [13] verified liveness properties by computing the *gfp* which was found using negation based transformation described earlier. With coinduction, the *gfp* can be computed directly as in standard model checking as described above. We do not give details due to lack of space. Coinductive LP can also be used to check for *bisimilarity*. Bisimilarity is reduced to coinductively checking if two ω-automata accept the same set of rational infinite strings.

4.2 Verifying Properties of Timed Automata

Timed automata are simple extensions of ω-automata with stopwatches [1], and are easily modeled as coinductive logic programs with CLP(R) [9]. Timed automata can be modeled with coinductive logic programs together with constraints over reals for modeling clock constraints. The coinductive logic program with CLP(R) constraints for modeling the classic train-gate-controller problem is shown below. This program runs on our implementation of coinduction on YAP [19] extended with CLP(R). The system can be queried to enumerate all the infinite strings that will be accepted by the automata and that meet the time constraints. Safety and liveness properties can be checked by negating those properties, and checking that they fail for each string accepted by the automata with the help of `comember/2` predicate.

The code for the timed automata represented in Fig 2 is given below. The predicate `driver/9`, that composes the 3 automata, is coinductive, as it executes forever.

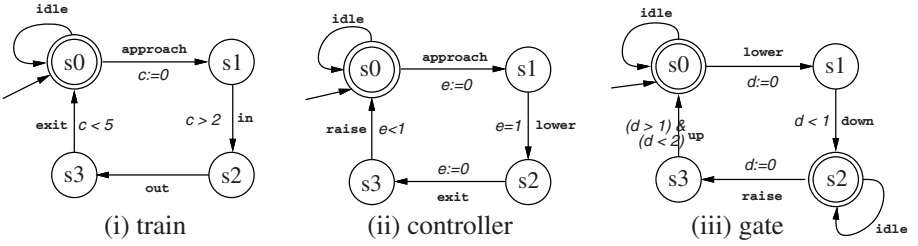


Fig. 2. Train-Controller-Gate Timed Automata

```
:- use_module(library(clpr)).
:- coinductive driver/9.

train(X,up,X,T1,T2,T2).
train(s0,approach,s1,T1,T2,T3) :-
        {T3 = T1}.
train(s1,in,s2,T1,T2,T3) :-
        {T1 - T2 > 2,
        T3 = T2}.
train(s2,out,s3,T1,T2,T2).
train(s3,exit,s0,T1,T2,T3) :-
        {T3 = T2,
        T1 - T2 < 5}.
train(X,lower,X,T1,T2,T2).
train(X,down,X,T1,T2,T2).
train(X,raise,X,T1,T2,T2).

gate(s0,lower,s1,T1,T2,T3) :-
                {T3 = T1}.
gate(s1,down,s2,T1,T2,T3) :-
                {T3 = T2,
                 T1 - T2 < 1}.
gate(s2,raise,s3,T1,T2,T3) :-
                {T3 = T1}.
gate(s3,up,s0,T1,T2,T3) :-
       {T3 = T2, T1 - T2 > 1,
                T1 - T2 < 2}.
gate(X,approach,X,T1,T2,T2).
gate(X,in,X,T1,T2,T2).
gate(X,out,X,T1,T2,T2).
gate(X,exit,X,T1,T2,T2).

contr(s0,approach,s1,T1,T2,T1).
contr(s1,lower,s2,T1,T2,T3) :- {T3 = T2, T1 - T2 = 1}.
contr(s2,exit,s3,T1,T2,T1).
contr(s3,raise,s0,T1,T2,T2) :- {T1-T2 < 1}.
contr(X,in,X,T1,T2,T2).
contr(X,up,X,T1,T2,T2).
contr(X,out,X,T1,T2,T2).
contr(X,down,X,T1,T2,T2).

driver(S0,S1,S2,T,T0,T1,T2,[X|Rest],[(X,T)|R]) :-
        train(S0,X,S00,T,T0,T00),
        contr(S1,X,S10,T,T1,T10) ,
        gate(S2,X,S20,T,T2,T20),
        {TA > T},
        driver(S00,S10,S20,TA,T00,T10,T20,Rest,R).
```

Given the query:

```
| ?- driver(s0, s0, s0, T, Ta, Tb, Tc, X, R).
```

We obtain the following infinite lists as answers (`A, B, C, ..`, etc. are the time on the wall clock when the corresponding event occurs).

```
R = [(approach,A),(lower,B),(down,C),(in,D),(out,E),
        (exit,F),(raise,G),(up,H)|R],
X = [approach,lower,down,in,out,exit,raise,up | X] ? ;

R= [(approach,A),(lower,B),(down,C),(in,D),(out,E),
        (exit,F),(raise,G),(approach,H),(up,I)|R],
X = [approach,lower,down,in,out,exit,raise,approach,up|X] ? ;

no
```

A call to coinductively defined sublist/2 predicate (not shown here) can then be used to check the safety property that the signal `down` occurs before `in` by checking that the infinite list `Y = [down, in | Y]` is coinductively contained in the infinite string `X` above. This ensure that the system satisfies the safety property, namely, that the gate is down before the train is in the gate area. A similar approach can be used to verify the liveness property, namely that the gate will eventually go up [9], by finding the maximum difference between the times the gate goes down and later comes up.

Note that from the answers above, one can see that another train can approach before the gate goes up, however, this behavior is entirely consistent as the gate will go up and will come down again, by the time the second train arrives in the gate area (see Figure 2). We can find out the minimum time that must intervene between two trains for the system to remain safe by finding he minimum value the time that elapses between two approach signals given the above constraints (answer is computed to be 7 units of time).

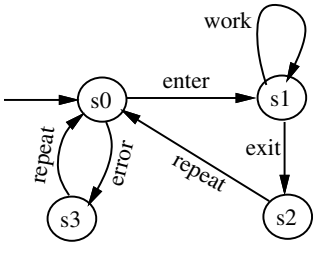
4.3 Verification of Nested Finite and Infinite Automata

We next illustrate application of coinductive logic programming to verification in which infinite (coinductive) and finite (inductive) automata are nested. It is well known that reachability-based inductive techniques are not suitable for verifying liveness properties [17]. Further, it is also well known that, in general, verification of liveness properties can be reduced to verification of termination under the assumption of fairness [24] and that fairness properties can be specified in terms of alternating fixed-point temporal logic formulas [11]. Earlier we showed that co-inductive LP allows one to verify a class of liveness properties in the absence of fairness constraints. Coinductive LP further permits us to verify a more general class of all liveness properties that can only be verified in the presence of fairness constraints.

Essentially, a coinductive LP based approach demonstrates that if a model satisfies the fairness constraint then, it also satisfies the liveness property. This is

achieved by composing a program P_M, which encodes the model, with a program P_F, which encodes the fairness constraint and a program P_{NP}, which encodes the negation of the liveness property, to obtain a composite program P_μ. We then compute the stratified alternating fixed-point of the logic program P_μ and check for the presence of the initial state of the model in the stratified alternating fixed-point. If the alternating fixed-point contains the initial state, then that implies the presence of a valid counterexample that violates the given liveness property. On the other hand, if the alternating fixed-point is empty, then that implies that no counterexample can be constructed, which in turn implies that the model satisfies the given liveness property.

We will now illustrate our approach using a very simple example (which can be programmed on our coinductive LP implementation). Consider the model shown in Figure 3, consisting of four states. The system starts off in state `s0`, enters state `s1`, performs a finite amount of work in state `s1` and then exits to state `s2`, from where it transitions back to state `s0`, and repeats the entire loop again, an infinite number of times. The system might encounter an error, causing a transition to state `s3`; corrective action is taken, followed by a transition back to `s0` (this can also happen infinitely often). The system is modeled by the Prolog code shown in Figure 3.




```
:- coinductive state/2.
state(s0,[s0,is1|T]):-enter, work,
                       state(s1,T).
state(s1,[s1|T]):-exit, state(s2,T).
state(s2,[s2|T]):-repeat, state(s0,T).
state(s0,[s0|T]):-error, state(s3,T).
state(s3,[s3|T]):-repeat, state(s0,T).
work :- work.
work.
enter.                repeat.
exit.                 error.
```

Fig. 3. Nested Automata

This simple example illustrates the power of co-logic programming (co-LP, for brevity, that contains both inductive and coinductive LP) when compared to purely inductive or purely coinductive LP. Note that the computation represented by the state machine in the example consists of two stratified loops, represented by recursive predicates. The outer loop (predicate `state/2`) is coinductive and represents an infinite computation (hence it is declared as coinductive as we are interested in its gfp). The inner loop (predicate `work/0`) is inductive and represents a bounded computation (we are interested in its lfp). The semantics therefore evaluates `work/0` using SLD resolution and `state/2` using co-SLD resolution.

The property that we would like to verify is that the computation in the state `s1` represented by `work` always terminates. In order to do so, we require the fairness property: "if the transition `enter` occurs infinitely often, then the transition `exit` also occurs infinitely often". The stratified alternating fixed-point semantics ensures that this fairness constraint holds by computing the

minimal model of the inductive program represented by the predicate `state/1` and then composing it with the coinductive program. The resulting program is then composed with the property, "the state `s2` is not present in any trace of the infinite computation," which is the negation of the given liveness property. The negated property is represented by the predicate `absent/2`. Thus, given the program above, the user will pose the query:

```
| ?- state(s0,X), absent(s2,X).
```

where `absent/2` is a coinductive predicate that checks that the state `s2` is not present in the (infinite) list `X` infinitely often (it is the negated version of the coinductive comember predicate described earlier). The co-LP system will respond with a solution: `X = [s0, s3 | X]`, a counterexample which states that there is an infinite path not containing `s2`. One can see that this corresponds to the (infinite) behavior of the system if `error` is encountered.

5 Applications to Non-monotonic Reasoning

We next consider application of coinductive LP to non-monotonic reasoning, in particular its manifestation as *answer set programming* (ASP) [5,14]. ASP has been proposed as an elegant way of introducing non-monotonic reasoning into logic programming [3]. ASP has been steadily gaining popularity since its inception due to its applications to planning, action-description, AI, etc.

We believe that if one were to add negation as failure to coinductive LP then one would obtain something resembling answer set programming. To support negation as failure in coinductive LP, we have to extend the coinductive hypothesis rule: given the goal `not(G)`, if we encounter `not(G')` during the proof, and `G` and `G'` are unifiable, then `not(G)` succeeds. Since the coinductive hypothesis rule provides a method for goal-directed execution of coinductive logic programs, it can also be used for goal-directed execution of answer set programs. As discussed earlier, coinduction is a technique for specifying unfounded set; likewise, answer set programs are also recursive specifications (containing negation as failure) for computing unfounded sets. Obviously, coinductive LP and ASP must be related.

All approaches to implementing ASP are based on bottom up execution of finitely grounded programs. If a top-down execution scheme can be designed for ASP, then ASP can be extended to include predicates over general terms. We outline how this can be achieved using our top-down implementation of coinduction. In fact, a top-down interpreter for ASP (restricted to propositions at present) can be trivially realized on top of our implementation of coinductive LP. Work is in progress to implement a top-down interpreter for ASP with general predicates [15].

5.1 A Top-Down Algorithm for Computing Answer Sets

In top-down execution of answer set programs, given a (propositional) query goal `Q`, we are interested in finding out all the answer sets that contain `Q`, one by one, via backtracking. If `Q` is not in any answer set or if there are no answer sets at all, then the query should fail. In the former case, the query `not(Q)` should

succeed and should enumerate all answer sets which do not contain `Q`, one by one, via backtracking.

The top down execution algorithm is quite simply realized with the help of coinduction. The traditional Gelfond-Lifschitz (GL) method [3] starts with a candidate answer set, computes a residual program via the GL-transformation, and then finds the lfp of the residual program. The candidate answer set is an answer set, if it equals the lfp of the residual program. Intuitively, in our top down execution algorithm, the propositions in the candidate answer set are regarded as hypotheses which are treated as facts during top-down coinductive execution. A call is said to be a *positive* call if it is in the scope of even number of negations, similarly, a call is said to be a *negative* call if it is in the scope of odd number of negations.

The top down query processing algorithm works as follows: suppose the current call (say, `p`) is a positive call, then it will be placed in a *positive coinductive hypothesis set* (PCHS), a matching rule will be found and the Prolog-style expansion done. The new resolvent will be processed left to right, except that every positive call will be continued to be placed in the positive hypothesis set, while a negative call will be placed in the *negative coinductive hypothesis set* (NCHS). If a positive call `p` is encountered again, then if `p` is in PCHS, the call immediately succeeds; if it is in NCHS, then there is an inconsistency and backtracking takes place. If a negative call (say, `not(p)`) is encountered for the first time, `p` will be placed in the NCHS. If a negative proposition `not(p)` is encountered later, then if `p` is in NCHS, `not(p)` succeeds; if `p` is in PCHS, then there is an inconsistency and backtracking takes place. Once the execution is over with success, (part of) the *potential* answer set can be found in the PCHS. The set NCHS contains propositions that are *not* in the answer set.

Essentially, the algorithm explicitly keeps track of propositions that are in the answer set (PCHS) and those that are not in the answer set (NCHS). Any time, a situation is encountered in which a proposition is both in the answer set and not in the answer set, an inconsistency is declared and backtracking ensues.

We still need one more step. ASP can specify the falsification of a goal via constraints. For example, the constraint `p :- q, not p.` restricts `q` (and `p`) to not be in the answer set (unless `p` happens to be in the answer via other rules). For such rules of the form

```
p :- B.
```

if `not(p)` is reachable via goals in the body `B`, we need to explicitly ensure that the potential answer set does not contain a proposition that is falsified by this rule.

Given an answer set program, a rule `p :- B.` is said to be non-constraint rule (NC-rule) if `p` is reachable through calls in the body `B` through an even number of negation as failure calls, otherwise it is said to be a constraint rule (C-rule). Thus, given the ASP program:

```
p :- a, not q.          .......(i)
q :- b, not r.          .......(ii)
r :- c, not p.          .......(iii)
q :- d, not p.          .......(iv)
```

rules (i), (ii) and (iii) are C-rules, while (i) and (iv) are NC-rules. A rule can be both an NC-rule as well as a C-rule (such as rule (i)). NC-rules will be used to compute the potential answers sets, while C-rules will only be used to reject or accept potential answer sets. Rejection or acceptance of a potential answer set is accomplished as follows: For each C-rule of the form r_i `:- B`, where `B` directly or indirectly leads to `not(`r_i`)`, we construct a new rule:

`chk_`r_i `:- not(`r_i`), B.`

Next, we construct a new rule:

`nmr_check :- not(chk_`r_1`), not(chk_`r_2`), ..., not(chk_`r_i`), ...`

Now, the top level query, `?-Q`, is transformed into: `?- Q, nmr_check. Q` will be executed using only the NC-rules to generate potential answer sets, which will be subsequently either rejected or accepted by the call to `nmr_check`. If `nmr_check` succeeds, the potential answer set is an answer set. If `nmr_check` fails, the potential answer set is rejected and backtracking occurs.

For simplicity of illustration, we assume that for each NC-rule, we construct its negated version which will be expanded when a corresponding negative call is encountered (in the implementation, however, this is done implicitly). Thus, given an NC-rule for a proposition `p` of the form:

`p :-` B_1`.`
`p :-` B_2`.`
`...`
`p :-` B_i`.`
`...`

its negated version will be:

`not_p :- not(`B_1`), not(`B_2`), ..., not(`B_i`), ...`

If a call to `not(p)` is encountered, then this negated `not_p` rule will be used to expand it.

Note, finally, that the answer sets reported may be partial, because an answer set may be a union of multiple independent answer subsets, and the other subsets may not be deducible from the query goal due to the nature of the rules. The top-down algorithm for computing the (partial) answer set of an ASP program can be summarized as follows.

1. Initialize PCHS and NCHS to empty (these are maintained as global variables in our implementation of top-down ASP atop our coinductive YAP implementation). Declare every proposition in the ASP as a coinductive proposition.
2. Identify the set of C-rules and NC-rules in the program.
3. Assert `chk_`r_i rule for every C-rule with r_i as head and build the `nmr_check` rule; append the call `nmr_check` to the initial query.
4. For each NC-rule, construct its negated version.
5. For every positive call `p`: if `p` $\in$ PCHS, then `p` succeeds coinductively and the next call in the resolvent is executed, else if `p` $\in$ NCHS, then there is an inconsistency and backtracking ensues, else (`p` is not in PCHS or in NCHS) add `p` to PCHS and expand `p` using NC-rules that have `p` in the head (create a choice-point if there are multiple matching rules).

6. For every negative call of the form `not(p)`: if `p` ∈ NCHS, then `not(p)` succeeds coinductively and the next call in the resolvent is executed, else if `p` ∈ PCHS, then there is an inconsistency and backtracking ensues, else (`p` is not in PCHS or in NCHS) add `p` to NCHS and expand `not(p)` using the negated `not_p` rule for `p`.

Next, let's consider an example. Consider the following program:

```
p :- not(q).
q :- not(r).
r :- not(p).
q :- not(p).
```

After, step 1-4, we obtain the following program.

```
:- coinductive p/0, q/0, r/0.

p :- not(q).
q :- not(r).
r :- not(p).
q :- not(p).

chk_p :- not(p), not(q).
chk_q :- not(q), not(r).
chk_r :- not(r), not(p).

not_p :- q.
not_q :- r,p.
not_r :- p.

nmr_chk :- not(chk_p), not(chk_q), not(chk_r).
```

The ASP program above has {`q`,`r`} as the only answer set. Given the transformed program, the query: `?- q` will produce {`q`} as the answer set (with `p` known to be not in the answer set). The query `?- r` will produce the answer set {`q`,`r`} (with `p` known to be not in the answer set). It is easy to see why the first query `?- q` will not deduce `r` to be in the answer set: there is nothing in the NC-rules that relates `q` and `r`.

5.2 Correctness of the Top-Down Algorithm

We next outline a proof of correctness of the top-down method. First, note that the above top down method corresponds to computing the greatest fix point of the residual program rather than the least fix point. Second, we argue that the Gelfond-Lifschitz method for checking if a given set is an answer set [5] should compute the greatest fix point of the residual program instead of the least fixed point. A little thought will reflect that circular reasoning entailed by rules such as

```
p :- p.
```

is present in ASP through rules of the form:

```
p :- not(q).
q :- not(p).
```

If we extend the GL method, so that instead of computing the lfp, we compute the gfp of the residual program, then the GL transformation can be modified to remove positive goals as well. Given an answer set program, whose answer set is guessed to be `A`, the modified GL transform then becomes as follows:

1. Remove all those rules whose body contains `not(p)`, where `p` $\in$ `A`.
2. Remove all those rules whose body contains `p`, where `p` $\notin$ `A`.
3. From body of each rule, remove all goals of the form `not(p)`, where `p` $\notin$ `A`.
4. From body of each rule, remove all positive goals `p` such that `p` $\in$ `A`.

After application of this transform, the residual program is a set of facts. If this set of facts is the same as `A`, then `A` is an answer set. It is easy to see that our top-down method mimics the modified GL-transformation (note, however, that our top-down algorithm can be easily modified to work with the original GL method which computes the lfp of the residual program: we merely have to disallow making inference from positive coinductive loops of the form `p :- p.`).

In the top down algorithm, whenever a positive (resp. negative) predicate is called, it is stored in the positive (resp. negative) coinductive hypothesis set. This is equivalent to removing the positive (resp. negative) predicate from the body of all the rules, since a second call to predicate will trivially succeed by the principle of coinduction. Given a predicate `p` (resp. `not(p)`) where `p` is in the positive (resp. negative) coinductive hypothesis set, if a call is made to `not(p)` (resp. `p`), then the call fails. This amounts to 'removing the rule' in GL transform [2].

The top down method described above can also be extended for deducing answers sets of programs containing first order predicates [15].

6 Conclusions

In this paper we gave an introduction to coinductiive logic programming. Practical applications of coinductive LP to verification and model checking and to non-monotonic reasoning were also discussed. Coinductive reasoning can also be included in first order theorem proving in a manner similar to coinductive LP [15].

Acknowledgments. We are grateful to Vítor Santos Costa and Ricardo Rocha.

References

1. Alur, R., Dill, D.L.: A theory of timed automata. TCS 126, 183–235 (1994)
2. Bansal, A.: Towards next generation logic programming systems. Ph.D. thesis forthcoming
3. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press, Cambridge (2003)

4. Barwise, J., Moss, L.: Vicious Circles: On the Mathematics of Non-Wellfounded Phenomena. CSLI Publications, Stanford (1996)
5. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: Kowalski, R., Bowen, K. (eds.) Logic Programming: Proc. of the Fifth International Conference and Symposium, pp. 1070–1080 (1988)
6. Goldin, D., Keil, D.: Interaction, Evolution and Intelligence. In: Proc. Congress on Evolutionary Computing (2001)
7. Goguen, J., Lin, K.: Behavioral Verification of Distributed Concurrent Systems with BOBJ. In: Proc. Conference on Quality Software, pp. 216–235. IEEE Press, Los Alamitos (2003)
8. Gordon, A.: A Tutorial on Co-induction and Functional Programming. In: Proceedings of the 1994 Glasgow Workshop on Functional Programming, Springer Workshops in Computing (Functional Programming). pp. 78–95 (1995)
9. Gupta, G., Pontelli, E.: Constraint-based Specification and Verification of Real-time Systems. In: Proc. IEEE Real-time Symposium '97, pp. 230–239
10. Jacobs, B.: Introduction to Coalgebra: Towards Mathematics of States and Observation. Draft manuscript
11. Liu, X., Ramakrishnan, C.R., Smolka, S.A.: Fully local and efficient evaluation of alternating fixed-points. In: Steffen, B. (ed.) ETAPS 1998 and TACAS 1998. LNCS, vol. 1384, Springer, Heidelberg (1998)
12. Lloyd, J.W.: Foundations of Logic Programming, 2nd edn. Springer, Heidelberg (1987)
13. Mallya, A.: Multivalued Deductive Multi-valued Model Checking. Ph.d. thesis. UT Dallas (2006)
14. Marek, W., Truszczynski, M.: Stable models and an alternative logic programming paradigm. In: the Logic Programming Paradigm: a 25-Year Perspective, pp. 375–398. Springer, Heidelberg (1999)
15. Min, R.: Coinduction in monotonic and non-monotonic reasoning. Ph.D. thesis forthcoming
16. Pierce, B.: Types and Programming Languages. MIT Press, Cambridge, MA (2002)
17. Podelski, A., Rybalchenko, A.: Transition Predicate Abstraction and Fair Termination. In: POPL '05, pp. 132–144. ACM Press, New York (2005)
18. Ramakrishna, Y.S., et al.: Efficient Model Checking Using Tabled Resolution. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 143–154. Springer, Heidelberg (1997)
19. Costa, V.S., Rocha, R.: The YAP Prolog System
20. Schuppan, V., Biere, A.: Liveness Checking as Safety Checking for Infinite State Spaces. ENTCS 149(1), 79–96 (2006)
21. Simon, L.: Extending Logic Programming with Coinduction. Ph.D. Thesis (2006)
22. Simon, L., Mallya, A., Bansal, A., Gupta, G.: Coinductive Logic Programming. In: Etalle, S., Truszczyński, M. (eds.) ICLP 2006. LNCS, vol. 4079, pp. 330–344. Springer, Heidelberg (2006)
23. Simon, L., Bansal, A., Mallya, A., Gupta, G.: Co-Logic Programming. In: Arge, L., Cachin, C., Jurdzinski, T., Tarlecki, A. (eds.) ICALP 2007, LNCS, vol. 4596, Springer, Heidelberg (2007)
24. Vardi, M.: Verification of Concurrent Programs: The Automata-Theoretic Framework. In: LICS '87, pp. 167–176. IEEE, Los Alamitos (1987)
25. Wegner, P., Goldin, D.: Mathematical models of interactive computing. Brown University Technical Report CS 99-13 (1999)

Multi-paradigm Declarative Languages*

Michael Hanus

Institut für Informatik, CAU Kiel, D-24098 Kiel, Germany
mh@informatik.uni-kiel.de

Abstract. Declarative programming languages advocate a programming style expressing the properties of problems and their solutions rather than how to compute individual solutions. Depending on the underlying formalism to express such properties, one can distinguish different classes of declarative languages, like functional, logic, or constraint programming languages. This paper surveys approaches to combine these different classes into a single programming language.

1 Introduction

Compared to traditional imperative languages, declarative programming languages provide a higher and more abstract level of programming that leads to reliable and maintainable programs. This is mainly due to the fact that, in contrast to imperative programming, one does not describe *how* to obtain a solution to a problem by performing a sequence of steps but *what* are the properties of the problem and the expected solutions. In order to define a concrete declarative language, one has to fix a base formalism that has a clear mathematical foundation as well as a reasonable execution model (since we consider a *programming language* rather than a *specification language*). Depending on such formalisms, the following important classes of declarative languages can be distinguished:

- *Functional languages:* They are based on the lambda calculus and term rewriting. Programs consist of functions defined by equations that are used from left to right to evaluate expressions.
- *Logic languages:* They are based on a subset of predicate logic to ensure an effective execution model (linear resolution). Programs consist of predicates defined by definite clauses. The execution model is goal solving based on the resolution principle.
- *Constraint languages:* They are based on constraint structures with specific methods to solve constraints. Programs consist of specifications of constraints of the considered application domain based on a set of primitive constraints and appropriate combinators. Constraint languages are often embedded in other languages where logic programming is a natural candidate. In this case, *constraint logic programming* [61] can be considered as a generalization of logic programming where unification on Herbrand terms is considered as a specific built-in constraint solver.

* This work was partially supported by the German Research Council (DFG) under grant Ha 2457/5-2 and the NSF under grant CCR-0218224.

V. Dahl and I. Niemelä (Eds.): ICLP 2007, LNCS 4670, pp. 45–75, 2007.

The different declarative programming paradigms offer a variety of programming concepts to the programmer. For instance, functional programming emphasizes generic programming using higher-order functions and polymorphic typing, and efficient and (under particular conditions) optimal evaluation strategies using demand-driven evaluation, which contributes to modularity in programming [59]. Logic programming supports the computation with partial information (logic variables) and nondeterministic search for solutions, where constraint programming adds efficient constraint solving capabilities for particular domains. Since all these features have been shown to be useful in application programming and declarative languages are based on common grounds, it is a natural idea to combine these worlds of programming into a single *multi-paradigm declarative language*. However, the interactions between the different features are complex in detail so that the concrete design of a multi-paradigm declarative language is non-trivial. This is demonstrated by many different proposals and a lot of research work on the semantics, operational principles, and implementation of multi-paradigm declarative languages since more than two decades. In the following, we survey some of these proposals.

One can find two basic approaches to amalgamate functional and logic languages: either extend a functional language with logic programming features or extend a logic language with features for functional programming. Since functions can be considered as specific relations, there is a straightforward way to implement the second approach: extend a logic language with syntactic sugar to allow functional notation (e.g., defining equations, nested functional expressions) which is translated by some preprocessor into the logic kernel language. A recent approach of this kind is [25] where functional notation is added to Ciao-Prolog. The language Mercury [83] is based on a logic programming syntax with functional and higher-order extensions. Since Mercury is designed towards a highly efficient implementation, typical logic programming features are restricted. In particular, predicates and functions must have distinct modes so that their arguments are either ground or unbound at call time. This inhibits the application of typical logic programming techniques, like computation with partially instantiated structures, so that some programming techniques developed for functional logic programming languages [11,43,44] can not be applied. This condition has been relaxed in the language HAL [33] which adds constraint solving possibilities. However, Mercury as well as HAL are based on a strict operational semantics that does not support optimal evaluation as in functional programming. This is also true for Oz [82]. The computation model of Oz extends the concurrent constraint programming paradigm [78] with features for distributed programming and stateful computations. It provides functional notation but restricts their use compared to predicates, i.e., function calls are suspended if the arguments are not instantiated in order to reduce them in a deterministic way. Thus, nondeterministic computations must be explicitly represented as disjunctions so that functions used to solve equations require different definitions than functions to rewrite expressions. In some sense, these approaches do not exploit the semantical information provided by the presence of functions.

Extensions of functional languages with logic programming features try to retain the efficient demand-driven computation strategy for purely functional computations and add some additional mechanism for the extended features. For instance, Escher [63] is a functional logic language with a Haskell-like syntax [75] and a demand-driven reduction strategy. Similarly to Oz, function calls are suspended if they are not instantiated enough for deterministic reduction, i.e., nondeterminism must be expressed by explicit disjunctions. The operational semantics is given by a set of reduction rules to evaluate functions in a demand-driven manner and to simplify logical expressions. The languages Curry [41,58] and TOY [66] try to overcome the restrictions on evaluating function calls so that there is no need to implement similar concepts as a function and a predicate, depending on their use. For this purpose, functions can be called, similarly to predicates, with unknown arguments that are instantiated in order to apply a rule. This mechanism, called *narrowing*, amalgamates the functional concept of reduction with unification and nondeterministic search from logic programming. Moreover, if unification on terms is generalized to constraint solving, features of constraint programming are also covered. Based on the narrowing principle, one can define declarative languages integrating the good features of the individual paradigms, in particular, with a sound and complete operational semantics that is optimal for a large class of programs [9].

In the following, we survey important concepts of such multi-paradigm declarative languages. As a concrete example, we consider the language Curry that is based on these principles and intended to provide a common platform for research, teaching, and application of integrated functional logic languages.

Since this paper is a survey of limited size, not all of the numerous papers in this area can be mentioned and relevant topics are only sketched. Interested readers might look into the references for more details. In particular, there exist other surveys on particular topics related to this paper. [38] is a survey on the development and the implementation of various evaluation strategies for functional logic languages that have been explored until a decade ago. [7] contains a good survey on more recent evaluation strategies and classes of functional logic programs. [77] is more specialized but reviews the efforts to integrate constraints into functional logic languages.

The rest of this paper is structured as follows. The next main section introduces and reviews the foundations of functional logic programming that are relevant in current languages. Section 3 discusses practical aspects of multi-paradigm languages. Section 4 contains references to applications of such languages. Finally, Section 5 contains our conclusions.

2 Foundations of Functional Logic Programming

2.1 Basic Concepts

In the following, we use functional programming as our starting point, i.e., we develop functional logic languages by extending functional languages with features for logic programming.

A *functional program* is a set of functions defined by *equations* or *rules*. A *functional computation* consists of replacing subexpressions by equal (w.r.t. the function definitions) subexpressions until no more replacements (or *reductions*) are possible and a value or normal form is obtained. For instance, consider the function `double` defined by[1]

```
double x = x+x
```

The expression "`double 1`" is replaced by `1+1`. The latter can be replaced by `2` if we interpret the operator "`+`" to be defined by an infinite set of equations, e.g., `1+1=2`, `1+2=3`, etc (we will discuss the handling of such functions later). In a similar way, one can evaluate nested expressions (where the subexpression to be replaced is underlined):

`double (1+2)` → `(1+2)+(1+2)` → `3+(1+2)` → `3+3` → `6`

There is also another order of evaluation if we replace the arguments of operators from right to left:

`double (1+2)` → `(1+2)+(1+2)` → `(1+2)+3` → `3+3` → `6`

In this case, both derivations lead to the same result. This indicates a *fundamental property of declarative languages*, also termed *referential transparency*: the value of a computed result does not depend on the order or time of evaluation due to the absence of side effects. This simplifies the reasoning about and maintenance of declarative programs.

Obviously, these are not all possible evaluation orders since one can also evaluate the argument of `double` before applying its defining equation:

`double (1+2)` → `double 3` → `3+3` → `6`

In this case, we obtain the same result with less evaluation steps. This leads to questions about appropriate *evaluation strategies*, where a strategy can be considered as a function that determines, given an expression, the next subexpression to be replaced: which strategies are able to compute values for which classes of programs? As we will see, there are important differences in case of recursive programs. If there are several strategies, which strategies are better w.r.t. the number of evaluation steps, implementation effort, etc? Many works in the area of functional logic programming have been devoted to find appropriate evaluation strategies. A detailed account of the development of such strategies can be found in [38]. In the following, we will survey only the strategies that are relevant for current functional logic languages.

Although functional languages are based on the lambda calculus that is purely based on function definitions and applications, modern functional languages offer more features for convenient programming. In particular, they support the

[1] For concrete examples in this paper, we use the syntax of Curry which is very similar to the syntax of Haskell [75], i.e., (type) variables and function names usually start with lowercase letters and the names of type and data constructors start with an uppercase letter. The application of a function f to an expression e is denoted by juxtaposition ("$f\ e$"). Moreover, binary operators like "`+`" are written infix.

definition of algebraic data types by enumerating their *constructors*. For instance, the type of Boolean values consists of the constructors `True` and `False` that are declared as follows:

```
data Bool = True | False
```

Functions on Booleans can be defined by pattern matching, i.e., by providing several equations for different argument values:

```
not True  = False
not False = True
```

The principle of replacing equals by equals is still valid provided that the actual arguments have the required form, e.g.:

`not (not False)` → `not True` → `False`

More complex data structures can be obtained by recursive data types. For instance, a list of elements, where the type of elements is arbitrary (denoted by the type variable `a`), is either the empty list "`[]`" or the non-empty list "`e:l`" consisting of a first element `e` and a list `l`:

```
data List a = [] | a : List a
```

The type "`List a`" is usually written as `[a]` and finite lists e_1`:`e_2`:...:`e_n`:[]` are written as `[`e_1`,`e_2`,...,`e_n`]`. We can define operations on recursive types by inductive definitions where pattern matching supports the convenient separation of the different cases. For instance, the concatenation operation "`++`" on polymorphic lists can be defined as follows (the optional type declaration in the first line specifies that "`++`" takes two lists as input and produces an output list, where all list elements are of the same unspecified type):

```
(++) :: [a] -> [a] -> [a]
[]     ++ ys = ys
(x:xs) ++ ys = x : xs++ys
```

Beyond its application for various programming tasks, the operation "`++`" is also useful to specify the behavior of other functions on lists. For instance, the behavior of a function `last` that yields the last element of a list can be specified as follows: for all lists l and elements e, `last` $l = e$ iff $\exists xs : xs$ `++[`e`]` $= l$.[2] Based on this specification, one can define a function and verify that this definition satisfies the given specification (e.g., by inductive proofs as shown in [20]). This is one of the situations where functional logic languages become handy. Similarly to logic languages, functional logic languages provide search for solutions for existentially quantified variables. In contrast to pure logic languages, they support equation solving over nested functional expressions so that an equation like xs `++[`e`]` $=$ `[1,2,3]` is solved by instantiating xs to the list `[1,2]` and e to the value `3`. For instance, in Curry one can define the operation `last` as follows:

```
last l | xs++[e] =:= l = e   where xs,e free
```

[2] The exact meaning of the equality symbol is omitted here since it will be discussed later.

Here, the symbol "`=:=`" is used for *equational constraints* in order to provide a syntactic distinction from defining equations. Similarly, *extra variables* (i.e., variables not occurring in the left-hand side of the defining equation) are explicitly declared by "`where...free`" in order to provide some opportunities to detect bugs caused by typos. A *conditional equation* of the form `l | c = r` is applicable for reduction if its condition c has been solved. In contrast to purely functional languages where conditions are only evaluated to a Boolean value, functional logic languages support the *solving* of conditions by guessing values for the unknowns in the condition. As we have seen in the previous example, this reduces the programming effort by reusing existing functions and allows the direct translation of specifications into executable program code. The important question to be answered when designing a functional logic language is: How are conditions solved and are there constructive methods to avoid a blind guessing of values for unknowns? This is the purpose of narrowing strategies that are discussed next.

2.2 Narrowing

Techniques for goal solving are well developed in the area of logic programming. Since functional languages advocate the equational definition of functions, it is a natural idea to integrate both paradigms by adding an equality predicate to logic programs, leading to *equational logic programming* [73,74]. On the operational side, the resolution principle of logic programming must be extended to deal with replacements of subterms. *Narrowing*, originally introduced in automated theorem proving [81], is a constructive method to deal with such replacements. For this purpose, defining equations are interpreted as rewrite rules that are only applied from left to right (as in functional programming). In contrast to functional programming, the left-hand side of a defining equation is *unified* with the subterm under evaluation. In order to provide more detailed definitions, some basic notions of term rewriting [18,29] are briefly recalled. Although the theoretical part uses notations from term rewriting, its mapping into the concrete programming language syntax should be obvious.

Since we ignore polymorphic types in the theoretical part of this survey, we consider a many-sorted *signature* Σ partitioned into a set $\mathcal{C}$ of *constructors* and a set $\mathcal{F}$ of (defined) *functions* or *operations*. We write $c/n \in \mathcal{C}$ and $f/n \in \mathcal{F}$ for n-ary constructor and operation symbols, respectively. Given a set of variables $\mathcal{X}$, the set of *terms* and *constructor terms* are denoted by $\mathcal{T}(\mathcal{C} \cup \mathcal{F}, \mathcal{X})$ and $\mathcal{T}(\mathcal{C}, \mathcal{X})$, respectively. The set of variables occurring in a term t is denoted by $\mathcal{V}ar(t)$. A term t is *ground* if $\mathcal{V}ar(t) = \varnothing$. A term is *linear* if it does not contain multiple occurrences of one variable. A term is *operation-rooted* (*constructor-rooted*) if its root symbol is an operation (constructor). A *head normal form* is a term that is not operation-rooted, i.e., a variable or a constructor-rooted term.

A *pattern* is a term of the form $f(d_1, \ldots, d_n)$ where $f/n \in \mathcal{F}$ and $d_1, \ldots, d_n \in \mathcal{T}(\mathcal{C}, \mathcal{X})$. A *term rewriting system* (TRS) is set of rewrite rules, where an (unconditional) *rewrite rule* is a pair $l \to r$ with a linear pattern l as the *left-hand side* (*lhs*) and a term r as the *right-hand side* (*rhs*). Note that this definition reflects

the specific properties of functional logic programs. Traditional term rewriting systems [29] differ from this definition in the following points:

1. We have required that the left-hand sides must be linear patterns. Such rewrite systems are also called *constructor-based* and exclude rules like

   ```
   (xs ++ ys) ++ zs  =  xs ++ (ys ++zs)        (assoc)
   last (xs ++ [e])  =  e                      (last)
   ```

   Although this seems to be a restriction when one is interested in writing equational specifications, it is not a restriction from a programming language point of view, since functional as well as logic programming languages enforces the same requirement (although logic languages do not require linearity of patterns, this can be easily obtained by introducing new variables and adding equations for them in the condition; conditional rules are discussed below). Often, non-constructor-based rules specify properties of functions rather than providing a constructive definition (compare rule *assoc* above that specifies the associativity of "`++`"), or they can be transformed into constructor-based rules by moving non-constructor terms in left-hand side arguments into the condition (e.g., rule *last*). Although there exist narrowing strategies for non-constructor-based rewrite rules (see [38,74,81] for more details), they often put requirements on the rewrite system that are too strong or difficult to check in universal programming languages, like termination or confluence. An important insight from recent works on functional logic programming is the restriction to constructor-based programs since this supports the development of efficient and practically useful evaluation strategies (see below).
2. Traditional rewrite rules $l \to r$ require that $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$. A TRS where all rules satisfy this restriction is also called a *TRS without extra variables*. Although this makes sense for rewrite-based languages, it limits the expressive power of functional logic languages (see the definition of `last` in Section 2.1). Therefore, functional logic languages usually do not have this variable requirement, although some theoretical results have only been proved under this requirement.

In order to formally define computations w.r.t. a TRS, we need a few further notions. A *position* p in a term t is represented by a sequence of natural numbers. Positions are used to identify particular subterms. Thus, $t|_p$ denotes the *subterm* of t at position p, and $t[s]_p$ denotes the result of *replacing the subterm* $t|_p$ by the term s (see [29] for details). A *substitution* is an idempotent mapping $\sigma : \mathcal{X} \to \mathcal{T}(\mathcal{C} \cup \mathcal{F}, \mathcal{X})$ where the *domain* $\mathcal{D}om(\sigma) = \{x \in \mathcal{X} \mid \sigma(x) \neq x\}$ is finite. Substitutions are obviously extended to morphisms on terms. We denote by $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ the *substitution* σ with $\sigma(x_i) = t_i$ $(i = 1, \ldots, n)$ and $\sigma(x) = x$ for all other variables x. A substitution σ is *constructor* (*ground constructor*) if $\sigma(x)$ is a constructor (ground constructor) term for all $x \in \mathcal{D}om(\sigma)$.

A *rewrite step* $t \to_{p,R} t'$ (in the following, p and R will often be omitted in the notation of rewrite and narrowing steps) is defined if p is a position in

t, $R = l \to r$ is a rewrite rule with fresh variables,[3] and σ is a substitution with $t|_p = \sigma(l)$ and $t' = t[\sigma(r)]_p$. The instantiated lhs $\sigma(l)$ is also called a *redex* (*red*ucible *ex*pression). A term t is called in *normal form* if there is no term s with $t \to s$. $\stackrel{*}{\to}$ denotes the reflexive and transitive closure of a relation $\to$.

Rewrite steps formalize functional computation steps with pattern matching as introduced in Section 2.1. The goal of a sequence of rewrite steps is to compute a normal form. A *rewrite strategy* determines for each rewrite step a rule and a position for applying the next step. A *normalizing strategy* is one that terminates a rewrite sequence in a normal form, if it exists. Note, however, that normal forms are not necessarily the interesting results of functional computations, as the following example shows.

Example 1. Consider the operation

```
idNil [] = []
```

that is the identity on the empty list but undefined for non-empty lists. Then, a normal form like "`idNil [1]`" is usually considered as an error rather than a result. Actually, Haskell reports an error for evaluating the term "`idNil [1+2]`" rather than delivering the normal form "`idNil [3]`". □

Therefore, the interesting results of functional computations are *constructor terms* that will be also called *values*. Evaluation strategies used in functional programming, such as lazy evaluation, are not normalizing, as the previous example shows.

Functional logic languages are able to do more than pure rewriting since they instantiate variables in a term (also called *free* or *logic variables*) so that a rewrite step can be applied. The combination of variable instantiation and rewriting is called *narrowing*. Formally, $t \leadsto_{p,R,\sigma} t'$ is a *narrowing step* if p is a non-variable position in t (i.e., $t|_p$ is not a variable) and $\sigma(t) \to_{p,R} t'$. Since the substitution σ is intended to instantiate the variables in the term under evaluation, one often restricts $\mathcal{D}om(\sigma) \subseteq \mathcal{V}ar(t)$. We denote by $t_0 \leadsto^*_\sigma t_n$ a sequence of narrowing steps $t_0 \leadsto_{\sigma_1} \ldots \leadsto_{\sigma_n} t_n$ with $\sigma = \sigma_n \circ \cdots \circ \sigma_1$ (where $\sigma = \{\}$ in the case of $n = 0$). Since in functional logic languages we are interested in computing *values* (constructor terms) as well as *answers* (substitutions), we say that the narrowing derivation $t \leadsto^*_\sigma c$ *computes the value* c *with answer* σ if c is a constructor term.

The above definition of narrowing is too general for a realistic implementation since it allows arbitrary instantiations of variables in the term under evaluation. Thus, all possible instantiations must be tried in order to compute all possible values and answers. Obviously, this does not lead to a practical implementation. Therefore, older narrowing strategies (see [38] for a detailed account) were influenced by the resolution principle and required that the substitution used in a narrowing step must be a most general unifier of $t|_p$ and the left-hand side of the applied rule. As shown in [9], this condition prevents the development of optimal evaluation strategies. Therefore, most recent narrowing strategies relax this

[3] In classical traditional term rewriting, fresh variables are not used when a rule is applied. Since we consider also rules containing extra variables in right-hand sides, it is important to replace them by fresh variables when the rule is applied.

traditional requirement but provide another constructive method to compute a small set of unifiers in narrowing steps, as we will see below. The next example shows the non-optimality of narrowing with most general unifiers.

Example 2. Consider the following program containing a declaration of natural numbers in Peano's notation and two operations for addition and a "less than or equal" test (the pattern "`_`" denotes an unnamed *anonymous variable*):

```
data Nat = O | S Nat

add O     y = y
add (S x) y = S (add x y)

leq O     _     = True                    (leq1)
leq (S _) O     = False                   (leq2)
leq (S x) (S y) = leq x y                 (leq3)
```

Consider the initial term "`leq v (add w O)`" where `v` and `w` are free variables. By applying rule leq_1, `v` is instantiated to `O` and the result `True` is computed:

$$\texttt{leq v (add w O)} \leadsto_{\{\texttt{v}\mapsto\texttt{O}\}} \texttt{True}$$

Further answers can be obtained by instantiating `v` to `(S...)`. This requires the evaluation of the subterm `(add w O)` in order to allow the application of rule leq_2 or leq_3. For instance, the following narrowing derivation computes the value `False` with answer $\{\texttt{v} \mapsto \texttt{S z}, \texttt{w} \mapsto \texttt{O}\}$:

$$\texttt{leq v (add w O)} \leadsto_{\{\texttt{w}\mapsto\texttt{O}\}} \texttt{leq v O} \leadsto_{\{\texttt{v}\mapsto\texttt{S z}\}} \texttt{False}$$

However, we can also apply rule leq_1 in the second step of the previous narrowing derivation and obtain the following derivation:

$$\texttt{leq v (add w O)} \leadsto_{\{\texttt{w}\mapsto\texttt{O}\}} \texttt{leq v O} \leadsto_{\{\texttt{v}\mapsto\texttt{O}\}} \texttt{True}$$

Obviously, the last derivation is not optimal since it computes the same value as the first derivation with a less general answer and needs one more step. This derivation can be avoided by instantiating `v` to `S z` in the first narrowing step:

$$\texttt{leq v (add w O)} \leadsto_{\{\texttt{v}\mapsto\texttt{S z},\ \texttt{w}\mapsto\texttt{O}\}} \texttt{leq (S z) O}$$

Now, rule leq_1 is no longer applicable, as intended. However, this first narrowing step contains a substitution that is not a most general unifier between the evaluated subterm `(add w O)` and the left-hand side of some rule for `add`. □

Needed Narrowing. The first narrowing strategy that advocated the use of non-most general unifiers and for which optimality results have been shown is needed narrowing [9]. Furthermore, needed narrowing steps can be efficiently computed. Therefore, it has become the basis of modern functional logic languages[4].

Needed narrowing is based on the idea to perform only narrowing steps that are in some sense necessary to compute a result (such strategies are also called

[4] Concrete languages and implementations add various extensions in order to deal with larger classes of programs that will be discussed later.

lazy or *demand-driven*). For doing so, it analyzes the left-hand sides of the rewrite rules of a function under evaluation (starting from an outermost function). If there is an argument position where all left-hand sides are constructor-rooted, the corresponding actual argument must be also rooted by one of the constructors in order to apply a rewrite step. Thus, the actual argument is evaluated to head normal form if it is operation-rooted and, if it is a variable, nondeterministically instantiated with some constructor.

Example 3. Consider again the program of Example 2. Since the left-hand sides of all rules for `leq` have a constructor-rooted first argument, needed narrowing instantiates the variable `v` in "`leq v (add w 0)`" to either `0` or `S z` (where `z` is a fresh variable). In the first case, only rule leq_1 becomes applicable. In the second case, only rules leq_2 or leq_3 become applicable. Since the latter rules have both a constructor-rooted term as the second argument, the corresponding subterm `(add w 0)` is recursively evaluated to a constructor-rooted term before applying one of these rules. □

Since there are TRSs with rules that do not allow such a reasoning, needed narrowing is defined on the subclass of *inductively sequential* TRSs. This class can be characterized by definitional trees [4] that are also useful to formalize and implement various narrowing strategies. Since only the left-hand sides of rules are important for the applicability of needed narrowing, the following characterization of definitional trees [5] considers patterns partially ordered by subsumption (the *subsumption ordering* on terms is defined by $t \leq \sigma(t)$ for a term t and substitution σ).

A *definitional tree* of an operation f is a non-empty set T of linear patterns partially ordered by subsumption having the following properties:

Leaves property: The maximal elements of T, called the *leaves*, are exactly the (variants of) the left-hand sides of the rules defining f. Non-maximal elements are also called *branches*.

Root property: T has a minimum element, called the *root*, of the form $f(x_1, \ldots, x_n)$ where $x_1, \ldots, x_n$ are pairwise distinct variables.

Parent property: If $\pi \in T$ is a pattern different from the root, there exists a unique $\pi' \in T$, called the *parent* of π (and π is called a *child* of π'), such that $\pi' < \pi$ and there is no other pattern $\pi'' \in \mathcal{T}(\mathcal{C} \cup \mathcal{F}, \mathcal{X})$ with $\pi' < \pi'' < \pi$.

Induction property: All the children of a pattern π differ from each other only at a common position, called the *inductive position*, which is the position of a variable in π[5].

An operation is called *inductively sequential* if it has a definitional tree and its rules do not contain extra variables. A TRS is inductively sequential if all its defined operations are inductively sequential. Intuitively, inductively sequential functions are defined by structural induction on the argument types. Purely

[5] There might be more than one potential inductive position when constructing a definitional tree. In this case one can select any of them since the results about needed narrowing do not depend on the selected definitional tree.

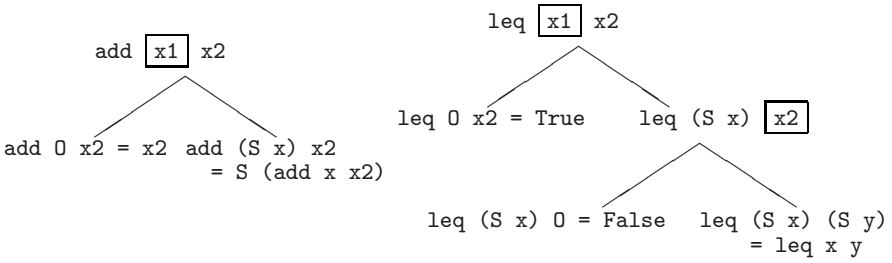


Fig. 1. Definitional trees of the operations `add` and `leq`

functional programs and the vast majority of functions in functional logic programs are inductively sequential. Thus, needed narrowing is applicable to most functions, although extensions are useful for particular functions (see below).

It is often convenient and simplifies the understanding to provide a graphic representation of definitional trees, where each inner node is marked with a pattern, the inductive position in branches is surrounded by a box, and the leaves contain the corresponding rules. For instance, the definitional trees of the operations `add` and `leq`, defined in Example 2, are illustrated in Figure 1.

The formal definition of needed narrowing is based on definitional trees and can be found in [9]. A definitional tree can be computed at compile time (see [7,41] for algorithms to construct definitional trees) and contains all information for the efficient implementation of the decisions to be made at run time (compare Example 3). Intuitively, a needed narrowing step is applied to an operation-rooted term t by considering a definitional tree (with fresh variables) for the operation at the root. The tree is recursively processed from the root until one finds a maximal pattern that unifies with t. Thus, to compute a needed narrowing step, one starts with the root pattern of the definitional tree and performs at each level with pattern π the following case distinction:

- If π is a leaf, we apply the corresponding rule.
- If π is a branch and p its inductive position, we consider the corresponding subterm $t|_p$:
  1. If $t|_p$ is rooted by a constructor c and there is a child π' of π having c at the inductive position, we proceed by examining π'. If there is no such child, we fail, i.e., no needed narrowing step is applicable.
  2. If $t|_p$ is a variable, we nondeterministically instantiate this variable by the constructor term at the inductive position of a child π' of π and proceed with π'.
  3. If $t|_p$ is operation-rooted, we recursively apply the computation of a needed narrowing step to $\sigma(t|_p)$, where σ is the instantiation of the variables of t performed in the previous case distinctions.

As discussed above, the failure to compute a narrowing step in case (1) is not a weakness but advantageous when we want to compute values. For instance, consider the term t = `idNil [1+2]` where the operation `idNil` is as defined

in Example 1. A normalizing strategy performs a step to compute the normal form `idNil [3]` whereas needed narrowing immediately fails since there exists no value as a result. Thus, the early failure of needed narrowing avoids wasting resources.

As a consequence of the previous behavior, the properties of needed narrowing are stated w.r.t. constructor terms as results. In particular, the equality symbol "`=:=`" in goals is interpreted as the *strict equality* on terms, i.e., the equation t_1 `=:=` t_2 is satisfied iff t_1 and t_2 are reducible to the same ground constructor term. In contrast to the mathematical notion of equality as a congruence relation, strict equality is not reflexive. Similarly to the notion of result values, this is intended in programming languages where an equation between functional expressions that do not have a value, like "`idNil [1] =:= idNil [1]`", is usually not considered as true. Furthermore, normal forms or values might not exist so that reflexivity is not a feasible property of equational constraints (see [34] for a more detailed discussion on this topic).

Strict equality can be defined as a binary function by the following set of (inductively sequential) rewrite rules. The constant `Success` denotes a solved (equational) constraint and is used to represent the result of successful evaluations.

$$\begin{array}{rcll} c \texttt{ =:= } c & = & \texttt{Success} & \forall c/0 \in \mathcal{C} \\ c\ x_1 \ldots x_n \texttt{ =:= } c\ y_1 \ldots y_n & = & x_1\texttt{=:=}y_1\ \texttt{\&...\&}\ x_n\texttt{=:=}y_n & \forall c/n \in \mathcal{C}, n > 0 \\ \texttt{Success \& Success} & = & \texttt{Success} & \end{array}$$

Thus, it is sufficient to consider strict equality as any other function. Concrete functional logic languages provide more efficient implementations of strict equality where variables can be bound to other variables instead of instantiating them to ground terms (see also Section 3.3).

Now we can state the main properties of needed narrowing. A (correct) *solution* for an equation t_1 `=:=` t_2 is a constructor substitution σ (note that constructor substitutions are desired in practice since a broader class of solutions would contain unevaluated or undefined expressions) if $\sigma(t_1)$ `=:=` $\sigma(t_2) \stackrel{*}{\rightarrow}$ `Success`. Needed narrowing is sound and complete, i.e., all computed solutions are correct and for each correct solution a possibly more general one is computed, and it does not compute redundant solutions in different derivations:

Theorem 1 ([9]). *Let $\mathcal{R}$ be an inductively sequential TRS and e an equation.*

1. *(Soundness) If $e \leadsto^*_\sigma$* `Success` *is a needed narrowing derivation, then σ is a solution for e.*
2. *(Completeness) For each solution σ of e, there exists a needed narrowing derivation $e \leadsto^*_{\sigma'}$* `Success` *with $\sigma'(x) \leq \sigma(x)$ for all $x \in \mathcal{V}ar(e)$.*
3. *(Minimality) If $e \leadsto^*_\sigma$* `Success` *and $e \leadsto^*_{\sigma'}$* `Success` *are two distinct needed narrowing derivations, then σ and σ' are independent on $\mathcal{V}ar(e)$, i.e., there is some $x \in \mathcal{V}ar(e)$ such that $\sigma(x)$ and $\sigma'(x)$ are not unifiable.*

Furthermore, in successful derivations, needed narrowing computes only steps that are necessary to obtain the result and, consequently, it computes the *shortest of all possible narrowing derivations* if derivations on common subterms are

shared (a standard implementation technique in non-strict functional languages) [9, Corollary 1]. Needed narrowing is currently the only narrowing strategy with such strong results. Therefore, it is an adequate basis for modern functional logic languages, although concrete implementations support extensions that are discussed next.

Weakly Needed Narrowing. Inductively sequential TRS are a proper subclass of (constructor-based) TRSs. Although the majority of function definitions are inductively sequential, there are also functions where it is more convenient to relax this requirement. An interesting superclass are *weakly orthogonal TRSs*. These are rewrite systems where left-hand sides can overlap in a semantically trivial way. Formally, a TRS without extra variables (recall that we consider only left-linear constructor-based rules) is *weakly orthogonal* if $\sigma(r_1) = \sigma(r_2)$ for all (variants of) rules $l_1 \to r_1$ and $l_2 \to r_2$ and substitutions σ with $\sigma(l_1) = \sigma(l_2)$.

Example 4. A typical example of a weakly orthogonal TRS is the *parallel-or*, defined by the rules:

```
or True  _     = True          (or1)
or _     True  = True          (or2)
or False False = False         (or3)
```

A term like "`or` s t" could be reduced to `True` whenever one of the arguments s or t evaluates to `True`. However, it is not clear which of the arguments should be evaluated first, since any of them could result in a nonterminating derivation. `or` has no definitional tree and, thus, needed narrowing can not be applied. □

In rewriting, several normalizing strategies for weakly orthogonal TRSs have been proposed, like parallel outermost [72] or weakly needed [80] rewriting that are based on the idea to replace several redexes in parallel in one step. Since strategies for functional logic languages already support nondeterministic evaluations, one can exploit this feature to extend needed narrowing to a *weakly needed narrowing* strategy. The basic idea is to generalize the notion of definitional trees to include *or-branches* which conceptually represent a union of definitional trees [4,8,64]. If such an or-branch is encountered during the evaluation of a narrowing step, weakly needed narrowing performs a nondeterministic guess and proceeds with the subtrees below the or-branches. Weakly needed narrowing is no longer optimal in the sense of needed narrowing but sound and complete for weakly orthogonal TRS in the sense of Theorem 1 [8].

Overlapping Inductively Sequential Systems. Inductively sequential and weakly orthogonal TRSs are *confluent*, i.e., each term has at most one normal form. This property is reasonable for functional languages since it ensures that operations are well defined (partial) functions in the mathematical sense. Since the operational mechanism of functional logic languages is more powerful due to its built-in search mechanism, in this context it makes sense to consider also operations defined by non-confluent TRSs. Such operations are also called *nondeterministic*. The prototype of such a nondeterministic operation is a binary operation "`?`" that returns one of its arguments:

```
x ? y = x
x ? y = y
```

Thus, the expression "`0 ? 1`" has two possible results, namely `0` or `1`.

Since functional logic languages already handle nondeterministic computations, they can deal with such nondeterministic operations. If operations are interpreted as mappings from values into sets of values (actually, due to the presence of recursive non-strict functions, algebraic structures with cones of partially ordered sets are used instead of sets, see [36] for details), one can provide model-theoretic and proof-theoretic semantics with the usual properties (minimal term models, equivalence of model-theoretic and proof-theoretic solutions, etc). Thus, functional logic programs with nondeterministic operations are still in the design space of declarative languages. Moreover, nondeterministic operations have advantages w.r.t. demand-driven evaluation strategies so that they became a standard feature of recent functional logic languages (whereas older languages put confluence requirements on their programs). The following example discusses this in more detail.

Example 5. Based on the binary operation "`?`" introduced above, one can define an operation `insert` that nondeterministically inserts an element at an arbitrary position in a list:

```
insert e []     = [e]
insert e (x:xs) = (e : x : xs) ? (x : insert e xs)
```

Exploiting this operation, one can define an operation `perm` that returns an arbitrary permutation of a list:

```
perm []     = []
perm (x:xs) = insert x (perm xs)
```

One can already see an important property when one reasons about nondeterministic operations: the computation of results is arbitrary, i.e., one result is as good as any other. For instance, if one evaluates `perm [1,2,3]`, any permutation (e.g., `[3,2,1]` as well as `[1,3,2]`) is an acceptable result. If one puts specific conditions on the results, the completeness of the underlying computational model (e.g., INS, see below) ensures that the appropriate results meeting these conditions are selected.

For instance, one can use `perm` to define a sorting function `psort` based on a "partial identity" function `sorted` that returns its input list if it is sorted:

```
sorted []                                  = []
sorted [x]                                 = [x]
sorted (x1:x2:xs) | leq x1 x2 =:= True = x1 : sorted (x2:xs)

psort xs = sorted (perm xs)
```

Thus, `psort xs` returns only those permutations of `xs` that are sorted. The advantage of this definition of `psort` in comparison to traditional "generate-and-test" solutions becomes apparent when one considers the demand-driven evaluation strategy (note that one can apply the weakly needed narrowing strategy

to such kinds of programs since this strategy is based only on the left-hand sides of the rules but does not exploit confluence). Since in an expression like `sorted (perm xs)` the argument `(perm xs)` is only evaluated as demanded by `sorted`, the permutations are not fully computed at once. If a permutation starts with a non-ordered prefix, like `S 0 : 0 : perm xs`, the application of the third rule of `sorted` fails and, thus, the computation of the remaining part of the permutation (which can result in $n!$ different permutations if n is the length of the list `xs`) is discarded. The overall effect is a reduction in complexity in comparison to the traditional generate-and-test solution. □

This example shows that nondeterministic operations allow the transformation of "generate-and-test" solutions into "test-of-generate" solutions with a lower complexity since the demand-driven narrowing strategy results in a demand-driven construction of the search space (see [5,36] for further examples). Antoy [5] shows that desirable properties of needed narrowing can be transferred to programs with nondeterministic functions if one considers *overlapping inductively sequential systems*. These are TRSs with inductively sequential rules where each rule can have multiple right-hand sides (basically, inductively sequential TRSs with occurrences of "`?`" in the top-level of right-hand sides), possibly containing extra variables. For instance, the rules defining `insert` form an overlapping inductively sequential TRS if the second rule is interpreted as a single rule with two right-hand sides ("`e:x:xs`" and "`x : insert e xs`"). The corresponding strategy, called *INS* (*inductively sequential narrowing strategy*), is defined similarly to needed narrowing but computes for each narrowing step a set of replacements. INS is a conservative extension of needed narrowing and optimal modulo nondeterministic choices of multiple right-hand sides, i.e., if there are no multiple right-hand sides or there is an oracle for choosing the appropriate element from multiple right-hand sides, INS has the same optimality properties as needed narrowing (see [5] for more details).

A subtle aspect of nondeterministic operations is their treatment if they are passed as arguments. For instance, consider the operation `coin` defined by

```
coin = 0 ? 1
```

and the expression "`double coin`" (where `double` is defined as in Section 2.1). If the argument `coin` is evaluated (to `0` or `1`) before it is passed to `double`, we obtain the possible results `0` and `2`. However, if the argument `coin` is passed unevaluated to `double`, we obtain after one rewrite step the expression `coin+coin` which has four possible rewrite derivations resulting in the values `0`, `1`, `1`, and `2`. The former behavior is referred to as *call-time choice* semantics [60] since the choice for the desired value of a nondeterministic operation is made at call time, whereas the latter is referred to as *need-time choice* semantics. There are arguments for either of these semantics depending on the programmer's intention (see [7] for more examples).

Although call-time choice suggests an eager or call-by-value strategy, it fits well into the framework of demand-driven evaluation where arguments are shared to avoid multiple evaluations of the same subterm. For instance, the actual

subterm (e.g., `coin`) associated to argument `x` in the rule "`double x = x+x`" is not duplicated in the right-hand side but a reference to it is passed so that, if it is evaluated by one subcomputation, the same result will be taken in the other subcomputation. This technique, called *sharing*, is essential to obtain efficient (and optimal) evaluation strategies. If sharing is used, the call-time choice semantics can be implemented without any further machinery. Furthermore, in many situations call-time choice is the semantics with the "least astonishment". For instance, consider the reformulation of the operation `psort` in Example 5 to

```
psort xs = idOnSorted (perm xs)

idOnSorted xs | sorted xs =:= xs = xs
```

Then, for the call `psort xs`, the call-time choice semantics delivers only sorted permutations of `xs`, as expected, whereas the need-time choice semantics delivers all permutations of `xs` since the different occurrences of `xs` in the rule of `idOnSorted` are not shared. Due to these reasons, current functional logic languages usually adopt the call-time choice semantics.

Conditional Rules. The narrowing strategies presented so far are defined for rewrite rules without conditions, although some of the concrete program examples indicate that conditional rules are convenient in practice. Formally, a *conditional rewrite rule* has the form $l \to r \Leftarrow C$ where l and r are as in the unconditional case and the condition C consists of finitely many equational constraints of the form s `=:=` t. In order to apply weakly needed narrowing to conditional rules, one can transform a conditional rule of the form

$$l \to r \Leftarrow s_1 \texttt{=:=} t_1 \ \ldots \ s_n \texttt{=:=} t_n$$

into an unconditional rule

$$l \to \mathit{cond}(s_1 \texttt{=:=} t_1 \ \texttt{\&} \ldots \texttt{\&} \ s_n \texttt{=:=} t_n, \ r)$$

where the "*cond*itional" is defined by $\mathit{cond}(\texttt{Success}, x) \to x$. Actually, Antoy [6] has shown a systematic method to translate any conditional constructor-based TRS into an overlapping inductively sequential TRS performing equivalent computations.

2.3 Rewriting Logic

As discussed in the previous section on overlapping inductively sequential TRS, sharing becomes important for the semantics of nondeterministic operations. This has the immediate consequence that traditional equational reasoning is no longer applicable. For instance, the expressions `double coin` and `coin+coin` are not equal since the latter can reduce to `1` while this is impossible for the former w.r.t. a call-time choice semantics. In order to provide a semantical basis for such general functional logic programs, González-Moreno et al. [36] have proposed the rewriting logic *CRWL* (Constructor-based conditional ReWriting Logic) as a logical (execution- and strategy-independent) foundation for declarative programming with non-strict and nondeterministic operations and call-time choice semantics. This logic has been also used to link a natural model theory

as an extension of the traditional theory of logic programming and to establish soundness and completeness of narrowing strategies for rather general classes of TRSs [28].

To deal with non-strict functions, CRWL considers signatures $\Sigma_\bot$ that are extended by a special symbol $\bot$ to represent *undefined values*. For instance, $\mathcal{T}(\mathcal{C} \cup \{\bot\}, \mathcal{X})$ denotes the set of partial constructor terms, e.g., `1:2:`$\bot$ denotes a list starting with elements `1` and `2` and an undefined rest. Such *partial terms* are considered as finite approximations of possibly infinite values. CRWL defines the deduction of two kinds of basic statements: *approximation statements* $e \to t$ with the intended meaning "the partial constructor term t approximates the value of e", and *joinability statements* e_1 `:=:` e_2 with the intended meaning that e_1 and e_2 have a common *total* approximation $t \in \mathcal{T}(\mathcal{C}, \mathcal{X})$ with $e_1 \to t$ and $e_2 \to t$, thus modeling strict equality with terms containing variables. To model call-time choice semantics, rewrite rules are only applied to partial *values*. Hence, the following notation for *partial constructor instances* of a set of (conditional) rules $\mathcal{R}$ is useful:

$$[\mathcal{R}]_\bot = \{\sigma(l \to r \Leftarrow C) \mid l \to r \Leftarrow C \in \mathcal{R}, \sigma : \mathcal{X} \to \mathcal{T}(\mathcal{C} \cup \{\bot\}, \mathcal{X})\}$$

Then CRWL is defined by the following set of inference rules (where the program is represented by a TRS $\mathcal{R}$):

(Bottom) $e \to \bot$ for any $e \in \mathcal{T}(\mathcal{C} \cup \mathcal{F} \cup \{\bot\}, \mathcal{X})$

(Restricted reflexivity) $x \to x$ for any variable $x \in \mathcal{X}$

(Decomposition)
$$\frac{e_1 \to t_1 \ \cdots \ e_n \to t_n}{c(e_1, \ldots, e_n) \to c(t_1, \ldots, t_n)}$$
for any $c/n \in \mathcal{C}$, $t_i \in \mathcal{T}(\mathcal{C} \cup \{\bot\}, \mathcal{X})$

(Function reduction)
$$\frac{e_1 \to t_1 \ \cdots \ e_n \to t_n \quad C \quad r \to t}{f(e_1, \ldots, e_n) \to t}$$
for any $f(t_1, \ldots, t_n) \to r \Leftarrow C \in [\mathcal{R}]_\bot$ and $t \neq \bot$

(Joinability)
$$\frac{e_1 \to t \qquad e_2 \to t}{e_1 \texttt{ :=: } e_2}$$
for any total term $t \in \mathcal{T}(\mathcal{C}, \mathcal{X})$

The first rule specifies that $\bot$ approximates any expression. The condition $t \neq \bot$ in rule (Function reduction) avoids unnecessary applications of this rule since this case is already covered by the first rule. The restriction to partial constructor instances in this rule formalizes non-strict functions with a call-time choice semantics. Functions might have non-strict arguments that are not evaluated since the corresponding actual arguments can be derived to $\bot$ by the first rule. If the value of an argument is required to evaluate the right-hand side of a function's rule, it must be evaluated to a partial constructor term before it is passed to the right-hand side (since $[\mathcal{R}]_\bot$ contains only partial constructor instances), which corresponds to a call-time choice semantics. Note that this does not prohibit the use of lazy implementations since this semantical behavior can be enforced by sharing unevaluated expressions. Actually, [36] defines a lazy narrowing calculus that reflects this behavior.

CRWL can be used as the logical foundation of functional logic languages with non-strict nondeterministic operations. It is a basis for the verification of functional logic programs [27] and has been extended in various directions, e.g., higher-order operations [37], algebraic types [17], polymorphic types [35], failure [68], constraints [67] etc. An account on CRWL and its applications can be found in [77].

2.4 Residuation

Although narrowing extends soundness and completeness results of logic programming to the general framework of functional logic programming, it is not the only method that has been proposed to integrate functions into logic programs. An alternative technique, called *residuation*, is based on the idea to delay or suspend function calls until they are ready for deterministic evaluation. The residuation principle is used, for instance, in the languages Escher [63], Le Fun [2], Life [1], NUE-Prolog [71], and Oz [82]. Since the residuation principle evaluates function calls by deterministic reduction steps, nondeterministic search must be encoded by predicates [1,2,71] or disjunctions [63,82]. Moreover, if some part of a computation might suspend, one needs a primitive to execute computations concurrently. For instance, the conjunction of constraints "`&`" needs to evaluate both arguments to `Success` so that it is reasonable to do it concurrently, i.e., if the evaluation of one argument suspends, the other one is evaluated.

Example 6. Consider Example 2 together with the operation

```
nat 0     = Success
nat (S x) = nat x
```

If the function `add` is evaluated by residuation, i.e., suspends if the first argument is a variable, the expression "`add y 0 =:= S 0 & nat y`" is evaluated as follows:

$$
\begin{array}{lll}
\texttt{add y 0 =:= S 0 \& }\underline{\texttt{nat y}} & \rightarrow_{\{\texttt{y}\mapsto\texttt{S x}\}} & \underline{\texttt{add (S x) 0}}\texttt{ =:= S 0 \& nat x} \\
 & \rightarrow_{\{\}} & \underline{\texttt{S (add x 0) =:= S 0}}\texttt{ \& nat x} \\
 & \rightarrow_{\{\}} & \texttt{add x 0 =:= 0 \& }\underline{\texttt{nat x}} \\
 & \rightarrow_{\{\texttt{x}\mapsto\texttt{0}\}} & \underline{\texttt{add 0 0}}\texttt{ =:= 0 \& Success} \\
 & \rightarrow_{\{\}} & \underline{\texttt{0 =:= 0}}\texttt{ \& Success} \\
 & \rightarrow_{\{\}} & \underline{\texttt{Success \& Success}} \\
 & \rightarrow_{\{\}} & \texttt{Success}
\end{array}
$$

Thus, the solution $\{\texttt{y} \mapsto \texttt{S 0}\}$ is computed by switching between the residuating function `add` and the constraint `nat` that instantiates its argument to natural numbers. □

Narrowing and residuation are quite different approaches to integrate functional and logic programming. Narrowing is sound and complete but requires the nondeterministic evaluation of function calls if some arguments are unknown. Residuation might not compute some result due to the potential suspension of evaluation but avoids guessing on functions. From an operational point of view,

there is no clear advantage of one of the strategies. One might have the impression that the deterministic evaluation of functions in the case of residuation is more efficient, but there are examples where residuation has an infinite computation space whereas narrowing has a finite one (see [39] for more details). On the other hand, residuation offers a concurrent evaluation principle with synchronization on logic variables (sometimes also called *declarative concurrency* [84]) and a conceptually clean method to connect *external functions* to declarative programs [21] (note that narrowing requires functions to be explicitly defined by rewrite rules). Therefore, it is desirable to integrate both principles in a single framework. This has been proposed in [41] where residuation is combined with weakly needed narrowing by extending definitional trees with branches decorated with a *flexible/rigid* tag. Operations with flexible tags are evaluated as with narrowing whereas operations with rigid tags suspend if the arguments are not sufficiently instantiated. The overall strategy is similar to weakly needed narrowing with the exception that a rigid branch with a free variable in the corresponding inductive position results in the suspension of the function under evaluation. For instance, if the branch of `add` in Figure 1 has a rigid tag, then `add` is evaluated as shown in Example 6.

3 Aspects of Multi-paradigm Languages

This section discusses some aspects of multi-paradigm languages that are relevant for their use in application programming. As before, we use the language Curry for concrete examples. Its syntax has been already introduced in an informal manner. Conceptually, a Curry program is a constructor-based TRS. Thus, its *declarative semantics* is given by the rewriting logic CRWL, i.e., operations and constructors are non-strict with a call-time choice semantics for nondeterministic operations. The *operational semantics* is based on weakly needed narrowing with sharing and residuation. Thus, for (flexible) inductively sequential operations, which form the vast majority of operations in application programs, the evaluation strategy is optimal w.r.t. the length of derivations and number of computed solutions and always computes a value if it exists (in case of nondeterministic choices only if the underlying implementation is fair w.r.t. such choices, as [14,15,56]). Therefore, the programmer can concentrate on the declarative meaning of programs and needs less attention to the consequences of the particular evaluation strategy (see [45] for a more detailed discussion).

3.1 External Operations

Operations that are externally defined, i.e., not implemented by explicit rules, like basic arithmetic operators or I/O operations, can not be handled by narrowing. Therefore, residuation is an appropriate model to connect external operations in a conceptually clean way (see also [21]): their semantics can be considered as defined by a possibly infinite set of rules (e.g., see the definition of "`+`" in Section 2.1) whose behavior is implemented in some other programming

language. Usually, external operations can not deal with unevaluated arguments possibly containing logic variables. Thus, the arguments of external operations are reduced to ground values before they are called. If some arguments are not ground but contain logic variables, the call is suspended until the variables are bound to ground values, which corresponds to residuation.

3.2 Higher-Order Operations

The use of higher-order operations, i.e., operations that take other operations as arguments or yields them as results, is an important programming technique in functional languages so that it should be also covered by multi-paradigm declarative languages. Typical examples are the mapping of a function to all elements of a list (`map`) or a generic accumulator for lists (`foldr`):

```
map :: (a->b) -> [a] -> [b]
map _ []     = []
map f (x:xs) = f x : map f xs

foldr :: (a->b->b) -> b -> [a] -> b
foldr _ z []     = z
foldr f z (x:xs) = f x (foldr f z xs)
```

Logic languages often provide higher-order features through a transformation into a first-order program [87] by defining a predicate *apply* that implements the application of an arbitrary function of the program to an expression. This technique is also known as "defunctionalization" [76] and enough to support the higher-order features of current functional languages (e.g., lambda abstractions can be replaced by new function definitions). An important difference to purely functional languages shows up when the function to be applied is a logic variable. In this case, one can instantiate this variable to all possible functions occurring in the program [37]. Since this might result also in instantiations that are not intended w.r.t. the given types, one can restrict these instantiations to well-typed ones which requires to keep type information at run time [16,35]. Another option is the instantiation of function variables to (well-typed) lambda terms in order to cover programs that can reason about bindings and block structure [55]. Since all these options might result in huge search spaces due to function instantiation and their feasibility and usefulness for larger application programs is not clear, one can also choose a more pragmatic solution: function application is rigid, i.e., it suspends if the functional argument is a logic variable.

3.3 Constraints

Functional logic languages are able to solve equational constraints. As shown in Section 2.2, such constraints occur in conditions of conditional rules and are intended to restrict the applicability of the rewrite rule, i.e., a replacement with a conditional rule is only performed if the condition has been shown to be satisfied (e.g., compare the definition of `last` in Section 2.1). Thus, constraints are solved when conditional rules are applied.

In general, a syntactic extension is not necessary to include constraints. For instance, the language Curry has no specific constructs for constraints but constraints are simply expressions of type `Success`, i.e., the equational constraint "`=:=`" is a function of type "`a -> a -> Success`", and the *concurrent conjunction* "`&`" on constraints that evaluates both arguments in a non-specified order (see Section 2.4) is a function of type "`Success -> Success -> Success`".

If constraints are ordinary expressions, they are first-class values that can be passed in arguments or data structures. For instance, the following "constraint combinator" takes a list of constraints as input and creates a new constraint that is satisfied if all constraints in the input list are satisfied:

```
allValid :: [Success] -> Success
allValid []     = success
allValid (c:cs) = c & allValid cs
```

Here, `success` is not a constructor but denotes the trivial constraint that is always satisfied. Exploiting higher-order functions, one can define it also by

```
allValid = foldr (&) success
```

Note that the constructor `Success` was introduced in Section 2.2 only to provide a rewrite-based definition of strict equality. However, functional logic languages like Curry, Escher, or TOY use a more efficient implementation of strict equality. The main difference shows up when an equational constraint "`x =:= y`" between two logic variables `x` and `y` is solved. Solving it with the rewrite rules shown in Section 2.2, `x` and `y` are nondeterministically bound to ground constructor terms which usually results in an infinite search space. This can be avoided by binding one variable to the other, similar to logic programming.

One can easily integrate the features of constraint programming by adding basic constraints that deal with other constraint domains, like real arithmetic, Boolean, or finite domain constraints. Thus, typical applications of constraint logic programming can be covered and combined with features of lazy higher-order programming [10,19,30,31,67,70,77]. As an example demonstrating the compactness obtained by combining constraint programming with higher-order features, consider a solver for SuDoku puzzles[6] with finite domain constraints. If we represent the SuDoku matrix `m` as a list of lists of finite domain variables, the "SuDoku constraints" can be easily specified by

```
allValid (map allDifferent m) &
allValid (map allDifferent (transpose m)) &
allValid (map allDifferent (squaresOfNine m))
```

where `allDifferent` is the usual constraint stating that all variables in its argument list must have different values, `transpose` is the standard matrix transposition, and `squaresOfNine` computes the list of 3×3 sub-matrices. Then, a

[6] A SuDoku puzzle consists of a 9×9 matrix of digits between 1 and 9 so that each row, each column, and each of the nine 3×3 sub-matrices contain pairwise different digits. The challenge is to find the missing digits if some digits are given.

SuDoku puzzle can be solved with these constraints by adding the usual domain and labeling constraints (see [49] for more details).

3.4 Function Patterns

We have discussed in Section 2.2 the fundamental requirement of functional languages for *constructor-based* rewrite systems. This requirement is the key for practically useful implementations and excludes rules like

```
last (xs ++ [e])  =  e                                    (last)
```

The non-constructor pattern `(xs ++ [e])` in this rule can be eliminated by moving it into the condition part (see Section 2.1):

```
last l | xs++[e] =:= l = e   where xs,e free              (lastc)
```

However, the strict equality used in (*lastc*) has the disadvantage that all list elements are completely evaluated. Hence, an expression like `last [failed,3]` (where `failed` is an expression that has no value) leads to a failure. This disadvantage can be avoided by allowing *function patterns*, i.e., expressions containing defined functions, in arguments of a rule's left-hand side so that (*last*) becomes a valid rule. In order to base this extension on the existing foundations of functional logic programming as described so far, a function pattern is interpreted as an abbreviation of the set of constructor terms that is the result of evaluating (by narrowing) the function pattern. Thus, rule (*last*) abbreviates the following (infinite) set of rules:

```
last [x] = x
last [x1,x] = x
last [x1,x2,x] = x
...
```

Hence, the expression `last [failed,3]` reduces to `3` w.r.t. these rules. In order to provide a constructive implementation of this concept, [13] proposes a specific demand-driven unification procedure for function pattern unification that can be implemented similarly to strict equality. Function patterns are a powerful concept to express transformation problems in a high-level way. Concrete programming examples and syntactic conditions for the well-definedness of rules with function patterns can be found in [13].

3.5 Encapsulating Search

An essential difference between functional and logic computations is their determinism behavior. Functional computations are deterministic. This enables a reasonable treatment of I/O operations by the monadic approach where I/O actions are considered as transformations on the outside world [86]. The monadic I/O approach is also taken in languages like Curry, Escher, or Mercury. However, logic computations might cause (don't know) nondeterministic choices, i.e., a computation can be cloned and continued in two different directions. Since one can not clone the entire outside world, nondeterministic choices during monadic

I/O computations must be avoided. Since this might restrict the applicability of logic programming techniques in larger applications, there is a clear need to *encapsulate nondeterministic search* between I/O actions. For this purpose, one can introduce a primitive search operator [57,79] that returns nondeterministic choices as data so that typical search operators of Prolog, like `findall`, `once`, or negation-as-failure, can be implemented using this primitive. Unfortunately, the combination with demand-driven evaluation and sharing causes some complications. For instance, in an expression like

```
let y=coin in findall(...y...)
```

it is not obvious whether the evaluation of `coin` (introduced outside but demanded inside the search operator) should be encapsulated or not. Furthermore, the order of the solutions might depend on the evaluation time. These and more peculiarities are discussed in [22] where another primitive search operator is proposed:

```
getSearchTree :: a -> IO (SearchTree a)
```

Since `getSearchTree` is an I/O action, its result (in particular, the order of solutions) depends on the current environment, e.g., time of evaluation. It takes an expression and delivers a search tree representing the search space when evaluating the input:

```
data SearchTree a = Or [SearchTree a] | Val a | Fail
```

Based on this primitive, one can define various concrete search strategies as tree traversals. To avoid the complications w.r.t. shared variables, `getSearchTree` implements a *strong encapsulation view*, i.e., conceptually, the argument of `getSearchTree` is cloned before the evaluation starts in order to cut any sharing with the environment. Furthermore, the structure of the search tree is computed lazily so that an expression with infinitely many values does not cause the nontermination of the search operator if one is interested in only one solution.

3.6 Implementation

The definition of needed narrowing and its extensions shares many similarities with pattern matching in functional or unification in logic languages. Thus, it is reasonable to use similar techniques to implement functional logic languages. Due to the coverage of logic variables and nondeterministic search, one could try to translate functional logic programs into Prolog programs in order to exploit the implementation technology available for Prolog. Actually, there are various approaches to compile functional logic languages with demand-driven evaluation strategies into Prolog (e.g., [3,10,26,40,62,64]). Narrowing-based strategies can be compiled into pure Prolog whereas residuation (as necessary for external operations, see Section 3.1) demands for coroutining[7]. The compilation into Prolog has many advantages. It is fairly simple to implement, one can use constraint

[7] Note that external operations in Prolog do not use coroutining since they are implemented in a non-declarative way.

solvers available in many Prolog implementations in application programs, and one can exploit the advances made in efficient implementations of Prolog.

Despite these advantages, the transformation into Prolog has the drawback that one is fixed to Prolog's backtracking strategy to implement nondeterministic search. This hampers the implementation of encapsulated search or fair search strategies. Therefore, there are various approaches to use other target languages than Prolog. For instance, [15] presents techniques to compile functional logic programs into Java programs that implement a fair search for solutions, and [23] proposes a translation of Curry programs into Haskell programs that offers the primitive search operator `getSearchTree` introduced in Section 3.5. *Virtual machines* to compile functional logic programs are proposed in [14,56,69].

4 Applications

Since multi-paradigm declarative languages amalgamate the most important declarative paradigms, their application areas cover the areas of languages belonging to the individual paradigms. Therefore, we discuss in this section only applications that demonstrate the feasibility and advantages of multi-paradigm declarative programming.

A summary of design patterns exploiting combined functional and logic features for application programming can be found in [11]. These patterns are unique to *functional logic* programming and can not be directly applied in other paradigms. For instance, the *constraint constructor* pattern exploits the fact that functional logic languages can deal with failure so that conditions about the validity of data represented by general structures can be encoded directly in the data structures rather than in application programs. This frees the application programs from dealing with complex conditions on the constructed data. Another pattern, called *locally defined global identifier*, has been used to provide high-level interfaces to libraries dealing with complex data, like programming of dynamic web pages or graphical user interfaces (GUIs, see below). This pattern exploits the fact that functional logic data structures can contain logic variables which are globally unique when they are introduced. This is helpful to create local structures with globally unique identifiers and leads to improved abstractions in application programs. Further design patterns and programming techniques are discussed in [11,12].

The combination of functional and logic language features are exploited in [43] for the high-level programming of GUIs. The hierarchical structure of a GUI (e.g., rows, columns, or matrices of primitive and combined widgets) is represented as a data term. This term contains call-back functions as event handlers, i.e., the use of functions as first-class objects is natural in this application. Since event handlers defined for one widget should usually influence the appearance and contents of other widgets (e.g., if a slider is moved, values shown in other widgets should change), GUIs have also a logical structure that is different from its hierarchical structure. To specify this logical structure, logic variables in data structures are handy, since a logic variable can specify relationships between

Fig. 2. A simple counter GUI

different parts of a data term. As a concrete example, consider the simple counter GUI shown in Figure 2. Using a library designed with these ideas, one can specify this GUI by the following data term:

```
Col [Entry [WRef val, Text "0", Background "yellow"],
     Row [Button (updateValue incrText val) [Text "Increment"],
          Button (setValue val "0")         [Text "Reset"],
          Button exitGUI                    [Text "Stop"]]]
   where val free
```

The hierarchical structure of the GUI (a column with two rows) is directly reflected in the tree structure of this term. The first argument of each `Button` is the corresponding event handler. For instance, the invocation of `exitGUI` terminates the GUI, and the invocation of `setValue` assigns a new value (second argument) to the referenced widget (first argument). For this purpose, the logic variable *val* is used. Since the attribute `WRef` of the `Entry` widget defines its origin and it is used in various event handlers, it appropriately describes the logical structure of the GUI, i.e., the dependencies between different widgets. Note that other (more low level) GUI libraries or languages (e.g., Tcl/Tk) use strings or numbers as widget references which is potentially more error prone.

Similar ideas are applied in [44] to provide a high-level programming interface for web applications (dynamic web pages). There, HTML terms are represented as data structures containing event handlers associated to submit buttons and logic variables referring to user inputs in web pages that are passed to event handlers. These high-level APIs have been used in various applications, e.g., to implement web-based learning systems [52], constructing web-based interfaces for arbitrary applications [49] (there, the effectiveness of the multi-paradigm declarative programming style is demonstrated by a SuDoku solver with a web-based interface where the complete program consists of 20 lines of code), graphical programming environments [48,54], and documentation tools [46]. Furthermore, there are proposals to use multi-paradigm languages for high-level distributed programming [42,85], programming of embedded systems [50,51], object-oriented programming [53,82], or declarative APIs to databases [32,47].

5 Conclusions

In this paper we surveyed the main ideas of multi-paradigm declarative languages, their foundations, and some practical aspects of such languages for application programming. As a concrete example, we used the multi-paradigm

declarative language Curry. Curry amalgamates functional, logic, constraint, and concurrent programming features, it is based on strong foundations (e.g., soundness and completeness and optimal evaluation on inductively sequential programs) and it has been also used to develop larger applications. For the latter, Curry also offers useful features, like modules, strong typing, polymorphism, and declarative I/O, that are not described in this paper since they are not specific to multi-paradigm declarative programming (see [58] for such features).

We conclude with a summary of the advantages of combining different declarative paradigms in a single language. Although functions can be considered as predicates (thus, logic programming is sometimes considered as more general than functional programming), functional notation should not be used only as syntactic sugar: we have seen that the properties of functions (i.e., functional dependencies between input and output arguments) can be exploited to construct more efficient evaluation strategies without loosing generality. For instance, needed narrowing ensures soundness and completeness in the sense of logic programming and it is also optimal, whereas similar results are not available for pure logic programs. As a consequence, functional logic languages combine the flexibility of logic programming with the efficiency of functional programming. This leads to a more declarative style of programming without loosing efficiency. For instance, most functional logic languages do not have a Prolog-like "cut" operator since functions can be interpreted as a declarative replacement for it (see also [24,65]). Moreover, searching for solutions with a demand-driven evaluation strategy results in a demand-driven search strategy that can considerably reduce the search space. Finally, narrowing can be appropriately combined with constraint solving and residuation. The latter provides for a declarative integration of external operations and concurrent programming techniques.

Acknowledgments

I am grateful to thank Sergio Antoy, Bernd Braßel, Sebastian Fischer and Germán Vidal for their comments related to this survey.

References

1. Aït-Kaci, H.: An Overview of LIFE. In: Schmidt, J.W., Stogny, A.A. (eds.) Next Generation Information System Technology. LNCS, vol. 504, pp. 42–58. Springer, Heidelberg (1991)
2. Aït-Kaci, H., Lincoln, P., Nasr, R.: Le Fun: Logic, equations, and Functions. In: Proc. 4th IEEE Internat. Symposium on Logic Programming, pp. 17–23. IEEE Computer Society Press, San Francisco (1987)
3. Antoy, S.: Non-Determinism and Lazy Evaluation in Logic Programming. In: Proc. Int. Workshop on Logic Program Synthesis and Transformation (LOPSTR'91), Workshops in Computing, pp. 318–331. Springer, Heidelberg (1991)
4. Antoy, S.: Definitional Trees. In: Kirchner, H., Levi, G. (eds.) Algebraic and Logic Programming. LNCS, vol. 632, pp. 143–157. Springer, Heidelberg (1992)

5. Antoy, S.: Optimal Non-Deterministic Functional Logic Computations. In: Hanus, M., Heering, J., Meinke, K. (eds.) ALP 1997 and HOA 1997. LNCS, vol. 1298, pp. 16–30. Springer, Heidelberg (1997)
6. Antoy, S.: Constructor-based Conditional Narrowing. In: Proc. of the 3rd International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP 2001), pp. 199–206. ACM Press, New York (2001)
7. Antoy, S.: Evaluation Strategies for Functional Logic Programming. Journal of Symbolic Computation 40(1), 875–903 (2005)
8. Antoy, S., Echahed, R., Hanus, M.: Parallel Evaluation Strategies for Functional Logic Languages. In: Proc. of the Fourteenth International Conference on Logic Programming (ICLP'97), pp. 138–152. MIT Press, Cambridge (1997)
9. Antoy, S., Echahed, R., Hanus, M.: A Needed Narrowing Strategy. Journal of the ACM 47(4), 776–822 (2000)
10. Antoy, S., Hanus, M.: Compiling Multi-Paradigm Declarative Programs into Prolog. In: Kirchner, H. (ed.) Frontiers of Combining Systems. LNCS, vol. 1794, pp. 171–185. Springer, Heidelberg (2000)
11. Antoy, S., Hanus, M.: Functional Logic Design Patterns. In: Hu, Z., Rodríguez-Artalejo, M. (eds.) FLOPS 2002. LNCS, vol. 2441, pp. 67–87. Springer, Heidelberg (2002)
12. Antoy, S., Hanus, M.: Concurrent Distinct Choices. Journal of Functional Programming 14(6), 657–668 (2004)
13. Antoy, S., Hanus, M.: Declarative Programming with Function Patterns. In: Hill, P.M. (ed.) LOPSTR 2005. LNCS, vol. 3901, pp. 6–22. Springer, Heidelberg (2006)
14. Antoy, S., Hanus, M., Liu, J., Tolmach, A.: A Virtual Machine for Functional Logic Computations. In: Grelck, C., Huch, F., Michaelson, G.J., Trinder, P. (eds.) IFL 2004. LNCS, vol. 3474, pp. 108–125. Springer, Heidelberg (2005)
15. Antoy, S., Hanus, M., Massey, B., Steiner, F.: An Implementation of Narrowing Strategies. In: Proc. of the 3rd International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP 2001), pp. 207–217. ACM Press, New York (2001)
16. Antoy, S., Tolmach, A.: Typed Higher-Order Narrowing without Higher-Order Strategies. In: Middeldorp, A. (ed.) FLOPS 1999. LNCS, vol. 1722, pp. 335–352. Springer, Heidelberg (1999)
17. Arenas-Sánchez, P., Rodríguez-Artalejo, M.: A Semantic Framework for Functional Logic Programming with Algebraic Polymorphic Types. In: Bidoit, M., Dauchet, M. (eds.) CAAP 1997, FASE 1997, and TAPSOFT 1997. LNCS, vol. 1214, pp. 453–464. Springer, Heidelberg (1997)
18. Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge University Press, Cambridge (1998)
19. Berghammer, R., Fischer, S.: Implementing Relational Specifications in a Constraint Functional Logic Language. Electronic Notes in Theoretical Computer Science 177, 169–183 (2007)
20. Bird, R.S., Wadler, P.: Introduction to Functional Programming. Prentice-Hall, Englewood Cliffs (1988)
21. Bonnier, S., Maluszynski, J.: Towards a Clean Amalgamation of Logic Programs with External Procedures. In: Proc. 5th Conference on Logic Programming & 5th Symposium on Logic Programming (Seattle), pp. 311–326. MIT Press, Cambridge (1988)
22. Braßel, B., Hanus, M., Huch, F.: Encapsulating Non-Determinism in Functional Logic Computations. Journal of Functional and Logic Programming 2004(6) (2004)

23. Braßel, B., Huch, F.: Translating Curry to Haskell. In: Proc. of the ACM SIGPLAN 2005 Workshop on Curry and Functional Logic Programming (WCFLP 2005), pp. 60–65. ACM Press, New York (2005)
24. Caballero, R., García-Ruiz, Y.: Implementing Dynamic-Cut in TOY. Electronic Notes in Theoretical Computer Science 177, 153–168 (2007)
25. Casas, A., Cabeza, D., Hermenegildo, M.V.: A Syntactic Approach to Combining Functional Notation, Lazy Evaluation, and Higher-Order in LP Systems. In: Hagiya, M., Wadler, P. (eds.) FLOPS 2006. LNCS, vol. 3945, pp. 146–162. Springer, Heidelberg (2006)
26. Cheong, P.H., Fribourg, L.: Implementation of Narrowing: The Prolog-Based Approach. In: Apt, K.R., de Bakker, J.W., Rutten, J.J.M.M. (eds.) Logic programming languages: constraints, functions, and objects, pp. 1–20. MIT Press, Cambridge (1993)
27. Cleva, J.M., Leach, J., López-Fraguas, F.J.: A logic programming approach to the verification of functional-logic programs. In: Proceedings of the 6th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, pp. 9–19. ACM Press, New York (2004)
28. del Vado Virseda, R.: A Demand-Driven Narrowing Calculus with Overlapping Definitional Trees. In: Proceedings of the 8th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'03), pp. 253–263. ACM Press, New York (2003)
29. Dershowitz, N., Jouannaud, J.-P.: Rewrite Systems. In: van Leeuwen, J. (ed.) Handbook of Theoretical Computer Science, vol. B, pp. 243–320. Elsevier, Amsterdam (1990)
30. Fernández, A.J., Hortalá-González, M.T., Sáenz-Pérez, F.: Solving Combinatorial Problems with a Constraint Functional Logic Language. In: Dahl, V., Wadler, P. (eds.) PADL 2003. LNCS, vol. 2562, pp. 320–338. Springer, Heidelberg (2002)
31. Fernández, A.J., Hortalá-González, M.T., Sáenz-Pérez, F., del Vado-Vírseda, R.: Constraint Functional Logic Programming over Finite Domains. Theory and Practice of Logic Programming (to appear, 2007)
32. Fischer, S.: A Functional Logic Database Library. In: Proc. of the ACM SIGPLAN 2005 Workshop on Curry and Functional Logic Programming (WCFLP 2005), pp. 54–59. ACM Press, New York (2005)
33. de la Banda, M.J.G., Demoen, B., Marriott, K., Stuckey, P.J.: To the Gates of HAL: A HAL Tutorial. In: Hu, Z., Rodríguez-Artalejo, M. (eds.) FLOPS 2002. LNCS, vol. 2441, pp. 47–66. Springer, Heidelberg (2002)
34. Giovannetti, E., Levi, G., Moiso, C., Palamidessi, C.: Kernel LEAF: A Logic plus Functional Language. Journal of Computer and System Sciences 42(2), 139–185 (1991)
35. Gonzáles-Moreno, J.C., Hortalá-González, M.T., Rodríguez-Artalejo, M.: Polymorphic Types in Functional Logic Programming. Journal of Functional and Logic Programming, 2001(1) (2001)
36. González-Moreno, J.C., Hortalá-González, M.T., López-Fraguas, F.J., Rodríguez-Artalejo, M.: An approach to declarative programming based on a rewriting logic. Journal of Logic Programming 40, 47–87 (1999)
37. González-Moreno, J.C., Hortalá-González, M.T., Rodríguez-Artalejo, M.: A Higher Order Rewriting Logic for Functional Logic Programming. In: Proc. of the Fourteenth International Conference on Logic Programming (ICLP'97), pp. 153–167. MIT Press, Cambridge (1997)
38. Hanus, M.: The Integration of Functions into Logic Programming: From Theory to Practice. Journal of Logic Programming 19&20, 583–628 (1994)

39. Hanus, M.: Analysis of Residuating Logic Programs. Journal of Logic Programming 24(3), 161–199 (1995)
40. Hanus, M.: Efficient Translation of Lazy Functional Logic Programs into Prolog. In: Proietti, M. (ed.) LOPSTR 1995. LNCS, vol. 1048, pp. 252–266. Springer, Heidelberg (1996)
41. Hanus, M.: A Unified Computation Model for Functional and Logic Programming. In: Proc. of the 24th ACM Symposium on Principles of Programming Languages (Paris), pp. 80–93 (1997)
42. Hanus, M.: Distributed Programming in a Multi-Paradigm Declarative Language. In: Nadathur, G. (ed.) PPDP 1999. LNCS, vol. 1702, pp. 376–395. Springer, Heidelberg (1999)
43. Hanus, M.: A Functional Logic Programming Approach to Graphical User Interfaces. In: Pontelli, E., Santos Costa, V. (eds.) PADL 2000. LNCS, vol. 1753, pp. 47–62. Springer, Heidelberg (2000)
44. Hanus, M.: High-Level Server Side Web Scripting in Curry. In: Ramakrishnan, I.V. (ed.) PADL 2001. LNCS, vol. 1990, pp. 76–92. Springer, Heidelberg (2001)
45. Hanus, M.: Reduction Strategies for Declarative Programming. In: Gramlich, B., Lucas, S. (eds.) Electronic Notes in Theoretical Computer Science, vol. 57, Elsevier Science Publishers, Amsterdam (2001)
46. Hanus, M.: CurryDoc: A Documentation Tool for Declarative Programs. In: Proc. 11th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2002) Research Report UDMI/18/2002/RR, University of Udine, pp. 225–228 (2002)
47. Hanus, M.: Dynamic Predicates in Functional Logic Programs. Journal of Functional and Logic Programming, 2004(5) (2004)
48. Hanus, M.: A Generic Analysis Environment for Declarative Programs. In: Proc. of the ACM SIGPLAN 2005 Workshop on Curry and Functional Logic Programming (WCFLP 2005), pp. 43–48. ACM Press, New York (2005)
49. Hanus, M.: Type-Oriented Construction of Web User Interfaces. In: Proceedings of the 8th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'06), pp. 27–38. ACM Press, New York (2006)
50. Hanus, M., Höppner, K.: Programming Autonomous Robots in Curry. Electronic Notes in Theoretical Computer Science, 76 (2002)
51. Hanus, M., Höppner, K., Huch, F.: Towards Translating Embedded Curry to C. Electronic Notes in Theoretical Computer Science, 86(3) (2003)
52. Hanus, M., Huch, F.: An Open System to Support Web-based Learning. In: Proc. 12th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2003) Technical Report DSIC-II/13/03, Universidad Politécnica de Valencia, pp. 269–282 (2003)
53. Hanus, M., Huch, F., Niederau, P.: An Object-Oriented Extension of the Declarative Multi-Paradigm Language Curry. In: Mohnen, M., Koopman, P. (eds.) IFL 2000. LNCS, vol. 2011, pp. 89–106. Springer, Heidelberg (2001)
54. Hanus, M., Koj, J.: An Integrated Development Environment for Declarative Multi-Paradigm Programming. In: Proc. of the International Workshop on Logic Programming Environments (WLPE'01), Paphos (Cyprus), Also available from the Computing Research Repository (CoRR) pp. 1–14 (2001), at `http://arXiv.org/abs/cs.PL/0111039`
55. Hanus, M., Prehofer, C.: Higher-Order Narrowing with Definitional Trees. Journal of Functional Programming 9(1), 33–75 (1999)
56. Hanus, M., Sadre, R.: An Abstract Machine for Curry and its Concurrent Implementation in Java. Journal of Functional and Logic Programming, 1999(6) (1999)

57. Hanus, M., Steiner, F.: Controlling Search in Declarative Programs. In: Palamidessi, C., Meinke, K., Glaser, H. (eds.) ALP 1998 and PLILP 1998. LNCS, vol. 1490, pp. 374–390. Springer, Heidelberg (1998)
58. Hanus, M. (ed.): Curry: An Integrated Functional Logic Language (Vers. 0.8.2) (2006), Available at `http://www.informatik.uni-kiel.de/~curry`
59. Hughes, J.: Why Functional Programming Matters. In: Turner, D.A. (ed.) Research Topics in Functional Programming, pp. 17–42. Addison-Wesley, Reading (1990)
60. Hussmann, H.: Nondeterministic Algebraic Specifications and Nonconfluent Term Rewriting. Journal of Logic Programming 12, 237–255 (1992)
61. Jaffar, J., Lassez, J.-L.: Constraint Logic Programming. In: Proc. of the 14th ACM Symposium on Principles of Programming Languages, Munich, pp. 111–119 (1987)
62. Jiménez-Martin, J.A., Marino-Carballo, J., Moreno-Navarro, J.J.: Efficient Compilation of Lazy Narrowing into Prolog. In: Proc. Int. Workshop on Logic Program Synthesis and Transformation (LOPSTR'92) Springer Workshops in Computing Series, pp. 253–270 (1992)
63. Lloyd, J.: Programming in an Integrated Functional and Logic Language. Journal of Functional and Logic Programming, 3, 1–49 (1999)
64. Loogen, R., López Fraguas, F., Rodríguez Artalejo, M.: A Demand Driven Computation Strategy for Lazy Narrowing. In: Penjam, J., Bruynooghe, M. (eds.) PLILP 1993. LNCS, vol. 714, pp. 184–200. Springer, Heidelberg (1993)
65. Loogen, R., Winkler, S.: Dynamic Detection of Determinism in Functional Logic Languages. Theoretical Computer Science 142, 59–87 (1995)
66. López-Fraguas, F., Sánchez-Hernández, J.: TOY: A Multiparadigm Declarative System. In: Narendran, P., Rusinowitch, M. (eds.) RTA 1999. LNCS, vol. 1631, pp. 244–247. Springer, Heidelberg (1999)
67. López-Fraguas, F.J., Rodríguez-Artalejo, M., del Vado Virseda, R.: A lazy narrowing calculus for declarative constraint programming. In: Proceedings of the 6th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, pp. 43–54. ACM Press, New York (2004)
68. López-Fraguas, F.J., Sánchez-Hernández, J.: A Proof Theoretic Approach to Failure in Functional Logic Programming. Theory and Practice of Logic Programming 4(1), 41–74 (2004)
69. Lux, W.: Implementing Encapsulated Search for a Lazy Functional Logic Language. In: Middeldorp, A. (ed.) FLOPS 1999. LNCS, vol. 1722, pp. 100–113. Springer, Heidelberg (1999)
70. Lux, W.: Adding Linear Constraints over Real Numbers to Curry. In: Kuchen, H., Ueda, K. (eds.) FLOPS 2001. LNCS, vol. 2024, pp. 185–200. Springer, Heidelberg (2001)
71. Naish, L.: Adding equations to NU-Prolog. In: Małuszyński, J., Wirsing, M. (eds.) PLILP 1991. LNCS, vol. 528, pp. 15–26. Springer, Heidelberg (1991)
72. O'Donnell, M.J.: Computing in Systems Described by Equations. LNCS, vol. 58. Springer, Heidelberg (1977)
73. O'Donnell, M.J.: Equational Logic as a Programming Language. MIT Press, Cambridge (1985)
74. Padawitz, P.: Computing in Horn Clause Theories. EATCS Monographs on Theoretical Computer Science, vol. 16. Springer, Heidelberg (1988)
75. Peyton Jones, S. (ed.): Haskell 98 Language and Libraries—The Revised Report. Cambridge University Press, Cambridge (2003)
76. Reynolds, J.C.: Definitional Interpreters for Higher-Order Programming Languages. In: Proceedings of the ACM Annual Conference, pp. 717–740. ACM Press, New York (1972)

77. Rodríguez-Artalejo, M.: Functional and Constraint Logic Programming. In: Comon, H., Marché, C., Treinen, R. (eds.) Constraints in Computational Logics. Theory and Applications. LNCS, vol. 2002, pp. 202–270. Springer, Heidelberg (2001)
78. Saraswat, V.A.: Concurrent Constraint Programming. MIT Press, Cambridge (1993)
79. Schulte, C., Smolka, G.: Encapsulated Search for Higher-Order Concurrent Constraint Programming. In: Proc. of the 1994 International Logic Programming Symposium, pp. 505–520. MIT Press, Cambridge (1994)
80. Sekar, R.C., Ramakrishnan, I.V.: Programming in Equational Logic: Beyond Strong Sequentiality. Information and Computation 104(1), 78–109 (1993)
81. Slagle, J.R.: Automated Theorem-Proving for Theories with Simplifiers, Commutativity, and Associativity. Journal of the ACM 21(4), 622–642 (1974)
82. Smolka, G.: The Oz Programming Model. In: van Leeuwen, J. (ed.) Computer Science Today. LNCS, vol. 1000, pp. 324–343. Springer, Heidelberg (1995)
83. Somogyi, Z., Henderson, F., Conway, T.: The execution algorithm of Mercury, an efficient purely declarative logic programming language. Journal of Logic Programming 29(1-3), 17–64 (1996)
84. Van Roy, P., Haridi, S.: Concepts, Techniques, and Models of Computer Programming. MIT Press, Cambridge (2004)
85. Van Roy, P., Haridi, S., Brand, P., Smolka, G., Mehl, M., Scheidhauer, R.: Mobile Objects in Distributed Oz. ACM Transactions on Programming Languages and Systems 19(5), 804–851 (1997)
86. Wadler, P.: How to Declare an Imperative. ACM Computing Surveys 29(3), 240–263 (1997)
87. Warren, D.H.D.: Higher-order extensions to PROLOG: are they needed. Machine Intelligence 10, 441–454 (1982)

Logic Programming for Knowledge Representation

Mirosław Truszczyński

Department of Computer Science, University of Kentucky,
Lexington, KY 40506-0046, USA
mirek@cs.uky.edu

Abstract. This note provides background information and references to the tutorial on recent research developments in logic programming inspired by needs of knowledge representation.

1 Introduction

McCarthy and Hayes [46] wrote: "[...] intelligence has two parts, which we shall call the epistemological and the heuristic. The epistemological part is the representation of the world in such a form that the solution of problems follows from the facts expressed in the representation. The heuristic part is the mechanism that on the basis of the information solves the problem and decides what to do." The epistemological part is the concern of knowledge representation. The heuristic part is typically addressed by search.

While not stated explicitly in McCarthy and Hayes' definition, the knowledge representation and the search are closely intertwined. Modeling features of the language affect the design of search methods. Conversely, the availability of fast search techniques for particular computational tasks, for instance, proof finding or model computation, in the case of particular classes of theories, such as Horn theories or propositional theories in CNF, influences the design of modeling languages. Effective computational knowledge representation systems require that the two are integrated.

Logic programming has long been regarded as a prime candidate for a practical instantiation of computational knowledge representation. First, logic program clauses align well with natural language constructs humans use to specify constraints [31,32]. Next, logic programs (Horn logic programs, to be precise) are Turing complete [1]. Finally, there are well understood automated proof techniques for reasoning with logic programs [50,32]. These considerations led to the design of Prolog [6], still a dominant computational knowledge representation language.

However, the presence of negation in the bodies of logic program rules, while convenient from the modeling standpoint, posed a challenge. The semantics of logic programs with negation was not clear and resolving that issue required a major research effort. The completion semantics [5] was the first attempt to address the problem. The answer-set semantics [27,28] provided a definitive solution within the class of 2-valued semantics.

V. Dahl and I. Niemelä (Eds.): ICLP 2007, LNCS 4670, pp. 76–88, 2007.

For some time after its introduction, the answer-set semantics was a source of confusion. It was unclear how to use it in practice when modeling application domains, and how to reconcile it with the traditional proof-theory approach to logic programming. The *answer-set programming* paradigm [43,48] offered an alternative to proof-based logic programming and shifted the focus from proof-finding to model-finding. Under the answer-set programming paradigm, a problem is modeled as a logic program so that *answer sets* of the program expanded with an encoding of a particular instance of the problem (not substitutions associated with proofs) correspond to solutions to the problem for that instance.

With the understanding of the meaning of programs with negation came additional evidence of the applicability of logic programs in knowledge representation. According to [26], the goal of knowledge representation is "to design and study languages to capture knowledge about environments, their entities and their behaviors." Such languages must be able to handle modeling challenges posed by the qualification and frame problems, defaults, conditionals and normative statements, (inductive) definitions, and by the need for the elaboration tolerance. Research showed that answer-set programming, provides means that adequately address these problems [4,25,26,3]. While answer-set programming comes with the penalty of syntactic restrictions (no function symbols) and so, the limited expressive power (the class NP-search for normal logic programs, and Σ_2^P-search for disjunctive programs), it has major advantages. First, arguably the task of modeling gets much simpler than in proof-based logic programming — answer-set programs are truly declarative. Second, it becomes possible to take advantage of fast search techniques developed in the area of propositional satisfiability.

In this note I will discuss three recent research directions in logic programming inspired by knowledge representation needs. First, knowledge bases must be constructed in a modular fashion. This brings up the question of equivalence of answer-set programs, as well as the need for methods to decompose programs so that answer-sets of the program can be recovered from answer sets of its components. Second, there has been a need for extending the syntax of answer-set programs with means to model numeric constraints. Such constraints are ubiquitous and, in particular, are common in knowledge representation applications. Finally, there are logics other than logic programs with the answer-set semantics that also give rise to knowledge representation systems based on the principle that models of theories describe problem solutions. One of the most promising such approaches, the ID-logic [8], combines first-order logic (which is used to model constraints) with logic programs under the well-founded semantics [58], a 3-valued approximation to the answer-set semantics (which are used to represent definitions).

Theoretical advances in answer-set programming would remain just that, if they were not followed by practical applications. These applications require working and effective answer-set programming software. There has been much work in that area. I will conclude this paper with brief comments on and references to some of the state-of-the-art implementations.

2 Modularity

A basic design principle in knowledge representation is the principle of *modularity*. It stipulates that knowledge bases be composed of *modules*, each representing a fragment of an application domain being modeled. The principle of modularity gave rise to research problems that have generated much interest among logic programming and answer-set programming researchers. Two of them that I will discuss are:

1. To characterize cases when two knowledge base modules are *equivalent for substitution*, and
2. To improve the efficiency of processing algorithms by taking advantage of the modular structure of the knowledge base.

Informally, two modules are equivalent for substitution if replacing one with the other does not affect the meaning of the knowledge base. When optimizing the knowledge base for conciseness, efficiency of processing, robustness to change, or other appropriate measure of quality, one approach is to optimize each module separately. For that approach to be safe, though, the optimized module should be equivalent for substitution to the original one. Otherwise, optimization might have unwanted side effects. In a related way, the concept of equivalence for substitution has applications in knowledge-base query optimization. Thus, the notion of equivalence for substitution is an important one.

When a knowledge base is represented by a logic program under the answer-set semantics, one can formalize the concept of the equivalence for substitution, or *strong equivalence*, the term more commonly used in answer-set programming, as follows: two programs P and Q are *strongly equivalent* if for every program R, the programs $P \cup R$ and $Q \cup R$ have the same answer sets. Indeed, if P and Q have this property, one can replace P with Q or Q with P within any larger program, and the answer sets (a formal description of the "meaning") will not change.

The concept of strong equivalence was introduced in [33]. That paper also presented a complete characterization of strong equivalence in terms of the equivalence in the logic *here-and-there* [29]. We note in passing that while a necessary condition, having the same answer sets is not sufficient for two programs to be strongly equivalent. Thus, a most direct attempt at extending the concept of equivalence from the classical logic to answer-set programming does not work. [36,56] presented simple characterizations of strong equivalence that do not make explicit references to the logic *here-and-there*. [55] cast the concept in terms of the equivalence in the modal logic S4F. Several generalizations and variations of the notion were proposed and studied in [15,17,19,16,34,23]. [54] extended the problem of strong equivalence to an abstract algebraic setting of the approximation theory of operators on lattices [9].

Modularity can also be exploited in processing programs. It is well known that the problem of the existence of an answer set of a logic program is NP-complete for normal propositional logic programs [45] and Σ_2^P-complete for the

disjunctive ones [18]. Modular structure of the program can have dramatic effect on the complexity of this decision problem and the associated search task.

This has been known for quite some time in the case when the dependencies among modules are acyclic. [2] proposed the class of stratified programs, in which dependencies between modules are acyclic, and dependencies within modules are subject to certain restrictions. Each stratified normal logic program has exactly one answer set. Moreover, it can be computed efficiently. The results on stratification have been generalized in the form of a splitting theorem, to the case when the dependencies between modules are still acyclic but the structure of each module is not restricted anymore [35]. When a program can be "split", a form of divide-and-conquer approach can significantly speed up the process of computing answer sets. It is worth noting that, as strong equivalence, splitting also has an algebraic description within the approximation theory [60,59].

Recently, researchers focused on the general case, when dependencies among modules are not acyclic [21,30]. By controlling the way, in which modules interact, [30] managed to characterize answer sets of the overall program in terms of answer sets of individual modules and outlined several possible applications for this general result.

3 Programs with Constraints

Numeric constraints are common. Modeling them in the basic language of first-order logic is, however, a tedious task. It requires auxiliary atoms and leads to large programs with no transparent meaning. The problem has been long recognized in the area of database systems, where queries often concern numeric properties of sets of records. To make the process of formulating such queries easier, database query languages are equipped with syntax that provides explicit means to express numeric constraints, referred to in the field of databases as *aggregates*. The same holds true of constraint programming languages.

Applications of answer-set programming in knowledge representation brought up the same problem and motivated extensions of the basic language of program rules with syntax to model constraints directly [51,7,10,20,22,49,52]. These approaches agree on many classes of programs. However, they differ in intuitions, as well as in some technical aspects. To gain a better understanding of extensions of answer-set programming with constraints, and to offer a more principled approach to the semantics of such extensions, researchers proposed and studied answer-set programming formalisms based on *abstract constraint atoms* [42,44,41,53,38]. This approach lead to a theory of programs with constraints based on the concept of a *computation* driven by a generalization of the one-step provability operator [57]. In the case of programs with *monotone* and *convex* constraints the parallels with the normal logic programming are very strong [41]. Recent work extended these parallels also to the case of arbitrary constraints [38]. We will now present, following closely [44] and [40], some of the basic aspects of the theory of programs with abstract constraints concentrating on the case when these constraints are monotone.

We consider a language determined by a fixed countable set At of *propositional variables*. An *abstract constraint atom* is a syntactic expression $A = (C, X)$, where $X \subseteq At$ is a *finite* set of propositional variables, called the *domain* of A, and $C \subseteq \mathcal{P}(At)$ is the set of *satisfiers* of A. We will write A_{dom} for X and A_{sat} for C.

A propositional interpretation $M \subseteq At$ *satisfies* an abstract constraint atom A, denoted $M \models A$, if $M \cap A_{dom} \in A_{sat}$, that is, if the set of elements in the domain of A that are true in M is a satisfier of A.

A *constraint program* is a set of *constraint* rules, that is, expressions of the form

$$A \leftarrow A_1, \ldots, A_k, \mathbf{not}(A_{k+1}), \ldots, \mathbf{not}(A_m) \tag{1}$$

where A, $A_1, \ldots, A_m$ are constraints and **not** denotes *default negation*. The constraint A is the *head* and the set $\{A_1, \ldots, A_k, \mathbf{not}(A_{k+1}), \ldots, \mathbf{not}(A_m)\}$ is the *body* of r. The concept of satisfiability described above extends in the standard way to constraint rules and programs.

We denote by $At(P)$ the set of atoms in the domains of constraints in a constraint program P. We denote by $hset(P)$, the *headset* of P, that is, the union of the domains of the heads of all rules in P.

We will now list several basic definitions and properties of constraint programs mirroring those of normal ones, and culminating with a generalization of the concept of the answer-set.

***M*-applicable rules.** Let $M \subseteq At$ be an interpretation. A rule (1) is *M-applicable* if M satisfies every literal in the body of r. We write $P(M)$ for the set of all M-applicable rules in P.

Supported models. An interpretation M is a *supported* model of a constraint program P if M is a model of P and $M \subseteq hset(P(M))$.

Nondeterministic one-step provability. An interpretation $M' \subseteq At$ is *nondeterministically one-step provable* from an interpretation $M \subseteq At$ by means of a constraint program P, if $M' \subseteq hset(P(M))$ and for every head A of a rule in $P(M)$, $M' \models A$. Given a constraint program P, the *nondeterministic one-step provability operator* T_P^{nd} is an operator on $\mathcal{P}(At)$ such that for every $M \subseteq At$, $T_P^{nd}(M)$ consists of all sets that are nondeterministically one-step provable from M by means of P.

Monotone constraints and monotone-constraint programs. A constraint A is *monotone* if for every $M, M' \subseteq At$, $M \subseteq M'$ and $M \models A$ together imply that $M' \models A$. A *monotone-constraint program* is a program built of monotone constraints.

Horn constraint programs. A rule (1) is *Horn* if constraints A, $A_1, \ldots, A_k$ are monotone, and if $k = m$ (no occurrences of default negation). A constraint program is *Horn* if every constraint rule in the program is Horn.

Bottom-up computations for Horn programs. Let P be a Horn constraint program. A *P-computation* is a sequence $\langle X_k \rangle_{k=0}^{\infty}$ such that $X_0 = \emptyset$ and for every k,

$$X_k \subseteq X_{k+1}, \quad \text{and} \quad X_{k+1} \in T_P^{nd}(X_k).$$

The *result* of a P-computation $t = \langle X_k \rangle$ is the set $\bigcup_k X_k$. We denote it by R_t.

We note that if P is a Horn constraint program and t is a P-computation, R_t is a supported model of P.

Derivable models. A set M of atoms is a *derivable model* of a Horn constraint program P if there exists a P-computation t such that $M = R_t$. If a Horn constraint program has a model, one can show that it has computations and, consequently, derivable models. By our comment above, derivable models of a Horn constraint program P are supported models and so, in particular, models of P.

The reduct. Let P be a monotone-constraint program and M a subset of $At(P)$. The *reduct* of P with respect to M, P^M, is the Horn constraint program obtained from P by: (1) removing from P all rules whose body contains a literal $\mathbf{not}(B)$ such that $M \models B$; and (2) removing literals $\mathbf{not}(B)$ for the bodies of the remaining rules.

Answer sets. Let P be a program. A set of atoms M is an *answer set* of P if M is a derivable model of P^M. Since P^M is a Horn constraint program, the definition is sound.

The definitions of the reduct and of answer sets follow and generalize the corresponding definitions proposed for normal logic programs, as in the setting of Horn constraint programs, derivable models play the role of a least model.

As in normal logic programming, answer sets of monotone-constraint programs are supported models and, consequently, models. As shown in [51], monotone-constraint programs generalize programs with weight atoms [51]. Some other results one can prove for monotone-constraint programs are extensions of the characterizations of strong and uniform equivalence of programs, of the concept of the program completion [5], and of loop formulas [37]. The latter two concepts proved useful in designing a solver *pbmodels* [39] for computing answer sets of programs with weight constraints in the syntax of *smodels*. *Pbmodels* placed second in two events of the 1st Answer-Set Programming Contest [24].

We concentrated here on monotone-constraint programs. Programs built of more general classes of constraints offer additional challenges. The class of *convex* constraint atoms (A is convex if for every M, M' and M'' such that $M' \models A$, $M'' \models A$ and $M' \subseteq M \subseteq M''$, $M \models A$, as well) is most closely related to the class of monotone constraints and the basic approach described above extends. For arbitrary constraint atoms there is no simple generalization due to non-monotone behavior of such constraints. [38] developed a principled approach to the problem of the answer-set semantics for programs with arbitrary constraints. However, the general approach of [38] resulted in three candidate semantics, none of which has emerged as the definitive one for programs with arbitrary constraints.

4 Model Expansion and ID-Logic

Several fundamental knowledge representation and reasoning problems can be stated formally as search problems, and solved by general search methods. In

fact, this observation lies behind the answer-set programming paradigm. We will now introduce a simple formalism for modeling search problems and show how to integrate it with logic programming to produce an effective answer-set programming formalism. This approach stems from the work on the formalism called datalog with constraints [13,14] and research on the role of definitions in knowledge representation [8]. It has been actively studied and developed further in [11,47,12].

We start with basic terminology. A *signature* is a nonempty set σ of relation symbols, each with a positive integer arity. Let U be a fixed infinite countable set (the *universe*), and let σ be a signature. An *instance* of σ over U is a set I of finite relations over U, such that there is a one-to-one correspondence $r \leftrightarrow r^I$ between σ and I, and the corresponding relation symbols r and relations r^I are of the same arity.

We denote by $Inst_\sigma$ the set of all instances of σ. If $I \in Inst_\sigma$, we define the *domain* of I, $dom(I)$, to be the set of all those elements of U that appear in a tuple of a relation in I. Since all relations in I are finite, $dom(I)$ is finite, too. Let $\sigma' \subseteq \sigma$ be signatures. We say that $K \in Inst_\sigma$ *expands* $I \in Inst_{\sigma'}$ if $dom(I) = dom(K)$ and for every $r \in \sigma'$, $r^I = r^K$. We write $I = K_{|\sigma'}$ to denote that K expands I.

The formalism of *model expansion* (the *logic MX*) is based on the language $\mathcal{L}_\sigma$ of the first-order logic, determined by a signature σ (thus, we assume here no function symbols).

An instance $I \in Inst_\sigma$ determines a first-order logic interpretation $\langle dom(I), I\rangle$ of $\mathcal{L}_\sigma$. With some abuse of notation, for an instance $I \in Inst_\sigma$ and a sentence $\varphi \in \mathcal{L}_\sigma$, we write $I \models \varphi$ instead of $\langle dom(I), I\rangle \models \varphi$.

Let $\sigma' \subseteq \sigma$ be signatures and let $\varphi \in \mathcal{L}_\sigma$ be a sentence. Given an instance $I \in Inst_{\sigma'}$ we call an instance $K \in Inst_\sigma$ an *I-model* of φ if K expands I and $K \models \varphi$ (hence the term *model* expansion). The concept extends in a straightforward way to sets of sentences. We will refer to finite sets of sentences interpreted by I-models as *MX-theories*.

Given signatures $\sigma' \subseteq \sigma$, we can regard an MX-theory T from $\mathcal{L}_\sigma$ as an encoding of a search problem. This problem has $Inst_\sigma$ as the set of its instances and, for every instance $I \in Inst_\sigma$, I-models as solutions to the instance I.

One can show that the expressive power of MX-theories is the same as that of (normal) logic programs [13]. In some cases, though, the task of modeling the search problems as MX-theories is significantly more complicated than when answer-set programs are used. Modeling definitions is often particularly cumbersome. To address that issue, researchers proposed extensions of the formalism MX with logic programs which, under the *well-founded* semantics [58], are well suited to model definitions [8].

We will present one such extension, the ID-logic [8,12,47]. Let $\sigma' \subseteq \sigma$ be signatures. A theory T modeling a search problem in the formalism of the ID-logic consists of two parts. An MX-theory T' in $\mathcal{L}_\sigma$ forms one of the parts. A normal logic program T'', also in $\mathcal{L}_\sigma$ but with no relation symbol from σ' in the head of a rule, forms the other one. Let σ^{def} be the set of relation symbols in

the heads of the rules of the program T''. Intuitively, these are the symbols that need to be defined and that are defined in the ID-logic theory T by its component T''.

Given an instance $I \in Inst_{\sigma'}$, an instance $K \in Inst_{\sigma}$ is an *I-model* of T if K is an I-model of T' and $K_{|\sigma^{def}}$ is the *total* well-founded model of the program $T'' \cup K_{|\sigma \setminus \sigma^{def}}$[1]. Informally, to obtain an I-model of an ID-logic theory T, we guess the extensions of all relation symbols in σ (keeping the extensions of the relation symbols in σ' as they are specified by I and not introducing any new constants), and verify that the extensions of the relation symbols in σ^{def} are fully determined by the program T'' and the extensions of the relation symbols not in σ^{def} under the well-founded semantics.

The ID-logic (under the restriction of no function symbols in the language) has the same expressive power as (non-disjunctive) answer-set programming [14]. However, arguably, its semantics is simpler, as it relies on two intuitive concepts: constraints and definitions. Consequently, the ID-logic has a significant potential for knowledge representation applications. A practical demonstration of the modeling capabilities of ID-logic can be found in [11].

5 Tools

While theoretical challenges make answer-set programming an exciting research area, it is because of the existence of effective computational tools, effective enough to handle several classes of "industrial-grade" problems, that it is steadily gaining on importance. We will now briefly review some of these tools.

The general approach to processing answer-set programs consists of two steps: (1) *grounding*, that is, instantiating and simplifying an input program with variables into a propositional program, and (2) *solving*, that is, computing answer sets of the program resulting from step (1). The grounding step is designed so that all answer sets of the original program can be recovered in step (2).

The *lparse/smodels* software (`www.tcs.hut.fi/Software/smodels/`) constitutes the earliest attempt at making answer-set programming practical. It remains widely used. Arguably, *lparse* is the most widely used grounder by solver developers, and *smodels* remains one of the most competitive solvers.

The *dlv* system (`www.dbai.tuwien.ac.at/proj/dlv/`) was proposed soon after *lparse/smodels*. The *dlv* offers an integrated grounder and solver package capable of computing answer-sets of disjunctive programs with aggregates. With its front-ends for SQL, inheritance, planning, and abduction and diagnosis, it emerges as the most flexible answer-set programming system at present. In this context, it is important to note that *dlv* won the 1st Answer-Set Programming Contest in the modeling/grounding/solving category [24].

Among software addressing only one of the stages in the computation of answer sets, *gringo* (`gringo.sourceforge.net/`) is a recent newcomer in the area of grounders. Noteworthy solvers are *clasp* (`www.cs.uni-potsdam.de/`

[1] We abuse the notation here by viewing instances as sets of the corresponding ground atoms in the language extending $\mathcal{L}_\sigma$ with the elements of U as constants.

clasp/), *pbmodels* (www.cs.uky.edu/ai/pbmodels/), *cmodels* (www.cs.utexas.edu/users/tag/cmodels.html), as well as *gnt* (www.tcs.hut.fi/Software/gnt/). Each of these solvers performed very well in the 1st Answer-Set Programming Contest [24], with *clasp* winning in two events. Other notable solver is *assat*, which pioneered the idea of using loop formulas to reduce answer-set computation to propositional satisfiability.

Finally, we mention software for answer-set programming systems based on the ID-logic. The grounder *psgrnd* (www.cs.uky.edu/ai/aspps/) can process ID-logic theories, in which the first-order component is in the clausal form (possibly with weight atoms), and the logic program component is a Horn program. The solver *aspps* (www.cs.uky.edu/ai/aspps/) is designed to compute answer-sets of ground ID-logic theories satisfying these restrictions. Recently more general tools have been developed, including *MXG* (www.cs.sfu.ca/research/groups/mxp/mxg/), an integrated software package for grounding and computing models of ID-logic theories, and *GidL/MidL* package (www.cs.kuleuven.be/~dtai/krr/software/idp.html) for grounding (GidL) and model generation (MidL) of theories in the ID-logic.

6 Closing Comments

Answer-set programming, a variant of logic programming with the answer-set semantics, and formalisms such as ID-logic, which combine first-order logic with logic programming under the well-founded semantics, are well suited for knowledge representation applications. After about a decade since they have been proposed, they continue to generate theoretical research challenges. In the same time, thanks to the development of computational software, their practical importance is steadily growing.

Acknowledgments

The author acknowledges the support of NSF grant IIS-0325063 and KSEF grant 1036-RDE-008.

References

1. Andréka, H., Németi, I.: The generalized completeness of Horn predicate logic as a programming language. Acta Cybernetica 4(1), 3–10 (1978/79)
2. Apt, K., Blair, H.A., Walker, A.: Towards a theory of declarative knowledge. In: Foundations of deductive databases and logic programming, pp. 89–142. Morgan Kaufmann, San Francisco (1988)
3. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press, Cambridge (2003)
4. Baral, C., Gelfond, M.: Logic programming and knowledge representation. Journal of Logic Programming 19/20, 73–148 (1994)

5. Clark, K.L.: Negation as failure, Logic and data bases, pp. 293–322. Plenum Press, New York, London (1978)
6. Colmerauer, A., Kanoui, H., Pasero, R., Roussel, P.: Un systeme de communication homme-machine en francais, Tech. report, University of Marseille (1973)
7. Dell'Armi, T., Faber, W., Ielpa, G., Leone, N., Pfeifer, G.: Aggregate functions in disjunctive logic programming: semantics, complexity, and implementation in DLV. In: Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI 2003), pp. 847–852. Morgan Kaufmann, San Francisco (2003)
8. Denecker, M.: The well-founded semantics is the principle of inductive definition. In: Dix, J., Fariñas del Cerro, L., Furbach, U. (eds.) JELIA 1998. LNCS (LNAI), vol. 1489, pp. 1–16. Springer, Heidelberg (1998)
9. Denecker, M., Marek, V., Truszczyński, M.: Approximations, stable operators, well-founded fixpoints and applications in nonmonotonic reasoning. In: Logic-Based Artificial Intelligence, pp. 127–144. Kluwer Academic Publishers, Dordrecht (2000)
10. Denecker, M., Pelov, N., Bruynooghe, M.: Ultimate well-founded and stable semantics for logic programs with aggregates. In: Codognet, P. (ed.) ICLP 2001. LNCS, vol. 2237, pp. 212–226. Springer, Heidelberg (2001)
11. Denecker, M., Ternovska, E.: Inductive situation calculus. In: Proceedings of the 9th International Conference on Principles of Knowledge Representation and Reasoning (KR 2004), pp. 545–553. AAAI Press, Stanford (2004)
12. Denecker, M., Ternovska, E.: A logic for non-monotone inductive definitions, ACM Transactions on Computational Logic (to appear, 2008)
13. East, D., Truszczyński, M.: Datalog with constraints. In: Proceedings of the 17th National Conference on Artificial Intelligence (AAAI 2000), pp. 163–168. AAAI Press, Stanford (2000)
14. East, D., Truszczyński, M.: Predicate-calculus based logics for modeling and solving search problems. ACM Transactions on Computational Logic 7, 38–83 (2006)
15. Eiter, T., Fink, M.: Uniform equivalence of logic programs under the stable model semantics. In: Palamidessi, C. (ed.) ICLP 2003. LNCS, vol. 2916, pp. 224–238. Springer, Heidelberg (2003)
16. Eiter, T., Fink, M., Tompits, H., Woltran, S.: Strong and uniform equivalence in answer-set programming: Characterizations and complexity results for the non-ground case. In: Proceedings of the 20th National Conference on Artificial Intelligence (AAAI 2005), pp. 695–700 (2005)
17. Eiter, T., Fink, M., Woltran, S.: Semantical characterizations and complexity of equivalences in answer set programming, ACM Transactions on Computational Logic (to appear, 2006)
18. Eiter, T., Gottlob, G.: On the computational cost of disjunctive logic programming: propositional case. Annals of Mathematics and Artificial Intelligence 15(3-4), 289–323 (1995)
19. Eiter, T., Tompits, H., Woltran, S.: On solution correspondences in answer-set programming. In: Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI 2005), pp. 97–102. Morgan Kaufmann, San Francisco (2005)
20. Elkabani, I., Pontelli, E., Son, T.C.: Smodels with CLP and its applications: a simple and effective approach to aggregates in ASP. In: Demoen, B., Lifschitz, V. (eds.) ICLP 2004. LNCS, vol. 3132, pp. 73–89. Springer, Heidelberg (2004)
21. Faber, W., Greco, G., Leone, N.: Magic sets and their application to data integration. In: Eiter, T., Libkin, L. (eds.) ICDT 2005. LNCS, vol. 3363, pp. 306–320. Springer, Heidelberg (2004)

22. Faber, W., Leone, N., Pfeifer, G.: Recursive aggregates in disjunctive logic programs: semantics and complexity. In: Alferes, J.J., Leite, J.A. (eds.) JELIA 2004. LNCS(LNAI), vol. 3229, pp. 200–212. Springer, Heidelberg (2004)
23. Fink, M., Pichler, R., Tompits, H., Woltran, S.: Complexity of rule redundancy in non-ground answer-set programming over finite domains. In: Proceedings of the 9th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2007). LNCS (LNAI), vol. 4483, pp. 123–135. Springer, Heidelberg (2007)
24. Gebser, M., Liu, L., Namasivayam, G., Neumann, A., Schaub, T., Truszczyński, M.: The first answer set programming system competition. In: Proceedings of the 9th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2007). LNCS (LNAI), vol. 4483, pp. 3–17. Springer, Heidelberg (2007)
25. Gelfond, M.: Representing knowledge in A-Prolog. In: Kakas, A.C., Sadri, F. (eds.) Computational Logic: Logic Programming and Beyond. LNCS(LNAI), vol. 2408, pp. 413–451. Springer, Heidelberg (2002)
26. Gelfond, M., Leone, N.: Logic programming and knowledge representation – the A-prolog perspective. Artificial Intelligence 138, 3–38 (2002)
27. Gelfond, M., Lifschitz, V.: The stable semantics for logic programs. In: Proceedings of the 5th International Conference on Logic Programming (ICLP 1988), pp. 1070–1080. MIT Press, Cambridge (1988)
28. Gelfond, M., Lifschitz, V.: Classical negation in logic programs and disjunctive databases. New Generation Computing 9, 365–385 (1991)
29. Heyting, A.: Die formalen Regeln der intuitionistischen Logik, Sitzungsberichte der Preussischen Akademie von Wissenschaften. Physikalisch-mathematische Klasse, 42–56 (1930)
30. Janhunen, T., Oikarinen, E., Tompits, H., Wotran, S.: Modularity aspects of disjunctive stable models. In: Proceedings of the 9th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2007). LNCS(LNAI), vol. 4483, pp. 175–187. Springer, Heidelberg (2007)
31. Kowalski, R.: Predicate logic as a programming language. In: Proceedings of the Congress of the International Federation for Information Processing (IFIP-1974) (Amsterdam), North Holland, pp. 569–574 (1974)
32. Kowalski, R.: Logic for problem solving. North Holland, Amsterdam (1979)
33. Lifschitz, V., Pearce, D., Valverde, A.: Strongly equivalent logic programs. ACM Transactions on Computational Logic 2(4), 526–541 (2001)
34. Lifschitz, V., Pearce, D., Valverde, A.: A characterization of strong equivalence for logic programs with variables. In: Proceedings of the 9th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2007). LNCS(LNAI), vol. 4483, pp. 188–200. Springer, Heidelberg (2007)
35. Lifschitz, V., Turner, H.: Splitting a logic program. In: Proceedings of the 11th International Conference on Logic Programming (ICLP 1994), pp. 23–37 (1994)
36. Lin, F.: Reducing strong equivalence of logic programs to entailment in classical propositional logic. In: Proceedings of the 8th International Conference on Principles of Knowledge Representation and Reasoning (KR 2002), Morgan Kaufmann, San Francisco (2002)
37. Lin, F., Zhao, Y.: ASSAT: Computing answer sets of a logic program by SAT solvers. In: Proceedings of the 18th National Conference on Artificial Intelligence (AAAI 2002), pp. 112–117. AAAI Press, Stanford (2002)
38. Liu, L., Pontelli, E., Son, T.C., Truszczyński, M.: Logic programs with abstract constraint atoms: the role of computations. In: Proceedings of the 23rd International Conference on Logic Programming (ICLP 2007). LNCS, Springer, Heidelberg (2007) (this volume)

39. Liu, L., Truszczyński, M.: Pbmodels - software to compute stable models by pseudoboolean solvers. In: Baral, C., Greco, G., Leone, N., Terracina, G. (eds.) LPNMR 2005. LNCS(LNAI), vol. 3662, pp. 410–415. Springer, Heidelberg (2005)
40. Liu, L., Truszczyński, M.: Properties of programs with monotone and convex constraints. In: Proceedings of the 20th National Conference on Artificial Intelligence (AAAI-05), pp. 701–706. AAAI Press, Stanford (2005)
41. Liu, L., Truszczyński, M.: Properties and applications of programs with monotone and convex constraints. Journal of Artificial Intelligence Research 27, 299–334 (2006)
42. Marek, V.W., Remmel, J.B.: Set constraints in logic programming. In: Lifschitz, V., Niemelä, I. (eds.) Logic Programming and Nonmonotonic Reasoning. LNCS(LNAI), vol. 2923, pp. 167–179. Springer, Heidelberg (2003)
43. Marek, V.W., Truszczyński, M.: Stable models and an alternative logic programming paradigm. In: The Logic Programming Paradigm: a 25-Year Perspective, pp. 375–398. Springer, Berlin (1999)
44. Marek, V.W., Truszczyński, M.: Logic programs with abstract constraint atoms. In: Proceedings of the 19th National Conference on Artificial Intelligence (AAAI 2004), pp. 86–91. AAAI Press, Stanford (2004)
45. Marek, W., Truszczyński, M.: Autoepistemic logic. Journal of the ACM 38(3), 588–619 (1991)
46. McCarthy, J., Hayes, P.: Some philosophical problems from the standpoint of artificial intelligence, Machine Intelligence 4, pp. 463–502. Edinburgh University Press (1969)
47. Mitchell, D.G., Ternovska, E.: A framework for representing and solving NP search problems. In: Proceedings of the 20th National Conference on Artificial Intelligence (AAAI 2005), pp. 430–435. AAAI Press, Stanford (2005)
48. Niemelä, I.: Logic programming with stable model semantics as a constraint programming paradigm. Annals of Mathematics and Artificial Intelligence 25(3-4), 241–273 (1999)
49. Pelov, N.: Semantics of logic programs with aggregates, PhD Thesis. Department of Computer Science, K.U. Leuven, Leuven, Belgium (2004)
50. Robinson, J.A.: A machine-oriented logic based on resolution principle. Journal of the ACM 12, 23–41 (1965)
51. Simons, P., Niemelä, I., Soininen, T.: Extending and implementing the stable model semantics. Artificial Intelligence 138, 181–234 (2002)
52. Son, T., Pontelli, E.: A constructive semantic characterization of aggregates in answer set programming, Theory and Practice of Logic Programming (Accepted 2007), available at `http://arxiv.org/abs/cs.AI/0601051`
53. Son, T., Pontelli, E., Tu, P.H.: Answer sets for logic programs with arbitrary abstract constraint atoms. In: Proceedings of the 21st National Conference on Artificial Intelligence (AAAI 2006), pp. 129–134. AAAI Press, Stanford (2006)
54. Truszczyński, M.: Strong and uniform equivalence of nonmonotonic theories — an algebraic approach. In: Proceedings of the 10th International Conference on Principles of Knowledge Representation and Reasoning (KR 2006), pp. 389–399. AAAI Press, Stanford (2006)
55. Truszczyński, M.: The modal logic S4F, the default logic, and the logic here-and-there. In: Proceedings of the 22nd National Conference on Artificial Intelligence (AAAI 2007), AAAI Press, Stanford (2007)
56. Turner, H.: Strong equivalence made easy: nested expressions and weight constraints. Theory and Practice of Logic Programming 3, 609–622 (2003)

57. van Emden, M.H., Kowalski, R.A.: The semantics of predicate logic as a programming language. Journal of the ACM 23(4), 733–742 (1976)
58. Van Gelder, A., Ross, K.A., Schlipf, J.S.: The well-founded semantics for general logic programs. Journal of the ACM 38(3), 620–650 (1991)
59. Vennekens, J., Denecker, M.: An algebraic account of modularity in ID-logic. In: Baral, C., Greco, G., Leone, N., Terracina, G. (eds.) LPNMR 2005. LNCS (LNAI), vol. 3662, pp. 291–303. Springer, Heidelberg (2005)
60. Vennekens, J., Gilis, D., Denecker, M.: Splitting an operator: an algebraic modularity result and its applications to logic programming. In: Demoen, B., Lifschitz, V. (eds.) ICLP 2004. LNCS, vol. 3132, pp. 195–209. Springer, Heidelberg (2004)

On Finitely Recursive Programs

S. Baselice, P.A. Bonatti, and G. Criscuolo

Università di Napoli "Federico II"

Abstract. *Finitary programs* are a class of logic programs admitting functions symbols and hence infinite domains. In this paper we prove that a larger class of programs, called *finitely recursive programs*, preserves most of the good properties of finitary programs under the stable model semantics, namely: (i) finitely recursive programs enjoy a compactness property; (ii) inconsistency check and skeptical reasoning are semidecidable; (iii) skeptical resolution is complete. Moreover, we show how to check inconsistency and answer skeptical queries using finite subsets of the ground program instantiation.

1 Introduction

Answer Set Programming (ASP) is one of the most interesting achievements in the area of Logic Programming and Nonmonotonic Reasoning. It is a declarative problem solving paradigm, mainly centered around some well-engineered implementations of the stable model semantics of logic programs [8,9], such as SMODELS and DLV [14,5].

The most popular ASP languages are extensions of Datalog, that is, function-free, possibly disjunctive logic programs with negation as failure. The lack of function symbols has several drawbacks, related to expressiveness and encoding style [4]. In order to overcome such limitations and reduce the memory requirements of current implementations, a class of logic programs called *finitary programs* has been introduced [4].

In finitary programs function symbols (hence infinite domains) and recursion are allowed. However, recursion is restricted by requiring each ground atom to depend on finitely many ground atoms; such programs are called *finitely recursive*. Moreover, only finitely many ground atoms must occur in *odd-cycles*—that is, cycles of recursive calls involving an odd number of negative subgoals—which means that there should be only finitely many potential sources of inconsistencies. These two restrictions bring a number of nice semantic and computational properties [4]. In general function symbols make the stable model semantics highly undecidable [13]. On the contrary, if the given program is finitary, then consistency checking, ground credulous queries, and ground skeptical queries are decidable. Nonground queries were proved to be r.e.-complete. Moreover, a form of compactness holds: an inconsistent finitary program has always a finite *unstable kernel*, i.e. a finite subset of the program's ground instantiation with no stable models. All of these properties are quite unusual for a nonmonotonic logic.

As function symbols are being integrated in state-of-the-art reasoners such as DLV,[1] it is interesting to extend these good properties to larger program classes. This goal requires a better understanding of the role of each restriction in the definition of finitary programs. It has already been noted [4] that by dropping the first condition

[1] A PRIN project on this topic, funded by the Italian government, has just started.

V. Dahl and I. Niemelä (Eds.): ICLP 2007, LNCS 4670, pp. 89–103, 2007.

(i.e., if the program is not finitely recursive) one obtains a superclass of stratified programs, whose complexity is then far beyond computability. In the same paper, it is argued that the second restriction (on odd-cycles) is needed for the decidability of ground queries. However, if a program is only finitely recursive (and infinitely many odd-cycles are allowed), then the results of [4] do not characterize the exact complexity of reasoning and say nothing about compactness, nor about the completeness of the skeptical resolution calculus [3].

In this paper we shall extend and refine those results, and prove that several important properties of finitary programs carry over to all finitely recursive programs. We shall prove that for all finitely recursive programs the compactness property still holds, and that inconsistency check and skeptical reasoning are semidecidable. Moreover, we will extend the completeness of skeptical resolution [3,4] to all finitely recursive programs. Our results will clarify the role that each of the two restrictions defining finitary programs has in ensuring their properties.

In order to prove these results we use program splittings [11], but the focus is shifted from splitting sequences (whose elements are sublanguages) to the corresponding sequences of subprograms. For this purpose we introduce the notion of *module sequence*. It turns out that finitely recursive programs are exactly those programs whose module sequences are made of finite elements. Moreover a finitely recursive program P has a stable model iff each element P_i of the sequence has a stable model, a condition which is not valid in general for normal programs.

The paper is organized as follows. The next section is devoted to preliminaries. In Section 3, we give the definition of module sequences for a program and study their properties. In Section 4, we study the theoretical properties and the expressiveness of finitely recursive programs. Section 5 deals with the completeness of skeptical resolution, and Section 6 concludes the paper with a final discussion of our results.

2 Preliminaries

We assume the reader to be familiar with the classical theory of logic programming [12].

Normal logic programs are sets of rules $A \quad L_1 \quad L_n$ ($n \quad 0$), where A is a logical atom and each L_i ($i \quad 1 \quad n$) is a literal. If r is such a rule, then let $head(r) \quad A$ and $body(r) \quad L_1 \quad L_n$. Moreover, let $body \ (r)$ (respectively $body \ (r)$) be the set of all atoms A s.t. A (respectively $\quad A$) belongs to $body(r)$.

The ground instantiation of a program P is denoted by $\mathsf{Ground}(P)$, and the set of atoms occurring in $\mathsf{Ground}(P)$ is denoted by $atom(P)$. Similarly, $atom(r)$ denotes the set of atoms occurring in a rule r.

A Herbrand model M of P is a *stable model* of P iff $M \quad \mathsf{lm}(P^M)$, where $\mathsf{lm}(X)$ denotes the least model of a positive program X, and P^M is the *Gelfond-Lifschitz transformation* of P, obtained from $\mathsf{Ground}(P)$ by (i) removing all rules r such that $body \ (r)$ $M \quad$, and (ii) removing all negative literals from the body of the remaining rules [8,9].

The *dependency graph of a normal program P* is a labelled directed graph, denoted by $DG(P)$, whose vertices are the ground atoms of P's language. Moreover, (i) there exists an edge labelled '+' (called positive edge) from A to B iff for some rule r $\mathsf{Ground}(P)$, $A \quad head(r)$ and $B \quad body(r)$; (ii) there exists an edge labelled '-' (called

negative edge) from A to B iff for some rule r Ground(P), A $head(r)$ and B $body(r)$.

The *dependency graph of an atom A* occurring in a normal program P is the subgraph of $DG(P)$ including A and all vertices reachable from A in $DG(P)$.

An atom A *depends positively* (respectively *negatively*) on B if there is a directed path from A to B in the dependency graph with an even (respectively odd) number of negative edges. Moreover, each atom depends positively on itself. If A depends positively (respectively negatively) on B we write A B (respectively A B). A *depends* on B (in symbols: A B) if either A B or A B.

Definition 1. *An* odd-cycle *is a cycle in the dependency graph with an odd number of negative edges. A ground atom is* odd-cyclic *if it occurs in an odd-cycle.*

Note that there exists an odd-cycle iff for some ground atom A, A A.

Definition 2 (Finitely recursive programs). *A normal program P is* finitely recursive *iff each ground atom A depends on finitely many ground atoms.*

For example, most standard list manipulation programs (`m m d m v` etc.) are finitely recursive. The reader can find numerous examples of finitely recursive programs in [4]. Many interesting programs are finitely recursive but not finitary, due to integrity constraints that apply to infinitely many individuals.

Example 1. Fig. 4 of [4] illustrates a finitary program for reasoning about actions, defining—among others—two predicates `ds`(*fluent time*) and `d` (*action time*). The simplest way to add a constraint that forbids any parallel execution of two incompatible actions a_1 and a_2 is including a rule f f `d` (a_1 T) `d` (a_2 T) in that program, where f is a fresh propositional symbol (often such rules are equivalently expressed as *denials* `d` (a_1 T) `d` (a_2 T)). This program is not finitary (because f depends on infinitely many atoms since T has an infinite range of values) but it can be reformulated as a finitely recursive program by replacing the above rule with

$$f(T) \qquad f(T) \texttt{ d } (a_1\ T) \texttt{ d } (a_2\ T)$$

Note that the new program is finitely recursive but not finitary, because the new rule introduces infinitely many odd cycles (one for each instance of $f(T)$).

If P is a finitely recursive program then, for each atom A occurring in Ground(P), the dependency graph of A is a finite graph. Moreover, if A is an odd-cyclic atom then its dependency graph contains all the odd-cycles in which A occurs. So, the set of atoms on which A depends and the set of odd-cycles in which A occurs are finite.

The class of *finitary programs* is obtained by imposing a further constraint on the dependency graph, that is, a finite upper bound on the number of odd-cyclic atoms.

Definition 3 (Finitary programs). *We say that a normal program P is* finitary *if the following conditions hold:*

1. *P is finitely recursive.*
2. *There are finitely many odd-cyclic atoms in the dependency graph of P.*

The stable models [8,9] of a normal program can be obtained by splitting a program into two modules—of which one is self-contained, while the other module depends on the former—and then combining the stable models of the two modules.

Definition 4 (Splitting set [2],[10]). *A* splitting set *of a normal program P is any set U of atoms such that, for any rule r Ground(P), if $head(r)$ U then $atom(r) \subseteq U$. If U is a splitting set for P, we also say that U splits P. The set of rules r Ground(P) such that $atom(r) \subseteq U$ is called the* bottom *of P relative to the splitting set U and is denoted by $bot_U(P)$.*

The top module should be partially evaluated w.r.t. the bottom.

Definition 5 (Partial evaluation [2],[10]). *The* partial evaluation *of a normal program P with splitting set U w.r.t. a set of ground atoms X is the program e_U(Ground(P) X) defined as follows:*

$$e_U(\mathsf{Ground}(P)\ X) \quad r' \quad \text{there exists } r \quad \mathsf{Ground}(P)\ s\ t\ (body\ (r) \quad U) \subseteq X$$
$$\text{and } (body\ (r) \quad U) \quad X \quad \text{and } head(r') \quad head(r)$$
$$body\ (r') \quad body\ (r) \quad U\ body\ (r') \quad body\ (r) \quad U$$

Then stable models can be computed in two stages.

Theorem 1 (Splitting theorem [10]). *Let U be a splitting set for a logic program P. An interpretation M is a stable model of P iff M J I, where*

1. *I is a stable model of $bot_U(P)$, and*
2. *J is a stable model of e_U(Ground(P) $bot_U(P)$ I).*

The splitting theorem has been extended to transfinite sequences in [11].

Definition 6. *A* (transfinite) sequence *is a family whose index set is an initial segment of ordinals,* : μ *. The ordinal μ is the* length *of the sequence.*
A sequence $U \rangle_{\mu}$ of sets is monotone *if $U \subseteq U$ whenever β, and* continuous *if, for each limit ordinal μ, $U \bigcup U$.*

Definition 7 (Lifschitz-Turner, [11]). *A* splitting sequence *for a program P is a monotone, continuous sequence $U \rangle_{\mu}$ of splitting sets for P s.t. $\bigcup_{\mu} U$ $atom(P)$.*

In [11] Lifschitz and Turner generalized the splitting theorem to splitting sequences. They proved that each stable model M of P equals the infinite union of a sequence of models $M \rangle_{\mu}$ such that (i) M_0 is a stable model of $bot_{U_0}(P)$, (ii) for all successor ordinals μ, M is a stable model of $e_{U_{\ 1}}(bot_U(P)\ bot_{U_{\ 1}}(P) \bigcup M$), and (iii) for all limit ordinals μ, M_{λ} . They proved also that each sequence of models with these properties yields a stable model of P.

3 Module Sequences and a Normal Form for Splitting Sequences

In our study of the computational properties of finitely recursive programs, we shall replace the notion of splitting sequence with suitable sequences of bottom programs whose length is bounded by .

Definition 8 (GH, Module sequence). *Let P be a normal program and let the set of its* ground heads *be*

$$GH \quad p \quad p \quad head(r) \quad r \quad \mathsf{Ground}(P)$$

The module sequence $P_1 \; P_2 \; P_3 \quad P_n$ induced by an enumeration $p_1 \; p_2 \; p_3 \; p_n$ of GH *is defined as follows:*

$$P_1 \quad r \quad \mathsf{Ground}(P) \quad p_1 \textit{ depends on } head(r)$$
$$P_{i\ 1} \quad P_i \quad r \quad \mathsf{Ground}(P) \quad p_{i\ 1} \textit{ depends on } head(r) \quad (i \quad 1)$$

Of course, we are interested in those properties of module sequences which are independent of the enumeration of GH. The following proposition follows easily from the definitions. We say that a ground subprogram $P' \subseteq \mathsf{Ground}(P)$ is *downward closed*, if for each atom A occurring in P', P' contains all the rules $r \quad \mathsf{Ground}(P)$ s.t. $A \quad head(r)$.

Proposition 1. *Let P be a normal program. For all module sequences $P_1 \; P_2 \quad$, for P:*

1. $\bigcup_{i\ 1} P_i \quad \mathsf{Ground}(P)$,
2. *for each* $i \quad 1$ *and* $j \quad i$, $atom(P_i)$ *is a splitting set of* P_j *and* $P_i \quad bot_{atom(P_i)}(P_j)$,
3. *for each* $i \quad 1$, $atom(P_i)$ *is a splitting set of* P *and* $P_i \quad bot_{atom(P_i)}(P)$,
4. *for each* $i \quad 1$, P_i *is downward closed.*

Therefore, we immediately see that each module sequence for P consists of the bottom programs corresponding to a particular splitting sequence $atom(P_i)\rangle_i$ that depends on the underlying enumeration of GH. Roughly speaking, such sequences constitute a *normal form* for splitting sequences. If P is finitely recursive, then "normal form" sequences can be required to satisfy an additional property:

Definition 9 (Smoothness). *A transfinite sequence of sets $X \rangle_{\mu}$ is* smooth *iff X_0 is finite and for each non-limit ordinal $1 \quad \mu$, the difference $X_{\ 1} \quad X$ is finite.*

Note that when μ (as in module sequences), smoothness implies that each X in the sequence is finite. Finitely recursive programs are completely characterized by smooth module sequences:

Theorem 2. *The following are equivalent:*

1. *P is finitely recursive;*
2. *P has a smooth module sequence (where each P_i is finite);*
3. *all module sequences for P are smooth.*

Since smooth module sequences clearly correspond to smooth splitting sequences, the above theorem implies that by working with module sequences we are implicitly restricting our attention to *smooth splitting sequences of length* . Then the characterization of finitely recursive programs can be completed as follows, using standard splitting sequences:

Corollary 1. *For all programs P, the following are equivalent:*

1. *P is finitely recursive;*
2. *P has a smooth splitting sequence with length .*

Next we illustrate how module sequences provide an incremental characterization of the stable models of normal logic programs.

Remark 1. By Proposition 1 and the Splitting Theorem, if P is a normal program and $P_1\ P_2\ \ P_n$ is a module sequence for P, then for all $j\ \ i\ \ 1$ and for all stable models M_j of P_j, the set $M_i\ \ M_j\ \ atom(P_i)$ is a stable model of P_i. Similarly, for each stable model M of P, the set $M_i\ \ M\ \ atom(P_i)$ is a stable model of P_i.

Roughly speaking, the following theorem rephrases the splitting sequence theorem of [11] in terms of module sequences. The original splitting sequence theorem applies to sequences of disjoint program "slices", while our theorem applies to monotonically increasing program sequences. Since no direct proof of the splitting sequence theorem was ever published (only the proof of a more general result for default logic was published [15]), here we give a direct proof of our result.

Theorem 3 (Module sequence theorem). *Let P be a normal program and $P_1\ P_2$ be a module sequence for P. Then M is a stable model of P iff there exists a sequence $M_1\ M_2\ \ $ s.t. :*

1. *for each $i\ \ 1$, M_i is a stable model of P_i,*
2. *for each $i\ \ 1$, $M_i\ \ M_{i\ 1}\ \ atom(P_i)$,*
3. *$M\ \ \bigcup_{i\ 1} M_i$.*

Proof. Let M be a stable model of P. Since $P_1\ P_2\ \ $ is a module sequence for P then for each $i\ \ 1$, $atom(P_i)$ is a splitting set of P and $P_i\ \ bot_{atom(P_i)}(P)$. So, we consider the sequence $M_1\ \ M\ \ atom(P_1)\ M_2\ \ M\ \ atom(P_2)\ \ $ where, by the splitting theorem [10], for each $i\ \ 1$, M_i is a stable model of P_i, $M_{i\ 1}\ \ atom(P_i)\ \ M_i$ and, by definition of $M_1\ M_2\ \ $ and by property 1 of proposition 1, $\bigcup_i M_i\ \ M$. Then for each stable model M of P there exists a sequence of finite sets of ground atoms that satisfies the properties 1, 2 and 3.

Now, without loss of generality, suppose P ground and suppose that there exists a sequence $M_1\ M_2\ \ $ that satisfies the properties 1, 2 and 3. We have to prove that the set $M\ \ \bigcup_{i\ 1} M_i$ is a stable model of P; equivalently,

$$\bigcup_{i\ 1} M_i\ \ \mathsf{lm}(P^M)$$

First we show that $\bigcup_{i\ 1} M_i \subseteq \mathsf{lm}(P^M)\ $ Property 2 implies that for all $i\ \ 1$, $(M\ atom(P_i))\ \ M_i$; consequently $P_i^M\ \ P_i^{M_i}$. Moreover, since $P_i \subseteq P$, we have $P_i^M \subseteq P^M$, and hence

$$P_i^{M_i}\ \ P_i^M \subseteq P^M$$

By the monotonicity of $\mathsf{lm}(\ )$ [1,6], $\ i\ \ 1\ \mathsf{lm}(P_i^{M_i}) \subseteq \mathsf{lm}(P^M)\ $ Moreover, M_i is a stable model of P_i and then $M_i\ \ \mathsf{lm}(P_i^{M_i})$, so

$$i\ \ 1\ M_i \subseteq \mathsf{lm}(P^M)$$

We are left to prove the opposite inclusion, that is $\mathsf{lm}(P^M) \subseteq \bigcup_{i\ 1} M_i\ $ Suppose $p\ \ \mathsf{lm}(P^M)$. From $P_i^M\ \ P_i^{M_i}$ and $\bigcup_i P_i\ \ P$ it follows that $\bigcup_i P_i^{M_i}\ \ \bigcup_i P_i^M\ \ P^M$. Moreover, by definition, for each $i\ \ 1$, P_i is downward closed, then there must be a k s.t. $p\ \ \mathsf{lm}(P_k^{M_k})$. We conclude that $p\ \ M_k$ and then $p\ \ \bigcup_i M_i$.

The module sequence theorem suggests a relationship between the consistency of a program P and the consistency of each step in P's module sequences.

Definition 10. *A module sequence $P_1\ P_2$ for a normal program P is* inconsistent *if there exists an i s.t. P_i has no stable model,* consistent *otherwise.*

Proposition 2. *If a normal program P has an inconsistent module sequence then P is inconsistent.*

Proof. Suppose that P has an inconsistent module sequence $P_1\ P_2$. Then there exists a P_i that has no stable models. Hence, P has an inconsistent bottom set and then P is inconsistent by the splitting theorem.

Next we show that the inconsistency of a module sequence S is invariant w.r.t. the enumeration of GH inducing S.

Theorem 4. *Let S $P_1\ P_2$ be a module sequence for a normal program P. If S is inconsistent then each module sequence for P is inconsistent.*

Proof. Let S $P_1\ P_2$ be an inconsistent module sequence for P induced by the enumeration $p_1\ p_2$ of GH and let i be the least index s.t. P_i is inconsistent. Let S' $P'_1\ P'_2$ be any module sequence for P induced by the enumeration $p'_1\ p'_2$ of GH. Since i is finite, there exists a finite k s.t. $p_1\ p_2$ $p_i \subseteq p'_1\ p'_2$ p'_k . So, by construction, $P_i \subseteq P'_k$ and then $atom(P_i) \subseteq atom(P'_k)$. Moreover, by definition, P_i is downward closed and then P_i $bot_{atom(P_i)}(P'_k)$. Since P_i is inconsistent then P'_k is inconsistent (by the splitting theorem) and then also S' is inconsistent.

In other words, for a given program P, either all module sequences are inconsistent, or they are all consistent. In particular, if P is consistent, then every member P_i of any module sequence for P must be consistent.

It may be tempting to assume that the converse holds, that is, if a module sequence for P is consistent, then P is consistent, too. Unfortunately, this statement is not valid in general, as the following example shows.

Example 5. *Consider the following program P_f (due to Fages [7]):*

$$\begin{array}{ll} q(X) & q(f(X)) \\ q(X) & q(f(X)) \\ r(0) & \end{array}$$

Note that P_f is not finitely recursive because, for each grounding substitution , $q(X)$ depends on the infinite set of ground atoms $q(f(X))$ $q(f(f(X)))$.

The first two rules in P_f are classically equivalent to

$$q(X) \quad [q(f(X)) \vee q(f(X))]$$

Since the body is a tautology and the stable models of a program are also classical models of the program, we have that a stable model of P_f should satisfy all ground instances of $q(X)$. However, the Gelfond-Lifschitz transformation w.r.t. such a model would contain only the first and the third program rules, and hence the least model of the transformation would contain no instance of $q(X)$. It follows that P_f is inconsistent (it has no stable models). Now consider the following extension P of P_f:

1. $q(X) \quad q(f(X)) \; p(X)$
2. $q(X) \quad q(f(X)) \; p(X)$
3. $r(0)$
4. $p(X) \quad p'(X)$
5. $p'(X) \quad p(X)$
6. $c(X) \quad c(X) \quad p(X)$

To see that P is inconsistent, suppose M is a stable model of P. By rules 4 and 5, each ground instance of $p(X)$ *can be either true or false. But each ground instance of* $c(X)$ *is odd-cyclic, therefore if* $p(X)$ *is false then rule 6 produces an inconsistency. It follows that all ground instances of* $p(X)$ *must be true in M. In this case, however, rules 1, 2 and 3 are equivalent to program* P_f *and prevent M from being a stable model, as explained above. So P is inconsistent.*

Next, consider the enumeration e $r(0) \; q(0) \; p(0) \; p'(0) \; c(0) \; q(f(0)) \; p(f(0)) \; p'(f(0)) \; c(f(0))$ *of the set GH. This enumeration induces the following module sequence for P.*

$$
\begin{array}{lll}
P_0 & r(0) & \\
P_1 \quad P_0 \quad \bigcup_k & q(X) \quad q(f(X)) \; p(X) & \\
& q(X) \quad q(f(X)) \; p(X) & \\
& p(X) \quad p'(X) & \\
& p'(X) \quad p(X) \quad [X \; f^k(0)] & \\
P_{i\ 1} \quad P_i & c(X) \quad c(X) \quad p(X) \; [X \; f^{i\ 1}(0)] & (i \quad 1)
\end{array}
$$

Note that $M_0 \quad r(0)$ *is a stable model of* P_0 *and for each* $i \quad 1$ *and* $k \quad i \quad 2$

$$
\begin{array}{ll}
M_i^k & r(0) \; p(f^0(0)) \; p(f^1(0)) \; p(f^2(0)) \quad p(f^k(0)) \\
& p'(f^{k\ 1}(0)) \; p'(f^{k\ 2}(0)) \quad p'(f^{k\ j}(0)) \\
& q(f^0(0)) \; q(f^1(0)) \; q(f^2(0)) \quad q(f^k(0))
\end{array}
$$

is a stable model of P_i*. Therefore, each* P_i *is consistent although* $\bigcup_i P_i \quad \mathsf{Ground}(P)$ *is inconsistent. This happens because for each stable model M of* P_1 *there exists a* P_j $(j \quad 1)$ *such that M is not the bottom part of any stable model of* P_j*. Intuitively, M has been "eliminated" at step j. In this example* P_1 *has infinitely many stable models, and it turns out that no finite step eliminates all of them. Consequently, each* P_i *in the module sequence is consistent, but the entire program is not.*

4 Properties of Finitely Recursive Programs

The smoothness of finitely recursive programs overcomes the problem illustrated by the above example. Since every module P_i is finite, no step in the sequence has infinitely many stable models. Therefore if every P_i is consistent, the entire program P must be consistent, too, and the following theorem holds:

Theorem 6. *For all finitely recursive programs P:*

1. *if P is consistent then every module sequence for P is consistent;*
2. *if some module sequence for P is consistent, then P is consistent.*

Proof. We prove 1 by contraposition. Suppose that P has an inconsistent module sequence $P_1\ P_2\ \ldots$. Then, there exists a P_i with no stable models. Therefore, P has an inconsistent bottom set and hence P is inconsistent by the splitting theorem.

To prove point 2, consider a module sequence S for P. If S is consistent then each P_i has a nonempty set of stable models. It suffices to prove that there exists a sequence $M_1\ M_2\ \ldots$ of stable models of $P_1\ P_2\ \ldots$, respectively, that satisfies the properties of Theorem 3, because this implies that $M\ \ \bigcup_i M_i$ is a stable model of P.

We call a stable model M_i of P_i *"bad"* if there exists a $k\ \ i$ s.t. no model M_k of P_k extends M_i, *"good"* otherwise. We say that M_k extends M_i if $M_k\ \ atom(P_i)\ \ M_i$ (see remark 1). We claim that each P_i must have at least a *"good"* model.

To prove the claim, suppose that all models of P_i are *"bad"*. Since P_i is a finite program it has a finite number $M_{i_1}\ \ \ldots\ \ M_{i_r}$ of models. By assumption, for each M_{i_j} there is a program $P_{k_{i_j}}$ none of whose models extends M_{i_j}. Let $k\ \ max\ k_{i_1}\ \ \ldots\ \ k_{i_r}$; then no model of P_k extends a model of P_i, and this is a contradiction because P_k, by hypotheses, has at least a stable model M_k and by the splitting theorem M_k extends a stable model of P_i. This proves the claim.

Now, let M_1 be a *"good"* stable model of P_1; then there must exist a *"good"* stable model of P_2 that extends M_1, exactly for the same reasons, and so on. Therefore there exists an infinite sequence $M_1\ M_2\ \ldots$ that satisfies both properties 1 and 2 of Theorem 3 and hence $M\ \ \bigcup_i M_i$ is a stable model of P.

Note that if the enumeration of GH is effective, then for each i the corresponding subprogram P_i can be effectively constructed. In this case $\langle atom(P_i)\rangle_i$ is an effective enumerable splitting sequence for P. This observation is the basis for the complexity results proved in the rest of the paper.

4.1 Compactness

Here we prove that the compactness theorem for finitary programs actually holds for all finitely recursive programs.

Definition 11 ([4]). *An* unstable kernel *for a normal program P is a set $K \subseteq \mathsf{Ground}(P)$ with the following properties:*

1. *K is* downward closed*, that is, for each atom A occurring in K's rules, K contains all the rules $r\ \ \mathsf{Ground}(P)$ s.t. $A\ \ head(r)$.*
2. *K has no stable model.*

Theorem 7 (Compactness). *A finitely recursive program P has no stable model iff it has a finite unstable kernel.*

Proof. By Theorems 2 and 6, P has no stable model iff it has an inconsistent module sequence. So, let $P_1\ P_2\ \ldots\ P_n$ be an inconsistent module sequence for P and let $i\ \ 1$ s.t. P_i is inconsistent. By proposition 1, $P_i \subseteq \mathsf{Ground}(P)$ and P_i is also downward closed. So it is an unstable kernel for P. Moreover, by Theorem 2, P_i is finite.

The compactness theorem for finitary programs [4] can now be regarded as a corollary of the above theorem.

4.2 The Complexity of Reasoning with Finitely Recursive Programs

As we said before, by taking an effective enumeration of the set GH, one can effectively compute each element of the corresponding module sequence. Let us call CONSTRUCT $(P\ i)$ an effective procedure that, given the finitely recursive program P and the index i, returns the ground program P_i, and let $SM(P_i)$ be an algorithm that computes the finite set of the finite stable models of P_i:

Theorem 8. *Let P be a finitely recursive program. Deciding whether P is inconsistent is at most semidecidable.*

Proof. Given a module sequence $P_1\ P_2\ \ P_n\ \ $ for P, consider the algorithm CONSISTENT (P).

Algorithm CONSISTENT (P)

```
1: i   0;
2: answer   TRUE;
3: repeat
4:    i   i   1;
5:    P_i   CONSTRUCT(P i);
6:    if SM(P_i)   then
7:       answer   FALSE;
8: until   answer OR P_i   Ground(P)
9: return answer;
```

By Theorems 2 and 6, P is inconsistent iff there exists an $i\ \ 1$ s.t. P_i is inconsistent (note that we can always check the consistency of P_i because P_i is finite). Then, the algorithm returns $FALSE$ iff P is inconsistent.

Note that if Ground(P) is infinite then any module sequence for P is infinite and the algorithm CONSISTENT (P) terminates iff P is not consistent.

Next we deal with skeptical inference. Recall that a closed formula F is a skeptical consequence of P iff F is satisfied (according to classical semantics) by all the stable models of P.

Theorem 9. *Let P be a finitely recursive program and $P_1\ P_2\ \ $ be a module sequence for P. A ground formula F is a skeptical consequence of P iff there exists a finite $k\ \ 1$ s.t. F is a skeptical consequence of P_k and $atom(F) \subseteq atom(P_k)$.*

Proof. Let h be the least integer s.t. $atom(F) \subseteq atom(P_h)$ (note that there always exists such a h because $atom(F)$ is finite). Suppose that there exists a $k\ \ h$ s.t. F is a skeptical consequence of P_k. Since P_k is a bottom for P, then each stable model of P contains a model of P_k and then satisfies F. So, F is a skeptical consequence of P. This proves the "if" part.

Now suppose that, for each $k\ \ h$, F is not a skeptical consequence of P_k. This implies that each P_k is consistent (hence P is consistent) and, moreover, the set S of all the stable models of P_k that falsify F is not empty.

Note that S is finite because P_k is finite (as P is finitely recursive). So, if all the models in S are "*bad*" (cf. the proof of Theorem 6), then there exists a finite integer $j \quad k$ s.t. no model of P_j contains any model of S. Consequently, F is a skeptical consequence of P_j—a contradiction.

Therefore at least one of these model must be *good*. Then there must be a model M of P that contains this "*good*" model of P_k, and hence F is not a skeptical consequence of P.

The next theorem follows easily.

Theorem 10. *Let P be a finitely recursive program. For all ground formulas F, the problem of deciding whether F is a skeptical consequence of P is at most semidecidable.*

For a complete characterization of the complexity of ground queries and inconsistency checking, we are only left to prove that the above upper bounds are tight.

Theorem 11. *Deciding whether a finitely recursive program P is inconsistent is r.e.-complete.*

Proof. By Theorem 8 we have that the inconsistency check over the class of finitely recursive programs is at most semidecidable.

Now we shall prove that it is also r.e.-hard by reducing the problem of skeptical inference of a quantified formula over a finitary program (that is an r.e.-complete problem [4, corollary 23]) to the problem of inconsistency check over a finitely recursive program.

Let P be a finitary program and F be a closed quantified formula. Let $((l_{11} \vee l_{12} \vee \quad) \wedge (l_{21} \vee l_{22} \vee \quad) \wedge \quad)$ be the conjunctive normal form of $\neg F$. Then F is a skeptical consequence of P iff the program $P \quad C$ is inconsistent, where

$$C \quad \begin{array}{llll} p_1(\boldsymbol{x_1}) & l_{11} & l_{12} & p_1(\boldsymbol{x_1}) \\ p_2(\boldsymbol{x_2}) & l_{21} & l_{22} & p_2(\boldsymbol{x_2}) \end{array}$$

p_1, p_2, ... are new atom symbols not occurring in P or F, and $\boldsymbol{x_i}$ is the vector of all variables occurring in $(l_{i1} \vee l_{i2} \vee \quad)$. Note that $P \quad C$ is a finitely recursive program.

The constraints in C add no model to P, but they only discard those models of P that satisfy $F\theta$ (for some substitution θ). So, let $SM(P)$ be the set of stable models of P. Then each model in $SM(P \quad C)$ satisfies $\neg F$. $SM(P \quad C)$ (that is $P \quad C$ is inconsistent) iff either $SM(P)$ or all stable models of P satisfy F. Then $SM(P \quad C)$ iff F is a skeptical consequence of P.

Theorem 12. *Deciding whether a finitely recursive program P skeptically entails a ground formula F is r.e.-complete.*

Proof. As proved in Theorem 10 deciding whether a finitely recursive program P skeptically entails a ground formula F is at most semidecidable.

Now we shall prove that it is also r.e.-hard by reducing the problem of inconsistency check over a finitely recursive program to the problem of skeptical inference of a ground formula over a finitely recursive program.

Let P be a finitely recursive program and q be a new ground atom that doesn't occur in P. Then, P is inconsistent iff q is a skeptical consequence of P. Since q occurs in the head of no rule of P, q cannot occur in a model of P. So, P skeptically entails q iff P has no model.

Corollary 2. *Deciding whether a finitely recursive program P doesn't credulously entail a ground formula F is co-r.e. complete.*

5 Skeptical Resolution for Finitely Recursive Programs

In this section we extend the work in [3,4] by proving that skeptical resolution (a top-down calculus which is known to be complete for Datalog and finitary programs under the skeptical stable model semantics) is complete also for the class of finitely recursive programs. Skeptical resolution has several interesting properties. For example, it does not require the input program P to be instantiated before reasoning, and it can produce nonground (i.e., universally quantified) answer substitutions.

Due to space limitations, we are not able to describe the five inference rules of skeptical resolution here—the reader is referred to [3]. We only recall that a crucial rule called *failure rule* is expressed in terms of an abstract negation-as-failure mechanism derived from the notion of *support*. Recall that a support for a ground atom A is a set of negative literals obtained by unfolding the goal A w.r.t. the given program P until no positive literal is left.

Definition 12 ([3]). *Let A be a ground atom.* A ground counter-support *for A in a program P is a set of atoms K with the following properties:*

1. *For each support S for A, there exists* $B \quad S$ *s.t.* $B \quad K$.
2. *For each* $B \quad K$, *there exists a support S for A s.t.* $B \quad S$.

In other words, the first property says that K contradicts all possible ways of proving A, while the second property is a sort of relevance property. Informally speaking, the failure rule of skeptical resolution says that if all atoms in a counter-support are true, then all attempts to prove A fail, and hence A can be concluded.

Of course, in general, counter-supports are not computable and may be infinite (while skeptical derivations and their goals should be finite). In [3] the notion of counter-support is generalized to non ground atoms in the following way:

Definition 13. *A (generalized)* counter-support *for A is a pair* $K\ \theta\rangle$ *where K is a set of atoms and* θ *a substitution, s.t. for all grounding substitutions* $\ , K$ *is a ground counter-support for* $A\theta$.

The actual mechanism for computing counter-supports can be abstracted by means of a suitable function CounterSupp, mapping each (possibly nonground) atom A onto a set of *finite* generalized counter-supports for A. The underlying intuition is that function CounterSupp captures all the negative inferences that can actually be computed by the chosen implementation. To achieve completeness for the nonground skeptical resolution calculus, we need the negation-as-failure mechanism to be complete in the following sense.

Definition 14. *The function* CounterSupp *is* complete *iff for each atom A, for all of its ground instances Aγ, and for all ground counter-supports K for Aγ, there exist* $K' \theta\rangle$ CounterSupp(A) *and a substitution s.t.* $A\theta$ $A\gamma$ *and* K' K.

Skeptical resolution is based on *goals with hypotheses* (*h-goals* for short) which are pairs $(G \quad H)$ where H and G are finite sequences of literals. Roughly speaking, the answer to a query $(G \quad H)$ should be *yes* if G holds in all the stable models that satisfy H. Hence $(G \quad H)$ has the same meaning in answer set semantics as the formula $(\quad G \quad H)$. Finally, a *skeptical goal* (*s-goal* for short) is a finite sequence of h-goals, and a *skeptical derivation from P and* CounterSupp *with restart goal* G_0 is a (possibly infinite) sequence of s-goals $g_0, g_1, \ldots$, where each $g_{i\ 1}$ is obtained from g_i through one of the five rewrite rules of the calculus, as explained in [3]. This calculus is sound for *all* normal programs and counter-support calculation mechanisms, as stated in the following theorem.

Theorem 13 (Soundness, [3]). *Suppose that an* s-goal $(G \quad H)$ *has a successful skeptical derivation from P and* CounterSupp *with restart goal G and answer substitution θ. Then, for all grounding substitution , all the stable models of P satisfy* $(\quad G\theta \quad H\theta)$ *(equivalently,* $(\quad G\theta \quad H\theta)$ *is skeptically entailed by P).*

However, skeptical resolution is not always complete. Completeness analysis is founded on ground skeptical derivations, that require a ground version of CounterSupp.

Definition 15. *For all ground atoms A, let* CounterSupp$^g(A)$ *be the least set s.t. if* $K \ \theta\rangle$ CounterSupp(A') *and for some grounding , A* $A'\theta$ *, then* $K \quad \epsilon\rangle$ CounterSupp$^g(A)$*, where ϵ is the empty substitution.*

Theorem 14 (Finite Ground Completeness, [3]). *If some ground implication* G H *is skeptically entailed by a* finite *ground program P and* CounterSupp *is complete w.r.t. P, then* $(G \quad H)$ *has a successful skeptical derivation from P and* CounterSupp^g *with restart goal G. In particular, if G is skeptically entailed by P, then* $(G \quad)$ *has such a derivation.*

This basic theorem and the following standard lifting lemma allow to prove completeness for all finitely recursive programs.

Lemma 1 (Lifting, [3]). *Let* CounterSupp *be complete. For all skeptical derivations from* Ground(P) *and* CounterSupp^g *with restart goal* G_0*, there exists a substitution and a skeptical derivation ′ from P and* CounterSupp *with restart goal* G'_0 *and answer substitution θ, s.t.* $'\theta$ *and* $G_0 \quad G'_0\theta$.

Theorem 15 (Completeness for finitely recursive programs). *Let P be a finitely recursive program. Suppose* CounterSupp *is complete w.r.t. P and that for some grounding substitution γ,* $(\quad G \quad H)\gamma$ *holds in all the stable models of P. Then* $(G \quad H)$ *has a successful skeptical derivation from P and* CounterSupp *with restart goal G and some answer substitution θ more general than γ.*

Proof. By Theorems 2 and 9, there exists a smooth module sequence for P with finite elements $P_1, P_2, \ldots$, and a finite k s.t. $(\quad G \quad H)\gamma$ holds in all the stable models of P_k.

Since each P_i is downward closed, the ground supports of any given $A \quad atom(P_k)$ w.r.t. program P_k coincide with the ground supports of A w.r.t. the entire program P. Consequently, also ground counter-supports and (generalized) counter-supports, respectively, coincide in P_k and P. Therefore, CounterSupp is complete w.r.t. P_k, too. As a consequence, since P_k is a ground, finite program, the ground completeness theorem can be applied to conclude that $(G \quad H)\gamma$ has a successful skeptical derivation from P_k and $\mathsf{CounterSupp}^g$ with restart goal $G\gamma$. The same derivation is also a derivation from P (as $P_k \subseteq \mathsf{Ground}(P)$) and $\mathsf{CounterSupp}^g$. Then, by the Lifting lemma, $(G \quad H)$ has a successful skeptical derivation from P and CounterSupp, with restart goal G and some answer substitution θ, s.t. $(G \quad H)\gamma$ is an instance of $(G \quad H)\theta$. It follows that θ is more general than γ.

6 Conclusions

In this paper we have shown some important properties of the class of finitely recursive programs, a very expressive fragment of logic programs under the stable model semantics. Finitely recursive programs extend the class of finitary programs by dropping the restrictions on odd-cycles. We extended to finitely recursive programs many of the nice properties of finitary programs: (i) a compactness property (Theorem 7); (ii) the r.e.-completeness of inconsistency checking and skeptical inference (Theorem 11); (iii) the completeness of skeptical resolution (Theorem 15).

Unfortunately, some of the nice properties of finitary programs do *not* carry over to finitely recursive programs: (i) ground queries are not decidable (Theorem 12 and Corollary 2); (ii) nonground credulous queries are not semidecidable (as ground queries are co-r.e.).

As a side benefit, our techniques introduced a normal form for splitting and module sequences, where sequence length is limited to and—if the program is finitely recursive—the sequence is smooth (i.e., the "delta" between each non-limit element and its predecessor is finite). Such properties constitute an alternative characterization of finitely recursive programs. Moreover, if the module sequence of a finitely recursive program is consistent, then the whole program is (this result extends the splitting sequence theorem).

Acknowledgements

This work is partially supported by the PRIN project *Enhancement and Applications of Disjunctive Logic Programming*, funded by the Italian Ministry of Research (MIUR).

References

1. Apt, K.R.: Introduction to Logic Programming. In: van Leeuwen, J. (ed.) Handbook of Theoretical Computer Science. Formal Model and Semantics, vol. B, pp. 495–574. Elsevier, Amsterdam and The MIT Press, Cambridge (1990)
2. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press, Cambridge (2003)

3. Bonatti, P.A.: Resolution for skeptical stable model semantics. J. Autom. Reasoning 27(4), 391–421 (2001)
4. Bonatti, P.A.: Reasoning with infinite stable models. Artif. Intell. 156(1), 75–111 (2004)
5. Eiter, T., Leone, N., Mateis, C., Pfeifer, G., Scarcello, F.: A deductive system for nonmonotonic reasoning. In: Fuhrbach, U., Dix, J., Nerode, A. (eds.) LPNMR 1997. LNCS, vol. 1265, pp. 364–375. Springer, Heidelberg (1997)
6. Van Emden, M.H., Kowalski, R.A.: The semantics of predicate logic as a programming language. J. ACM 23(4), 733–742 (1976)
7. Fages, F.: Consistency of Clark's completion and existence of stable models. Methods of Logic in Computer Science 1, 51–60 (1994)
8. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: Proc. of the 5th ICLP, pp. 1070–1080. MIT Press, Cambridge (1988)
9. Gelfond, M., Lifschitz, V.: Classical negation in logic programs and disjunctive databases. New Generation Computing 9(3-4), 365–386 (1991)
10. Lifschitz, V., Turner, H.: Splitting a Logic Program. In: Proceedings of the 12th International Conference on Logic Programming, Kanagawa 1995. MIT Press Series Logic Program, pp. 581–595. MIT Press, Cambridge (1995)
11. Lifschitz, V., Turner, H.: Splitting a logic program. In: International Conference on Logic Programming, pp. 23–37 (1994)
12. Lloyd, J.W.: Foundations of Logic Programming, 1st edn. Springer, Heidelberg (1984)
13. Marek, V.W., Remmel, J.B.: On the expressibility of stable logic programming. In: Eiter, T., Faber, W., Truszczyński, M. (eds.) LPNMR 2001. LNCS (LNAI), vol. 2173, pp. 107–120. Springer, Heidelberg (2001)
14. Niemelä, I., Simons, P.: Smodels — an implementation of the stable model and well-founded semantics for normal lp. In: Fuhrbach, U., Dix, J., Nerode, A. (eds.) LPNMR 1997. LNCS, vol. 1265, pp. 421–430. Springer, Heidelberg (1997)
15. Turner, H.: Splitting a default theory. In: Shrobe, H., Senator, T. (eds.) Proceedings of the Thirteenth National Conference on Artificial Intelligence, pp. 645–651. AAAI Press, Menlo Park, California (1996)

Minimal Logic Programs

Pedro Cabalar[1], David Pearce[2], and Agustín Valverde[3,⋆]

[1] Corunna University Corunna, Spain
cabalar@udc.es
[2] Universidad Rey Juan Carlos Madrid, Spain
davidandrew.pearce@urjc.es
[3] University of Málaga Málaga, Spain
a_valverde@ctima.uma.es

Abstract. We consider the problem of obtaining a minimal logic program strongly equivalent (under the stable models semantics) to a given arbitrary propositional theory. We propose a method consisting in the generation of the set of prime implicates of the original theory, starting from its set of countermodels (in the logic of Here-and-There), in a similar vein to the Quine-McCluskey method for minimisation of boolean functions. As a side result, we also provide several results about *fundamental* rules (those that are not tautologies and do not contain redundant literals) which are combined to build the minimal programs. In particular, we characterise their form, their corresponding sets of countermodels, as well as necessary and sufficient conditions for entailment and equivalence among them.

Keywords: logic programming, answer set programming, minimisation of boolean and multivalued functions.

1 Introduction

The nonmonotonic formalism of *Equilibrium Logic* [1], based on the nonclassical logic of *here-and-there* (henceforth: HT), yields a logical characterisation for the *answer set* semantics of logic programs [2]. By capturing concepts such as the strong equivalence of programs and providing a means to generalise all previous extensions of answer set semantics, HT and equilibrium logic provide a useful foundation for Answer Set Programming (ASP) (see [3,4]). There has recently been an increasing interest in ASP in the expressiveness of logic programs and the analysis of program rules in a more logical setting. Properties of Equilibrium Logic have allowed, for instance, the extension of the traditional definition of answer set (now equivalent to the concept of equilibrium model) to the most general syntax of arbitrary propositional [5] and first order [6] theories.

In the case of propositional theories (of crucial importance for current ASP solvers), in [7] it was actually shown that any arbitrary theory in Equilibrium Logic is strongly equivalent to (that is, can always be replaced by) a disjunctive logic program possibly allowing default negation in the head. In this way,

⋆ This research was partially supported by Spanish MEC coordinated project TIN-2006-15455-C03, subprojects 01, 02, 03.

V. Dahl and I. Niemelä (Eds.): ICLP 2007, LNCS 4670, pp. 104–118, 2007.

this type of rule constitutes a *normal form* for general ASP when dealing with arbitrary theories. Aside from its interest as an expressiveness result, as [5] has shown, the important concept of *aggregate* in ASP [8] can be captured by formulas with nested implications, making such syntactic extensions of practical value. In addition, the problem of generating a program from an arbitrary theory, or merely from another logic program, immediately raises the question of how good is this generation. A study of the complexity and efficiency of translation methods from arbitrary theories to logic programs was already carried out in [9]. A different and perhaps more practically relevant topic has to do with the quality or simplicity of the resulting program, rather than the effort required to obtain it. In ASP various methods, including equilibrium logic, have been used to analyse program simplification, both for ground programs [10,11,12,13] and more recently for programs with variables [14,15].

In this paper we consider the problem of generating a minimal or simpler logic program strongly equivalent to some initial propositional theory (what includes, of course, the case of a logic program). We propose a method that follows steps similar to the Quine-McCluskey algorithm [16,17], well-known from the problem of minimising boolean functions. Our algorithm computes the set of prime implicates of the original theory starting from its set of countermodels in HT. In a second step, a minimal set of prime implicates is selected to cover the whole set of countermodels. Obviously, these two steps mean a considerable computational cost whose reward is the guarantee of syntactic minimality of the obtained program, something not achieved before, to the best of our knowledge.

As we will discuss later in Section 7, the interest of such a minimisation method lies in its two main potential application areas: (i) it may become a useful tool for theoretical research, helping to find minimal logic program patterns obtained from translations of other constructions; and (ii) what possibly has a greater practical impact, it allows an offline minimisation of a ground logic program, which can be perhaps later (re-)used in several different contexts. Apart from the method itself, the paper also provides side results of additional interest from the viewpoint of expressiveness. We identify the normal form of *fundamental* rules, used to conform the minimal programs. In this way, these rules are non-trivial in the sense that they are not tautological and do not contain redundant literals. The paper characterises the set of countermodels of any fundamental rule and provides necessary and sufficient conditions (see Section 6) for capturing entailment and equivalence among them.

The paper is organised as follows. Section 2 recalls the Quine-McCluskey algorithm for minimising boolean functions. Next, Section 3 contains a brief overview of Equilibrium Logic, including some basic definitions about logic programs. Section 4 presents the algorithm for obtaining prime implicates and the next section contains a small example. Then, Section 6 presents an alternative definition of minimal program (based on semantic entailment) and finally, Section 7 concludes the paper. Proofs have been included in the Appendix of an extended version of this document [18].

2 Quine-McCluskey Algorithm

The *Quine-McCluskey algorithm* [16,17] allows obtaining a minimal normal form equivalent to any classical propositional theory Γ. The algorithm is dual in the sense that it can be equally used to obtain a minimal DNF from the set of models of Γ or to obtain a minimal CNF from its set of countermodels. Although the former is the most popular version, we will actually focus on the latter for the sake of comparison. After all, as shown in [7], logic programs constitute a CNF for HT. The Quine-McCluskey algorithm computes the prime implicates of a theory Γ starting from its countermodels. To get a minimal CNF, a second algorithm (typically, *Petrick's method* [19]) must be used afterwards to select a minimal subset of prime implicates that suffice to cover all countermodels of Γ. In what follows, we skip many definitions from classical propositional logic assuming the reader's familiarity.

Let At be a set of atoms (called the propositional *signature*). We represent a classical propositional interpretation as a set of atoms $I \subseteq At$ selecting those assigned truth value 1 (true). As usual, a *literal* is an atom p (*positive* literal) or its negation $\neg p$ (*negative* literal). A *clause* is a disjunction of literals and a *formula in CNF* is a conjunction of clauses. The empty disjunction and conjunction respectively correspond to $\bot$ and $\top$. We will use letters $C, D, \ldots$ to denote clauses. Satisfaction of formulas is defined in the usual way. A clause is said to be *fundamental* if it contains no repeated atom occurrences. Non-fundamental clauses are irrelevant: any repeated literal can be just removed, and any clause containing $p \vee \neg p$ too, since it is a tautology. We say that a clause C *subsumes* a clause D, written $C \subseteq D$, iff all literals of C occur in D.

Proposition 1. *Let C, D be fundamental clauses. Then, $C \subseteq D$ iff $\models C \rightarrow D$.*

Proposition 2. *An interpretation I is a countermodel of a fundamental clause C iff $p \notin I$ for all positive literals p occurring in C and $p \in I$ for all negative literals $\neg p$ occurring in C.*

Proposition 2 provides a compact way to represent a fundamental clause C (and its set of countermodels). We can just define a *labelling*, let us write it $\overrightarrow{C}$, that for each atom $p \in At$ contains a pair:

- $(p, 1) \in \overrightarrow{C}$ when $\neg p$ occurs in C (meaning p true in all countermodels),
- $(p, 0) \in \overrightarrow{C}$ when p occurs as positive literal in C (meaning p false in all countermodels) or
- $(p, \text{-}) \in \overrightarrow{C}$ if p does not occur in C (meaning that the value of p is indifferent).

Note that it is also possible to retrieve the fundamental clause C corresponding to an arbitrary labelling $\overrightarrow{C}$, so the relation is one-to-one. Typically, we further omit the atom names (assuming alphabetical ordering) and we just write $\overrightarrow{C}$ as a vector of labels. As an example, given the ordered signature $\{p, q, r, s\}$ and the clause $C = \neg p \vee q \vee \neg s$, we would have $\overrightarrow{C} = \{(p, 1), (q, 0), (r, \text{-}), (s, 1)\}$ or

simply $\overrightarrow{C}$ = 10-1. When all atoms in the signature occur in a fundamental clause, the latter is said to be *developed*. Proposition 2 implies that a developed clause C has a single countermodel – note that in this case $\overrightarrow{C}$ does not contain indifference symbols. By abuse of notation, we identify a countermodel I with its corresponding developed clause $\{(p,1) \mid p \in I\} \cup \{(p,0) \mid p \in At \setminus I\}$.

An *implicate* C of a theory Γ is any fundamental clause satisfying $\Gamma \models C$. Clearly, countermodels of C are countermodels of Γ too. As any CNF formula Π equivalent to Γ will consist of implicates of Γ, the task of finding Π can be seen as a search of a set of implicates whose countermodels suffice to *comprise* the whole set of countermodels of Γ. However, if we want Π to be as simple as possible, we will be interested in implicates with minimal size. An implicate of Γ is said to be *prime* iff it is not subsumed by another implicate of Γ. The Quine-McCluskey algorithm computes all implicates of Γ by collecting their labelling vectors in successive sets S_i. In fact, at each step, S_i collects all the vectors with i indifference labels, starting with $i = 0$ indifferences (i.e. with S_0 equal to the set of countermodels of Γ). To compute the next set S_{i+1} the algorithm groups pairs of implicate vectors $\overrightarrow{C}, \overrightarrow{D}$ in S_i that just differ in one atom p, say, for instance $(p,1) \in \overrightarrow{C}$ and $(p,0) \in \overrightarrow{D}$. Then, it inserts a new vector $\overrightarrow{E} = \big(\overrightarrow{C} \setminus \{(p,1)\}\big) \cup \{(p,\text{-})\}$ in S_{i+1} and marks both $\overrightarrow{C}$ and $\overrightarrow{D}$ as non-prime (it is easy to see that clauses C, D are both subsumed by the new implicate E). When no new vector is formed, the algorithm finishes, returning the set of non-marked (i.e., prime) implicates.

Example 1. The table in Figure 2 schematically shows the result of applying the algorithm for a signature $At = \{p, q, r, s\}$ and starting from a set of countermodels shown in column S_0. Numbers between parentheses represent the countermodels of a given implicate (using the decimal representation of their vector of labels). The horizontal lines separate the implicates in each S_i by their number of 1's. Finally, '*' marks mean that the implicate has been subsumed by another one. The prime (i.e., non-marked) implicates are therefore: -100 ($\neg q \vee r \vee s$), 10-- ($\neg p \vee q$), 1--0 ($\neg p \vee s$) and 1-1- ($\neg p \vee \neg r$).

3 Equilibrium Logic

We begin defining the (monotonic) logic of *here-and-there* (HT). A *formula* is defined in the usual way as a well-formed combination of the operators $\bot, \wedge, \vee, \rightarrow$ with atoms in a propositional signature At. We define $\neg\varphi \stackrel{\text{def}}{=} \varphi \rightarrow \bot$ and $\top \stackrel{\text{def}}{=} \neg\bot$. As usual, by *theory* we mean a set of formulas.

An *HT-interpretation* is a pair $\langle H, T\rangle$ of sets of atoms with $H \subseteq T$. When $H = T$ the HT-interpretation is said to be *total*. We recursively define when $\langle H, T\rangle$ *satisfies* a formula φ, written $\langle H, T\rangle \models \varphi$ as follows:

- $\langle H, T\rangle \not\models \bot$
- $\langle H, T\rangle \models p$ if $p \in H$, for any atom $p \in At$
- $\langle H, T\rangle \models \varphi \wedge \psi$ if $\langle H, T\rangle \models \varphi$ and $\langle H, T\rangle \models \psi$

S_0		S_1		S_2	
		-100	$(4,12)$		
$*$ 0100	(4)	$*$ 100-	$(8,9)$		
$*$ 1000	(8)	$*$ 10-0	$(8,10)$		
$*$ 1001	(9)	$*$ 1-00	$(8,12)$		
$*$ 1010	(10)	$*$ 10-1	$(9,11)$	10--	$(8,9,10,11)$
$*$ 1100	(12)	$*$ 101-	$(10,11)$	1--0	$(8,10,12,14)$
$*$ 1011	(11)	$*$ 1-10	$(10,14)$	1-1-	$(10,11,14,15)$
$*$ 1110	(14)	$*$ 11-0	$(12,14)$		
$*$ 1111	(15)	$*$ 1-11	$(11,15)$		
		$*$ 111-	$(14,15)$		

Fig. 1. Example of computation of Quine McCluskey algorithm

- $\langle H,T\rangle \models \varphi \vee \psi$ if $\langle H,T\rangle \models \varphi$ or $\langle H,T\rangle \models \psi$
- $\langle H,T\rangle \models \varphi \rightarrow \psi$ if both (i) $\langle H,T\rangle \not\models \varphi$ or $\langle H,T\rangle \models \psi$; and (ii) $T \models \varphi \rightarrow \psi$ (in classical logic).

An HT-interpretation is a *model* of a theory Γ if it satisfies all formulas in Γ. As usual, we say that a theory Γ *entails* a formula φ, written $\Gamma \models \varphi$, when any model of Γ is a model of φ. A formula true in all models is said to be *valid* or a *tautology*. An *equilibrium model* of a theory Γ is any total model $\langle T,T\rangle$ of Γ such that no $\langle H,T\rangle$ with $H \subset T$ is model of Γ. *Equilibrium Logic* is the logic induced by equilibrium models.

An alternative definition of HT can be given in terms of a three-valued semantics (Gödel's three-valued logic). Under this viewpoint, an HT-interpretation $M = \langle H,T\rangle$ can be seen as a three-valued mapping $M : At \rightarrow \{0,1,2\}$ where, for any atom p, $M(p) = 2$ if $p \in H$ (p is *true*), $M(p) = 0$ if $p \in At \setminus T$ (p is *false*) and $M(p) = 1$ if $p \in T \setminus H$ (p is *undefined*). The valuation of formulas is defined so that the valuation of conjunction (resp. disjunction) is the minimum (resp. maximum) value, $M(\bot) = 0$ and $M(\varphi \rightarrow \psi) = 2$ if $M(\varphi) \leq M(\psi)$ or $M(\varphi \rightarrow \psi) = M(\psi)$ otherwise. Finally, models of a theory Γ are captured by those HT-interpretations M such that $M(\varphi) = 2$ for all $\varphi \in \Gamma$.

Lemma 1. *In HT, the following formulas are valid:*

$$\alpha \wedge \varphi \wedge \neg\varphi \rightarrow \beta \tag{1}$$

$$\alpha \wedge \varphi \rightarrow \beta \vee \varphi \tag{2}$$

$$(\alpha \wedge \varphi \rightarrow \beta \vee \neg\varphi) \leftrightarrow (\alpha \wedge \varphi \rightarrow \beta) \tag{3}$$

$$(\alpha \wedge \neg\varphi \rightarrow \beta \vee \varphi) \leftrightarrow (\alpha \wedge \neg\varphi \rightarrow \beta) \tag{4}$$

$$(\alpha \rightarrow \beta) \rightarrow (\alpha \wedge \gamma \rightarrow \beta) \tag{5}$$

$$(\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \beta \vee \gamma) \tag{6}$$

The following property of HT will have important consequences for adapting the Quine-McCluskey method to this logic:

Property 1. For any theory Γ, $\langle H, T\rangle \models \Gamma$ implies $\langle T, T\rangle \models \Gamma$.

We can rephrase the property in terms of countermodels: if $\langle T, T\rangle$ is a countermodel of Γ then any $\langle H, T\rangle$ will also be a countermodel. As a result, contrarily to what happened in classical logic (or in Łukasiewicz three-valued logic, for instance), we cannot use an *arbitrary* set S of HT-interpretations to represent the countermodels of some theory. Let us define the *total-closure* (*t-closure*) of a set of interpretations S as:

$$S^t \stackrel{\text{def}}{=} S \cup \{\ \langle H, T\rangle \mid \langle T, T\rangle \in S,\ H \subseteq T\ \}$$

We say that S is *t-closed* if $S^t = S$.

Property 2 (Theorem 2 in [7]). Each t-closed set S of interpretations is the set of countermodels of a logic program.

This fact was used in [7] to compute the number of possible HT-theories (modulo semantic equivalence) for a finite number n of atoms by counting the possible t-closed sets of interpretations. The resulting amount happens to be considerably smaller than 2^{3^n}, which corresponds, for instance, to the number of possible theories we can form in Łukasiewicz logic.

3.1 Logic Programs

A *logic program* is a conjunction of clauses (also called *rules*) of the form:

$$a_1 \wedge \cdots \wedge a_n \wedge \neg b_1 \wedge \cdots \wedge \neg b_m \rightarrow c_1 \vee \cdots \vee c_h \vee \neg d_1 \vee \cdots \vee \neg d_k \qquad (7)$$

with $n, m, h, k \geq 0$ and all subindexed letters representing atoms. As before, empty disjunctions stand for $\bot$ while empty conjunctions stand for $\top$. The *empty rule* would therefore correspond to $\top \rightarrow \bot$ or simply $\bot$. We will use letters r, r', ... to denote rules. We will sometimes abbreviate a rule r like (7) using the expression:

$$B_r^+ \wedge \neg B_r^- \rightarrow Hd_r^+ \vee \neg Hd_r^-$$

where B_r^+ (resp. B_r^-) denotes the sets of atoms occurring positively (resp. negatively) in the antecedent or body of r, while Hd_r^+ (resp. Hd_r^-) denotes the sets of atoms occurring positively (resp. negatively) in the consequent or head of r.

For sets of rules of form (7), equilibrium models and answer sets coincide, [3]. Two rules r, r' are said to be *strongly equivalent*, in symbols $r \equiv_s r'$, if for any set Π of rules, $\Pi \cup \{r\}$ and $\Pi \cup \{r'\}$ have the same equilibrium models; strong equivalence for sets of rules is defined analogously. Rules (and sets of rules) are strongly equivalent iff they are logically equivalent in HT, i.e. they have the same HT-models, [3]. Hence, strongly equivalent rules can be interchanged without loss in any context, and deduction in HT provides a key instrument for replacing rules by equivalent, simpler ones.

As happened with non-fundamental clauses in classical logic, some rules may contain irrelevant or redundant information. To avoid this, we define:

Definition 1 (fundamental rule). *A rule r is said to be* fundamental *when all pairwise intersections of sets $Hd_r^+, Hd_r^-, B_r^+, B_r^-$ are empty, with the possible exception of $Hd_r^+ \cap Hd_r^-$.*

Lemma 2. *For any rule r:*

i) If one of the intersections $B_r^+ \cap B_r^-$, $B_r^+ \cap Hd_r^+$ and $B_r^- \cap Hd_r^-$ is not empty, then r is a tautology.

ii) Otherwise, r is strongly equivalent to the fundamental rule:

$$r' : B_r^+ \wedge \neg B_r^- \rightarrow (Hd_r^+ \setminus B_r^-) \vee \neg(Hd_r^- \setminus B_r^+).$$

The proof follows from (1)-(4). Lemma 2 plus the main result in [7] implies

Theorem 1. *Fundamental rules constitute a normal form for the logic of HT.*

4 Generation of Prime Implicates

As can be imagined, our goal of obtaining minimal programs will depend in a crucial way on how we define the relation *simpler than* between two programs, or more specifically, between two rules. To this aim, we can either rely on a syntactic or on a semantic definition. For classical logic, Proposition 1 shows that both choices are interchangeable: we can either use the inclusion of literals among fundamental clauses $C \subseteq D$ (a syntactic criterion) or just use entailment $C \models D$ instead. In the case of HT, however, we will see later that this correspondence is not preserved in a direct way. For this reason, we maintain two different concepts of *smaller rule* in HT: *entailment*, written $r \models r'$ with its usual meaning; and *subsumption*, written $r \subseteq r'$ and defined below.

Definition 2 (subsumption). *A rule r* subsumes *another rule r', written $r \subseteq r'$, when $Hd_r^+ \subseteq Hd_{r'}^+$, $Hd_r^- \subseteq Hd_{r'}^-$, $B_r^+ \subseteq B_{r'}^+$ and $B_r^- \subseteq B_{r'}^-$.*

Rule subsumption $r \subseteq r'$ captures the idea that r results from removing some literals in r'. It follows that the empty rule $\bot$ subsumes all rules. As usual, we handle the strict versions of subsumption, written $r \subset r'$ and meaning both $r \subseteq r'$ and $r \neq r'$, and of entailment, written $r \models\!\!\!\not\, r'$ and meaning that $r \models r'$ but $r' \not\models r$. From (5),(6) we conclude that subsumption is stronger than entailment:

Theorem 2. *For any pair of rules r, r', if $r \subseteq r'$ then $r \models r'$.*

However, in contrast to Proposition 1 for classical logic, in HT the converse direction does not hold. As a counterexample:

Example 2. Given rules $r : p \rightarrow q$ and $r' : p \wedge \neg q \rightarrow \bot$ we have $r \models r'$ but $r \not\subseteq r'$.

In the rest of this section we will focus on syntactic subsumption, providing later (Section 6) a variation that uses semantic entailment instead.

Definition 3 (Syntactically simpler program). *We say that program Π is* syntactically simpler *than program Π', written $\Pi \preceq \Pi'$, if there exists some $\Gamma \subseteq \Pi'$ such that Π results from replacing each rule $r' \in \Gamma$ by some r, $r \subseteq r'$.*

In other words, we might remove some rules from Π' and, from the remaining rules, we might remove some literals at any position. Note that $\Pi \subseteq \Pi'$ implies $\Pi \preceq \Pi'$ but the converse implication does not hold, and that, of course, syntactically simpler does not generally entail any kind of semantic relation.

Theorem 3. *Let Π be a $\preceq$-minimal logic program among those strongly equivalent to some theory Γ. Then Π consists of fundamental rules.*

Definition 4 (implicate/prime implicate). *A fundamental rule r is an* implicate *of a theory Γ iff $\Gamma \models r$. Moreover, r is said to be* prime *iff it is not strictly subsumed by another implicate of Γ.*

To sum up, we face again the same setting as in classical logic: to find a $\preceq$-minimal program Π equivalent to some theory Γ means to obtain a set of prime implicates that cover all countermodels of Γ. Therefore, it is crucial that we are able to characterise the countermodels of fundamental rules, as we did with Proposition 2 in the classical case. Given a fundamental rule r, we define the set of HT-interpretations $CMs(r) \stackrel{\text{def}}{=}$

$$\left\{ \langle H, T\rangle \;\middle|\; B_r^+ \subseteq H,\ B_r^- \cap T = \emptyset,\ Hd_r^+ \cap H = \emptyset,\ Hd_r^- \subseteq T \right\}$$

Theorem 4. *Given a fundamental rule r and an HT-interpretation M: $M \not\models r$ iff $M \in CMs(r)^t$.*

Theorem 4 and the fact that $CMs(r)$ is never empty leads to:

Observation 1. *Fundamental rules always have countermodels.*

Given two fundamental rules r, r' we say that r *covers* r' when $CMs(r') \subseteq CMs(r)$. Notice that this definition of *covering* is stronger than entailment: if $CMs(r') \subseteq CMs(r)$ then the same still holds for $CMs(r')^t \subseteq CMs(r)^t$ and from Theorem 4 we conclude $r \models r'$. On the other hand, it is very easy to see that covering is a weaker condition than subsumption:

Proposition 3. *If $r \subseteq r'$ then $CMs(r') \subseteq CMs(r)$.*

As with classical clauses, the countermodels in $CMs(r)$ can also be compactly described by a mapping of the set of atoms into a set of labels (which contains more elements now). To do so, note first that the definition of $CMs(r)$ can be directly rephrased in terms of three-valued interpretations as follows: $M \in CMs(r)$ iff the following conditions hold for any atom p: (i) $p \in B_r^+ \Rightarrow M(p) = 2$; (ii) $p \in B_r^- \Rightarrow M(p) = 0$; (iii) $p \in Hd_r^+ \Rightarrow M(p) \neq 2$; and (iv) $p \in Hd_r^- \Rightarrow M(p) \neq 0$. As r is a fundamental rule, the last two conditions are not mutually exclusive. Thus, when $p \in Hd_r^+ \cap Hd_r^-$ we would just have $M(p) = 1$.

Definition 5 (Rule labelling). *Given a fundamental rule r, we define its* labelling $\overrightarrow{r}$ *as a set containing, for each atom $p \in At$, a pair:*

$(p, 2)$ *if* $p \in B_r^+$	$(p, \bar{2})$ *if* $p \in Hd_r^+ \setminus Hd_r^-$
$(p, 0)$ *if* $p \in B_r^-$	$(p, \bar{0})$ *if* $p \in Hd_r^- \setminus Hd_r^+$
$(p, 1)$ *if* $p \in Hd_r^+ \cap Hd_r^-$	$(p, \text{-})$ *if p does not occur in r*

Note that $(p, \bar{2})$ stands for $M(p) \neq 2$ whereas $(p, \bar{0})$ stands for $M(p) \neq 0$. We will sometimes write $\overrightarrow{r}(p) = v$ when $(p, v) \in \overrightarrow{r}$ and we also use the abbreviation $\overrightarrow{r}|_{\neq p}$ to stand for $\{(q, v) \in \overrightarrow{r} \mid q \neq p\}$. As an example of labelling, given the signature $At = \{a, b, c, d, e, p, q\}$, the fundamental clause $r : a \wedge b \wedge \neg d \rightarrow e \vee p \vee \neg p \vee \neg q$ would correspond to $\overrightarrow{r} = 22\text{-}0\bar{2}1\bar{0}$. In fact, there actually exists a one-to-one correspondence between a fundamental rule and its labelling, i.e., we can get back the rule r from $\overrightarrow{r}$. Thus, the set of countermodels $CMs(r)$, or its corresponding labelling $\overrightarrow{r}$, can be used to univocally represent the original rule r. It is important to remember, however, that $CMs(r)$ *is not* the set of countermodels of r – by Theorem 4, the latter actually corresponds to $CMs(r)^t$.

A single countermodel $M = \langle H, T \rangle$ can also be seen as a labelling that assigns a label in $\{0, 1, 2\}$ to each atom in the signature. Using the above definitions, the rule[1] r_M corresponding to M would be: $B^+_{r_M} = H$, $B^-_{r_M} = At \setminus T$ and $Hd^+_{r_M} = Hd^-_{r_M} = T \setminus H$. It is easy to see that rules derived from countermodels are $\preceq$-*maximal* – in fact, there is no way to construct *any* fundamental rule strictly subsumed by r_M. These implicates will constitute the starting set S_0 of the algorithm, which will generate new implicates that always subsume at least one of those previously obtained. The proposed algorithm (Generation of Prime Implicates, GPI) is shown in Table (a) of Figure 4. The algorithm applies three basic matching steps to implicates whose labels just differ in one atom p. Note that the possible matches would be much more than three, but only three cases i), ii) and iii) yield any effect. To understand the purpose behind these three operations, consider the form of the corresponding involved fundamental rules. Using the correspondence between labelling and rule we obtain Table (b) in Figure 4.

Proposition 4. *Let α, β and p be arbitrary propositional formulas. For the three cases i), ii), iii) in Table (b) Figure 4, the equivalence $r'' \leftrightarrow r \wedge r'$ is an HT-valid formula.*

Table (b) in Figure 4 also explains the way in which the algorithm assigns the non-prime marks. For instance, in cases i) and ii) only r is subsumed by the new generated rule r'' and so, GPI leaves r' without being marked in the current step $i + 1$. However, in case iii) the obtained rule r'' subsumes not only r and r' but also $\alpha \wedge \neg p \rightarrow \beta$ and $\alpha \wedge p \rightarrow \beta$ which were left previously unmarked, and must therefore be marked now too.

To prove termination and correctness of the algorithm, we will provide a pair of useful definitions. Given a label $v \in \{0, 1, 2, \bar{2}, \bar{0}, \text{-}\}$ we define its *weight*, $w(v)$, as a value in $\{0, 1, 2\}$ specified as follows: $w(0) = w(1) = w(2) = 0$, $w(\bar{2}) = w(\bar{0}) = 1$ and $w(\text{-}) = 2$. Intuitively, $w(v)$ points out the number of matches like i), ii) and iii) in the algorithm required to obtain label v. The *weight* of a fundamental rule r, $w(r)$, is defined as the sum of the weights $w(v)$ of all labels in $\overrightarrow{r}$, that is, $w(r) \stackrel{\text{def}}{=} \Sigma_{p \in At}\ w(\overrightarrow{r}(p))$. To put an example, given $\overrightarrow{r} = 22\text{-}0\bar{2}1\bar{0}$, $w(r) = 0 + 0 + 2 + 0 + 1 + 0 + 1 = 4$. Clearly, there exists a

[1] In fact, constructing a rule per each countermodel was one of the techniques used in [7] to show that any arbitrary theory is equivalent to a set of rules.

$S_0 := \{\overrightarrow{r_M} \mid M \not\models \Gamma\}$;
$i := 0$;
while $S_i \neq \emptyset$ **do**
  $S_{i+1} := \emptyset$;
  for each $\overrightarrow{r}, \overrightarrow{r}' \in S_i$ **such that** $\overrightarrow{r} \neq \overrightarrow{r}'$ and $\overrightarrow{r}|_{\neq p} = \overrightarrow{r}'|_{\neq p}$ for some p **do**
    case i): $(p,1) \in \overrightarrow{r}$ and $(p,0) \in \overrightarrow{r}'$ **then**
      Add $\overrightarrow{r}'' : \overrightarrow{r}|_{\neq p} \cup \{\ (p,\bar{2})\ \}$ to S_{i+1};
      Mark $\overrightarrow{r}$ as non-prime;
    case ii): $(p,1) \in \overrightarrow{r}$ and $(p,2) \in \overrightarrow{r}'$ **then**
      Add $\overrightarrow{r}'' : \overrightarrow{r}|_{\neq p} \cup \{\ (p,\bar{0})\ \}$ to S_{i+1};
      Mark $\overrightarrow{r}$ as non-prime;
    case iii): $(p,\bar{2}) \in \overrightarrow{r}$ and $(p,\bar{0}) \in \overrightarrow{r}'$ **then**
      Add $\overrightarrow{r}'' : \overrightarrow{r}|_{\neq p} \cup \{\ (p,\text{-})\ \}$ to S_{i+1};
      Mark $\overrightarrow{r}$, $\overrightarrow{r}'$, $\overrightarrow{r}|_{\neq p} \cup \{(p,0)\}$ and $\overrightarrow{r}|_{\neq p} \cup \{(p,2)\}$ as non-prime;
    end case
  end for each
  $i := i + 1$;
end while

(a) Pseudocode.

case i)	case ii)	case iii)
$r : \alpha \to \beta \vee p \vee \neg p$	$r : \alpha \to \beta \vee p \vee \neg p$	$r : \alpha \to \beta \vee p$
$r' : \alpha \wedge \neg p \to \beta$	$r' : \alpha \wedge p \to \beta$	$r' : \alpha \to \beta \vee \neg p$
$r'' : \alpha \to \beta \vee p$	$r'' : \alpha \to \beta \vee \neg p$	$r'' : \alpha \to \beta$

(b) Rule form of generated implicates.

Fig. 2. Algorithm for generation of prime-implicates (GPI)

maximum value for a rule weight which corresponds to assigning label '-' to all atoms, i.e., $2 \cdot |At|$.

Lemma 3. *Let $IMP_i(\Gamma)$ denote the set of implicates of Γ of weight i. Then, in the algorithm GPI, $S_i = IMP_i(\Gamma)$.*

Theorem 5. *Algorithm GPI stops after a finite number of steps $i = n$ and $\bigcup_{i=0\ldots n} S_i$ is the set of implicates of Γ.*

Theorem 6. *Let Γ be some arbitrary theory. The set of unmarked rules obtained by GPI is the set of prime implicates of Γ.*

5 Examples

To illustrate algorithm GPI, we begin considering its application to translate the theory $\Gamma = \{(\neg p \to q) \to p\}$ into a strongly equivalent minimal logic program. The set of countermodels is represented by the labels at the first column of the table below, assuming alphabetical ordering for atoms in $\{p, q\}$. For instance,

labelling 10 stands for $M(p) = 1$ and $M(q) = 0$, that is, $M = \langle \emptyset, \{p\} \rangle$ whereas 02 stands for $M'(p) = 0$ and $M'(q) = 2$, i.e., $M' = \langle \{q\}, \{q\} \rangle$. The remaining columns show all matches that take place when forming each remaining S_i. The marks '*' show those implicates that result marked at each match.

S_0	S_1	S_2
10	$10, *11 \longmapsto 1\bar{2}$	
02	$02, *12 \longmapsto \bar{2}\bar{2}$	$*1\bar{2}, *1\bar{0} \longmapsto$ 1- (mark $*10, *12$ too)
12	$02, *01 \longmapsto 0\bar{0}$	$\bar{2}2, *\bar{2}1 \longmapsto \bar{2}\bar{0}$
01	$12, *11 \longmapsto 1\bar{0}$	$0\bar{0}, *1\bar{0} \longmapsto \bar{2}\bar{0}$
11	$01, *11 \longmapsto \bar{2}1$	

Notice that implicate 01 in S_0 is not marked until we form -1 in S_2. Observe as well that implicate $\bar{0}\bar{2}$ is obtained in two different ways at that same step. Since no match can be formed at S_2 the algorithm stops at that point. If we carefully observe the previous table, the final unmarked labellings $\overrightarrow{r}$ (i.e., prime implicates r) are shown below:

$\overrightarrow{r}$	r	$CMs(r)$	$CMs(r)^t$
02	$\neg p \wedge q \rightarrow \bot$	02	02, 01
$\bar{2}\bar{2}$	$q \rightarrow p$	02, 12	02, 12, 01
$0\bar{0}$	$\neg p \rightarrow \neg q$	02, 01	02, 01
1-	$p \vee \neg p$	10, 11, 12	10, 11, 12
$\bar{2}\bar{0}$	$p \vee \neg q$	11, 12, 01, 02	11, 12, 01, 02

Finally, to obtain a minimal program we must select now a minimal set of implicates that covers all the countermodels of Γ. To this aim, any method from minimisation of boolean functions (like for instance, Petrick's method [19]) is still applicable here, as we just have a "coverage" problem and do not depend any more on the underlying logic. Without entering into details, the application of Petrick's method, for instance, would proceed to the construction of a chart:

	01	$\bar{2}1$	$0\bar{0}$	1-	$\bar{2}\bar{0}$
10				×	
02	×	×	×		×
12		×		×	×
01	×	×	×		×
11				×	×

Implicate 1- will be *essential*, as it is the only one covering countermodel 10. If we remove it plus the countermodels it covers we obtain the smaller chart:

	02	$\bar{2}2$	$0\bar{0}$	$\bar{2}\bar{0}$
02	×	×	×	×
01	×	×	×	×

that just points out that any of the remaining implicates can be used to form a minimal program equivalent to Γ. So, we get the final four $\preceq$-minimal programs:

$$\Pi_1 = \{p \vee \neg p,\ \neg p \wedge \neg q \rightarrow \bot\} \qquad \Pi_3 = \{p \vee \neg p,\ \neg p \rightarrow \neg q\}$$
$$\Pi_2 = \{p \vee \neg p,\ q \rightarrow p\} \qquad \Pi_4 = \{p \vee \neg p,\ \neg q \vee p\}$$

This example can be used to compare to transformations in [9] (for reducing arbitrary theories to logic programs) that applied on Γ yield program Π_2 plus the additional rule $\{p \vee \neg p \vee \neg q\}$ trivially subsumed by $p \vee \neg p$ in Π_2. Nevertheless, not all examples are so trivial. For instance, the theory $\{(\neg p \rightarrow q) \rightarrow (\neg p \rightarrow r)\}$ from Example 1 in [9] yielded in that case a translation consisting of the six rules $\{(q \wedge \neg p \rightarrow \bot), (q \rightarrow \neg r), (\neg p \rightarrow \neg p), (\neg p \vee \neg r), (\neg p \rightarrow \neg p \vee \neg q), (\neg p \vee \neg q \vee \neg r)\}$ that can be trivially reduced (after removing tautologies and subsumed rules) to $\{(q \wedge \neg p \rightarrow \bot), (q \rightarrow \neg r), (\neg p \vee \neg r)\}$ but which is not a minimal program. In fact, the GPI algorithm obtains (among others) the strongly equivalent minimal program $\{(q \wedge \neg p \rightarrow \bot), (\neg p \vee \neg r)\}$. Note that another important advantage is that the GPI method obtains *all* the possible minimal programs when several choices exist.

As an example of the use of GPI on logic programs, consider the case where we must combine two pieces of program from different knowledge sources. If we take, for instance $\Pi = \{(\neg r \wedge q \rightarrow p), (\neg p \rightarrow r \vee s)\}$ and $\Pi' = \{(r \rightarrow p), (\neg r \rightarrow q)\}$, these two programs are minimal when analysed independently, whereas none of the rules in $\Pi \cup \Pi'$ is subsumed by another in that set. After applying GPI to $\Pi \cup \Pi'$ however, we obtain that the unique minimal strongly equivalent program is just $\{(\neg r \rightarrow p), (r \rightarrow p), (\neg r \rightarrow q)\}$.

6 Entailment

Looking at the first example in the previous section, and particularly at the first implicates chart, it may be surprising to find that, although all of them are prime, some implicates entail others. The reason for this was explained before: contrarily to the classical case, (our definition of) syntactically simpler does not mean entailment in HT. Note, for instance, how all implicates excepting 1- are entailed by $\bar{2}\bar{0}$. We claim, however, that our criterion is still reasonable, as there is no clear reason why $\neg p \vee q$ should be syntactically simpler than $p \wedge \neg q \rightarrow \bot$ or $p \rightarrow q$. On the other hand, it seems that most imaginable criteria for syntactic simplicity should be probably stronger than our $\preceq$ relation. In this section we consider an alternative semantic definition of prime implicate that relies on the concept of entailment.

Definition 6 (s-prime implicate). *An implicate r of a theory Γ is said to be* semantically prime *(*s-prime *for short) if there is no implicate r' of Γ such that $r' \mid\!< r$.*

The only s-prime implicates in the example would be now 1- and $\bar{2}\bar{0}$. The intuition behind this variant is that, rather than focusing on syntactic simplicity, we are interested now in collecting a minimal set of rules that are as strong as possible, but considering rule by rule (remember that the program as a whole has to be strongly equivalent to the original theory). Since subsumption implies entailment, finding the s-prime implicates could be done as a postprocessing to

the GPI algorithm seen before (we would just remove some prime implicates that are not s-prime). Another, more efficient alternative would be a suitable modification of the algorithm to disregard entailed implicates from the very beginning. For space reasons, we do not enter into the details of this modification. What is perhaps more important is that in any of these two alternatives for computing s-prime implicates we can use the following simple syntactic characterisations of entailment and equivalence of rules.

Theorem 7. *For every rule r, and every fundamental rule r', $r \models r'$ iff the following conditions hold:*

1. $B_r^- \subseteq B_{r'}^-$
2. $Hd_r^- \subseteq Hd_{r'}^- \cup B_{r'}^+$
3. $B_r^+ \subseteq B_{r'}^+ \cup Hd_{r'}^-$
4. $Hd_r^+ \subseteq Hd_{r'}^+ \cup B_{r'}^-$
5. *Either* $B_r^+ \cap Hd_{r'}^- = \varnothing$ *or* $Hd_r^+ \cap Hd_{r'}^+ = \varnothing$.

For instance, it is easy to see that rules r and r' in Example 2 satisfy the above conditions. Theorem 7 leads to a characterisation of equivalence between fundamental rules.

Corollary 1. *If r and r' are fundamental rules, then: $r \equiv_s r'$ iff the following conditions hold:*

1. $B_r^- = B_{r'}^-$
2. $Hd_r^+ = Hd_{r'}^+$
3. $Hd_r^- \cup B_r^+ = Hd_{r'}^- \cup B_{r'}^+$
4. *If* $Hd_r^+ = Hd_{r'}^+ \neq \varnothing$, *then* $B_r^+ = B_{r'}^+$ *and* $Hd_r^- = Hd_{r'}^-$

So we can actually classify fundamental rules into two categories: if their positive head is not empty $Hd_r^+ \neq \varnothing$, then there is no other equivalent fundamental rule. On the other hand, if $Hd_r^+ = \varnothing$, then r can be called a *constraint*, since it is strongly equivalent to $B_r^+ \wedge Hd_r^- \wedge \neg B_r^- \rightarrow \bot$ but also to any rule that results from removing atoms from the constraint's positive body $B_r^+ \cup Hd_r^-$ and adding them negated in the head (this is called *shifting* in [20]). For instance:

$$p \wedge q \wedge \neg r \rightarrow \bot \;\equiv_s\; q \wedge \neg r \rightarrow \neg p \;\equiv_s\; p \wedge \neg r \rightarrow \neg q \;\equiv_s\; \neg r \rightarrow \neg p \vee \neg q$$

7 Discussion and Related Work

We provided a method for generating a minimal logic program strongly equivalent to some arbitrary propositional theory in equilibrium logic. We actually considered two alternatives: a syntactic one, exclusively treating programs with a smaller syntax in the sense of rule and literal occurrences; and a semantic one, in the sense of programs that make use of rules that are as deductively strong as possible.

Some results in the paper were known before or were previously given for more restrictive cases. For instance, Lemma 2 (in presence of Observation 1)

provides a necessary and sufficient condition for tautological rules. This becomes a confirmation using HT logic of Theorem 4.4 in [21]. On the other hand, our Theorem 7 is a generalisation of Theorem 6 in [13] for programs with negation in the head. Finally, the *shifting* operation we derived from Corollary 1 was introduced before in [20] (see Corollary 4.8 there) – in fact, we have been further able to show that it preserves *strong* equivalence.

As commented in the introduction, we outline two main potential applications for our method: as a help for theoretical research and as a tool for simplifying ground logic programs. In the first case, the method may become a valuable support when exploring a particular group or pattern of expressions. Consider for instance the translation into logic programs of other constructions (say aggregates, classes of complex expressions in arbitrary theories, high level action languages, etc) or even think about an hypothetical learning algorithm that must explore a family of rules to cover input examples. In all these cases, we may be reasonably interested in finding the smallest strongly equivalent representation of the set of rules handled. To put an example, the translation of a nested implication $(p \rightarrow q) \rightarrow r$ into the logic program $\{(q \rightarrow r), (\neg p \rightarrow r), (p \vee \neg q \vee r)\}$ found in [7] and of crucial importance in that paper, can be now shown, using our algorithm, to be the most compact (strongly equivalent) possible one.

Regarding the application of our method as a tool for simplifying ground logic programs, we may use the basic methodology proposed in [22] to clarify the situation. Three basic types of simplifications are identified, depending on whether the result is: (i) smaller in size (less atoms or rules, rules with less literals, etc); (ii) belonging to a simpler class (Horn clauses, normal programs, disjunctive programs, etc); and (iii) more efficient for a particular algorithm or solver. Apart from this classification, they also distinguish between online (i.e., embedded in the algorithm for computing answer sets) and offline simplifications. Our orientation in this paper has been completely focused on (offline) simplifications of type (i). This has a practical interest when we deal with program modules to be (re-)used in several different contexts, or for instance, with rules describing a transition in a planning scenario, as this piece of program is then replicated many times depending on the plan length. Adapting the algorithm to simplifications of type (ii) is left for future work: we can force the generation of a particular class of programs (for instance, by just disregarding implicates not belonging to that class), or use the method to show that this generation is not possible. Another topic for future study is the analysis of complexity.

References

1. Pearce, D.: Equilibrium logic. Ann. Math. Artif. Intell. 47, 3–41 (2006)
2. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: Kowalski, R.A., Bowen, K. (eds.) Proc. of the Fifth International Conference on Logic Programming, ICLP'88, Seattle, WA, USA, pp. 1070–1080. The MIT Press, Cambridge, Massachusetts (1988)
3. Lifschitz, V., Pearce, D., Valverde, A.: Strongly equivalent logic programs. ACM Transactions on Computational Logic 2(4), 526–541 (2001)

4. Lifschitz, V., Pearce, D., Valverde, A.: A characterization of strong equivalence for logic programs with variables. In: Baral, C., Brewka, G., Schlipf, J. (eds.) LPNMR 2007, LNCS (LNAI), vol. 4483. Springer, Heidelberg (2007)
5. Ferraris, P.: Answer sets for propositional theories. In: Baral, C., Greco, G., Leone, N., Terracina, G. (eds.) LPNMR 2005. LNCS(LNAI), vol. 3662, pp. 119–131. Springer, Heidelberg (2005)
6. Ferraris, P., Lee, J., Lifschitz, V.: A new perspective on stable models. In: Proc. of the Intl. Joint Conf. on Artificial Intelligence (IJCAI'07) (2007)
7. Cabalar, P., Ferraris, P.: Propositional theories are strongly equivalent to logic programs. Theory and Practice of Logic Programming (to appear, 2007)
8. Faber, W., Leone, N., Pfeifer, G.: Recursive aggregates in disjunctive logic programs: Semantics and complexity. In: Alferes, J.J., Leite, J.A. (eds.) JELIA 2004. LNCS(LNAI), vol. 3229, Springer, Heidelberg (2004)
9. Cabalar, P., Pearce, D., Valverde, A.: Reducing propositional theories in equilibrium logic to logic programs. In: Bento, C., Cardoso, A., Dias, G. (eds.) EPIA 2005. LNCS(LNAI), vol. 3808, pp. 4–17. Springer, Heidelberg (2005)
10. Osorio, M., Navarro, J.A., Arrazola, J.: Equivalence in answer set programming. In: Pettorossi, A. (ed.) LOPSTR 2001. LNCS, vol. 2372, pp. 57–75. Springer, Heidelberg (2002)
11. Eiter, T., Fink, M., Tompits, H., Woltran, S.: Simplifying logic programs under uniform and strong equivalence. In: Lifschitz, V., Niemelä, I. (eds.) Logic Programming and Nonmonotonic Reasoning. LNCS(LNAI), vol. 2923, pp. 87–99. Springer, Heidelberg (2003)
12. Pearce, D.: Simplifying logic programs under answer set semantics. In: Demoen, B., Lifschitz, V. (eds.) ICLP 2004. LNCS, vol. 3132, pp. 210–224. Springer, Heidelberg (2004)
13. Lin, F., Chen, Y.: Discovering classes of strongly equivalent logic programs. In: Proc. of the Intl. Joint Conf. on Artificial Intelligence (IJCAI'05), pp. 516–521 (2005)
14. Eiter, T., Fink, M., Tompits, H., Traxler, P., Woltran, S.: Replacements in non-ground answer-set programming. In: Proc. of KR'06, pp. 340–350. AAAI, Stanford (2006)
15. Fink, M., Pichler, R., Tompits, H., Woltran, S.: Complexity of rule redundancy in non-ground answer-set programming over finite domains. In: Baral, C., Brewka, G., Schlipf, J. (eds.) LPNMR 2007, LNCS (LNAI), vol. 4483. Springer, Heidelberg (2007)
16. Quine, W.V.O.: The problem of simplifying truth functions. American Mathematical Monthly 59, 521–531 (1952)
17. McCluskey, E.J.: Minimization of boolean functions. Bell System Technical Journal 35, 1417–1444 (1956)
18. Cabalar, P., Pearce, D., Valverde, A.: Minimal logic programs (extended report), Technical report (2007), available at `http://www.dc.fi.udc.es/~cabalar/minlp-ext.pdf`
19. Petrick, S.R.: A direct termination of the irredundant forms of a boolean function from the set of prime implicants. Technical Report AFCRC-TR-56-110, Air Force Cambridge Res. Center, Cambridge, MA (1956)
20. Inoue, K., Sakama, C.: Negation as failure in the head. Journal of Logic Programming 35(1), 39–78 (1998)
21. Inoue, K., Sakama, C.: Equivalence of logic programs under updates. In: Alferes, J.J., Leite, J.A. (eds.) JELIA 2004. LNCS (LNAI), vol. 3229, pp. 174–186. Springer, Heidelberg (2004)
22. Eiter, T., Fink, M., Tompits, H., Woltran, S.: Formal methods for comparing and optimizing nonmonotonic logic programs Research project (last updated 2007), web page `http://www.kr.tuwien.ac.at/research/eq.html`

Generic Tableaux for Answer Set Programming

Martin Gebser and Torsten Schaub*

Institut für Informatik, Universität Potsdam, August-Bebel-Str. 89, D-14482 Potsdam, Germany

Abstract. We provide a general and modular framework for describing inferences in Answer Set Programming (ASP) that aims at an easy incorporation of additional language constructs. To this end, we generalize previous work characterizing computations in ASP by means of tableau methods. We start with a very basic core fragment in which rule heads and bodies consist of atomic literals. We then gradually extend this setting by focusing on the concept of an aggregate, understood as an operation on a collection of entities. We exemplify our framework by applying it to conjunctions in rule bodies, cardinality constraints as used in *smodels*, and finally to disjunctions in rule heads.

1 Introduction

Answer Set Programming (ASP; [1]) has become an appealing tool for declarative problem solving. Unlike the related area of satisfiability checking (SAT; [2]), it however lacked a formal framework for describing inferences conducted by ASP solvers, such as the resolution proof theory in SAT [3]. This deficiency has led to a great heterogeneity in the description of ASP algorithms and has impeded their formal comparability. We addressed this problem in [4] by introducing a family of tableau calculi [5] for ASP. The idea is to view answer set computations as derivations in an inference system: A branch in a tableau corresponds to a successful or unsuccessful computation of an answer set; an entire tableau represents a traversal of the search space. This approach provided a uniform proof-theoretic framework for analyzing and comparing different operations, strategies, or even algorithms of ASP solvers. Among others, we related the approaches of existing ASP solvers to appropriate tableau calculi, in the sense that computations of solvers are described by tableaux in corresponding calculi.

In this work, our primary goal is to generalize this approach towards a flexible and modular framework that is easily amenable to new language constructs. We begin with characterizing inferences in a basic core fragment in which rule heads and bodies consist of atomic literals. We then gradually extend this setting by focusing on *aggregates*. An aggregate is understood as an operation on a collection of entities. To obtain a basic understanding of how to describe inferences on aggregates, we view conjunctions in rule bodies as simple Boolean aggregates. We then extend the framework to cardinality constraints, as used in *smodels* [6], and to disjunctions in rule heads.

After establishing the formal background, we introduce in Section 3 our generic tableau framework. In Section 4, 5, and 6, we gradually extend this framework with

* Affiliated with the School of Computing Science at Simon Fraser University, Burnaby, Canada, and IIIS at Griffith University, Brisbane, Australia.

V. Dahl and I. Niemelä (Eds.): ICLP 2007, LNCS 4670, pp. 119–133, 2007.

conjunctions, cardinality constraints, and disjunctions, respectively. Section 7 is dedicated to the proof complexity associated with these constructs. Finally, we discuss our approach in Section 8.

2 Background

We are interested in an extensible proof-theoretic framework dealing with logic programs incorporating composite language constructs, like aggregates. To this end, we need a semantic account of answer sets being, on the one hand, flexible and general enough to accommodate a variety of composite language constructs, and, on the other hand, conservative enough to correspond to standard answer set semantics on well-established language fragments. Among several proposals [6,7,8,9,10], we have chosen Paolo Ferraris' approach [10] that deals with propositional theories and gives meaning to complex constructs by mapping them into propositional formulas.

We start by recalling Ferraris' answer set semantics [10]. A propositional theory is a finite set of propositional formulas, constructed of atoms from an alphabet $\mathcal{P}$ and the connectives $\bot$, $\wedge$, $\vee$, and $\rightarrow$. Any other connective is considered as an abbreviation, in particular, $\neg\phi$ stands for $(\phi \rightarrow \bot)$. An interpretation X, represented by the set of atoms true in X, is a model of a propositional theory Φ if $X \models \phi$ for all $\phi \in \Phi$. The *reduct*, Φ^X, of Φ wrt X is a propositional theory, (recursively) defined as follows:

$$\Phi^X = \left\{\phi^X \mid \phi \in \Phi\right\}$$

$$\phi^X = \begin{cases} \bot & \text{if } X \not\models \phi \\ \phi & \text{if } \phi \in X \\ \phi_1^X \circ \phi_2^X & \text{if } X \models \phi \text{ and } \phi = \phi_1 \circ \phi_2 \text{ for } \circ \in \{\wedge, \vee, \rightarrow\} \end{cases}$$

Intuitively, all (maximal) subformulas of Φ that are false in X are replaced by $\bot$ in Φ^X, all other subformulas of Φ stay unchanged. It is clear that any model of Φ^X is also a model of Φ, since Φ^X contains $\bot$ if $X \not\models \phi$ for some $\phi \in \Phi$. Also note that all occurrences of negation, that is, subformulas of the form $(\phi \rightarrow \bot)$, are replaced by constants in Φ^X, since either ϕ or $(\phi \rightarrow \bot)$ is false in X. An interpretation X is an *answer set* of a propositional theory Φ if X is a minimal model of Φ^X.

We next consider logic programs. Given an alphabet $\mathcal{P}$, a *logic program* is a finite set of *rules* of the form $\alpha \leftarrow \beta$ where α and β are *literals*, understood here as *expressions* over $\mathcal{P}$ possibly preceded by the negation as failure operator not. In the following sections, we gradually refine heads α and bodies β for obtaining particular classes of logic programs. The semantics of logic programs is given by the answer sets of corresponding propositional theories, obtained via particular translations to be provided in the following sections. However, the proof-theoretic characterizations below apply directly to logic programs, without translating them into propositional theories.

We describe calculi for the construction of answer sets from logic programs. Such constructions are associated with an (initial) assignment and a binary tree called a tableau [5]. An *assignment* A is a partial mapping from the expressions in a program into $\{\boldsymbol{T}, \boldsymbol{F}\}$, indicating whether a member of the *domain* of A, denoted by $dom(A)$, is true or false, respectively. The domain of A, $dom(A)$, varies throughout this paper

depending on the considered language fragment. We define $A^{\boldsymbol{T}} = \{v \in dom(A) \mid A(v) = \boldsymbol{T}\}$ and $A^{\boldsymbol{F}} = \{v \in dom(A) \mid A(v) = \boldsymbol{F}\}$. We also denote A by a set of signed expressions: $\{\boldsymbol{T}v \mid v \in A^{\boldsymbol{T}}\} \cup \{\boldsymbol{F}v \mid v \in A^{\boldsymbol{F}}\}$. Furthering this notation, we call an assignment that leaves all expressions undefined *empty* and denote it by $\emptyset$.

The nodes of tableaux are (mainly) *signed expressions*, that is, expressions preceded by either $\boldsymbol{T}$ or $\boldsymbol{F}$, indicating an assumed truth value. A *tableau* for a logic program Π and an initial assignment A is a binary tree such that the root node of the tree consists of the rules in Π and all members of A. The other nodes in the tree are *entries* of the form $\boldsymbol{T}v$ or $\boldsymbol{F}v$, where $v \in dom(A)$, generated by extending a tableau using tableau rules (given in the following sections) in the standard way [5]: Given a tableau rule and a branch in the tableau containing the prerequisites of the rule, the tableau can be extended by adding new entries to the end of the branch as specified by the rule. If the rule is the cut rule (cf. *(g)* in Figure 1), then entries $\boldsymbol{T}v$ and $\boldsymbol{F}v$ are added as the left and the right children to the end of the branch. For the other rules, the consequents are added to the end of the branch. For convenience, the representation of tableau rules makes use of two conjugation functions, $\boldsymbol{t}$ and $\boldsymbol{f}$. For a literal l, define:

$$\boldsymbol{t}l = \begin{cases} \boldsymbol{T}l \text{ if } l \in dom(A) \\ \boldsymbol{F}v \text{ if } l = not\ v \text{ for } v \in dom(A) \end{cases} \qquad \boldsymbol{f}l = \begin{cases} \boldsymbol{F}l \text{ if } l \in dom(A) \\ \boldsymbol{T}v \text{ if } l = not\ v \text{ for } v \in dom(A) \end{cases}$$

Some rule applications are subject to provisos. For instance, $(v \in \Gamma)$ guides the application of the cut rule by restricting cut objects to members of Γ (cf. Figure 1).

A *tableau calculus* $\mathcal{T}$ is a set of tableau rules. An entry $\boldsymbol{T}v$ (or $\boldsymbol{F}v$) can be deduced in a branch if $\boldsymbol{T}v$ (or $\boldsymbol{F}v$) can be generated from nodes in the branch by applying rules of $\mathcal{T}$. Note that any *branch* corresponds to a pair (Π, A) consisting of a program Π and an assignment A; we draw on this relationship for identifying branches in the sequel. A branch in a tableau is *contradictory* if it contains both entries $\boldsymbol{T}v$ and $\boldsymbol{F}v$ for some $v \in dom(A)$. A branch is *complete* if it is contradictory or if it is closed under all rules in $\mathcal{T}$, except for the cut rule, and either $\boldsymbol{T}v \in A$ or $\boldsymbol{F}v \in A$ for each $v \in dom(A)$. A tableau is complete if all its branches are complete. A complete tableau for a program Π and the empty assignment $\emptyset$ such that all branches are contradictory is a *refutation* for Π; provided that $\mathcal{T}$ is a complete calculus, it means that Π has no answer set.

3 Generic Tableau Rules

We begin with a simple class of *unary programs* where rules $\alpha \leftarrow \beta$ are restricted to *atomic literals*, that is, α and β are either equal to p or $not\ p$ for some atom $p \in \mathcal{P}$. The semantics of a unary program Π is given by the answer sets of a propositional theory, $\tau[\Pi]$, (recursively) defined as follows:

$$\tau[\Pi] = \{\tau[\beta] \rightarrow \tau[\alpha] \mid (\alpha \leftarrow \beta) \in \Pi\} \tag{1}$$

$$\tau[\pi] = \begin{cases} \neg\tau[v] & \text{if } \pi = not\ v \\ \pi & \text{if } \pi \in \mathcal{P} \end{cases} \tag{2}$$

For illustration, consider $\Pi_1 = \{a \leftarrow not\ b; not\ a \leftarrow c; b \leftarrow c; c \leftarrow b\}$, whose corresponding propositional theory is $\tau[\Pi_1] = \{\neg b \rightarrow a; c \rightarrow \neg a; c \rightarrow b; b \rightarrow c\}$. The

sets $X_1 = \{a\}$ and $X_2 = \{b, c\}$ are models of $\tau[\Pi_1]$, their reducts are $(\tau[\Pi_1])^{X_1} = \{\neg\bot \rightarrow a; \bot \rightarrow \bot\}$ and $(\tau[\Pi_1])^{X_2} = \{\bot \rightarrow \bot; c \rightarrow \neg\bot; c \rightarrow b; b \rightarrow c\}$. Clearly, X_1 is the unique minimal model of $(\tau[\Pi_1])^{X_1}$, thus, X_1 is an answer set of Π_1. The unique minimal model of $(\tau[\Pi_1])^{X_2}$ is $\emptyset$, that is, X_2 is not an answer set of Π_1.

Unlike the semantics, our tableau framework directly deals with logic programs. The global design, however, follows the two semantic requirements for answer sets: modelhood wrt a program and (non-circular) support wrt the reduct. To begin with, we define for a program Π, two sets $S, S' \subseteq atom(\Pi)$, viz. the set of atoms occurring in Π, and an assignment A:

$$sup_A(\Pi, S, S') = \{(\alpha \leftarrow \beta) \in \Pi \mid \boldsymbol{f}\beta \notin A,\ \overleftarrow{sup}_A(\alpha, S),\ \overrightarrow{sup}_A(\beta, S')\} \tag{3}$$

The purpose of $sup_A(\Pi, S, S')$ is to determine all rules in Π that can, wrt A, provide a support for the atoms in S external wrt S'. Of particular interest are the cases where $sup_A(\Pi, S, S')$ is empty or a singleton $\{\alpha \leftarrow \beta\}$. In the first case, the atoms in S cannot be supported and are prone to be false, while the second case tells us that $\alpha \leftarrow \beta$ is the unique support for S external wrt S' so that β must be true to (non-circularly) support S.

Looking at the definition of $sup_A(\Pi, S, S')$ in (3), we note that a rule $\alpha \leftarrow \beta$ such that $\boldsymbol{f}\beta \in A$ cannot provide any support wrt A. Otherwise, we check via $\overleftarrow{sup}_A(\alpha, S)$ that α can support S and via $\overrightarrow{sup}_A(\beta, S')$ that β does not (positively) rely on S'. For the simple case of unary programs, these concepts are defined as follows:

$$\overleftarrow{sup}_A(p, S) \quad \text{if } p \in S \tag{4}$$
$$\overrightarrow{sup}_A(p, S') \quad \text{if } p \in (\mathcal{P} \setminus S') \tag{5}$$
$$\overrightarrow{sup}_A(not\ v, S') \quad \text{for every expression } v \tag{6}$$

The universal validity of (6) is because only *positive* dependencies are taken into account. Also note that a rule $\alpha \leftarrow \beta$ such that $\alpha = not\ v$ cannot support any set S of atoms. (We further illustrate the above concepts after Theorem 1.)

The tableau rules constituting our basic calculus are given in Figure 1. Rules $I{\uparrow}$ and $I{\downarrow}$ provide basic *rule-based* inferences such as modus ponens and modus tollens. Tableau rules $N{\uparrow}$ and $N{\downarrow}$ amount to *negation* and *support* for atoms stemming from Clark's completion [11]. Note that the derivability of an atom p and thus the applicability of tableau rules $N{\uparrow}$ and $N{\downarrow}$, respectively, is determined by $sup_A(\Pi, \{p\}, \emptyset)$. In the general case, rule $N{\downarrow}$ makes use of two further constructs, $min_A(\alpha, S)$ and $max_A(\beta, S')$, that are used to determine entries that must necessarily be added to A in order to support some atom in S via rule $\alpha \leftarrow \beta$ without positively relying on S'. However, these concepts play no role in the setting of unary programs:

$$min_A(p, S) = \emptyset \quad \text{for } p \in \mathcal{P} \tag{7}$$
$$max_A(p, S') = \emptyset \quad \text{for } p \in \mathcal{P} \tag{8}$$
$$max_A(not\ v, S') = \emptyset \quad \text{for every expression } v \tag{9}$$

Tableau rules $U{\uparrow}$ and $U{\downarrow}$ take care of *unfounded sets* [12], either by identifying atoms that cannot be non-circularly supported ($U{\uparrow}$) or by preventing true atoms to become

$$\frac{\alpha \leftarrow \beta \quad \boldsymbol{t}\beta}{\boldsymbol{t}\alpha} \qquad \frac{\alpha \leftarrow \beta \quad \boldsymbol{f}\alpha}{\boldsymbol{f}\beta}$$

(a) Implication ($I\uparrow$) *(b) Contraposition ($I\downarrow$)*

$$\frac{\Pi, A}{\boldsymbol{F}p}(p \in atom(\Pi), sup_A(\Pi, \{p\}, \emptyset) = \emptyset)$$

(c) Negation ($N\uparrow$)

$$\frac{\Pi, A}{\boldsymbol{t}\beta, min_A(\alpha, \{p\}), max_A(\beta, \emptyset)}(p \in (A^{\boldsymbol{T}} \cap atom(\Pi)), sup_A(\Pi, \{p\}, \emptyset) = \{\alpha \leftarrow \beta\})$$

(d) Support ($N\downarrow$)

$$\frac{\Pi, A}{\boldsymbol{F}p}(S \subseteq atom(\Pi), p \in S, sup_A(\Pi, S, S) = \emptyset)$$

(e) Unfounded Set ($U\uparrow$)

$$\frac{\Pi, A}{\boldsymbol{t}\beta, min_A(\alpha, S), max_A(\beta, S)}(S \subseteq atom(\Pi), (A^{\boldsymbol{T}} \cap S) \neq \emptyset, sup_A(\Pi, S, S) = \{\alpha \leftarrow \beta\})$$

(f) Well-founded Set ($U\downarrow$)

$$\frac{}{\boldsymbol{T}v \mid \boldsymbol{F}v}(v \in \Gamma)$$

(g) Cut ($C[\Gamma]$)

Fig. 1. Generic tableau rules for rules *(a),(b)*; atoms *(c),(d)*; sets of atoms *(e),(f)*; and cutting *(g)*

unfounded ($U\downarrow$). The applicability of $U\uparrow$ and $U\downarrow$ is determined by $sup_A(\Pi, S, S)$ for a set S of atoms; hence, these rules subsume $N\uparrow$ and $N\downarrow$ relying on $sup_A(\Pi, \{p\}, \emptyset)$. We nonetheless include $N\uparrow$ and $N\downarrow$ since their applicability is easy to determine, and so they have counterparts in virtually all ASP solvers. Finally, the cut rule $C[\Gamma]$ in *(g)* constitutes the only rule introducing multiple branches. It allows for case analysis on the expressions in Γ, if the deterministic tableau rules do not yield a complete branch.

Note that the definition of $sup_A(\Pi, S, S')$ in (3) is common to both support-driven (viz. $N\uparrow$ and $N\downarrow$) and unfoundedness-driven (viz. $U\uparrow$ and $U\downarrow$) inferences. As we demonstrate in the following sections, an additional language construct is then incorporated into the basic tableau calculus by supplying tableau rules for handling its truth value and by extending the definition of $\overleftarrow{sup}_A(\alpha, S)$ and $\overrightarrow{sup}_A(\beta, S')$ (along with $min_A(\alpha, S)$ and $max_A(\beta, S')$) to impose an appropriate notion of support.

For a unary program Π, we fix the domain of assignments A as well as the cut objects used by $C[\Gamma]$ to $dom(A) = \Gamma = atom(\Pi)$. This allows us to characterize the answer sets of unary programs by tableaux.

Theorem 1. *Let Π be a unary program and $\emptyset$ the empty assignment.*
Then, the following hold for the tableau calculus consisting of tableau rules (a-g):

1. *Program Π has no answer set iff every complete tableau for Π and $\emptyset$ is a refutation.*
2. *If Π has an answer set X, then every complete tableau for Π and $\emptyset$ has a unique non-contradictory branch (Π, A) such that $X = A^{\boldsymbol{T}} \cap atom(\Pi)$.*

3. *If a tableau for Π and $\emptyset$ has a non-contradictory complete branch (Π, A), then $A^{\boldsymbol{T}} \cap atom(\Pi)$ is an answer set of Π.*

For illustration, consider the program $\Pi_2 = \{a \leftarrow not\ b; b \leftarrow not\ a; c \leftarrow not\ a\}$. Cutting on c results in branches with $\boldsymbol{T}c$ and $\boldsymbol{F}c$, respectively. The first one can be extended by $\boldsymbol{t}not\ a = \boldsymbol{F}a$ via $N{\downarrow}$. Indeed, $sup_{\{\boldsymbol{T}c\}}(\Pi_2, \{c\}, \emptyset) = \{c \leftarrow not\ a\}$ tells us that $c \leftarrow not\ a$ is the only rule that allows for supporting c, which necessitates a to be false. To be more precise, we have $\boldsymbol{f}not\ a = \boldsymbol{T}a \notin \{\boldsymbol{T}c\}$, and both $\overleftarrow{sup}_{\{\boldsymbol{T}c\}}(c, \{c\})$ and $\overrightarrow{sup}_{\{\boldsymbol{T}c\}}(not\ a, \emptyset)$ are satisfied; the proviso of $N{\downarrow}$ is thus established with respect to rule $c \leftarrow not\ a$, so we deduce $\boldsymbol{t}not\ a = \boldsymbol{F}a$. The two remaining sets, $min_{\{\boldsymbol{T}c\}}(c, \{c\}) = max_{\{\boldsymbol{T}c\}}(not\ a, \emptyset) = \emptyset$, are superfluous in the simple language fragment of unary programs. Having deduced $\boldsymbol{F}a$, we can either apply $I{\uparrow}$ or $I{\downarrow}$ to conclude $\boldsymbol{T}b$ and so to obtain a complete branch. The second branch with $\boldsymbol{F}c$ can be extended by $\boldsymbol{f}not\ a = \boldsymbol{T}a$ via $I{\downarrow}$. From this, we deduce $\boldsymbol{F}b$, either by $N{\uparrow}$ or $N{\downarrow}$, obtaining a second complete branch. The two complete branches tell us that $\{b, c\}$ and $\{a\}$ are the answer sets of Π_2.

Further, consider the program $\Pi_3 = \{a \leftarrow b; b \leftarrow a; b \leftarrow c; c \leftarrow not\ d; d \leftarrow not\ c\}$ along with the following two complete branches:

Π_3	
$\boldsymbol{T}d$	$(C[atom(\Pi)])$
$\boldsymbol{F}c$	$(N{\uparrow}, N{\downarrow}, U{\uparrow}, U{\downarrow})$
$\boldsymbol{F}a$	$(U{\uparrow})$
$\boldsymbol{F}b$	$(I{\downarrow}, N{\uparrow}, U{\uparrow})$

Π_3	
$\boldsymbol{T}a$	$(C[atom(\Pi)])$
$\boldsymbol{T}b$	$(I{\uparrow}, N{\downarrow}, U{\downarrow})$
$\boldsymbol{T}c$	$(U{\downarrow})$
$\boldsymbol{F}d$	$(N{\uparrow}, N{\downarrow}, U{\uparrow}, U{\downarrow})$

We have chosen these branches for illustrating the application of the unfounded set rule ($U{\uparrow}$) and the well-founded set rule ($U{\downarrow}$), respectively. (Along the branches, we indicate all possible inferences leading to the same result.) We first inspect the deduction of $\boldsymbol{F}a$ by $U{\uparrow}$ in the left branch. Taking set $\{a, b\}$ (and its element a) makes us check whether $sup_{\{\boldsymbol{T}d, \boldsymbol{F}c\}}(\Pi_3, \{a, b\}, \{a, b\})$ is empty. To this end, we have to inspect all rules that allow for deriving an atom in $\{a, b\}$ (as stipulated via $\overleftarrow{sup}_{\{\boldsymbol{T}d, \boldsymbol{F}c\}}(\alpha, \{a, b\})$). Given $\boldsymbol{f}c = \boldsymbol{F}c$, we only need to consider $a \leftarrow b$ and $b \leftarrow a$. Neither rule satisfies $\overrightarrow{sup}_{\{\boldsymbol{T}d, \boldsymbol{F}c\}}(\beta, \{a, b\})$, which leaves us with $sup_{\{\boldsymbol{T}d, \boldsymbol{F}c\}}(\Pi_3, \{a, b\}, \{a, b\}) = \emptyset$. The well-founded set inference of $\boldsymbol{T}c$ in the right branch requires a set of atoms, some of whose elements is true, such that only one rule can non-circularly support the set. Taking $\{a, b\}$, we can verify that $sup_{\{\boldsymbol{T}a, \boldsymbol{T}b\}}(\Pi_3, \{a, b\}, \{a, b\}) = \{b \leftarrow c\}$. The membership of $b \leftarrow c$ is justified by the fact that $\boldsymbol{f}c = \boldsymbol{F}c \notin \{\boldsymbol{T}a, \boldsymbol{T}b\}$, and that both $\overleftarrow{sup}_{\{\boldsymbol{T}a, \boldsymbol{T}b\}}(b, \{a, b\})$ and $\overrightarrow{sup}_{\{\boldsymbol{T}a, \boldsymbol{T}b\}}(c, \{a, b\})$ hold. Furthermore, neither $a \leftarrow b$ nor $b \leftarrow a$ is included in $sup_{\{\boldsymbol{T}a, \boldsymbol{T}b\}}(\Pi_3, \{a, b\}, \{a, b\})$ because $\overrightarrow{sup}_{\{\boldsymbol{T}a, \boldsymbol{T}b\}}(b, \{a, b\})$ and $\overrightarrow{sup}_{\{\boldsymbol{T}a, \boldsymbol{T}b\}}(a, \{a, b\})$ do not hold. Hence, only $\boldsymbol{t}c = \boldsymbol{T}c$ can justify $\boldsymbol{T}a$ and $\boldsymbol{T}b$.

4 Conjunctive Bodies

Having settled our basic framework, we now allow rule bodies to contain conjunctions. While rule bodies themselves are often regarded as conjunctions, we here take a slightly different perspective in viewing conjunctions as (simple) Boolean aggregates, which like atoms can be preceded by not. This gives us some first insights into the treatment

$$\frac{\boldsymbol{t}l_1,\ldots,\boldsymbol{t}l_n}{\boldsymbol{T}\{l_1,\ldots,l_n\}}$$

(h) True Conjunction ($TC\uparrow$)

$$\frac{\boldsymbol{F}\{l_1,\ldots,l_{i-1},l_i,l_{i+1},\ldots,l_n\}\quad \boldsymbol{t}l_1,\ldots,\boldsymbol{t}l_{i-1},\boldsymbol{t}l_{i+1},\ldots,\boldsymbol{t}l_n}{\boldsymbol{f}l_i}$$

(i) Falsify Conjunction ($TC\downarrow$)

$$\frac{\boldsymbol{f}l_i}{\boldsymbol{F}\{l_1,\ldots,l_i,\ldots,l_n\}}$$

(j) False Conjunction ($FC\uparrow$)

$$\frac{\boldsymbol{T}\{l_1,\ldots,l_n\}}{\boldsymbol{t}l_1,\ldots,\boldsymbol{t}l_n}$$

(k) Justify Conjunction ($FC\downarrow$)

Fig. 2. Tableau rules for conjunctions

of more sophisticated aggregates like cardinality constraints to be dealt with in the next section.

A *conjunction* over an alphabet $\mathcal{P}$ is an expression of the form $\{l_1,\ldots,l_n\}$ where l_i is an atomic literal for $1 \leq i \leq n$. We denote by $conj(\mathcal{P})$ the set of all conjunctions that can be constructed from atoms in $\mathcal{P}$. A rule $\alpha \leftarrow \beta$ such that α is an atomic literal and β is an atomic literal or a (possibly negated) conjunction of atomic literals is a *conjunctive rule*. A logic program is a *conjunctive program* if every rule in it is conjunctive. For defining the semantics of conjunctive programs, we add the following case to translation $\tau[\pi]$ in (2):

$$\tau[\pi] = \textstyle\bigwedge_{l\in\pi}\tau[l] \qquad \text{if } \pi \in conj(\mathcal{P})$$

For accommodating conjunctions within the generic tableau rules in Figure 1, we extend the previous concepts in (4-9) in a straightforward way:

$$\begin{aligned}&\overrightarrow{sup}_A(\{l_1,\ldots,l_n\},S') \quad \text{if } \overrightarrow{sup}_A(l,S') \text{ for every } l \in \{l_1,\ldots,l_n\}\\ &max_A(\{l_1,\ldots,l_n\},S') = \textstyle\bigcup_{l\in\{l_1,\ldots,l_n\}} max_A(l,S')\end{aligned}$$

Note that $max_A(\{l_1,\ldots,l_n\},S')$ is still empty since $max_A(l,S') = \emptyset$ for every atomic literal $l \in \{l_1,\ldots,l_n\}$. It thus has no effect yet, but this changes in the next section.

For a conjunctive program Π, we fix the domain $dom(A)$ of assignments A and the cut objects Γ of $C[\Gamma]$ to $dom(A) = \Gamma = atom(\Pi) \cup conj(\Pi)$, where $conj(\Pi)$ is the set of conjunctions occurring in Π. The additional tableau rules for handling conjunctions are shown in Figure 2. Their purpose is to ensure that $\boldsymbol{T}\{l_1,\ldots,l_n\} \in A$ iff $(A^{\boldsymbol{T}} \cap \mathcal{P}) \models (\tau[l_1] \wedge \cdots \wedge \tau[l_n])$. By augmenting the basic calculus with the tableau rules in Figure 2, Theorem 1 extends to conjunctive programs.

Theorem 2. *Let Π be a conjunctive program and $\emptyset$ the empty assignment.*

Then, statements 1. to 3. given in Theorem 1 hold for the tableau calculus consisting of tableau rules (a-k).

5 Cardinality Constraints

We define a *cardinality constraint* over an alphabet $\mathcal{P}$ as an expression of the form $j\{l_1,\ldots,l_n\}k$ where l_i is an atomic literal for $1 \leq i \leq n$ and j,k are integers such that

$0 \leq j \leq k \leq n$. We denote by $card(\mathcal{P})$ the set of all cardinality constraints that can be constructed from atoms in $\mathcal{P}$. For $v \in (\mathcal{P} \cup card(\mathcal{P}))$, we say that v and $not\ v$ are *cardinality literals*. A rule $\alpha \leftarrow \beta$ such that α is a cardinality literal and β is a cardinality literal or a (possibly negated) conjunction of cardinality literals is a *cardinality rule*. A logic program is a *cardinality program* if it consists of cardinality rules.

Several syntactic classes of programs with cardinality constraints (and other aggregates) can be found in the literature [6, 7, 8, 9, 10]. Furthermore, the semantics differ on aggregates in the heads of rules; which semantics is the right one depends on the intention and cannot be answered a priori. On the one hand, we here adopt the approach of [6, 7, 8] and interpret cardinality constraints in heads as "choice constructs," that is, derived atoms within a cardinality constraint are not minimized. On the other hand, the syntactic class of programs (or formulas, respectively) considered in [10] is presumably the most general one. It allows for arbitrary aggregates over arbitrary formulas, so that cardinality constraints and rules as defined here form a syntactic fragment. The other approaches [6, 7, 8, 9], however, do not cover our notion of a cardinality rule. As before, we thus embed cardinality programs into the framework of [10] to fix their semantics.

The fact that true atoms are not minimized within cardinality constraints in heads of rules necessitates an extended translation of cardinality programs into propositional theories, since the semantics of the latter per default relies on the minimization of true atoms.[1] We now replace the definition of $\tau[\Pi]$ in (1) with the following one:

$$\begin{aligned}\tau[\Pi] = \{&\tau[\beta] \rightarrow \tau[\alpha] \mid (\alpha \leftarrow \beta) \in \Pi, \alpha \notin card(\mathcal{P})\} \cup \\ \{&\tau[\beta] \rightarrow \big(\tau[\alpha] \wedge \textstyle\bigwedge_{p \in atom(\alpha)}(p \vee \neg p)\big) \mid (\alpha \leftarrow \beta) \in \Pi, \alpha \in card(\mathcal{P})\}\end{aligned} \tag{10}$$

where $atom(j\{l_1, \ldots, l_n\}k) = \{l_1, \ldots, l_n\} \cap \mathcal{P}$ for $(j\{l_1, \ldots, l_n\}k) \in card(\mathcal{P})$. Note that conjuncts $\bigwedge_{p \in atom(\alpha)}(p \vee \neg p)$ are tautological and thus neutral as regards the (classical) models of $\tau[\Pi]$. Given an interpretation X, they rather justify the truth of all $p \in atom(\alpha) \cap X$ in $(\tau[\Pi])^X$ where $\neg p$ is replaced by $\bot$. We further add another case to translation $\tau[\pi]$ in (2), which amounts to the aggregate translation given in [10]:

$$\begin{aligned}\tau[\pi] = \big(\textstyle\bigvee_{C \subseteq \{l_1, \ldots, l_n\}, |C| = j} \tau[C]\big) \wedge \neg\big(\textstyle\bigvee_{C \subseteq \{l_1, \ldots, l_n\}, |C| = k+1} \tau[C]\big) \\ \text{if } (\pi = j\{l_1, \ldots, l_n\}k) \in card(\mathcal{P})\end{aligned}$$

Notably, the upper bound k is translated into a negative subformula, and only the subformula for the lower bound j still appears potentially in the reduct wrt a set of atoms. Also note that $\tau[j\{l_1, \ldots, l_n\}k]$ is of exponential size. Even if auxiliary atoms are used to obtain a polynomial translation, like the one described in [6], compilation approaches incur a significant blow-up in space. Our proof-theoretic characterizations, provided in the following, apply directly to cardinality constraints and thus avoid any such blow-up.

Figure 3 shows the tableau rules for cardinality constraints. For a cardinality program Π, we fix the domain $dom(A)$ of assignments A and the cut objects Γ of $C[\Gamma]$ to

[1] Interpreting aggregates in rule heads as "choice constructs" avoids an increase of complexity by one level in the polynomial hierarchy. If derived atoms were to be minimized, it would be straightforward to embed disjunctive programs (see next section) into cardinality programs.

$$\frac{\boldsymbol{t}l_1, \ldots, \boldsymbol{t}l_j, \boldsymbol{f}l_{k+1}, \ldots, \boldsymbol{f}l_n}{\boldsymbol{T}j\{l_1, \ldots, l_j, \ldots, l_{k+1}, \ldots, l_n\}k}$$

(l) True Bounds ($TLU\uparrow$)

$$\frac{\boldsymbol{F}j\{l_1, \ldots, l_{j-1}, l_j, \ldots, l_k, l_{k+1}, \ldots, l_n\}k \quad \boldsymbol{t}l_1, \ldots, \boldsymbol{t}l_{j-1}, \boldsymbol{f}l_{k+1}, \ldots, \boldsymbol{f}l_n}{\boldsymbol{f}l_j, \ldots, \boldsymbol{f}l_k}$$

(m) Falsify Lower Bound ($TL\downarrow$)

$$\frac{\boldsymbol{F}j\{l_1, \ldots, l_j, l_{j+1}, \ldots, l_{k+1}, l_{k+2}, \ldots, l_n\}k \quad \boldsymbol{t}l_1, \ldots, \boldsymbol{t}l_j, \boldsymbol{f}l_{k+2}, \ldots, \boldsymbol{f}l_n}{\boldsymbol{t}l_{j+1}, \ldots, \boldsymbol{t}l_{k+1}}$$

(n) Falsify Upper Bound ($TU\downarrow$)

$$\frac{\boldsymbol{f}l_j, \ldots, \boldsymbol{f}l_n}{\boldsymbol{F}j\{l_1, \ldots, l_j, \ldots, l_n\}k}$$

(o) False Lower Bound ($FL\uparrow$)

$$\frac{\boldsymbol{t}l_1, \ldots, \boldsymbol{t}l_{k+1}}{\boldsymbol{F}j\{l_1, \ldots, l_{k+1}, \ldots, l_n\}k}$$

(p) False Upper Bound ($FU\uparrow$)

$$\frac{\boldsymbol{T}j\{l_1, \ldots, l_j, l_{j+1} \ldots, l_n\}k \quad \boldsymbol{f}l_{j+1}, \ldots, \boldsymbol{f}l_n}{\boldsymbol{t}l_1, \ldots, \boldsymbol{t}l_j}$$

(q) Justify Lower Bound ($FL\downarrow$)

$$\frac{\boldsymbol{T}j\{l_1, \ldots, l_k, l_{k+1} \ldots, l_n\}k \quad \boldsymbol{t}l_1, \ldots, \boldsymbol{t}l_k}{\boldsymbol{f}l_{k+1}, \ldots, \boldsymbol{f}l_n}$$

(r) Justify Upper Bound ($FU\downarrow$)

Fig. 3. Tableau rules for cardinality constraints

$dom(A) = \Gamma = atom(\Pi) \cup conj(\Pi) \cup card(\Pi)$, where $card(\Pi)$ is the set of cardinality constraints occurring in Π. For a cardinality constraint $j\{l_1, \ldots, l_n\}k$, the tableau rules in Figure 3 ensure that $\boldsymbol{T}(j\{l_1, \ldots, l_n\}k) \in A$ iff $(A^{\boldsymbol{T}} \cap \mathcal{P}) \models \tau[j\{l_1, \ldots, l_n\}k]$.

For illustration, consider the cardinality constraint

$$\gamma = 2\{a, b, c, not\ d, not\ e\}3\ ,$$

having $n = 5$ literals, lower bound $j = 2$, and upper bound $k = 3$. For an assignment A, tableau rule $TLU\uparrow$ allows for deducing $\boldsymbol{T}\gamma$ if at least $j = 2$ literals l of γ are true (i.e., $\boldsymbol{t}l \in A$) and at least $n - k = 2$ literals l of γ are false (i.e., $\boldsymbol{f}l \in A$). This holds, for instance, for assignment $A_1 = \{\boldsymbol{T}a, \boldsymbol{F}b, \boldsymbol{T}d, \boldsymbol{F}e\}$; hence, $\boldsymbol{T}\gamma$ can be deduced via $TLU\uparrow$. Indeed, the lower and upper bound of γ are satisfied in every non-contradictory branch that extends A_1, no matter whether $\boldsymbol{T}c$ or $\boldsymbol{F}c$ is additionally included. Rules $TL\downarrow$ and $TU\downarrow$ are the contrapositives of $TLU\uparrow$, ensuring that either the lower or the upper bound of γ is violated if $\boldsymbol{F}\gamma$ belongs to an assignment. For instance, $\boldsymbol{F}c$ and $\boldsymbol{T}e$ can be deduced via $TL\downarrow$ wrt assignment $A_2 = \{\boldsymbol{F}\gamma, \boldsymbol{T}a, \boldsymbol{F}b, \boldsymbol{T}d\}$. Observe that the upper bound $k = 3$ cannot be violated in non-contradictory extensions of A_2 since $n-k = 2$ literals of γ are already false wrt A_2, as it contains $\boldsymbol{F}b$ and $\boldsymbol{T}d$. Conversely, $TU\downarrow$ allows for deducing $\boldsymbol{T}c$ and $\boldsymbol{F}e$ wrt $\{\boldsymbol{F}\gamma, \boldsymbol{T}a, \boldsymbol{F}b, \boldsymbol{F}d\}$ in order to violate the upper bound of γ; the lower bound of γ cannot be violated because of $\boldsymbol{T}a$ and $\boldsymbol{F}d$. The remaining four tableau rules in Figure 3 either allow for deducing $\boldsymbol{F}\gamma$, if its lower bound ($FL\uparrow$) or its upper bound ($FU\uparrow$) is violated, or make sure that the lower bound ($FL\downarrow$) and the upper bound ($FU\downarrow$) are satisfied, if $\boldsymbol{T}\gamma$ belongs to an assignment. For γ as above, $\boldsymbol{F}\gamma$ can be deduced via $FL\uparrow$ wrt assignment $\{\boldsymbol{F}b, \boldsymbol{F}c, \boldsymbol{T}d, \boldsymbol{T}e\}$ or via $FU\uparrow$ wrt assignment $\{\boldsymbol{T}b, \boldsymbol{T}c, \boldsymbol{F}d, \boldsymbol{F}e\}$. Conversely, $FL\downarrow$ allows for deducing $\boldsymbol{T}a$ and $\boldsymbol{F}e$ wrt $\{\boldsymbol{T}\gamma, \boldsymbol{F}b, \boldsymbol{F}c, \boldsymbol{T}d\}$, and $FU\downarrow$ allows for deducing $\boldsymbol{F}a$ and $\boldsymbol{T}e$ wrt $\{\boldsymbol{T}\gamma, \boldsymbol{T}b, \boldsymbol{T}c, \boldsymbol{F}d\}$.

To integrate cardinality constraints into the generic setting of the tableau rules in Figure 1, we also have to extend the concepts in (4-9):

$$\overleftarrow{sup}_A(j\{l_1,\ldots,l_n\}k,S) \text{ if } \{l_1,\ldots,l_n\}\cap S\neq\emptyset \text{ and } |\{l\in(\{l_1,\ldots,l_n\}\setminus S)\mid \boldsymbol{t}l\in A\}|<k$$

$$\overrightarrow{sup}_A(j\{l_1,\ldots,l_n\}k,S') \text{ if } |\{l\in(\{l_1,\ldots,l_n\}\setminus S')\mid \boldsymbol{f}l\notin A\}|\geq j$$

$$min_A(j\{l_1,\ldots,l_n\}k,S) = \begin{cases}\{\boldsymbol{f}l\mid l\in(\{l_1,\ldots,l_n\}\setminus S),\ \boldsymbol{t}l\notin A\} \\ \quad\text{if } |\{l\in(\{l_1,\ldots,l_n\}\setminus S)\mid \boldsymbol{t}l\in A\}|=k-1\\ \emptyset \text{ if } |\{l\in(\{l_1,\ldots,l_n\}\setminus S)\mid \boldsymbol{t}l\in A\}|\neq k-1\end{cases}$$

$$max_A(j\{l_1,\ldots,l_n\}k,S') = \begin{cases}\{\boldsymbol{t}l\mid l\in(\{l_1,\ldots,l_n\}\setminus S'),\ \boldsymbol{f}l\notin A\} \\ \quad\text{if } |\{l\in(\{l_1,\ldots,l_n\}\setminus S')\mid \boldsymbol{f}l\notin A\}|=j\\ \emptyset \text{ if } |\{l\in(\{l_1,\ldots,l_n\}\setminus S')\mid \boldsymbol{f}l\notin A\}|\neq j\end{cases}$$

Recall that $\overleftarrow{sup}_A(\alpha,S)$ is used to determine whether a rule with head α can provide support for the atoms in S. If $\alpha=j\{l_1,\ldots,l_n\}k$, then some atom of S must be contained in $\{l_1,\ldots,l_n\}$. Furthermore, if $(\{l_1,\ldots,l_n\}\setminus S)$ already contains k (or more) literals that are true wrt A, then the addition of $\boldsymbol{T}p$ to A for $p\in(\{l_1,\ldots,l_n\}\cap S)$ would violate the upper bound k, so that the corresponding rule $\alpha\leftarrow\beta$ cannot support S. This also explains the false literals in $min_A(j\{l_1,\ldots,l_n\}k,S)$ that can be deduced if $k-1$ literals of $(\{l_1,\ldots,l_n\}\setminus S)$ are already true wrt A. If one more literal in $(\{l_1,\ldots,l_n\}\setminus S)$ were made true, S would lose its last (external) support $\alpha\leftarrow\beta$, though containing some atom that is true wrt A (cf. $N\downarrow$ and $U\downarrow$ in Figure 1). In addition, $\overrightarrow{sup}_A(\beta,S')$ is used to verify whether a support via β is external wrt S'. If, for a rule $\alpha\leftarrow\beta$, either $\beta=j\{l_1,\ldots,l_n\}k$ or β is a conjunction such that $j\{l_1,\ldots,l_n\}k\in\beta$, then there must be enough non-false literals in $(\{l_1,\ldots,l_n\}\setminus S')$ to achieve the lower bound j. Furthermore, if the number of such literals is exactly j, then all of them must be true for providing a support that is external wrt S'. This is expressed by $max_A(j\{l_1,\ldots,l_n\}k,S')$.

For illustration, consider the following cardinality program Π_4:[2]

$$\Pi_4=\left\{\begin{array}{lll} r_1: & 0\{c,d,e\}3\leftarrow & \\ r_2: & 1\{a,b\}2\leftarrow c,d & r_4:\ 1\{b,d\}2\leftarrow 1\{a,c\}2\\ r_3: & 0\{a,d\}1\leftarrow 1\{b,not\ e\}2 & r_5:\ 1\{a,d\}2\leftarrow b\end{array}\right\}$$

Let assignment A contain $\boldsymbol{T}a$, $\boldsymbol{F}c$, and $\boldsymbol{F}\{c,d\}$, and note that tableau rule $U\downarrow$ (or $N\downarrow$) does not apply to set $\{a\}$ since $sup_A(\Pi_4,\{a\},\{a\})=\{r_3,r_5\}$. Let us however consider set $\{a,b\}$. Given that $\{c,d,e\}\cap\{a,b\}=\emptyset$ and $\boldsymbol{F}\{c,d\}\in A$, we have $r_1\notin sup_A(\Pi_4,\{a,b\},\{a,b\})$ and $r_2\notin sup_A(\Pi_4,\{a,b\},\{a,b\})$. Considering r_5, we have $b\in\{a,b\}$ and thus $\overrightarrow{sup}_A(b,\{a,b\})$ does not hold. For r_4, we have $\{a,c\}\setminus\{a,b\}=\{c\}$ and $\boldsymbol{f}c=\boldsymbol{F}c\in A$, so that there are no non-false literals in $\{a,c\}\setminus\{a,b\}$. That is, the lower bound 1 of $1\{a,c\}2$ cannot be achieved independently from $\{a,b\}$, and thus $\overrightarrow{sup}_A(1\{a,c\}2,\{a,b\})$ does not hold. We have now established that only r_3 is potentially contained in $sup_A(\Pi_4,\{a,b\},\{a,b\})$. Since $(not\ e)\in\{b,not\ e\}$ is a non-false literal not belonging to $\{a,b\}$, $\overrightarrow{sup}_A(1\{b,not\ e\}2,\{a,b\})$ holds. Furthermore, $\overleftarrow{sup}_A(0\{a,d\}1,\{a,b\})$ holds because $\boldsymbol{t}d=\boldsymbol{T}d$ does not belong to A. We thus have

[2] In the sequel, we skip set notation for conjunctive bodies within rules, like $\{c,d\}$ in rule r_2.

$$\frac{\boldsymbol{t}l_i}{\boldsymbol{T}\{l_1;\ldots;l_i;\ldots;l_n\}}$$
(s) True Disjunction ($TD\uparrow$)

$$\frac{\boldsymbol{F}\{l_1;\ldots;l_n\}}{\boldsymbol{f}l_1,\ldots,\boldsymbol{f}l_n}$$
(t) Falsify Disjunction ($TD\downarrow$)

$$\frac{\boldsymbol{f}l_1,\ldots,\boldsymbol{f}l_n}{\boldsymbol{F}\{l_1;\ldots;l_n\}}$$
(u) False Disjunction ($FD\uparrow$)

$$\frac{\boldsymbol{T}\{l_1;\ldots;l_{i-1};l_i;l_{i+1};\ldots;l_n\}\quad \boldsymbol{f}l_1,\ldots,\boldsymbol{f}l_{i-1},\boldsymbol{f}l_{i+1},\ldots,\boldsymbol{f}l_n}{\boldsymbol{t}l_i}$$
(v) Justify Disjunction ($FD\downarrow$)

Fig. 4. Tableau rules for disjunctions

$sup_A(\Pi_4,\{a,b\},\{a,b\})=\{r_3\}$. As literals deducible via $U\downarrow$, we obtain $\boldsymbol{T}(1\{b, not\ e\}2)$, $max_A(1\{b, not\ e\}2,\{a,b\}) = \{\boldsymbol{t}not\ e = \boldsymbol{F}e\}$, and $min_A(0\{a,d\}1,\{a,b\}) = \{\boldsymbol{f}d = \boldsymbol{F}d\}$. Finally, note that, if $\boldsymbol{T}d$ had been contained in A, we would have obtained $sup_A(\Pi_4,\{a,b\},\{a,b\}) = \emptyset$, so that deducing $\boldsymbol{F}a$ via $U\uparrow$ would have led to a contradiction. Indeed, $\{a,b\}$ is the only answer set of Π_4 compatible with $\boldsymbol{T}a$ and $\boldsymbol{F}c$.

The tableau calculus for cardinality programs is obtained by adding the rules in Figure 3 to those in Figure 1 and 2. Then, Theorem 1 extends to cardinality programs.

Theorem 3. *Let Π be a cardinality program and $\emptyset$ the empty assignment.*

Then, statements 1. to 3. given in Theorem 1 hold for the tableau calculus consisting of tableau rules (a-r).

6 Disjunctive Heads

A *disjunction* over an alphabet $\mathcal{P}$ is an expression of the form $\{l_1;\ldots;l_n\}$ where l_i is an atomic literal for $1 \leq i \leq n$. We denote by $disj(\mathcal{P})$ the set of all disjunctions that can be constructed from atoms in $\mathcal{P}$. For $v \in (\mathcal{P} \cup card(\mathcal{P}) \cup disj(\mathcal{P}))$, v and $not\ v$ are *disjunctive literals*. A rule $\alpha \leftarrow \beta$ such that α is a disjunctive literal and β is a cardinality literal or a (possibly negated) conjunction of cardinality literals is a *disjunctive rule*. A logic program is a *disjunctive program* if it consists of disjunctive rules.

In contrast to cardinality constraints serving as "choice constructs," the common semantics for disjunctions relies on the minimization of derived atoms. Hence, we adhere to the definition of $\tau[\Pi]$ in (10) and just add another case to $\tau[\pi]$ in (2):

$$\tau[\pi] = \bigvee_{l\in\pi}\tau[l] \qquad \text{if } \pi \in disj(\mathcal{P})$$

We further extend the concepts in (4-9) to disjunctive heads:

$$\begin{aligned}\overleftarrow{sup}_A(\{l_1;\ldots;l_n\},S)\ \text{ if }\ &\{l_1,\ldots,l_n\}\cap S\neq\emptyset\ \text{ and}\\ &\{l\in(\{l_1,\ldots,l_n\}\setminus S)\mid \boldsymbol{t}l\in A\}=\emptyset\\ min_A(\{l_1;\ldots;l_n\},S)\ =\ &\{\boldsymbol{f}l\mid l\in(\{l_1,\ldots,l_n\}\setminus S)\}\end{aligned}$$

For a disjunctive program Π, we fix the domain $dom(A)$ of assignments A and the cut objects Γ of $C[\Gamma]$ to $dom(A) = \Gamma = atom(\Pi)\cup conj(\Pi)\cup card(\Pi)\cup disj(\Pi)$, where

$disj(\Pi)$ is the set of disjunctions occurring in Π. The additional tableau rules for handling disjunctions are shown in Figure 4. Their purpose is to ensure that $\boldsymbol{T}\{l_1;\ldots;l_n\}\in A$ iff $(A^{\boldsymbol{T}}\cap\mathcal{P})\models(\tau[l_1]\vee\cdots\vee\tau[l_n])$. The tableau calculus for disjunctive programs is obtained by adding the rules in Figure 4 to those in Figure 1, 2, and 3. In this way, Theorem 1 extends to disjunctive programs.

Theorem 4. *Let Π be a disjunctive program and $\emptyset$ the empty assignment.*

Then, statements 1. to 3. given in Theorem 1 hold for the tableau calculus consisting of tableau rules (a-v).

7 Proof Complexity

For comparing different tableau calculi, we use the concept of *proof complexity* [3, 13, 14]. Intuitively, proof complexity is concerned with lower bounds on the run-times of proof-finding algorithms independent from heuristic influences. We thus compare the sizes of *minimal* refutations for unsatisfiable logic programs, that is, programs without answer sets. The size of a tableau is determined in the standard way as the number of nodes it contains. A tableau calculus $\mathcal{T}$ is *not polynomially simulated* by another tableau calculus $\mathcal{T}'$ if there is an infinite (witnessing) family $\{\Pi^n\}$ of unsatisfiable programs such that minimal refutations of $\mathcal{T}'$ for Π are asymptotically exponential in the size of minimal refutations of $\mathcal{T}$ for Π. A tableau calculus $\mathcal{T}$ is *exponentially stronger* than a tableau calculus $\mathcal{T}'$ if $\mathcal{T}$ polynomially simulates $\mathcal{T}'$, but not vice versa.

We have shown in [4] that the cut rule has a major influence on proof complexity. For normal logic programs Π, both $C[atom(\Pi)]$ and $C[conj(\Pi)]$ yield sound and complete tableau calculi. However, the calculus obtained with $C[atom(\Pi)\cup conj(\Pi)]$ is exponentially stronger than both of them [4]. It is thus an interesting question whether cutting on other composite constructs, namely, cardinality constraints and disjunctions, leads to stronger calculi. In this context, let us stress that cutting on atoms is sufficient for obtaining complete calculi even in the presence of language extensions, provided that the truth values of all composite constructs can be deduced from atomic literals via (deterministic) tableau rules. For cardinality constraints and disjunctions, this is possible using the tableau rules in Figure 3 and 4. In general, the assumption that knowing the atoms' truth values is enough to evaluate all constructs in a program seems reasonable.

We first consider cardinality programs Π. Let $\mathcal{T}_c = \{$*(a-f), (h-r),* $C[atom(\Pi)\cup conj(\Pi)\cup card(\Pi)]\}$ and $\mathcal{T}_{\overline{c}} = \{$*(a-f), (h-r),* $C[atom(\Pi)\cup conj(\Pi)]\}$. Both calculi contain all deterministic tableau rules dealing with cardinality programs; the difference is that cutting on cardinality constraints is allowed with $\mathcal{T}_c$, but not with $\mathcal{T}_{\overline{c}}$. Since every refutation of $\mathcal{T}_{\overline{c}}$ is as well a refutation of $\mathcal{T}_c$, it is clear that $\mathcal{T}_c$ polynomially simulates $\mathcal{T}_{\overline{c}}$.

As the following result shows, the converse does not hold.

Theorem 5. *Tableau calculus $\mathcal{T}_c$ is exponentially stronger than $\mathcal{T}_{\overline{c}}$.*

This result is witnessed by the following unsatisfiable cardinality programs:

$$\Pi_c^n=\left\{\begin{array}{ll} x\leftarrow 1\{a_1,b_1\}2,\ldots,1\{a_n,b_n\}2,\mathit{not}\ x & \\ a_i\leftarrow \mathit{not}\ b_i & b_i\leftarrow \mathit{not}\ a_i \end{array}\;\middle|\; i=1..n\right\}$$

Roughly, a branch containing $\boldsymbol{F}(1\{a_i, b_i\}2)$ for $1 \leq i \leq n$ yields an immediate contradiction because $\boldsymbol{F}a_i$ and $\boldsymbol{F}b_i$ can be deduced via $TL{\downarrow}$, violating the last two rules in Π_c^n. The unrestricted cut rule of $\mathcal{T}_c$ permits cutting on $1\{a_i, b_i\}2$ for $1 \leq i \leq n$, and the resulting minimal refutation has linear size in n. In contrast to this, $\mathcal{T}_{\overline{c}}$ must cut on atoms a_i or b_i, spanning a complete binary tree whose size is exponential in n.

The practical consequence of Theorem 5 is that ASP solvers dealing with cardinality constraints can gain significant speed-ups by branching on them. Notably, the compilation of cardinality constraints into so-called "basic constraint rules" [6], as done by *lparse* [15], introduces auxiliary atoms abbreviating cardinality constraints. In this way, *smodels* can (implicitly) branch on cardinality constraints although its choices are restricted to atoms. In contrast to this compilation approach introducing abbreviations, we here however consider cardinality constraints as self-contained structural entities.

Regarding disjunctive programs, we mention that verifying the non-applicability of tableau rules $U{\uparrow}$ and $U{\downarrow}$, dealing with unfounded sets, is coNP-hard [16].[3] Thus, tableaux for disjunctive programs are generally not polynomially verifiable, unless P=NP. Different from cardinality constraints occurring in bodies of rules, the syntax of disjunctive programs usually restricts disjunctions to occur in heads of rules; this is done here as well. If this restriction were dropped, program Π_c^n could be rewritten using disjunctions $\{a_i; b_i\}$ rather than $1\{a_i, b_i\}2$ for $1 \leq i \leq n$. This would yield the same exponential separation between cut rules with and without disjunctions as encountered on cardinality constraints. With disjunctions $\{l_1; \ldots; l_n\}$ restricted to heads of rules, the crux is that the information gained in the branch of $\boldsymbol{T}\{l_1; \ldots; l_n\}$ is rather weak. We thus conjecture that, with disjunctions restricted to heads, the possibility to branch on them does not yield exponential improvements, though it is certainly convenient to apply the cut rule to disjunctions as well. The effect of allowing or disallowing cutting on disjunctions in heads of rules however remains a subject to future investigation.

Finally, when looking at the inferences of existing ASP solvers, an asymmetry can be observed in unfounded set handling [4]. While (non-SAT-based) ASP solvers make inferences that can be described by tableau rule $U{\uparrow}$, no solver implements $U{\downarrow}$. This brings our attention to the question whether omitted inferences deteriorate proof complexity. (Of course, the available inferences must still be strong enough to guarantee soundness.) In what follows, we denote by $R{\uparrow}$ and $R{\downarrow}$ the forward and backward variant, respectively, of any of the (deterministic) tableau rules in Figure 1 to 4. Given a tableau calculus $\mathcal{T}$, we say that $\mathcal{T}' \subseteq \mathcal{T}$ is an *approximation* of $\mathcal{T}$ if $(\mathcal{T} \setminus \mathcal{T}') \subseteq \{R{\downarrow} \mid R{\uparrow} \in \mathcal{T}'\}$. (We assume that $TLU{\uparrow} \in \mathcal{T}'$ if $\{TL{\downarrow}, TU{\downarrow}\} \cap (\mathcal{T} \setminus \mathcal{T}') \neq \emptyset$ given that $TLU{\uparrow}$ has *two* backward counterparts, viz. $TL{\downarrow}$ and $TU{\downarrow}$.) That is, if $\mathcal{T}$ contains both $R{\uparrow}$ and $R{\downarrow}$, then an approximation $\mathcal{T}'$ of $\mathcal{T}$ might drop $R{\downarrow}$. Of course, $R{\downarrow}$ can also be kept, so that $\mathcal{T}$ is an (the greatest) approximation of itself. It is clear that every approximation $\mathcal{T}'$ of $\mathcal{T}$ is polynomially simulated by $\mathcal{T}$.

Assuming an unrestricted cut rule, the next result shows that the converse also holds.

Theorem 6. *Let $\mathcal{T}$ be a tableau calculus and $\mathcal{T}'$ an approximation of $\mathcal{T}$.*

If $C[atom(\Pi) \cup conj(\Pi) \cup card(\Pi) \cup disj(\Pi)] \in \mathcal{T}'$, then $\mathcal{T}$ is polynomially simulated by $\mathcal{T}'$.

[3] For cardinality programs, verifying the absence of unfounded sets is tractable. Suitably adjusted, the operations described in [8] could be used to accomplish this task.

In fact, an inference conducted by $R{\downarrow}$ can alternatively be obtained by cutting on the consequent of $R{\downarrow}$. Then, one of the two branches becomes contradictory by applying $R{\uparrow}$. However, recall that proof complexity assumes an optimal heuristics determining the "right" objects to cut on, which is inaccessible in practice. Hence, it is reasonable to also deduce the consequents of $R{\downarrow}$ within an ASP solver whenever it is feasible.

8 Discussion

In contrast to propositional logic, where the proof-theoretic foundations of SAT solvers are well-understood [2, 3, 13], little work has so far been done on proof theory for ASP and its solving approaches. The imbalance becomes even more apparent in view of the comprehensive semantic characterizations that nowadays exist for ASP. For instance, the approach in [10] specifies answer sets for propositional theories, going beyond the syntax of logic programs; it also provides a general semantics for aggregates. Though the latter are a key issue in ASP, where knowledge representation is a major objective, the support of such composite constructs in ASP solvers is rather ad hoc. This can, for instance, be seen with *lparse* compiling away negative weights within weight constraints, sometimes leading to counterintuitive results [10], and with *dlv* [17] not supporting aggregates in the heads of rules. While we have in [4] restricted our attention to normal logic programs, whose role in ASP is comparable to CNF in SAT, in this work, we have primarily aimed at language extensions and their integration into a common proof-theoretic framework. On the examples of cardinality constraints and disjunctive heads, we have seen that tableaux are well-suited for augmenting the core language and basic inference mechanisms with additional constructs.

Several lessons can be learned from the illustrative integration of cardinality constraints and disjunctive heads. Independently of the construct under consideration, inference rules follow two major objectives: first, making sure that solutions correspond to models of programs, and second, verifying that all true atoms are non-circularly supported. Different notions of support are possible. For instance, atoms derived via composite constructs in heads of rules might be subject to minimization, as with disjunctions, or not, as with cardinality constraints allowing for "choices." Such issues need to be settled in order to specify the required inference patterns. This also concerns computational complexity; for instance, it increases by one level in the polynomial hierarchy with disjunctive heads or negative weights within weight constraints. Furthermore, the proof complexity of an approach critically depends on the cut rule determining the objects available for case analysis. Composite constructs often constitute structures that are valuable in this respect, as it has been shown for conjunctions and cardinality constraints. We conclude that complex language constructs deserve particular attention in the context of ASP solving. For solvers using learning, like *clasp* [18], it is important not only that inferences are performed but also that their reasons are properly identified. The declarative nature of tableau rules provides a basis to describe such reasons. Importantly, the extensibility of the framework also allows for combining the processes of designing novel language constructs and of fixing their proof-theoretic meaning.

Acknowledgments. We are grateful to Robert Mercer, Richard Tichy, and the anonymous referees for many helpful suggestions.

References

1. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press, Cambridge (2003)
2. Mitchell, D.: A SAT solver primer. Bulletin of the European Association for Theoretical Computer Science 85, 112–133 (2005)
3. Beame, P., Kautz, H., Sabharwal, A.: Towards understanding and harnessing the potential of clause learning. Journal of Artificial Intelligence Research 22, 319–351 (2004)
4. Gebser, M., Schaub, T.: Tableau calculi for answer set programming. In: Etalle, S., Truszczyński, M. (eds.) ICLP 2006. LNCS, vol. 4079, pp. 11–25. Springer, Heidelberg (2006)
5. D'Agostino, M., Gabbay, D., Hähnle, R., Posegga, J. (eds.): Handbook of Tableau Methods. Kluwer Academic Publishers, Dordrecht (1999)
6. Simons, P., Niemelä, I., Soininen, T.: Extending and implementing the stable model semantics. Artificial Intelligence 138(1-2), 181–234 (2002)
7. Ferraris, P., Lifschitz, V.: Weight constraints as nested expressions. Theory and Practice of Logic Programming 5(1-2), 45–74 (2005)
8. Liu, L., Truszczyński, M.: Properties of programs with monotone and convex constraints. In: Veloso, M., Kambhampati, S. (eds.) Proceedings of the National Conference on Artificial Intelligence (AAAI'05), pp. 701–706. AAAI/MIT Press (2005)
9. Faber, W.: Unfounded sets for disjunctive logic programs with arbitrary aggregates. [19] 40–52
10. Ferraris, P.: Answer sets for propositional theories. [19] 119–131
11. Clark, K.: Negation as failure. In: Gallaire, H., Minker, J. (eds.) Logic and Data Bases, pp. 293–322. Plenum Press, New York (1978)
12. van Gelder, A., Ross, K., Schlipf, J.: The well-founded semantics for general logic programs. Journal of the ACM 38(3), 620–650 (1991)
13. Beame, P., Pitassi, T.: Propositional proof complexity: Past, present, and future. Bulletin of the European Association for Theoretical Computer Science 65, 66–89 (1998)
14. Järvisalo, M., Junttila, T., Niemelä, I.: Unrestricted vs restricted cut in a tableau method for Boolean circuits. Annals of Mathematics and Artificial Intelligence 44(4), 373–399 (2005)
15. Syrjänen, T.: Lparse 1.0 user's manual, `http://www.tcs.hut.fi/Software/smodels/lparse.ps.gz`
16. Leone, N., Rullo, P., Scarcello, F.: Disjunctive stable models: Unfounded sets, fixpoint semantics, and computation. Information and Computation 135(2), 69–112 (1997)
17. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV system for knowledge representation and reasoning. ACM Transactions on Computational Logic 7(3), 499–562 (2006)
18. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Conflict-driven answer set solving. In: Veloso, M. (ed.) Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI'07), pp. 386–392. AAAI/MIT Press (2007)
19. Baral, C., Greco, G., Leone, N., Terracina, G. (eds.): LPNMR 2005. LNCS (LNAI), vol. 3662. Springer, Heidelberg (2005)

Extended ASP Tableaux and Rule Redundancy in Normal Logic Programs

Matti Järvisalo and Emilia Oikarinen

Laboratory for Theoretical Computer Science
P.O. Box 5400, FI-02015 Helsinki University of Technology (TKK), Finland

Abstract. We introduce an extended tableau calculus for answer set programming (ASP). The proof system is based on the ASP tableaux defined in [Gebser&Schaub, ICLP 2006], with an added extension rule. We investigate the power of Extended ASP Tableaux both theoretically and empirically. We study the relationship of Extended ASP Tableaux with the Extended Resolution proof system defined by Tseitin for clause sets, and separate Extended ASP Tableaux from ASP Tableaux by giving a polynomial length proof of a family of normal logic programs $\{\Pi_n\}$ for which ASP Tableaux has exponential length minimal proofs with respect to n. Additionally, Extended ASP Tableaux imply interesting insight into the effect of program simplification on the length of proofs in ASP. Closely related to Extended ASP Tableaux, we empirically investigate the effect of redundant rules on the efficiency of ASP solving.

1 Introduction

Answer set programming (ASP) is a declarative problem solving paradigm which has proven successful for a variety of knowledge representation and reasoning tasks. The success has been brought forth by efficient solver implementations bringing the theoretical underpinnings into practice. However, there has been an evident lack of theoretical studies into the reasons for the efficiency of current ASP solvers (e.g. [1,2,3,4]). Solver implementations and their inference techniques can be seen as determinisations of the underlying rule-based *proof systems*. Due to this strong interplay between theory and practice, the study of the relative efficiency of these proof systems reveals important new viewpoints and explanations for the successes and failures of particular solver techniques. While such proof complexity [5] studies are frequent in the closely related field of propositional satisfiability (SAT), where typical solvers have been shown to be based on refinements of the well-known Resolution proof system [6], this has not been the case for ASP. Especially, the inference techniques applied in current state-of-the-art ASP solvers have been characterised by a family of tableau-style ASP proof systems for normal logic programs only very recently [7], with some related preliminary proof complexity theoretic investigations [8]. The close relation of ASP and SAT and the respective theoretical underpinning of practical solver techniques has also received little attention up until recently [9,10], although the fields could gain much by further studies on these connections.

This paper continues in part bridging the gap between ASP and SAT. Influenced by Tseitin's *Extended Resolution* proof system [11] for clausal formulas, we introduce

V. Dahl and I. Niemelä (Eds.): ICLP 2007, LNCS 4670, pp. 134–148, 2007.

Extended ASP Tableaux, an extended tableau calculus based on the proof system in [7]. The motivations for Extended ASP Tableaux are many-fold. Theoretically, Extended Resolution has proven to be among the most powerful known proof systems, equivalent to, e.g., extended Frege systems; no exponential lower bounds for the lengths of proofs are known for Extended Resolution. We study the power of Extended ASP Tableaux, showing a tight correspondence with Extended Resolution.

The contributions of this paper are not only of theoretical nature. Extended ASP Tableaux is in fact based on *adding structure* into programs by introducing additional *redundant rules*. On the practical level, structure of problem instances has an important role in both ASP and SAT solving. Typically, it is widely believed that redundancy can and should be removed for practical efficiency. However, the power of Extended ASP Tableaux reveals that this is not generally the case, and such redundancy removing *simplification* mechanism can drastically hinder efficiency. In addition, we contribute by studying the effect of redundancy on the efficiency of a variety of ASP solvers. The results show that the role of redundancy in programs is not as simple as typically believed, and controlled addition of redundancy may in fact prove to be relevant in further strengthening the robustness of current solver techniques.

The paper is organised as follows. After preliminaries on ASP and SAT (Sect. 2), the relationship of Resolution and ASP Tableaux proof systems and concepts related to the complexity of proofs are discussed (Sect. 3). By introducing the Extended ASP Tableaux proof system (Sect. 4), proof complexity and simplification are then studied w.r.t. Extended ASP Tableaux (Sect. 5). Experimental results related to Extended ASP Tableaux and redundant rules in normal logic programs are presented in Sect. 6.

2 Preliminaries

As preliminaries we review basic concepts related to answer set programming (ASP) in the context of normal logic programs, propositional satisfiability (SAT), and translations between ASP and SAT.

2.1 Normal Logic Programs and Stable Models

We consider *normal logic programs* (NLPs) in the *propositional* case. The symbol "$\sim$" denotes *default negation*. A *default literal* is an atom, a, or its default negation, $\sim a$. We define shorthands $L^+ = \{a \mid a \in L\}$ and $L^- = \{a \mid \sim a \in L\}$ for a set of default literals L, and $\sim A = \{\sim a \mid a \in A\}$ for a set of atoms A. A program Π over the set of propositional atoms $\mathsf{atoms}(\Pi)$ consists of a finite set of rules r of the form

$$h \leftarrow a_1, \ldots, a_n, \sim b_1, \ldots, \sim b_m, \tag{1}$$

where $h \in \mathsf{atoms}(\Pi) \cup \{\bot\}$ and $a_i, b_j \in \mathsf{atoms}(\Pi)$. A rule consists of a *head*, $\mathsf{head}(r) = h$, and a *body*, $\mathsf{body}(r) = \{a_1, \ldots, a_n, \sim b_1, \ldots, \sim b_m\}$. This allows the shorthand $\mathsf{head}(r) \leftarrow \mathsf{body}(r)^+ \cup \sim \mathsf{body}(r)^-$ for (1). A rule r is a *fact* if $|\mathsf{body}(r)| = 0$. We define $\mathsf{head}(\Pi) = \bigcup_{r \in \Pi} \{\mathsf{head}(r)\}$ and $\mathsf{body}(\Pi) = \bigcup_{r \in \Pi} \{\mathsf{body}(r)\}$. The set of default literals of a program Π is $\mathsf{dlits}(\Pi) = \{a, \sim a \mid a \in \mathsf{atoms}(\Pi)\}$.

In ASP, we are interested in *stable models* [12] (or *answer sets*) of a program Π. An *interpretation* $M \subseteq \mathsf{atoms}(\Pi)$ defines which atoms of Π are true ($a \in M$) and which are false ($a \notin M$). An interpretation $M \subseteq \mathsf{atoms}(\Pi)$ is a *(classical) model* of Π if and only if $\mathsf{body}(r)^+ \subseteq M$ and $\mathsf{body}(r)^- \cap M = \emptyset$ imply $\mathsf{head}(r) \in M$ for each rule $r \in \Pi$. A model M is a stable model of a program Π if and only if there is no model $M' \subset M$ for Π^M, where

$$\Pi^M = \{\mathsf{head}(r) \leftarrow \mathsf{body}(r)^+ \mid r \in \Pi \text{ and } M \cap \mathsf{body}(r)^- = \emptyset\}$$

is called the *Gelfond-Lifschitz reduct* of Π with respect to M. We say that a program Π is *satisfiable* if it has a stable model, and *unsatisfiable* otherwise.

Given $a, b \in \mathsf{atoms}(\Pi)$, we say that b *depends directly* on a, denoted $a \leq_1 b$, if and only if there is a rule $r \in \Pi$ such that $b = \mathsf{head}(r)$ and $a \in \mathsf{body}(r)^+$. The *positive dependency graph* of Π, denoted by $\mathrm{Dep}^+(\Pi)$, is a directed graph with $\mathsf{atoms}(\Pi)$ and $\{\langle b, a\rangle \mid a \leq_1 b\}$ as the sets of vertices and edges, respectively. A NLP is *tight* if and only if its positive dependency graph is acyclic. We denote by $\mathsf{loop}(\Pi)$ the set of all loops in $\mathrm{Dep}^+(\Pi)$. Furthermore, the *external bodies* of a set of atoms A in Π is $\mathsf{eb}(A) = \{\mathsf{body}(r) \mid r \in \Pi,\ \mathsf{head}(r) \in A,\ \mathsf{body}(r)^+ \cap A = \emptyset\}$. A set $U \subseteq \mathsf{atoms}(\Pi)$ is *unfounded* if $\mathsf{eb}(U) = \emptyset$. We denote the *greatest unfounded set*, i.e., the union of all unfounded sets, of Π by $\mathsf{gus}(\Pi)$.

2.2 Propositional Satisfiability

Let X be a set of Boolean variables. Associated with every variable $x \in X$ there are two *literals*, the positive literal, denoted by x, and the negative literal, denoted by $\bar{x}$. A *clause* is a disjunction of distinct literals. We adopt the standard convention of viewing a clause as a finite set of literals and a CNF formula as a finite set of clauses. The sets of variables and literals appearing in a set of clauses $\mathcal{C}$ are denoted by $\mathsf{vars}(\mathcal{C})$ and $\mathsf{lits}(\mathcal{C})$.

A *truth assignment* τ associates a truth value $\tau(x) \in \{\mathsf{false}, \mathsf{true}\}$ with each variable $x \in X$. A truth assignment *satisfies* a set of clauses if it satisfies every clause in it. A clause is satisfied if it contains at least one satisfied literal, where a literal x (respectively, $\bar{x}$) is satisfied if $\tau(x) = \mathsf{true}$ (respectively, $\tau(x) = \mathsf{false}$). A clause set is *satisfiable* if there is a truth assignment that satisfies it, and *unsatisfiable* otherwise.

2.3 SAT as ASP

There is a natural linear-size translation from sets of clauses to normal logic programs so that the stable models of the encoding represent the satisfying truth assignments of the original clause set [13] *faithfully*, i.e., there is a bijective correspondence between the satisfying truth assignments and stable models of the translation. Given a clause set $\mathcal{C}$, this translation $\mathsf{nlp}(\mathcal{C})$ introduces a new atom c for each clause $C \in \mathcal{C}$, and atoms a_x and $\hat{a}_x$ for each variable $x \in \mathsf{vars}(\mathcal{C})$. The resulting NLP is then

$$\mathsf{nlp}(\mathcal{C}) := \bigcup_{x \in \mathsf{vars}(\mathcal{C})} \{\{a_x \leftarrow {\sim}\hat{a}_x\} \cup \{\hat{a}_x \leftarrow {\sim}a_x\}\} \cup \bigcup_{C \in \mathcal{C}} \{\bot \leftarrow {\sim}c\} \cup \tag{2}$$

$$\bigcup_{C \in \mathcal{C}} \{\{c \leftarrow a_x \mid x \in \mathsf{lits}(C)\} \cup \{c \leftarrow \hat{a}_x \mid \bar{x} \in \mathsf{lits}(C)\}\}. \tag{3}$$

The rules (2) encode the facts that (i) each variable is assigned an unambiguous truth value and that (ii) each clause in $\mathcal{C}$ must be satisfied, while (3) encodes that each clause is satisfied if at least one of its literals is satisfied.

2.4 ASP as SAT

Contrarily to the case of translating SAT into ASP, there is no modular[1] and faithful translation from normal logic programs to propositional logic [13]. Moreover, any faithful translation is potentially of exponential size when additional variables are not allowed[14] [2]. However, if a program Π satisfies the syntactic *tightness* condition, the answer sets of Π can be characterised faithfully by the classical models of a linear-size propositional formula called *Clark's completion* [19,20] of Π, defined using a Boolean variable x_a for each $a \in \mathsf{atoms}(\Pi)$ as

$$C(\Pi) = \bigwedge_{h \in \mathsf{atoms}(\Pi)} \left(x_h \Leftrightarrow \bigvee_{B \in \mathsf{body}(h)} \left(\bigwedge_{b \in B^+} x_b \wedge \bigwedge_{b \in B^-} \bar{x}_b \right) \right), \tag{4}$$

where $\mathsf{body}(h) = \{\mathsf{body}(r) \mid \mathsf{head}(r) = h\}$. For simplicity, we have the special cases that (i) if x_h is $\bot$ then the equivalence becomes the negation of the right hand side, and (ii) if $h \in \mathsf{facts}(\Pi)$ then the equivalence reduces to the clause $\{x_h\}$.

As in this paper, often one needs to consider the clausal representation of Boolean formulas. For a linear-size clausal translation of $C(\Pi)$, introduce additionally a new Boolean variable x_B for each $B \in \mathsf{body}(\Pi) \setminus \{\emptyset\}$. Using these new variables, we arrive at the *clausal completion*

$$\mathsf{comp}(\Pi) := \bigcup_{B \in \mathsf{body}(\Pi) \setminus \{\emptyset\}} \left\{ x_B \equiv \bigwedge_{b \in B^+} x_b \wedge \bigwedge_{b \in B^-} \bar{x}_b \right\} \cup \bigcup_{B \in \mathsf{body}(\bot)} \{\bar{x}_B\} \tag{5}$$

$$\cup \bigcup_{\substack{h \in \mathsf{head}(\Pi) \setminus \{\bot\} \\ h \notin \mathsf{facts}(\Pi)}} \left\{ x_h \equiv \bigvee_{B \in \mathsf{body}(h)} x_B \right\} \cup \bigcup_{h \in \mathsf{facts}(\Pi)} \{x_h\} \tag{6}$$

$$\cup \bigcup_{a \in \mathsf{atoms}(\Pi) \setminus \mathsf{head}(\Pi)} \{\bar{x}_a\}, \tag{7}$$

where the shorthands $x \equiv \bigwedge_{x_i \in X} x_i$ and $x \equiv \bigvee_{x_i \in X} x_i$ stand for the sets of clauses $\{\bar{x}_1, \ldots \bar{x}_n, x\} \cup \bigcup_{x_i \in X} \{x_i, \bar{x}\}$ and $\{x_1, \ldots x_n, \bar{x}\} \cup \bigcup_{x_i \in X} \{\bar{x}_i, x\}$, respectively. For an example of a logic program's clausal completion, see Fig. 1(left).

[1] Intuitively, for a modular translation, adding an atom to a program leads to a local change not involving the translation of the rest of the program [13].

[2] However, polynomial size propositional encodings using extra variables are known, e.g. [15,16]. Also, ASP as Propositional Satisfiability approaches for solving normal logic programs have been developed, e.g., ASSAT [17] (based on incrementally adding loop formulas) and ASP-SAT [18] (based on generating a classical model and testing its minimality).

3 Proof Systems for ASP and SAT

In this section we review concepts related to proof complexity (see, e.g., [5]) in the context of this paper, and discuss the relationship of Resolution and ASP Tableaux [7].

3.1 Propositional Proof Systems and Complexity

Formally, a *(propositional) proof system* is a polynomial-time computable predicate S such that a propositional expression E is unsatisfiable if and only if there is a *proof* p for which $S(E, p)$. A proof system is thus a polynomial-time procedure for checking the correctness of proofs in a certain format. While proof checking is efficient, finding short proofs may be difficult, or, generally, impossible since short proofs may not exist for a too weak proof system. As a measure of hardness of proving unsatisfiability of an expression E in a proof system S, the *(proof) complexity* of E in S is the *length* of the shortest proof of E in S. For a family $\{E_n\}$ of unsatisfiable expressions over increasing number of variables, the (asymptotic) complexity of $\{E_n\}$ is measured with respect to the sizes of E_n.

For two proof systems S,S', we say that S' *(polynomially) simulates* S if for all families $\{E_n\}$ it holds that $C_{S'}(E_n) \leq p(C_S(E_n))$ for all E_n, where p is a polynomial, and C_S and $C_{S'}$ are the complexities in S and S', respectively. If S simulates S' and vice versa, then S and S' are *polynomially equivalent*. If there is a family $\{E_n\}$ for which S' does not polynomially simulate S, we say that $\{E_n\}$ *separates* S from S', and S is *stronger* than S'.

3.2 Resolution

The well-known Resolution proof system (RES) for clause sets is based on the *resolution rule*. Let C, D be clauses, and x a Boolean variable. The resolution rule states that we can *directly derive* $C \cup D$ from $\{x\} \cup C$ and $\{\bar{x}\} \cup D$ by *resolving on* x.

A RES *derivation* of a clause C from a clause set $\mathcal{C}$ is a sequence of clauses $\pi = (C_1, C_2, \ldots, C_n)$, where $C_n = C$ and each C_i, where $1 \leq i < n$, is either (i) a clause in $\mathcal{C}$ (an *initial clause*), or (ii) derived with the resolution rule from two clauses C_j, C_k where $j, k < i$ (a *derived clause*). The *length* of π is n, the number of clauses occurring in it. Any derivation of the empty clause $\emptyset$ from $\mathcal{C}$ is a RES *proof* of $\mathcal{C}$.

Any RES proof $\pi = (C_1, C_2, \ldots, C_n)$ can be presented as a directed acyclic graph, in which the leafs are initial clauses, inner nodes are derived clauses, and the root is the empty clause. There are edges from C_i and C_j to C_k iff C_k has been directly derived from C_i and C_j using the resolution rule. Many *Resolution refinements*, in which the structure of the graph representation is restricted, have been proposed and studied. Of particular interest here is *Tree-like Resolution* (T-RES), in which it is required that proofs are represented by trees. This implies that a derived clause, if subsequently used multiple times in the proof, must be derived anew each time from initial clauses.

T-RES is a *proper* RES refinement, i.e., RES is stronger than T-RES [21]. On the other hand, it is well known that the DPLL method [22], the basis of most state-of-the-art SAT solvers, is polynomially equivalent to T-RES. However, conflict-learning DPLL is stronger than T-RES, and polynomially equivalent to RES under a slight generalisation [6].

3.3 ASP Tableaux

Although ASP solvers for normal logic programs have been available for many years, the deduction rules applied in such solvers have only recently been formally defined as a proof system, which we will here refer to as ASP Tableaux [7] (ASP-T).

An ASP tableau for a NLP Π is a binary tree of the following structure. The *root* of the tableau consists of the rules Π and the *entry* $\mathbf{F}\bot$ for capturing that $\bot$ is always false. The non-root nodes of the tableau are single *entries* of the form $\mathbf{T}a$ or $\mathbf{F}a$, where $a \in \mathsf{atoms}(\Pi) \cup \mathsf{body}(\Pi)$. As typical for tableau methods, entries are generated by *extending* a *branch* (a path from the root to a leaf node) by applying one of the rules in Fig.2; if the prerequisites of a rule hold in a branch, the branch can be extended with the entries specified by the rule. For convenience, we have the shorthand $\mathbf{t}l$ ($\mathbf{f}l$) for literals, defined as $\mathbf{T}l$ if l is positive (negative), and $\mathbf{F}l$ if l is negative (positive).

A branch is *closed under* the *deduction rules* (b)-(i) if the branch cannot be extended using the rules. A branch is *contradictory* if there are entries $\mathbf{T}a, \mathbf{F}a$ for some a. A branch is *complete* if it is contradictory or if there is the entry $\mathbf{T}a$ or $\mathbf{F}a$ for each $a \in \mathsf{atoms}(\Pi) \cup \mathsf{body}(\Pi)$ and the branch is closed under the deduction rules. A tableau is complete if all its branches are complete. A complete tableau from Π in which all branches are contradictory is an ASP-T proof of the unsatisfiability of Π. The *length* of an ASP-T proof is the number of entries in it. In Fig. 1 an ASP-T proof is presented for the program Π given on the left of the proof, with the rule applied for deducing each entry given in parenthesis.

$$
\begin{aligned}
\Pi &= \{a \leftarrow {\sim}a, b.\ b \leftarrow c.\ c \leftarrow {\sim}b\} \\
\mathsf{comp}(\Pi) &= \{\{\bar{x}_{\{{\sim}a,b\}}, \bar{x}_a\}, \{\bar{x}_{\{{\sim}a,b\}}, x_b\}, \\
&\quad \{x_{\{{\sim}a,b\}}, x_a, \bar{x}_b\}, \{\bar{x}_{\{c\}}, x_c\}, \\
&\quad \{x_{\{c\}}, \bar{x}_c\}, \{\bar{x}_{\{{\sim}b\}}, \bar{x}_b\}, \\
&\quad \{x_{\{{\sim}b\}}, x_b\}, \{\bar{x}_a, x_{\{{\sim}a,b\}}\}, \\
&\quad \{x_a, \bar{x}_{\{{\sim}a,b\}}\}, \{\bar{x}_b, x_{\{c\}}\}, \\
&\quad \{x_b, \bar{x}_{\{c\}}\}, \{\bar{x}_c, x_{\{{\sim}b\}}\}, \{x_c, \bar{x}_{\{{\sim}b\}}\}\}
\end{aligned}
$$

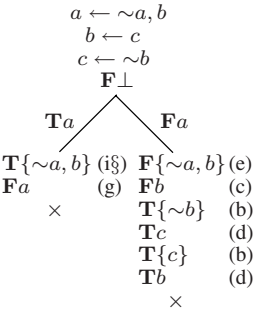


Fig. 1. A logic program Π, its clausal completion $\mathsf{comp}(\Pi)$, and an ASP-T proof for Π

Any branch B describes a *partial assignment* $\mathcal{A}$ on $\mathsf{atoms}(\Pi)$ in a natural way, i.e., if there is an entry $\mathbf{T}a$ ($\mathbf{F}a$, respectively) in B for $a \in \mathsf{atoms}(\Pi)$, then $(a, \mathsf{true}) \in \mathcal{A}$ ($(a, \mathsf{false}) \in \mathcal{A}$, respectively). ASP-T is a sound and complete proof system for normal logic programs [7], i.e., there is a complete non-contradictory ASP tableau from Π if and only if Π is satisfiable. Thus the assignment $\mathcal{A}$ described by a complete non-contradictory branch gives a stable model $M = \{a \in \mathsf{atoms}(\Pi) \mid (a, \mathsf{true}) \in \mathcal{A}\}$. As argumented in [7], current ASP solver implementations are tightly related to ASP-T, with the intuition that the cut rule is determinised with decision heuristics, while the deduction rules describe the propagation mechanism in ASP solvers. For instance, the noMore++ system [2] is a determinisation of the rules (a)-(g),(h§),(h†),(i§), while smodels [1] applies the same rules with the cut rule restricted to $\mathsf{atoms}(\Pi)$.

$$\frac{}{\mathbf{T}\phi \mid \mathbf{F}\phi}\ (\natural)$$

(a) Cut

$$\frac{h \leftarrow l_1, \ldots, l_n \quad \mathbf{t}l_1, \ldots, \mathbf{t}l_n}{\mathbf{T}\{l_1, \ldots, l_n\}}$$

(b) Forward True Body

$$\frac{\mathbf{F}\{l_1, \ldots, l_i, \ldots, l_n\} \quad \mathbf{t}l_1, \ldots, \mathbf{t}l_{i-1}, \mathbf{t}l_{i+1}, \ldots, \mathbf{t}l_n}{\mathbf{f}l_i}$$

(c) Backward False Body

$$\frac{h \leftarrow l_1, \ldots, l_n \quad \mathbf{T}\{l_1, \ldots, l_n\}}{\mathbf{T}h}$$

(d) Forward True Atom

$$\frac{h \leftarrow l_1, \ldots, l_n \quad \mathbf{F}h}{\mathbf{F}\{l_1, \ldots, l_n\}}$$

(e) Backward False Atom

$$\frac{h \leftarrow l_1, \ldots, l_i, \ldots, l_n \quad \mathbf{f}l_i}{\mathbf{F}\{l_1, \ldots, l_i, \ldots, l_n\}}$$

(f) Forward False Body

$$\frac{\mathbf{T}\{l_1, \ldots, l_i, \ldots, l_n\}}{\mathbf{t}l_i}$$

(g) Backward True Body

$$\frac{\mathbf{F}B_1, \ldots, \mathbf{F}B_m}{\mathbf{F}h}\ (\flat)$$

(h)

$$\frac{\mathbf{T}h \quad \mathbf{F}B_1, \ldots, \mathbf{F}B_{i-1}, \mathbf{F}B_{i+1}, \ldots, \mathbf{F}B_m}{\mathbf{T}B_i}\ (\sharp)$$

(i)

($\natural$): $\phi \in \mathsf{atoms}(\Pi) \cup \mathsf{body}(\Pi)$
($\flat$): § (Forward False Atom) or † (Well-Founded Negation) or ‡ (Forward Loop)
($\sharp$): § (Backward True Atom) or † (Well-Founded Justification) or ‡ (Backward Loop)
(§): $\mathsf{body}(h) = \{B_1, \ldots, B_m\}$
(†): $\{B_1, \ldots, B_m\} \subseteq \mathsf{body}(\Pi)$ and $h \in \mathsf{gus}(\{r \in \Pi \mid \mathsf{body}(r) \notin \{B_1, \ldots B_m\}\})$
(‡): $h \in L$, $L \in \mathsf{loop}(\Pi)$, $\mathsf{eb}(L) = \{B_1, \ldots, B_m\}$

Fig. 2. Rules in ASP Tableaux

Interestingly, ASP-T and T-RES are polynomially equivalent under the translations comp and nlp. Although the similarity of DPLL's unit propagation and propagation in ASP solvers is discussed in [9,10], here we want to stress the direct connection between ASP-T and T-RES.

Theorem 1. *Considering tight programs,* T-RES *under the translation* comp *can polynomially simulate* ASP-T.

The intuitive idea of the proof of Theorem 1 is the following. Consider again the tight NLP Π and the ASP-T proof T in Fig. 1. The completion $\mathsf{comp}(\Pi)$ is also shown in Fig. 1. We transform T into a binary *cut tree* T' where every entry generated by a deduction rule in T is replaced by an application of the cut rule on the corresponding entry. See Fig. 3 (left) for the cut tree corresponding to the ASP-T proof in Fig. 1. Now there is a T-RES proof of $\mathsf{comp}(\Pi)$ such that for any prefix p of an arbitrary branch B in T' there is a clause $C \in \pi$ contradictory to the partial assignment in p, i.e., there is the entry $\mathbf{F}a$ ($\mathbf{T}a$) in p for each corresponding positive literal x_a (negative literal $\bar{x}_a$) in C. Furthermore, each such C has a Tree-like Resolution derivation from $\mathsf{comp}(\Pi)$ of polynomial length w.r.t. the postfix of B starting directly after p. When reaching the root of T', we must have derived $\emptyset$ since it is contradictory with the empty assignment. The T-RES proof resulting from the cut tree in Fig. 3 (left) is shown in Fig. 3 (right).

The reverse direction, as stated by Theorem 2, follows from a similar argument.

Theorem 2. *Considering clause sets,* ASP-T *under the translation* nlp *can polynomially simulate* T-RES.

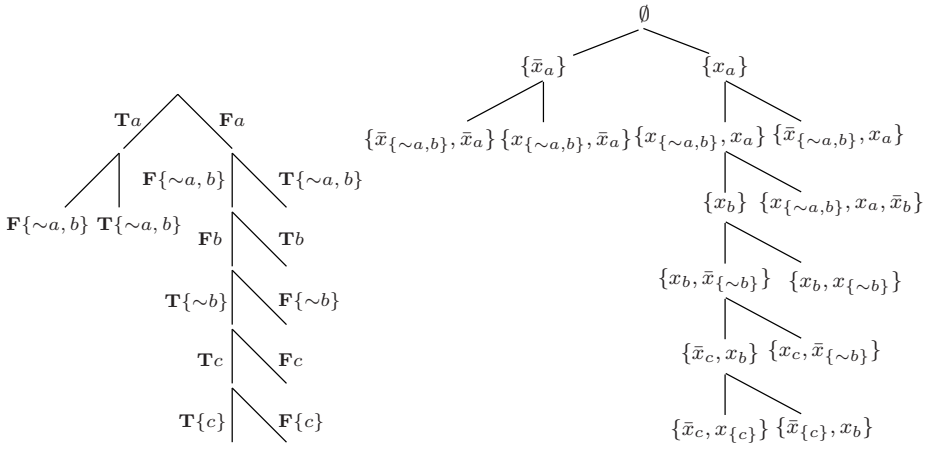


Fig. 3. Left: cut tree based on the ASP-T proof in Fig. 1. Right: resulting T-RES proof

4 Extended ASP Tableaux

We will now introduce an *extension rule* to ASP-T, which results in *Extended ASP Tableaux* (E-ASP-T), an extended tableau proof system for ASP. The idea is that one can define names for conjunctions of default literals, i.e., given two $l_1, l_2 \in \mathsf{dlits}(\Pi)$, the (elementary) extension rule allows adding the rule $p \leftarrow l_1, l_2$ to Π, where $p \notin \mathsf{atoms}(\Pi) \cup \{\bot\}$. It is essential that p is a new atom for preserving satisfiability.

When convenient, we will apply a generalisation of the elementary extension. By allowing one to introduce multiple bodies for p, the general extension rule[3] is

$$\Pi := \Pi \cup \bigcup_i \{p \leftarrow l_{i,1}, \ldots, l_{i,k_i} \mid l_{j,k} \in \mathsf{dlits}(\Pi),\ p \notin \mathsf{atoms}(\Pi) \cup \{\bot\}\}.$$

An E-ASP-T proof of program Π is an ASP-T proof T of $\Pi \cup E$, where E is a set of *extending rules* generated with the extension rule. The length of an E-ASP-T proof is the length of T plus the number of rules in E.

Since $\mathsf{head}(r) \notin \mathsf{atoms}(\Pi) \cup \{\bot\}$ for all extending rules $r \in E$, the extension rule does not affect the existence of stable models, i.e., for each stable model M of Π, there is a unique $N \subseteq \mathsf{atoms}(E) \setminus \mathsf{atoms}(\Pi)$ such that $M \cup N$ is a stable model of $\Pi \cup E$. Thus E-ASP-T is a sound and complete proof system.

5 Proof Complexity

In this section we study proof complexity theoretic issues related to E-ASP-T from several viewpoints: we (i) consider the relationship between E-ASP-T and Tseitin's Extended Resolution, (ii) give an explicit separation of E-ASP-T from ASP-T, and (iii) relate the extension rule to the effect of program simplification on proof lengths.

[3] Notice equivalent constructs can be introduced with the elementary rule. For example, using additional new atoms, bodies with more than two literals can be decomposed with balanced parentheses.

5.1 Relationship with Extended Resolution

E-ASP-T is motivated by Extended Resolution (E-RES), a proof system by Tseitin [11]. E-RES consists of the resolution rule and an extension rule which allows one to introduce equivalences of the form $x \equiv l_1 \wedge l_2$, where x is a new variable and l_1, l_2 literals in the clause set. In other words, given a clause set $\mathcal{C}$, one application of the extension rule adds the clauses $\{\bar{x}, l_1\}$, $\{\bar{x}, l_2\}$, and $\{x, \bar{l}_1, \bar{l}_2\}$ to $\mathcal{C}$. E-RES is known to be more powerful than RES; in fact, E-RES is polynomially equivalent with, e.g., extended Frege systems, and no superpolynomial proof complexity lower bounds are known for E-RES. We will now relate E-ASP-T with E-RES, and show that they are polynomially equivalent under the translations comp and nlp.

Theorem 3. E-RES *and* E-ASP-T *are polynomially equivalent proof systems in the sense that*

(i) considering tight normal logic programs, E-RES *under the translation* comp *polynomially simulates* E-ASP-T*, and*
(ii) considering clause sets, E-ASP-T *under the translation* nlp *polynomially simulates* E-RES*.*

Proof. (i): Let T be an E-ASP-T proof for a tight NLP Π, i.e., T is an ASP-T proof of $\Pi \cup E$, where E is the extension of Π. We use the shorthand x_l for the variable corresponding to default literal l in $\mathsf{comp}(\Pi)$, i.e., $x_l = x_a$ ($x_l = \bar{x}_a$) if $l = a$ ($l = {\sim}a$) for $a \in \mathsf{atoms}(\Pi)$. By Theorem 1 there is a polynomial T-RES proof for $\mathsf{comp}(\Pi \cup E)$. Since $\mathsf{head}(E) \cap (\mathsf{atoms}(\Pi) \cup \{\bot\}) = \emptyset$, the clauses introduced for $\mathsf{head}(E)$ in $\mathsf{comp}(\Pi \cup E)$ can be seen as extensions in E-RES, i.e., for each $h \leftarrow l_1, l_2 \in E$ there are the clauses $x_h \equiv x_{l_1} \wedge x_{l_2}$ in $\mathsf{comp}(\Pi \cup E)$. Thus there is an extension E' for $\mathsf{comp}(\Pi)$ such that the T-RES proof of $\mathsf{comp}(\Pi \cup E)$ is an E-RES proof of $\mathsf{comp}(\Pi)$.

(ii): Let $\pi = (C_1, \ldots, C_n = \emptyset)$ be an E-RES proof of a set of clauses $\mathcal{C}$. Let E be the set of clauses generated with the extension rule in π. We introduce shorthands for atoms corresponding to literals, i.e., $a_l = a_x$ ($a_l = {\sim}a_x$) if $l = x$ ($l = \bar{x}$) for $x \in \mathsf{atoms}(\mathcal{C})$. We add the following rules to $\mathsf{nlp}(\mathcal{C})$ with the ASP extension rule: $a_x \leftarrow a_{l_1}, a_{l_2}$ for each extension $x \equiv l_1 \wedge l_2$; $c \leftarrow a_l$ for each $l \in C$ in π such that $C \notin \mathcal{C}$; and $p_1 \leftarrow c_1$ and $p_i \leftarrow c_i, p_{i-1}$ for each $C_i \in \pi$ and $2 \leq i < n$.

An E-ASP-T proof for $\mathsf{nlp}(\mathcal{C})$ is generated as follows. From $i = 1$ to $n - 1$ apply the cut rule on p_i in the branch with $\mathbf{T}p_j$ for all $j < i$. We notice that each branch with $\mathbf{F}p_i$ and $\mathbf{T}p_j$ for all $j < i$ closes without further application of the cut. We can deduce $\mathbf{F}c_i$ from $\mathbf{F}p_i$. Now either (i) $C_i \in \mathcal{C}$, (ii) C_i is a derived clause, or (iii) $C_i \in E$. For instance, if $C_i = \{\bar{x}, l_1\}$ from the extension $x \equiv l_1 \wedge l_2$, then from $c_i \leftarrow {\sim}a_x$ and $c_i \leftarrow a_{l_1}$ we deduce $\mathbf{T}a_x$ and $\mathbf{F}a_{l_1}$. The branch closes as $\mathbf{T}\{a_{l_1}, a_{l_2}\}$ and $\mathbf{T}a_{l_1}$ are deduced from $a_x \leftarrow a_{l_1}, a_{l_2}$. Other cases are similar.

Now, consider the branch with $\mathbf{T}p_i$ for all $i = 1 \ldots n - 1$. The empty clause C_n is obtained by resolving $C_j = \{x\}$ and $C_k = \{\bar{x}\}$, $j, k < n$. Thus we can deduce $\mathbf{T}c_j$ and $\mathbf{T}c_k$ from $p_j \leftarrow c_j, p_{j-1}$ and $p_k \leftarrow c_k, p_{k-1}$, respectively, and furthermore, $\mathbf{T}a_x$ and $\mathbf{F}a_x$ from $c_j \leftarrow l$ and $c_k \leftarrow \bar{l}$ ($l = x$ or $l = \bar{x}$), closing the branch. The obtained contradictory ASP tableau is of linear length w.r.t. π. □

5.2 Pigeonhole Principle Separates Extended ASP Tableaux from ASP Tableaux

As an example, we now consider a family of normal logic programs $\{\Pi_n\}$ which separates E-ASP-T from ASP-T, i.e., we give an explicit polynomial length proof of Π_n for which ASP-T has exponential length minimal proofs with respect to n. We will consider this family also in the experiments of this paper.

The program family $\{\mathrm{PHP}_n^{n+1}\}$ in question is the following typical encoding of the *pigeon-hole principle* as a normal logic program:

$$\mathrm{PHP}_n^{n+1} := \bigcup_{1\leq i\leq n+1} \{\bot \leftarrow \sim p_{i,1}, \ldots, \sim p_{i,n}\} \cup \bigcup_{\substack{1\leq i<j\leq n+1\\ 1\leq k\leq n}} \{\bot \leftarrow p_{i,k}, p_{j,k}\} \quad (8)$$

$$\cup \bigcup_{\substack{1\leq i\leq n+1\\ 1\leq j\leq n}} \{\{p_{i,j} \leftarrow \sim p'_{i,j}\} \cup \{p'_{i,j} \leftarrow \sim p_{i,j}\}\} \quad (9)$$

In the above, $p_{i,j}$ has the interpretation that pigeon i sits in hole j. The rules in (8) require that (i) each pigeon must sit in some hole and that (ii) no two pigeons can sit in the same hole. The rules in (9) enforce that for each pigeon and each hole, the pigeon either sits in the hole or does not sit in the hole. Each PHP_n^{n+1} is unsatisfiable since there is no bijective mapping from an $(n+1)$-element set to an n-element set.

Theorem 4. *The complexity of* $\{\mathrm{PHP}_n^{n+1}\}$ *with respect to* n *is polynomial in* E-ASP-T *and exponential in* ASP-T

Proof. (i): Following Cook's extension [23] for achieving a polynomial-length E-RES proof of a clausal encoding of the pigeonhole principle[4], we define the polynomial size program extension

$$\mathrm{EXT}^l := \bigcup_{\substack{1\leq i\leq l\\ 1\leq j\leq l-1}} \{\{e_{i,j}^l \leftarrow e_{i,j}^{l+1}\} \cup \{e_{i,j}^l \leftarrow e_{i,l}^{l+1}, e_{l+1,j}^{l+1}\}\} \quad (10)$$

for $1 \leq l \leq n$, where each $e_{i,j}^{n+1}$ is interpreted as $p_{i,j}$.

Although not explicitly given by Cook, the extension given in [23] does not seem to yield a polynomial length *tree-like* proof of the clausal representation, so Theorem 3 does not directly imply a polynomial length ASP-T proof for $\mathrm{PHP}_n^{n+1} \cup \bigcup_{1\leq l\leq n} \mathrm{EXT}^l$. However, given the polynomial length E-RES proof[5] $\pi = (C_1, C_2, \ldots, C_n = \emptyset)$ of the clausal representation, we can follow the general strategy given in the proof of Theorem 3 for defining an additional extension $E(\pi)$ which allows a polynomial length ASP-T proof for the resulting program

$$\mathrm{EPHP}_n^{n+1} := \mathrm{PHP}_n^{n+1} \cup \bigcup_{1\leq l\leq n} \mathrm{EXT}^l \cup E(\pi).$$

(ii): $\mathsf{comp}(\mathrm{PHP}_n^{n+1})$ consists of the clausal encoding of the pigeon-hole principle and additional clauses (tautologies) for rules of the form $a \leftarrow \sim a'$, $a' \leftarrow \sim a$. Assume

[4] The particular encoding is $\bigcup_{1\leq i\leq n+1}\{\bigvee_{j=1}^n x_{i,j}\} \cup \bigcup_{1\leq i<i'\leq n+1, 1\leq j\leq n}\{\neg x_{i,j} \vee \neg x_{i',j}\}$.

[5] The intuitive idea is that the extension allows for reducing PHP_n^{n+1} to PHP_{n-1}^n with a polynomial number of Resolution steps. Due to space constraints we do not give π explicitly here.

now that there is a polynomial ASP-T proof for PHP_n^{n+1}. By Theorem 1 there is a polynomial T-RES of $\mathsf{comp}(\mathrm{PHP}_n^{n+1})$. It is easy to see that the additional tautologies in $\mathsf{comp}(\mathrm{PHP}_n^{n+1})$ do not help in the resolution proof. Thus there is a polynomial length T-RES proof for the clausal pigeonhole encoding. However, this contradicts the fact that the complexity of the clausal pigeonhole principle is exponential w.r.t. n for (Tree-like) Resolution [24]. □

In fact, Theorem 4 is also witnessed by *non-tight* programs. Consider the family $\{\mathrm{PHP}_n^{n+1} \cup \{p_{i,j} \leftarrow p_{i,j} \mid 1 \leq i \leq n+1, 1 \leq j \leq n\}\}$, which is non-tight with the additional self-loops $\{p_{i,j} \leftarrow p_{i,j}\}$, but preserves (un)satisfiability of PHP_n^m for all n, m. Since the self-loops do not contribute to the proofs of PHP_n^{n+1}, ASP-T still has exponential length minimal proofs for these programs, while the E-ASP-T proof presented in the proof of Theorem 4 is still valid.

5.3 Program Simplification and Complexity

We will now give an interesting corollary of Theorem 4, addressing the effect of program simplification on the length of proofs.

Tightly related to the development of efficient solver implementations for resolving ASP programs arising from practical applications is the development of techniques for *simplifying* programs. Efficient program simplification through *local transformation rules* becomes especially important as practically relevant programs are often produced automatically, because often a high number of redundant constraints is produced in the process. While various satisfiability-preserving local transformation rules for simplifying logic programs have been introduced (see, e.g., [25]), the effect of applying such transformations on the lengths of proofs has not received attention.

Taking a first step into this direction, we now show that even simple transformation rules may have a drastic negative effect on proof complexity. Consider the local transformation rule $\mathsf{red}(\Pi) := \Pi \setminus \{r \in \Pi \mid \mathsf{head}(r) \notin \mathsf{body}(\Pi)\}$. The rules removed by red are redundant with respect to satisfiability of the program in the sense that red preserves *visible equivalence* [16]. The visible equivalence relation takes the interfaces of programs into account: $\mathsf{atoms}(\Pi)$ is partitioned into $\mathsf{v}(\Pi)$ and $\mathsf{h}(\Pi)$ determining the *visible* and the *hidden* atoms in Π, respectively. Programs Π_1 and Π_2 are visibly equivalent, denoted by $\Pi_1 \equiv_{\mathrm{v}} \Pi_2$, if and only if $\mathsf{v}(\Pi_1) = \mathsf{v}(\Pi_2)$ and there is a bijective correspondence between the stable models of Π_1 and Π_2 mapping each $a \in \mathsf{v}(\Pi_1)$ onto itself. Defining $\mathsf{v}(\Pi) = \mathsf{v}(\mathsf{red}(\Pi)) = \mathsf{atoms}(\mathsf{red}(\Pi))$, one can see that $\mathsf{red}(\Pi) \equiv_{\mathrm{v}} \Pi$.

A polynomial time, satisfiability-preserving simplification algorithm $\mathsf{red}^*(\Pi)$ is obtained by closing program Π under red. However, notice that, in the worst case when we define $\mathsf{v}(\mathrm{EPHP}_n^{n+1}) = \mathsf{v}(\mathrm{PHP}_n^{n+1}) = \mathsf{atoms}(\mathrm{PHP}_n^{n+1})$, we have $\mathsf{red}^*(\mathrm{EPHP}_n^{n+1}) = \mathrm{PHP}_n^{n+1}$. Thus, by Theorem 4, red^* transforms a program family having polynomial complexity in ASP Tableaux into one with exponential complexity with respect to n.

6 Experiments

We evaluate empirically how well current state-of-the-art ASP solvers can make use of the additional structure introduced to programs using the extension rule. We run the solvers smodels [1] (version 2.32, a widely used lookahead solver), clasp [4] (rc4,

with many techniques—including conflict learning—adopted from DPLL-based SAT solvers), and cmodels [18] (version 3.66, a SAT-based ASP solver running the conflict-learning SAT solver zChaff version 2004.11.15 as the back-end). The experiments are run on standard PCs with 2-GHz AMD 3200+ processors under Linux.

First, we investigate whether ASP solvers are able to benefit from the extension in EPHP_n^{n+1}. We compare the number of decisions and running times of each of the solvers on PHP_n^{n+1}, $\mathrm{CPHP}_n^{n+1} := \mathrm{PHP}_n^{n+1} \cup \bigcup_{1 \leq l \leq n} \mathrm{EXT}^l$, and EPHP_n^{n+1}. By Theorem 4 the solvers should in theory be able to exhibit polynomially scaling number of decisions for EPHP_n^{n+1}. In fact with conflict-learning this might also be possible for CPHP_n^{n+1} due to the tight correspondence with conflict-learning SAT solvers and Resolution. The results for $n = 10 \ldots 12$ are shown in Table 1. While the number of decisions for the conflict-learning clasp and cmodels is somewhat reduced by the extensions, the solvers do not seem to be able to reproduce the polynomial size proofs, and we do not observe a dramatic change in the running times. With a timeout of 2 hours, smodels gives no answer for $n = 12$ on PHP_n^{n+1} or CPHP_n^{n+1}. However, for EPHP_n^{n+1} smodels returns without any branching, which should be due to the fact that smodels's complete lookahead notices that by branching on the critical extension atoms (as in part (ii) of the proof of Theorem 4) the false branch closes immediately. With this in mind, an interesting further study out of the scope of this paper would be the possibilities of integrating conflict learning techniques with (partial) lookahead.

In the second experiment, we study the effect of having a modest number of redundant rules on the behaviour of ASP solvers. For this we apply the following procedure $\textsc{AddRandomRedundancy}(\Pi, n, p)$:

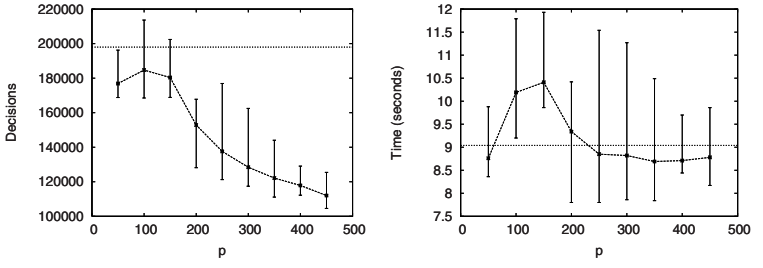
1. **For** $i = 1$ **to** $\lfloor \frac{p}{100} n \rfloor$:
   1a. Randomly select $l_1, l_2 \in \mathsf{dlits}(\Pi)$ such that $l_1 \neq l_2$.
   1b. $\Pi := \Pi \cup \{r_i \leftarrow l_1, l_2\}$, where $r_i \notin \mathsf{atoms}(\Pi)$
2. **Return** Π

Given a program Π, the procedure iteratively adds rules of the form $r_i \leftarrow l_1, l_2$ to Π, where l_1, l_2 are random default literals currently in the program and r_i is a new atom. The number of introduced rules is $p\%$ of the integer n.

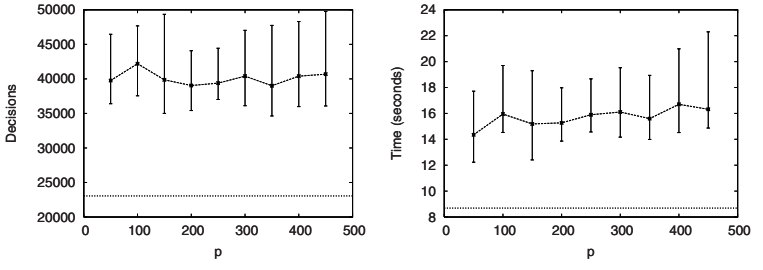
In Fig. 4, the median, minimum, and maximum number of decisions and running times for the solvers on $\textsc{AddRandomRedundancy}(\mathrm{PHP}_n^{n+1}, n, p)$ are shown for $p = 50, 100 \ldots, 450$ over 15 trials at each data point. The mean number of decisions (left) and running times (right) on the original PHP_n^{n+1} are presented by the horizontal

Table 1. Results on PHP_n^{n+1}, CPHP_n^{n+1}, and EPHP_n^{n+1} with timeout (-) of 2 hours

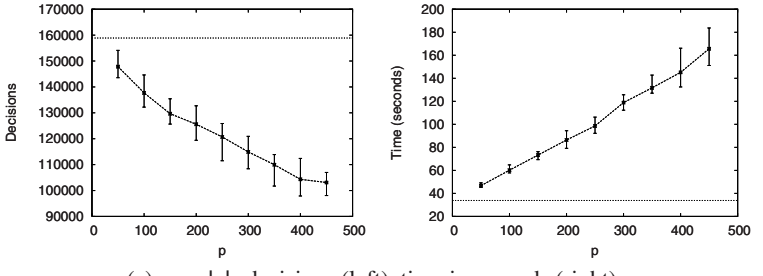
		Time (s)			Decisions		
Solver	n	PHP_n^{n+1}	CPHP_n^{n+1}	EPHP_n^{n+1}	PHP_n^{n+1}	CPHP_n^{n+1}	EPHP_n^{n+1}
smodels	10	32.28	120.24	9.28	158878	141177	0
smodels	11	471.54	1828.40	23.07	1885949	1619703	0
smodels	12	-	-	52.20	-	-	0
clasp	10	8.60	7.78	19.26	197982	114840	38842
clasp	11	72.78	62.74	97.23	1072358	574874	116534
clasp	12	900.33	1046.86	881.90	7787578	4964309	646278
cmodels	10	1.91	2.23	27.42	9455	9916	20615
cmodels	11	7.99	10.28	70.39	23058	26283	38648
cmodels	12	48.36	56.70	270.63	87864	98994	97745



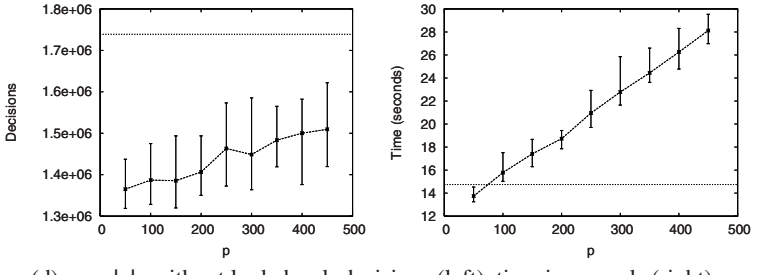
(a) clasp decisions (left), time in seconds (right)



(b) cmodels decisions (left), time in seconds (right)



(c) smodels decisions (left), time in seconds (right)



(d) smodels without lookahead: decisions (left), time in seconds (right)

Fig. 4. Effects of adding randomly generated redundant rules to PHP_n^{n+1}

lines. Notice that the number of added atoms and rules is linear to n, which is negligible to the number of atoms (in the order of n^2) and rules (n^3) in PHP_n^{n+1}. For similar running times, the number of holes n is 10 for clasp and smodels and 11 for cmodels. The results are very interesting: each of the solvers seems to react individually to the added redundancy. For cmodels (b), only a few added redundant rules are enough to worsen its behaviour. For smodels (c), the number of decisions decreases linearly with

the number of added rules. However, the running times grow fast at the same time, most probably due to smodels's lookahead. We also ran the experiment for smodels (d) without using lookahead. This had a visible effect on the number of decisions, showing a benefit from the added rules compared to smodels on PHP_n^{n+1}.

The most interesting effect is seen for clasp; clasp benefits from the added rules w.r.t. the number of decision, while the running times stay similar on the average, contrarily to the other solvers. In addition to this robustness against redundancy, we believe that this shows promise for further exploiting redundancy added in a controlled way during search; the added rules give new possibilities to branch on definitions which were not available in the original program. However, for benefiting from redundancy with running times in mind, optimised lightweight propagation mechanisms are essential.

As a final remark, an interesting observation is that the effect of the transformation presented in [8], which enables smodels to branch on the bodies of rules, having an exponential effect on the proof complexity of a particular program family, can be equivalently obtained by applying the ASP extension rule. This may in part explain the effect on adding redundancy on the number of decision made by smodels.

7 Conclusions

We introduce Extended ASP Tableaux, an extended tableau calculus for normal logic programs under the stable model semantics. We study the strength of the calculus, showing a tight correspondence with Extended Resolution, which is among the most powerful known propositional proof systems. This sheds further light on the relation of ASP and propositional satisfiability solving and their underlying proof systems, something which we believe is for the benefit of both of the communities.

Furthermore, this work shows the intricate nature of the interplay of structure and the hardness of solving ASP instances. We anticipate that controlled use of the extension rule is possible and will yield performance gains by considering in more detail the structural properties of programs in particular problem domains. One could also consider implementing branching on any possible formula *inside* a solver. However, this would require novel heuristics, since choosing the formula to branch on from the exponentially many alternatives is nontrivial and is not applied in current solvers. We find this an interesting future direction of research. Another important research direction set forth by this study is a more in-depth investigation into the effect of program simplification on the hardness of solving ASP instances.

Acknowledgements. Financial support from Academy of Finland (grant #211025), Helsinki Graduate School in Computer Science and Engineering, Emil Aaltonen Foundation, the Finnish Cultural Foundation (EO), the Technological Foundation TES, and the Nokia Foundation (EO) is gratefully acknowledged.

References

1. Simons, P., Niemelä, I., Soininen, T.: Extending and implementing the stable model semantics. Artificial Intelligence 138(1-2), 181–234 (2002)
2. Anger, C., Gebser, M., Linke, T., Neumann, A., Schaub, T.: The nomore++ approach to answer set solving. In: Sutcliffe, G., Voronkov, A. (eds.) LPAR 2005. LNCS(LNAI), vol. 3835, pp. 95–109. Springer, Heidelberg (2005)

3. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV system for knowledge representation and reasoning. ACM TOCL 7(3), 499–562 (2006)
4. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Conflict-driven answer set solving. In: IJCAI, pp. 286–392 (2007)
5. Beame, P., Pitassi, T.: Propositional proof complexity: Past, present, and future. Bulletin of the EATCS 65, 66–89 (1998)
6. Beame, P., Kautz, H., Sabharwal, A.: Towards understanding and harnessing the potential of clause learning. Journal of Artificial Intelligence Research 22, 319–351 (2004)
7. Gebser, M., Schaub, T.: Tableau calculi for answer set programming. In: Etalle, S., Truszczyński, M. (eds.) ICLP 2006. LNCS, vol. 4079, pp. 11–25. Springer, Heidelberg (2006)
8. Anger, C., Gebser, M., Janhunen, T., Schaub, T.: What's a head without a body? In: ECAI, pp. 769–770. IOS Press, Amsterdam (2006)
9. Giunchiglia, E., Maratea, M.: On the relation between answer set and SAT procedures (or, between cmodels and smodels). In: Gabbrielli, M., Gupta, G. (eds.) ICLP 2005. LNCS, vol. 3668, pp. 37–51. Springer, Heidelberg (2005)
10. Gebser, M., Schaub, T.: Characterizing ASP inferences by unit propagation. In: LaSh ICLP Workshop, pp. 41–56 (2006)
11. Tseitin, G.: On the complexity of derivation in propositional calculus. In: Automation of Reasoning 2: Classical Papers on Computational Logic, pp. 466–483. Springer, Heidelberg (1983)
12. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: ICLP, pp. 1070–1080. MIT Press, Cambridge (1988)
13. Niemelä, I.: Logic programs with stable model semantics as a constraint programming paradigm. Annals of Mathematics and Artificial Intelligence 25(3-4), 241–273 (1999)
14. Lifschitz, V., Razborov, A.: Why are there so many loop formulas? ACM Transactions on Computational Logic 7(2), 261–268 (2006)
15. Ben-Eliyahu, R., Dechter, R.: Propositional semantics for disjunctive logic programs. Annals of Mathematics and Artificial Intelligence 12(1-2), 53–87 (1994)
16. Janhunen, T.: Some (in)translatability results for normal logic programs and propositional theories. Journal of Applied Non-Classical Logics 16(1-2), 35–86 (2006)
17. Lin, F., Zhao, Y.: ASSAT: Computing answer sets of a logic program by SAT solvers. Artificial Intelligence 157(1–2), 115–137 (2004)
18. Giunchiglia, E., Lierler, Y., Maratea, M.: Answer set programming based on propositional satisfiability. Journal of Automated Reasoning 36(4), 345–377 (2006)
19. Clark, K.: Negation as failure. In: Readings in nonmonotonic reasoning, pp. 311–325. Morgan Kaufmann Publishers, San Francisco (1987)
20. Fages, F.: Consistency of Clark's completion and existence of stable models. Journal of Methods of Logic in Computer Science 1, 51–60 (1994)
21. Ben-Sasson, E., Impagliazzo, R., Wigderson, A.: Near optimal separation of tree-like and general resolution. Combinatorica 24(4), 585–603 (2004)
22. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem proving. Communications of the ACM 5(7), 394–397 (1962)
23. Cook, S.A.: A short proof of the pigeon hole principle using extended resolution. SIGACT News 8(4), 28–32 (1976)
24. Haken, A.: The intractability of resolution. TCS 39(2-3), 297–308 (1985)
25. Eiter, T., Fink, M., Tompits, H., Woltran, S.: Simplifying logic programs under uniform and strong equivalence. In: Lifschitz, V., Niemelä, I. (eds.) Logic Programming and Nonmonotonic Reasoning. LNCS (LNAI), vol. 2923, pp. 87–99. Springer, Heidelberg (2003)

Querying and Repairing Inconsistent Databases Under Three-Valued Semantics

Sergio Greco and Cristian Molinaro

DEIS, Univ. della Calabria, 87030 Rende, Italy
{greco,cmolinaro}@deis.unical.it

Abstract. The problem of managing and querying inconsistent databases has been deeply investigated in the last few years. As, in the general case, the problem of consistent query answering is hard, most of the techniques so far proposed have an exponential complexity. Moreover, polynomial techniques have been proposed only for restricted forms of constraints (e.g. functional dependencies) and queries.

In this paper we present a technique which allows us to compute "approximated" consistent answers in polynomial time, for general constraints and queries. The paper presents a three-valued semantics for constraint satisfaction, called *deterministic*, and considers three valued databases. Thus, a repaired database is a database where atoms may be either true or undefined (whereas missing atoms are false) that satisfies constraints under the deterministic three valued semantics. We show that in querying possibly inconsistent databases the answer is safe (true and false atoms in the answers are, respectively, true and false under the classical two-valued semantics) and the answers can be computed in polynomial time. We also show that deterministic answers can be computed by rewriting constraints into logic programs.

1 Introduction

Integrity constraints represent an important source of information about the real world. They are usually used to define constraints on data (functional dependencies, inclusion dependencies, etc.). Since the satisfaction of integrity constraints cannot be, generally, guaranteed, in the evaluation of queries, we must compute answers which are consistent with the integrity constraints. The presence of inconsistencies might arise, for instance, when the database is obtained from the integration of different information sources. 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.

In the presence of inconsistencies, an important problem investigated in recent years consists in characterizing and retrieving the consistent information from the database. It is well known that the presence of inconsistent data can be resolved by "repairing" the database, i.e. by providing a computational mechanism that ensures consistent "scenarios" of the information (repairs) are obtained in

V. Dahl and I. Niemelä (Eds.): ICLP 2007, LNCS 4670, pp. 149–164, 2007.

an inconsistent environment. However, there are many different ways of repairing a database, even if we limit ourselves to the minimal ones. So it is natural to leave the database inconsistent, so that no piece of information is lost, and to identify pieces of data which are consistent in answering queries, i.e. to consider the information present in every repaired database. This leads to the notion of consistent query answer (CQA): an element of query result in every repaired database.

Example 1. Consider the database schema $Teaches(Course, Prof)$ with the integrity constraint $Course \rightarrow Prof$ and the following (inconsistent) database $\mathcal{DB}$.

Course	*Prof*
c_1	p_1
c_2	p_2
c_2	q_2
c_3	p_3
c_3	q_3

$\mathcal{DB}$

Course	*Prof*
c_1	p_1
c_2	p_2
c_3	p_3

$\mathcal{DB}_1$

Course	*Prof*
c_1	p_1
c_2	p_2
c_3	q_3

$\mathcal{DB}_2$

Course	*Prof*
c_1	p_1
c_2	q_2
c_3	p_3

$\mathcal{DB}_3$

Course	*Prof*
c_1	p_1
c_2	q_2
c_3	q_3

$\mathcal{DB}_4$

There are four possible repaired databases: $\mathcal{DB}_1$, $\mathcal{DB}_2$, $\mathcal{DB}_3$, $\mathcal{DB}_4$ obtained by deleting minimal sets of tuples which make the repaired database consistent. Moreover, while we are not able to answer the query $Teaches(c_2, X)$, asking for the professor teaching course c_2, we are able to answer the query $Teaches(c_1, X)$ asking for the professor teaching course c_1. □

Therefore, it is very important, in the presence of inconsistent data, to compute the set of consistent answers, but also to know which facts are unknown and if there are possible repairs for the database.

The problem of managing and querying inconsistent databases has been deeply investigated in the last years. As, in the general case, the problem of consistent query answering is hard, most of the techniques so far proposed have an exponential complexity. Moreover, polynomial techniques have been proposed only for restricted forms of constraints (e.g. functional dependencies) and queries.

In this paper we present a technique which allows us to compute "approximated" consistent answers in polynomial time, for general constraints and queries. The paper presents a three-valued semantics for constraint satisfaction, called *deterministic*, and considers three valued databases. Thus, a repaired database is a database where atoms may be either true or undefined (whereas missing atoms are false) that satisfies constraints under the deterministic (three-valued) semantics. We show that in querying inconsistent databases the answer is sound (true and false atoms in the answers are also true and false, respectively, under the classical two-valued semantics) and can be computed in polynomial time. We also show that deterministic answers can be computed by rewriting constraints into logic programs.

Related Work

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 [3,22].

Arenas et al. [3] proposed a method to compute consistent query answers based on query rewriting. This technique is simple, but it has a very limited applicability (first-order queries without disjunction or quantification, and binary universal integrity constraints). Recently, that approach has been generalized by Fuxman and Miller [13] to allow restricted existential quantification in queries in the context of primary key FDs.

Several works have considered the use of logic programs to capture repairs as answer sets of logic programs with negation and disjunction [4,15,16]. These approaches are quite general, being able to handle arbitrary universal constraints and first-order queries. However, the problem of deciding whether an atom is a member of all answer sets of such a logic program is Π_2^p-complete and for the class of programs derived from the rewriting of constraints is in Π_2^p and $co\mathcal{NP}$-hard.

The use of preferences and priorities, as well as the definition of constructs to define unfeasible repairs and answers, have been also proposed [16,17,12]. However, the use of preferences, in most of the approaches proposed, further increases the computational complexity. A technique for the computation of repairs based on updating both tuples and attribute-values has been proposed by Wijsen [23], whereas special classes of constraints have been studied in [6].

Three-valued interpretation of database theories have been proposed as well and technique for query answering under LCWA (local CWA) computing a three-valued interpretation that approximates all two-valued interpretation of a database theory has been presented in [11].

For recent surveys on repairing and querying inconsistent databases we refer to the recent papers of Bertossi [5] and Chomicki [10].

1.1 Contributions

The novelty of our approach consists in i) the definition of a formal declarative three-valued semantics, called *deterministic*, for constraint satisfaction under three-valued databases, ii) the definition of (three-valued) repairs under such a semantics, iii) the definition of deterministic consistent query answers for (stratified) Datalog queries over databases with integrity constraints. More specifically:

- We consider databases whose atoms may be either true or undefined (missing tuples are assumed to be false). The reason for considering three-valued databases is that in the computation of repairs under deterministic (three-valued) semantics, some of the atoms may not be assumed to be true or false and they are assumed to be undefined.
- As databases may be three-valued, we propose a different semantics for the satisfaction of integrity constraints which for standard (two-valued) databases coincides with the classical semantics.

- We propose three-valued repairs consisting of substitutions which make the truth value of database atoms *true*, *false* or *undefined*. We show that the set of three-valued repairs defines a lower semi-lattice whose top elements are standard (two-valued) repairs and whose bottom element defines the deterministic repair.
- CQA can be computed by considering the deterministic repaired database (i.e. the database repaired by means of the deterministic repair). We show that CQA is sound (true and false atoms in the answers are, respectively, true and false under the classical two-valued semantics), but not complete.
- Finally, we show that deterministic repairs and answers can be computed in polynomial time and that the deterministic answers can be obtained by rewriting constraints and queries into a logic program and computing the fixpoint of this program.

2 Preliminaries

We assume familiarity with relational database theory, logic programs and deductive databases [1,14].

Predicate symbols are partitioned into two distinct sets: *base predicates* and *derived predicates*. Base predicates correspond to database relations defined over a given domain and they do not appear in the head of any rule; derived predicates are defined by means of rules. Given a database $\mathcal{DB}$ and a program $\mathcal{P}$, $\mathcal{P}_{\mathcal{DB}}$ denotes the program derived from the union of $\mathcal{P}$ with the facts in $\mathcal{DB}$, i.e. $\mathcal{P}_{\mathcal{DB}} = \mathcal{P} \cup \mathcal{DB}$. In the following a tuple t of a relation r will also be denoted as a fact $r(t)$. Given a set of ground atoms $\mathcal{DB}$ and a predicate symbol p, $\mathcal{DB}[p]$ denotes the set of p-tuples in $\mathcal{DB}$. The semantics of $\mathcal{P}_{\mathcal{DB}}$ is given by the set of its stable models by considering their intersection (*certain semantics* or *cautious reasoning*). A *(Datalog) query* Q is a pair $(g, \mathcal{P})$ where g is a predicate symbol, called the *query goal*, and $\mathcal{P}$ is a (Datalog) program. The answer to a Datalog query $Q = (g, \mathcal{P})$ over a database $\mathcal{DB}$ (under the certain semantics) is given by $\mathcal{DB}'[g]$ where $\mathcal{DB}' = \bigcap_{M \in \mathcal{SM}(\mathcal{P}_{\mathcal{DB}})} M$. A (relational) query can be expressed by means of 'safe' non recursive Datalog, even though alternative equivalent languages could be used [1].

3 Databases and Integrity Constraints

Databases contain, other than data, intentional knowledge expressed by means of integrity constraints. Database schemata contain the knowledge of the structure of data, i.e. they make constraints on the form the data must have. The relationships among data are usually defined by constraints such as functional dependencies, inclusion dependencies and others. Integrity constraints, which express information that is not directly derivable from the database, are introduced to prevent the insertion or deletion of data which could produce incorrect states. They are used to restrict the state a database can take and provide information on the relationships among data.

3.1 Integrity Constraints

Generally, a database $\mathcal{DB}$ has an associated schema $\langle \mathcal{DS}, \mathcal{IC} \rangle$ defining the intentional properties of $\mathcal{DB}$: $\mathcal{DS}$ denotes the structure of the relations, while $\mathcal{IC}$ is a set of integrity constraints expressing semantic information over data.

Definition 1. An *integrity constraint* (or *embedded dependency*) is a formula of the first order predicate calculus of the form:

$$(\forall X)\ [\ \Phi(X) \supset (\exists Z)\Gamma(W)\ ]$$

where X, W and Z are sets of variables, Φ and Γ are two conjunctions of literals such that X and W are sets of variables appearing in Φ and Γ respectively, $Z = W - X$ is the set of variables existentially quantified. □

In the definition above, Φ is called the *body* and Γ the *head* of the integrity constraint. In the rest of the paper we consider general constraints of the form[1]

$$(\forall\ X)[\ \bigwedge_{j=1}^{m} b_j(X_j),\ \varphi(X_0) \supset \bigvee_{j=m+1}^{n} (\exists Z_j) b_j(X_j, Z_j)\ ]$$

where $\varphi(X_0)$ denotes a conjunction of built-in atoms, $X = \bigcup_{j=1}^{m} X_j$, $X_i \subseteq X$ for $i \in [0\,..\,n]$. The body of a constraint ic is denoted by $Body(ic)$ whereas its head is denoted by $Head(ic)$.

The reason for considering constraints of the above form is that we want to consider range restricted constraints, i.e. constraints whose variables take values from finite domains only.

Often we shall write our constraints in a different format by moving literals from the head to the body and vice-versa. For instance, by rewriting the above constraint as denial (i.e. with empty head) we obtain:

$$(\forall\ X)[\ \bigwedge_{j=1}^{m} b_j(X_j),\ \bigwedge_{j=m+1}^{n} (\nexists Z_j) b_j(X_j, Z_j),\ \varphi(X_0) \supset\ ].$$

3.2 Repairing and Querying Inconsistent Databases

In this section the formal definition of consistent database and repair is first recalled and, then, a computational mechanism is presented that ensures that repairs and consistent answers for inconsistent databases are selected.

An update atom $\pm a(X)$ is in the form $+a(X)$ (*inserting atom*) or $-a(X)$ (*deleting atom*). Intuitively a ground atom $+a(t)$ states that $a(t)$ will be inserted into the database whereas a ground atom $-a(t)$ states that $a(t)$ will be deleted from the database. Given a set $\mathcal{U}$ of ground update atoms we define the sets $\mathcal{U}^+ = \{a(t) \mid +a(t) \in \mathcal{U}\}, \mathcal{U}^- = \{a(t) \mid -a(t) \in \mathcal{U}\}$. We say that $\mathcal{U}$ is *consistent* if it does not contain two update atoms $+a(t)$ and $-a(t)$ (i.e. if $\mathcal{U}^+ \cap \mathcal{U}^- = \emptyset$). Given a database $\mathcal{DB}$ and a consistent set of update atoms $\mathcal{U}$, we denote as $\mathcal{U}(\mathcal{DB})$ the updated database $\mathcal{DB}\ \cup \mathcal{U}^+ - \mathcal{U}^-$.

[1] The meaning of the symbols '$\wedge$' and ',' is the same.

Definition 2. Given a database $\mathcal{DB}$ and a set of integrity constraints $\mathcal{IC}$, a *repair* for $\langle \mathcal{DB}, \mathcal{IC} \rangle$ is a consistent set of update atoms $\mathcal{R}$ such that 1) $\mathcal{R}(\mathcal{DB}) \models \mathcal{IC}$ and 2) there is no consistent set of update atoms $\mathcal{U} \subset \mathcal{R}$ such that $\mathcal{U}(\mathcal{DB}) \models \mathcal{IC}$. □

For any database $\mathcal{DB}$ and set of integrity constraints $\mathcal{IC}$, the set of all possible repairs for $\langle \mathcal{DB}, \mathcal{IC} \rangle$ is denoted as $\mathbf{R}(\mathcal{DB}, \mathcal{IC})$. Thus, repaired databases are consistent databases, derived from the source database by means of a minimal set of update operations.

Observe that for constraints containing unrestricted variables the set of possible repairs could be infinite. Thus, in the rest of paper we only consider universally quantified (or full) integrity constraints of the form:

$$(\forall\ X)[\ \bigwedge_{j=1}^{m} b_j(X_j),\ \varphi(X_0) \supset \bigvee_{j=m+1}^{n} b_j(X_j)\ ]$$

where $X = \bigcup_{j=1}^{m} X_j$ and $X_i \subseteq X$ for $i \in [0\,..\,n]$.

In the following we assume that the constants appearing in constraints also appear in the database and that for a given database $\mathcal{DB}$ its domain is $\mathcal{U}_{\mathcal{DB}}$ (the Herbrand universe of $\mathcal{DB}$).

Given a set of universally quantified constraints $\mathcal{IC}$ on a database $\mathcal{DB}$ and an integrity constraint $r \in \mathcal{IC}$, a ground instantiation of r with respect to $\mathcal{DB}$ can be obtained by replacing variables with constants of $\mathcal{U}_{\mathcal{DB}}$ and eliminating the quantifiers $\forall$. The set of ground instances of r is denoted by $ground(r)$, whereas $ground(\mathcal{IC}) = \bigcup_{r \in \mathcal{IC}} ground(r)$ denotes the set of ground instances of constraints in $\mathcal{IC}$. Clearly, for any set of universally quantified constraints $\mathcal{IC}$, the cardinality of $ground(\mathcal{IC})$ is polynomial in the size of the database.

Definition 3. Given a database $\mathcal{DB}$ and a set of integrity constraints $\mathcal{IC}$, an atom A is *true* (resp. *false*) with respect to $\langle \mathcal{DB}, \mathcal{IC} \rangle$ if A belongs to all repaired databases (resp. there is no repaired database containing A). The atoms which are neither true nor false are *undefined*. □

Thus, true atoms appear in all repaired databases, whereas undefined atoms appear in a non empty proper subset of repaired databases.

Definition 4. Given a database $\mathcal{DB}$, a set $\mathcal{IC}$ of integrity constraints over $\mathcal{DB}$ and a query $Q = (g, \mathcal{P})$, the *consistent answer* of the query Q on $\langle \mathcal{DB}, \mathcal{IC} \rangle$, denoted as $Q(\mathcal{DB}, \mathcal{IC})$, gives three sets, denoted as $Q(\mathcal{DB}, \mathcal{IC})^+$, $Q(\mathcal{DB}, \mathcal{IC})^-$ and $Q(\mathcal{DB}, \mathcal{IC})^u$. These contain, respectively, the sets of g-tuples which are *true* (i.e. belonging to $\bigcap_{R \in \mathbf{R}(\mathcal{DB}, \mathcal{IC})} Q(R(\mathcal{DB}))$), *false* (i.e. not belonging to $\bigcup_{R \in \mathbf{R}(\mathcal{DB}, \mathcal{IC})} Q(R(\mathcal{DB}))$) and *undefined* (i.e. set of tuples which are neither true nor false). □

In [16] it has been shown that given a database $\mathcal{DB}$, a set of integrity constraints $\mathcal{IC}$ and a query $Q = (g, \mathcal{P})$, then 1) checking if there exists a (two-valued) repair for $\mathcal{DB}$ such that the answer of Q is not empty is in Σ_2^P and $\mathcal{NP}$-hard, 2) checking whether the consistent answer of Q is not empty is in Π_2^P and *co*$\mathcal{NP}$-hard. In

the same work it has been shown that also when considering special constraints such as functional dependencies the problem of checking, for a given ground tuple t, whether i) $t \in Q(\mathcal{DB}, \mathcal{IC})^+$ is $co\mathcal{NP}$-complete, ii) $t \in Q(\mathcal{DB}, \mathcal{IC})^-$ is $co\mathcal{NP}$-complete, iii) $t \in Q(\mathcal{DB}, \mathcal{IC})^u$ is $\mathcal{NP}$-complete. The problem becomes polynomial only when we consider functional dependencies and queries of the form $(g, \emptyset)$.

4 Deterministic Three-Valued Semantics

The problem with the approaches previously proposed is that, in the general case, computing consistent answers is too expensive and in the presence of large databases unpracticable. Thus, in this section we propose an alternative solution which allows us to compute "approximated" answers in polynomial time.

The approach we propose is based on a three-valued interpretation for integrity constraints, i.e. on the assumption that atoms belonging to a database may be either *true* or *false* or *undefined*.

We assume that a three-valued database consists of two sets: the set of *true* atoms and the set of *undefined* atoms (atoms which are not included in the database are *false*). In the following we term three-valued databases simply databases. Given a database $\mathcal{DB}$, we denote by $\mathcal{DB}^+$ and $\mathcal{DB}^u$, respectively, the set of true and undefined tuples in $\mathcal{DB}$, whereas $\mathcal{DB}^- = \mathcal{B}_{\mathcal{DB}} - \mathcal{DB}^+ - \mathcal{DB}^u$ denotes the set of false tuples. A database $\mathcal{DB}$ is said to be two-valued if $\mathcal{DB}^u = \emptyset$.

Definition 5. Given a positive Datalog program $\mathcal{P}$ and a database $\mathcal{DB}$, the evaluation of $\mathcal{P}$ on $\mathcal{DB}$, denoted $\mathcal{P}(\mathcal{DB})$, gives a minimum model consisting of two sets M^+ and M^u defined as follows:

- $M^+ = T^{\infty}_{\mathcal{P}_{\mathcal{DB}}}(\emptyset)$,
- $M^u = W^{\infty}_{\mathcal{P}_{\mathcal{DB}}}(\emptyset)$,

where $T_{\mathcal{P}_{\mathcal{DB}}}$ is the classical immediate consequence operator defined as

$$T_{\mathcal{P}_{\mathcal{DB}}}(I^+) = \{a |\ a \leftarrow b_1, \ldots, b_m \in (ground(\mathcal{P}) \cup \mathcal{DB}^+) \wedge b_1, \ldots, b_m \in I^+\}$$

whereas

$$W_{\mathcal{P}_{\mathcal{DB}}}(I^u) = \{a |\ a \leftarrow b_1, \ldots, b_m \in (ground(\mathcal{P}) \cup \mathcal{DB}^+ \cup \mathcal{DB}^u) \wedge b_1, \ldots, b_m \in (M^+ \cup I^u)\} - M^+$$

□

It is worth noting that the operator $W_{\mathcal{P}_{\mathcal{DB}}}$ is monotonic and that its fixpoint can be computed in a finite number of steps (polynomial in the size of $\mathcal{DB}$). The two sets $M^+ = T^{\infty}_{\mathcal{P}_{\mathcal{DB}}}(\emptyset)$ and $M^u = W^{\infty}_{\mathcal{P}_{\mathcal{DB}}}(\emptyset)$, containing *true* and *undefined* atoms which are derived by applying $\mathcal{P}$ to $\mathcal{DB}$, will also be denoted by $\mathcal{P}(\mathcal{DB})^+$ and $\mathcal{P}(\mathcal{DB})^u$, respectively.

For stratified $\mathcal{P}$ it is sufficient to partition it into a sequence of subprograms $[\mathcal{P}_1, \ldots, \mathcal{P}_n]$ so that negated literals appearing in a subprogram $\mathcal{P}_i$ are defined in strata below $\mathcal{P}_1, \ldots, \mathcal{P}_{i-1}$ or are database literals. The computation of the stable model is performed by evaluating one subprogram at a time and by considering the stable model of the program $\mathcal{DB} \cup \mathcal{P}_1 \cup \cdots \cup \mathcal{P}_{i-1}$ as the input database for the program $\mathcal{P}_i$. As negation is applied only to database atoms (or atoms derived

by means of strata below), for the evaluation of rules with negated literals it is sufficient to refine the definition of $T_{\mathcal{P}_{\mathcal{DB}}}$ and $W_{\mathcal{P}_{\mathcal{DB}}}$ by checking that negated body literals are *true* with respect to the current database.

Given a (stratified) Datalog query $Q = (g, \mathcal{P})$ and a database $\mathcal{DB}$, the answer of Q over $\mathcal{DB}$, denoted by $Q(\mathcal{DB})$, gives three sets denoted as $Q(\mathcal{DB})^+$, $Q(\mathcal{DB})^u$ and $Q(\mathcal{DB})^-$. These sets contain, respectively, the g-tuples which are derived as *true*, *undefined* and *false* by applying $\mathcal{P}$ over $\mathcal{DB}$, i.e. $Q(\mathcal{DB})^+ = \mathcal{P}(\mathcal{DB})^+[g]$, $Q(\mathcal{DB})^u = \mathcal{P}(\mathcal{DB})^u[g]$ and $Q(\mathcal{DB})^- = (\mathcal{B}_{\mathcal{P}_{\mathcal{DB}}} - Q(\mathcal{DB})^+ - Q(\mathcal{DB})^u)[g]$.

4.1 Deterministic Semantics

The two-valued semantics states that a constraint is satisfied if the head truth value is greater than or equal to the truth value of the body; clearly, a constraint with empty head is satisfied if the body is false, whereas a constraint with empty body is satisfied if the head is true.

Standard constraints satisfaction, stating that a database $\mathcal{DB}$ satisfies a constraint ic if $value_{\mathcal{DB}}(Head(ic)) \geq value_{\mathcal{DB}}(Body(ic))$, cannot be applied immediately to three-valued database. Indeed, the property stating that under the two-valued semantics literals appearing in constraints can be moved from the head to the body and vice versa, without modifying the truth value of constraints, does not hold under the three-valued semantics.

Example 2. Consider for instance the database $\mathcal{DB}$ consisting of $\mathcal{DB}^+ = \{\mathtt{a}\}$ and $\mathcal{DB}^u = \{\mathtt{b}, \mathtt{c}\}$. This database satisfies the constraint $ic = \mathtt{a} \wedge \mathtt{b} \supset \mathtt{c}$, but does not satisfy the constraint $ic' = \mathtt{b} \wedge \mathtt{not\ c} \supset \mathtt{not\ a}$ which is derived from ic by moving $\mathtt{a}$ to the head and $\mathtt{c}$ to the body. In fact, $\mathcal{DB} \models ic$ as $value_{\mathcal{DB}}(Body(ic)) = value_{\mathcal{DB}}(Head(ic)) = undefined$, while $\mathcal{DB} \not\models ic'$ as $value_{\mathcal{DB}}(Body(ic')) = undefined$ and $value_{\mathcal{DB}}(Head(ic')) = false$. □

Therefore, we introduce a different definition for the satisfaction of constraints under three-valued semantics which we call *deterministic semantics*. For the sake of clarity, in the following we assume that constraints are written under the form of denial (i.e. with empty head).

As for the two valued semantics, constraints whose body is *false* or whose head is *true* are satisfied. Therefore, a literal appearing in a (denial) constraint must be *false* if all other literals are *true*. For instance, a constraint of the form $l \supset$ states that the literal l must be *false*.

Given a set $\mathcal{IC}$ of ground constraints, we define the set $T_{\mathcal{IC}} = \mathcal{T}_{\mathcal{IC}}^{\infty}(\emptyset)$ where

$$\mathcal{T}_{\mathcal{IC}}(T) = \{not\ l \mid \exists ic = l \wedge l_1 \wedge \cdots \wedge l_n \wedge \varphi \supset \in \mathcal{IC}\ s.t.\ l_1, \ldots, l_n \in T\ \wedge\ \varphi \text{ is } true\}$$

Intuitively, $T_{\mathcal{IC}}$ defines the set of literals which must be *true* in order to satisfy $\mathcal{IC}$. Observe that the set $T_{\mathcal{IC}}$ is different from the fixpoint computed by the operator $T_{\mathcal{P}_{\mathcal{DB}}}$ as it contains atoms which can be either *true* or *false*. Clearly, if the derived set $T_{\mathcal{IC}}$ contains both l and *not* l, then the set of constraints $\mathcal{IC}$ cannot be satisfied, but for satisfiable sets of constraints, complementary literals cannot be derived. As the literals in $T_{\mathcal{IC}}$ must be *true*, the set $\mathcal{IC}$ of constraints can be "simplified" as follows:

1. each atom l s.t. either l or *not* l is in $T_{\mathcal{IC}}$ can be replaced with the corresponding truth value
2. built-in literals can be replaced with their truth value so that the resulting set contains only database literals,
3. each constraint in $\mathcal{IC}$ which contains the truth value *false* in the body is satisfied and can be deleted, whereas the truth value *true* can be deleted from the body conjunctions.

The derived reduced set of constraints will be denoted by $Red(\mathcal{IC})$.

Example 3. Consider the set $\mathcal{IC} = \{\ \ a \supset,\ \ not\ a \wedge not\ b \supset,\ \ b \wedge c \wedge d \supset\ \}$ of (ground) integrity constraints. The set $T_{\mathcal{IC}}$ is equal to $\{not\ a, b\}$. The reduced set of constraints contains only the constraint $c \wedge d \supset$ (the first two constraints are deleted as a and *not* b are *false*). □

Given a set $\mathcal{IC}$ of ground constraints, observe that all reduced constraints in $Red(\mathcal{IC})$ contain at least two literals and no built-in literal.

Definition 6. Let $\mathcal{IC}$ be a set of ground constraints of the form

$$l_1 \wedge \cdots \wedge l_n \wedge \varphi \supset \tag{1}$$

where $n > 1$, each l_i $(i = 1..n)$ is either a positive literal $b_i(x_i)$ or a negative literal *not* $b_i(x_i)$ and φ is a conjunction of built-in atoms. Given a constraint ic in $\mathcal{IC}$, we define

$$Ext(ic) = \{ \bigwedge_{l_j \in S} l_j, \varphi \supseteq \bigvee_{l_j \in \{l_1,\cdots,l_n\}-S} not\ l_j \mid S \subset \{l_1, ..., l_n\}\ \wedge\ |S| > 0\}$$

as the set of constraints which are derived from ic by moving database body literals to the head so that both the head and the body are not empty. Moreover, $Ext(\mathcal{IC}) = \cup_{ic \in \mathcal{IC}} Ext(ic)$. □

The following definition states when a set of constraints is satisfiable under deterministic semantics.

Definition 7. Let $\mathcal{DB}$ be a database and $\mathcal{IC}$ be a set of constraints. Then, $\mathcal{DB}$ satisfies $\mathcal{IC}$ under deterministic semantics (denoted as $\mathcal{DB} \models_{_{DS}} \mathcal{IC}$) if

- $\mathcal{DB} \models T_{ground(\mathcal{IC})}$ (i.e. $T_{ground(\mathcal{IC})} \subseteq \mathcal{DB}^+ \cup not\ \mathcal{DB}^-$), and
- $\mathcal{DB} \models\ Ext(Red(ground(\mathcal{IC})))$ □

Since under the three-valued semantics a (ground) constraint ic in $Red(ground(\mathcal{IC}))$ can change its truth value by moving literals from the body to the head and vice versa (see Example 2), in the above definition we say that a database $\mathcal{DB}$ satisfies ic under deterministic semantics if $\mathcal{DB}$ satisfies (under classical three-valued semantics) all constraints which can be derived from ic by moving any set of literals to the head (provided that both the head and the body of the derived constraints are not empty).

Example 4. Consider the following set $\mathcal{IC}$ of ground integrity constraints:

$$\begin{array}{ll} \texttt{not a} \supset & \texttt{a} \wedge \texttt{c} \wedge \texttt{not d} \supset \\ \texttt{a} \wedge \texttt{b} \supset & \texttt{b} \wedge \texttt{e} \wedge \texttt{not f} \supset \\ & \texttt{not g} \wedge \texttt{h} \supset \end{array}$$

Thus $T_{\mathcal{IC}} = \{\texttt{a}, \texttt{not b}\}$, $Red(\mathcal{IC})$ is as follows

$$\begin{array}{l} \texttt{c} \wedge \texttt{not d} \supset \\ \texttt{not g} \wedge \texttt{h} \supset \end{array}$$

whereas $Ext(Red(\mathcal{IC}))$ is as follows

$$\begin{array}{ll} \texttt{c} \supset \texttt{d} & \texttt{not g} \supset \texttt{not h} \\ \texttt{not d} \supset \texttt{not c} & \texttt{h} \supset \texttt{g} \end{array}$$

The database $\mathcal{DB}$ consisting of $\mathcal{DB}^+ = \{\texttt{a}\}$ and $\mathcal{DB}^u = \{\texttt{g}, \texttt{h}\}$ satisfies $\mathcal{IC}$ under deterministic semantics as $\mathcal{DB} \models T_{\mathcal{IC}}$ and $\mathcal{DB} \models Ext(Red(\mathcal{IC}))$. □

Proposition 1. *Let $\mathcal{DB}$ be a database and $\mathcal{IC}$ be a set of ground integrity constraints. Then, $\mathcal{DB} \models Ext(Red(\mathcal{IC}))$ iff for each constraint $ic = l_1 \wedge \cdots \wedge l_n \supset$ in $Red(\mathcal{IC})$ either (i) some literal l_i is false w.r.t. $\mathcal{DB}$ or (ii) all literals $l_1, \cdots, l_n$ are undefined w.r.t. $\mathcal{DB}$.* □

Example 5. Consider the set $\mathcal{IC} = \{\texttt{a} \wedge \texttt{b} \supset,\ \texttt{b} \wedge \texttt{not c} \supset\}$ of ground integrity constraints. Then, $Ext(\mathcal{IC}) = \{\texttt{a} \supset \texttt{not b}, \texttt{b} \supset \texttt{not a}, \texttt{b} \supset \texttt{c}, \texttt{not c} \supset \texttt{not b}\}$. These constraints are satisfied iff either i) b is *false*, or ii) a is *false* and c is *true*, or iii) all atoms a, b and c are *undefined*. □

Observe that the cardinality of $Ext(ic)$ is exponential in the number of database literals appearing in *ic*. We now show that it is sufficient to consider a number of derived constraints equal to the number of database literals appearing in *ic*.

Definition 8. Let $\mathcal{IC}$ be a set of ground constraints and *ic* be a constraint in $\mathcal{IC}$, i.e. *ic* is of the form (1) with $n > 0$. We define by

$$ext(ic) = \{l_1 \wedge \cdots \wedge l_{i\text{-}1} \wedge l_{i+1} \wedge \cdots \wedge l_n \wedge \varphi \supset not\ l_i \mid i \in [1..n]\ \}$$

the set of constraints which are derived from *ic* by moving exactly one (database) body literal to the head. Moreover, $ext(\mathcal{IC}) = \cup_{ic \in \mathcal{IC}}\ ext(ic)$. □

Proposition 2. *Given a database $\mathcal{DB}$ and a set $\mathcal{IC}$ of ground integrity constraints, then $\mathcal{DB} \models Ext(Red(\mathcal{IC}))$ iff $\mathcal{DB} \models ext(Red(\mathcal{IC}))$.* □

Therefore, in the definition of deterministic semantics (Definition 7), instead of considering the set $Ext(Red(ground(\mathcal{IC})))$, whose size is exponential in the number of database literals appearing in the body of constraints, we can consider $ext(Red(ground(\mathcal{IC})))$, whose size is linear in the number of ground constraints and the number of literals appearing in the body of constraints. Consequently, we have that $\mathcal{DB} \models_{\mathcal{DS}} \mathcal{IC}$ if i) $\mathcal{DB} \models T_{ground(\mathcal{IC})}$, and ii) $\mathcal{DB} \models ext(Red(ground(\mathcal{IC})))$.

4.2 Repairing Databases

A *substitution* S for a database $\mathcal{DB}$ is a triplet $\langle S^+, S^u, S^- \rangle$ such that

1. $S^+ \cup S^u \cup S^- \subseteq \mathcal{B}_{\mathcal{DB}}$,
2. $S^+ \cap S^u = \emptyset$, $S^+ \cap S^- = \emptyset$ and $S^- \cap S^u = \emptyset$,
3. $S^+ \subseteq \mathcal{DB}^u \cup \mathcal{DB}^-$, $S^u \subseteq \mathcal{DB}^+ \cup \mathcal{DB}^-$ and $S^- \subseteq \mathcal{DB}^+ \cup \mathcal{DB}^u$.

Given a database $\mathcal{DB}$ and a substitution S for $\mathcal{DB}$, the application of S to $\mathcal{DB}$, denoted by $S(\mathcal{DB})$, gives the following sets: i) $S(\mathcal{DB})^+ = (\mathcal{DB}^+ - S^u - S^-) \cup S^+$ and ii) $S(\mathcal{DB})^u = (\mathcal{DB}^u - S^+ - S^-) \cup S^u$. Observe that the set $S(\mathcal{DB})^- = \mathcal{B}_{\mathcal{DB}} - S(\mathcal{DB})^+ - S(\mathcal{DB})^u$ is also equal to $(\mathcal{DB}^- - S^+ - S^u) \cup S^-$.

Definition 9. Given a database $\mathcal{DB}$ and a set $\mathcal{IC}$ of constraints, a repair for $\langle \mathcal{DB}, \mathcal{IC} \rangle$ under the deterministic (three-valued) semantics is a substitution R for $\mathcal{DB}$ s.t. (i) $R(\mathcal{DB}) \models_{_{DS}} \mathcal{IC}$ and (ii) there is no substitution S s.t. $S \neq R$, $S(\mathcal{DB}) \models_{_{DS}} \mathcal{IC}$ and $S^+ \subseteq R^+$, $S^u \subseteq R^u$, $S^- \subseteq R^-$. □

Repaired databases are consistent databases derived from the source database by applying repairs over it. Given a database $\mathcal{DB}$ and a set $\mathcal{IC}$ of integrity constraints, the set of all possible repairs for $\langle \mathcal{DB}, \mathcal{IC} \rangle$, under deterministic (three-valued) semantics, is denoted as $\mathbf{R}_{\mathcal{DS}}(\mathcal{DB}, \mathcal{IC})$. Moreover, $\mathbf{DB}_{\mathbf{R}}(\mathcal{DB}, \mathcal{IC})$ denotes the set of all possible repaired databases for $\langle \mathcal{DB}, \mathcal{IC} \rangle$, i.e. $\mathbf{DB}_{\mathbf{R}}(\mathcal{DB}, \mathcal{IC}) = \{R(\mathcal{DB}) |\ R \in \mathbf{R}_{\mathcal{DS}}(\mathcal{DB}, \mathcal{IC})\}$.

For the computation of answers to queries of the form $\langle g, \emptyset \rangle$[2] we will use the standard definition stating that for a given database $\mathcal{DB}$ and a set $\mathcal{IC}$ of integrity constraints over $\mathcal{DB}$, an atom A is *true* (resp. *false*) with respect to $\langle \mathcal{DB}, \mathcal{IC} \rangle$ if A is *true* (resp. *false*) w.r.t. all repaired databases in $\mathbf{DB}_{\mathbf{R}}(\mathcal{DB}, \mathcal{IC})$. The atoms which are neither true nor false are *undefined*.

Lemma 1. *Let $\mathcal{DB}$ be a database and $\mathcal{IC}$ a set of integrity constraints over $\mathcal{DB}$. If there exist two repairs R_i and R_j for $\langle \mathcal{DB}, \mathcal{IC} \rangle$ and an atom a such that $a \in R_i^+$ and $a \notin R_j^+$ (resp. $a \in R_i^-$ and $a \notin R_j^-$), then there is a repair R_k such that $a \in R_k^u$.* □

Given two triplet $R = \langle R^+, R^u, R^- \rangle$ and $S = \langle S^+, S^u, S^- \rangle$ containing *true*, *undefined* and *false* ground atoms, we say that $R \sqsubseteq S$ if $R^+ \subseteq S^+$ and $R^- \subseteq S^-$. The relation $\sqsubseteq$ can be used to define a partial order on databases, repairs and query answers.

Theorem 1. *Given a database $\mathcal{DB}$ and a set $\mathcal{IC}$ of integrity constraints over $\mathcal{DB}$, then $\langle \mathbf{R}_{\mathcal{DS}}(\mathcal{DB}, \mathcal{IC}), \sqsubseteq \rangle$ is a lower semi-lattice whose top elements are two-valued.* □

The repair defining the bottom element of the semi-lattice $\langle \mathbf{R}_{\mathcal{DS}}(\mathcal{DB}, \mathcal{IC}), \sqsubseteq \rangle$ is called *deterministic* and is denoted as R_{det}. The database $R_{det}(\mathcal{DB})$ obtained by applying R_{det} to $\mathcal{DB}$ is called the *deterministic* repaired database.

[2] The case of general queries $\langle g, \mathcal{P} \rangle$ will be discussed in the next subsection.

Corollary 1. *Given a database $\mathcal{DB}$ and a set $\mathcal{IC}$ of integrity constraints over $\mathcal{DB}$, then $\langle \mathbf{DB_R}(\mathcal{DB}, \mathcal{IC}) \sqsubseteq \rangle$ is a lower semi-lattice whose top elements are two-valued and whose bottom element is $R_{det}(\mathcal{DB})$.* □

Theorem 2. *Given a database $\mathcal{DB}$ and a set of integrity constraints $\mathcal{IC}$ over $\mathcal{DB}$, an atom A is* true *(resp.* false, undefined*) with respect to $\langle \mathcal{DB}, \mathcal{IC} \rangle$ if A is* true *(resp.* false, undefined*) w.r.t. the deterministic repaired database.* □

4.3 Query Answers

Given a database $\mathcal{DB}$, a set $\mathcal{IC}$ of integrity constraints and a (stratified) Datalog query $Q = (g, \mathcal{P})$. The *consistent answer* of Q on $\langle \mathcal{DB}, \mathcal{IC} \rangle$, under deterministic (three-valued) semantics, denoted as $Q_{\mathcal{DS}}(\mathcal{DB}, \mathcal{IC})$, consists of the three sets:

$$Q_{\mathcal{DS}}(\mathcal{DB}, \mathcal{IC})^+ = \bigcap_{R \in \mathbf{R}_{\mathcal{DS}}(\mathcal{DB}, \mathcal{IC})} Q(R(\mathcal{DB}))^+$$

$$Q_{\mathcal{DS}}(\mathcal{DB}, \mathcal{IC})^u = \bigcup_{R \in \mathbf{R}_{\mathcal{DS}}(\mathcal{DB}, \mathcal{IC})} Q(R(\mathcal{DB}))^u$$

$$Q_{\mathcal{DS}}(\mathcal{DB}, \mathcal{IC})^- = \mathcal{B}_{\mathcal{P}_{\mathcal{DB}}} - Q_{\mathcal{DS}}(\mathcal{DB}, \mathcal{IC})^+ - Q_{\mathcal{DS}}(\mathcal{DB}, \mathcal{IC})^u$$

Theorem 3. *Let $\mathcal{DB}$ be a database, $\mathcal{IC}$ be a set of integrity constraints, $Q = (g, \mathcal{P})$ be a (stratified) Datalog query and R_{det} be the deterministic repair for $\langle \mathcal{DB}, \mathcal{IC} \rangle$. Then, $Q_{\mathcal{DS}}(\mathcal{DB}, \mathcal{IC}) = Q(R_{det}(\mathcal{DB}))$* □

Theorem 4 (Soundness). *Let $\mathcal{DB}$ be a two-valued database, $\mathcal{IC}$ be a set of integrity constraints over $\mathcal{DB}$ and $Q = (g, \mathcal{P})$ be a (stratified) Datalog query. Then, $Q_{\mathcal{DS}}(\mathcal{DB}, \mathcal{IC}) \sqsubseteq Q(\mathcal{DB}, \mathcal{IC})$.* □

Corollary 2. *Let $\mathcal{DB}$ be a database, $\mathcal{IC}$ be a set of integrity constraints and $Q = (g, \mathcal{P})$ be a (stratified) Datalog query. Then, $Q_{\mathcal{DS}}(\mathcal{DB}, \mathcal{IC})$ can be computed in polynomial time.* □

5 Rewriting into Logic Programs

Given a database $\mathcal{DB}$, a set $\mathcal{IC}$ of integrity constraints and a query $Q = \langle g, \mathcal{P} \rangle$, in this section we present how the constraints and the query can be rewritten into a Datalog program $Rew(ground(\mathcal{IC})) \cup Rew(\mathcal{P})$ so that from the stable model of $Rew(ground(\mathcal{IC})) \cup \mathcal{DB}$ (if such a model exists) it is possible to derive the deterministic repair, whereas from the stable model of $Rew(\mathcal{P}) \cup Rew(ground(\mathcal{IC})) \cup \mathcal{DB}$, by selecting the g-tuples, it is possible to compute the deterministic answer.

Given a ground literal $a(t)$ (resp. *not* $a(t)$) we denote by $Up(a(t))$ (resp. $Up(not\ a(t))$) the atom $a^+(t)$ (resp. $a^-(t)$); moreover $Und(a(t))$ and $Und(not\ a(t))$ denote the atom $a^u(t)$. It is worth noting that, when considering atoms of the form $a(t)$, $a^+(t)$, $a^-(t)$ and $a^u(t)$, the symbols a, a^+, a^- and a^u are assumed to be different predicate symbols.

Definition 10. Let $\mathcal{IC}$ be a set of ground constraints and let ic be a (ground) constraint in $ext(\mathcal{IC})$, i.e. ic is of the form:

$$\bigwedge_{j=1}^{m} b_j(x_j), \bigwedge_{j=m+1}^{n} not\ b_j(x_j), \varphi(x_0) \supset l(x_l)$$

where $l(x_l)$ is a literal. We define $rew_1(ic)$ as the following Datalog rule:

$$Up(l(x_l)) \leftarrow \bigwedge_{j=1}^{m} Up(b_j(x_j)), \bigwedge_{j=m+1}^{n} Up(not\ b_j(x_j)), \varphi(x_0)$$

whereas $Rew_1(\mathcal{IC})$ is equal to

$$\{\ rew_1(ic) \mid ic \in ext(\mathcal{IC})\ \} \cup \{\leftarrow Up(L), Up(not\ L) \mid L \text{ is a literal appearing in } \mathcal{IC}\}$$

Moreover $rew_2(ic)$ denotes the following set of Datalog rules

$$Und(l(x_l)) \leftarrow \bigwedge_{j=1}^{m} b'_j(x_j), \bigwedge_{j=m+1}^{n} b''_j(x_j), not\ l(x_l), not\ Up(l(x_l)), not\ Up(not\ l(x_l)), \varphi(x_0)$$

$$b'_j(x_j) \leftarrow (b_j(x_j) \wedge not\ b_j^-(x_j)) \vee b_j^+(x_j) \vee b_j^u(x_j) \qquad j = 1..m$$

$$b''_j(x_j) \leftarrow (not\ b_j(x_j) \wedge not\ b_j^+(x_j)) \vee b_j^-(x_j) \vee b_j^u(x_j) \qquad j = m+1..n$$

Finally, we define the set $Rew_2(\mathcal{IC}) = \bigcup_{ic \in ext(\mathcal{IC})} rew_2(ic)$ and $Rew(\mathcal{IC}) = Rew_1(\mathcal{IC}) \cup Rew_2(\mathcal{IC})$. □

It is worth noting that in the above definition $Rew_1(\mathcal{IC})$ is a positive Datalog program which computes $T_{\mathcal{IC}}$ (namely $T_{\mathcal{IC}} = \{l(x_l) |\ Up(l(x_l)) \in \mathcal{MM}(Rew_1(\mathcal{IC}))\}$).

Example 6. Consider the following set $\mathcal{IC}$ of (ground) integrity constraints

$$\begin{array}{l} \mathtt{a} \supset \\ \mathtt{not\ a} \wedge \mathtt{b} \supset \\ \mathtt{not\ b} \wedge \mathtt{c} \wedge \mathtt{not\ d} \supset \end{array}$$

The set of constraints $ext(\mathcal{IC})$ is as follows

$$\begin{array}{ll} \supset \mathtt{not\ a} & \mathtt{not\ b} \wedge \mathtt{c} \supset \mathtt{d} \\ \mathtt{not\ a} \supset \mathtt{not\ b} & \mathtt{c} \wedge \mathtt{not\ d} \supset \mathtt{b} \\ \mathtt{b} \supset \mathtt{a} & \mathtt{not\ b} \wedge \mathtt{not\ d} \supset \mathtt{not\ c} \end{array}$$

The program $Rew_1(\mathcal{IC})$ is

$$\begin{array}{ll} \mathtt{a}^- \leftarrow & \leftarrow \mathtt{a}^+, \mathtt{a}^- \\ \mathtt{b}^- \leftarrow \mathtt{a}^- & \leftarrow \mathtt{b}^+, \mathtt{b}^- \\ \mathtt{a}^+ \leftarrow \mathtt{b}^+ & \leftarrow \mathtt{c}^+, \mathtt{c}^- \\ \mathtt{d}^+ \leftarrow \mathtt{b}^-, \mathtt{c}^+ & \leftarrow \mathtt{d}^+, \mathtt{d}^- \\ \mathtt{b}^+ \leftarrow \mathtt{c}^+, \mathtt{d}^- & \leftarrow \mathtt{e}^+, \mathtt{e}^- \\ \mathtt{c}^- \leftarrow \mathtt{b}^-, \mathtt{d}^- & \end{array}$$

whereas $Rew_2(\mathcal{IC})$ is

$$
\begin{array}{llll}
\mathtt{a^u} \leftarrow & \mathtt{a}, & \mathtt{not\ a^+}, \mathtt{not\ a^-} & \mathtt{a''} \leftarrow (\mathtt{not\ a} \wedge \mathtt{not\ a^+}) \vee \mathtt{a^-} \vee \mathtt{a^u} \\
\mathtt{b^u} \leftarrow \mathtt{a''}, & \mathtt{b}, & \mathtt{not\ b^+}, \mathtt{not\ b^-} & \mathtt{b'} \leftarrow (\mathtt{b} \wedge \mathtt{not\ b^-}) \vee \mathtt{b^+} \vee \mathtt{b^u} \\
\mathtt{a^u} \leftarrow \mathtt{b'}, & \mathtt{not\ a}, & \mathtt{not\ a^+}, \mathtt{not\ a^-} & \mathtt{b''} \leftarrow (\mathtt{not\ b} \wedge \mathtt{not\ b^+}) \vee \mathtt{b^-} \vee \mathtt{b^u} \\
\mathtt{d^u} \leftarrow \mathtt{b''}, \ \mathtt{c'}, & \mathtt{not\ d}, & \mathtt{not\ d^+}, \mathtt{not\ d^-} & \mathtt{c'} \leftarrow (\mathtt{c} \wedge \mathtt{not\ c^-}) \vee \mathtt{c^+} \vee \mathtt{c^u} \\
\mathtt{b^u} \leftarrow \mathtt{c'}, \ \mathtt{d''}, & \mathtt{not\ b}, & \mathtt{not\ b^+}, \mathtt{not\ b^-} & \mathtt{d''} \leftarrow (\mathtt{not\ d} \wedge \mathtt{not\ d^+}) \vee \mathtt{d^-} \vee \mathtt{d^u} \\
\mathtt{c^u} \leftarrow \mathtt{b''}, \ \mathtt{d''}, & \mathtt{c}, & \mathtt{not\ c^+}, \mathtt{not\ c^-} &
\end{array}
$$

Let $\mathcal{DB}$ be a two-valued database, $\mathcal{IC}$ be a set of integrity constraints on $\mathcal{DB}$ and M the unique stable model (if it exists) for $Rew(ground(\mathcal{IC})) \cup \mathcal{DB}$. Then, $S(M) = \langle \{a(t) | a^+(t) \in M\}, \{a(t) | a^u(t) \in M\} \{a(t) | a^-(t) \in M\} \rangle$ denotes the substitution derived from M.

Theorem 5. *Let $\mathcal{DB}$ be a two-valued database, $\mathcal{IC}$ be a set of integrity constraints on $\mathcal{DB}$ and $M = \mathcal{SM}(Rew(ground(\mathcal{IC})) \cup \mathcal{DB})$. Then,*

- *if M does not exist, then $ground(\mathcal{IC})$ is not satisfiable,*
- *else $S(M)$ is the deterministic repair for $\langle \mathcal{DB}, \mathcal{IC} \rangle$.* □

Example 7. Consider the database $\mathcal{DB}$ which consists of $\mathcal{DB}^+ = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ and the set $\mathcal{IC}$ of integrity constraints of Example 6. Then, $S(\mathcal{SM}(Rew(\mathcal{IC}) \cup \mathcal{DB})) = \langle \{\}, \{\mathtt{c}, \mathtt{d}\}, \{\mathtt{a}, \mathtt{b}\} \rangle$ which is the deterministic repair for $\langle \mathcal{DB}, \mathcal{IC} \rangle$. □

Theorem 6. *Let $\mathcal{DB}$ be a two-valued database and $\mathcal{IC}$ be a set of integrity constraints on $\mathcal{DB}$. The deterministic repair for $\langle \mathcal{DB}, \mathcal{IC} \rangle$ can be computed in polynomial time in the size of $\mathcal{DB}$.* □

Let us now present how queries have to be rewritten to compute (three-valued) answers.

Definition 11. Let $\mathcal{P}$ be a (stratified) Datalog program and r be a rule in $\mathcal{P}$, i.e. r is of the form

$$A \leftarrow \bigwedge_{i=1}^{h} B_i, \bigwedge_{j=h+1}^{m} not\ B_j, \bigwedge_{k=m+1}^{n} L_k, \Phi$$

where A is an atom of the form $p(X)$, B_i $(i = 1..m)$ is an atom of the form $p_i(X_i)$ where p_i is a base predicate symbol, L_k $(k = m+1..n)$ is either a positive literal $p_k(X_k)$ or a negative literal $not\ p_k(X_k)$ such that p_k is a derived predicate symbol and Φ is a conjunction of built-in atoms. We denote by $rew^T(r)$ the following set of rules:

$$A \leftarrow \bigwedge_{i=1}^{h} B_i^T, \bigwedge_{j=h+1}^{m} B_j^F, \bigwedge_{k=m+1}^{n} L_k, \Phi$$

$$
\begin{array}{ll}
A^T \leftarrow A & \\
B_i^T \leftarrow (B_i \wedge not\ B_i^- \wedge not\ B_i^u) \vee B_i^+ & i = 1..h \\
B_j^F \leftarrow (not\ B_j \wedge not\ B_j^+ \wedge not\ B_j^u) \vee B_j^- & j = h+1..m
\end{array}
$$

where B_i (resp. B_i^+, B_i^-, B_i^u, B_i^T, B_j^F), for $i = 1..m$, is an atom of the form $p_i(X_i)$ (resp. $p_i^+(X_i)$, $p_i^-(X_i)$, $p_i^u(X_i)$, $p_i^T(X_i)$, $p_i^F(X_i)$). $Rew^T(\mathcal{P}) = \bigcup_{r\in\mathcal{P}} rew^T(r)$. Moreover, we denote by $rew^U(r)$ the following set of rules:

$$A^{und} \leftarrow B_l^u, \bigwedge_{i=1\wedge i\neq l}^{h}(B_i^T \vee B_i^u), \bigwedge_{j=h+1}^{m}(B_j^F \vee B_j^u), \bigwedge_{k=m+1}^{n}(L_k \vee L_k^{und}), \Phi \quad l = 1..h$$

$$A^{und} \leftarrow B_l^u, \bigwedge_{i=1}^{h}(B_i^T \vee B_i^u), \bigwedge_{j=h+1\wedge j\neq l}^{m}(B_j^F \vee B_j^u), \bigwedge_{k=m+1}^{n}(L_k \vee L_k^{und}), \Phi \quad l = h+1..m$$

$$A^{und} \leftarrow L_l^{und}, \bigwedge_{i=1}^{h}(B_i^T \vee B_i^u), \bigwedge_{j=h+1}^{m}(B_j^F \vee B_j^u), \bigwedge_{k=m+1\wedge k\neq l}^{n}(L_k \vee L_k^{und}), \Phi \quad l = m+1..n$$

$$A^U \leftarrow A^{und}, not\ A^T$$

Observe that in the above rules, L_k is a positive literal A_k or a negative literal *not* A_k and in both cases $L_k^{und} = A_k^{und}$. Moreover, $Rew^U(\mathcal{P}) = \bigcup_{r\in\mathcal{P}} rew^U(r)$ and $Rew(\mathcal{P}) = Rew^T(\mathcal{P}) \cup Rew^U(\mathcal{P}) \cup \{B^U \leftarrow B^u |\ B$ is a database atom $\}$. □

Let $\mathcal{DB}$ be a two-valued database, $\mathcal{IC}$ be a set of integrity constraints on $\mathcal{DB}$, $\mathcal{P}$ be a (stratified) Datalog program and M the unique stable model (if it exists) for $Rew(\mathcal{P}) \cup Rew(ground(\mathcal{IC})) \cup \mathcal{DB}$. We define the sets $T(M) = \{p(t) |\ p^T(t) \in M\}$ and $U(M) = \{p(t) |\ p^U(t) \in M\}$, respectively, the set of true and undefined atoms in M. $F(M) = \mathcal{B}_{\mathcal{P}_{\mathcal{DB}}} - T(M) - T(F)$ denotes the set of atoms which are false in M.

Theorem 7. *Let $Q = (g, \mathcal{P})$ be a (stratified) Datalog query, $\mathcal{DB}$ be a two-valued database, $\mathcal{IC}$ be a set of integrity constraints over $\mathcal{DB}$ and $M = \mathcal{SM}(Rew(\mathcal{P}) \cup Rew(ground(\mathcal{IC})) \cup \mathcal{DB})$. Then,*

- *if M does not exist, then $ground(\mathcal{IC})$ is not satisfiable,*
- *else $Q_{\mathcal{DS}}(\mathcal{DB}, \mathcal{IC}) = \langle T(M)[g], U(M)[g], F(M)[g]\rangle$.* □

Clearly, as we have already stressed, only the sets of true and undefined atoms are effectively computed.

References

1. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley, Reading (1994)
2. Andritsos, P., Fuxman, A., Miller, R.: Clean Answers over Dirty Databases. ICDE (2006)
3. Arenas, M., Bertossi, L., Chomicki, J.: Consistent query answers in inconsistent databases. In: Proc. PODS, pp. 68–79 (1999)
4. Arenas, M., Bertossi, L., Chomicki, J.: Answer Sets for Consistent Query Answering in Inconsistent Databases. TPLP 3(45), 393–424 (2003)

5. Bertossi, L.: Consistent Query Answering in Databases. SIGMOD Rec., 35(2) (2006)
6. Cali, A., Lembo, D., Rosati, R.: On the Decidability and Complexity of Query Answering over Inconsistent and Incomplete Databases. In: PODS, pp. 260–271 (2003)
7. Caroprese, L., Greco, S., Sirangelo, S., Zumpano, E.: Declarative Semantics of Production Rules for Integrity Maintenance. In: Etalle, S., Truszczyński, M. (eds.) ICLP 2006. LNCS, vol. 4079, pp. 26–40. Springer, Heidelberg (2006)
8. Chomicki, J., Lobo, J., Naqvi, S.A.: Conflict resolution using logic programming. IEEE TKDE 15(1), 244–249 (2003)
9. Chomicki, J., Marcinkowski, J.: Minimal-change integrity maintenance using tuple deletions. Information & Computation 197(1-2), 90–121 (2005)
10. Chomicki, J.: Consistent Query Answering: Five Easy Pieces. In: Schwentick, T., Suciu, D. (eds.) ICDT 2007. LNCS, vol. 4353, pp. 1–17. Springer, Heidelberg (2006)
11. Corts-Calabuig, A., Denecker, M., Arieli, O., Bruynooghe, M.: Representation of Partial Knowledge and Query Answering in Locally Complete Databases. In: Hermann, M., Voronkov, A. (eds.) LPAR 2006. LNCS (LNAI), vol. 4246, Springer, Heidelberg (2006)
12. Flesca, S., Furfaro, F., Parisi, F.: Consistent Query Answers on Numerical Databases under Aggregate Constraints. In: Bierman, G., Koch, C. (eds.) DBPL 2005. LNCS, vol. 3774, pp. 279–294. Springer, Heidelberg (2005)
13. Fuxman, A., Miller, R.J.: First-Order Query Rewriting for Inconsistent Databases. In: Eiter, T., Libkin, L. (eds.) ICDT 2005. LNCS, vol. 3363, pp. 337–351. Springer, Heidelberg (2004)
14. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: Proc. ICLP, pp. 1070–1080 (1988)
15. Greco, G., Zumpano, E.: Querying Inconsistent Databases. In: Parigot, M., Voronkov, A. (eds.) LPAR 2000. LNCS (LNAI), vol. 1955, pp. 308–325. Springer, Heidelberg (2000)
16. Greco, G., Greco, S., Zumpano, E.: A Logical Framework for Querying and Repairing Inconsistent Databases. IEEE TKDE 15(6), 1389–1408 (2003)
17. Greco, S., Sirangelo, C., Trubitsyna, I., Zumpano, E.: Preferred Repairs for Inconsistent Databases. In: Galindo, F., Takizawa, M., Traunmüller, R. (eds.) DEXA 2004. LNCS, vol. 3180, pp. 44–55. Springer, Heidelberg (2004)
18. Leone, N., Pfeifer, G., Faber, W., Calimeri, F., Dell'Armi, T., Eiter, T., Gottlob, G., Ianni, G., Ielpa, G., Koch, K., Perri, S., Polleres, A.: The dlv System. Jelia (2002)
19. Rao, P., Sagonas, K.F., Swift, T., Warren, D.S., Freire, J.: XSB: A System for Effciently Computing WFS. In: Fuhrbach, U., Dix, J., Nerode, A. (eds.) LPNMR 1997. LNCS, vol. 1265, pp. 431–441. Springer, Heidelberg (1997)
20. Sagonas, K.F., Swift, T., Warren, D.S.: An abstract machine for efficiently computing queries to well-founded models. J. of Logic Programm. 45(1-3), 1–41 (2000)
21. Syrjänen, T., Niemelä, I.: The Smodels System. In: Eiter, T., Faber, W., Truszczyński, M. (eds.) LPNMR 2001. LNCS (LNAI), vol. 2173, pp. 434–438. Springer, Heidelberg (2001)
22. Subrahmanian, V.S.: Amalgamating knowledge bases. ACM TKDE 19(2), 291–331 (1994)
23. Wijsen, J.: Database repairing using updates. ACM TODS 30(3), 722–768 (2005)

Logic Programming Approach to Automata-Based Decision Procedures

Gulay Unel and David Toman

D.R. Cheriton School of Computer Science, University of Waterloo
{gunel,david}@cs.uwaterloo.ca

Abstract. We propose a novel technique that maps decision problems in WS1S (weak monadic second-order logic with n successors) to the problem of query evaluation of Complex-value Datalog queries. We then show how the use of advanced implementation techniques for Logic Programs, in particular the use of tabling in the XSB system, yields a considerable improvement in performance over more traditional approaches. We also explore various optimizations of the proposed technique based on variants of tabling and goal reordering. Although our primary focus is on WS1S, it is straightforward to adapt our approach for other logics with existing automata-theoretic decision procedures, for example WS2S.

1 Introduction

Monadic second-order logics provide means to specify regular properties of systems in a succinct way. In addition, these logics are decidable by the virtue of the connection to automata theory. However, only recently tools based on these ideas—in particular the MONA system [17]—have been developed and shown to be efficient enough for practical applications [15].

However, for reasoning in large theories consisting of relatively simple constraints, such as theories capturing UML class diagrams or database schemata, the MONA system runs into a serious state-space explosion problem—the size of the automaton capturing the (language of) models for a given formula quickly exceeds the space available in most computers. Surprisingly, the problem can be traced to the *automata product* operation that is used to translate conjunction in the original formulæ rather than to the projection/determinization operations needed to handle quantifier alternations.

This paper introduces a technique that combats this problem. However, unlike most other approaches that usually attempt to use various compact representation techniques for automata, e.g., based on BDDs [5] or on state space factoring using a *guided* automaton [17], our approach is based on techniques developed for program evaluation in deductive databases, in particular on the *Magic Set transformation* [2] and *SLG resolution*, a top-down resolution-based approach augmented with memoing [9,10]. We also study the impact of using other optimization techniques developed for Logic Programs, such as goal reordering.

The main contribution of the paper is establishing the connection between the automata-based decision procedures for WS1S (and, analogously, for WS2S)

V. Dahl and I. Niemelä (Eds.): ICLP 2007, LNCS 4670, pp. 165–179, 2007.

and query evaluation in Complex-value Datalog (Datalog^{cv}). Indeed, the complexity of query evaluation in Datalog^{cv} matches the complexity of the WS1S decision procedure and thus it seems like an appropriate tool for this task. Our approach is based on representing automata using nested relations and on defining the necessary automata-theoretic operations using Datalog^{cv} programs. This reduces to posing a closed Datalog^{cv} goal over a Datalog^{cv} program representing implicitly the final automaton. This observation combined with powerful program execution techniques developed for deductive databases, such as the Magic Set rewriting and SLG resolution, limit the explored state space to elements needed to show non-emptiness of the automaton and, in turn, satisfiability of the corresponding formula.

In addition to showing the connection between the automata-based decision procedures and query evaluation in Datalog^{cv}, we have also conducted experiments with the XSB [27] system that demonstrate the benefits of the proposed method over more standard approaches.

The remainder of the paper is organized as follows. In Section 2 we formally introduce monadic second-order logic and the connection to finite automata. We also define Datalog^{cv} programs, state their computational properties, and briefly discuss techniques used for program evaluation. Section 3 shows how Datalog^{cv} programs can be used to implicitly represent a finite automaton and to implement automata-theoretic operations on such a representation. In this paper we only present results for the WS1S logic. However, the results extend immediately to WS2S. Experimental results for the proposed methods are presented in Section 4. Related work is discussed in Section 5. Finally, conclusions and future research directions are given in Section 6.

2 Background and Definitions

In this section we provide definitions needed for the technical development in the rest of the paper.

2.1 Weak Second-Order Logic (WS1S)

First, we define the syntax and semantics of the weak second-order logic of one successor and comment on techniques used to show its decidability.

Definition 1. *The formulas of second-order logics are defined as follows.*

- *the expressions $s(x, y)$, $x \subseteq y$ for x, y second-order variables are atomic formulas, and*
- *given formulas φ and ϕ and a variable x, the expressions $\varphi \wedge \phi$, $\neg\varphi$, and $\exists x : \varphi$ are also formulas.*

Additional common syntactic features can be defined by the following. Variables for individuals (first-order variables) can be simulated using second-order variables bound to singleton sets; a property expressible in WS1S. Thus we allow

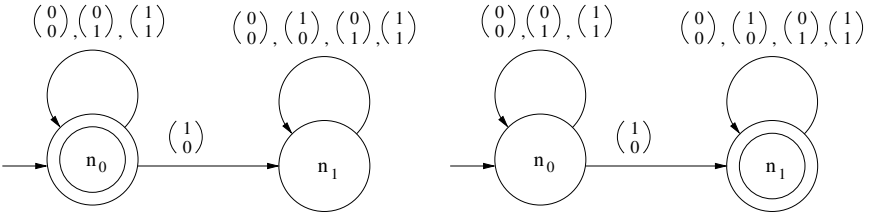


Fig. 1. Automata representing the formulae $x \subseteq y$ and $\neg(x \subseteq y)$

$x \in y$ for $x \subseteq y$ whenever we know that x is a singleton. We also use the standard abbreviations $\varphi \vee \psi$ for $\neg(\neg\varphi \wedge \neg\psi)$, $\varphi \rightarrow \psi$ for $\neg\varphi \vee \psi$, and $\forall x : \varphi$ for $\neg\exists x : \neg\varphi$.

The semantics of WS1S is defined w.r.t. the set of natural numbers (successors of 0); second-order variables are interpreted as finite sets of natural numbers. The interpretation of the atomic formula $s(x, y)$ is fixed to relating singleton sets $\{n + 1\}$ and $\{n\}$, $n \in \mathbf{N}$.[1] Truth and satisfiability of formulas is defined with the help of valuations mapping variables to finite sets of natural numbers in a standard way.

Connection to Finite Automata. The crux of the connection lies in an observation that, for every WS1S formula, there is an automaton that accepts exactly the (string representations of) models of a given formula [28]. Since each variable of WS1S is interpreted by a finite set of natural numbers, such an interpretation can be captured by a finite string. Satisfying interpretations of formulas (with k free variables) can be represented as sets of strings over $\{0, 1\}^k$. The i-th component corresponds to the interpretation of the i-th variable and is called a *track*. It turns out that sets of the above strings form regular languages and thus can be recognized using an automaton. Satisfiability then reduces to checking for non-emptiness of the language accepted by such an automaton.

Definition 2. *A finite automaton is a 5-tuple $A = (N, X, S, T, F)$, where N is the set of states (nodes), X is the alphabet, S is the initial (starting) state, $T \subseteq N \times N \times X$ is the transition relation, and F is the set of final states.*

Given a WS1S formula φ, the automaton A_φ can be effectively constructed starting from automata for atomic formulæ using automata-theoretic operations.

Proposition 1. *Let φ be a WS1S formula. Then there is an automaton A_φ such that φ is satisfiable if and only if $L(A_\varphi) \neq \emptyset$, where $L(A)$ is the language accepted by A.*

Example 1. The automaton A_φ for the formula $\varphi = x \subseteq y$ is shown in the left part of Figure 1, the complement automaton $A_{\neg\varphi}$ that represents $\neg\varphi = \neg(x \subseteq y)$ is shown in the right part of Figure 1. The labels on the edges are elements of

[1] The atomic formula $s(x, y)$ is often written as $x = s(y)$ in literature, emphasizing its nature as a *successor* function.

the alphabet of the automaton and capture the valuations of variables allowed for a particular transition. The tracks in the strings accepted by the automata represent the valuation for the variables x (first track), and y (second track).

Similarly, the automaton $A_{\varphi\wedge\phi}$ is the product automaton of A_φ and A_ϕ and accepts $L(A_\varphi) \cap L(A_\phi)$, the satisfying interpretations of $\varphi \wedge \phi$. The automaton $A_{\exists x:\varphi}$, the projection automaton of A_φ, accepts satisfying interpretations of $\exists x : \varphi$. Intuitively, the automaton $A_{\exists x:\varphi}$ acts as the automaton A_φ for φ except that it is allowed to guess the bits on the track of the variable x. We give the actual algorithms for the inductive construction of A_φ in the following section (in a Datalog$^{\text{cv}}$ syntax). While checking for emptiness can be done in time polynomial in the size of an automaton, the size of A_φ is non-elementary in the size of φ (more precisely, in the depth of quantifier alternation of φ). This bound is tight for WS1S decision problem yielding an overall non-elementary decision procedure.

2.2 Datalog for Complex Values

In the remainder of this section we define a query language that serves as the target of our approach to WS1S decision procedure.

Complex Data Model. The complex-value data model is an extension of the standard relational model that allows tuples and finite sets to serve as values in the place of atomic values [1]. Each value is assigned a finite *type* generated by the type grammar "$\tau := \iota \mid [\tau_1, \ldots, \tau_k] \mid \{\tau\}$", where ι stands for the type of uninterpreted atomic constants, $[\tau_1, \ldots, \tau_k]$ for a k-tuple consisting of values belonging to the types $\tau_1, \ldots, \tau_k$, respectively, and $\{\tau\}$ for a finite set of values of type τ. Relations are interpreted as sets of values of a given type[2]. The model is equipped with several *built-in* relations, e.g., the equality $=$ (extended to all types), the subset relation $\subseteq$ (defined for set types), the tuple constructor (that relates tuples of values to the individual values), the singleton set constructor (relating values of a type to singleton sets of the appropriate set type), etc.

Complex-value Queries. The extended data model induces extensions to relational query languages and leads to the definition of *complex-value relational calculus* (calc$^{\text{cv}}$) and a deductive language for complex values, Datalog$^{\text{cv}}$—the language of Horn clauses built from literals whose arguments range over complex-valued variables and constants [2,26]. Datalog$^{\text{cv}}$ programs and queries are defined as follows:

Definition 3. *A* Datalog$^{\text{cv}}$ atom *is a predicate symbol with variables or complex-value constants as arguments.*

A Datalog$^{\text{cv}}$ database (program) *is a finite collection of Horn clauses of the form* $h \leftarrow g_1, \ldots, g_k$, *where* h *(called head) is an atom with an optional* grouping specification *and* $g_1, \ldots, g_k$ *(called goals) are literals (atoms or their negations).*

[2] We use relations of arity higher than one as a shorthand for sets—unary relations—of tuples of the same arity.

The grouping is syntactically indicated by enclosing the grouped *argument in the* $\langle\cdot\rangle$ *constructor; the values then range over the set type of the original argument.*

We require that in every occurrence of an atom the corresponding arguments have the same finite type and that the clauses are stratified with respect to negation.

A Datalog$^{\text{cv}}$ *query is a clause of the form* $\leftarrow g_1, \ldots, g_k$.

Evaluation of a Datalog$^{\text{cv}}$ *query (with respect to a* Datalog$^{\text{cv}}$ *database* P*) determines whether* $P \models g_1, \ldots, g_k$.

Datalog$^{\text{cv}}$ is equivalent to the complex-value calculus in expressive power [1]. However, the ability to express transitive closure without resorting to the powerset construction aids our goal of using Datalog$^{\text{cv}}$ to represent finite automata and to test for emptiness.

Proposition 2. *The complexity of* Datalog$^{\text{cv}}$ *query evaluation is non-elementary.*

Note that the complexity matches that of the decision procedures for WS1S and thus mapping of WS1S formulas to Datalog$^{\text{cv}}$ queries can be done efficiently.

To simplify the notation in the following we allow terms constructed of constants, variables, and finite number of applications of tuple and set constructors to appear as arguments of atoms. For example $p(\{x\}, y) \leftarrow q([x, y])$ is a shorthand for $p(z, y) \leftarrow q(w), w = [x, y], z = \{x\}$, where $w = [x, y]$ is an instance of a tuple constructor and $z = \{x\}$ of a set constructor built-in relations as discussed in our overview of the complex-value data model.

Evaluation of Datalog$^{\text{cv}}$ **Programs.** The basic technique for evaluation of Datalog$^{\text{cv}}$ programs is commonly based on a fixed-point construction of the minimal Herbrand model (for Datalog$^{\text{cv}}$ programs with *stratified negation* the model is constructed w.r.t. the stratification) and then testing whether a ground (instance of the) query is contained in the model. The type restrictions guarantee that the fixpoint iteration terminates after finitely many steps. While the naive fixed-point computation can be implemented directly, efficient query evaluation engines use more involved techniques such as the semi-naive evaluation, goal/join ordering, etc. In addition, whenever the query is known as part of the input, techniques that allow constructing only the relevant parts of the minimal Herbrand model have been developed. Among these the most prominent are the magic set rewriting (followed by subsequent fixed-point evaluation) and the top-down resolution with memoing—the SLG resolution.

Magic Sets. The main idea behind this approach is to restrict the values derived by a fixpoint computation to those that can potentially aid answering a given query. This is achieved by program transformation based on adding *magic predicates* to clauses that limit the breadth of the fixpoint computation at each step. These predicates are *seeded* by the values in the query (as those are the only ones the user desires to derive); more values are added to the interpretations of the magic predicates by means of additional clauses that *relax* the limit depending on what additional subqueries for a particular predicate need to be asked

to answer the original query. This process then becomes a part of the fixpoint evaluation itself.

SLG Resolution. In contrast, the SLG resolution is based on the more common SLD resolution used in PROLOG systems. The difference is in memoing what subgoals have been already resolved against and what answer substitutions were obtained, if any. Thus, whenever a more specific goal is to be selected, the memoing information is inspected to prevent repetitive computation. This mechanism also prevents infinite resolution paths in evaluation of Datalog$^{\text{cv}}$ queries. There are efficient scheduling strategies implemented for tabled logic programs:

- Batched Scheduling: provides space and time reduction over the naive strategy which is called single stack scheduling.
- Local Scheduling: provides speedups for programs that require answer subsumption.

Surprisingly, it can be shown that both of the techniques essentially simulate each other and that the magic predicates match the memoing data structures (modulo open terms). For detailed description of the above techniques see [3,20] and [27], respectively.

3 Automata and Datalog for Complex Values

In this section we present the main contribution of our approach: Given a WS1S formula φ we create a Datalog$^{\text{cv}}$ program P_φ such that an answer to a reachability/transitive closure goal w.r.t. this program proves satisfiability of φ.

However, we do not attempt to map the formula φ itself to Datalog$^{\text{cv}}$. Rather, we represent the construction of A_φ—the finite automaton that captures models of φ—as a Datalog$^{\text{cv}}$ program P_φ. This enables the use of the efficient evaluation techniques discussed in Section 2.2.

3.1 Representation of Automata

First, we fix the representation for automata that capture models of WS1S formulæ. Given a WS1S formula φ with free variables $x_1, \ldots, x_k$ we define a Datalog$^{\text{cv}}$ program P_φ that defines the following predicates:

1. $\mathsf{Node}_\varphi(n)$ representing the nodes of A_φ,
2. $\mathsf{Start}_\varphi(n)$ representing the starting state,
3. $\mathsf{Final}_\varphi(n)$ representing the set of final states, and
4. $\mathsf{Trans}_\varphi(nf_1, nt_1, \overline{x})$ representing the transition relation.

where $\overline{x} = \{x_1, x_2, \ldots, x_k\} \in X_\varphi$ is the set of free variables of φ; concatenation of their binary valuations represents a letter of A_φ's alphabet.

Example 2. The following program P_φ represents the automaton A_φ shown in the left part of Figure 1:

$$\begin{array}{lll}
\mathsf{Node}_\varphi(n_0) \leftarrow & \mathsf{Trans}_\varphi(n_0, n_0, 0, 0) \leftarrow & \mathsf{Trans}_\varphi(n_1, n_1, 0, 0) \leftarrow \\
\mathsf{Node}_\varphi(n_1) \leftarrow & \mathsf{Trans}_\varphi(n_0, n_0, 0, 1) \leftarrow & \mathsf{Trans}_\varphi(n_1, n_1, 1, 0) \leftarrow \\
\mathsf{Start}_\varphi(n_0) \leftarrow & \mathsf{Trans}_\varphi(n_0, n_0, 1, 1) \leftarrow & \mathsf{Trans}_\varphi(n_1, n_1, 0, 1) \leftarrow \\
\mathsf{Final}_\varphi(n_0) \leftarrow & \mathsf{Trans}_\varphi(n_0, n_1, 1, 0) \leftarrow & \mathsf{Trans}_\varphi(n_1, n_1, 1, 1) \leftarrow
\end{array}$$

Note that while for atomic formulas, the values representing nodes are atomic, for automata corresponding to complex formulæ these values become complex.

3.2 Automata-Theoretic Operations

We define the appropriate automata-theoretic operations: negation, conjunction, projection, and determinization used in decision procedures for the logics under consideration as programs in Datalog$^{\mathrm{cv}}$ as follows.

Definition 4. *The program $P_{\neg\alpha}$ consists of the following clauses added to the program P_α:*

1. $\mathsf{Node}_{\neg\alpha}(n) \leftarrow \mathsf{Node}_\alpha(n)$
2. $\mathsf{Start}_{\neg\alpha}(n) \leftarrow \mathsf{Start}_\alpha(n)$
3. $\mathsf{Final}_{\neg\alpha}(n) \leftarrow \mathsf{Node}_\alpha(n), \neg\mathsf{Final}_\alpha(n)$
4. $\mathsf{Trans}_{\neg\alpha}(nf_1, nt_1, \overline{x}) \leftarrow \mathsf{Trans}_\alpha(nf_1, nt_1, \overline{x})$

The following lemma is immediate:

Lemma 1. *If P_α represents A_α then $P_{\neg\alpha}$ represents $A_{\neg\alpha}$.*

The conjunction automaton which represents the conjunction of the two formulæ that original automata represent is defined as follows.

Definition 5. *The program $P_{\alpha_1\wedge\alpha_2}$ consists of the union of programs P_{α_1} and P_{α_2} and the following clauses*

1. $\mathsf{Node}_{\alpha_1\wedge\alpha_2}([n_1, n_2]) \leftarrow \mathsf{Node}_{\alpha_1}(n_1), \mathsf{Node}_{\alpha_2}(n_2)$
2. $\mathsf{Start}_{\alpha_1\wedge\alpha_2}([n_1, n_2]) \leftarrow \mathsf{Start}_{\alpha_1}(n_1), \mathsf{Start}_{\alpha_2}(n_2)$
3. $\mathsf{Final}_{\alpha_1\wedge\alpha_2}([n_1, n_2]) \leftarrow \mathsf{Final}_{\alpha_1}(n_1), \mathsf{Final}_{\alpha_2}(n_2)$
4. $\mathsf{Trans}_{\alpha_1\wedge\alpha_2}([nf_1, nf_2], [nt_1, nt_2], \overline{x}, \overline{y}, \overline{z}) \leftarrow$
   $\mathsf{Trans}_{\alpha_1}(nf_1, nt_1, \overline{x}, \overline{y}),\ \mathsf{Trans}_{\alpha_2}(nf_2, nt_2, \overline{y}, \overline{z})$

The sets of variables $\overline{x}, \overline{y}$ represent the free variables of the formula A_{α_1} and $\overline{y}, \overline{z}$ of the formula A_{α_2}.

Again, immediately from the definition we have:

Lemma 2. *Let P_{α_1} represent A_{α_1} and P_{α_2} represent A_{α_2}. Then $P_{\alpha_1\wedge\alpha_2}$ represents $A_{\alpha_1\wedge\alpha_2}$.*

The projection automaton which represents the existential quantification of a given formula is defined as follows.

Definition 6. *The program $P^u_{\exists\overline{x}:\alpha}$ is defined as the union of P_α with the clauses*

1. $\mathsf{Node}^u_{\exists\overline{x}:\alpha}(n) \leftarrow \mathsf{Node}_\alpha(n)$
2. $\mathsf{Start}^u_{\exists\overline{x}:\alpha}(n) \leftarrow \mathsf{Start}_\alpha(n)$

3. $\mathsf{Final}^u_{\exists\overline{x}:\alpha}(n) \leftarrow \mathsf{Final}_\alpha(n)$
   $\mathsf{Final}^u_{\exists\overline{x}:\alpha}(n_0) \leftarrow \mathsf{Trans}_\alpha(n_0, n_1, \overline{x}, \overline{o}), \mathsf{Final}^u_{\exists\overline{x}:\alpha}(n_1)$
4. $\mathsf{Trans}^u_{\exists\overline{x}:\alpha}(nf_1, nt_1, \overline{y}) \leftarrow \mathsf{Trans}_\alpha(nf_1, nt_1, \overline{x}, \overline{y})$

The sets of variables $\overline{y}$ *and* $\overline{x}$ *represent the free variables of the formula* α*, and* $\overline{o} = \{0, 0, \ldots, 0\}$ *where* $|\overline{o}| = |\overline{y}|$*.*

Lemma 3. *If* P_α *represents* A_α *then* $P^u_{\exists\overline{x}:\alpha}$ *represents* $A^u_{\exists\overline{x}:\alpha}$ *which is nondeterministic automaton for the formula* $\exists\overline{x} : \alpha$*.*

The automaton obtained by the projection operation is nondeterministic. The following Datalog$^{\mathrm{cv}}$ program produces the representation of a deterministic automaton which accepts the same language as the nondeterministic one.

Definition 7. *The program* $P_{\exists\overline{x}:\alpha}$ *consists of the program* $P^u_{\exists\overline{x}:\alpha}$ *and the following clauses*

1. $\mathsf{Node}_{\exists\overline{x}:\alpha}(N) \leftarrow \mathsf{Start}_{\exists\overline{x}:\alpha}(N)$
   $\mathsf{Node}_{\exists\overline{x}:\alpha}(N) \leftarrow \mathsf{Node}_{\exists\overline{x}:\alpha}(N_1), \mathsf{Trans}_{\exists\overline{x}:\alpha}(N_1, N, \overline{x})$
2. $\mathsf{Start}_{\exists\overline{x}:\alpha}(\{n\}) \leftarrow \mathsf{Start}^u_{\exists\overline{x}:\alpha}(n)$
3. $\mathsf{Final}_{\exists\overline{x}:\alpha}(N) \leftarrow \mathsf{Node}_{\exists\overline{x}:\alpha}(N), \mathsf{Final}^u_{\exists\overline{x}:\alpha}(n), n \in N$
4. $\mathsf{Trans}_{\exists\overline{x}:\alpha}(N_1, \langle n\rangle, \overline{x}) \leftarrow \mathsf{Node}_{\exists\overline{x}:\alpha}(N_1), \mathsf{Next}_{\exists\overline{x}:\alpha}(N_1, n, \overline{x})$
   $\mathsf{Next}_{\exists\overline{x}:\alpha}(N_1, n_2, \overline{x}) \leftarrow n_1 \in N_1, \mathsf{Trans}^u_{\exists\overline{x}:\alpha}(n_1, n_2, \overline{x})$

Lemma 4. *If* $P^u_{\exists\overline{x}:\alpha}$ *represents* $A^u_{\exists\overline{x}:\alpha}$ *then* $P_{\exists\overline{x}:\alpha}$ *represents a deterministic automaton* $A_{\exists\overline{x}:\alpha}$*.*

Last, the test for emptiness of an automaton has to be defined: To find out whether the language accepted by A_α is non-empty and thus whether α is satisfiable, a *reachability (transitive closure)* query is used.

Definition 8. *The following program* TC_α *computes the transitive closure of the transition function of* A_α*.*

1. $\mathsf{TransClos}_\alpha(n, n) \leftarrow$
2. $\mathsf{TransClos}_\alpha(nf_1, nt_1) \leftarrow \mathsf{Trans}_\alpha(nf_1, nt_2, \overline{x}), \mathsf{TransClos}_\alpha(nt_2, nt_1)$

Note that the use of magic sets and/or SLG resolution automatically transforms the transitive closure query into a reachability query.

Theorem 1. *Let* φ *be a WS1S (WS2S) formula. Then* φ *is satisfiable if and only if* $P_\varphi, TC_\varphi \models \mathsf{Start}_\varphi(x), \mathsf{Final}_\varphi(y), \mathsf{TransClos}_\varphi(x, y)$*.*

Example 3. Suppose that we have a formula $\exists y : y \subseteq x$, let A_ϕ be the automaton for the subformula $\phi = y \subseteq x$, we can use the following logic program to construct the automaton $A_{\exists y:\phi}$:

$\mathsf{Node}^u_{\exists y:\phi}(n) \leftarrow \mathsf{Node}_\phi(n)$
$\mathsf{Start}^u_{\exists y:\phi}(n) \leftarrow \mathsf{Start}_\phi(n)$
$\mathsf{Final}^u_{\exists y:\phi}(n) \leftarrow \mathsf{Final}_\phi(n)$
$\mathsf{Final}^u_{\exists y:\phi}(n_0) \leftarrow \mathsf{Trans}_\phi(n_0, n_1, 0, y), \mathsf{Final}^u_{\exists y:\phi}(n_1)$
$\mathsf{Trans}^u_{\exists y:\phi}(n_1, n_2, x) \leftarrow \mathsf{Trans}_\phi(n_1, n_2, x, y)$

This part computes the nondeterministic automaton ($A^u_{\exists y:\phi}$) representing the formula (see Definition 6).

$$\mathsf{Node}_{\exists y:\phi}(N) \leftarrow \mathsf{Start}_{\exists y:\phi}(N)$$
$$\mathsf{Node}_{\exists y:\phi}(N) \leftarrow \mathsf{Node}_{\exists y:\phi}(N_1),\ \mathsf{Trans}_{\exists y:\phi}(N_1, N, x)$$
$$\mathsf{Start}_{\exists y:\phi}(\{n\}) \leftarrow \mathsf{Start}^u_{\exists y:\phi}(n)$$
$$\mathsf{Final}_{\exists y:\phi}(N) \leftarrow \mathsf{Node}_{\exists y:\phi}(N), \mathsf{Final}^u_{\exists y:\phi}(n), n \in N$$
$$\mathsf{Trans}_{\exists y:\phi}(N_1, \langle n \rangle, x) \leftarrow \mathsf{Node}_{\exists y:\phi}(N_1), \mathsf{Next}_{\exists y:\phi}(N_1, n, x)$$
$$\mathsf{Next}_{\exists y:\phi}(N_1, n_2, x) \leftarrow n_1 \in N_1, \mathsf{Trans}^u_{\exists y:\phi}(n_1, n_2, x)$$
$$\mathsf{TransClos}_{\exists y:\phi}(n, n) \leftarrow$$
$$\mathsf{TransClos}_{\exists y:\phi}(n_1, n_2) \leftarrow \mathsf{Trans}_{\exists y:\phi}(n_1, n_3, x), \mathsf{TransClos}_{\exists y:\phi}(n_3, n_2)$$

This part computes the deterministic automaton ($A_{\exists y:\phi}$) representing the formula (see Definition 7), and the transitive closure of its transition function (see Definition 8). Note that determinization is not needed unless there is a negation operation after this step. The satisfiability query is:

$$\leftarrow \mathsf{Start}_{\exists y:\phi}(n), \mathsf{Final}_{\exists y:\phi}(m), \mathsf{TransClos}_{\exists y:\phi}(n, m).$$

The use of SLG resolution to evaluate the transitive closure goal allows us to construct only the relevant parts of the automaton in a goal-driven way:

Example 4. For the formula $\phi = \neg(x \in v) \vee \neg(\exists w : (y \in w) \wedge (z \in w))$ the bottom-up evaluation creates 240 transitions, and 16 transitive closure tuples for the starting node while the top-down evaluation with memoing technique creates only 1 transition, and 1 tuple in the transitive closure for the starting node as shown in Figure 2.

4 Experimental Evaluation

We compare the performance of the technique proposed in this paper and implemented using the XSB system[3] with the MONA system [15,17], one of the most advanced tools for reasoning in weak second-order logics (WS1S and WS2S).

The performance results for a set of formulas are given in Figures 3 and 4. We present a sample set of size 10 from the set of formulas we used in the experiments where #i represents a particular formula. The response times are measured in seconds; N/A means "Not Answered" in 120 seconds. The formulas are similar to the ones in T98 satisfiability test suite except we varied their sizes, the number of existential quantifiers, and free variables.

The results show that XSB outperforms MONA for the formulas with many free variables since it performs large numbers of conjunction operations very efficiently with the use of top-down query evaluation and pruning techniques. This can be easily traced to the effects of goal-driven evaluation of P_φ which become more pronounced for large theories consisting of relatively simple formulæ, such as those corresponding to constraints used in database schemata

[3] We simulate the set values in Datalog^{cv} by lists in XSB.

$? - \mathsf{Start}_\phi(n), \mathsf{Final}_\phi(m), \mathsf{TransClos}_\phi(n, m).$
(0) $Call : \mathsf{Start}_\phi(n)?$
...
...
(0) $Exit : \mathsf{Start}_\phi([1, \{[1, 1]\}])?$
(8) $Call : \mathsf{Final}_\phi(m)?$
...
...
(8) $Exit : \mathsf{Final}_\phi([1, \{[1, 1]\}])?$
(18) $Call : \mathsf{TransClos}_\phi([1, \{[1, 1]\}], [1, \{[1, 1]\}])?$
(19) $Call : \mathsf{Trans}_\phi([1, \{[1, 1]\}], [1, \{[1, 1]\}], x, y, z, v)?$
...
...
(19) $Exit : \mathsf{Trans}_\phi([1, \{[1, 1]\}], [1, \{[1, 1]\}], 0, 0, 0, 0)?$
(18) $Exit : \mathsf{TransClos}_\phi([1, \{[1, 1]\}], [1, \{[1, 1]\}])?$

$n = [1, \{[1, 1]\}]$
$m = [1, \{[1, 1]\}]$

Fig. 2. Top-down evaluation of the program in Example 4

or UML diagrams. The experiments also compared different scheduling strategies of XSB namely the batched(XSB B) and the local(XSB L) ones. Batched scheduling performs better than local since our programs do not require answer subsumption. Experiments also show that tabling more predicates in addition to the auto-tabled ones (results in columns XSB B(T) and XSB L(T)) increases space requirements but enhances the performance substantially. The additional predicates we tabled are the Trans predicates in the programs that represent the determinization step. Since this step is critical in automaton construction, tabling the Trans predicate in addition to the Node predicate gives better results (see formulas 5, 9, 10).

On the other hand, MONA usually performs better on formulas that have less free variables and more quantifiers as it performs the projection operation faster than XSB. We believe that this is a practical problem caused by the implementation XSB uses for the evaluation of programs with nested relations and can be avoided using a more sophisticated implementation of Datalog$^{\text{cv}}$. In addition to this MONA uses a compact representation of automata based on BDDs [15,17,18] to enhance its performance, whereas XSB uses tries as the basis for tables combined with unification factoring [23,12]. The size of the trie structures is, in general, larger than the size of a corresponding BDD. However it is easier to insert tuples to a trie than into a BDD.

In the preliminary experiments [30] we conducted we also used CORAL, a deductive system that supports Datalog$^{\text{cv}}$ and Magic sets. Our results showed that CORAL also performs better than MONA for the same formulas as XSB, however XSB is faster than CORAL in all cases.

	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10
MONA	2.66	4.95	N/A	N/A	0.42	0.01	0.01	0.05	0.09	0.39
XSB B	0.01	0.01	0.11	0.01	35.72	1.74	N/A	0.01	6.02	94.64
XSB B(T)	0.01	0.01	0.01	0.01	15.88	0.18	N/A	0.01	0.29	10.96
XSB L	0.01	0.01	1.68	0.01	41.33	N/A	N/A	12.59	8.52	N/A
XSB L(T)	0.01	0.01	1.73	0.01	15.03	N/A	N/A	6.63	0.73	N/A

Fig. 3. Performance (secs) w.r.t. increasing number of quantifiers

	#7	#6	#8	#10	#5	#9	#1	#2	#3	#4
MONA	0.01	0.01	0.05	0.39	0.42	0.09	2.66	4.95	N/A	N/A
XSB B	N/A	1.74	0.01	94.64	35.72	6.02	0.01	0.01	0.11	0.01
XSB B(T)	N/A	0.18	0.01	10.96	15.88	0.29	0.01	0.01	0.01	0.01
XSB L	N/A	N/A	12.59	N/A	41.33	8.52	0.01	0.01	1.68	0.01
XSB L(T)	N/A	N/A	6.63	N/A	15.03	0.73	0.01	0.01	1.73	0.01

Fig. 4. Performance (secs) w.r.t. increasing number of variables

4.1 Heuristics for (Large) Conjunctions of Formulas

Representing theories that capture database schemas and/or UML diagrams often leads to large conjunctions of relatively simple formulas. Hence we develop heuristics that improve on the naive translation of a formula φ to a Datalog$^{\text{cv}}$ program P_φ presented in Section 3. Many of these heuristics are based on adapting existing optimization techniques for logic programs.

First, given a formula $\varphi = \varphi_1 \wedge \varphi_2 \wedge \ldots \wedge \varphi_n$ we have to decide which way the conjunctions should be associated (parenthesized). Figure 5 shows how performance depends on parenthesizing of a 4-way conjunction. In the experiment we test all permutations of two sets of 4 formulas w.r.t. all possible parenthesizations. The table reports the best and average times over all permutations for a given parenthesization. The results show that left associative parenthesizing is generally preferable.

To take advantage of the structure of the input conjunction, we propose another heuristics that produces a more appropriate goal ordering. We use a structure called a *formula graph*: the nodes of the graph G_φ are the conjuncts of φ and the edges connect formulas that share variables (the edge labels list the shared variables).

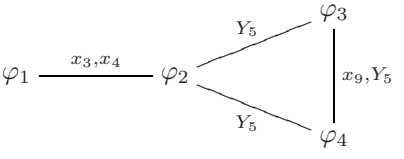
Example 5. Consider a formula $\varphi = \varphi_1 \wedge \varphi_2 \wedge \varphi_3 \wedge \varphi_4$ where:

$$\begin{aligned}
\varphi_1 &= \exists x_{10} : (((\exists Y_{30} : ((x_{10} \in Y_{30}) \wedge (x_4 \in Y_{30}))) \wedge (\exists Y_{40} : (x_{10} \in Y_{40}))) \\
&\quad \wedge \neg(\exists Y_{20} : ((x_{10} \in Y_{10}) \wedge (x_3 \in Y_{20})))) \\
\varphi_2 &= \neg((\exists x_1 : ((x_1 \in Y_5) \wedge (x_2 \in Y_5))) \wedge ((x_3 \in Y_5) \wedge (x_4 \in Y_5))) \\
\varphi_3 &= \neg(x_9 \in Y_5) \\
\varphi_4 &= (x_9 \in Y_5)
\end{aligned}$$

Formula	Parenthesizing	Best Time	Average Time	# Not Answered
1	$\varphi_i \wedge (\varphi_j \wedge (\varphi_k \wedge \varphi_l))$	0.56	23.23	5
	$((\varphi_i \wedge \varphi_j) \wedge (\varphi_k \wedge \varphi_l))$	0.77	7.00	4
	$((\varphi_i \wedge \varphi_j) \wedge \varphi_k) \wedge \varphi_l$	0.62	5.67	0
	$(\varphi_i \wedge (\varphi_j \wedge \varphi_k)) \wedge \varphi_l$	0.61	8.60	1
	$\varphi_i \wedge ((\varphi_j \wedge \varphi_k) \wedge \varphi_l)$	0.59	10.26	6
2	$\varphi_i \wedge (\varphi_j \wedge (\varphi_k \wedge \varphi_l))$	0.72	14.56	12
	$((\varphi_i \wedge \varphi_j) \wedge (\varphi_k \wedge \varphi_l))$	1.26	25.43	8
	$((\varphi_i \wedge \varphi_j) \wedge \varphi_k) \wedge \varphi_l$	0.94	24.18	2
	$(\varphi_i \wedge (\varphi_j \wedge \varphi_k)) \wedge \varphi_l$	1.65	27.00	5
	$\varphi_i \wedge ((\varphi_j \wedge \varphi_k) \wedge \varphi_l)$	0.74	18.96	11

Fig. 5. Performance (secs) Results w.r.t. Associativity

The formula graph for φ is as follows:



For this heuristic we also need to estimate size of the automaton A_φ. We use only very simple estimation rules for the automata operations:

- $|A_{\neg\varphi}| = |A_\varphi|$
- $|A_{\varphi_1 \wedge \varphi_2}| = |A_{\varphi_1}| \times |A_{\varphi_2}|$
- $|A_{\exists \overline{x}:\varphi}| = 2^{|A_\varphi|}$

The goal ordering heuristics for a formula $\varphi = \varphi_1 \wedge \varphi_2 \wedge \ldots \wedge \varphi_n$ constructs a left-associative permutation as follows: it starts from the conjunct that has the largest estimated automaton and then finds its neighbor with the largest automaton (alternatively selecting another conjunct if there are no conjuncts left that satisfy this criteria). This step is repeated until all the conjuncts are processed. Intuitively in the case where a conjunction is applied on a large automaton and a small automaton, when top-down evaluation is used, for every final state we find in the first automaton we check all the final states of the second one and see if they form a final state in the conjunction automaton. Since we iterate on a small sized automaton in this case, ordering the formulas starting from the large ones is heuristically better. The experimental results shown in Figure 6 support this optimization. In the table *heuristic time* is the response time of the program for the rewriting generated by the proposed heuristics, *best time* is the fastest response time among all the programs generated for the formula, and similarly *worst time* is the slowest response time. The experiments show that in many cases the heuristic achieves a performance close to the performance of the program for the best possible ordering.

# of Conjuncts	Formula	Heuristic Time	Best Time	Worst Time
3	1	68.17	67.97	N/A
	2	68.45	68.45	N/A
	3	7.60	7.60	N/A
	4	94.46	1.04	N/A
	5	N/A	5.14	N/A
	6	0.42	0.42	3.18
4	1	1.06	0.56	N/A
	2	3.81	0.72	N/A
	3	0.66	0.64	N/A
5	1	12.61	0.94	N/A
	2	15.94	0.92	N/A
	3	2.6	0.50	N/A

Fig. 6. Performance (secs) results on ordering

5 Related Work

The connection between logic and automata was first considered by Büchi [6] and Elgot [13]. They have shown that monadic second-order logic over finite words and finite automata have the same expressive power, and we can transform formulas of this logic to finite automata and vice versa. Later, Büchi [7], McNaughton [19], and Rabin [22] proved that monadic second-order logic over infinite words (and trees) and finite automata also have the same expressive power. The practical use of this connection was investigated for temporal logics and fixed-point logics which led to the theory of model checking [4,32]. Another automata-theoretic construction was for μ-calculus [16,31] and could be used, in turn, for reasoning in expressive description logics. An extensive survey on automata and logic can be found in [28].

The logic-automaton connection has been used for implementing decision procedures for various logics. It is argued that the success of these procedures relies on efficient operations on a compact representation of automata based on BDDs [17,18].

We have used deductive techniques to represent and query nested-relational representation of finite automata. The systems used to test our implementation are CORAL [24,25,26] (in the preliminary experiments [30]), which provides the set-oriented data manipulation characteristics of relational systems, and XSB [27] (here sets have to be explicitly simulated). In terms of evaluation strategy, XSB uses top-down evaluation with memoing, whereas CORAL uses magic sets. The resolution technique XSB supports performs basic operations such as unification very efficiently also providing alternative scheduling strategies [14]. There are numerous other deductive systems which support logic-programming languages with sets and tuples, e.g., LDL [11,21]. There is also a BDD-based deductive database system called bddbddb [33] implemented to be used in program verification which translates Datalog programs into efficient BDD implementations. Hence, the techniques presented in this paper are complementary to BDDs.

Considerable work has been done on query optimization in relational and deductive database systems [20]. Query optimization in relational systems includes choosing join orders and cost models [8,29]. We use the ideas of SLG resolution [9,10], and magic sets rewriting [3] as deductive database optimization methods, and we are planning to use cost-based optimization methods to improve our query evaluation.

6 Conclusions and Future Work

In this paper, we presented a translation technique that maps satisfiability questions for formulas in WS1S to query answering in Datalog^{cv}. We have also demonstrated how evaluation techniques used for answering queries on these programs can provide efficient decision procedures for second-order logics. In addition we also study the impact of goal reordering and various other query optimization techniques on the performance of the decision procedure and propose heuristics for this purpose.

Future extensions of the proposed approach include extending the translation to other types of automata on infinite objects, e.g., to Rabin [22] and Alternating Automata [31], and on improving the upper complexity bounds by restricting the form of Datalog^{cv} programs generated by the translation (when used for decision problems in, e.g., EXPTIME). In all these cases, the goal is to match the optimal theoretical bounds while avoiding the worst-case behavior (inherent in most automata-based techniques) in as many situations as possible. We also consider the integration of the technique proposed in this paper with more standard techniques such as BDDs.

References

1. Abiteoul, S., Beeri, C.: The power of languages for the manipulation of complex values. VLDB Journal 4(4), 727–794 (1995)
2. Beeri, C., Naqvi, S., Shmueli, O., Tsur, S.: Set construction in a logic database language. JLP 10(3&4), 181–232 (1991)
3. Beeri, C., Ramakrishnan, R.: On the power of Magic. JLP 10(1/2/3&4), 255–299 (1991)
4. Bernholtz, O., Vardi, M.Y., Wolper, P.: An automata-theoretic approach to branching-time model checking. In: Dill, D.L. (ed.) CAV 1994. LNCS, vol. 818, pp. 142–155. Springer, Heidelberg (1994)
5. Bryant, R.E.: Symbolic boolean manipulation with Ordered Binary Decision Diagrams. ACM Computing Surveys 24(3), 293–318 (1992)
6. Büchi, J.R.: Weak second-order arithmetic and finite automata. Z. Math. Logik Grundl. Math. 6, 66–92 (1960)
7. Büchi, J.R.: On a decision method in restricted second-order arithmetic. In: Proc. Int. Congr. for Logic, Methodology and Philosophy of Science, pp. 1–11 (1962)
8. Chaudhuri, S.: An overview of query optimization in relational systems. In: PODS, pp. 34–43 (1998)
9. Chen, W., Swift, T., Warren, D.S.: Efficient top-down computation of queries under the well-founded semantics. JLP 24(3), 161–199 (1995)
10. Chen, W., Warren, D.S.: Query evaluation under the well-founded semantics. In: PODS, pp. 168–179 (1993)

11. Chimenti, D., Gamboa, R., Krishnamurthy, R., Naqvi, S.A., Tsur, S., Zaniolo, C.: The LDL system prototype. IEEE Trans. Knowl. Data Eng. 2(1), 76–90 (1990)
12. Dawson, S., Ramakrishnan, C.R., Skiena, S., Swift, T.: Principles and practice of unification factoring. TOPLAS 18(5), 528–563 (1996)
13. Elgot, C.C.: Decision problems of finite automata design and related arithmetics. Trans. Amer. Math. Soc. 98, 21–52 (1961)
14. Freire, J., Swift, T., Warren, D.S.: Beyond depth-first strategies: Improving tabled logic programs through alternative scheduling. Journal of Functional and Logic Programming 1998(3) (1998)
15. Henriksen, J.G., Jensen, J.L., Jörgensen, M.E., Klarlund, N., Paige, R., Rauhe, T., Sandholm, A.: MONA: Monadic second-order logic in practice. In: Brinksma, E., Steffen, B., Cleaveland, W.R., Larsen, K.G., Margaria, T. (eds.) TACAS 1995. LNCS, vol. 1019, pp. 89–110. Springer, Heidelberg (1995)
16. Janin, D., Walukiewicz, I.: Automata for the modal μ-calculus and related results. In: Hájek, P., Wiedermann, J. (eds.) MFCS 1995. LNCS, vol. 969, pp. 552–562. Springer, Heidelberg (1995)
17. Klarlund, N.: MONA & FIDO: The logic-automaton connection in practice. In: Computer Science Logic, pp. 311–326 (1997)
18. Klarlund, N., Møller, A., Schwartzbach, M.I.: MONA implementation secrets. Int. J. Found. Comput. Sci. 13(4), 571–586 (2002)
19. McNaughton, R.: Testing and generating infinite sequences by a finite automaton. Information and Control 9, 521–530 (1966)
20. Mumick, I.S.: Query Optimization in Deductive and Relational Databases. PhD thesis, Department of Computer Science, Stanford University (1991)
21. Naqvi, S., Tsur, S.: A Logical Language for Data and Knowledge Bases. Computer Science Press (1989)
22. Rabin, M.O.: Decidability of second-order theories and automata on infinite trees. Trans. Amer. Math. Soc. 141, 1–35 (1969)
23. Ramakrishnan, I., Rao, P., Sagonas, K., Swift, T., Warren, D.: Efficient tabling mechanisms for logic programs. In: Proc. ICLP, pp. 697–711 (1995)
24. Ramakrishnan, R., Bothner, P., Srivastava, D., Sudarshan, S.: CORAL - a database programming language. In: Workshop on Deductive Databases (1990)
25. Ramakrishnan, R., Srivastava, D., Sudarshan, S.: Coral - control, relations and logic. In: Proc. VLDB Conference, pp. 238–250 (1992)
26. Ramakrishnan, R., Srivastava, D., Sudarshan, S., Seshadri, P.: The CORAL deductive system. VLDB Journal 3(2), 161–210 (1994)
27. Sagonas, K.F., Swift, T., Warren, D.S.: XSB as an efficient deductive database engine. In: Proc. SIGMOD, pp. 442–453 (1994)
28. Thomas, W.: Languages, automata, and logic. In: Handbook of Formal Languages, vol. 3, Springer, Heidelberg (1997)
29. Ullman, J.D.: Principles of Database and Knowledge-Base Systems, vol. 1&2. Computer Science Press (1989)
30. Unel, G., Toman, D.: Deciding weak monadic second-order logics using complex-value datalog. In: Proc. LPAR (Short Paper) (2005)
31. Vardi, M.Y.: Reasoning about the past with two-way automata. In: Larsen, K.G., Skyum, S., Winskel, G. (eds.) ICALP 1998. LNCS, vol. 1443, pp. 628–641. Springer, Heidelberg (1998)
32. Vardi, M.Y., Wolper, P.: An automata-theoretic approach to automatic program verification. In: Proc. LICS, pp. 322–331 (1986)
33. Whaley, J., Lam, M.S.: Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In: Proc. PLDI '04, pp. 131–144 (2004)

A Logic Programming Framework for Combinational Circuit Synthesis

Paul Tarau[1] and Brenda Luderman[2]

[1] Department of Computer Science and Engineering
University of North Texas
paul.tarau@gmail.com
[2] Intel Corp.
Austin, Texas
brenda.luderman@gmail.com

Abstract. Logic Programming languages and combinational circuit synthesis tools share a common "combinatorial search over logic formulae" background. This paper attempts to reconnect the two fields with a fresh look at Prolog encodings for the combinatorial objects involved in circuit synthesis. While benefiting from Prolog's fast unification algorithm and built-in backtracking mechanism, efficiency of our search algorithm is ensured by using parallel bitstring operations together with logic variable equality propagation, as a mapping mechanism from primary inputs to the leaves of candidate Leaf-DAGs implementing a combinational circuit specification. After an exhaustive expressiveness comparison of various minimal libraries, a surprising first-runner, Strict Boolean Inequality "$<$" together with constant function "1" also turns out to have small transistor-count implementations, competitive to NAND-only or NOR-only libraries. As a practical outcome, a more realistic circuit synthesizer is implemented that combines rewriting-based simplification of $(<, 1)$ circuits with exhaustive Leaf-DAG circuit search.

Keywords: logic programming and circuit design, combinatorial object generation, exact combinational circuit synthesis, universal boolean logic libraries, symbolic rewriting, minimal transistor-count circuit synthesis.

1 Introduction

Various logic programming applications and circuit synthesis tools share algorithmic techniques ranging from search over combinatorial objects and constraint solving to symbolic rewriting and code transformations.

The significant semantic distance between the two fields, coming partly from the application focus and partly from the hardware/software design gap has been also widened by the use of lower level procedural languages for implementing circuit design tools - arguably for providing better performance fine tuning opportunities.

While intrigued by the semantic and terminological gap between the two fields, our interest in the use of logic programming for circuit design has been encouraged because of the following facts:

V. Dahl and I. Niemelä (Eds.): ICLP 2007, LNCS 4670, pp. 180–194, 2007.

- the simplicity and elegance of combinatorial generation algorithms in the context of Prolog's backtracking, unification and logic grammar mechanisms
- the structural similarity between Prolog terms and the Leaf-DAGs typically used as a data structure for synthesized circuits
- elegant implementations of circuit design tools in high level functional languages [10]
- the presence of new flexible constraint solving Prolog extensions like *CHR* [6] that could express layout, routing and technology mapping aspects of the circuit design process needed, besides circuit synthesis, for realistic design tools.

The paper summarizes our efforts on solving some realistic combinational circuit synthesis problems with logic programming tools. It is organized as follows. Section 2 describes the generation of combinatorial objects needed for exact circuit synthesis in Prolog, section 3 shows uniform bitstring representations for functions and primary inputs that check function equivalence without backtracking. Section 4 compares various universal boolean function libraries in terms of total cost of minimal representations of the set of 16 2-argument operators as an indicator of expressiveness for minimal cost synthesis purposes. As result of this comparison, section 5 focuses on a surprisingly interesting library consisting of Strict Boolean Inequality and constant function 1 with subsection 5.1 showing universality of $(<, 1)$ and subsection 5.2 presenting our library specific rewriting algorithm, usable as minimization heuristics when exact synthesis becomes intractable. Section 6 describes low transistor-count implementations of the $<$-gate and compares transistor counts for $(<, 1)$ with equivalent NAND-based circuits. Sections 7 and 8 discuss related and future work and section 9 concludes the paper. The Prolog code for the exact synthetizer and various libraries is available at `http://logic.csci.unt.edu/tarau/research/2007/csyn.zip`.

2 Combinatorial Objects and Combinational Circuit Synthesis

Our exact synthesis algorithm uses Prolog's depth-first backtracking to find minimal circuits representing boolean functions, based on a given library of primitives, through composition of combinatorial generation steps and efficient checking against an output pattern specified as a truth table.

The general structure of the algorithm is as follows:

Through the paper, Leaf-DAGs will be used to represent the synthetized circuits. The general structure of the algorithm is as follows:

1. First, the algorithm runs a library specific rewriting module (see 5.2 for a library specific rewriting module) on the input formula (in symbolic, CNF or DNF form). This (or a conservative higher estimate) provides an upper limit (in terms of a cost function, for instance the number of gates) on the size of the synthesized expression. It also provides a (heuristically) minimized

formula, in therms of the library, that can be returned as output if exact synthesis times out.
2. Next, if the formula qualifies for exact synthesis, we enumerate candidate trees *in increasing cost order*, to ensure that minimal trees are generated first. This involves the following steps:
   (a) First, we implement a mapping from the set primary inputs to the set of their occurrences in a tree. This involves generating functions from N variables to M occurrences. We achieve this without term construction, through logical variable bindings, efficiently undone on backtracking by Prolog's trailing mechanism.
   (b) Next, N-node binary trees of increasing sizes are generated. The combination of the expression trees and the mapping of logic variables (representing primary inputs) to their (possibly multiple) occurrences, generates Leaf-DAGs.
3. Finally, we evaluate candidate Leaf-DAGs for equivalence with the output specification.

We will describe the details of the algorithm and the key steps of their Prolog implementation in the following subsections.

2.1 Boolean Expression Trees

Size-constrained expression trees are combinatorial objects providing the skeletons for the Leaf-DAGs generated by our algorithm, as shown in predicate `enumerate_tree_candidates/5`. The constraints are expressed as input parameters `UniqueVarAndConstCount` and `LeafCount`. The generator produces an expression tree `ETree` and computes its truth table `OutSpec` encoded as a bitstring-integer. Size-constraints, ensuring termination of the recursive predicate `generate_expression_tree/7`, are encoded as a finite list of nodes, using DCG notation. Termination is ensured by having each recursive step consume exactly one node. A finite list of leaf variables provides leaves to the generated tree in the first clause of predicate `generate_expression_tree/7`.

```
enumerate_tree_candidates(UniqueVarAndConstCount,LeafCount,
                                          Leaves,ETree,OutputSpec):-
  N is LeafCount-1,
  length(Nodes,N),
  generate_expression_tree(UniqueVarAndConstCount,ETree,OutputSpec,
                                                  Leaves,[],Nodes,[]).

generate_expression_tree(_,V,V,[V|Leaves],Leaves)-->[].
generate_expression_tree(NbOfBits,ETree,OutputSpec,Vs1,VsN)-->[_],
  generate_expression_tree(NbOfBits,L,O1,Vs1,Vs2),
  generate_expression_tree(NbOfBits,R,O2,Vs2,VsN),
  {combine_expression_values(NbOfBits,L,R,O1,O2,ETree,OutputSpec)}.
```

The predicate `combine_expression_values/7`, shown below for the $(*, \oplus, 1)$ library, produces tree nodes like `L*R` and `L^R`, while computing their bitstring-integer encoded truth table `O` from the left and right branch values `O1` and `O2`.

```
combine_expression_values(_,L,R,O1,O2, L*R,O):-bitand(O1,O2,O).
combine_expression_values(_,L,R,O1,O2, L^R,O):-bitxor(O1,O2,O).
```

The generated trees have binary operators as internal nodes and variables as leaves. They are counted by Catalan numbers [15]), with 4^N as a (rough) upper bound for N leave trees.

2.2 Implementing Finite Functions as Logical Variable Bindings

We express finite functions as *bindings* of a list of logic variables (the range of the function) to values in the domain of the function.

```
functions_from([],_).
functions_from([V|Vs],Us):-member(V,Us),functions_from(Vs,Us).

Example: A call like

?- functions_from([A,B,C],[0,1])

enumerates the 8 functions as variable bindings like:

{A->0,B->0,C->0}
{A->0,B->0,C->1}
...
{A->1,B->1,C->1}
```

Assuming the first set has M elements and the second has N elements, a total of N^M backtracking steps are involved in the enumeration, one for each function between the two sets. As a result, a finite function can be seen simply as a set of variable occurrences. This provides fast combinatorial enumeration of function objects, for which backtracking only involves trailing of variable addresses and no term construction.

2.3 Leaf-DAG Circuit Representations for Combinational Logic

Definition 1. *A* Leaf-DAG *is a directed acyclic graph where only vertices (called leaves) that have no outgoing edges can have multiple incoming edges.*

Leaf-DAGs can be seen as trees with possibly merged leaves. Note that Leaf-DAGs are naturally represented as Prolog terms with multiple occurrences of some variables - like X and Y in $f(X, g(X, Y, Z), Y)$.

Our Leaf-DAG generator combines the size-constrained tree generator from subsection 2.1 and the functions-as-bindings generator from subsection 2.2, as follows:

```
generate_leaf_dag(UniqueVarAndConstCount,LeafCount,
                  UniqueVarsAndConsts,ETree,OutputSpec):-
  length(Leaves,LeafCount),
  functions_from(Leaves,UniqueVarsAndConsts),
  enumerate_tree_candidates(UniqueVarAndConstCount,LeafCount,
                                    Leaves,ETree,OutputSpec).
```

Proposition 1. *Let catalan(M) denote the M-th Catalan number. The total backtracking steps for generating all Leaf DAGs with N primary inputs and M binary operation nodes is catalan(M) $* N^{M+1}$.*

Proof. It follows from the fact that Catalan numbers count the trees and N^{M+1} counts the functions corresponding to mapping the primary inputs to their leaves, because a binary tree with M internal nodes, each corresponding to an operation, has $M + 1$ leaves.

Note that if constant functions like 0 or 1 are part of the library, they are simply added to the list of primary inputs.

The predicate `synthesize_leaf_dag/4` describes a (simplified version) of our Leaf-DAG generator. Note that if the OutputSpec truth table is given as a constant value, unification ensures that only LeafDAGs matching the specification are generated. With OutputSpec used as a free variable, the predicate can be used in combination with Prolog's dynamic database as part of a dynamic programming algorithm that tables and reuses subcircuits to avoid recomputation.

```
synthesize_leaf_dag(MaxGates,UniqueBitstringIntVars,
                    UniqueVarAndConstCount,PIs:LeafDag=OutputSpec):-
  constant_functions(UniqueVarAndConstCount,ICs,OCs),
  once(append(ICs,UniqueBitstringIntVars,UniqueVarsAndConsts)),
  for(NbOfGates,1,MaxGates),
    generate_leaf_dag(UniqueVarAndConstCount,NbOfGates,
                      UniqueVarsAndConsts,ETree,OutputSpec),
  decode_leaf_dag(ETree,UniqueVarsAndConsts,LeafDag,DecodedVs),
  once(append(OCs,PIs,DecodedVs)).
```

Proposition 2. *The predicate* `synthesize_leaf_dag/4` *generates Leaf-DAGs in increasing size order.*

Proof. It follows from the fact that each call to `generate_leaf_dag/5` enumerates on backtracking all Leaf-DAGs of size `NbOfGates` and the predicate `for/3` provides increasing `NbOfGates` bounds.

Assuming zero cost for constant functions and a fixed transistor cost for each operator, it follows that the synthesizer produces circuits based on *single-operator* libraries in increasing cost order. For more complex cost models adaptations to the tree generator can be implemented easily.

3 Fast Boolean Evaluation with Bitstring Truth Table Encodings

We use an adaptation of the clever bitstring-integer encoding described in the Boolean Evaluation section of [7] of n variables as truth tables. Let x_k be a variable for $0 \leq k < n$. Then $x_k = (2^{2^n} - 1)/(2^{2^{n-k-1}} + 1)$, where the number of distinct variables in a boolean expression gives n, the number of bits for the encoding. The mapping from variables, denoted as integers, to their truth table equivalents, is provided by the following Prolog code:

```
% Maps variable K in 0..Mask-1 to truth table
% Xk packed as a bitstring-integer.
var_to_bitstring_int(NbOfBits,K,Xk):-
  all_ones_mask(NbOfBits,Mask),
  NK is NbOfBits-(K+1),
  D is (1<<(1<<NK))+1,
  Xk is Mask//D.

% Mask is a bitstring-integer ending with NbOfBits of the form
% 11...1. It also provides an encoding of constant function 1.
all_ones_mask(NbOfBits,Mask):-Mask is (1<<(1<<NbOfBits))-1.
```

Variables representing such bitstring-truth tables can be combined with the usual bitwise integer operators to obtain new bitstring truth tables encoding all possible value combinations of their arguments, like in:

```
bitand(X1,X2,X3):-X3 is '/\'(X1,X2).
bitor(X1,X2,X3):-X3 is '\/'(X1,X2).
bitxor(X1,X2,X3):-X3 is '#'(X1,X2).
bitless(X1,X2,X3):-X3 is '#'(X1,'\/'(X1,X2)).
bitgt(X1,X2,X3):-X3 is '#'(X1,'/\'(X1,X2)).
bitnot(NbOfBits,X1,X3):-all_ones_mask(NbOfBits,M),X3 is '#'(X1,M).
biteq(NbOfBits,X,Y,Z):-all_ones_mask(NbOfBits,M),Z is '#'(M,'#'(X,Y)).
bitimpl(NbOfBits,X1,X2,X3):-bitnot(NbOfBits,X1,NX1),bitor(NX1,X2,X3).
bitnand(NbOfBits,X1,X2,X3):-bitand(X1,X2,NX3),bitnot(NbOfBits,NX3,X3).
bitnor(NbOfBits,X1,X2,X3):-bitor(X1,X2,NX3),bitnot(NbOfBits,NX3,X3).
```

The length of the bitstring-truth tables is sufficient for most perfect synthesis problems involving exhaustive search, as most problems become intractable above the 64 bits corresponding to 6 variables (see Proposition 1). However, using arbitrary length integer packages, available for most Prologs, allows extending the mechanism further. In practice, a timeout mechanism can be used to decide if falling back to a heuristic synthesis method is needed.

4 A Comparison of Universal Boolean Function Libraries

Definition 2. *A set of boolean functions F is universal if any boolean function can be written as a composition of functions in F.*

A well known universal set is (conjunction, negation) i.e. $(*, \sim)$ - this follows immediately from the rewriting of a truth table in terms of conjunction, disjunction and negation followed by elimination of disjunctions using De Morgan's laws. Universality of a library is usually proven by expressing conjunction and negation with its operations.

The table in Fig. 1 compares a few libraries used in synthesis with respect to the total gates needed to express all the 16 2-argument boolean operations

Library	total for 16 operators	non-redundant
$nand$	46	yes
nor	46	yes
$nand, 1$	33	no
$nor, 0$	33	no
$*, nand$	32	no
$<, nor$	31	no
$\Rightarrow, 0$	28	yes
$<, \mathbf{1}$	**28**	**yes**
$*, <, 1$	26	no
$*, \oplus, 1$	25	yes
$<, nand, 1$	25	no
$<, nor, 1$	24	no
$*, =, 0$	23	yes
$\Rightarrow, =, 0$	21	no
$<, =, 1$	21	no

Fig. 1. Relative Expressiveness of Libraries

(themselves included). The last column marks if the library is *non-redundant* (or *minimal*), i.e. if none of its functions can be expressed in terms of the others.

The comparison gives a hint on the relative expressiveness of libraries.

By including operations like "$\oplus$" and "$=$", that are known to require a relatively high number of other gates (or a high transistor count) to express, one can minimize the number of operators (and circuit size) required. Using only gates known to have low transistor-count implementations like `nand` and `nor`, the expressiveness drops significantly (46 required).

Surprisingly, $(\Rightarrow, 0)$ and its dual $(<, 1)$ do significantly better than `nand` and `nor`: they can express all 16 operators with only 28 gates. As section 6 will show, $(<, 1)$ turns out to have very interesting low transistor implementations. Given also that it has not been used in any work on synthesis that we are aware of, we will explore this library in depth in section 5.

Interestingly enough, the libraries $(*, =, 0)$ and $(\Rightarrow, =, 0)$ that provide, arguably, some the most human readable expressions when expressing other operators, have relatively small gate counts (23 and 21). For instance the first one expresses $A \Rightarrow B$ as $A = A * B$ and $A \oplus B$ as $(A = B) = 0$, the second one expresses $(A + B)$ as $(0 = A) => B$, and both express $(A \oplus B \oplus C)$ as $(A = (B = C))$.

Note also, that besides spotting out the minimal universal library $(<, 0)$, the comparison also identifies $(<, nor, 1)$ as a highly expressive library, with potential for practical design uses, given that $<$ and nor have both low transistor-count implementations.

Finally, one of the overall "winners" of the comparison is $(<, =, 1)$. It expresses the 16 operators with only 21 gates and it is a superset of the $(<, 1)$ library. This also suggests exploring in more detail the potential of $<$ for synthesis.

5 Using Strict Boolean Inequality for Combinational Circuit Synthesis

While Strict Boolean Inequality[1] $A < B$ together with 1 is a universal boolean function pair, it has been neglected by logicians as well as circuit designers, to the point where there are surprisingly few references to it in the literature in both fields. Interestingly enough, its *dual*, $(\Rightarrow, 0)$[2] – is a well known universal function pair that has been extensively studied as an axiomatic basis for both classical and intuitionistic propositional logic.

One can only speculate about the reasons for this neglect. The lack of algebraic grouping properties like commutativity and associativity comes to mind. Or, that its intuitive meaning would be harder to map to common reasoning patterns.

In any case, none of these are critical for the synthesis problem, which, stated generically, is about *finding minimal representations of finite functions*[3] in terms of a *universal subset* of them, given as a *library*.

As an indication of the usefulness of $(<, 1)$ for synthesis, let's note that $A \oplus B$ (known to be part of notoriously hard to synthesize boolean functions) is in fact $(A < B) + (B < A)$ and therefore $A < B$ can provide half of $A \oplus B$. Note that it also provides a form of conjunction (with first argument inverted), given its equivalence to $\sim A * B$. It follows from this equivalence, that $<$ also works as an inference rule: from its truth, one can determine uniquely the truth values of both of its arguments, i.e. the first should be `false` and the second `true`. As a side note, the reader might notice that this is similar, but in a way stronger than the mechanism through which *Modus Ponens* works. In the case of Modus Ponens, if one looks at its premises as a formula, then $A * (A \Rightarrow B)$ is equivalent to $A * B$ implying the truth of B in addition to the (already assumed) truth of A. The key difference, that makes Modus Ponens more intuitive is, of course, that it provides an inference mechanism that conserves and extends truth, while using $A < B$ as an inference mechanism would force one to deal with both true and false consequences.

5.1 Strict Boolean Inequality as a Universal Boolean Operator

Definition 3. *Strict Boolean Inequality is defined by the following truth table:*

A	B	$A < B$
0	*0*	*0*
0	*1*	*1*
1	*0*	*0*
1	*1*	*0*

[1] (Equivalent to $(\sim A) * B$ as well as $\sim (A \Leftarrow B)$). Called Converse Nonimplication as well as "NOT A BUT B" by Knuth [7]. Also called NIF standing for NOT (A IF B) and Half-XOR.

[2] Logical Implication with Falsehood (also denoted $\perp$).

[3] All finite functions can be expressed as boolean functions, by using binary encodings of their arguments and values.

Proposition 3. *Strict Boolean Inequality $A < B$ together with constant function 1 is a universal boolean function.*

Proof. Given that conjunction and negation form a universal boolean function pair, the proposition follows from the fact that conjunction $A * B$ has the same truth table as $(A < 1) < B$ and that negation $\sim A$ has the same truth table as $(A < 1)$.

5.2 The Symbolic Rewriting Algorithm

Our symbolic rewriting recurses over a given formula, and after each rewriting step, it proceeds with simplifications using propositional tautologies. We will illustrate the algorithm with the table in Fig. 2 showing how various expressions are transformed after the recursive rewriting of their arguments. For a given argument A we denote $`A$ the result of recursive application of the algorithm to A. The algorithm preserves constants and primary input variables unchanged. We also assume that simplification occurs *implicitly* after each transformation step.

From	To
0	0
1	1
$A < B$	$`A < `B$
$\sim A$	$`A < 1$
$A \Leftarrow B$	$(`A < `B) < 1$
$A * B$	$(`A < 1) < `B$
$nor(A, B)$	$`A < (`B < 1)$
$A + B$	$`(A \Leftarrow (\sim B))$
$A \Rightarrow B$	$(`B < `A) < 1$
$A \oplus B$	$(`A < `B) + (`B < `A)$
$A = B$	$`(nor((A < B), (B < A)) < 1)$
$ite(C, T, F)$	$`((C \Rightarrow T) * (\sim C \Rightarrow F))$

Fig. 2. $(<, 1)$-Rewriting Rules

The algorithm reduces most simple tautologies to 1 and most simple contradictions to 0. As a result, it also may reduce the number of variables on which the expression actually depends.

Optimizing for Minimal Transistor Count. Given that constant function 1 is 0-cost and that function $<$ has a 4-transistor cost (see section 6), the synthesis algorithm can assume that the cost is given by the number of $<$ gates.

Delay-Constrained Minimal Circuit Synthesis. Given the uniform gate structure of the circuits, we can ensure that delays are within acceptable margins by simply constraining the maximum length of the longest path from the primary inputs to the primary outputs.

5.3 Minimal $(<, 1)$-Representations for Key Boolean Functions

Figure 3 shows minimal representations for 0, negation, some 2-input boolean functions and the 3-argument IF-THEN-ELSE, as produced by our synthesizer. Interestingly enough, the minimal formulae obtained by exhaustive search are identical (as in the case of most simple formulae) with those obtained using our rewriting algorithm.

Function	"$<$" *Representation*
0	$1 < 1$
$\sim A$	$A < 1$
$A * B$	$(A < 1) < B$
$A + B$	$(A < (B < 1)) < 1$
$A \Rightarrow B$	$(B < A) < 1$
$A \Leftarrow B$	$(A < B) < 1$
$A \oplus B$	$((A < B) < ((B < A) < 1)) < 1$
$A = B$	$(A < B) < ((B < A) < 1)$
A NAND B	$((A < 1) < B) < 1$
A NOR B	$A < (B < 1)$
IF A THEN B ELSE C	$(A < (C < 1)) < ((B < A) < 1)$

Fig. 3. $(<, 1)$-Representations of some functions

5.4 Synthesis from CNF and DNF Forms

As Disjunctive Normal Forms (DNF, also called sum-of-products) and Conjunctive Normal Forms (CNF , also called product-of-sums) are the result of repeated conjunctions and disjunctions, we first focus on optimal $(<, 1)$-representation of these.

Proposition 4. *A sequence of disjunctions of N variables has a minimal* $(<, 1)$*-representation with 2 occurrences of constant 1 and exactly one occurrence of each input variable, provided by the formula:*

$$A_1 + A_2 + \ldots + A_N = (A_1 < (A_2 < \ldots (A_N < 1) \ldots)) < 1$$

Proposition 5. *A sequence of conjunctions of N variables has a minimal* $(<, 1)$*-representation with* $N - 1$ *occurrences of constant* 1 *and exactly one occurrence of each input variable, provided by the formula:*

$$A_1 * A_2 * \ldots A_{N-1} * A_N = ((A_1 < 1) < ((A_2 < 1) < \ldots ((A_{N-1} < 1) < A_N) \ldots)$$

Proof. By induction on the number of input variables, N.

Synthesis from CNF and DNF formulae (that can be obtained directly from truth table descriptions of circuits) proceeds by applying the encodings provided by the previous propositions recursively, followed by (and interleaved with) simplification steps.

6 Transistor Implementations for $(<, 1)$-Circuits

Clearly as $A < B$ is equivalent to $nor(A, \sim B)$, an obvious 6-transistor implementation is obtained when input B drives a 2-transistor inverter while its output and input A drive a 4-transistor NOR gate. This logic circuit is shown in Fig. 4. The output node, $A < B$, has a direct path to the power nodes VDD and VSS through the source connections of the transistors connected to it. As a result, the output is called "buffered" and the logic circuit type is "powered".

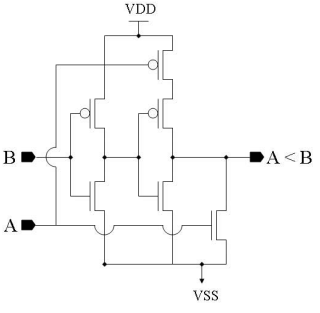


Fig. 4. Powered 6-Transistor $A < B$

To reduce transistor count, a *pass transistor logic* (PTL) circuit for $A < B$ can be implemented using 4 transistors. In this circuit, the output node, $A < B$, in Fig. 5 has a direct path to the power net VSS while input B provides the VDD power. Therefore, the logic circuit type is "semi-powered" and the output level for VDD is called "unbuffered".

Semi-Powered 4-Transistor

A	B	A < B
0	0	0
0	1	unbuffered '1'
1	0	0
1	1	0

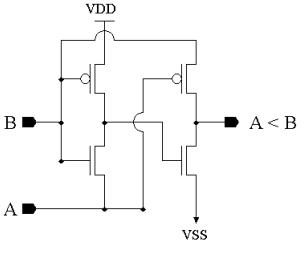


Fig. 5. Semi-Powered 4-Trans. $A < B$

The constant function 1 can be implemented by direct routing to the VDD power grid. Similary, the constant function 0 can be implemented by direct routing to the VSS power grid.

In conclusion, assuming a design using PTL-logic, the transistor count for an implementation of the $<$ function is 4, while constant functions 1 and 0 are essentially free, with transistor count 0.

Transistor Count Comparisons for $(<, 1)$ and NAND

To have a glimpse at the competitiveness of $(<, 1)$ as a universal pair, in comparison with a minimal NAND-based circuit, we have compared costs obtained as transistor counts of the resulting circuits.

While the comparison only involves Leaf-DAG representations and ignores the fact that stronger sharing could be present in the case of multiple outputs or the case arbitrary DAGs are used, it shows, at a small scale, that practical uses of the $(<, 1)$ for exact synthesis are likely to be competitive. Note also that in practice commutativity of NAND brings more sharing opportunities if general DAGs are used and that both $A < 1$ and $nand(A, A)$ can be replaced with 2-transistor inverters.

Function	$<$-cost	NAND-cost
$A = B$	4*4=16	5*4=20
$A \oplus B$	5*4=20	5*4=20
$A * B$	2*4= 8	3*4=12 (8 if sharing)
$(A * B) \Rightarrow C$	4*4=16	4*4=16
$A * B * C$	4*4=16	6*4=24
$A + B + C$	4*4=16	7*4=28
if-then-else	5*4=20	4*4=16
$(A \Rightarrow B) * (B \Rightarrow C))$	4*4=16	5*4=20
$nand(A, B)$	3*4=12	1*4=4
$A < B$	1*4	5*4=20 (16 if sharing)
2x2 half-adder	20+8=28	20+12=32

Fig. 6. Transistor Costs

7 Related Work

Mentions of Prolog for circuit simulation go back as early as [3]. Peter Reintjes in [12] mentions CMOS circuit design and Prolog as two *Elegant Technologies* with potential for interaction. Knuth in [7], section 7.1.2 mentions $A < B$ as forming one of the 5 (out of 16) boolean function used as part of a *boolean chain* (sequence of connected 2-argument boolean functions) needed for synthesis by exhaustive enumeration. Interestingly, the other 4 are: $>, *, +, \oplus$. Note that $>$ is the symmetric of $<$, and that with its exception, $*, +, \oplus$ have been heavily used in various synthesis algorithms. This supports our intuition that $<$ and $>$'s potential for synthesis is worth further exploration. Knuth also computes

minimal representations of all 5-argument functions using a clever reduction to equivalence classes and proves that the cost for representing almost every boolean function of N-arguments in terms of boolean chains exceeds $2^N/N$.

Rewriting/simplification has been used in various forms in recent work on multi-level synthesis [8,9] using non-SOP encodings ranging from And-Inverter Gates (AIGs) and XOR-AND nets to graph-based representations in the tradition of [1]. Interestingly, new synthesis targets, ranging from AIGs to cyclic combinational circuits [13], turned out to be competitive with more traditional minimization based synthesis techniques. Synthesis of reversible circuits with possible uses in low-power adiabatic computing and quantum computing [14] have emerged. Despite its super-exponential complexity, exact circuit synthesis efforts have been reported successful for increasingly large circuits [5,7]. While [12] describes the basics of CMOS technology, we refer the reader interested in full background information for our transistor models to [11].

8 Future Work

While we have provided unusually low cost transistor models for $<$ gates, the validation of their use in various context requires more extensive SPICE simulations as well precise area, delay and power estimates.

The relative simplicity of $A < B$ suggests its use in novel analog or non-silicon designs provided that one can measure that signal A is in a given sense weaker than B. Its relative expressiveness challenges, to some extent, the widely believed statement [4,2] that symmetric functions are genuinly more interesting for circuit synthesis. Future work is needed to substantiate our belief that this is not necessarily the case, based on the intuition that asymmetric operators cover a larger combinatorial space than their associative/commutative siblings.

The *dual* of the $(<, 1)$ library, $(\Rightarrow, 0)$ has the same expressive power as $(<, 1)$. It would be interesting to see if one can find similar low-transistor count implementations as the ones shown in this paper for $(<, 1)$. Given that $(\Rightarrow, 0)$ has been used as a foundation of various *implicative* formalizations of classic and intuitionistic logics, stronger rewriting mechanisms might be available for it than the ones we described in subsection 5.2 for $(<, 1)$, resulting in better heuristics for handling circuits for which exact synthesis is intractable.

On the general synthesis algorithm side, it would be interesting to add tabling of subcircuits (through the use of a system like XSB or by writing a special purpose circuit store) to avoid recomputation. Adapting intelligent backtracking mechanisms like those used in modern SAT-solvers should also be considered to improve performance.

Given our focus – to point out the usefulness of a relatively simple and unexplored primitive $<$ as a universal boolean function with small transistor count implementation – we have not invested an implementation effort comparable to high quality synthesis tools like [16]. An interesting development would be adapting a tool like **abc** [16] to specifically use $<$ as a primitive.

9 Conclusion

We have described a general logic programming based exact circuit synthesis algorithm and shown how Prolog language features like logic variables and backtracking can be used to provide efficient, a concise and elegant implementations. The synthesis algorithm has been used to identify the universal boolean function pair $(<, 1)$ as a primitive for circuit synthesis, after noticing the possibility of a 4-transistor PTL-implementation. We have also shown that, surprisingly, despite being non-commutative and non-associative, Strict Boolean Inequality "$<$" allows very low transistor-count implementations of typical small circuits. We have also provided a rewriting-based simplification algortithm in terms of $(<, 1)$ that handles symbolic boolean expressions as well as CNF or DNF forms. This rewriting algorithm is usable for heuristic circuit synthesis when formula complexity makes exact synthesis intractable. We hope that these results will provide practical opportunities for the use of logic programming languages and their constraint handling extensions as components of circuit design automation software.

References

1. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. IEEE Transactions on Computers 35(8), 677–691 (1986), citeseer.ist.psu.edu/bryant86graphbased.html
2. Canteaut, A., Videau, M.: Symmetric boolean functions. IEEE Transactions on Information Theory 51(8), 2791–2811 (2005)
3. Clocksin, W.F., Mellish, C.S.: Programming in Prolog, 3rd edn. Springer, Heidelberg (1987)
4. Dietmeyer, D.L.: Logic Design of Digital Systems. Allyn and Bacon (1971)
5. Drechsler, R., Gunther, W.: Exact Circuit Synthesis. In: International Workshop on Logic Synthesis (1998), http://citeseer.ist.psu.edu/drechsler98exact.html
6. Fruhwirth, T.: Theory and practice of constraint handling rules. J. Logic Programming, 19, 20 (1994), http://citeseer.ist.psu.edu/641466.html
7. Knuth, D.: The Art of Computer Programming, draft 4 (2006), http://www.cs.utsa.edu/~wagner/knuth/
8. Mishchenko, A., Brayton, R.: A boolean paradigm for multivalued logic synthesis. In: Proc. IgVLS'02, June, 2002, pp. 173–177 (2002), http://citeseer.ist.psu.edu/article/mishchenko02boolean.html
9. Mishchenko, A., Sasao, T.: Encoding of Boolean functions and its application to LUT cascade synthesis. In: International Workshop on Logic Synthesis (2002), http://citeseer.ist.psu.edu/mishchenko02encoding.html
10. O'Donnell, J.: Hardware description with recursion equations. In: Proceedings of the IFIP 8th International Symposium on Computer Hardware Description Languages and their Applications, NorthHolland, pp. 363–382 (April 1987)
11. Rabaey, J., Chandrakasan, A., Nikolic, B.: Digital Integrated Circuits: A Design Perspective. Prentice-Hall, Englewood Cliffs (2003)
12. Reintjes, P.: Elegant technologies (April 1992), published electronically at http://z.zhurnal.net/ElegantTechnologies.pdf

13. Riedel, M.: Cyclic Combinational Circuits. In: Ph.D. Dissertation, Caltech (2004), http://citeseer.ist.psu.edu/riedel04cyclic.html
14. Shende, V., Prasad, A., Markov, I., Hayes, J.: Synthesis of reversible logic circuits. In: IEEE Trans. on CAD 22, pp. 710–722 (June 2003), http://citeseer.ist.psu.edu/article/shende03synthesis.html
15. Sloane, N.J.A.: A000108, The On-Line Encyclopedia of Integer Sequences (2006), published electronically at http://www.research.att.com/~njas/sequences
16. Berkeley Logic Synthesis and Verification Group. ABC: A System for Sequential Synthesis and Verification, Release 61218 (2006), http://www.eecs.berkeley.edu/~alanmi/abc/

Spatial-Yap: A Logic-Based Geographic Information System

David Vaz, Michel Ferreira, and Ricardo Lopes

DCC-FC & LIACC
University of Porto, Portugal
{davidvaz,michel,rslopes}@ncc.up.pt

Abstract. Coupled deductive database systems join together logic programming systems and relational database management systems, in order to combine the best of both worlds. The current state-of-the-art of these interfaces is restricted to access extensional data in databases in Datalog form, disallowing access to compound terms. However, recent years have seen the evolution of relational database management systems in order to enable them to store and manage more complex information. Of this complex data, one of the most interesting and fast growing is that of *spatial data*. In this paper we describe the application of the MYDDAS deductive database system to the handling of spatial data, and the needed extensions, namely the ability to handle vectorial geometric attributes from database relations, the definition of spatial operators, and a visualization framework, in order to obtain a spatial deductive database system, that can be used as a geographic information system. We argue that such a system can improve the state-of-the-art of spatial data handling in all of its aspects, namely in spatial data modeling, spatial querying and spatial data mining. We describe, in particular, the application of such a logic-powered geographic information system to two real-world problems.

1 Introduction

Logic programming and relational databases have common foundations based on first-order logic. Despite these common foundations, logic programming (LP) systems and relational database management systems (RDBMS) have very different objectives. While the former focus on efficient implementation of core programming constructs, with emphasis on control and unification, the later focus on efficient implementation of data manipulation, with emphasis on indexing structures and massive storage. Examples of this state-of-the-art include the undeveloped support for persistence in most LP systems [1], and the undeveloped support for recursion in most RDBMS.

A current trend for augmenting LP systems with the ability of efficiently managing large amounts of data, is to couple such systems with RDBMS, developing coupled deductive database systems, such as XSB [2] and MYDDAS [3], instead of integrated deductive database systems, such as the Aditi system [4]. As the

V. Dahl and I. Niemelä (Eds.): ICLP 2007, LNCS 4670, pp. 195–208, 2007.

frontier between data and code in LP languages, such as Prolog, is thin, coupled deductive databases systems extend not only the ability of logic systems to deal with large amounts of data, but also the ability of executing larger programs, using RDBMS as abstract machines for the execution of logic code. A good example is the trend of relational inductive logic programming [5], where efficient and scalable results have been reported using coupled deductive database systems for the coverage computation of Prolog *clauses* [6].

Although there is no apparent reason for not using RDBMS for managing the internal database of Prolog systems, most coupled systems allow only the storage of datalog facts (callable terms) within the database system, instead of that of generic terms. Database relations are associated to predicate names and relational tuples are seen as Prolog facts. In these coupled systems the interface between the LP and the database systems is normally done by translating the query written in logic to the language understood by the RDBMS, SQL. A fundamental work on the translation of Prolog to SQL is the compiler written by Draxler [7]. The current state-of-the-art of these interfaces is restricted to access extensional data in databases in Datalog form, disallowing access to compound terms. However, recent years have seen the evolution of RDBMS in order to enable them to store and manage more complex information, leading to the development of what is referred as object-relational database systems. Of this complex data, one of the most interesting and fast growing is that of spatial data. In this paper we describe the application of the deductive database framework to the problem of spatial data handling, in all of its aspects, namely in spatial data modeling, spatial query languages and spatial data-mining. We describe the work developed in the context of the MYDDAS deductive database system and its use with two real-world applications. In MYDDAS, Datalog is enriched with *spatial terms*, that can be imported from vectorial maps stored in databases. The term *Spatial Datalog* as been used in the area of Constraint Databases [8], where the relational calculus has been extended with polynomial inequalities, providing an elegant and powerful model of spatial databases [9]. Datalog has been introduced in this context in order to query topological connectivity [10]. In this paper we describe a different use of Datalog for spatial databases, based on spatial terms rather than on polynomial inequalities, that will be allowed to occur in Datalog queries.

This paper describes the application of a deductive database system, coupling YapTab [11] and MySQL extended with geometry types, as a Geographic Information System (GIS). In particular, we describe the application of such a logic-powered GIS to two real-world problems: study of traffic behavior in the city of Porto [12]; and monitoring of biodiversity change. The paper is organized as follows: the next section briefly introduces the area of spatial databases and spatial objects; Section 3 describes the Spatial-Yap system, focusing on its three components of spatial terms support, visualization and spatial predicates; Section 4 presents the two real-world problems where we are currently applying Spatial-Yap; and Section 5 presents some conclusions and outlines future work.

2 Spatial Databases

Over the past years, the use of spatial data has increased in many areas of computer science. Spatial databases have been an active area of research for the last two decades, with a very wide range of areas of application, from Global Position Systems (GPS) to medical applications, where the human body is represented using the same spatial objects used to represent geographic data. Spatial data made clear that RDBMS needed to be updated to include spatially referenced data and, for the last few years, RDBMS are being extended to support spatial data, as well as to provide manipulation functions for that data.

The Open Geospatial Consortium (OGC), which is a non-profit, international, voluntary consensus standards organization that is leading the development of standards for geospatial and location based services, has proposed a standard to extend SQL-92 in "OpenGis Simple Features Specification for SQL" (OGC99) [13]. The purpose of this specification is to define a standard SQL schema that supports storage, retrieval, query and update of simple geospatial feature collections.

Spatial types, or geometry types as they are presented in OGC99 standard, are defined in an object oriented scheme described in Figure 1. There are three simple types, *Point*, *Linestring* and *Polygon*, and the *Geometry Collection* type. The latter expands to subclasses that specialize the member elements types. Each type has a set of well defined attributes and restrictions, that can be consulted in the OGC99 document. We list examples of the attributes and restrictions for the following simple types:

Point is a 0-dimensional geometry type that represents a single location in the coordinate system and has two coordinates (x, y);

Linestring is one-dimensional geometry type stored as a sequence of points with linear interpolation;

Polygon is a two-dimensional spatial object that defines a planar surface. It is defined by one exterior ring and zero or more interior rings, defining holes in the polygon.

Spatial database systems should integrate the representation and manipulation of geometric information with traditional data at the logical level, while providing an efficient support at the physical level to store and process this information. This is usually achieved through an integrated approach, which is based on RDBMS extensibility. Examples of such spatial database systems are the Oracle Spatial [14] and PostGIS [15].

3 A Spatial Deductive Database System

Spatial Yap results from a sophisticated interface between several components. The two main components are the Yap Prolog system and MySQL RDBMS, which are coupled through the MYDDAS interface [3] (Mysql/Yap Deductive DAtabase System). This interface is responsible for coupling these two systems,

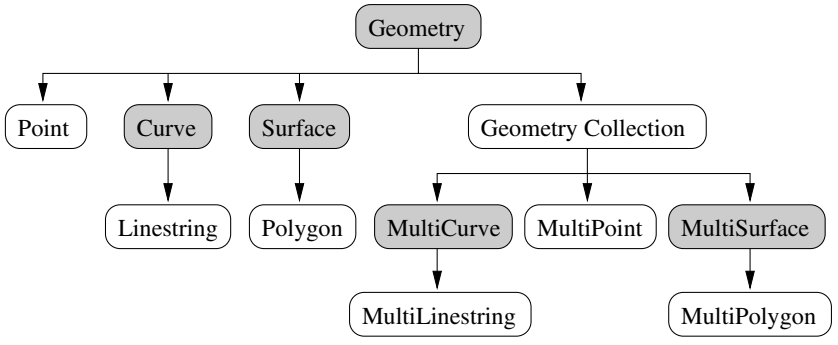


Fig. 1. Geometry-types (abstract types in gray)

as illustrated in Figure 2. MYDDAS transparently translates logic queries into SQL statements, implements the conversion into Yap terms of MySQL attributes and explores the YapTab tabling engine for solving recursive queries involving database goals. The level of sophistication of this interface is very high, with the fetching of relational tuples being implemented directly in WAM choice-points, supporting pruning operators [16].

To build the spatial deductive database system we extended the MYDDAS interface to support MySQL Geometry Types. Geometry objects are stored in MySQL relations as binary sequences, following a structure similar to the Well-Known Binary (WKB) format, which is well documented in the OGC99 specifications.

Two more components are fundamental to build Spatial Yap: a spatial operators library and a visualization component. We will next describe the extensions of MYDDAS to support spatial terms, the visualization component and the spatial predicates component.

3.1 Spatial Terms

We have extended the MYDDAS interface to support spatial terms, which are stored in MySQL relations as binary sequences. They can be retrieved using OGC standard *asBinary* operator, but as MySQL stores spatial types using as base the WKB adding extra information, we retrieve spatial types directly ignoring that extra information.

Spatial terms were defined similarly to the *Well Known Text* of OGC, and conform to the grammar in Figure 3. A valid spatial term has to conform to the additional restrictions documented in OGC99 Specifications, such as:

- A *Linestring* has at least two *Points*;
- A *Polygon* is defined by linear rings, being a linear ring a simple and closed *Linestring*. A *Polygon* has one exterior ring, and zero or more interior rings. The exterior ring is always the first in the list;
- The multi-types have one or more elements of the same type;
- An empty *GeometryCollection* represents the empty set.

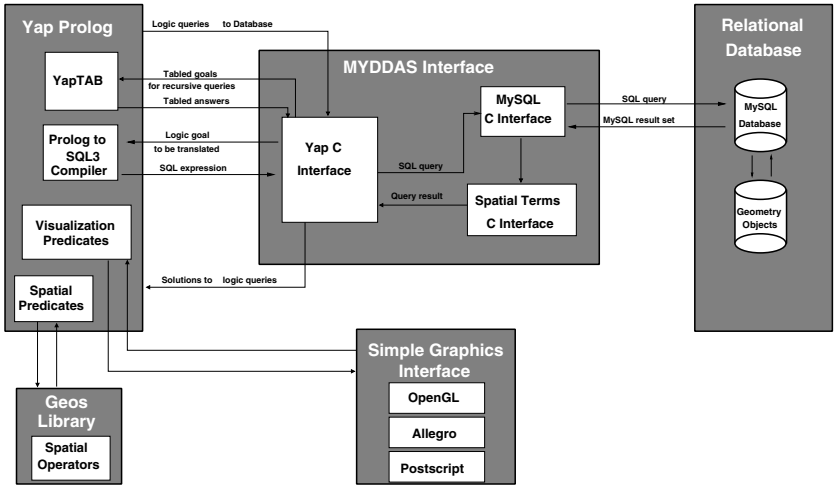


Fig. 2. Spatial Yap Blueprint

```
SpatialTerm = Point
            | LineString
            | Polygon
            | MultiPoint
            | MultiLineString
            | MultiPolygon
            | GeometryCollection ;
Point = "point" PointTerm ;
LineString = "linestring(" PointTermList ")" ;
Polygon = "polygon(" PointTermListList ")" ;
MultiPoint = "multipoint(" PointTermList ")" ;
MultiLinestring = "multilinestring(" PointTermListList ")" ;
MultiPolygon = "multipolygon(" PointTermListListList ")" ;
GeometryCollection = "geometrycollection(" SpatialTermList ")" ;
PointTerm = "(" Number "," Number ")" ;
```

Fig. 3. EBNF of Spatial Terms

In Spatial-Yap spatial terms can be created by querying the database tables and its geometry attributes, or created textually by the programmer, or result from the application of spatial predicates. Spatial-Yap can also store persistently the spatial terms in MySQL database relations.

3.2 Visualization of Spatial Data with Simple Graphics

When dealing with spatial data it is essential to be able to graphically represent such data. Representing a map as a set of Prolog terms is visually unacceptable, from the point of view of a GIS user. Even more important is the representation

of a spatial operation, such as the intersection of two polygons, in a graphical way. User-driven spatial analysis and the representation of spatial queries result sets require a visualization component to be added to any spatial database system. There are some Prolog systems providing native Graphical User Interfaces (GUI), such as SWI-Prolog with the XPCE [17], and systems providing coupling interfaces to external GUIs libraries such as TCL/TK. Our needs were slightly different as we do not need to have a complete GUI, but simply a basic drawing system conforming to the OGC99 spatial types. This led to the development of a graphical interface, Simple Graphics (SG), in the context of Spatial-Yap.

SG receives spatial terms and draws them in an interactive window where the user can zoom, pan and take screenshots. Alternatively, the visualization primitives can create the image and export it to a Postscript file. The core system uses a stack of geometric objects to cope with Prolog backtracking. However, the drawing sees this stack as a first-in-first-out structure, so that a later object is drawn after a previous one. For example, to build a window showing the Europe countries based on a MySQL relation `europe` with the following schema: `name varchar(30), pop bigint, currency varchar(30), geom geometry;` we write the following code:

```
:- use_module(library(myddas)).
:- use_module(library(simplegraphics)).

:- db_open(mysql,'localhost'/'geodb','user','passwd').
:- db_import(europe,europe).

draw_base_map :-
        findall(C,europe(_Name,_Pop,_Currency,G),Countries),
        sg_color(150,150,150), sg_outline_color(50,50,50),
        sg_create(geometrycollection(Countries)).

:-      sg_initialize([system(opengl), size(800,500),
                       background(200,200,200)]),
        draw_base_map, sg_recenter.
```

Sometimes we need to remove objects previously drawn. This is done through backtracking: the first call pushes the geometry object into the stack; the second call pops the object from the stack, failing. Using our previous example and assuming we add the following predicate:

```
country(Name) :-
        europe(Name,_Pop,_Currency,Geom),
        sg_color(100,100,100),
        sg_create(Geom).
```

the goal `'?- country(X).'` can backtrack on the alternative solutions for `X` while highlighting its geometric object on the opened SG window, as shown in Figure 4.

Table 1 summarizes some of the available SG predicates. SG is built in a very modular way to allow several export formats. For now it supports OpenGL [18] and Allegro [19] as interactive alternatives, and Postscript as image creation alternative. In the future we will expand the non interactive alternatives to create other image formats natively.

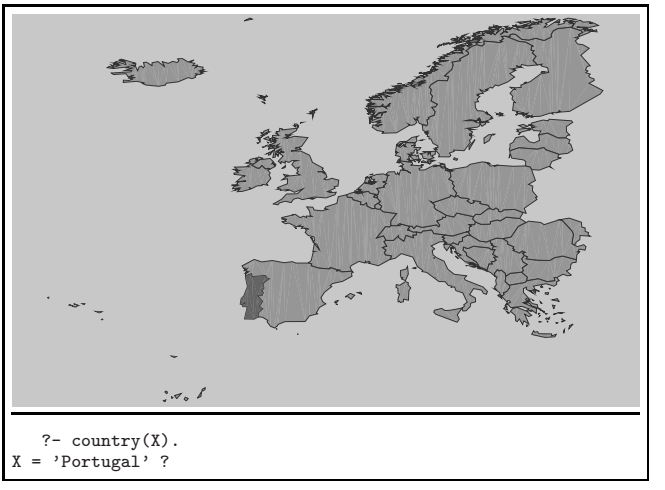


Fig. 4. Europe Map

Table 1. Simple Graphics Interface

Type	Predicate
SG module handling	`sg_initialize(+Options)`
	`sg_shutdown`
	`sg_recenter`
	`sg_export(ps,+File)`
Drawing Spatial Terms	`sg_create(+SpatialTerm)`
	`sg_color(+Red,+Green,+Blue)`
	`sg_color(+Color)`
	`sg_outline_color(+Red,+Green,+Blue)`
	`sg_outline_color(+Color)`
Yap and SG interaction	`sg_pause(+N)`
	`sg_yield`
	`sg_repeat`

3.3 Spatial Predicates

The third key component of Spatial-Yap is the spatial operators module, which we name *ogc*, as it follows the OGC standard, providing a number of spatial predicates for the manipulation of spatial data. Table 2 summarizes these predicates. They are divided in three main groups:

- Predicates for testing spatial properties. Including spatial relations between spatial terms;
- Predicates that support spatial analysis. These are predicates that construct new spatial terms based on operations on the input terms, it also includes metric predicates;
- Predicates on specific spatial terms. These are specific to each type.

Table 2. Spatial Predicates

Type	Predicate	
Predicates for testing spatial properties	`ogc_is_empty(+Geom)` `ogc_is_simple(+Geom)`	

	`ogc_equals(+Geom1,+Geom2)`
	`ogc_disjoint(+Geom1,+Geom2)`
	`ogc_touches(+Geom1,+Geom2)`
	`ogc_within(+Geom1,+Geom2)`
	`ogc_overlaps(+Geom1,+Geom2)`
	`ogc_crosses(+Geom1,+Geom2)`
	`ogc_intersects(+Geom1,+Geom2)`
	`ogc_contains(+Geom1,+Geom2)`
	`ogc_relate(+Geom1,+Geom2,?PatternMatrix)`	

Predicates that support spatial analysis	`ogc_envelope(+Geom,?GeomEnvelope)`
	`ogc_boundary(+Geom,?GeomBoundary)`
	`ogc_buffer(+Geom,+Distance,?GeomBuffer)`
	`ogc_convex_hull(+Geom,?GeomConvexHull)`
	`ogc_intersection(+Geom1,+Geom2,?GeomIntersection)`
	`ogc_union(+Geom1,+Geom,?GeomUnion)`
	`ogc_difference(+Geom1,+Geom,?GeomDifference)`
	`ogc_symmetric_difference(+Geom1,+Geom2,?GeomSymDiff)`
	`ogc_distance(+Geom1,+Geom2,?Distance)`	

Specific type predicates	Linestring Multilinestring	`ogc_length(+Geom,?Length)`
		`ogc_is_closed(+Geom)`
		`ogc_is_ring(+Geom)`

	Polygon Multipolygon	`ogc_area(+Geom,?Area)`
		`ogc_centroid(+Geom,?GeomPoint)`
		`ogc_point_on_surface(+Geom,?GeomPoint)`

These predicates are accessible to Prolog via the C API using the GEOS [20] library, which is also used by the RDBMS PostgreSQL [21] in the PostGIS [15] module (spatial database extension for PostgreSQL).

As an example of the use of the three components of Spatial Yap, suppose we have a MySQL table storing the Europe map, as in the previous example (see Figure 4) and that we want to find the centroid of the `multipolygon` formed by the polygons defining the boundaries of the countries of Europe, and see if it belongs to any of the countries in Europe. We would extend the code of the previous example with the code shown in Figure 5. Note the use of the spatial predicates `ogc_within/2` and `ogc_centroid/2`. Note also the generation of the image as a Postscript file, through predicate `sg_export/2` (all the Postscript images in this paper representing maps have been generated with Spatial-Yap).

The spatial predicates provided by the *ogc* module supply the infrastructure of another key component of a spatial database system, which is its *data-mining* module. In the context of LP, data-mining can be done using *induction*, based on the paradigm of Inductive Logic Programming (ILP). Data-mining using ILP offers the substantial advantage of being *descriptive*, as compared to other

Code to generate this image

```
...

find_country(Point,Country,Polygon) :-
        europe(Name,_,_,Polygon),
        ogc_within(Point,Polygon), !.
find_country(Point,none,Point).

find_europe_centroid(Point,Country,Polygon) :-
        findall(P,europe(_,_,_,polygon(P)),Countries),
        ogc_centroid(multipolygon(Countries),Point),
        find_country(Point,Country,Polygon).

gen_ps(PS) :-
        sg_initialize([system(nowindow),
                       background(200,200,200),
                       size(800,600)]),
        draw_base_map, sg_recenter,
        find_europe_centroid(Point,_,CountryPolygon),
        sg_color(100,100,100), sg_create(CountryPolygon),
        sg_color(black), sg_create(Point),
        sg_export(ps,PS),
        sg_shutdown.
```

Fig. 5. Europe countries centroid

machine learning techniques, deriving a logic theory which explains a number of observations. Spatial-Yap coupled with an ILP system, such as APRIL [22], can use the spatial predicates of the *ogc* module to build theories which explain observations based on large geographic data-sets stored in MySQL spatial databases. A classic example of spatial data-mining and the theories an ILP system should derive, is provided by [23]:

"In 1855 when the Asiatic cholera was sweeping through London, an epidemiologist marked all locations on a map where the disease had struck and discovered that the locations formed a cluster whose centroid turned out to be a water pump. When the authorities turned off the pump, the cholera began to subside."

An ILP system, with a geographic background knowledge which includes the locations of where the disease had struck and the locations of water pumps, together with the *ogc* module spatial predicates, such as `ogc_centroid/2`, would be able to induct the same spatial correlation.

4 Real-World Applications

In this section we describe two undergoing projects where we are applying Spatial-Yap.

4.1 Study of Traffic Behavior in the City of Porto

The aim of this project is to study traffic behavior in the city of Porto. We are interested in understanding factors affecting traffic, not only time and day related, but also including intrinsic geographic entities, such as the presence of a school in a street segment and its influence in traffic congestion time-slots. More ambitious goals include the automatic derivation of a road signalization layer, including traffic lights and stop signs locations, based on mobility patterns.

We are using a large spatial database storing the streets of the city of Porto, granted by the Porto City Council. This map covers an area of 62 square kilometers, with 1941 streets summing up to 965 kilometers of extension. We have been collecting GPS logs in Porto to create a data warehouse with the mobility information. We use Spatial-Yap to manipulate all the data. For instance, we can query the area covered by the GPS logs of a given user and create the respective image, such as in Figure 6.

Routing algorithms and displaying of computed routes are also implemented using Spatial-Yap and its tabling engine. We also use LP to map GPS coordinates to street segments. GPS devices in urban-like environments can report positions

Fig. 6. User Coverage

which show an error of more than 30 meters in relation to the actual position. This causes uncertainty as to which road segment the log is actually traversing. This is represented as choice-points which are pruned by a Prolog cut, as soon as the topological layer allows discarding those alternatives.

4.2 Monitoring of Biodiversity Change

Another interesting project where we are using Spatial-Yap aims at the global monitoring of biodiversity change [24]. Governments have set the ambitious target of reducing biodiversity loss by the year 2010, and scientists now face the challenge of accessing the progress made towards this target. The European Corine Land Cover (CLC) project (`http://terrestrial.eionet.eu.int/CLC2000`) provides data for two different years (1990 and 2000), using 44 land-cover classes. This data is obtained from satellite images and the vectorization in the polygons of each of the 44 land-cover classes is done automatically, based on color recognition. Unfortunately, the land-cover classes of CLC are not the most

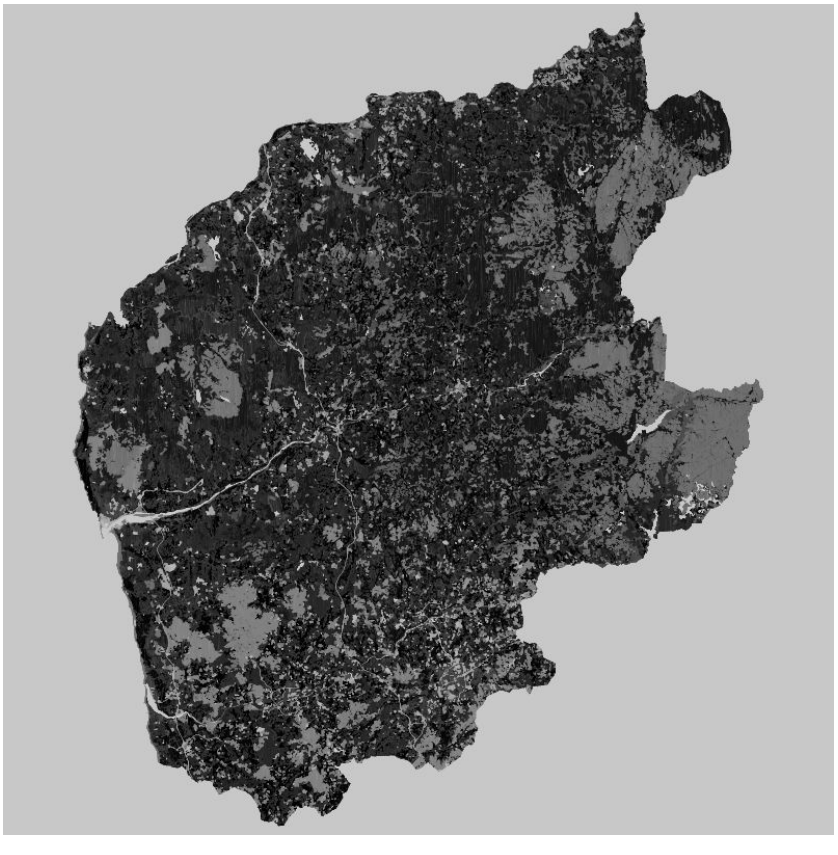


Fig. 7. Alto Minho Map

appropriate to monitor biodiversity. For instance, currently CLC has only three classes for forest (broad-leaved, coniferous and mixed); therefore, an observed increase in broadleaved forest area could be due to an increase in plantation area of an exotic species, such as Eucalyptus globulus, or an increase in native broad-leaved forest, two phenomena with different implications for biodiversity. The group of biologists with whom we are working has detailed regional maps from the area of Alto Minho, in Portugal, also covered by CLC. These regional maps are done based on expensive and slow on-site mapping techniques and on-site identification of forest species, allowing a much higher detail on the list of classes. Our project is trying to use these detailed regional maps to derive a set of spatial logic rules that allow the detailed characterization of CLC data for biodiversity monitoring. We are using Spatial-Yap and the APRIL ILP system over data-sets created based on the intersection of the regional maps and CLC maps. The inducted rules can then be used to improve the categorization of new CLC data, allowing its use for biodiversity monitoring.

The regional map of Alto Minho we are using with Spatial-Yap has a total of 43028 polygons, stored in a MySQL relation of 62MBytes. We used Spatial-Yap to generate the Postscript image for this map, shown in Figure 7.

5 Conclusions and Future Work

In this paper we have described the use of Spatial-Yap as a logic-based geographic information system, describing what we believe to be a very useful application of the deductive database framework. We believe that logical databases have their features and capabilities enhanced when the data they deal with increases in complexity. It is clear that SQL, the *lingua franca* of database systems, is inadequate for handling complex and structured data, such as geometric data. Representing such data as logic terms and embedding spatial querying and spatial data manipulation within LP systems makes spatial data modeling and handling simple and natural. Furthermore, it allows spatial data-mining to be done through descriptive machine learning techniques, such as inductive logic programming.

This paper presented Spatial-Yap with emphasis on its features, on how spatial data is handled by a logic system and how it is visualized. We omitted implementation details and performance discussion. Clearly Spatial-Yap is being used with spatial databases of several MBytes, but, in the future, efficiency will become an issue, and indexing of spatial terms will, in particular, have to be addressed.

A lot of work remains to be done in order for Spatial-Yap to be widely used as a geographic information system. A web interface such as ka-map [25] and support for basic map objects, such as graphic scales, will have to be added. But Spatial-Yap presents over other geographic information systems the advantage of its high-level, declarative and logic-based modeling of data, based on spatial terms, which makes the system very attractive for users of very different areas of knowledge, such as the biologists with whom we are working, physicians, for

medical spatial data and many other users from many other areas where efficient modeling, querying and mining of spatial data is becoming crucial.

Acknowledgments

This work has been partially supported by MYDDAS (POSC/EIA/59154/2004) and by funds granted to LIACC through the Programa de Financiamento Plurianual, Fundação para a Ciência e Tecnologia and Programa POSC. David Vaz is funded by FCT PhD grant SFRH/BD/29648/2006.

References

1. Correas, J., Gomez, J., Carro, M., Cabeza, D., Hermenegildo, M.: A Generic Persistence Model for (C)LP Systems (and Two Useful Implementations). In: Jayaraman, B. (ed.) PADL 2004. LNCS, vol. 3057, pp. 104–119. Springer, Heidelberg (2004)
2. Sagonas, K., Swift, T., Warren, D.S.: XSB as an Efficient Deductive Database Engine. In: ACM SIGMOD International Conference on the Management of Data, pp. 442–453. ACM Press, New York (1994)
3. Soares, T., Ferreira, M., Rocha, R.: The MYDDAS Programmer's Manual. Technical Report DCC-2005-10, Department of Computer Science, University of Porto (2005)
4. Vaghani, J., Ramamohanarao, K., Kemp, D., Somogyi, Z., Stuckey, P., Leask, T., Harland, J.: The Aditi Deductive Database System. Technical Report 93/10, School of Information Technology and Electrical Engineering, Univ. of Melbourne (1993)
5. Morik, K.: Knowledge Discovery in Databases - an Inductive Logic Programming Approach. In: Foundations of Computer Science: Potential - Theory - Cognition, pp. 429–436. Springer, Heidelberg (1997)
6. Ferreira, M., Fonseca, N.A., Rocha, R., Soares, T.: Efficient and Scalable Induction of Logic Programs using a Deductive Database System. In: Muggleton, S., Otero, R., Tamaddoni-Nezhad, A. (eds.) ILP 2006. LNCS (LNAI), vol. 4455, Springer, Heidelberg (2006)
7. Draxler, C.: Accessing Relational and Higher Databases Through Database Set Predicates. PhD thesis, Zurich University (1991)
8. Kanellakis, P.C., Kuper, G.M., Revesz, P.Z.: Constraint query languages. J. Comput. Syst. Sci. 51(1), 26–52 (1995)
9. Paredaens, J., den Bussche, J.V., Gucht, D.V.: Towards a theory of spatial database queries (extended abstract). In: PODS '94: Proceedings of the thirteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems, Minneapolis, Minnesota, United States, pp. 279–288. ACM Press, New York (1994)
10. Kuijpers, B., Paredaens, J., Smits, M., den Bussche, J.V.: Termination properties of spatial datalog programs. In: Pedreschi, D., Zaniolo, C. (eds.) LID 1996. LNCS, vol. 1154, pp. 101–116. Springer, Heidelberg (1996)
11. Rocha, R., Silva, F., Santos Costa, V.: YapTab: A Tabling Engine Designed to Support Parallelism. In: Conference on Tabulation in Parsing and Deduction, pp. 77–87 (2000)
12. Ferreira, M.: The MYDDAS Project: Using a Deductive Database for Traffic Characterization. In: Gabbrielli, M., Gupta, G. (eds.) ICLP 2005. LNCS, vol. 3668, pp. 424–426. Springer, Heidelberg (2005)

13. Open GIS Consortium, I.: OpenGIS Simple Features Specifications For SQL (1999), Available from http://www.opengis.org/docs/99-049.pdf
14. Ravada, S., Sharma, J.: Oracle8i spatial: Experiences with extensible databases. In: Güting, R.H., Papadias, D., Lochovsky, F.H. (eds.) SSD 1999. LNCS, vol. 1651, pp. 355–359. Springer, Heidelberg (1999)
15. The Postgis Development Team: (Postgis adds support for geographic objects to the postgresql object-relational database.), Available from http://postgis.refractions.net/
16. Soares, T., Rocha, R., Ferreira, M.: Generic Cut Actions for External Prolog Predicates. In: Van Hentenryck, P. (ed.) PADL 2006. LNCS, vol. 3819, pp. 16–30. Springer, Heidelberg (2005)
17. Anjewierden, A., Wielemaker, J.: (Xpce: the swi-prolog native gui library), Available from http://www.swi-prolog.org/packages/xpce/
18. Shreiner, D.: OpenGL(R) 1.4 Reference Manual, 4th edn. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA (2004)
19. Hargreaves, S.: (Allegro: A game programming library), Available from http://alleg.sourceforge.net/
20. The GEOS Development Team: (GEOS: Geometry Engine Open Source), Available from http://geos.refractions.net/
21. The PostgreSQL Development Team: Postgresql user's guide (2006), Available from http://www.postgresql.org/docs/
22. Fonseca, N.A., Silva, F., Camacho, R.: April - An Inductive Logic Programming System. In: Fisher, M., van der Hoek, W., Konev, B., Lisitsa, A. (eds.) JELIA 2006. LNCS (LNAI), vol. 4160, pp. 481–484. Springer, Heidelberg (2006)
23. Griffith, D.A.: Statistical and mathematical sources of regional science theory: Map pattern analysis as an example. Papers in Regional Science 78(1), 21–45 (1999)
24. Pereira, H.M., Cooper, H.D.: Towards the global monitoring of biodiversity change. Trends in Ecology & Evolution 21(3), 123–129 (2006)
25. The ka-Map Development Team: (ka-map), Available from http://ka-map.maptools.org/

The Correspondence Between the Logical Algorithms Language and CHR*

Leslie De Koninck**, Tom Schrijvers***, and Bart Demoen

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

Abstract. This paper investigates the relationship between the Logical Algorithms language (LA) of Ganzinger and McAllester and Constraint Handling Rules (CHR). We present a translation scheme from LA to CHR^{rp}: CHR with rule priorities and show that the meta-complexity theorem for LA can be applied to a subset of CHR^{rp} via inverse translation. This result is compared with previous work. Inspired by the high-level implementation proposal of Ganzinger and McAllester, we demonstrate how LA programs can be compiled into CHR rules that interact with a scheduler written in CHR. This forms the first actual implementation of LA. Our implementation achieves the complexity required for the meta-complexity theorem to hold and can execute a subset of CHR^{rp} with strong complexity bounds.

1 Introduction

Constraint Handling Rules (CHR) [5] is a high-level rule based language, originally designed for the implementation of constraint solvers, but also more and more used as a general purpose programming language. Recently, it was shown that all algorithms can be implemented in CHR while preserving both time and space complexity [16]. We assume familiarity with CHR (see [5,13]).

In "Logical Algorithms" (LA) [9] (and based on previous work in [8,12]), Ganzinger and McAllester present a bottom-up logic programming language for the purpose of facilitating the derivation of complexity results of algorithms described by logical inference rules. This language resembles CHR in many ways and has often been referred to in the discussion of complexity results of CHR programs [1,6,15,17]. The aim of this paper is to investigate the relationship between both languages. More precisely, we look at how the meta-complexity theorem for LA can be applied to (a subset of) CHR, and how CHR can be used to implement LA.

First, we present a translation from LA to CHR^{rp}: CHR with rule priorities [10]. LA derivations of the original program correspond to CHR^{rp} derivations in

* A longer version of this paper is available as Technical Report [11].

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

*** Post-Doctoral Researcher of the Fund for Scientific Research - Flanders (Belgium) (F.W.O. - Vlaanderen).

V. Dahl and I. Niemelä (Eds.): ICLP 2007, LNCS 4670, pp. 209–223, 2007.

the translation and vice versa. We show how to translate a subclass of CHR^{rp} to LA. This allows the meta-complexity theorem for LA to be applied to these CHR^{rp} programs as well. Because the LA meta-complexity theorem is based on an optimized implementation, it gives more accurate results than the implementation independent meta-complexity theorem of [7,6] while being more general than the ad-hoc complexity derivations in [15,17].

Our current prototype implementation of CHR^{rp} does not achieve the complexity required for the meta-complexity theorem to hold. Therefore, we propose an implementation of LA in (regular) CHR, which consists of the compilation of LA programs to CHR rules, combined with a scheduler written in CHR. By using a CHR implementation with advanced indexing support, such as the K.U.Leuven CHR system [14], our implementation achieves the required complexity. It is the first actual implementation of LA[1] and also a first implementation of a subset of CHR^{rp} with strong complexity bounds.

The rest of this paper is organized as follows. In Section 2, the syntax and semantics of the Logical Algorithms language and CHR^{rp} are reviewed. In Section 3 a translation of LA programs to CHR^{rp} programs is presented and in Section 4, the opposite is done for a subset of CHR^{rp}. The implementation of Logical Algorithms in refined operational semantics based CHR is given in Section 5. Section 6 shows that it has the required complexity. We conclude in Section 7.

2 Logical Algorithms and CHR^{rp}

In this section, we give an overview of the syntax and semantics of the Logical Algorithms language and CHR^{rp}.

2.1 Logical Algorithms

A Logical Algorithms program $P = \{r_1, \ldots, r_n\}$ is a set of rules. In [9], a graphical notation is used to represent rules. We use a textual representation that is closer to the syntax of CHR. A Logical Algorithms rule is an expression

$$r \ @ \ p : A_1, \ldots, A_n \Rightarrow C$$

where r is the rule *name*, the atoms A_i (for $1 \leq i \leq n$) are the *antecedents* and C is the *conclusion*, which is a conjunction of atoms whose variables appear in the antecedents. Rule r has *priority* p where p is an arithmetic expression whose variables (if any) occur in the first antecedent A_1. If p contains variables, then r is called a dynamic priority rule. Otherwise, it is called a static priority rule.

The arguments of an atom are either Herbrand terms or (integer) arithmetic expressions. There are two types of atoms: comparisons and user-defined atoms. A comparison has the form $x < y$, $x \leq y$, $x = y$ or $x \neq y$ with x and y arithmetic expressions or, in case of $(=)/2$ and $(\neq)/2$, Herbrand terms. Comparisons are

[1] To the best of our knowledge and confirmed in personal communication with "Logical Algorithms" author David McAllester.

only allowed in the antecedents of a rule and all variables in a comparison must appear in earlier antecedents. A user-defined atom can be positive or negative. A negative user-defined atom has the form $del(A)$ where A is a positive user-defined atom. A ground user-defined atom is called an assertion.

Example 1. An example rule (from Dijkstra's shortest path algorithm as presented in [9]) with name `d2` and priority 1 is

```
d2 @ 1 : dist(V,D1), dist(V,D2), D2 < D1 => del(dist(V,D1)).
```

The antecedent `D2 < D1` is a comparison, the atoms `dist(V,D1)` and `dist(V,D2)` are positive user-defined antecedents. The negative ground atom `del(dist(a,5))` is an example of a negative assertion.

A Logical Algorithms state σ consists of a set of (positive and negative) assertions. Let $\mathcal{D}$ be the usual interpretation for the comparisons. Given a program P, the following transition converts one state into the next:

1. Apply $\sigma \overset{LA}{\mapsto}_P \sigma \cup \theta(C)$ if there exists a (renamed apart) rule r in P of priority p of the form

$$r \ @ \ p : A_1, \ldots, A_n \Rightarrow C$$

and a ground substitution θ such that for every antecedent A_i,
- $\mathcal{D} \models \theta(A_i)$ if A_i is a comparison
- $\theta(A_i) \in \sigma$ and $del(\theta(A_i)) \notin \sigma$ if A_i is a positive user-defined atom
- $\theta(A_i) \in \sigma$ if A_i is a negative user-defined atom

Furthermore, $\theta(C) \nsubseteq \sigma$ and no rule of priority p' and substitution θ' exists with $\theta'(p') < \theta(p)$ for which the above conditions hold.

A state is called final if no more transitions apply to it. A non-final state has priority p if the next firing rule instance has priority p. The condition $\theta(C) \nsubseteq \sigma$ ensures that no rule instance fires more than once and prevents trivial non-termination. Although the priorities restrict the possible derivations, the choice of which rule instance to fire from those with equal priority is non-deterministic.

A prefix instance of rule $r \ @ \ p : A_1, \ldots, A_n \Rightarrow C$ is a tuple $\langle \theta(r), i \rangle$ with θ a ground substitution and $1 \leq i \leq n$. Its antecedents are $\theta(A_1), \ldots, \theta(A_i)$. The time complexity for running Logical Algorithms programs is given in [9] as $\mathcal{O}(|\sigma_0| + P_s + (P_d + A_d) \cdot \log N)$ where σ_0 is the initial state and $|\sigma_0|$ is its size. P_s is the number of *strong* prefix firings of static priority rules and P_d is the number of *strong* prefix firings of dynamic priority rules. A strong prefix firing is a prefix instance for which all antecedents hold in a state with priority lower or equal to the prefix' rule priority. A_d is the number of assertions that may participate in a dynamic priority rule instance. Finally, N is the number of distinct priorities.

2.2 CHR$^{\mathrm{rp}}$: CHR with Rule Priorities

CHR$^{\mathrm{rp}}$ is CHR extended with user-definable rule priorities. It is introduced in [10] as a solution to the lack of high-level execution control in CHR. In CHR$^{\mathrm{rp}}$,

every rule is annotated with a rule priority that may depend on the arguments of the constraints in the rule heads, and a rule instance is only allowed to fire if no higher priority rule instance can. The operational semantics of CHR$^{\text{rp}}$, denoted by ω_p, is described as a state transition system.

As in ω_t, the theoretical operational semantics for CHR [2], we represent a state σ as a tuple $\langle G, S, B, T\rangle_n$, where G is the goal, a multiset of constraints; S is a set of identified CHR constraints, B is a conjunction of built-in constraints, T is the propagation history and n the next free identifier. The propagation history has a similar function as the $\theta(C) \notin \sigma$ condition in the LA semantics, but is less restrictive. The transitions of the ω_p semantics are shown below[2].

1. Solve $\langle \{c\} \uplus G, S, B, T\rangle_n \overset{\omega_p}{\mapsto}_P \langle G, S, c \wedge B, T\rangle_n$ where c is a built-in constraint.

2. Introduce $\langle \{c\} \uplus G, S, B, T\rangle_n \overset{\omega_p}{\mapsto}_P \langle G, \{c\#n\} \cup S, B, T\rangle_{n+1}$ where c is a CHR constraint.

3. Apply $\langle \emptyset, H_1 \cup H_2 \cup S, B, T\rangle_n \overset{\omega_p}{\mapsto}_P \langle \theta(C), \theta(H_1 \cup S), \theta(B), T'\rangle_n$ where there exists a (renamed apart) rule in P of priority p of the form

$$r \ @ \ H_1' \backslash H_2' \iff g \mid C \text{ pragma priority}(p)$$

and a matching substitution θ such that $chr(H_1) = \theta(H_1')$, $chr(H_2) = \theta(H_2')$, $\mathcal{D} \models B \rightarrow \bar{\exists}_B(\theta \wedge g)$ and $t = id(H_1) +\!\!+ id(H_2) +\!\!+ [r] \notin T$. Furthermore, no rule of priority p' and substitution θ' exists with $\theta'(p') < \theta(p)$ for which the above conditions hold. $T' = T \cup \{t\}$.

The following theorem on the correspondence between the ω_p semantics of CHR$^{\text{rp}}$ and the ω_t semantics of CHR, is proven in [10].

Theorem 1. *Every derivation D under ω_p is also a derivation under ω_t. If a state σ is a final state under ω_p, then it is also a final state under ω_t.*

CHR$^{\text{rp}}$ differs from LA in the following ways. A Logical Algorithms state is a set of ground assertions. The CHR constraint store is a multi-set and may also contain non-ground constraints. In LA, built-in constraints are ask constraints and only include comparisons. CHR$^{\text{rp}}$ supports any kind of built-in constraints. A removed CHR constraint may be reasserted and can then participate again in rule firings whereas a removed LA assertion cannot be asserted again. Finally, a LA rule may contain negated heads. In contrast, CHR$^{\text{rp}}$ requires all heads to be positive[3].

In the refined operational semantics of CHR [2], the textual order of the program rules determines which rule is tried next for the current *active* constraint. However, only rule instances in which the active constraint takes part are considered, and so a higher priority fireable rule instance in which the active constraint

[2] We apply matching substitutions directly to the goal, CHR store and built-in store. If the rule bodies contain no built-in constraints, the built-in store remains empty.

[3] See [18] for an extension of CHR that allows negated heads.

does not participate, will not fire. The textual rule order also does not support dynamic rule priorities.

3 Translating Logical Algorithms to CHR^{rp}

In this section, we show how Logical Algorithms can be translated into CHR^{rp} programs. CHR states of the translated program can be mapped on LA states of the original. With respect to this mapping, both programs have the same derivations.

3.1 The Translation Schema

The translation of a LA program P is denoted by $T(P) = T_{S/D}(P) \cup T_R(P)$. The contents of $T_{S/D}(P)$ and $T_R(P)$ are given below.

Set and Deletion Semantics. We use an internal representation for assertions as CHR constraints consisting of the assertion itself and an extra argument, called the *mode indicator*, denoting whether it is positively asserted ("p"), negatively asserted ("n") or both ("b"). For every user-defined predicate a/n occurring in P, $T_{S/D}(P)$ contains the following rules:

$$\begin{aligned} a_r(\bar{X}, M) \setminus a(\bar{X}) &\iff M \neq \mathtt{n} \mid \mathit{true} \text{ pragma priority(1)} \\ a_r(\bar{X}, \mathtt{n}), a(\bar{X}) &\iff a_r(\bar{X}, \mathtt{b}) \text{ pragma priority(1)} \\ a(\bar{X}) &\iff a_r(\bar{X}, \mathtt{p}) \text{ pragma priority(2)} \end{aligned}$$

$$\begin{aligned} a_r(\bar{X}, M) \setminus \mathit{del}(a(\bar{X})) &\iff M \neq \mathtt{p} \mid \mathit{true} \text{ pragma priority(1)} \\ a_r(\bar{X}, \mathtt{p}), \mathit{del}(a(\bar{X})) &\iff a_r(\bar{X}, \mathtt{b}) \text{ pragma priority(1)} \\ \mathit{del}(a(\bar{X})) &\iff a_r(\bar{X}, \mathtt{n}) \text{ pragma priority(2)} \end{aligned}$$

If a representation already exists, one of the priority 1 rules updates this representation. Otherwise, one of the priority 2 rules generates a new representation. At lower priorities, it is guaranteed that every assertion, whether asserted positively, negatively or both, is represented by exactly one constraint in the store[4].

Rules. Given a LA rule $r \in P$ of the form

$$r \mathbin{@} p : A_1, \ldots, A_n \Rightarrow C$$

We first split up the antecedents into user-defined antecedents and comparison antecedents by using the *split* function defined below.

$$\mathit{split}([A|T]) = \begin{cases} \langle [A|A^u], A^c \rangle & \text{if } A \text{ is a user-defined atom} \\ \langle A^u, [A|A^c] \rangle & \text{if } A \text{ is a comparison} \end{cases}$$

$$\text{where } \mathit{split}(T) = \langle A^u, A^c \rangle$$

$$\mathit{split}([]) = \langle [], [] \rangle$$

[4] Under the ω_t semantics, this non-monotonic semantics cannot be ensured.

In the Logical Algorithms language, a given assertion may participate multiple times in the same rule instance, whereas in CHR all constraints in a single rule instance must be different. To overcome this semantic difference, a single LA rule is translated as a set of CHR rules such that every CHR rule covers a case of syntactically equal head constraints. Let $\langle A^u, A^c\rangle = \mathit{split}([A_1, \ldots, A_n])$ with $A^u = [A^u_1, \ldots, A^u_m]$ and $A^c = [A^c_1, \ldots, A^c_l]$. Let $\mathcal{P}$ be the set of all partitions of $\{1, \ldots, m\}$.[5] For a given partition $\rho \in \mathcal{P}$, the following function returns the most general unifier that unifies all antecedents $\{A_i \mid i \in S\}$ for every $S \in \rho$.

$$\mathit{partition\_to\_mgu}(\rho, [A^u_1, \ldots, A^u_m]) = \underset{S\in\rho}{\circ} \mathit{mgu}(\{A^u_i \mid i \in S\})$$

Let $\mathcal{PU} = \{\langle \rho, \theta\rangle \mid \rho \in \mathcal{P} \wedge \theta = \mathit{partition\_to\_mgu}(\rho, A^u) \wedge \mathcal{D} \models \bar{\exists}_\emptyset \theta(A^c)\}$. $\mathcal{PU}$ contains all partitions for which *partition_to_mgu* is defined and for which the comparison antecedents A^c are still satisfiable after applying the unifier. The next step is to filter out antecedents so that every set in the partition has only one representative. This is done by computing $\mathit{filter}(A^u, \langle \rho, \theta\rangle)$ for each $\langle \rho, \theta\rangle \in \mathcal{PU}$ where the *filter* function is as follows:

$$\mathit{filter}([A^u_i|T], \langle \rho, \theta\rangle) = \begin{cases} [\theta(A^u_i)|\mathit{filter}(T, \langle \rho, \theta\rangle)] & \text{if } \exists S \in \rho : i = \min(S) \\ \mathit{filter}(T, \langle \rho, \theta\rangle) & \text{otherwise} \end{cases}$$
$$filter([], \_) = []$$

Finally, we add mode indicators to all remaining user-defined antecedents:

$$\mathit{modes}([A^{u'}|T]) = \begin{cases} \langle [a_r(\bar{X}, \mathtt{p})|A^m], N\rangle & \text{if } A^{u'} = a(\bar{X}) \\ \langle [a_r(\bar{X}, N')|A^m], [N' \neq \mathtt{p}|N]\rangle & \text{if } A^{u'} = \mathit{del}(a(\bar{X})) \end{cases}$$
$$\text{where } \langle A^m, N\rangle = \mathit{modes}(T)$$
$$\mathit{modes}([]) = \langle [], []\rangle$$

For every $\langle \rho, \theta\rangle \in \mathcal{PU}$, the CHR translation $T_R(P)$ contains a rule

$$r_\rho \ @ \ H \implies g_1, g_2 \mid C' \text{ pragma priority}(p + 2)$$

where $\langle H, g_1\rangle = \mathit{modes}(\mathit{filter}(A^u, \langle \rho, \theta\rangle))$, $g_2 = \theta(A^c)$ and $C' = \theta(C)$.

Example 2. A LA implementation of Dijkstra's shortest path algorithm is

```
d1 @ 1   : source(V) => dist(V,0).
d2 @ 1   : dist(V,D1), dist(V,D2), D2 < D1 => del(dist(V,D1)).
d3 @ D+2 : dist(V,D), e(V,C,U) => dist(U,D+C).
```

Its translation is

[5] $\mathcal{P}$ contains B_m elements in the worst case with B_m the m^{th} Bell number.

```
e_r(V,C,U,M) \ e(V,C,U) <=> M \= n | true pragma priority(1).
e_r(V,C,U,n) , e(V,C,U) <=> e_r(V,C,U,b) pragma priority(1).
               e(V,C,U) <=> e_r(V,C,U,p) pragma priority(2).

e_r(V,C,U,M) \ del(e(V,C,U)) <=> M \= p | true pragma priority(1).
e_r(V,C,U,p) , del(e(V,C,U)) <=> e_r(V,C,U,b) pragma priority(1).
               del(e(V,C,U)) <=> e_r(V,C,U,n) pragma priority(2).

... % (similar rules for source/1 and dist/2)

d1_1   @ source_r(V,p) ==> dist(V,0) pragma priority(3).
d2_1/2 @ dist_r(V,D_1,p), dist_r(V,D_2,p) ==>
                          D_2 < D_1 | del(dist(V,D_1)) pragma priority(3).
d3_1/2 @ dist_r(V,D,p), e_r(V,C,U,p) ==> dist(U,D+C) pragma priority(D+4).
```

Example 3. A rule from the union-find implementation of [9] is the following:

```
uf4 @ 1 : union(X,Y), find(X,Z), find(Y,Z) => del(union(X,Y)).
```

Because antecedents `find(X,Z)` and `find(Y,Z)` are unifiable, this leads to the following two CHR rules:

```
uf4_1/2/3 @ union_r(X,Y,p), find_r(X,Z,p), find_r(Y,Z,p) ==>
                                    del(union(X,Y)) pragma priority(3).
uf4_1/23  @ union_r(X,X,p), find_r(X,Z,p) ==>
                                    del(union(X,X)) pragma priority(3).
```

3.2 The Correspondence Between LA and CHR^{rp} Derivations

In this section, we show that every derivation of the original program under the Logical Algorithms semantics, corresponds to a derivation of the translation under the ω_p semantics of CHR^{rp}. In order to do so, we introduce a mapping between (reachable) CHR execution states and LA states:

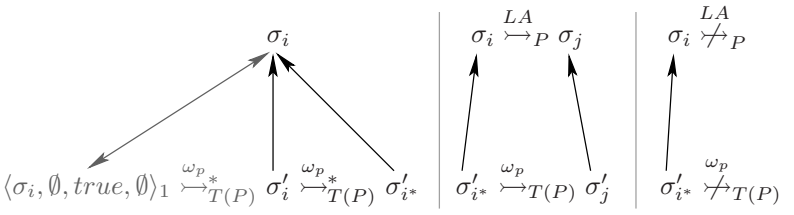
$$\begin{aligned} chr\_to\_la(\sigma) = &\{a(\bar{X}) \mid a(\bar{X}) \in A \vee (a_r(\bar{X}, M) \in A \wedge M \neq \mathtt{n})\} \\ &\cup \{del(a(\bar{X})) \mid del(a(\bar{X})) \in A \vee (a_r(\bar{X}, M) \in A \wedge M \neq \mathtt{p})\} \end{aligned}$$

where $\sigma = \langle G, S, B, T \rangle_n$ and $A = G \cup chr(S)$. The mapping function takes into account the constraints that are still in the goal or for which the set and deletion semantics rules have not fired yet.

Theorem 2. *For every reachable CHR^{rp} state σ, if $\sigma \overset{\omega_p}{\longmapsto}_{T(P)} \sigma'$ then either $chr\_to\_la(\sigma) = chr\_to\_la(\sigma')$ or $chr\_to\_la(\sigma) \overset{LA}{\longmapsto}_P chr\_to\_la(\sigma')$.*

Theorem 3. *For every Logical Algorithms state σ_i and reachable CHR^{rp} state σ'_i with $chr\_to\_la(\sigma'_i) = \sigma_i$, there exists a finite CHR^{rp} derivation $\sigma'_i \overset{\omega_p}{\longmapsto}^*_{T(P)} \sigma'_{i*}$ with $chr\_to\_la(\sigma'_{i*}) = \sigma_i$ such that if $\sigma_i \overset{LA}{\longmapsto}_P \sigma_j$ then $\sigma'_{i*} \overset{\omega_p}{\longmapsto}_{T(P)} \sigma'_j$ with $chr\_to\_la(\sigma'_j) = \sigma_j$ and if σ_i is a final state then σ'_{i*} is also a final state.*

Given a Logical Algorithms state σ, we can use $\langle \sigma, \emptyset, true, \emptyset \rangle_1$ as initial state for the CHR$^{\text{rp}}$ derivation. If we extend the LA and CHR$^{\text{rp}}$ transitions with appropriate labels, a LA program P and its translation $T(P)$ are weakly bisimilar[6]. Theorem 3 is illustrated in the figure below.



4 Translating CHR$^{\text{rp}}$ Programs into Logical Algorithms

In the previous section, we have shown that Logical Algorithms can be translated into equivalent CHR$^{\text{rp}}$ programs. In this section, we show how a particular subset of CHR$^{\text{rp}}$ can be translated into equivalent Logical Algorithms programs. This allows us to apply the meta-complexity theorem for Logical Algorithms to these CHR$^{\text{rp}}$ programs. The following three properties are required:

1. In all *reachable* states $\sigma = \langle G, S, B, T \rangle_n$: $vars(S) = \emptyset$.
2. All built-in constraints are comparisons; there are no built-in tell constraints.
3. A rule's priority depends on the arguments of at most *one* of its heads.

Here a state σ is reachable if there exists a derivation $\langle G, \emptyset, true, \emptyset \rangle_1 \overset{\omega_p}{\rightarrowtail}^*_P \sigma$. Because the order of heads in a CHR$^{\text{rp}}$ rule is not important, we can assume without loss of generality that the priority depends on the left-most head only[7]. Given a program P satisfying these properties, a CHR$^{\text{rp}}$ rule $r \in P$ of the form

$$r \ @ \ A_1, \ldots, A_m \backslash A_{m+1}, \ldots, A_n \iff g \mid C_1, \ldots, C_l \text{ pragma priority}(p)$$

is translated as

$$\begin{aligned} r \ @ \ p \ : \ A_1^{id}, \ldots, A_n^{id}, \mathit{alldiff}(Id_1, \ldots, Id_n), g, \mathit{next\_id}(Id_{next}) \Rightarrow \\ \mathit{del}(A_{m+1}^{id}), \ldots, \mathit{del}(A_n^{id}), \mathit{del}(\mathit{next\_id}(Id_{next})), \\ C_1^{id}, \ldots, C_l^{id}, \mathit{next\_id}(Id_{next} + l) \end{aligned}$$

where $A_i^{id} = \texttt{a(}\bar{X}\texttt{,}Id_i\texttt{)}$ if $A_i = \texttt{a(}\bar{X}\texttt{)}$, $C_i^{id} = \texttt{a(}\bar{X}\texttt{,}Id_{next} + i - 1\texttt{)}$ if $C_i = \texttt{a(}\bar{X}\texttt{)}$ and $\mathit{alldiff}\,(S) = \{(x \neq y) \mid x, y \in S \wedge x \neq y\}$. The initial database consists of the goal (where each constraint is extended with a unique identifier) and a `next_id(`Id_{next}`)` assertion (with Id_{next} the next free identifier).

[6] A LA transition $\sigma \overset{LA}{\rightarrowtail}_P \sigma'$ is labeled $\sigma' \backslash \sigma$, a CHR$^{\text{rp}}$ transition $\sigma \overset{\omega_p}{\rightarrowtail}_{T(P)} \sigma'$ is labeled $\mathit{chr\_to\_la}(\sigma') \setminus \mathit{chr\_to\_la}(\sigma)$ if this set is not empty, and τ (internal action) otherwise.

[7] We make abstraction of the syntactical limitations of simpagation rules.

Example 4. The following CHR^{rp} program implements a merge sort algorithm. Its input consists of a series of n (a power of 2) `number`/1 constraints. Its output is a sorted list of the numbers in the input, represented as `arrow`/2 constraints, where `arrow(`X`,`Y`)` indicates that X is right before Y.

```
ms1 @ arrow(X,A) \ arrow(X,B) <=> A < B | arrow(A,B) pragma priority(1).
ms2 @ merge(N,A),  merge(N,B) <=> A < B |
                          merge(2*N+1,A), arrow(A,B) pragma priority(2).
ms3 @ number(X) <=> merge(0,X) pragma priority(3).
```

Its Logical Algorithms translation is

```
ms1 @ 1 : arrow(X,A,Id1), arrow(X,B,Id2), A < B, next_id(NId) =>
                del(arrow(X,B,Id2)), arrow(A,B,NId),
                del(next_id(NId)), next_id(NId+1).
ms2 @ 2 : merge(N,A,Id1), merge(N,B,Id2), A < B, next_id(NId) =>
                del(merge(N,A,Id1)), del(merge(N,B,Id2)),
                merge(2*N+1,A,NId), arrow(A,B,NId+1),
                del(next_id(NId)), next_id(NId+2).
ms3 @ 3 : number(X,Id), next_id(NId) => del(number(X,Id)),
                merge(0,X,NId), del(next_id(NId)), next_id(NId+1).
```

Note that since the guard prevents the head constraints from being equal, the all different constraint on the constraint identifiers is not needed.

For this LA program, we can derive that given n initial `number`/2 assertions, there are $\mathcal{O}(n \log n)$ strong prefix firings and so using the meta-complexity theorem, we derive that the total runtime is $\mathcal{O}(n \log n)$. In contrast, when applying the meta-complexity theorem for CHR of [6], we find a runtime upperbound of $\mathcal{O}(n^3 \log n)$. The LA theorem clearly gives considerably more accurate results.

5 The Implementation of LA in CHR Under ω_r

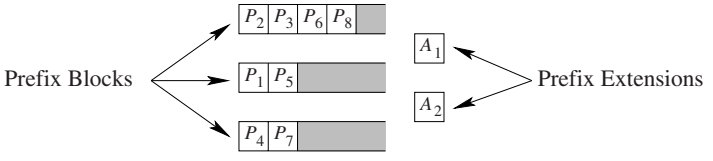
In this section, we present an implementation for LA in CHR under the refined operational semantics [2]. This implementation consists of the compilation of LA programs to CHR rules, combined with a scheduler module that is responsible for the execution control. It is based on the high-level implementation proposal of [9]. By using a CHR compiler with advanced indexing support, our implementation achieves the complexity required for the meta-complexity theorem of LA to hold. We make use of Prolog as host language, but the implementation can easily be adapted to work with other host languages.

5.1 Overview

The implementation is based on a form of eager matching, similar to the RETE algorithm [3]. Partial matches (called prefix instances in [9]) are stored and extended by new assertions. Full matches (rule instances) are inserted in a global

priority queue and the highest priority rule instance is fired. Only the highest priority partial matches are extended. This is enforced by storing partial matches in priority queues as well. Every partial match knows its priority as it is a rule instance prefix, and thus contains the leftmost head.

Every new assertion is scheduled to be combined with the highest priority partial match with which it has not been combined yet. After combining, it is rescheduled to be combined with the next partial match. This is done by using a data structure consisting of a series of *prefix blocks*: priority queues containing partial matches, alternated with a series of *prefix extensions*: the assertions with which the prefixes need to be combined. This is illustrated in the figure below. The highest priority elements in the data structure are scheduled for combination on the global priority queue. In the figure, these are the combination of prefix P_2 with extension A_1 and that of prefix P_1 with extension A_2. Full matches (rule instances) are scheduled on the global priority queue directly.



If a prefix and its extension share arguments, there is one data structure for each combination of these arguments, so that only feasible prefix/extension combinations are scheduled. For example, in the shortest path algorithm shown in the examples so far, there is a prefix/extension data structure for each different node, both for rule `d2` and rule `d3`.

5.2 The Compilation of LA Programs to CHR Rules

We present the compilation of Logical Algorithms to CHR rules by example. As example program, we use Dijkstra's shortest path algorithm. A compiled LA program consists of rules for

- Maintaining a representation for the assertions
- Removing the representation of invalidated prefix instances, prefix extensions and rule instances
- Generating and scheduling new such representations after a new assertion
- Extending prefix instances and firing rule instances

These rules are now described in more detail.

Representation of the Assertions. For every positive user-defined atom $A = a(\bar{X})$ such that A is asserted or $del(A)$ is asserted, there exists a unique constraint representation $a_r(\bar{X}, M)$ where M is

- an uninstantiated Prolog variable if A is asserted and $del(A)$ is not
- the atom "`n`" if $del(A)$ is asserted and A is not
- the atom "`b`" if both A and $del(A)$ are asserted

In contrast with the representation used in Section 3.1, we use a Prolog variable as mode indicator for (strictly) positive assertions. Further on we show how this simplifies the task of removing invalidated prefix instances, prefix extensions and rule instances after a negative assertion.

Handling New Assertions. For every new assertion, a constraint representation is created and merged with the existing representation if such exists. For a new positive or negative `dist`/2 assertion, this looks as follows:

```
    dist(V,D)  <=> dist_r(V,D,_).
del(dist(V,D)) <=> dist_r(V,D,n).

dist_r(V,D,M) \ dist_r(V,D,n) <=> var(M) | M = b.
dist_r(V,D,_) \ dist_r(V,D,_) <=> true.
```

Here, `var`/1 succeeds if its argument is an uninstantiated variable. The first two rules create a new representation. The third and fourth rule merge this representation with the existing representation if such exists. We rely on the refined operational semantics which states that rules are tried in textual order and occurrences are tried from right to left (so that the fourth rule always removes the most recently created representation).

Clean-up. Prefix instances, prefix extensions and rule instances are all represented as constraints, $\alpha - \beta$ constraints for short using RETE terminology. The $\alpha - \beta$ constraints contain all arguments of their constituent antecedents that appear in the remaining antecedents or in the conclusion. They also contain the mode indicators of the representations of their positive constituent antecedents. Every $\alpha - \beta$ constraint has a unique identifier argument.

If A is a positive assertion, then a negative assertion $del(A)$ causes the mode indicator for A to be instantiated to the atom "`b`". This triggers all $\alpha - \beta$ constraints in which A occurs as a positive antecedent, which are then removed by rules like the following.

```
d1_ri(_,b,Id)     <=> remove_ri(Id).
d2_pi_1(_,_,b,Id) <=> remove_pi(Id).
d2_pe_1(_,b,Id)   <=> remove_pe(Id).
```

Here, `d1_ri`/3 represents a rule instance of rule `d1`, `d2_pi_1`/4 represents the first prefix instance of rule `d2` and `d2_pe_1`/3 represents its prefix extension. Similar clauses for rule `d3` exist. The calls to `remove_ri`/1, `remove_pi`/1 and `remove_pe`/1 remove respectively the rule instance, prefix instance and prefix extension from the scheduling data structures.

Scheduling. New assertions are scheduled as rule instance (for `source`/1 in rule `d1`), as prefix instance (for `dist`/2 as first antecedent in rules `d2` and `d3`) or as prefix extension (for `dist`/2 as second antecedent in rule `d2` and `e`/3 as second antecedent in rule `d3`). For prefixes instances and extensions, the matching

prefix/extension data structure is found by matching on a key consisting of the rule name, prefix number and arguments shared between prefix and extension.

```
source_r(V,M) ==> var(M) | d1_ri(V,M,Id_1), schedule_ri(1,Id_1).
dist_r(V,D,M) ==> var(M) | d2_pi_1(V,D,M,Id_1), schedule_pi(d2_1(V),1,Id_1),
                           d2_pe_1(D,M,Id_2), schedule_pe(d2_1(V),Id_2),
                           d3_pi_1(V,D,M,Id_3), schedule_pi(d3_1(V),D+2,Id_3).
e_r(V,C,U,M)  ==> var(M) | d3_pe_1(C,U,M,Id_1), schedule_pe(d3_1(V),Id_1).
```

We use fresh variables as identifiers for the generated $\alpha - \beta$ constraints. Predicate `schedule_ri`/2 schedules a rule instance with given priority and identifier, `schedule_pi`/3 schedules a prefix instance with given matching key (for finding the correct prefix/extension data structure), priority and identifier, and `schedule_pe`/2 schedules a prefix extension with given key and identifier.

Matching and Firing. A rule instance is fired by asserting a `fire`/1 constraint which contains the identifier argument of its $\alpha - \beta$ constraint representation. A prefix instance is combined with a prefix extension by the assertion of a `cmb`/2 constraint which contains the identifiers of their $\alpha - \beta$ constraints.

```
d1_ri(V,_,I), fire(I) <=> dist(V,0).
d2_pi_1(V,D_1,_,I_1), d2_pe_1(D_2,_,I_2) \ cmb(I_1,I_2) <=> D_2 < D_1 | del(dist(V,D_1)).
d3_pi_1(V,D,_,I_1), d3_pe_1(C,U,_,I_2) \ cmb(I_1,I_2) <=> dist(U,D+C).
```

If the combination of a prefix instance with a prefix extension is another prefix instance, a new $\alpha - \beta$ constraint representation is made which is then scheduled for combination. Otherwise, the body of the combination rule corresponds to the conclusion of the rule that it implements.

5.3 Priority Queues

A priority queue or heap is a data structure that contains a set of prioritized items and supports the following operations: inserting and removing an item, finding a highest priority item and merging with another queue. The implementation proposal in [9] suggests the use of two types of priority queues, one for the fixed priorities, where each of the supported operations takes constant time, and a Fibonacci heap for the dynamic priorities.

Fibonacci heaps [4] are a type of priority queue that offer $\mathcal{O}(1)$ amortized time insertion, heap merging and finding a highest priority item, and $\mathcal{O}(\log n)$ amortized time item removal with n the number of items in the queue. It is suggested in [9] that by using only one node per priority, using linked lists to represent the items that share this priority, the item removal cost can be reduced to $\mathcal{O}(\log N)$ with N the number of distinct priorities. This increases the cost of heap merging from $\mathcal{O}(1)$ for a single merge operation to a total cost of $\mathcal{O}(n \log N)$ for merging heaps when there are n items in total and N distinct priorities. Fortunately, this increased complexity does not influence the total complexity given by the meta-complexity theorem for Logical Algorithms.

The CHR implementation of the Fibonacci heaps is based on the description in [17] and extended to allow multiple heaps that can be merged and to use only one node for each distinct priority per heap.

5.4 The Scheduler

The scheduler implements the predicates `schedule_ri`/2, `schedule_pi`/3 and `schedule_pe`/2. It maintains the prefix/extension data structures and makes use of the priority queue implementations for this purpose. The scheduler initiates the firing of rule instances (by asserting a `fire`/1 constraint) and the combination of a prefix and extension (by asserting a `cmb`/2 constraint).

6 A Complexity Result

In Section 2.1, we have given the time complexity of Logical Algorithms. Theorem 4 shows that our implementation has the required complexity to make this result valid. Empirical evidence has confirmed this.

Theorem 4. *The time complexity of LA programs executed using our implementation is* $\mathcal{O}(|S_0| + P_s + (P_d + A_d) \cdot \log N)$ *with* S_0*,* P_s*,* P_d*,* A_d *and* N *as defined in Section 2.1.*

Proof (Sketch). The proof for the high-level implementation description in [9] can be used if the following holds:

- Inserting an element in a priority queue takes $\mathcal{O}(1)$ time. Deleting an element from one takes $\mathcal{O}(1)$ time for elements with a static priority and $\mathcal{O}(\log N)$ (amortized) time for elements with a dynamic priority. Merging two priority queues takes $\mathcal{O}(1)$ (amortized) time.
- Finding the first prefix block of a prefix/extension data structure for a given key consisting of a rule name, prefix number and the arguments shared between prefix and extension, takes constant time. The same holds for creating such a data structure should it not exist, and for adding a new prefix block at the end of such a structure.
- Finding a prefix instance, prefix extension or rule instance given its identifier takes constant time.

By using the advanced indexing supplied by the CHR compiler, our implementation satisfies these requirements, except that merging takes more than $\mathcal{O}(1)$ time. The heaps that are merged (when deleting a prefix extension) contain together up to P_d items[8]. As a result, the total cost of heap merging is $\mathcal{O}(P_d \cdot \log N)$.

For merging local priority queues, both for static and dynamic priorities, we need an optimal implementation of the union find algorithm. This supports

[8] Although a prefix instance can "move" from one heap to another, each move operation corresponds to a (larger) prefix or rule instance.

quasi-constant lookup via, and unification of priority queue identifiers. Such an optimal implementation exists for CHR [15]. □

Although the cost of heap merging when using only one node for each distinct priority, appears to be larger than assumed in [9], the meta-complexity theorem remains valid. While the complexity requirements are very stringent, our high-level implementation in CHR is able to satisfy them.

7 Conclusions

In this paper, we have investigated the relationship between the Logical Algorithms language and Constraint Handling Rules. We have presented an elegant translation from LA to CHR^{rp}: CHR with rule priorities. The original program and its translation are essentially weakly bisimilar. A translation scheme is given that allows a subclass of CHR^{rp} to be translated into Logical Algorithms. This allows direct application of the meta-complexity theorem for Logical Algorithms to (the translation of) these CHR^{rp} programs, which gives more accurate results than the meta-complexity theorem for CHR given in [6].

By compiling LA rules into CHR rules and using a scheduler (also written in CHR) to control the execution, we are able to execute LA programs in any CHR implementation based on the refined operational semantics of CHR. We also achieve the required complexity by using the K.U.Leuven CHR system which uses optimized indexing structures. This result illustrates the strength of CHR for implementing complex systems in a concise way with optimal complexity. Moreover, when combining the CHR^{rp} to Logical Algorithms translation with the Logical Algorithms implementation in CHR, we have a first optimized implementation for a subset of CHR^{rp}.

Related Work. In [7,6], Frühwirth presents a meta-complexity theorem for CHR programs containing only simplification rules. It uses level mappings to find an upperbound on the number of rule firings and makes a (highly pessimistic) worst case estimate of the time spent on trying rules in each derivation step. The approach is more or less independent of the actual CHR implementation used and so it often largely overestimates the actual time complexity. Tight complexity results have been derived for particular programs using ad hoc techniques in [15,17].

Future Work. The Logical Algorithms approach applied to CHR^{rp} (Section 4), could be extended to a larger subclass of CHR^{rp} by allowing non-ground constraints and built-in (tell) constraints. This will require a more complex scheduler and it remains unclear what the effects will be for the complexity theorem. The derivation length of a program is bounded by monotone information growth in Logical Algorithms and by using level mapping in the work of Frühwirth. We plan to investigate the advantages and disadvantages of both approaches.

References

1. Christiansen, H.: A constraint-based bottom-up counterpart to definite clause grammars. In: Recent Advances in Natural Language Processing III, Selected Papers. Current Issues in Linguistic Theory vol. 260 (2004)
2. Duck, G.J., Stuckey, P.J., García de la Banda, M., Holzbaur, C.: The refined operational semantics of Constraint Handling Rules. In: Demoen, B., Lifschitz, V. (eds.) ICLP 2004. LNCS, vol. 3132, Springer, Heidelberg (2004)
3. Forgy, C.L.: Rete: A fast algorithm for the many pattern/many object pattern match problem. Artif. Intell. 19(1), 17–37 (1982)
4. Fredman, M.L., Tarjan, R.E.: Fibonacci heaps and their uses in improved network optimization algorithms. J. ACM 34(3), 596–615 (1987)
5. Frühwirth, T.: Theory and practice of constraint handling rules. J. Log. Program. 37(1-3), 95–138 (1998)
6. Frühwirth, T.: As time goes by II: More automatic complexity analysis of concurrent rule programs. Electr. Notes Theor. Comput. Sci., 59(3) (2001)
7. Frühwirth, T.: As time goes by: Automatic complexity analysis of concurrent rule programs. In: 8th Intl. Conf. Principles of Knowledge Representation and Reasoning, pp. 547–557 (2002)
8. Ganzinger, H., McAllester, D.A.: A new meta-complexity theorem for bottom-up logic programs. In: Goré, R.P., Leitsch, A., Nipkow, T. (eds.) IJCAR 2001. LNCS (LNAI), vol. 2083, pp. 514–528. Springer, Heidelberg (2001)
9. Ganzinger, H., McAllester, D.A.: Logical algorithms. In: Stuckey, P.J. (ed.) ICLP 2002. LNCS, vol. 2401, pp. 209–223. Springer, Heidelberg (2002)
10. De Koninck, L., Schrijvers, T., Demoen, B.: CHR$^{\mathrm{rp}}$: Constraint handling rules with rule priorities. Technical Report CW 479, K.U.Leuven, Belgium (March 2007)
11. De Koninck, L., Schrijvers, T., Demoen, B.: The correspondence between the logical algorithms language and CHR. Technical Report CW 480, K.U.Leuven, Belgium (March 2007)
12. McAllester, D.A.: On the complexity analysis of static analyses. In: Cortesi, A., Filé, G. (eds.) SAS 1999. LNCS, vol. 1694, pp. 312–329. Springer, Heidelberg (1999)
13. Schrijvers, T.: Analyses, Optimizations and Extensions of Constraint Handling Rules. PhD thesis, K.U.Leuven, Leuven, Belgium (2005)
14. Schrijvers, T., Demoen, B.: The K.U.Leuven CHR system: implementation and application. In: First Workshop on CHR: Selected Contributions. Ulmer Informatik-Berichte. Universität Ulm, vol. 2004-01 pp. 1–5 (2004)
15. Schrijvers, T., Frühwirth, T.: Optimal union-find in constraint handling rules. Theory and Practice of Logic Programming, 6(1&2) (2006)
16. Sneyers, J., Schrijvers, T., Demoen, B.: The computational power and complexity of Constraint Handling Rules. In: 2nd Workshop on Constraint Handling Rules. Reports CW, Dept. of Computer Science, K.U.Leuven, Belgium, vol. 421, pp. 3–17 (2005)
17. Sneyers, J., Schrijvers, T., Demoen, B.: Dijkstra's algorithm with Fibonacci heaps: An executable description in CHR. In: 20th Workshop on Logic Programming. INFSYS Research Report, TU Wien, vol. 1843-06-02 pp. 182–191 (2006)
18. Van Weert, P., Sneyers, J., Schrijvers, T., Demoen, B.: Extending CHR with negation as absence. In: 3rd Workshop on CHR. Reports CW, Dept. of Computer Science, K.U. Leuven, Belgium, vol. 452, pp. 125–140 (2006)

Observable Confluence for Constraint Handling Rules

Gregory J. Duck[1], Peter J. Stuckey[1], and Martin Sulzmann[2]

[1] NICTA Victoria Laboratory
Department of Computer Science and Software Engineering
University of Melbourne, 3010, Australia
{gjd,pjs}@cs.mu.oz.au
[2] School of Computing, National University of Singapore
S16 Level 5, 3 Science Drive 2, Singapore 117543
sulzmann@comp.nus.edu.sg

Abstract. Constraint Handling Rules (CHR) are a powerful rule based language for specifying constraint solvers. Critical for any rule based language is the notion of confluence, and for terminating CHR programs there is a decidable test for confluence. But many CHR programs that are in practice confluent fail this confluence test. The problem is that the states that illustrate non-confluence are not observable from the initial goals of interest. In this paper we introduce the notion of observable confluence, a more general notion of confluence which takes into account whether states are observable. We devise a test for observable confluence which allows us to verify observable confluence for a range of CHR programs dealing with agents, type systems, and the union-find algorithm.

1 Introduction

Constraint Handling Rules [3] (CHR) are a powerful rule based language for specifying constraint solvers. Constraint handling rules operate on a global multi-set (conjunction) of constraints. A constraint handling rule defines a rewriting from one multi-set of constraints to another.

A critical issue for any rule based language is the notion of confluence. A CHR program is confluent if all possible rewriting sequences from a given input lead to the same result. Thus confluent programs have a "deterministic behaviour" with respect to the input goals, i.e. given some input goal, we can uniquely determine the output. For terminating CHR programs there is a decidable test for confluence [1]. Unfortunately there are many (terminating) programs which are confluent in practice, but fail to pass the test.

In this paper, we make the following contributions:

- We introduce the notion of *observable confluence* which generalises the notion of confluence by only considering rewriting steps which are observable with respect to some invariant (Section 3.3).
- We give a generalised confluence test where we only need to consider joinability of critical pairs satisfying the invariant (Section 4).

V. Dahl and I. Niemelä (Eds.): ICLP 2007, LNCS 4670, pp. 224–239, 2007.

– We show that the generalised confluence test enables us to verify observable confluence of CHR used for the specification of agents, the union-find algorithm, and type systems (Section 5). All of these classes of CHR programs are non-confluent under the standard notion.

To the best of our knowledge, we are the first to study observable confluence in the context of a rule-based language. In the workshop papers [2,5], we reported some preliminary results. The present work represents a significantly revised and extended version of [2].

We continue in Section 2 where we consider a number of motivating examples. Section 3 provides background material on CHR.

2 Motivating Examples

The following examples fail the standard confluence test, but we can show that they are observable confluent with respect to some appropriate invariant.

2.1 Blocks World

In our first example, we consider a set of CHRs used for agent-oriented programming [5]. The following CHR program fragment describes the behaviour of an agent in a blocks world:[1]

```
g1 @ get(X), empty   <=> hold(X).
g2 @ get(X), hold(Y) <=> hold(X), clear(Y).
```

The constraint `hold(X)` denotes that the agent holds some element `X`, whereas `empty` denotes that the agent holds nothing. The constraint `clear(Y)` simply represents the fact that `Y` is not held. The constraint `get(X)` represents an action, to get some element `X`. The atoms preceding the '`@`' symbols are the *rule names*, thus the rules are named `g1` and `g2` respectively. Both rules are *simplification rules*, rewriting constraints matching the left-hand-side to the constraints in the right-hand-side. The first rule rewrites the constraints `get(X)` ∧ `empty` to `hold(X)`. The second rule rewrites `get(X)` ∧ `hold(Y)` to `hold(X)` ∧ `clear(Y)`.

It is clear that the rules are *non-confluent*. Consider the *critical state* `get(X)` ∧ `hold(Y)` ∧ `empty` formed by combining the heads of rules `g1` and `g2`. This critical state can be rewritten to either `hold(Y)` ∧ `hold(X)` by applying rule `g1`, or to `hold(X)` ∧ `clear(Y)` ∧ `empty` by applying rule `g2`. These two derived states are a *critical pair* between the two rules. The confluence test for CHR [1] states that a terminating program is confluent iff all critical pairs are *joinable*, i.e. can be rewritten to the same result. Since no rewriting steps can join `hold(Y)` ∧ `hold(X)` and `hold(X)` ∧ `clear(Y)` ∧ `empty`, the blocks world program is non-confluent.

[1] CHR follows a Prolog like notation, where identifiers starting with a lower case letter indicate predicates and function symbols, and identifiers starting with upper case letters indicate variables.

Let us consider the non-confluent state `get(X)` $\wedge$ `hold(Y)` $\wedge$ `empty` more closely: it represents the agent holding nothing (`empty`) whilst simultaneously holding an object Y (`hold(Y)`). Clearly, such a state is nonsense, so we would like to exclude it from consideration. To do this we need a weaker notion of confluence, i.e. confluence with respect to *valid* states. We refer to this as *observable confluence*.

Specifically, we can informally define valid states as follows:

"Either the agent holds some element `X` or holds nothing."
"There is at most one `get(_)` operation at one time."

The conjunction of these conditions is an invariant in the blocks world program. In this paper, we show that all non-confluent states violate this invariant, thus blocks world program is observable confluent.

A similar form of observable confluence under some invariant arises in our next example.

2.2 Union Find

Consider the following program which is part[2] of the simple union-find code from [9,8].

```
union    @ union(X,Y) <=> find(X,A), find(Y,B), link(A,B).
findNode @ X ~> PX \ find(X,R) <=> find(PX, R).
findRoot @ root(X) \ find(X,R) <=> R = X.
linkEq   @ link(X,X) <=> true.
link     @ link(X,Y), root(X), root(Y) <=> Y ~> X, root(X).
```

Both `findNode` and `findRoot` are *simpagation rules*, which are similar to simplification rules, except constraints on the LHS of the '`\`' symbol are not rewritten. This program defines an environment where the `root(X)` and `X ~> Y` constraints define trees, and `union(X,Y)` links trees so that they have the same root.

The union-find program is non-confluent, since there are eight non-joinable critical pairs. The authors of [8] classify the critical pairs as either *avoidable* (as in they should not arise in practice) and *unavoidable* (as inherent non-confluence in the union-find algorithm). For example, the critical state between the `linkEq` and `link` is `link(X,X)` $\wedge$ `root(X)` $\wedge$ `root(X)`, with two `root(X)` constraints.[3] The critical pair is `root(X)` $\wedge$ `root(X)` and `X ~> X` $\wedge$ `root(X)`, which is non-joinable. However, in [8] it is argued that this critical pair is *avoidable*, since the presence of two `root(X)` in the state violates the definition of a tree (i.e. X can only be the root of one tree). This kind of reasoning can be understood in terms of invariants and observable confluence.

As was the case with the blocks world example, we define an invariant that describes valid states. Firstly, we informally define *validTrees*, as follows:

[2] We have removed the `make` rule to simplify the invariant.
[3] CHR uses a multi-set semantics, thus here we consider $X \wedge X$ to be distinct from X.

> "For all `X` there is at most one `root(X)` or `X ~> Y`, and there are no cycles `X ~> Y`$_1$, ..., `Y`$_n$ `~> X`"

We are also interested in the confluence of a single `union(X,Y)` operation executed in isolation on valid trees. Some combinations of operations are not valid, thus we define *validOps* as follows:

> "There is at most one `union(_,_)` or `link(_,_)`, and if there is a `union(_,_)` there is no `find(_,_)`."

This condition helps enforce confluence: since the order in which these operations are executed can affect the final result. For example, executing a `find` before or after a `union` may produce different results, since the `union` may update the trees.

To ensure observable confluence, there is one final case to consider: a concurrent `link` and `find` operation. In this case, we can not simply make these operations mutually exclusive, since `link` and `find` do interact in the body of the `union` rule. However, the second argument to a `find` constraint must always be associated to the corresponding `link` constraint. Therefore, we define *validFind* as follows:

> "If there is a `find(X,Y)` then there is a `link(Y,_)` or `link(_,Y)` but no `link(Y,Y)`, and `Y` does not appear in any other constraint."

Define $\mathcal{U} = validTrees \wedge validOps \wedge validFind$, then we verify that $\mathcal{U}$ is preserved by rule application, and thus is an invariant. Furthermore, none of the non-joinable critical pairs, or any states extended from these critical pairs, satisfy $\mathcal{U}$.[4] Therefore the union-find program P is observable confluent with respect to $\mathcal{U}$. Or in other words, P is $\mathcal{U}$-confluent.

This shows that the `union` operation, executed in isolation on valid trees, is confluent, even though the program itself is not confluent.

2.3 Type Class Functional Dependencies

The invariants we have seen so far reject (critical) states by observing the constraints in the store. But in CHR, sometimes a state cannot be observable because of the *kind* of rules that have, or have not, fired so far. This is due to CHR *propagation rules* which only add new constraints but do not delete existing constraints. To avoid trivial non-termination, the CHR semantics maintains a *propagation history* to avoid re-application of the same rule on the same set of constraints. The short story is that certain states can never be observable because of their propagation histories. Our next example illustrates this point.

We consider CHRs which arise from the translation of type class constraints in Haskell [7] involving functional dependencies [4]. We directly give the CHRs and omit the type class program.

[4] In the original paper [8], one of the critical pairs, namely `link` + `findRoot`, was "unavoidable" under the author's conditions. However, we have sufficiently strengthened the conditions to make the "unavoidable" critical pair avoidable.

```
r1 @ f(int,bool,float)     <=> true.
r2 @ f(A,B1,C), f(A,B2,D) ==> B1 = B2.
r3 @ f(int,B,C)            ==> B = bool.
```

The first rule is a simplification rule. The second and third rule are *propagation rules*. Propagation rules do not delete the constraints matching the head, thus they only add constraints. For example, the second rule adds the constraint `B1 = B2` whenever we see `f(A,B1,C)` $\wedge$ `f(A,B2,D)` in the store. To avoid trivial non-termination, propagation rules maintain a history of applications, and avoid applying the same propagation rule more than once on the same set of constraints.

When testing for confluence, we examine critical states, which are minimal states where two different rule firings are possible. In order to be truly minimal, the propagation history is assumed to be as strong as possible, i.e. where no propagation rule can fire, except possibly the two rules used to generate the critical state itself.

Rules `r1`, `r2` give rise to the critical state `f(int,bool,float)`$\wedge$`f(int,B2,D)` from which we can derive two different states as shown by the following rewriting steps `f(int,bool,float)` $\wedge$ `f(int,B2,D)` $\rightarrowtail_{r1}$ `f(int,B2,D)` and

$$\begin{aligned} & \texttt{f(int,bool,float)} \wedge \texttt{f(int,B2,D)} \\ \rightarrowtail_{r2}\ & \texttt{f(int,bool,float)} \wedge \texttt{f(int,bool,D)} \wedge \texttt{B2 = bool} \\ \rightarrowtail_{r1}\ & \texttt{f(int,bool,D)} \wedge \texttt{B2 = bool} \end{aligned}$$

Note that we cannot apply the rule `r3` to the state `f(int,B2,D)` because the propagation history in the critical state disallows this. As the critical state has led to two different non-joinable states, the above CHR program is non-confluent.

But in practice the critical state `f(int,bool,float)` $\wedge$ `f(int,B2,D)` where the propagation history prevents rule `r3` from firing on the second constraint cannot arise. The initial state always begins with an empty (weakest) propagation history. Hence, rule `r3` must have fired already on the second constraint. If this were the case, then the constraint $B2 = bool$ should appear in the critical state, but $B2 = bool$ does not occur. Therefore, the critical state is not *reachable* from any initial goal. Further details are given in Section 5.1, where we show that this program is in fact observable confluent with respect to the reachability invariant.

Next, we review background material on CHR before introducing the notion of observable confluence and the observable confluence test.

3 Preliminaries

A CHR *simpagation* rule is of the form $(r \ @ \ H_1' \setminus H_2' \iff g \mid C)$ where we propagate H_1' and simplify H_2' by C if the guard g is satisfied. We call r a *propagation* rule if H_2' is empty and a *simplification* rule if H_1' empty. As seen in Section 2, $(r \ @ \ H_2' \iff g \mid C)$ is shorthand for the simplification rule $(r \ @ \ \emptyset \setminus H_2' \iff g \mid C)$, and $(r \ @ \ H_1' \implies g \mid C)$ is shorthand for the propagation rule $(r \ @ \ H_1' \setminus \emptyset \iff g \mid C)$.

In CHR there are two distinct types of constraints: *user constraints* and *built-in constraints*. Built-in constraints are provided by an external solver, whereas user constraints are defined by the rules. Only user constraints may appear in the rule head (H_1' and H_2'), and only built-in constraints in the guard g. The body C may contain both kinds of constraints.

Formally, CHR is a reduction system $\langle \rightarrowtail, \Sigma \rangle$ where $\rightarrowtail$ is the CHR rewrite relation and Σ is the set of all *CHR states*.

Definition 1 (CHR State). *A* state *is a tuple of the form*

$$\langle G, S, B, \mathcal{T}, \mathcal{V} \rangle$$

where goal G is a multi-set of constraints (both user and built-in), user store S is a multi-set of user constraints, built-in store B is a conjunction of built-in constraints, token store $\mathcal{T}$ is a set of tokens*, and variables of interest $\mathcal{V}$ is the set of variables present in the initial goal. Throughout this paper we use symbol 'σ' to represent a state, and Σ to represent the set of all states.* □

The *built-in constraint store* B contains any built-in constraint that has been passed to the built-in solver. Since we will usually have no information about the internal representation of B, we treat it as a conjunction of constraints. We assume $\mathcal{D}$ denotes the theory for the built-in constraints B.

The *token store*[5] $\mathcal{T}$ is a set of *tokens* of the from $(r@C)$, where r is a rule name, and C is a sequence of user constraints. A CHR propagation rule r may only be applied to C if the token $(r@C)$ exists in the token store. This is necessary to prevent trivial non-termination for propagation rules. Whenever a new constraint is added to the user store, the *token set* of that constraint is added to the token store.

Definition 2 (Token Set). *Let P be a CHR program, C be a set of user constraints, and S a user-store, then define*

$$T_{(C,S)} = \{r@H' \mid (r@H \Longrightarrow g \mid B) \in P, H' \subseteq C \uplus S, C \subseteq H', H' \textit{ unifies with } H\}$$

to be the token set *of C with respect to S.* □

In the above, we write $\uplus$ for multi-set union. Later, we will also use multi-set intersection $\cap$.

We define $vars(o)$ as the free variables in some object o: e.g. term, formula, constraint. We define an *initial state* as follows.

Definition 3 (Initial State). *Given a multi-set of constraints G (i.e. the* goal*) the* initial state *with respect to G is $\langle G, \emptyset, true, \emptyset, vars(G) \rangle$.* □

The operational semantics of CHR[6] is based on the following three transitions which map states to states:

[5] The *token store* is also known as the *propagation history*.

[6] There are many different versions of the operational semantics of CHR. In this paper we use a version that is close to the original operational semantics described in [1]. This version is the most suitable for the study of confluence.

Definition 4 (Operational Semantics)
1. Solve $\langle \{c\} \uplus G, S, B, \mathcal{T}, \mathcal{V}\rangle \rightarrowtail \langle G, S, c \wedge B, \mathcal{T}, \mathcal{V}\rangle$
where c is a built-in constraint.
2. Introduce $\langle \{c\} \uplus G, S, B, \mathcal{T}, \mathcal{V}\rangle \rightarrowtail \langle G, \{c\} \uplus S, B, T_{(\{c\},S)} \uplus \mathcal{T}, \mathcal{V}\rangle$
where c is a user constraint.
3. Apply $\langle G, H_1 \uplus H_2 \uplus S, B, T \uplus \mathcal{T}, \mathcal{V}\rangle \rightarrowtail \langle C \uplus G, H_1 \uplus S, \theta \wedge B, \mathcal{T}, \mathcal{V}\rangle$
where there exists a (renamed apart) rule $(r \ @ \ H'_1 \setminus H'_2 \iff g \mid C)$ in P, and $T = \{(r@H_1, H_2)\}$ if $H'_2 = \emptyset$, otherwise $T = \emptyset$. The matching substitution θ is such that

$$\begin{cases} H_1 = \theta(H'_1) \\ H_2 = \theta(H'_2) \\ \mathcal{D} \models B \rightarrow \exists \bar{a}(\theta \wedge g) \end{cases}$$

where $\bar{a} = vars(g) - vars(H'_1, H'_2)$ and $\mathcal{D}$ denotes the built-in theory. □

A *derivation* is a sequence of states connected by transitions. We use notation $\sigma_0 \rightarrowtail^* \sigma_1$ to represent a derivation from σ_0 to σ_1.

3.1 Confluence

Confluence depends on the notion of equivalence between CHR states. The equivalence relation for CHR states is known as *variance*:

Definition 5 (Variance). *Two states*

$$\sigma_1 = \langle G_1, S_1, B_1, \mathcal{T}_1, \mathcal{V}\rangle \qquad \textit{and} \qquad \sigma_2 = \langle G_2, S_2, B_2, \mathcal{T}_2, \mathcal{V}\rangle$$

are variants *(written $\sigma_1 \approx \sigma_2$) if there exists a unifier ρ of S_1 and S_2, G_1 and G_2, $(\mathcal{T}_1 \Cap T_{(S_1,\emptyset)})$ and $(\mathcal{T}_2 \Cap T_{(S_2,\emptyset)})$ such that*

1. $\mathcal{D} \models \bar{\exists}_{\mathcal{V}_1} B_1 \rightarrow \bar{\exists}_{\mathcal{V}_1} \rho \wedge B_2$
2. $\mathcal{D} \models \bar{\exists}_{\mathcal{V}_2} B_2 \rightarrow \bar{\exists}_{\mathcal{V}_2} \rho \wedge B_1$

where $\mathcal{V}_1 = \mathcal{V} \cup vars(G_1) \cup vars(S_1) \cup vars(\mathcal{T}_1)$ and $\mathcal{V}_2 = \mathcal{V} \cup vars(G_2) \cup vars(S_2) \cup vars(\mathcal{T}_2)$. Otherwise the two states are variants if $\mathcal{D} \models \neg\bar{\exists}_\emptyset B_1$ and $\mathcal{D} \models \neg\bar{\exists}_\emptyset B_2$ (i.e. both states are $false$). □

Confluence relies on whether two states can derive the same state. This property is known as *joinability*.

Definition 6 (Joinable). *Two states σ_1 and σ_2 are* joinable *if there exists states σ'_1 and σ'_2 such that $\sigma_1 \rightarrowtail^* \sigma'_1$ and $\sigma_2 \rightarrowtail^* \sigma'_2$ and $\sigma'_1 \approx \sigma'_2$. We use the notation $(\sigma_1 \downarrow \sigma_2)$ to indicate that σ_1 and σ_2 are joinable.* □

Finally, we can define confluence as follows:

Definition 7 (Confluence). *A CHR program P is* confluent *if the following holds for all states σ_0, σ_1 and σ_2: If $\sigma_0 \rightarrowtail^* \sigma_1$ and $\sigma_0 \rightarrowtail^* \sigma_2$ then σ_1 and σ_2 are joinable.* □

3.2 Confluence Test

In [1] it was shown that confluence is decidable for terminating CHR programs. The confluence test for CHR depends on calculating all critical pairs between rules in the program. First we define the notion of a *critical ancestor state*.

Definition 8 (Critical Ancestor States). *Given two (renamed apart) rule instances:* $(r1 \ @ \ H_1 \backslash H_2 \iff g_1 \mid B_1)$ *and* $(r2 \ @ \ H_3 \backslash H_4 \iff g_2 \mid B_2)$*, then the set of all* critical ancestor states *(or simply* ancestor states*)* $\Sigma_{\mathcal{CP}}$ *between* $r1$ *and* $r2$ *is:*

$$\left\{ \begin{array}{l|l} \langle \emptyset, H_{r1}^{\Delta} \uplus H_{r2}^{\Delta} \uplus H_{r1}^{\cap}, & H_{r1} = H_1 \uplus H_2 \\ \quad H_{r1}^{\cap} = H_{r2}^{\cap} \wedge g_1 \wedge g_2, \mathcal{T}_{\mathcal{CP}}, \mathcal{V}_{\mathcal{CP}} \rangle & H_{r2} = H_3 \uplus H_4 \\ & H_{r1} = H_{r1}^{\cap} \uplus H_{r1}^{\Delta} \\ & H_{r2} = H_{r2}^{\cap} \uplus H_{r2}^{\Delta} \\ & \mathcal{V}_{\mathcal{CP}} = \\ & \quad vars(H_1 \wedge H_2 \wedge H_3 \wedge H_4 \wedge g_1 \wedge g_2) \end{array} \right\}$$

where, given $e_1 = (r1@H_1, H_2)$ *and* $e_2 = (r2@H_3, H_4)$*, then* $\mathcal{T}_{\mathcal{CP}} = \{e_i \mid i \in \{1,2\}, r_i \textit{is a propagation rule}\}$*.*

Basically, a critical ancestor state is a minimal state applicable to both rules. The sets $H_{r1}^{\cap}$ and $H_{r2}^{\cap}$ represent some potential overlap between the two rules. If the rules heads H_{r1} and H_{r2} do not overlap, then $H_{r1}^{\cap} = H_{r2}^{\cap} = \emptyset$ gives the only non-$false$ ancestor state.

We define a *critical pair* in terms of an ancestor state.

Definition 9 (Critical Pair). *Given the rules* $r1$*,* $r2$ *and the set* $\Sigma_{\mathcal{CP}}$ *from Definition 8, for* $\sigma_{\mathcal{CP}} \in \Sigma_{\mathcal{CP}}$ *where*

$$\sigma_{\mathcal{CP}} = \langle \emptyset, H_{r1}^{\Delta} \uplus H_{r2}^{\Delta} \uplus H_{r1}^{\cap}, H_{r1}^{\cap} = H_{r2}^{\cap} \wedge g_1 \wedge g_2, \mathcal{T}_{\mathcal{CP}}, \mathcal{V}_{\mathcal{CP}} \rangle$$

then the critical pair (σ_A, σ_B) *for* $\sigma_{\mathcal{CP}}$ *is*

$$\begin{array}{c} (\langle B_1, (H_{r1}^{\Delta} \uplus H_{r2}^{\Delta} \uplus H_{r1}^{\cap}) - H_2, H_{r1}^{\cap} = H_{r2}^{\cap} \wedge g_1 \wedge g_2, \mathcal{T}_A, \mathcal{V}_{\mathcal{CP}} \rangle, \\ \langle B_2, (H_{r1}^{\Delta} \uplus H_{r2}^{\Delta} \uplus H_{r1}^{\cap}) - H_4, H_{r1}^{\cap} = H_{r2}^{\cap} \wedge g_1 \wedge g_2, \mathcal{T}_B, \mathcal{V}_{\mathcal{CP}} \rangle) \end{array}$$

where $\mathcal{T}_A = \mathcal{T}_{\mathcal{CP}} - \{e_1\}$ *and* $\mathcal{T}_B = \mathcal{T}_{\mathcal{CP}} - \{e_2\}$ *and where* e_1 *and* e_2 *are defined as in Definition 8.* □

Informally, Definition 9 simply states that (σ_A, σ_B) is the result of respectively firing $r1$ and $r2$ on $\sigma_{\mathcal{CP}}$, whilst being careful to specify exactly *how* the rules were applied (e.g. how the constraints were matched against the rule head).

For the rest of the paper, we use $\sigma_{\mathcal{CP}}$ to denote the ancestor state of a critical pair $\mathcal{CP}$.

Confluence can be proven by showing that all critical pairs are joinable.

Theorem 1 (Confluence Test). *[1] Given a terminating CHR program P, if all critical pairs between all rules in P are joinable, then P is confluent.*

This is known as the *confluence test* for terminating CHR programs.

3.3 $\mathcal{I}$-Confluence

In this section we formally define $\mathcal{I}$-confluence (i.e. observable confluence)[7] with respect to an invariant $\mathcal{I}$.

Definition 10 (Invariant). *An* invariant $\mathcal{I}(\sigma)$ *is a property such that for all* σ_0 *and* σ_1*, we have that if* $\sigma_0 \rightarrowtail \sigma_1$ *(or* $\sigma_0 \approx \sigma_1$*) and* $\mathcal{I}(\sigma_0)$ *then* $\mathcal{I}(\sigma_1)$*.* □

Example 1 (Blocks World Invariant). First we define $\texttt{exists}(\sigma, M)$, which decides if the multi-set of user constraints M exists in σ:

$$\texttt{exists}(\langle G,S,B,\mathcal{T},\mathcal{V}\rangle, M) \Leftrightarrow \\ \exists S' \subseteq user(G) \uplus S \quad \wedge \quad \mathcal{D} \models builtin(G) \wedge B \rightarrow \exists_{vars(M)}(M = S')$$

where $user(G)$ and $builtin(G)$ returns all user/built-in constraints in G respectively.

The invariant for the blocks world example from Section 2.1 is formally represented as $\mathcal{B}(\sigma)$ where

$$\mathcal{B}(\sigma) \Leftrightarrow \neg\texttt{exists}(\sigma, \{\texttt{empty}, \texttt{empty}\}) \wedge \neg\texttt{exists}(\sigma, \{\texttt{empty}, \texttt{holds(\_)}\}) \wedge \\ \neg\texttt{exists}(\sigma, \{\texttt{holds(\_)}, \texttt{holds(\_)}\}) \wedge \neg\texttt{exists}(\sigma, \{\texttt{get(\_)}, \texttt{get(\_)}\})$$

The first three conditions state that the agent either holds something or holds nothing. The outcome is determined by the order in which `get` operations are executed. Therefore, we impose the fourth condition which guarantees that we only consider isolated `get` operations. It is straightforward to verify that the Blocks-world program maintains $\mathcal{B}$ as an invariant. □

Given an invariant $\mathcal{I}$, we define confluence with respect to $\mathcal{I}$ as follows:

Definition 11 (Observable Confluence). *A CHR program* P *is* $\mathcal{I}$-confluent *with respect to invariant* $\mathcal{I}$ *if the following holds for all states* σ_0*,* σ_1 *and* σ_2 *where* $\mathcal{I}(\sigma_0)$ *holds: If* $\sigma_0 \rightarrowtail^* \sigma_1$ *and* $\sigma_0 \rightarrowtail^* \sigma_2$ *then* σ_1 *and* σ_2 *are joinable.* □

Alternatively, a CHR program P is *$\mathcal{I}$-confluent* with respect to invariant $\mathcal{I}$ iff the reduction system $\mathcal{R} = \langle \{\sigma \in \Sigma | \mathcal{I}(\sigma)\}, \rightarrowtail \rangle$ is confluent. Likewise, P is *$\mathcal{I}$-local-confluent* iff $\mathcal{R}$ is local-confluent and P is *$\mathcal{I}$-terminating* iff $\mathcal{R}$ is terminating.

Observable confluence is a weaker form of confluence,[8] thus the standard confluence test (see Theorem 1) is too strong. We desire a more general test for observable confluence.

4 Observable Confluence

4.1 Extensions

To help reduce the level of verbosity, we introduce the notion of a state *extension*.

[7] The terminology "$\mathcal{I}$-confluence" and "observable confluence" are largely interchangeable. The latter is useful when referring to a specific invariant $\mathcal{I}$.

[8] Observable confluence is only strictly weaker if $\mathcal{I} \neq true$.

Definition 12 (Extension). *A state* $\sigma = \langle G, S, B, \mathcal{T}, \mathcal{V}\rangle$ *can be* extended *by another state* $\sigma_e = \langle G_e, S_e, B_e, \mathcal{T}_e, \mathcal{V}_e\rangle$ *as follows*

$$\sigma \oplus \sigma_e = \langle G \uplus G_e, S \uplus S_e, B \wedge B_e, \mathcal{T} \uplus \mathcal{T}_e, \mathcal{V}_e\rangle$$

We say that σ_e *is an* extension *of* σ. □

An *extension* σ_e adds some "extra" information to an existing state σ. Notice that the variables of interest $\mathcal{V}$ in the original state σ are simply replaced by variables of interest $\mathcal{V}_e$ from state σ_e. We also assume that $\approx$ (see Definition 5) is the equivalence relation for extensions.

Example 2. The following equations are of the form $\sigma \oplus \sigma_e = \sigma'$ where σ and σ' are states, and σ_e is an extension.

$$\langle \emptyset, \{p(X)\}, true, \emptyset, \emptyset\rangle \oplus \langle \emptyset, \{q(X)\}, true, \emptyset, \emptyset\rangle = \langle \emptyset, \{p(X), q(X)\}, true, \emptyset, \emptyset\rangle$$
$$\langle \emptyset, \{p(X)\}, true, \emptyset, \emptyset\rangle \oplus \langle \emptyset, \emptyset, X = 0, \emptyset, \emptyset\rangle = \langle \emptyset, \{p(X)\}, X = 0, \emptyset, \emptyset\rangle$$
$$\langle \emptyset, \{p(X)\}, true, \emptyset, \emptyset\rangle \oplus \langle \emptyset, \emptyset, true, \emptyset, \{X\}\rangle = \langle \emptyset, \{p(X)\}, true, \emptyset, \{X\}\rangle$$

The first adds a user constraint $q(X)$ to the user store, the second adds a built-in constraint $X = 0$ to the built-in store, and the third replaces the variables of interest with the set $\{X\}$. □

One crucial property of extensions is that they do not affect the applicability of the CHR rewrite relation $\rightarrowtail$.

Lemma 1. *For all states* σ *and* σ_1 *such that* $\sigma \rightarrowtail^* \sigma_1$*, and for all extensions* σ_e *have that* $\sigma \oplus \sigma_e \rightarrowtail^* \sigma_1 \oplus \sigma_e$.

The notions of *variance* and *joinability* depend on the variables of interest $\mathcal{V}$. Therefore we must refine the definition of extension to ensure joinability is preserved.

Definition 13 (Valid Extension). *A* valid extension $\sigma_e = \langle G_e, S_e, B_e, \mathcal{T}_e, \mathcal{V}_e\rangle$ *of a state* $\sigma = \langle G, S, B, \mathcal{T}, \mathcal{V}\rangle$ *is an extension such that*

$$v \in vars(G, S, B, \mathcal{T}) \wedge v \notin \mathcal{V} \Rightarrow v \notin vars(G_e, S_e, B_e, \mathcal{T}_e, \mathcal{V}_e)$$

Example 3. Consider the state $\sigma = \langle \emptyset, \{leq(X, Y)\}, X = Y, \emptyset, \{X\}\rangle$. Then $\sigma_e = \langle \{leq(X, Z)\}, \emptyset, true, \emptyset, \{X\}\rangle$ is a valid extension of σ. However, the extension $\sigma'_e = \langle \{leq(Y, Z)\}, \emptyset, true, \emptyset, \{X\}\rangle$ is invalid since local variable Y is mentioned in the extension. □

For valid extensions, joinability is preserved.

Lemma 2. *For all states* $\sigma = \langle G, S, B, \mathcal{T}, \mathcal{V}\rangle$*,* σ_1*, and* σ_2 *such that*

$$\sigma \rightarrowtail^* \sigma_1 \quad \textit{and} \quad \sigma \rightarrowtail^* \sigma_2$$

If $\sigma_1 \downarrow \sigma_2$*, then for all valid extensions* σ_e *we have that* $\sigma_1 \oplus \sigma_e \downarrow \sigma_2 \oplus \sigma_e$.

4.2 $\mathcal{I}$-Confluence

If all variables in a state $\sigma = \langle G, S, B, \mathcal{T}, \mathcal{V}\rangle$ are in $\mathcal{V}$, i.e. $vars(G, S, B, \mathcal{T}) \subseteq \mathcal{V}$, then all extensions are valid for σ. The ancestor state of a critical pair has this property, thus proving $\mathcal{I}$-confluence is equivalent to showing that for all critical pairs (σ_A, σ_B) with ancestor state $\sigma_{\mathcal{CP}}$, and all extensions σ_e such that $\mathcal{I}(\sigma_{\mathcal{CP}} \oplus \sigma_e)$ holds, then $(\sigma_A \oplus \sigma_e, \sigma_B \oplus \sigma_e)$ are joinable. The problem is that the set of all extensions is infinite, so we need some way of reducing the number of extensions that we must test.

Our strategy is to define a partial order[9] $\preceq_\sigma$ over valid extensions that satisfy the invariant with respect to some state σ.

Definition 14 (Partial Order). *Given a state $\sigma = \langle G, S, B, \mathcal{T}, \mathcal{V}\rangle$, and valid extensions σ_{e1} and σ_{e2} of σ, then we define $\sigma_{e1} \preceq_\sigma \sigma_{e2}$ to hold if*

1. *there exists a valid extension σ_{e3} of $(\sigma \oplus \sigma_{e1})$ such that $(\sigma \oplus \sigma_{e1}) \oplus \sigma_{e3} \approx \sigma \oplus \sigma_{e2}$*
2. *$\mathcal{V} - \mathcal{V}_{e1} \subseteq \mathcal{V} - \mathcal{V}_{e2}$ holds.* □

We find that if $\sigma_{e1} \preceq_\sigma \sigma_{e2}$, $\sigma \rightarrowtail \sigma_1$, and $\sigma \rightarrowtail \sigma_2$, then $(\sigma_1 \oplus \sigma_{e1} \downarrow \sigma_2 \oplus \sigma_{e1})$ implies $(\sigma_1 \oplus \sigma_{e2} \downarrow \sigma_2 \oplus \sigma_{e2})$. The means that the $\preceq_\sigma$ order respects joinability, and thus reduces the number of states that must be tested in order to prove confluence.

We define the following for notational convenience.

Definition 15. *Let $\Sigma_e(\sigma)$ be the set of all valid extensions of some state σ, and let $\Sigma_e^{\mathcal{I}}(\sigma) = \{\sigma_e | \sigma_e \in \Sigma_e(\sigma) \wedge \mathcal{I}(\sigma \oplus \sigma_e)\}$ be the set of all valid extensions satisfying the invariant $\mathcal{I}$. Finally, let $\mathcal{M}_e^{\mathcal{I}}(\sigma)$ be the $\prec_\sigma$-minimal elements of $\Sigma_e^{\mathcal{I}}(\sigma)$.* □

We define the following property, which we show to be equivalent to $\mathcal{I}$-local-confluence.

Definition 16. *A program P is* minimal extension joinable *if for all critical pairs $\mathcal{CP} = (\sigma_1, \sigma_2)$ with ancestor state $\sigma_{\mathcal{CP}}$, and for all $\sigma_e \in \mathcal{M}_e^{\mathcal{I}}(\sigma_{\mathcal{CP}})$, we have that $(\sigma_1 \oplus \sigma_e, \sigma_2 \oplus \sigma_e)$ is joinable.*

Lemma 3. *Given that $\prec_{\sigma_{\mathcal{CP}}}$ is well-founded for all critical pairs $\mathcal{CP}$, then: P is $\mathcal{I}$-local-confluent iff P is minimal extension joinable.*

For terminating programs, we invoke Newman's Lemma [6] to show that $\mathcal{I}$-local-confluence implies $\mathcal{I}$-confluence.

Theorem 2. *For all $\mathcal{I}$-terminating programs P, given that $\prec_{\sigma_{\mathcal{CP}}}$ is well-founded for all critical pairs $\mathcal{CP}$, then: P is $\mathcal{I}$-confluent iff P is minimal extension joinable.*

[9] Although we believe that relation $\preceq_\sigma$ is a partial order, we omit a formal discussion since none of our theoretical results require it to be.

4.3 $\mathcal{I}$-Confluence Test

The standard confluence test for terminating CHR programs relies on showing that all critical pairs are joinable. Based on Theorem 2, we can define a similar test for the more general notion of $\mathcal{I}$-confluence. Instead of testing critical pairs, we test critical pairs $\mathcal{CP}$ extended by the $\mathcal{M}_e^{\mathcal{I}}(\sigma_{\mathcal{CP}})$ extension set.

For Theorem 2 to be used in practice, there are two issues that must be resolved: (1) the order $\prec_{\sigma_{\mathcal{CP}}}$ must be *well-founded*, and (2) for each critical pair $\mathcal{CP}$, the set of extensions $\mathcal{M}_e^{\mathcal{I}}(\sigma_{\mathcal{CP}})$ must be *computable*.

Well-foundedness. Ordering $\prec_{\sigma_{\mathcal{CP}}}$ is essentially a product order over the fields in the CHR state. Thus for the G, S, $\mathcal{T}$ fields of a state, $\prec_{\sigma_{\mathcal{CP}}}$ is the well-founded subset ordering with the minimal element $G = S = \mathcal{T} = \emptyset$. The set of variables of interest $\mathcal{V}$ is ordered differently. In this case, extensions are ordered based on the *difference* between $\mathcal{V}$ and some given reference set $\mathcal{V}_0$. Again, this is (a variant of) subset ordering with the minimal element $\mathcal{V} = \mathcal{V}_0$.

Where well-foundedness may be broken is the built-in store B. Indeed, for some constraint domains, the set of extensions is not well-founded.

Example 4. Consider the constraint domain $\mathcal{D}$ of (in)equalities over the integers. Consider the following sequence of extensions: $\sigma_e^i = \langle \emptyset, \emptyset, X < i, \emptyset, \{X\} \rangle$. Since for all j, k such that $k > j$ we have that $\mathcal{D} \models X < j \leftrightarrow (X < k \wedge X < j)$ we have that $\sigma_e^j \prec_\sigma \sigma_e^k$ holds. Since the sequence is infinite, the relation $\prec_\sigma$ is not well-founded. □

There are also important examples of constraint domains that do preserve well-foundedness:

Proposition 1. *The order $\prec_\sigma$ is well-founded for all σ if $\mathcal{D}$ is equations over the Herbrand domain.*

Proposition 2. *The order $\prec_\sigma$ is well-founded for all σ if $\mathcal{D}$ is a finite domain.*

We can use Proposition 2 to find a practical solution to Example 4. Instead of considering all possible integers, we can restrict ourselves to some finite range of integers (e.g. those representable on a 32-bit CPU). The example is now well founded, with the minimal element $\langle \emptyset, \emptyset, X < 2^{32}, \emptyset, \{X\} \rangle$.

Computability. Depending on the invariant $\mathcal{I}$, the set $\mathcal{M}_e^{\mathcal{I}}(\sigma_{\mathcal{CP}})$ may be either *undecidable* or be *infinite*. Even if $\mathcal{M}_e^{\mathcal{I}}(\sigma_{\mathcal{CP}})$ is decidable and finite, an algorithm to compute it is dependent on the nature of $\mathcal{I}$. The computation of $\mathcal{M}_e^{\mathcal{I}}(\sigma_{\mathcal{CP}})$ is therefore instance-dependent.

Despite this, in Section 5 we look at several instances for $\mathcal{I}$ and compute the $\mathcal{M}_e^{\mathcal{I}}(\sigma_{\mathcal{CP}})$ for each critical pair.

5 Examples

We use Theorem 2 to verify observable confluence under the invariants we have seen earlier in Section 2. In addition, we verify ground confluence.

5.1 Reachable Confluence

A naive definition of confluence states that: a program P is confluent if for all input I, there is only one possible O such that $I \rightarrowtail^* O$. However, in [2] it was shown that there exist non-confluent CHR programs that satisfy this alternative definition. In this section, we reformulate the main theorem from [2] in terms of our observable confluence results.

The key issue is the difference between *reachable* and *unreachable* states. A *reachable* state is one that can be derived from some initial state (i.e. from some initial goal).

Definition 17 (Reachability). *We define the property that a state is* reachable $R(\sigma)$ *as follows:*

- *For all initial states* $\sigma_i = \langle G, \emptyset, true, \emptyset, vars(G)\rangle$ $R(\sigma_i)$ *holds; and*
- *If* $\sigma_1 \rightarrowtail \sigma_2$ *(or* $\sigma_1 \approx \sigma_2$*) and* $R(\sigma_1)$ *holds, then* $R(\sigma_2)$ *holds.* □

By definition, $R(\sigma)$ is an invariant.

The naive definition of confluence is more precisely defined as R-confluence, i.e. confluence with respect to the reachability invariant. In some systems, e.g. in term rewriting, all states (terms) are potential initial states, and thus R-confluence and confluence are equivalent. However, as was show in Section 2.3, the same is not true for CHR. Our main counter-example is the following class of CHR programs, which arise from the study of multi-parameter typeclasses with functional dependencies [10].

Definition 18 (FD-CHR). *A CHR program P is said to be in the* FD-CHR *class of programs if it is of the form*

$$\texttt{r1 @ p}(X_1,...,X_d,X_{d+1},...,X_r,...)\texttt{, p}(X_1,...,X_d,Y_{d+1},...,Y_r,...)\texttt{ ==>}\ X_{d+1}=Y_{d+1},...,X_r=Y_r\,.$$
$$\texttt{r2 @ p}(f_1,...,f_n)\texttt{ <=> }B\,.$$
$$\texttt{r3 @ p}(f_1,...,f_d,Y_1,...,Y_r,...)\texttt{ ==> }Y_1=f_{d+1},...,Y_r=f_r\,.$$

where B is an arbitrary conjunction of built-in and user constraints, and f_i are arbitrary terms such that $vars(f_{d+1},...,f_r) \subseteq vars(f_1,...,f_d)$ We also require P to be terminating. Here the indices $1..d$ represent the domain *and indices $(d+1)..r$ represent the* range *of the functional dependency. Also note that r is allowed to be less than n.* □

In [2] it was shown that the *FD-CHR* class of programs are R-confluent, however many instances of Definition 18 are not confluent.

Example 5 (FD-CHR). Consider the following instance of Definition 18:[10]

```
r1 @ f(A,B1,C), f(A,B2,D) ==> B1 = B2.
r2 @ f(int,bool,float)    <=> true.
r3 @ f(int,B,C)           ==> B = bool.
```

[10] An informal version of this example was seen in Section 2.3.

Consider the critical pair (σ_1, σ_2) between rules `r1` and `r2`:

$$\sigma_{\mathcal{CP}} = \langle \emptyset, \{f(int, bool, float), f(int, B2, D)\}, true, \{t\}, \{B2, D\}\rangle$$
$$\sigma_1 = \langle \{bool = B2\}, \{f(int, bool, float), f(int, B2, D)\}, true, \emptyset, \{B2, D\}\rangle$$
$$\sigma_2 = \langle \emptyset, \{f(int, B2, D)\}, true, \{t\}, \{B2, D\}\rangle$$

where t is the token $(r1@f(int, bool, float), f(int, B2, D))$. The final states derived from σ_1 and σ_2 are:

$$\sigma_1 \rightarrowtail^* \langle \emptyset, \{f(int, bool, D)\}, B2 = bool, \emptyset, \{B2, D\}\rangle$$
$$\sigma_2 \rightarrowtail^* \langle \emptyset, \{f(int, B2, D)\}, true, \{t\}, \{B2, D\}\rangle$$

These states are not variants. In the final state for σ_1, the variable $B2$ is constrained to *bool*, but this is not the case for the final state for σ_2. Thus the critical pair is not joinable, and the program is not confluent. □

State $\sigma_{\mathcal{CP}}$ is not reachable, since the lack of a token $(r3@f(int, B2, D))$ suggests rule `r3` has already fired on constraint $f(int, B2, D)$. If that rule did fire, then we would expect the built-in store to entail $B2 = bool$, which is not the case.

Thus, we consider the minimal set of extensions that make $\sigma_{\mathcal{CP}}$ reachable. This set is:

$$\mathcal{M}_e^R(\sigma_{\mathcal{CP}}) = \{\langle \emptyset, \emptyset, true, \{(r3@f(int, B2, D))\}, \mathcal{V}\rangle, \langle \{B2 = bool\}, \emptyset, true, \emptyset, \mathcal{V}\rangle, \langle \emptyset, \emptyset, B2 = bool, \emptyset, \mathcal{V}\rangle\}$$

It is easy to verify that for all $\sigma_e \in \mathcal{M}_e^R(\sigma_{\mathcal{CP}})$ we have that $\sigma_1 \oplus \sigma_e \downarrow \sigma_2 \oplus \sigma_e$. We can verify similar results for all other critical pairs in P, and thus, by Theorem 2, program P is R-confluent.

We can generalise this basic approach, and restate the main theorem from [2].

Corollary 1. *All programs $P \in$FD-CHR are R-confluent.*

The alternative proof for Corollary 1 in [2] relied on showing that all programs $P \in$*FD-CHR* were related to a class of confluent programs, and that the relation was sufficient to show R-confluence. In this paper, the proof relies on Theorem 2, and thus is a far more direct proof of R-confluence.

5.2 Simple Confluence

It is common for programmers to implement non-confluent CHR programs that are well behaved for some certain input. For example, the union-find program [8] (also see Section 2.2) is non-confluent, however it is well behaved provided the initial goal satisfies some certain conditions.

Let $\mathcal{I}$ be an invariant that simply excludes non-joinable critical pairs from consideration, then P is always $\mathcal{I}$-confluent. We define this as *simple confluence*.

Corollary 2 (Simple Confluence). *Given an invariant $\mathcal{I}$ and an $\mathcal{I}$-terminating program P such that $\prec_\sigma$ is well-founded for all σ, then P is $\mathcal{I}$-confluent if for all critical pairs $\mathcal{CP} = (\sigma_1, \sigma_2)$, either:*

1. *$\mathcal{I}(\sigma_{\mathcal{CP}})$ holds, and $\sigma_1 \downarrow \sigma_2$; or*
2. *For all extensions σ_e we have that $\mathcal{I}(\sigma_{\mathcal{CP}} \oplus \sigma_e)$ does not hold.*

Via the above corollary and the blocks world invariant $\mathcal{B}$ from Example 1, we can straightforwardly verify $\mathcal{B}$-confluence of the blocks world program from Section 2.1. Similarly, we can verify observable confluence of the union-find algorithm in Section 2.2.

5.3 Ground Confluence

A state σ is *ground*, i.e. $\mathcal{G}(\sigma)$ holds, if all variables $vars(\sigma)$ are constrained to be one value by the built-in store B of σ. Groundness is an invariant for *range restricted*[11] CHR programs. Typically, the critical pair between two rules is not ground, however we can invoke Theorem 2 to show $\mathcal{G}$-confluence.

Corollary 3 (Ground Confluence). *Given a $\mathcal{G}$-terminating, range restricted program P such that $\prec_\sigma$ is well-founded for all σ, then P is $\mathcal{G}$-confluent if for all critical pairs $\mathcal{CP} = (\sigma_1, \sigma_2)$ we have that $(\sigma_1 \oplus \sigma_e) \downarrow (\sigma_2 \oplus \sigma_e)$ for all extensions $\sigma_e \in \mathcal{M}(\sigma_{\mathcal{CP}})$ where:*

$$\mathcal{M}(\sigma_{\mathcal{CP}}) = \{\langle \emptyset, \emptyset, X_0 = d_0 \wedge ... \wedge X_n = d_m, \emptyset, \mathcal{V}_{\mathcal{CP}} \rangle \mid \{X_0, ..., X_n\} = vars(\sigma_{\mathcal{CP}}), d_i \in \mathcal{D}\}$$

If $\mathcal{D}$ is an finite set, then $\mathcal{M}(\sigma_{\mathcal{CP}})$ can be computed.

Example 6. Consider the following CHR program over the Boolean domain.

```
p(X,Y) <=> not(X,Y).          xor(0,0,Z) <=> Z = 0.
p(X,Y) <=> xor(1,X,Y).        xor(0,1,Z) <=> Z = 1.
not(0,Y) <=> Y = 1.           xor(1,0,Z) <=> Z = 1.
not(1,Y) <=> Y = 0.           xor(1,1,Z) <=> Z = 0.
```

The critical state $\sigma_{\mathcal{CP}} = \langle \emptyset, \{p(X,Y)\}, true, \emptyset, \{X, Y\} \rangle$ between the first two rules is non-joinable, hence the program is non-confluent. Clearly $\mathcal{G}(\sigma_{\mathcal{CP}})$ does not hold, thus we evaluate $\mathcal{M}(\sigma_{\mathcal{CP}})$:

$$\mathcal{M}(\sigma_{\mathcal{CP}}) = \{\langle \emptyset, \emptyset, X = 0 \wedge Y = 0, \emptyset, \{X, Y\} \rangle, \langle \emptyset, \emptyset, X = 0 \wedge Y = 1, \emptyset, \{X, Y\} \rangle, \langle \emptyset, \emptyset, X = 1 \wedge Y = 0, \emptyset, \{X, Y\} \rangle, \langle \emptyset, \emptyset, X = 1 \wedge Y = 1, \emptyset, \{X, Y\} \rangle\}$$

For each of these extensions, the critical pair is joinable, and thus P is $\mathcal{G}$-confluent. □

[11] A CHR program is *range restricted* if $vars(H_1 \backslash H_2 \iff G|B) = vars(H_1 \wedge H_2)$ for all rules.

6 Conclusion

We have shown that many non-confluent CHR programs are in fact observably confluent in practice, and have presented a method for proving the observable confluence of programs with respect to invariants. Furthermore, we have specialised our results for some common cases, such as simple confluence and ground confluence.

To the best of our knowledge, we are the first to study observable confluence in the context of a rule-based language. However, the notion of observable confluence could easily be extended to other areas, such as term rewriting, which is something we intend to investigate in the future.

References

1. Abdennadher, S.: Operational semantics and confluence of constraint propagation rules. In: Smolka, G. (ed.) Principles and Practice of Constraint Programming - CP97. LNCS, vol. 1330, pp. 252–266. Springer, Heidelberg (1997)
2. Duck, G.J., Stuckey, P.J., Sulzmann, M.: Observable Confluence for Constraint Handling Rules. Technical Report CW 452, Katholieke Universteit Leuven. In: Proc. of CHR 2006, Third Workshop on Constraint Handling Rules (2006)
3. Frühwirth, T.: Constraint handling rules. In: Constraint Programming: Basics and Trends. LNCS, Springer, Heidelberg (1995)
4. Jones, M.P.: Type classes with functional dependencies. In: Smolka, G. (ed.) ESOP 2000 and ETAPS 2000. LNCS, vol. 1782, Springer, Heidelberg (2000)
5. Lam, E.S.L., Sulzmann, M.: Towards agent programming in CHR. Technical Report CW 452, Katholieke Universteit Leuven. In: Proc. of CHR 2006, Third Workshop on Constraint Handling Rules (2006)
6. Newman, M.H.A.: On theories with a combinatorial definition of equivalence. Annals of Mathematics 43(2), 223–243 (1942)
7. Peyton Jones, S. (ed.): Haskell 98 Language and Libraries: The Revised Report. Cambridge University Press, Cambridge (2003)
8. Schrijvers, T., Frühwirth, T.W.: Analysing the CHR Implementation of Union-Find. In: Wolf, A., Frühwirth, T.W., Meister, M. (eds.) W(C)LP. Ulmer Informatik-Berichte, Universität Ulm, Germany, vol. 2005-01, pp. 135–146 (2005)
9. Schrijvers, T., Frühwirth, T.W.: Optimal union-find in Constraint Handling Rules. TPLP 6(1-2), 213–224 (2006)
10. Sulzmann, M., Duck, G.J., Peyton Jones, S., Stuckey, P.J.: Understanding Functional Dependencies via Constraint Handling Rules. Journal of Functional Programming 17(1), 83–129 (2007)

Graph Transformation Systems in CHR

Frank Raiser

Faculty of Engineering and Computer Sciences, University of Ulm, Germany
Frank.Raiser@uni-ulm.de

Abstract. In this paper we show it is possible to embed graph transformation systems (GTS) soundly and completely in constraint handling rules (CHR). We suggest an encoding for the graph production rules and we investigate its soundness and completeness by ensuring equivalence of rule applicability and results. We furthermore compare the notion of confluence in both systems and show how to adjust a standard CHR confluence check to work for an embedded GTS.

1 Introduction

Constraint handling rules (CHR) [1] allow for rapid prototyping of constraint-based algorithms. Besides constraint reasoning, CHR have been used for various tasks including theorem proving, parsing, and multi-set rewriting. In this work we show that CHR also provide the means for concise implementations of graph transformations.

Graph transformation systems are used to describe complex structures and systems in a concise, readable, and easily understandable way. They have applications ranging from implementations of programming languages over model transformations to graph based models of computation [2,3,4]. The principal idea of a graph transformation is to apply a production rule to a graph. A part of the so-called host graph has to be matched to the rule and is then modified accordingly.

While there are several specialized tools available for performing graph transformations, it is usually cumbersome to combine them with general-purpose programming languages. Following the tradition of using CHR for rapid prototyping, we investigate the necessities for implementing a graph transformation system with CHR. We show that CHR are indeed a suitable choice for prototyping graph transformations, as every rule of a graph transformation system can be directly and intuitively translated into a corresponding CHR rule already yielding an executable graph transformation system. No effort has to be invested into creating an underlying transformation engine, as the CHR implementation in combination with an advantageous encoding of the rules is sufficient.

In order to arrive at an intuitive and concise encoding of graph production rules with CHR, we make restrictions to the graph transformation systems. These restrictions consist in requiring injective matchings and inclusions in the graph production rules. However, these are common restrictions often found in practical applications of graph transformation systems [4,5].

V. Dahl and I. Niemelä (Eds.): ICLP 2007, LNCS 4670, pp. 240–254, 2007.

From a theoretical point of view we investigate the soundness and completeness of our proposed encoding by ensuring equivalence of applicability and results. Thus we ensure that such a CHR rule is applicable if and only if the corresponding graph production rule is applicable. We further show that applying a constraint handling rule in this context results in a state encoding the graph obtained by applying the corresponding graph production rule in a graph transformation system and vice versa.

Finally, we compare the notion of confluence in both systems. We particularly concentrate on the investigation of critical pairs which is the basis of confluence checking. A slightly changed definition of CHR confluence is presented that allows the application of existing confluence checkers to a graph transformation system embedded in CHR.

This work is divided into six sections: We begin with the introduction of the necessary notions of graph transformation systems and CHR in Section 2. Section 3 then presents our proposed encoding of a GTS in CHR, the properties of which we analyze in Section 4. Section 5 compares the notion of confluence in both systems before we conclude in Section 6.

2 Preliminaries

In this section we introduce the required formalisms for graph transformation systems and constraint handling rules.

2.1 Graph Transformation System (GTS)

The following definitions for graphs and graph transformation systems have been adapted from [2].

Definition 1 (type graph, typed graph). *A* graph $G = (V, E, \mathrm{src}, \mathrm{tgt})$ *consists of a set* V *of nodes, a set* E *of edges and two morphisms* $\mathrm{src}, \mathrm{tgt} : E \to V$ *specifying source and target of an edge, respectively. A* type graph TG *is a graph with unique labels for all nodes and edges.*

For the purpose of simplicity, we avoid an additional label morphism in favor of identifying variable names with their labels. For multiple graphs we refer to the node set V *of a graph* G *as* V_G *and analogously for edge sets and the* $\mathrm{src}, \mathrm{tgt}$ *morphisms. We further define the degree of a node as* $\deg : V \to \mathbb{N}, v \mapsto \#\{e \in E \mid \mathrm{src}(e) = v\} + \#\{e \in E \mid \mathrm{tgt}(e) = v\}$.

A typed graph G *is a tuple* $(V, E, \mathrm{src}, \mathrm{tgt}, \mathrm{type}, TG)$ *where* $(V, E, \mathrm{src}, \mathrm{tgt})$ *is a graph,* TG *a type graph, and* type *a morphism with* $\mathrm{type} = (\mathrm{type}_V, \mathrm{type}_E)$ *and* $\mathrm{type}_V : V \to TG_V, \mathrm{type}_E : E \to TG_E$. *The* type *morphism is a graph morphism, therefore it has to satisfy the following condition:* $\forall e \in E : \mathrm{type}_V(\mathrm{src}(e)) = \mathrm{src}_{TG}(\mathrm{type}_E(e)) \wedge \mathrm{type}_V(\mathrm{tgt}(e)) = \mathrm{tgt}_{TG}(\mathrm{type}_E(e))$.

Example 1. Figure 1 shows an example for a type graph and a corresponding typed graph. The type graph is used to define two types of nodes: processes and resources. Furthermore it allows *use* edges going from processes to resources.

The typed graph is one possible instance of a graph modelling processes and resources being used by those processes. The type morphism is represented by the dotted lines, showing how the nodes are typed as processes or resources, respectively.

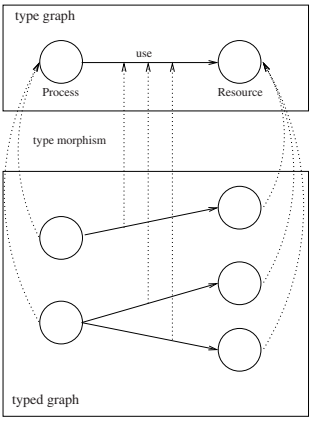


Fig. 1. Type graph and typed graph

Definition 2 (GTS, rule). *A* Graph Transformation System *(GTS) is a tuple consisting of a type graph and a set of graph production rules. A* graph production rule *– also simply called* rule *if the context is clear – is a tuple* $p = (L \xleftarrow{l} K \xrightarrow{r} R)$ *of graphs with inclusion morphisms* $l : K \to L$ *and* $r : K \to R$.

We distinguish two kinds of typed graphs: *rule graphs* and *host graphs*. Rule graphs are the graphs L, K, R of a graph production rule p and host graphs are graphs to which the graph production rules can be applied. We furthermore make use of graph transformations based on the double-pushout approach as defined in [2]. Most notably, we require a so-called match morphism $m : L \to G$ to apply a rule p to a typed host graph G. The transformation yielding the typed graph H is written as $G \overset{p,m}{\Longrightarrow} H$. For our work, we restrict the possible match morphisms to injective morphisms which is a common restriction [4,5].

A graph production rule p can only be applied to a host graph G if the gluing condition [2] is satisfied, which in our case of injective match morphisms is simplified such that p can be applied as long as there are no dangling edges. Section 3.1 explains the notion of dangling edges and how they are handled in our encoding.

As an example, we provide an implementation for recognizing cyclic lists, which is presented in [4].

Example 2. The GTS for recognizing cyclic lists consists of two rules depicted in Figure 2. The rule *unlink* shortens a list consisting of three linearly connected

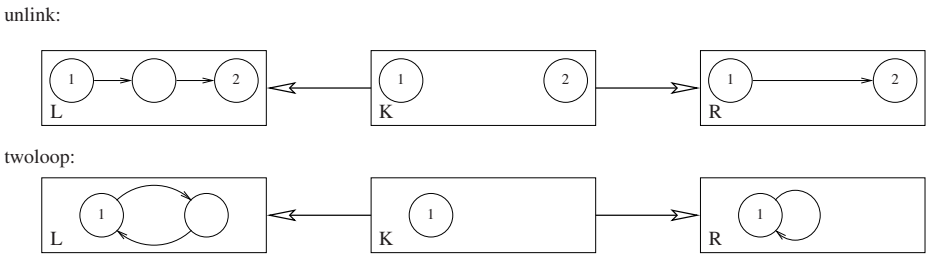


Fig. 2. Graph transformation system for recognizing cyclic lists

nodes by removing the intermediate node. For a cyclic list, this rule can be applied up to the point where only two nodes are left. In that case, the rule *twoloop* transforms those two nodes into the graph consisting of a single node with a loop. See [4] for a more thorough discussion of this example.

Note that we use a shorthand notation in Figure 2 which only shows the morphisms l and r implicitly by the labels of the nodes which are mapped onto each other. Nodes and edges which are removed or added in the graphs L or R are not labelled, as there is no node or edge in K which is mapped to them.

2.2 Constraint Handling Rules (CHR)

This section presents the syntax and operational semantics of Constraint Handling Rules [1,6]. Constraints are first-order predicates which we separate into *built-in constraints* and *user-defined constraints*. Built-in constraints are provided by the constraint solver while user-defined constraints are defined by a CHR program. For our purpose we only need a subset of CHR, namely *simplification rules*. Simplification rules are of the form

$$Rulename \ @ \ H_1, \ldots, H_i \Leftrightarrow G_1, \ldots, G_j \mid B_1, \ldots, B_k$$

where *Rulename* is a unique identifier of a rule, the head $H = H_1, \ldots, H_i$ is a non-empty conjunction of user-defined constraints, the guard $\overline{G} = G_1, \ldots, G_j$ is a conjunction of built-in constraints and the body $B = B_1, \ldots, B_k$ is a conjunction of built-in and user-defined constraints.

The operational semantics of a simplification rule is based on an underlying constraint theory CT for the built-in constraints and a state, which is a pair $\langle G, C\rangle$ where G is a goal, i.e. a conjunction of user-defined and built-in constraints, and C is a (built-in) constraint [6]. A simplification rule of the form $H \Leftrightarrow \overline{G} \mid B$ is applicable to a state $\langle E \wedge G, C\rangle$ if $CT \models \forall(C \rightarrow \exists\overline{x}(H \doteq E \wedge \overline{G}))$ where $\overline{x}$ are the variables in H. We define the following state transition for the application: $\langle E \wedge G, C\rangle \mapsto \langle B \wedge G, (H \doteq E) \wedge C \wedge \overline{G}\rangle$.

For the remainder of this work, we can further restrict these rules, as there is no need for the guard constraints, such that a CHR simplification rule is simply of the form

$$Rulename \ @ \ H \Leftrightarrow B.$$

3 Representation of Graphs in CHR

In order to embed a GTS in CHR, we have to encode the available graph production rules as simplification rules and provide a conjunction of goal constraints corresponding to the host graph. To this end we provide a bijective correspondence between graphs and their representation through CHR constraints given by the following constructions.

At first we have to find out what constraints will be needed for encoding the rules and host graph. At this point we require the GTS to be typed, as we can directly defer the necessary constraints from the corresponding type graph as explained in Definition 3. Note that this is a very weak restriction though, as every untyped graph can be typed over the trivial type graph consisting of a single node with a loop.

Definition 3 (type graph encoding). *For a type graph TG we define the following constraints to encode graphs typed over TG:*

- *Introduce a constraint* $\mathrm{degree}/2$.
- $\forall v \in V_{TG}$ *introduce a constraint* $v/1$.
- $\forall e \in E_{TG}$ *introduce a constraint* $e/3$.

We assume all nodes and edges of the type graph TG to be uniquely labelled such that the introduced constraints have unique names as well. The $\mathrm{degree}/2$ constraint is a special constraint we require to be able to check for dangling edges as is explained in Section 3.1.

Definition 4 (typed graph encoding). *A typed graph G based on a type graph TG is encoded with constraints as follows:*

- $\forall v \in V_G$ *with* $\mathrm{type}(v) = t$ *add the constraint* $t(T)$ *with* T *being a new variable.*
- $\forall e \in E_G$ *with* $\mathrm{type}(e) = t, \mathrm{src}(e) = v_1, \mathrm{tgt}(e) = v_2, \mathrm{type}(v_1) = t', \mathrm{type}(v_2) = t''$ *and previously created constraints* $t'(T^1), t''(T^2)$ *add another constraint* $t(E, T^1, T^2)$ *with* E *being a new variable.*
- *If G is a typed host graph the node degrees are known and thus* $\forall v \in V_G$ *with* $deg(v) = k, \mathrm{type}(v) = t$ *which add* $t(T)$ *we also add* $\mathrm{degree}(T, k)$.

A typed rule graph G encoded like this is denoted as $\mathrm{encode}(G)$.

Example 3 (cont). For our example of the GTS for recognizing cyclic lists we assume the trivial type graph consisting only of a node and a loop. Therefore every node in the typed graph has the same type, just like every edge has the same type. Based on this type graph we need the following constraints: $\mathrm{degree}/2, \mathrm{node}/1, \mathrm{edge}/3$.

A host graph which contains a simple cyclic list consisting of exactly two nodes is encoded as follows:
$\mathrm{node}(N_1) \wedge \mathrm{node}(N_2) \wedge \mathrm{edge}(E_1, N_1, N_2) \wedge \mathrm{edge}(E_2, N_2, N_1) \wedge \mathrm{degree}(N_1, 2) \wedge \mathrm{degree}(N_2, 2)$.

We can now encode a complete graph production rule based on these definitions. There are several possible ways to do this in CHR. However, we restrict ourselves to a single simplification rule here, as it is a very intuitive encoding which is useful in the upcoming proofs. The remaining results of this paper can also be transferred to different CHR representations like a simpagation rule approach.

Definition 5 (GTS rule in CHR). *A graph production rule* $p = (L \xleftarrow{l} K \xrightarrow{r} R)$ *from a GTS is translated into a CHR simplification rule of the form* $p \mathbin{@} H \Leftrightarrow B$ *with* H, B *being conjunctions of constraints as follows:*

- $H = \text{encode}(L) \wedge D_1$
- $B = \text{encode}(R) \wedge D_2$
- D_1 *and* D_2 *being conjunctions of* degree *constraints as follows:*
  - $\forall v \in L \setminus K$ *with* $\deg(v) = k$, $\text{type}(v) = t$, *and* $t(T) \in \text{encode}(L)$ *we have* $\text{degree}(T, k) \in D_1$.
  - $\forall v \in K$ *let* $n = \#\{e \in E_L \mid \text{src}(e) = v \vee \text{tgt}(e) = v\}$, $m = \#\{e \in E_R \mid \text{src}(e) = v \vee \text{tgt}(e) = v\}, type(v) = t, t(T) \in \text{encode}(K)$ *then* $\text{degree}(T, D) \in D_1$ *with* D *being an unused variable. Further add the constraint* $\text{degree}(T, D{+}m{-}n) \in D_2$.
  - $\forall v \in R \setminus K$ *with* $\deg(v) = k, type(v) = t$, *and* $t(T) \in \text{encode}(R)$ *let* $\text{degree}(T, k) \in D_2$.

We further require the encoding of K *found in* $\text{encode}(L)$ *and* $\text{encode}(R)$ *to share the same variables for the same nodes and edges as this implicitly models the inclusion morphisms analogously to the way we use node labels in Figure 2.*

A rule encoded like this is denoted as $\text{code}(p)$.

Example 4 (cont). As an example, consider the second rule from our example GTS, which reduces two cyclic nodes to a single node with a loop. Its encoding as a CHR simplification rule is given below:

$$\begin{aligned} \text{twoloop} \mathbin{@} \ & \text{node}(N_1), \text{node}(N_2), \\ & \text{edge}(E_1, N_1, N_2), \text{edge}(E_2, N_2, N_1), \\ & \text{degree}(N_2, 2), \text{degree}(N_1, D_1) \\ & \Leftrightarrow \\ & \text{node}(N_1), \text{edge}(E, N_1, N_1), \\ & \text{degree}(N_1, D_1{+}2{-}2). \end{aligned}$$

3.1 On Dangling Edges

A graph production rule can only be applied to a host graph when it is guaranteed that the result is consistent. While in the most general case of graph transformations there also exists an identification problem [2], our injective matchings allow for only one kind of inconsistency: When a node gets deleted by a rule, the corresponding node in the host graph may have edges adjacent to it which are not explicitly given in the rule. In such a case, the remaining edge would be left *dangling* as it is no longer adjacent to two nodes. Therefore this situation has to be avoided and before a rule is applied to a host graph, we first have to ensure that there are no dangling edges according to the following definition:

Definition 6 (dangling edge). *A* dangling edge *is an edge* $e \in E_G \setminus m(E_L)$ *such that there is a node* $v \in V_L \setminus V_K$ *with* $m(v) = \mathrm{src}(e) \vee m(v) = \mathrm{tgt}(e)$.

The degree /2 constraints introduced earlier are our means of detecting dangling edges: Let $e \in E_G$ be a dangling edge which is adjacent to $v_G \in V_G$, such that for $v \in V_L \setminus V_K : m(v) = v_G$. Due to Definition 5, the corresponding rule includes a degree(T, k) constraint for the node v with $k = \deg(v)$. This means that there are k edges adjacent to the node v in the rule graph. When matching this rule graph injectively to the host graph G we need to identify each of these edges with an edge in E_G adjacent to $m(v) = v_G$. By the definition of a dangling edge, the edge e is not among those k edges as $e \in E_G \setminus m(E_L)$. Therefore, we have $\deg(v_G) > k$. As G is a host graph and thus the degrees of nodes are known, there is a corresponding degree(T', l) constraint with $l > k$ for the node v_G. Here it can be seen clearly that a match between degree(T', l) and degree(T, k) is impossible due to $l > k$ and therefore the rule will not be applied as the dangling edge condition is violated.

Note that this check is only relevant to nodes which get removed by the graph production rule. If a node is in V_K, a dangling edge is not possible and thus for those nodes the rule contains degree(T, D) constraints which can always be matched due to D being a variable.

Example 5 (cont). Consider the *twoloop* rule given in Example 4 along with the following encoded host graph:

$$\mathrm{node}(A) \wedge \mathrm{node}(B) \wedge \mathrm{node}(C) \wedge \mathrm{edge}(E_1, A, B) \wedge \mathrm{edge}(E_2, B, A) \wedge \mathrm{edge}(E_3, B, C) \wedge \mathrm{degree}(A, 2) \wedge \mathrm{degree}(B, 3) \wedge \mathrm{degree}(C, 1)$$

Applying the *twoloop* rule to this graph to remove the node B would leave the edge E_3 going from B to C dangling. However, this is avoided as the encoding of the *twoloop* rule contains the following constraint in its head: degree$(N_2, 2)$. Only a node with a degree of exactly 2 can thus be removed by this rule, which rules out that the node B in the above host graph is removed. Nevertheless, the rule can be applied with $N_2 \doteq A$ as the node A has the necessary degree of 2.

3.2 Runtime Performance

After embedding graph transformations in CHR we are interested in the runtime performance needed to execute these transformations. In general the perfomance of a CHR program is determined by the time needed to find an applicable rule and the time needed to apply that rule.

As we have a one-to-one correspondence of CHR rules and graph transformation rules we know that executing D graph transformation steps is equal to applying D CHR rules in the corresponding CHR program. Given a host graph $G = (V, E, src, tgt)$ its encoding in CHR corresponding to Definition 4 uses $2|V| + |E|$ constraints. In general the search for an applicable rule needs to consider all possible combinations of these constraints for all constraints appearing in the head of a rule. Using the upper bound from [7] we get the runtime

complexity as $\mathcal{O}(D \cdot \sum_i c_{max}^{n_i} \cdot (O_{H_i} + O_{G_i}) + (O_{C_i} + O_{B_i}))$ where the sum iterates over all rules of the program, c_{max} is the maximal number of constraints present at any time during the computation, n_i is the number of head constraints in rule i, O_{H_i} is the time needed to unify the chosen goal constraints with the head constraints, O_{G_i} is the time needed to verify the guard of the rule, O_{C_i} is the time needed for adding the constraints on the right-hand side of the rule, and O_{B_i} is the time needed to remove the constraints on the left-hand side of the rule.

Note that in our embedding, CHR rules corresponding to graph transformation rules require no guards and thus $O_{G_i} = 0 \forall i$. Furthermore the time needed to add and remove constraints can be considered constant. For a worst-case analysis we can furthermore replace n_i with the $n = n_j$ of the rule j with the maximal number of head constraints. This yields a runtime complexity of $\mathcal{O}(D \cdot \sum_i c_{max}^n \cdot O_{H_i})$. The time required for unification with the head constraints can also be approximated by the rule with the largest number of head constraints. With m being the number of rules in the program we then get the complexity $\mathcal{O}(D \cdot m \cdot c_{max}^n \cdot O_{H_n})$.

Better bounds can be deferred by taking a closer look at the embedded GTS. For example if all rules have the property of not extending the graph's size, i.e. rules remove more edges and nodes than adding them, then the number of constraints is bounded by the number c_0 of initial constraints which encode the original host graph. Therefore the bound for such a GTS is $\mathcal{O}(D \cdot m \cdot c_0^n \cdot O_{H_n})$. Also note that actual CHR implementations use sophisticated methods to determine constraints for rule applications and thus the actual runtime may be considerably less.

Example 6. Consider the example GTS for recognizing cyclic lists from Figure 2. As it contains two rules which both remove a node and reduce the number of edges by one it is clear that the number of goal constraints monotonly decreases during execution. Therefore $c_{max} = c_0 = 2|V| + |E|$, i.e. the maximal number of constraints in a computation is bounded by the original host graph's encoding. Furthermore the number of rule applications is restricted by the number of nodes in the host graph, such that $D = |V|$. And as the two CHR rules are known we get $m = 2$ and $n = \max_i n_i = \max\{8, 6\} = 8$. Therefore runtime complexity is bounded by: $\mathcal{O}(|V| \cdot m \cdot (c_0)^8 \cdot O_{H_1})$. As $c_0 = 2|V| + |E|$ can be approximated by $|V|^2$ we get $\mathcal{O}(2|V| \cdot (|V|^2)^8 \cdot O_{H_1}) = \mathcal{O}(|V|^{17} \cdot O_{H_1})$. Note that this is a crude approximation and in practical settings runtime is much better depending on the chosen CHR implementation. Depending on the introduction order of the constraints, the GTS for recognizing cyclic lists can even execute with linear complexity.

4 Soundness and Completeness

After introducing our encoding for a GTS in CHR, we now investigate the properties of this encoding. First of all, we show that the CHR rules created for

a graph production rule are applicable if and only if the corresponding graph production rule is applicable. For a GTS one of the applicable rules is chosen nondeterministically. We get this exact behavior when using the standard semantics of CHR.

Lemma 1 (CHR applicability). *If* $\mathrm{code}(p)$ *is applicable, so is the corresponding graph production rule* $p = (L \xleftarrow{l} K \xrightarrow{r} R)$.

Proof. If the CHR rule is applicable, then there exist matching constraints for H *in the goal store. As no functions are used for the constraints in the goal store the matching only requires substitutions of the form* $[T/c]$ *or* $[T/T']$ *for variables* T, T' *and constant* c.

This CHR matching implies a match morphism m *for the graph production rule: Every node* $v \in V_L$ *(resp. edge* $e \in E_L$*) is represented as a constraint in* H *and there is a matching constraint* C' *which corresponds to a node* $v' \in V_G$ *(resp. edge* $e' \in E_G$*) such that we can define* $m(v) = v'$ *(resp.* $m(e) = e'$*). Due to the multi-set semantics of CHR, different nodes in* V_G *are encoded by different constraints and no single constraint can be matched to multiple head constraints. This property of CHR guarantees that* m *is injective.*

It remains to be shown that there are no dangling edges. Let us therefore assume that there is a dangling edge, i.e. there exists an edge $e \in E_G \setminus m(E_L)$ *with* $\mathrm{src}(e) = m(v) \vee \mathrm{tgt}(e) = m(v)$ *for a* $v \in L \setminus K$.

As $v \in L \setminus K$*, a constraint* $\mathrm{degree}(T, k)$ *is present in* H *according to Def. 5. Due to the CHR rule being applicable, this requires a matching constraint in the goal store. Therefore the node* $m(v)$ *has degree* k *as well. Due to* m *being an injective graph morphism, however, every edge adjacent to* v *is mapped to an edge adjacent to* $m(v)$*. Therefore the dangling edge* e *cannot exist.* □

Lemma 2 (graph rule applicability). *If* $p = (L \xleftarrow{l} K \xrightarrow{r} R)$ *is applicable, so is* $\mathrm{code}(p)$.

Proof. Let m *be the injective match morphism for the application of the graph production rule. Then* $\forall v \in V_L$ *with* $\mathrm{type}(v) = t$ *there is a constraint* $t(T)$ *in* H*. As* m *is a graph morphism, we have that* $\mathrm{type}(m(v)) = t$ *and thus a constraint* $t(T')$ *exists in the goal store corresponding to the node* $m(v)$*. The CHR rule is therefore applicable through a match with* $[T/T']$.

$\forall e \in E_L$ *with* $\mathrm{src}(e) = v_s, \mathrm{tgt}(e) = v_t, \mathrm{type}(e) = t_e, \mathrm{type}(v_s) = t_s, \mathrm{type}(v_t) = t_t$ *we have the following CHR rules in the conjunction* H*:* $t_s(T_1)$*,* $t_t(T_2)$*, and* $t_e(E, T_1, T_2)$*. Due to* m *being a graph morphism, there exist corresponding constraints* $t_s(T_1')$*,* $t_t(T_2')$ *for the nodes* $v_1 = m(v_s), v_2 = m(v_t)$ *in the goal. There further exists an edge* e' *with* $\mathrm{src}(e') = m(v_s), \mathrm{tgt}(e') = m(v_t), \mathrm{type}(e') = t_e$ *which is represented by a constraint* $t_e(E', T_1', T_2')$*. As discussed above, we already have* $[T_1/T_1'][T_2/T_2']$*, so we can extend this to* $[T_1/T_1'][T_2/T_2'][E/E']$ *for a CHR match.*

It remains to be shown that there is a match for the $\mathrm{degree}/2$ *constraints occurring in* H*:*

Case 1: a constraint of the form degree(T, D) with $[T/T']$ is always matchable, as the degree *constraint corresponding to the node represented by T' is guaranteed to be available in the goal store.*

Case 2: degree(T, k) *with $[T/T']$: Analogous to case 1, there exists a constraint* degree(T', l) *in the goal store. $k > l$ is a contradiction to m being an injective graph morphism. $k < l$ implies an edge $e' \notin m(E_L)$. However constraints of the form* degree(T, k) *are added only for nodes which are removed by the production rule and thus this is a dangling edge which contradicts the initial applicability of the graph production rule. Therefore $k = l$, which allows a CHR matching through $[T/T']$.* □

Theorem 1 (applicability). *A graph production rule $p = (L \xleftarrow{l} K \xrightarrow{r} R)$ is applicable to a typed host graph G if and only if* code(p) *is applicable to* encode(G).

Proof. This is immediate from the combination of Lemma 1 and Lemma 2. □

Theorem 2 (soundness and completeness). *Given a typed host graph G and* encode(G)*, a graph production rule $p = (L \xleftarrow{l} K \xrightarrow{r} R)$ represented by* code$(p) = H \Leftrightarrow B$*, and a match $m : L \to G$ the following transitions are equivalent:*

- $G \overset{p,m}{\Longrightarrow} \tilde{G}$
- $\langle \text{encode}(G), true \rangle \;\mapsto\; \langle \text{encode}(\tilde{G}), true \rangle$ *by applying* code(p).

Proof. "⇒": According to the proof of Lemma 2, the match m implies a match for applying code(p)*. We now show that the construction of $G \overset{p,m}{\Rightarrow} \tilde{G}$ is analogous to the application of the CHR rule* code(p)*:*

First the nodes and edges in $m(L)$ get deleted from G, but nodes and edges in $m(l(K)) = m(K)$ (l is an inclusion) are kept. For the CHR rule application, all constraints in the goal matched in H are removed. As H contains constraints encoding all nodes and edges of L we successfully remove nodes and edges in $m(L)$. Next we add nodes and edges in $R \setminus r(K)$. For the CHR rule application, the constraints in B are added. This adds constraints encoding all nodes and edges of R taking into account the substitutions made for matching H. As r is an inclusion, all nodes and edges in K are also encoded in encode(R) *and are added to the result. We therefore keep the constraints encoding $m(K)$ despite temporarily removing them. We showed, that all nodes and edges present in $\tilde{G}$ are also contained in the modified goal store after the CHR rule application.*

It remains to be shown that also the degree$/2$ *constraints remain consistent such that the CHR rule application indeed results in* encode$(\tilde{G})$ *being added to the goal store.*

We note that for nodes $v \in L \setminus K$ – i.e. nodes which are removed by the rule application – we only have a degree *constraint in H which also gets removed. Let now n, m as given in Def. 5 for a $v \in K$. There is a corresponding constraint* degree(T, D) *in H. For the node v the application of the graph production rule replaces n edges by m edges. Edges which are adjacent to $m(v)$, but are not in*

$m(L)$ remain unchanged. Therefore, the graph production rule changes the degree of $m(v)$ by $m-n$. The constraint degree$(T, D+m-n)$ *in B directly encodes this change, thus giving us a consistent degree encoding for nodes $v \in K$. Finally for nodes in $v \in R \setminus K$ which are newly added by the graph production rule we explicitly add a* degree *constraint to B with the correct degree according to Def. 5.*

"$\Leftarrow$": According to the proof of Lemma 1, the CHR match involved in the application of code(p) *implies a match morphism m. We can therefore argue analogously to the above by decomposing the effects of the CHR rule application into constraints which are removed and added. By combining this with the match morphism m, we can similarly show that the correct nodes and edges are removed and added when applied to the graph transformations and that the* degree *constraints remain consistent.* □

5 Confluence

Both graph transformation systems and constraint handling rules provide the notion of a confluence property. This property guarantees that any derivation made for an initial state results in the same final state no matter which applicable rules are applied. This section introduces the necessary definitions used for GTS and CHR confluence before comparing the two notions. It is shown how confluence checking in CHR can be adjusted to check the confluence property of a GTS encoded in CHR.

5.1 Preliminaries

Definition 7 (GTS confluence). *A GTS is called* confluent *if, for all typed graph transformations $G \overset{*}{\Rightarrow} H_1$ and $G \overset{*}{\Rightarrow} H_2$, there is a typed graph X together with typed graph transformations $H_1 \overset{*}{\Rightarrow} X$ and $H_2 \overset{*}{\Rightarrow} X$.* Local confluence *means that this property holds for all pairs of direct typed graph transformations $G \Rightarrow H_1$ and $G \Rightarrow H_2$. [2]*

A general result for rewriting systems is that it is sufficient to consider local confluence for terminating graph transformation systems. To verify local confluence we particularly need to study critical pairs and their joinability, according to this definition based on [2]:

Definition 8 (critical GTS pair). *A pair $P_1 \overset{r_1,m_1}{\Longleftarrow} K \overset{r_2,m_2}{\Longrightarrow} P_2$ of direct typed graph transformations is called a* critical GTS pair *if it is parallel dependent, and minimal in the sense that the pair (m_1, m_2) of matches $m_1 : L_1 \to K$ and $m_2 : L_2 \to K$ is jointly surjective.*

A pair $P_1 \overset{r_1,m_1}{\Longleftarrow} K \overset{r_2,m_2}{\Longrightarrow} P_2$ of direct typed graph transformations is called parallel independent *if $m_1(L_1) \cap m_2(L_2) \subseteq m_1(K_1) \cap m_2(K_2)$, otherwise it is called* parallel dependent.

A critical GTS pair $P_1 \overset{r_1,m_1}{\Longleftarrow} K \overset{r_2,m_2}{\Longrightarrow} P_2$ is called joinable *if there exists a typed graph K' together with typed graph transformations $P_1 \overset{*}{\Longrightarrow} K'$ and $P_2 \overset{*}{\Longrightarrow} K'$.*

A similar notion of confluence has been developed for CHR [6]:

Definition 9 (CHR confluence). *A CHR program is called* confluent *if for all states S, S_1, and S_2: If $S \mapsto^* S_1$ and $S \mapsto^* S_2$, then S_1 and S_2 are joinable. Two states S_1 and S_2 are called* joinable *if there exist states T_1 and T_2 such that $S_1 \mapsto^* T_1$ and $S_2 \mapsto^* T_2$ and T_1 and T_2 are variants.*

Analogous to a GTS, the confluence property for terminating CHR programs is determined by local confluence which can be checked through critical pairs:

Definition 10 (critical CHR pair). *Let R_1 be a simplification rule and R_2 be a (not necessarily different) rule whose variables have been renamed apart. Let $H_i \wedge A_i$ be the head and G_i be the guard of rule $R_i (i = 1, 2)$, then a* critical ancestor state *of R_1 and R_2 is*

$$\langle H_1 \wedge A_1 \wedge H_2, (A_1 \doteq A_2) \wedge G_1 \wedge G_2 \rangle,$$

provided A_1 and A_2 are non-empty conjunctions and $CT \models \exists((A_1 \doteq A_2) \wedge G_1 \wedge G_2)$.

Let S be a critical ancestor state of R_1 and R_2. If $S \mapsto S_1$ using rule R_1 and $S \mapsto S_2$ using rule R_2, then the tuple (S_1, S_2) is a critical CHR pair *of R_1 and R_2. A critical CHR pair (S_1, S_2) is* joinable *if S_1 and S_2 are joinable.*

5.2 Critical Pair Properties

After defining the different notions of confluence we now further investigate the difference between critical GTS pairs and critical CHR pairs for CHR programs encoding a GTS.

Lemma 3. *If $P_1 \overset{p_1,m_1}{\Longleftarrow} G \overset{p_2,m_2}{\Longrightarrow} P_2$ is a critical GTS pair, then there exists a conjunction of built-in constraints C such that $\langle \text{encode}(G), \top \rangle$ is a critical ancestor state for the critical CHR pair $(\langle \text{encode}(P_1), C \rangle, \langle \text{encode}(P_2), C \rangle)$.*

Proof. Theorem 1 guarantees that $\text{code}(P_1)$ *and* $\text{code}(P_2)$ *are applicable to the critical ancestor state.*

As m_1, m_2 are jointly surjective we know that all constraints in $\text{encode}(G)$ *are required for applying* $\text{code}(P_1)$ *and* $\text{code}(P_2)$*, i.e. there are no redundant constraints. We further know that there exist one or more constraints in* $\text{encode}(G)$ *which are required for both rule applications, as otherwise $m_1(L_1) \cap m_2(L_2) = \emptyset$ which is a contradiction to the critical GTS pair being parallel dependent. Let A be the conjunction consisting of all these constraints which are required for both rule applications.*

As $\text{code}(P_1)$ *and* $\text{code}(P_2)$ *are applicable, there are corresponding constraints A_1 and A_2 in the heads of those rules which are matched to A in a rule application. Let the heads of these rules be separated into $H_1 \wedge A_1$ and $H_2 \wedge A_2$. The match morphisms imply the existence of constraints H_1', H_2' in* $\text{encode}(G)$ *for all constraints in H_1 and H_2, respectively. These constraints have to be different from A for the rules to be applicable and they are disjoint because of*

the maximality of A. Due to the minimality of G, there are no further constraints in $\text{encode}(G)$*, therefore* $\text{encode}(G) = H_1' \wedge A \wedge H_2'$*.* $\langle \text{encode}(G), \top \rangle$ *is therefore a critical ancestor state. As* $\text{code}(P_1)$ *and* $\text{code}(P_2)$ *are both applicable to it, there is the corresponding critical pair:* $(\langle \text{encode}(P_1), A \doteq A_1 \wedge A \doteq A_2 \rangle, \langle \text{encode}(P_2), A \doteq A_1 \wedge A \doteq A_2 \rangle)$ □

Corollary 1. *If the CHR program for a terminating GTS is confluent, then all critical GTS pairs are joinable.*

Proof. As every critical GTS pair has a corresponding critical CHR pair and all critical CHR pairs are joinable, we know by the completeness property that the critical GTS pairs are also joinable. □

Due to Corollary 1, existing confluence checkers for CHR can be used to investigate confluence of a GTS encoded in CHR. Confluence of the CHR program is a sufficient condition for confluence of the corresponding GTS however, as can be seen in the example below, there are cases in which the CHR program may not be confluent although the corresponding GTS is confluent. Similarly, if we try to transfer the confluence property of a GTS to the corresponding CHR program, we cannot succeed as in general there are too many critical CHR pairs which could cause the CHR program to be non-confluent. To improve upon this situation, we therefore introduce a weaker kind of confluence:

Definition 11 (weak confluence, valid state). *A CHR program is called* weak confluent *if for all valid states S and states S_1 and S_2: If $S \mapsto^* S_1$ and $S \mapsto^* S_2$, then S_1 and S_2 are joinable.*

A state $S = \langle G', C \rangle$ is called valid *if there exists a graph G such that $G' =$* $\text{encode}(G)$*.*

Note that especially states which encode the same node of a graph with multiple node constraints or provide multiple degree constraints for the same node are invalid states. Example 7 includes a selection of such invalid states.

Using Definition 11 we can now investigate confluence for a CHR program corresponding to a confluent GTS:

Lemma 4. *If all critical GTS pairs are joinable, the corresponding CHR program is weak confluent.*

Proof. Let $S = \langle \text{encode}(G), C \rangle$ be a valid critical ancestor state. If the critical CHR pair resulting from S corresponds to a critical GTS pair, it is also joinable due to Theorem 2.

Let S be a valid critical ancestor state for which the resulting critical CHR pair does not correspond to a critical GTS pair. By definition of the critical ancestor state, we know that two rules $\text{code}(r_1)$ *and* $\text{code}(r_2)$ *are applicable and by Theorem 1 r_1 and r_2 are applicable to G, giving us the corresponding non-critical GTS pair $P_1 \overset{r_1,m_1}{\Longleftarrow} G \overset{r_2,m_2}{\Longrightarrow} P_2$. As S does not contain redundant constraints we know that m_1 and m_2 are jointly surjective. Thus this GTS pair has to be parallel independent as this is the only way for it to not be a critical GTS pair. If it is parallel*

independent however, we know that there exists P such that $P_1 \overset{r_2,m_2'}{\Longrightarrow} P \overset{r_1,m_1'}{\Longleftarrow} P_2$ and due to the previous soundness result this implies joinability for the critical CHR pair. □

Example 7. Consider a graph production rule for removing a loop from a node and its corresponding constraint handling rule:

$$R@\,\mathrm{node}(N), \mathrm{edge}(E, N, N), \mathrm{degree}(N, D) \Leftrightarrow \mathrm{node}(N), \mathrm{degree}(N, D-2)$$

For investigating confluence one must overlap this rule with itself which yields the following seven critical CHR ancestor states:

1. $\langle \mathrm{node}(N) \wedge \mathrm{edge}(E, N, N) \wedge \mathrm{degree}(N, D), \top \rangle$
2. $\langle \mathrm{node}(N) \wedge \mathrm{edge}(E, N, N) \wedge \mathrm{degree}(N, D) \wedge \mathrm{degree}(N, D'), \top \rangle$
3. $\langle \mathrm{node}(N) \wedge \mathrm{edge}(E, N, N) \wedge \mathrm{edge}(E', N, N) \wedge \mathrm{degree}(N, D), \top \rangle$
4. $\langle \mathrm{node}(N) \wedge \mathrm{node}(N') \wedge \mathrm{edge}(E, N, N) \wedge \mathrm{degree}(N, D), \top \rangle$
5. $\langle \mathrm{node}(N) \wedge \mathrm{edge}(E, N, N) \wedge \mathrm{edge}(E', N, N) \wedge \mathrm{degree}(N, D) \wedge \mathrm{degree}(N, D'), \top \rangle$
6. $\langle \mathrm{node}(N) \wedge \mathrm{node}(N') \wedge \mathrm{edge}(E, N, N) \wedge \mathrm{degree}(N, D) \wedge \mathrm{degree}(N, D'), \top \rangle$
7. $\langle \mathrm{node}(N) \wedge \mathrm{node}(N') \wedge \mathrm{edge}(E, N, N) \wedge \mathrm{edge}(E', N, N) \wedge \mathrm{degree}(N, D), \top \rangle$

States 2 and 4–7 are invalid states as they have multiple encodings of the same node or multiple degree encodings of a node. State 1 is the encoding of the corresponding critical pair for the graph production rule and state 3 is not critical because the corresponding pair of graph transformations is parallel independent:

$S_3 = \langle \mathrm{node}(N) \wedge \mathrm{edge}(E, N, N) \wedge \mathrm{edge}(E', N, N) \wedge \mathrm{degree}(N, D), \top \rangle$
$S_3 \overset{R}{\Rightarrow} S_3^1 = \langle \mathrm{node}(N) \wedge \mathrm{edge}(E', N, N) \wedge \mathrm{degree}(N, D-2), \top \rangle$
$S_3^1 \overset{R}{\Rightarrow} S_3' = \langle \mathrm{node}(N) \wedge \mathrm{degree}(N, D-4), \top \rangle$
$S_3 \overset{R}{\Rightarrow} S_3^2 = \langle \mathrm{node}(N) \wedge \mathrm{edge}(E, N, N) \wedge \mathrm{degree}(N, D-2), \top \rangle$
$S_3^2 \overset{R}{\Rightarrow} S_3'$

Theorem 3. *All critical GTS pairs are joinable if and only if the corresponding CHR program is weak confluent.*

Proof. The first direction follows directly from Lemma 4. By Lemma 3 we know that all critical GTS pairs have corresponding critical CHR pairs with valid critical ancestor states. However, if the CHR program is weak confluent all such critical CHR pairs are joinable and thus by Theorem 2 the critical GTS pairs are joinable as well. □

6 Conclusion

We have shown that constraint handling rules (CHR) provide an elegant way for embedding graph transformation systems (GTS). The resulting rules are very concise and directly related to the corresponding graph production rules. There is no need for further implementations besides these production rules which makes CHR a natural choice for prototyping a GTS. We have also shown that

a CHR implementation of a GTS shares many important properties with the formal GTS. Particularly the notion of confluence is similar and allows for standard CHR confluence checkers to be reused for embedded graph transformation systems with only slight adjustments.

To achieve this elegant solution, we restricted the GTS match morphisms to injective ones like [4] and [5] did. In the case of the standard nondeterministic semantics for CHR, all possible injective matches can be chosen. However, most CHR implementations make it difficult for a user to interactively modify or choose a match morphism, as it is chosen implicitly by the refined semantics of the implementation.

Apart from having a viable alternative for GTS implementations, there is room for further research. This work only considers typed graphs, but could be extended to support typed attributed graphs as well. It is also worthwhile to investigate further similarities between GTS and CHR by transferring existing results from one model to the other. For example, [2] provides criteria for layered graph transformation systems under which termination can be guaranteed and which might be applicable to CHR as well.

References

1. Frühwirth, T.: Theory and practice of constraint handling rules. Journal of Logic Programming, Special Issue on Constraint Logic Programming 37(1-3), 95–138 (1998)
2. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Springer, Heidelberg (2006)
3. Sadiq, W., Orlowska, M.E.: Applying graph reduction techniques for identifying structural conflicts in process models. In: Jarke, M., Oberweis, A. (eds.) CAiSE 1999. LNCS, vol. 1626, pp. 195–209. Springer, Heidelberg (1999)
4. Bakewell, A., Plump, D., Runciman, C.: Specifying pointer structures by graph reduction. In: Pfaltz, J.L., Nagl, M., Böhlen, B. (eds.) AGTIVE 2003. LNCS, vol. 3062, pp. 30–44. Springer, Heidelberg (2004)
5. Habel, A., Plump, D.: Relabelling in graph transformation. In: Corradini, A., Ehrig, H., Kreowski, H.-J., Rozenberg, G. (eds.) ICGT 2002. LNCS, vol. 2505, Springer, Heidelberg (2002)
6. Frühwirth, T., Abdennadher, S.: Essentials of Constraint Programming. Springer, Heidelberg (2003)
7. Frühwirth, T.: As time goes by: Automatic complexity analysis of simplification rules. In: 8th International Conference on Principles of Knowledge Representation and Reasoning, Toulouse, France (2002)

Multivalued Action Languages with Constraints in CLP(FD)

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

[1] Univ. di Udine, Dip. di Matematica e Informatica
dovier@dimi.uniud.it
[2] Univ. di Perugia, Dip. di Matematica e Informatica
formis@dipmat.unipg.it
[3] New Mexico State University, Dept. Computer Science
epontell@cs.nmsu.edu

Abstract. Action description languages, such as $\mathcal{A}$ and $\mathcal{B}$ [6], are expressive instruments introduced for formalizing planning domains and problems. The paper starts by proposing a methodology to encode an action language (with conditional effects and static causal laws), a slight variation of $\mathcal{B}$, using *Constraint Logic Programming over Finite Domains*. The approach is then generalized to lift the use of constraints to the level of the action language itself. A prototype implementation has been developed, and the preliminary results are presented and discussed.

1 Introduction

The construction of intelligent agents that can be effective in real-world environments has been a goal of researchers from the very first days of Artificial Intelligence. It has long been recognized that such an agent must be able to *acquire*, *represent*, and *reason* with knowledge. As such, a *reasoning component* has been an inseparable part of most agent architectures in the literature.

Although the underlying representations and implementations may vary between agents, the reasoning component of an agent is often responsible for making decisions that are critical to its existence. Logic programming languages offer many attributes that make them suitable as knowledge representation languages. Their declarative nature allows the modular development of provably correct reasoning modules [2]. Recursive definitions can be easily expressed and reasoned upon. Control knowledge and heuristic information can be declaratively introduced in the reasoning process. Furthermore, many logic programming languages offer a natural support for nonmonotonic reasoning, which is considered essential for commonsense reasoning. These features, along with the presence of efficient solvers [1, 11, 15, 7], make logic programming an attractive paradigm for knowledge representation and reasoning.

In the context of knowledge representation and reasoning, a very important application of logic programming has been in the domain of reasoning about actions and change and planning [2]. Planning problems have been effectively encoded using *Answer Set Programming (ASP)* [2]—where distinct answer sets represent different trajectories leading to the desired goal. Other logic programming paradigms,

V. Dahl and I. Niemelä (Eds.): ICLP 2007, LNCS 4670, pp. 255–270, 2007.

e.g., *Constraint Logic Programming over Finite Domains (CLP(FD))*, have been used less frequently to handle problems in reasoning about actions (e.g., [14, 17]). Comparably more emphasis has been placed in encoding planning problems as (non-logic programming) constraint satisfaction problems [10].

Recent proposals on representing and reasoning about actions and change have relied on the use of concise and high-level languages, commonly referred to as *action description languages*, e.g., the languages $\mathcal{A}$ and $\mathcal{B}$ [6]. Action languages allow one to write propositions that describe the effects of actions on states, and to create queries to infer properties of the underlying transition system. An *action description* is a specification of a planning problem using the action language.

The goal of this work is to explore the relevance of constraint solving and constraint logic programming [11, 1] in dealing with action languages and planning. The push towards this exploratory study came from a recent investigation [4, 5] aimed at comparing the practicality and efficiency of answer set programming versus constraint logic programming in solving various combinatorial and optimization problems. The study indicated that CLP offers a valid alternative, especially in terms of efficiency, to ASP when dealing with planning problems; furthermore, CLP offers the flexibility of programmer-developed search strategies and the ability to handle numerical constraints.

The first step, in this paper, is to demonstrate a scheme that directly processes an action description specification, in a language similar to $\mathcal{B}$, producing a CLP(FD) program that can be used to compute solutions to the planning problem. Our encoding has some similarities to the one presented in [10], although we rely on CLP instead of CSP, and our action language supports static causal laws and non-determinism—while the work of Lopez and Bacchus is restricted to STRIPS-like specifications.

While the first step relies on using constraints to compute solutions to a planning problem, the second step brings the expressive power of constraints to the level of the action language, by allowing multi-valued fluents and constraint-producing actions. The extended action language (named $\mathcal{B}^{FD}_{MV}$) can be as easily supported by the CLP(FD) framework, and it allows a declarative encoding of problems involving actions with resources, delayed effects, and maintenance goals. These ideas have been developed in a prototype, and some preliminary experiments are reported.

We believe that the use of CLP(FD) can greatly facilitate the transition of declarative extensions of action languages to concrete and effective implementations, overcoming some inherent limitations (e.g., efficiency and limited handling of numbers) of other logic-based systems (e.g., ASP).

2 An Action Language

"Action languages are formal models of parts of the natural language that are used for talking about the effect of actions" [6]. Action languages are used to define *action descriptions* that embed knowledge to formalize planning problems. In this section, we use a variant of the language $\mathcal{B}$, based on the syntax used in [16]. With a slight abuse of notation, we simply refer to this language as $\mathcal{B}$.

2.1 Syntax of $\mathcal{B}$

An action signature consists of a set $\mathcal{F}$ of fluent names, a set $\mathcal{A}$ of action names, and a set $\mathcal{V}$ of values for fluents in $\mathcal{F}$. In this section, we consider Boolean fluents, hence $\mathcal{V} = \{0, 1\}$.[1] A *fluent literal* is either a fluent f or its negation $\texttt{neg}(f)$. Fluents and actions are concretely represented by *ground* atomic formulae $p(t_1, \ldots, t_n)$ from a logic language $\mathcal{L}$. For simplicity, we assume that the set of admissible terms is finite.

The language $\mathcal{B}$ allows us to specify an *action description* $\mathcal{D}$, which relates actions, states, and fluents using predicates of the following forms:

- `executable(a, [list-of-conditions])` asserts that the given conditions have to be satisfied in the current state in order for the action `a` to be executable;
- `causes(a, l, [list-of-conditions])` encodes a dynamic causal law, describing the effect (the fluent literal `l`) of the execution of action `a` in a state satisfying the given conditions;
- `caused([list-of-conditions], l)` describes a static causal law—i.e., the fact that the fluent literal `l` is true in a state satisfying the given preconditions;

(where `[list-of-conditions]` denotes a list of fluent literals). An *action description* is a set of executability conditions, static, and dynamic laws. A specific *planning problem* contains an action description $\mathcal{D}$ along with a description of the *initial state* and the *desired goal* ($\mathcal{O}$):

- `initially(l)` asserts that the fluent literal `l` is true in the initial state,
- `goal(f)` asserts that the goal requires the fluent literal `f` to be true in the final state.

Fig. 4 reports an encoding of the three barrels problem using this language.

2.2 Semantics of $\mathcal{B}$

If $f \in \mathcal{F}$ is a fluent, and S is a set of fluent literals, we say that $S \models f$ iff $f \in S$ and $S \models \texttt{neg}(f)$ iff $\texttt{neg}(f) \in S$. Lists of literals $[\ell_1, \ldots, \ell_n]$ denote conjunctions of literals. We denote with $\neg S$ the set $\{f : \texttt{neg}(f) \in S\} \cup \{\texttt{neg}(f) : f \in S\}$. A set of fluent literals is *consistent* if there are no fluents f s.t. $S \models f$ and $S \models \texttt{neg}(f)$. If $S \cup \neg S \supseteq \mathcal{F}$ then S is *complete*. A set S of literals is *closed* under a set of static laws $\mathcal{SL} = \{\texttt{caused}(C_1, \ell_1), \ldots, \texttt{caused}(C_n, \ell_n)\}$, if for all $i \in \{1, \ldots, n\}$ it holds that $S \models C_i \Rightarrow S \models \ell_i$. The set $\texttt{Clo}_{\mathcal{SL}}(S)$ is defined as the smallest set of literals containing S and closed under $\mathcal{SL}$. $\texttt{Clo}_{\mathcal{SL}}(S)$ is uniquely determined and not necessarily consistent.

The semantics of an action language on the action signature $\langle \mathcal{V}, \mathcal{F}, \mathcal{A} \rangle$ is given in terms of a transition system $\langle \mathcal{S}, v, R \rangle$ [6], consisting of a set $\mathcal{S}$ of states, a total interpretation function $v : \mathcal{F} \times \mathcal{S} \longrightarrow \mathcal{V}$ (in this section $\mathcal{V} = \{0, 1\}$), and a transition relation $R \subseteq \mathcal{S} \times \mathcal{A} \times \mathcal{S}$. Given a transition system $\langle \mathcal{S}, v, R \rangle$ and a state $s \in \mathcal{S}$, let:

$$Lit(s) = \{f \in \mathcal{F} : v(f, s) = 1\} \cup \{\texttt{neg}(f) : f \in \mathcal{F}, v(f, s) = 0\}.$$

[1] Consequently, we often say that a fluent is true (resp., false) if its value is 1 (resp., 0).

Observe that $Lit(s)$ is consistent and complete. Given a set of dynamic laws $\mathcal{DL} = \{\texttt{causes}(a, \ell_1, C_1), \ldots, \texttt{causes}(a, \ell_n, C_n)\}$ for the action $a \in \mathcal{A}$ and a state $s \in \mathcal{S}$, we define the *effect of* a *in* s as follows: $E(a, s) = \{\ell_i : 1 \leqslant i \leqslant n, Lit(s) \models C_i\}$.

Let $\mathcal{D}$ be an action description defined on the action signature $\langle \mathcal{V}, \mathcal{F}, \mathcal{A}\rangle$, composed of dynamic laws $\mathcal{DL}$, executability conditions $\mathcal{EL}$, and static causal laws $\mathcal{SL}$. The transition system $\langle \mathcal{S}, v, R\rangle$ *described by* $\mathcal{D}$ is a transition system such that:

- If $s \in \mathcal{S}$, then $Lit(s)$ is closed under $\mathcal{SL}$;
- R is the set of all triples $\langle s, a, s'\rangle$ such that

$$Lit(s') = \texttt{Clo}_{\mathcal{SL}}(E(a, s) \cup (Lit(s) \cap Lit(s'))) \tag{1}$$

and $Lit(s) \models C$ for at least one condition $\texttt{executable}(a, C)$ in $\mathcal{EL}$.

Let $\langle \mathcal{D}, \mathcal{O}\rangle$ be a planning problem instance, where $\Delta = \{\ell \mid \texttt{initially}(\ell) \in \mathcal{O}\}$ is a complete set of fluent literals. A *trajectory* is a sequence $s_0 a_1 s_1 a_2 \ldots a_n s_n$ s.t. $\langle s_i, a_{i+1}, s_{i+1}\rangle \in R$ $(0 \leq i < n)$. Given the corresponding transition system $\langle \mathcal{S}, v, R\rangle$, a sequence of actions $a_1, \ldots, a_n$ is a solution (a *plan*) to the planning problem $\langle \mathcal{D}, \mathcal{O}\rangle$ if there is a trajectory $s_0 a_1 s_1 \ldots a_n s_n$ in $\langle \mathcal{S}, v, R\rangle$ s.t. $Lit(s_0) = \Delta$ and $Lit(s_n) \models \ell$ for each $\texttt{goal}(\ell) \in \mathcal{O}$. The plan is sequential and the desired length is given.

3 Modeling $\mathcal{B}$ and Planning Problems in CLP(FD)

Let us describe how action theories are mapped to finite domain constraints. We will focus on how constraints can be used to model the possible transitions from each individual state of the transition system. If s_v and s_u are the starting and ending states of a transition, we assert constraints that relate the truth value of fluents in s_v and s_u.

A Boolean variable is introduced to describe the truth value of each fluent in a state. The value of a fluent f in s_v (resp., s_u) is represented by the variable $\texttt{IV}_f^v$ (resp., $\texttt{EV}_f^u$). These variables can be used to build expressions $\texttt{IV}_l^v$ ($\texttt{EV}_l^u$) that represent the truth value of each fluent literal $\texttt{l}$. In particular, if $\texttt{l}$ is a fluent $\texttt{f}$, then $\texttt{IV}_l^v = \texttt{IV}_f^v$; if $\texttt{l}$ is the literal $\texttt{neg(f)}$, then $\texttt{IV}_l^v = 1 - \texttt{IV}_f^v$. Similar equations can be set for $\texttt{EV}_l$. In a similar spirit, given a conjunction of literals $\alpha \equiv [\texttt{l}_1, \ldots, \texttt{l}_n]$ we will denote with $\texttt{IV}_\alpha^v$ the expression $\texttt{IV}_{l_1}^v \wedge \cdots \wedge \texttt{IV}_{l_n}^v$; an analogous definition is given for $\texttt{EV}_\alpha^u$. We will also introduce, for each action a_i, a Boolean variable $\texttt{A}_i^v$, representing whether the action is executed or not in the transition from s_v to s_u under consideration.

Given a specific fluent f, we develop constraints that determine when $\texttt{EV}_f^u$ is true and false. Let us consider the dynamic causal laws that have f as a consequence:

$$\texttt{causes}(a_{t_1}, f, \alpha_1) \quad \cdots \quad \texttt{causes}(a_{t_m}, f, \alpha_m)$$

Analogously, we consider the static causal laws that assert $\texttt{neg}(f)$:

$$\texttt{causes}(a_{f_1}, neg(f), \beta_1) \quad \cdots \quad \texttt{causes}(a_{f_n}, neg(f), \beta_n)$$

Let us also consider the static causal laws related to f

$$\begin{array}{ccc} \texttt{caused}(\gamma_1, f) & \cdots & \texttt{caused}(\gamma_h, f) \\ \texttt{caused}(\psi_1, neg(f)) & \cdots & \texttt{caused}(\psi_\ell, neg(f)) \end{array}$$

$$\mathtt{EV}_f^u \leftrightarrow \mathtt{Posfired}_f^{v,u} \vee (\neg\mathtt{Negfired}_f^{v,u} \wedge \mathtt{IV}_f^v) \quad (2)$$

$$\neg\mathtt{Posfired}_f^{v,u} \vee \neg\mathtt{Negfired}_f^{v,u} \quad (3)$$

$$\mathtt{Posfired}_f^{v,u} \leftrightarrow \mathtt{DynP}_f^v \vee \mathtt{StatP}_f^u \quad (4)$$

$$\mathtt{Negfired}_f^{v,u} \leftrightarrow \mathtt{DynN}_f^v \vee \mathtt{StatN}_f^u \quad (5)$$

$$\mathtt{DynP}_f^v \leftrightarrow \bigvee_{i=1}^{m}(\mathtt{IV}_{\alpha_i}^v \wedge \mathtt{A}_{t_i}^v) \quad (6)$$

$$\mathtt{StatP}_f^u \leftrightarrow \bigvee_{i=1}^{h} \mathtt{EV}_{\gamma_i}^u \quad (7)$$

$$\mathtt{DynN}_f^v \leftrightarrow \bigvee_{i=1}^{n}(\mathtt{IV}_{\beta_i}^v \wedge \mathtt{A}_{f_i}^v) \quad (8)$$

$$\mathtt{StatN}_f^u \leftrightarrow \bigvee_{i=1}^{\ell} \mathtt{EV}_{\psi_i}^u \quad (9)$$

Fig. 1. The constraint $C_f^{v,u}$ for the generic fluent f

Finally, for each action a_i we will have its executability conditions:

$$\mathtt{executable}(a_i, \delta_1^i) \qquad \ldots \qquad \mathtt{executable}(a_i, \delta_p^i)$$

Figure 1 describes the Boolean constraints that can be used in encoding the relations that determine the truth value of the fluent f. We will denote with $C_f^{v,u}$ the conjunction of such constraints. Given an action specification over the set of fluents $\mathcal{F}$, the system of constraints $C_{\mathcal{F}}^{v,v+1}$ includes:

- the constraint $C_f^{v,v+1}$ for each $f \in \mathcal{F}$ and for each $0 \leq v < N$ where N is the chosen length of the plan;
- for each $f \in \mathcal{F}$ and $0 \leq v \leq N$, the constraints $\mathtt{IV}_f^v = \mathtt{EV}_f^v$
- for each $0 \leq v < N$, the constraint $\sum_{a_j \in \mathcal{A}} \mathtt{A}_j^v = 1$
- for each $0 \leq v < N$ and for each action $a_i \in \mathcal{A}$, the constraints $\mathtt{A}_i^v \rightarrow \bigvee_{j=1}^{p} \mathtt{IV}_{\delta_j^i}^v$

This modeling has been translated into a concrete implementation using SICStus Prolog. In this translation constrained CLP variables directly reflect the Boolean variables modeling fluent and action's occurrences. Consequently, causal laws and executability conditions are directly translated into CLP constraints. The complete code and proofs can be found at `www.dimi.uniud.it/dovier/CLPASP`. Let us proceed with a sketch of the correctness proof of the constraint-based encoding w.r.t. the semantics.

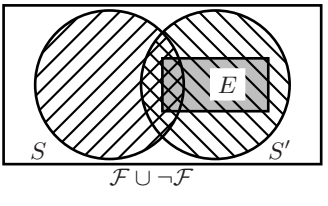


Fig. 2. Sets of fluents involved in a state transition

Let $S = Lit(s_v)$ (resp., $S' = Lit(s_{v+1})$) be the set of fluent literals that holds in s_v (resp., s_{v+1}). We denote by $E = E(a, s)$, and by $\mathtt{Clo} = \mathtt{Clo}_{\mathcal{SL}}$. Fig. 2 depicts the

equation $S' = \texttt{Clo}(E \cup (S \cap S'))$. From any specific, known, S (resp., S'), we can obtain a consistent assignment σ_S (resp., $\sigma_{S'}$) of truth values for all the variables $\texttt{IV}_f^v$ (resp., $\texttt{EV}_f^{v+1}$) of s_v (resp., s_{v+1}). Conversely, each truth assignment σ_S (resp., $\sigma_{S'}$) for all variables $\texttt{IV}_f^v$ (resp., $\texttt{EV}_f^{v+1}$) corresponds to a consistent set of fluents S (resp., S').

Let σ_a be the assignment of truth values for such variables such that $\sigma_a(\texttt{A}_i^v) = 1$ if and only if a_i occurs in the state transition from s_v to s_{v+1}. Note that the domains of σ_S, $\sigma_{S'}$, and σ_a are disjoint, so we can safely denote by $\sigma_S \circ \sigma_{S'} \circ \sigma_a$ the composition of the three assignments. Clearly, $E \subseteq S'$.

Theorem 1 states the completeness of the constraint-based modeling of the planning problem. For any given action description $\mathcal{D}$, if a triple $\langle s, a, s' \rangle$ belongs to the transition system described by $\mathcal{D}$, then the assignment $\sigma = \sigma_S \circ \sigma_{S'} \circ \sigma_a$ satisfies $C_{\mathcal{F}}^{v,v+1}$.

Theorem 1 (Completeness). *If* $S' = \texttt{Clo}_{\mathcal{SL}}(E(a_i, s_v) \cup (S \cap S'))$ *then* $\sigma_S \circ \sigma_{S'} \circ \sigma_a$ *is a solution of the constraint* $C_{\mathcal{F}}^{v,v+1}$.

Let us observe that the converse of the above theorem does not necessarily hold. The problem arises from the fact that the implicit minimality in the closure operation is not reflected in the computation of solutions to the constraint. Consider the action description where $\mathcal{F} = \{\texttt{f}, \texttt{g}, \texttt{h}\}$ and $\mathcal{A} = \{\texttt{a}\}$, with predicates:

```
executable(a,[]).  causes(a,f,[]).  caused([g],h).  caused([h],g).
```

Setting $S = \{\texttt{neg(f)}, \texttt{neg(g)}, \texttt{neg(h)}\}$ and $S' = \{\texttt{f}, \texttt{g}, \texttt{h}\}$ determines a solution of the constraint $C_{\mathcal{F}}^{v,v+1}$ with the execution of action a, but $\texttt{Clo}_{\mathcal{SL}}(E \cup (S \cap S')) = \{\texttt{f}\} \subset S'$. However, the following holds:

Theorem 2 (Weak Soundness). *Let* $\sigma_S \circ \sigma_{S'} \circ \sigma_a$ *identify a solution of the constraint* $C_{\mathcal{F}}^{v,v+1}$. *Then* $\texttt{Clo}_{\mathcal{SL}}(E(a_i, s_v) \cup (S \cap S')) \subseteq S'$.

Let us consider the set of static causal laws $\mathcal{SL}$. $\mathcal{SL}$ identifies a *definite propositional program* P as follows. For each positive fluent literal p, let $\varphi(\texttt{p})$ be the (fresh) predicate symbol p, and for each negative fluent literal neg(p) let $\varphi(\texttt{neg(p)})$ be the (fresh) predicate symbol $\tilde{p}$. The program P is the set of clauses of the form $\varphi(\texttt{p}) \leftarrow \varphi(\texttt{l1}), \ldots, \varphi(\texttt{lm})$, for each static causal law caused([l1,...,lm],p). Notice that p and $\tilde{p}$ are independent predicate symbols in P. From P one can extract the dependency graph $\mathcal{G}(P)$ in the usual way, and the following result can be stated.

Theorem 3 (Correctness). *Let* $\sigma_S, \sigma_{S'}, \sigma_a$ *be a solution of the constraint* $C_{\mathcal{F}}^{v,v+1}$. *If the dependency graph of* P *is acyclic, then* $\texttt{Clo}_{\mathcal{SL}}(E(a_i, s_v) \cup (S \cap S')) = S'$.

If the program P meets the conditions of the previous theorem, then the following holds.

Theorem 4. *There is a trajectory* $\langle s_0, a_1, s_1, a_2, \ldots, a_n, s_n \rangle$ *in the transition system if and only if there is a solution for the constraints* $C_{\mathcal{F}}^{0,1} \wedge C_{\mathcal{F}}^{1,2} \wedge \cdots \wedge C_{\mathcal{F}}^{n-1,n}$.

4 The Action Language $\mathcal{B}_{MV}^{FD}$

Constraints represent a very declarative notation to express relationships between unknowns; as such, the ability to use them directly in the action theory would greatly

enhance the declarative and expressive power of the action language, facilitating the encoding of complex action domains, such as those involving multivalued fluents.

Example 1 (Control Knowledge). Domain-specific control knowledge can be formalized as constraints that we expect to be satisfied by all the trajectories. For example, we may know that if a certain action occurs at a given time step (e.g., `ingest_poison`) then at the next time step we will always perform the same action (e.g., `call_doctor`). This could be encoded as `occ(ingest_poison)`$\Rightarrow$`occ(call_doctor)`^{1} where `occ(a)` is a fluent describing the occurrence of the action `a` and `f`^{1} indicates that the fluent `f` should hold at the next time step. The operator $\Rightarrow$ is an implication constraint.

Example 2 (Delayed Effect). Let us assume that the action `a`(e.g., `req_reimbursement`) has a delayed effect (e.g., `bank_account` increased by $50 after 30 time units). This could be expressed as a dynamic causal law:

`causes(request_reimbursement,incr(bank,50)`^{30}`,[])`

where `incr` is a constraint introduced to deal with additive computations.

Example 3 (Maintenance Goals). It is not uncommon to encounter planning problems where along with the type of goals described earlier (known as *achievement* goals), there are also *maintenance* goals, representing properties that must persist throughout the trajectory. Constraints are a natural way of encoding maintenance properties, and can be introduced along with simple temporal operators. E.g., if the fluent `fuel` represents the amount of fuel available, then the maintenance goal which guarantees that we will not be left stranded could be encoded as: `always(fuel > 0)`.

Furthermore, the encoding of an action theory using multivalued fluents leads to more compact and more efficient representations, facilitating constraint propagation during planning (with pruning of the search space) and better exposing non-determinism (that could be exploited, for example, by a parallel planner).

4.1 Syntax of $\mathcal{B}^{FD}_{MV}$

Let us introduce the syntax of $\mathcal{B}^{FD}_{MV}$. As for $\mathcal{B}$, the action signature consists of a set $\mathcal{F}$ of fluent names, a set $\mathcal{A}$ of action names, and a set $\mathcal{V}$ of values for fluents in $\mathcal{F}$.

In the definition of an action description, an assertion (*domain declaration*) of the kind $\texttt{fluent}(f, v_1, v_2)$ or $\texttt{fluent}(f, \{v_1, \ldots, v_k\})$ declares that f is a fluent and that its set of values $\mathcal{V}$ is the interval $[v_1, v_2]$ or the set $\{v_1, \ldots, v_k\}$.[2] An *annotated fluent* (`AF`) is an expression f^a, where f is a fluent and $a \in \mathbb{N}^-$.[3] Intuitively speaking, an annotated fluent f^a denotes the value the fluent f had in the past, $-a$ steps ago. Such fluents can be used in *fluent expressions* (`FE`), which are defined inductively as follows:

$$\texttt{FE} ::= n \mid \texttt{AF} \mid \texttt{abs(FE)} \mid \texttt{FE}_1 \oplus \texttt{FE}_2 \mid \texttt{rei(FC)}$$

where $n \in \mathbb{Z}$, $\oplus \in \{+, -, *, /, \texttt{mod}\}$. `rei(FC)` is the reification of fluent constraint `FC`.

[2] Note that we could generalize the notion of domain to more complex and non-numeric sets.

[3] With $\mathbb{N}^-$ we denote the set $\{0, -1, -2, -3, \ldots\}$. We will often denote f^0 simply by f.

Fluent expressions can be used to build *fluent constraints* (FC), i.e., formulae of the form $\texttt{FE}_1$ op $\texttt{FE}_2$, where $\texttt{FE}_1$ and $\texttt{FE}_2$ are fluent expressions and op $\in \{\texttt{eq}, \texttt{neq}, \texttt{geq}, \texttt{leq}, \texttt{lt}, \texttt{gt}\}$. The language $\mathcal{B}^{FD}_{MV}$ allows one to specify an *action description*, which relates actions, states, and fluents using predicates of the following forms:

- axioms of the form $\texttt{executable}(a, C)$ asserting that the fluent constraint C has to be entailed by the current state in order for the action a to be executable.
- axioms of the form $\texttt{causes}(a, C, C_1)$ encode dynamic causal laws. The action a can be executed if the constraint C_1 is entailed by the current state; the state produced by the execution of the action is required to entail the constraint C.
- axioms of the form $\texttt{caused}(C_1, C_2)$ describe static causal laws. If the fluent constraint C_1 is satisfied in a state, then the constraint C_2 must also hold in such state.

An *action description* is a set of executability conditions, static and dynamic laws.

Observe that traditional action languages like $\mathcal{B}$ are special cases of $\mathcal{B}^{FD}_{MV}$. For example, the dynamic causal law of $\mathcal{B}$:

$$\texttt{causes(a, f, [f}_1\texttt{,...,f}_k\texttt{, neg(g}_1\texttt{),..., neg(g}_h\texttt{)])}$$

can be encoded as

$$\texttt{causes(a, f}^0\texttt{ eq 1,[f}_1^0\texttt{ eq 1,...,f}_k^0\texttt{ eq 1, g}_1^0\texttt{ eq 0, ..., g}_h^0\texttt{ eq 0])}$$

A specific instance of a planning problem is a pair $\langle \mathcal{D}, \mathcal{O} \rangle$, where $\mathcal{D}$ is an action theory, and $\mathcal{O}$ contains any number of axioms of the form $\texttt{initially}(C)$ and $\texttt{goal}(C)$, where C is a fluent constraint.

4.2 Semantics of $\mathcal{B}^{FD}_{MV}$

Each fluent f is assigned uniquely to a domain $\mathrm{dom}(f)$ in the following way:

- If $\texttt{fluent}(f, v_1, v_2) \in \mathcal{D}$ then $\mathrm{dom}(f) = \{v_1, v_1 + 1, \ldots, v_2\}$.
- If $\texttt{fluent}(f, Set) \in \mathcal{D}$, then $\mathrm{dom}(f) = Set$.

A function $\nu : \mathcal{F} \to \mathbb{Z} \cup \{\bot\}$ is a *state* if $\nu(f) \in \mathrm{dom}(f)$ for all $f \in \mathcal{F}$. For a number $n \geq 1$, we define a *state sequence* $\bar{\nu}$ as a tuple $\langle \nu_0, \ldots, \nu_n \rangle$ where each ν_i is a state.

Let us consider a state sequence $\bar{\nu}$, a step $0 \leq i < |\bar{\nu}|$, a fluent expression φ, and let us define the concept of *value* of φ in $\bar{\nu}$ at step i (denoted by $\bar{\nu}(\varphi, i)$):

- $\bar{\nu}(x, i) = x$ if x is a number
- $\bar{\nu}(f^a, i) = \nu_{i-|a|}(f)$ if $|a| \leq i$, $\bot$ otherwise
- $\bar{\nu}(abs(\varphi), i) = abs(\bar{\nu}(\varphi, i))$
- $\bar{\nu}(\varphi_1 \oplus \varphi_2, i) = \bar{\nu}(\varphi_1, i) \oplus \bar{\nu}(\varphi_2, i)$

We treat the interpretation of the various $\oplus$ operations and relations as strict w.r.t. $\bot$. Given a fluent constraint $\varphi_1 \,\mathrm{op}\, \varphi_2$, a state sequence $\bar{\nu}$ and a time $0 \leq i < |\bar{\nu}|$, the notion of satisfaction $\bar{\nu} \models_i \varphi_1 \,\mathrm{op}\, \varphi_2$ is defined as $\bar{\nu} \models_i \varphi_1 \,\mathrm{op}\, \varphi_2 \;\Leftrightarrow\; \bar{\nu}(\varphi_1, i) \,\mathrm{op}\, \bar{\nu}(\varphi_2, i)$.

If $\bar{\nu}(\varphi_1, i)$ or $\bar{\nu}(\varphi_2, i)$ is $\bot$, then $\bar{\nu} \not\models_i \varphi_1 \,\mathrm{op}\, \varphi_2$. $\models_i$ can be generalized to the case of propositional combinations of fluent constraints. In particular, $\bar{\nu}(rei(C), i) = 1$ if $\bar{\nu} \models_i C$, else $\bar{\nu}(rei(C), i) = 0$. The operations $\cap$ and $\cup$ on states are defined next:

$$\nu_1 \cup \nu_2(f) = \begin{cases} \nu_1(f) \textit{ if } \nu_1(f) = \nu_2(f) \\ \nu_1(f) \textit{ if } \nu_2(f) = \bot \\ \nu_2(f) \textit{ if } \nu_1(f) = \bot \end{cases} \qquad \nu_1 \cap \nu_2(f) = \begin{cases} \nu_1(f) \textit{ if } \nu_1(f) = \nu_2(f) \\ \bot \quad \textit{otherwise} \end{cases}$$

Let $\bar{\nu}$ be a state sequence; we say that $\bar{\nu}$ is consistent if, for each $0 \leq i < |\bar{\nu}|$, and for each static causal law $\texttt{caused}(C_1, C_2)$ we have that: $\bar{\nu} \models_i C_1 \Rightarrow \bar{\nu} \models_i C_2$.

Let a be an action and $\bar{\nu}$ be a state sequence. The action a is executable in $\bar{\nu}$ at step i ($0 \leq i < |\bar{\nu}| - 1$) if there is an axiom $\texttt{executable}(a, C)$ such that $\bar{\nu} \models_i C$.

Let us denote with $Dyn(a)$ the set of dynamic causal law axioms for action a. The effects of executing a at step i in the state sequence $\bar{\nu}$, denoted by $\mathit{Eff}(a, \bar{\nu}, i)$, is

$$\mathit{Eff}(a, \bar{\nu}, i) = \bigwedge \{C \mid \texttt{causes}(a, C, C_1) \in Dyn(a), \bar{\nu} \models_i C_1\}$$

Furthermore, given a constraint C, a state sequence $\bar{\nu}$, and a step i, the reduct $Red(C, \bar{\nu}, i)$ is defined as the constraint

$$Red(C, \bar{\nu}, i) = C \wedge \bigwedge \{f^{-j} = \nu_{i-j}(f) \mid f \in \mathcal{F}, 1 \leq j \leq i\}$$

Let us denote with $Stat$ the set of static causal law axioms. We can define $Clo(\bar{\nu}, i)$ as

$$Clo(\bar{\nu}, i) = \bigwedge \{C_2 \mid \texttt{caused}(C_1, C_2) \in Stat, \bar{\nu} \models_i C_1\}$$

A state sequence $\bar{\nu}$ is a valid trajectory if the following conditions hold:

- for each axiom of the form $\texttt{initial}(C)$ in the action theory we have that $\bar{\nu} \models_0 C$
- for each axiom of the form $\texttt{goal}(C)$ we have that $\bar{\nu} \models_{|\bar{\nu}|-1} C$
- for each $0 \leq i < |\bar{\nu}| - 1$ there is an action a_i such that
  - action a_i is executable in $\bar{\nu}$ at step i
  - we have that $\nu_{i+1} = \sigma \cup (\nu_i \cap \nu_{i+1})$ (*) where σ is a solution of the constraint

$$Red(\mathit{Eff}(a_i, \bar{\nu}, i), \bar{\nu}, i + 1) \wedge Clo(\bar{\nu}, i + 1)$$

Let us conclude with some comments on the relationship between the semantics of $\mathcal{B}$ and of $\mathcal{B}^{FD}_{MV}$. First of all, the lack of backward references in $\mathcal{B}$ allows us to analyze each $\mathcal{B}$ transition independently. Conversely, in $\mathcal{B}^{FD}_{MV}$ we need to keep track of the complete trajectory—represented by a sequence of states.

In $\mathcal{B}$, the effect of a static or dynamic causal law is to set the truth value of a fluent. Hence, the sets E and $\texttt{Clo}$ can be deterministically determined (even if they can be inconsistent) and the equation (1) takes care of these two sets and of the inertia. In $\mathcal{B}^{FD}_{MV}$, instead, less determined effects can be set. For instance, $\texttt{causes}(a, f\ gt\ 1, [\,])$, if $\mathrm{dom}(f) = \{0, 1, 2, 3\}$, admits two possible values for f in the next state. One could extend the semantics of Sect. 2, by working on sets of sets for E or $\texttt{Clo}$. Instead, we have chosen to encode the nondeterminism within the solutions of the introduced constraints. Equation (1) is therefore replaced by equation (*) above.

The concrete implementation of $\mathcal{B}^{FD}_{MV}$ in SICStus Prolog is directly based on this semantics. We omit the detailed description, which is a fairly mechanical extension of the implementation of $\mathcal{B}$ (in this case the main difference w.r.t. what done in Sect. 3, is that the set of the admitted values for multivalued fluents determines the domain of the CLP constrained variables), and relative proof of correctness due to lack of space.

4.3 Some Concrete Extensions

The language $\mathcal{B}^{FD}_{MV}$ described above has been implemented using SICStus Prolog, as a fairly direct generalization of the encoding described for the Boolean case. In addition, we have introduced in the implementation some additional syntactic extensions, to facilitate the encoding of recurring problem features.

It is possible to add information about the *cost* of each action, fluent, and about the global cost of a plan. This can be done by writing rules of the form:

- `action_cost(action,VAL)` (if no information is given, the default cost is 1).
- `state_cost(FE)` (if no information is given, the default cost is 1) is the cost of a state, where FE is a fluent expression built on current fluents.
- `plan_cost(plan op n)` where `n` is a number, adds the information about the global cost admitted for the sequence of actions.
- `goal_cost(goal op NUM)` adds a constraint about the global cost admitted for the sequence of states.
- `minimize_action` to constrain the search to a plan with minimal global action cost.
- `minimize_state` which forces the search of a plan with minimal goal state cost.

The implementation of these cost-based constraints relies on the optimization features offered by SICStus' labeling predicate: the labeling phase is guided by imposing an objective function to be optimized.

The language allows the definition of *absolute temporal constraints*, i.e., constraints that refer to specific time instances in the trajectory. We define a timed fluent as a pair `FLUENT @ TIME`. Timed fluents are used to build timed fluent expressions (TE) and timed primitive constraints (TC). E.g., `contains(5) @ 2 leq contains(5) @ 4` states that, at time 2, barrel 5 contains at most the same amount of water as at time 4. `contains(12) @ 2 eq 3` states that, at time 3, barrel 12 contains 3 liters of water. These constructs can be used in the following expressions:

- `cross_constraint(TC)` imposes the timed constraint `TC` to hold.
- `holds(FC,StateNumber)` It is a simplification of the above constraint. states that the primitive fluent constraint FC holds at the desired State Number (0 is the number of the initial state). It is therefore a generalization of the `initially` primitive. It allows to drive the plan search with some point information.
- `always(FC)` states that the fluent constraints `FC` holds in all the states. Current fluents must be used in order to avoid negative references.

The semantics can be easily extended by conjoining these new constraints to the formulae of the previous subsection.

4.4 An Extended $\mathcal{B}^{FD}_{MV}$ Language: Looking Into the Future

We propose to extend $\mathcal{B}^{FD}_{MV}$ to allow constraints that reason about future steps of the trajectory (along lines similar to [3]).

Syntax: Let us generalize the syntax presented earlier as follows: an annotated fluent is of the form f^a, where $f \in \mathbb{Z}$. Constructing fluent formulae and fluent constraints now implies the ability of looking into the future steps of computation. In addition,

we introduce another fluent constraint, that will help us encoding interesting problems: $incr(f^a, n)$ where f^a is an annotated fluent and n is a number. The $incr$ constraint provides a simplified view of additive fluents [9].

Semantics: The definition of the semantics becomes a process of "validating" a sequence of states to verify their fitness to serve as a trajectory. Given a fluent formula/constraint φ and a time step i, we define the concept $Absolute(\varphi, i)$ as follows:

- if $\varphi \equiv n$ then $Absolute(\varphi, i) = n$
- if $\varphi \equiv f^a$ then $Absolute(\varphi, i) = f^{i+a}$
- if $\varphi \equiv \varphi_1 \oplus \varphi_2$ then $Absolute(\varphi, i) = Absolute(\varphi_1, i) \oplus Absolute(\varphi_2, i)$
- if $\varphi \equiv abs(\varphi_1)$ then $Absolute(\varphi, i) = abs(Absolute(\varphi_1, i))$
- if $\varphi \equiv rei(\varphi_1)$ then $Absolute(\varphi, i) = rei(Absolute(\varphi_1, i))$
- if $\varphi \equiv \varphi_1 \ \mathtt{op}\ \varphi_2$ then $Absolute(\varphi, i) = Absolute(\varphi_1, i) \ \mathtt{op}\ Absolute(\varphi_2, i)$
- if $\varphi \equiv incr(\varphi_1, n)$ then $Absolute(\varphi, i) = incr(Absolute(\varphi_1, i), n)$

For a sequence of states $\bar{\nu} = \langle \nu_0, \ldots, \nu_n \rangle$ and actions $\bar{a} = \langle a_0, \ldots, a_{n-1} \rangle$, let us define

$$Global(\bar{a}, \bar{\nu}) = \bigwedge_{i=0}^{n-1} Absolute(Eff(a_i, \bar{\nu}, i), i+1)$$

The sequence of states $\bar{\nu}$ is a trajectory if

- for each axiom of the form $\mathtt{initial}(C)$ in the action theory we have that $\bar{\nu} \models_0 C$
- for each axiom of the form $\mathtt{goal}(C)$ in the action theory we have that $\bar{\nu} \models_{|\bar{\nu}|-1} C$
- there is a sequence of actions $\bar{a} = \langle a_0, a_1, \ldots, a_{n-1} \rangle$ with the following properties: for each $0 \leq i < n$ we have that a_i is executable at step i of $\bar{\nu}$ and $\nu_{i+1} = \sigma \cup (\nu_i \cap \nu_{i+1})$, where σ is a solution of $Red(Global(\bar{a}, \bar{\nu}), \bar{\nu}, i+1) \wedge Clo(\bar{\nu}, i+1)$.

In particular, if $Incr(C, i) = \{n \mid incr(f^i, n) \in C\}$ are all the $incr$ constraints for an annotated fluent f^i, then θ is a solution of it w.r.t. a sequence of states $\bar{\nu}$ if $\nu_i(f) = \nu_{i-1}(f) + \sum_{n \in Incr(C,i)} n$.

5 Experimental Analysis

The prototype, implemented on an AMD Opteron 2.2GHz Linux machine, has been validated on a number of benchmarks. Extensive testing has been performed on the CLP encoding of $\mathcal{B}$, and additional results can be found at `www.dimi.uniud.it/dovier/CLPASP`. Here we concentrate on two representative examples.

5.1 Three-Barrel Problem

We experimented with different encodings of the three-barrel problem. There are three barrels of capacity N (even number), $N/2 + 1$, and $N/2 - 1$, respectively. At the beginning the largest barrel is full of wine, the other two are empty. We wish to reach a state in which the two larger barrels contain the same amount of wine. The only permissible action is to pour wine from one barrel to another, until the latter is full or the former is empty. Figure 4 shows the encodings of the problem (for $N = 12$)

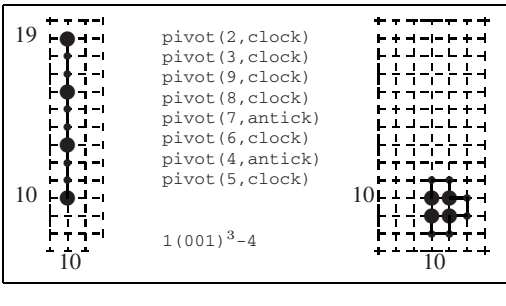
Table 1. Experimental results with various instances of the three-barrels problem (For CLP(FD) and $\mathcal{B}^{FD}_{MV}$ we reported the time required for the constrain and the labelling phases)

Barrels' capacities	Len.	Ans.	$\mathcal{B}$				$\mathcal{B}^{FD}_{MV}$		
							unconstrained plan cost	constrained plan cost (in parentheses)	
			lparse	Smodels	Cmodels	CLP(FD)			
8-5-3	6	N	8.95	0.10	0.85	0.16+0.36	0.02+0.11	(70)	0.01+0.10
8-5-3	7	Y	8.94	0.28	1.34	0.19+0.47	0.02+0.13	(70)	0.03+0.13
8-5-3	8	Y	9.16	0.39	2.07	0.18+2.66	0.03+0.85	(70)	0.01+0.79
8-5-3	9	Y	9.22	0.39	8.11	0.22+1.05	0.02+0.28	(70)	0.05+0.28
12-7-5	10	N	35.63	18.31	325.28	0.45+26.86	0.05+7.79	(90)	0.03+6.78
12-7-5	11	Y	35.70	45.91	781.28	0.52+28.87	0.05+9.46	(90)	0.04+5.03
12-7-5	12	Y	35.58	81.12	4692.08	0.58+203.34	0.06+57.42	(100)	0.04+35.31
12-7-5	13	Y	35.67	18.87	1581.49	0.66+66.52	0.06+25.65	(100)	0.03+23.26
16-9-7	14	N	114.16	2018.65	–	1.28+2560.90	0.07+564.68	(170)	0.07+518.78
16-9-7	15	Y	113.53	2493.61	–	1.29+2833.97	0.07+688.84	(170)	0.07+520.14
16-9-7	16	Y	115.36	6801.36	–	1.37+17765.62	0.06+4282.86	(170)	0.04+1904.17
16-9-7	17	Y	114.04	2294.15	–	1.55+6289.06	0.06+1571.78	(200)	0.06+1389.27

in $\mathcal{B}$ (where, it is also required that the smallest barrel is empty at the end) and $\mathcal{B}^{FD}_{MV}$. Table 1 provides the execution times (in seconds) for different values of N and different plan lengths. The $\mathcal{B}$ encoding has been processed by our CLP(FD) implementation and by two ASP solvers (Smodels and Cmodels)—the encoding of a $\mathcal{B}$ action description in ASP is natural (see [5]). The $\mathcal{B}^{FD}_{MV}$ descriptions have been solved using SICStus Prolog.

5.2 2-Dimensional Protein Folding Problem

The problem we have encoded is a simplification of the protein structure folding problem. The input is a chain $a_1 a_2 \cdots a_n$ with $a_i \in \{0, 1\}$, initially placed in a vertical position, as in Fig. 3-left. We will refer to each a_i as an *amino acid*. The permissible actions are the counterclockwise/clockwise *pivot moves*. Once one point i of the chain is selected, the points $a_1, a_2, \ldots, a_i$ will remain fixed, while the points $a_{i+1}, \ldots, a_n$ will perform a rigid counterclockwise/clockwise rotation. Each conformation must be a *self-avoiding-walk*, i.e., no two amino acids are in the same position. Moreover, the



Instance	Length	Answer	$\mathcal{B}^{FD}_{MV}$
1^7-2	4	N	0.08+0.00
1^7-2	5	Y	0.10+0.00
1^{13}-4	4	N	0.46+13.18
1^{13}-4	5	Y	0.64+25.58
$1(001)^2$-2	4	N	0.08+0.05
$1(001)^2$-2	5	Y	0.11+0.01
$1(001)^3$-4	8	N	0.49+8882.13
$1(001)^3$-4	9	Y	0.58+862.84

Fig. 3. On the left: Initial configuration, a plan, and final configuration with 4 contacts between 1-amino acids. On the right: some results for different sequences, energy levels, and plan lengths.

```
(1)     barrel(5). barrel(7). barrel(12).
(2)     liter(0). liter(1). liter(2). ... liter(11). liter(12).
(3)     fluent(cont(B,L)):- barrel(B),liter(L),L =< B.
(4)     action(fill(X,Y)):- barrel(X),barrel(Y), neq(X,Y).
(5)     causes(fill(X,Y),cont(X,0),[cont(X,LX),cont(Y,LY)]):-
(6)        action(fill(X,Y)), fluent(cont(X,LX)),
(7)        fluent(cont(Y,LY)), Y-LY >= LX.
(8)     causes(fill(X,Y),cont(Y,LYnew),[cont(X,LX),cont(Y,LY)]):-
(9)        action(fill(X,Y)), fluent(cont(X,LX)),
(10)       fluent(cont(Y,LY)), Y-LY >= LX, LYnew is LX + LY.
(11)    causes(fill(X,Y),cont(X,LXnew),[cont(X,LX),cont(Y,LY)]):-
(12)       action(fill(X,Y)), fluent(cont(X,LX)),
(13)       fluent(cont(Y,LY)), LX >= Y-LY, LXnew is LX-Y+LY.
(14)    causes(fill(X,Y),cont(Y,Y),[cont(X,LX),cont(Y,LY)]):-
(15)       action(fill(X,Y)), fluent(cont(X,LX)),
(16)       fluent(cont(Y,LY)), LX >= Y-LY.
(17)    executable(fill(X,Y),[cont(X,LX),cont(Y,LY)]) :-
(18)       action(fill(X,Y)), fluent(cont(X,LX)),
(19)       fluent(cont(Y,LY)), LX > 0, LY < Y.
(20)    caused([ cont(X,LX) ], neg(cont(X,LY)) ) :-
(21)       fluent(cont(X,LX)), fluent(cont(X,LY)),
(22)       botte(X),liter(LX),liter(LY),neq(LX,LY).
(23)    initially(cont(12,12)). initially(cont(7,0)). initially(cont(5,0)).
(24)    goal(cont(12,6)).    goal(cont(7,6)).    goal(cont(5,0)).
```

```
(1)     barrel(5). barrel(7). barrel(12).
(2)     fluent(cont(B),0,B):- barrel(B).
(3)     action(fill(X,Y)):- barrel(X),barrel(Y), neq(X,Y).
(4)     causes(fill(X,Y), cont(X) eq 0, [Y-cont(Y) geq cont(X)]):-
(5)        action(fill(X,Y)).
(6)     causes(fill(X,Y), cont(Y) eq cont(Y)^(-1) + cont(X)^(-1),
(7)        [Y-cont(Y) geq cont(X) ]):- action(fill(X,Y)).
(8)     causes(fill(X,Y), cont(Y) eq Y, [Y-cont(Y) lt cont(X)]):-
(9)        action(fill(X,Y)).
(10)    causes(fill(X,Y), cont(X) eq cont(X)^(-1)-Y+cont(Y)^(-1),
(11)       [Y-cont(Y) lt cont(X)]):- action(fill(X,Y)).
(12)    executable(fill(X,Y), [cont(X) gt 0, cont(Y) lt Y]):-
(13)       action(fill(X,Y)).
(14)    caused([], cont(12) eq 12-cont(5)-cont(7)).
(15)    initially(cont(12) eq 12).
(16)    goal(cont(12) eq cont(7)).
```

Fig. 4. $\mathcal{B}$ description (above) and $\mathcal{B}^{FD}_{MV}$ description (below) of the 12-7-5 barrels problem

chain cannot be broken—i.e., two consecutive amino acids are always at points at distance 1 (i.e., in contact). The goal is to perform a sequence of pivot moves leading to a configuration where at least k non-consecutive amino acids of value 1 are in contact. Fig. 3 shows a possible plan to reach a configuration with 4 contacts. The figure also reports some execution times. Fig. 5 reports the $\mathcal{B}^{FD}_{MV}$ action description encoding this problem. Since the goal is based on the notion of cost of a given state, for which reified constraints are used extensively, a direct encoding in $\mathcal{B}$ does not appear viable.

Let us consider the resolution of the problem starting from the input chain 1001001001. If $N = 10$, asking for a plan of 8 moves and for a solution with cost ≥ 4, our planner finds the plan shown in Fig. 3-center in 862.84s. Notice that, by adding the pair of constraints `holds(x(3) eq 11,1)` and `holds(y(3) eq 11,1)` the time is reduced to 61.05s, and with the constraint `holds(x(4) eq 11,2). holds(y(4)eq 10,2).` the plan is found in only 5.11s. In this case, multivalued fluents and the ability to introduce domain knowledge allows $\mathcal{B}^{FD}_{MV}$ to effectively converge to a solution.

```
(1)     length(10).
(2)     amino(A) :- length(N), interval(A,1,N).
(3)     direction(clock).     direction(antick).
(4)     fluent(x(A),1,M) :- length(N), M is 2*N, amino(A).
(5)     fluent(y(A),1,M) :- length(N), M is 2*N, amino(A).
(6)     fluent(type(A),0,1) :- amino(A).
(7)     fluent(saw,0,1).
(8)     action(pivot(A,D)):- length(N), amino(A), 1<A,A<N, direction(D).
(9)     executable(pivot(A,D),[]) :- action(pivot(A,D)).
(10)    causes(pivot(A,clock),x(B) eq x(A)^(-1) +y(B)^(-1)-y(A)^(-1),[]):-
(11)       action(pivot(A,clock)),amino(B),B > A.
(12)    causes(pivot(A,clock),y(B) eq y(A)^(-1)+x(A)^(-1)-x(B)^(-1),[]):-
(13)       action(pivot(A,clock)),amino(B),B > A.
(14)    causes(pivot(A,antick),x(B) eq x(A)^(-1)-y(B)^(-1)+y(A)^(-1),[]):-
(15)       action(pivot(A,antick)),amino(B),B > A.
(16)    causes(pivot(A,antick),y(B) eq y(A)^(-1)-x(A)^(-1)+x(B)^(-1),[]):-
(17)       action(pivot(A,antick)),amino(B),B > A.
(18)    caused([x(A) eq x(B), y(A) eq y(B)],saw eq 0) :-
(19)       amino(A),amino(B),A<B.
(20)    initially(saw eq 1).
(21)    initially(x(A) eq N) :- length(N), amino(A).
(22)    initially(y(A) eq Y) :- length(N), amino(A),Y is N+A-1.
(23)    initially(type(X) eq 1) :- amino(X), X mod 3 =:= 1.
(24)    initially(type(X) eq 0) :- amino(X), X mod 3 =\= 1.
(25)    goal(saw gt 0).
(26)    state_cost( FE ) :- length(N), auxc(1,4,N,FE).
(27)    auxc(I,J,N,0) :- I > N - 3,!.
(28)    auxc(I,J,N,FE) :- J > N,!,I1 is I+1,J1 is I1+3,auxc(I1,J1,N,FE).
(29)    auxc(I,J,N,FE1+type(I)*type(J)*
(30)       rei(abs(x(I)-x(J))+abs(y(I)-y(J)) eq 1)):-
(31)       J1 is J + 2, auxc(I,J1,N,FE1).
(32)    always(x(1) eq 10). always(y(1) eq 10).
(33)    always(x(2) eq 10). always(y(2) eq 11).
(34)    goal_cost(goal geq 4).
```

Fig. 5. $\mathcal{B}^{FD}_{MV}$ Encoding of the HP-protein folding problem with pivot moves on input of the form 1001001001... starting from a vertical straight line

5.3 Other Examples

We report results from two other planning problems. The first (3x3-puzzle) is an encoding of the 8-tile puzzle problem, where the goal is to find a sequence of moves to re-order the 8 tiles, starting from a random initial position. In the *Community-*M problem, there are M persons, identified by the numbers $1, 2, \ldots, M$. At each time step,

Table 2. Excerpt of experimental results with different instances of various problems (For CLP(FD) and $\mathcal{B}^{FD}_{MV}$ we reported the time required for the constrain and the labelling phases)

Instance	Length	Answer	$\mathcal{B}$				$\mathcal{B}^{FD}_{MV}$
			lparse	Smodels	Cmodels	CLP(FD)	unconstrained plan cost
3x3-puzzle	10	N	45.18	0.83	1.96	0.68+9.23	0.32+11.95
3x3-puzzle	11	Y	45.59	1.85	2.35	0.70+24.10	0.34+27.34
Community-5_{inst1}	6	N	208.39	96.71	3.55	2.70+73.91	0.02+34.86
Community-5_{inst1}	7	Y	205.48	10.57	2.45	3.17+0.18	0.04+0.03
Community-5_{inst2}	6	N	204.40	54.20	3.15	2.67+61.68	0.03+19.40
Community-5_{inst2}	7	Y	208.48	3.69	1.07	3.17+0.17	0.04+0.02

one of the persons, say j, provided (s)he owns more than j dollars, gives j dollars to someone else. The goal consists of reaching a state in which there are no two persons i and j such that the difference between what is owned by i and j is greater than 1. Table 2 lists some results for $M = 5$ and for two variants of the problem: The person i initially owns $i + 1$ dollars ($inst1$) or $2 * i$ dollars ($inst2$).

6 Conclusions and Future Work

The objective of this paper was to initiate an investigation of using constraint logic programming techniques in handling action description languages and planning problems. In particular, we presented an implementation of the language $\mathcal{B}$ using CLP(FD), and reported on its performance. We also presented the action language $\mathcal{B}^{FD}_{MV}$, which allows the use of multivalued fluents and the use of constraints as conditions and consequences of actions. We illustrated the application of $\mathcal{B}^{FD}_{MV}$ to two planning problems. Both languages have been implemented using SICStus Prolog.

The encoding in CLP(FD) allow us to think of extensions in several directions, such as encoding of qualitative and quantitative preferences (a preliminary study has been presented in [12]), introduction of global action constraints for improving efficiency (e.g., alldifferent among states), and use of constraints to represent incomplete states—e.g., to determine most general conditions for the existence of a plan and to conduct conformant planning [13]. We also believe that significant improvements in efficiency could be achieved by delegating parts of the constraint solving process to a dedicated solver (e.g., encoded using a constraint platform such as GECODE).

Acknowledgments. This work is partially supported by MIUR projects PRIN05-015491 and FIRB03-RBNE03B8KK, and by NSF grants 0220590, 0454066, and 0420407. The authors wish to thank Tran Cao Son for the useful discussions and suggestions.

References

[1] Apt, K.: Principles of Constraint Programming. Cambridge University Press, Cambridge (2003)
[2] Baral, C.: Knowledge representation, reasoning and declarative problem solving. Cambridge University Press, Cambridge (2003)
[3] Baral, C., Son, T.C., Tuan, L.C.: A Transition Function based Characterization of Actions with Delayed and Continuous Effects. In: KRR, Morgan Kaufmann, San Francisco (2002)
[4] Dovier, A., Formisano, A., Pontelli, E.: A Comparison of CLP(FD) and ASP Solutions to NP-Complete Problems. In: Gabbrielli, M., Gupta, G. (eds.) ICLP 2005. LNCS, vol. 3668, pp. 67–82. Springer, Heidelberg (2005)
[5] Dovier, A., Formisano, A., Pontelli, E.: An Empirical Study of CLP and ASP Solutions of Combinatorial Problems. J. of Experimental & Theoretical Artificial Intelligence (2007)
[6] Gelfond, M., Lifschitz, V.: Action Languages. Elect. Trans. Artif. Intell. 2, 193–210 (1998)
[7] Giunchiglia, E., Lierler, Y., Maratea, M.: SAT-Based Answer Set Programming. In: Proc. of AAAI'04, AAAI/Mit Press, pp. 61–66 (2004)
[8] Jonsson, A.K., Morris, P.H., Muscettola, N., Rajan, K., Smith, B.D.: Planning in Interplanetary Space: Theory and Practice. In: AIPS (2002)

[9] Lee, J., Lifschitz, V.: Describing Additive Fluents in Action Language C+. IJCAI (2003)
[10] Lopez, A., Bacchus, F.: Generalizing GraphPlan by Formulating Planning as a CSP. In: Proc. of IJCAI (2003)
[11] Marriott, K., Stuckey, P.J.: Programming with Constraints. MIT Press, Cambridge (1998)
[12] Phan, T., Son, T.C., Pontelli, E.: CPP: a Constraint Logic Programming based Planner with Preferences. In: LPNMR, Springer, Heidelberg (2007)
[13] Phan, T., Son, T.C., Baral, C.: Planning with Sensing Actions, Incomplete Information, and Static Causal Laws using Logic Programming. TPLP (to appear)
[14] Reiter, R.: Knowledge in Action: Logical Foundations for Describing and Implementing Dynamical Systems. MIT Press, Cambridge (2001)
[15] Simons, P.: Extending and Implementing the Stable Model Semantics. Doctoral dissertation. Report 58, Helsinki University of Technology (2000)
[16] Son, T.C., Baral, C., McIlraith, S.A.: Planning with different forms of domain-dependent control knowledge. In: Eiter, T., Faber, W., Truszczyński, M. (eds.) LPNMR 2001. LNCS (LNAI), vol. 2173, pp. 226–239. Springer, Heidelberg (2001)
[17] Thielscher, M.: Reasoning about Actions with CHRs and Finite Domain Constraints. In: Stuckey, P.J. (ed.) ICLP 2002. LNCS, vol. 2401, pp. 70–84. Springer, Heidelberg (2002)

Declarative Diagnosis of Temporal Concurrent Constraint Programs

M. Falaschi[1], C. Olarte[2,4], C. Palamidessi[2], and F. Valencia[3]

[1] Dip. Scienze Matematiche e Informatiche, Università di Siena, Italy
moreno.falaschi@unisi.it
[2] INRIA Futurs and LIX, École Polytechnique, France
[3] CNRS and LIX, École Polytechnique, France
{colarte,catuscia, fvalenci}@lix.polytechnique.fr
[4] Dept. Computer Science, Javeriana University, Cali, Colombia

Abstract. We present a framework for the declarative diagnosis of nondeterministic timed concurrent constraint programs. We present a denotational semantics based on a (continuous) immediate consequence operator, $T_{\mathcal{D}}$, which models the process behaviour associated with a program $\mathcal{D}$ given in terms of sequences of constraints. Then, we show that, given the intended specification of $\mathcal{D}$, it is possible to check the correctness of $\mathcal{D}$ by a single step of $T_{\mathcal{D}}$. In order to develop an effective debugging method, we approximate the *denotational semantics* of $\mathcal{D}$. We formalize this method by abstract interpretation techniques, and we derive a finitely terminating abstract diagnosis method, which can be used statically. We define an abstract domain which allows us to approximate the infinite sequences by a finite 'cut'. As a further development we show how to use a specific linear temporal logic for deriving automatically the debugging sequences. Our debugging framework does not require the user to either provide error symptoms in advance or answer questions concerning program correctness. Our method is compositional, that may allow to master the complexity of the debugging methodology.

Keywords: timed concurrent constraint programs, (modular) declarative debugging, denotational semantics, specification logic.

1 Introduction

The main motivation for this work is to provide a methodology for developing effective (modular) debugging tools for timed concurrent constraint (`tcc`) languages.

Finding program bugs is a long-standing problem in software construction. However, current debugging tools for `tcc` do not enforce program correctness adequately as they do not provide means to find bugs in the source code w.r.t. the intended program semantics. We are not aware of the existence of sophisticated debuggers for this class of languages. It would be certainly possible to define some trace debuggers based on suitable extended box models which help display the execution. However, due to the complexity of the semantics of `tcc`

V. Dahl and I. Niemelä (Eds.): ICLP 2007, LNCS 4670, pp. 271–285, 2007.

programs, the information obtained by tracing the execution would be difficult to understand. Several debuggers based on tracing have been defined for different declarative programming languages. For instance [11] in the context of integrated languages. To improve understandability, a graphic debugger for the multi–paradigm concurrent language Curry is provided within the graphical environment CIDER [12] which visualizes the evaluation of expressions and is based on tracing. TeaBag [3] is both a tracer and a runtime debugger provided as an accessory of a Curry virtual machine which handles non–deterministic programs. For Mercury, a visual debugging environment is ViMer [5], which borrows techniques from standard tracers, such as the use of spypoints. In [14], the functional logic programming language NUE-Prolog has a more declarative, algorithmic debugger which uses the declarative semantics of the program and works in the style proposed by Shapiro [19]. Thus, an oracle (typically the user) has to provide the debugger with error symptoms, and has to correctly answer oracle questions driven by proof trees aimed at locating the actual source of errors. Unfortunately, when debugging real code, the questions are often textually large and may be difficult to answer.

Abstract diagnosis [8] is a declarative debugging framework which extends the methodology in [10,19], based on using the immediate consequence operator to identify bugs in logic programs, to diagnosis w.r.t. computed answers. An important advantage of this framework is to be goal independent and not to require the determination of symptoms in advance. In [2], the declarative diagnosis methodology of [8] was generalized to the debugging of functional logic programs, and in [1] it was generalized to functional programs.

In this paper we aim at defining a framework for the abstract diagnosis of `tcc` programs. The idea for abstract debugging follows the methodology in [1], but we have to deal with the extra complexity derived from having constraints, concurrency, and time issues. Moreover, our framework introduces several novelties, since we delevop a compositional semantics and we show how to derive automatically the debugging symptoms from a specification given in terms of a linear temporal logic. We proceed as follows.

First, we associate a (continuous) denotational semantics to our programs. This leads us to a fixpoint characterization of the semantics of `tcc` programs. Then we show that, given the intended specification $\mathcal{I}$ of a program $\mathcal{D}$, we can check the correctness of $\mathcal{D}$ by a single step of this operator. The specification $\mathcal{I}$ may be partial or complete, and can be expressed in several ways: for instance, by (another) `tcc` program, by an assertion language [6] or by sets of constraint sequences (in the case when it is finite). The diagnosis is based on the detection of *incorrect rules* and *uncovered constraint sequences*, which both have a bottom-up definition (in terms of one application of the continuous operator to the abstract specification). It is worth noting that no fixpoint computation is required, since the (concrete) semantics does not need to be computed.

In order to provide a practical methodology, we also present an effective debugging methodology which is based on abstract interpretation. Following an idea inspired in [4,8], and developed in [1] we use *over* and *under* specifications $\mathcal{I}^+$

and $\mathcal{I}^-$ to correctly over- (resp. under-) approximate the intended specification $\mathcal{I}$ of the semantics. We then use these two sets respectively for approximating the input and the output of the operator associated by the denotational semantics to a given program, and by a simple static test we can determine whether some of the program rules are wrong. The method is sound in the sense that each error which is found by using $\mathcal{I}^+, \mathcal{I}^-$ is really a bug w.r.t. $\mathcal{I}$.

The rest of the paper is organized as follows. Section 2 recalls the syntax of the `ntcc` calculus, a non-deterministic extension of `tcc` model. Section 3 and 4 are devoted to present a denotational semantics for `ntcc` programs. We also formulate an operational semantics and show the correspondence with the least fixpoint semantics. Section 5 provides an abstract semantics which correctly approximates the fixpoint semantics of $\mathcal{D}$. In Section 6 we introduce the general notions of incorrectness and insufficiency symptoms. We give an example of a possible abstract domain, by considering a 'cut' to finite depth of the constraint sequences and show how it can be used to detect errors in some benchmark programs. We also show how to derive automatically the elements (sequences) of the domain for debugging from a specification in linear temporal logic. Section 7 concludes.

2 The Language and the Semantic Framework

In this section we describe the `ntcc` calculus [15], a non-deterministic temporal extension of the concurrent constraint (cc) model [18].

2.1 Constraint Systems

cc languages are parametrized by a *constraint system.* A constraint system provides a signature from which syntactically denotable objects called *constraints* can be constructed, and an entailment relation $\models$ specifying interdependencies between such constraints.

Definition 1. ***(Constraint System).*** *A* constraint system *is a pair* (Σ, Δ) *where* Σ *is a signature specifying constants, functions and predicate symbols, and* Δ *is a consistent first-order theory over* Σ*.*

Given a constraint system (Σ, Δ), let $\mathcal{L}$ be the underlying first-order language $(\Sigma, \mathcal{V}, \mathcal{S})$, where $\mathcal{V}$ is a countable set of variables and $\mathcal{S}$ is the set of logical symbols including $\wedge$, $\vee$, $\Rightarrow$, $\exists$, $\forall$, `true` and `false` which denote logical conjunction, disjunction, implication, existential and universal quantification, and the always true and false predicates, respectively. *Constraints,* denoted by $c, d, \ldots$ are first-order formulae over $\mathcal{L}$. We say that c *entails* d in Δ, written $c \models d$, if the formula $c \Rightarrow d$ holds in all models of Δ. As usual, we shall require $\models$ to be decidable.

We say that c is equivalent to d, written $c \approx d$, iff $c \models d$ and $d \models c$. Henceforth, $\mathcal{C}$ is the set of constraints modulo $\approx$ in (Σ, Δ).

2.2 Process Syntax

In `ntcc` time is conceptually divided into *discrete intervals* (or *time-units*). Intuitively, in a particular time interval, a cc process P receives a stimulus

(i.e. a constraint) from the environment, it executes with this stimulus as the initial store, and when it reaches its resting point, it responds to the environment with the resulting store. The resting point also determines a residual process Q, which is then executed in the next time interval.

Definition 2 (Syntax). *Processes P, Q, $\ldots \in$ Proc are built from constraints $c \in \mathcal{C}$ and variables $x \in \mathcal{V}$ in the underlying constraint system (Σ, Δ) as follows:*

$$P, Q, \ldots ::= \mathbf{skip} \mid \mathbf{tell}(c) \mid \sum_{j \in J} \mathbf{when}\ c_j\ \mathbf{do}\ P_j \mid P \parallel Q \mid (\mathbf{local}\ x)\ \mathbf{in}\ P \mid \mathbf{next}\ P \mid \star P \mid p(x)$$

Process **skip** does nothing. Process $\mathbf{tell}(c)$ adds the constraint c to the current store, thus making c available to other processes in the current time interval. Process $\sum_{j \in J} \mathbf{when}\ c_j\ \mathbf{do}\ P_j$ where J is a finite set of indexes, represents a process that non-deterministically choose a process P_j s.t c_j is entailed by the current store. The chosen alternative, if any, precludes the others. If no choice is possible in the current time unit, all the alternatives are precluded from execution. We shall use $\sum_{j \in J} P_j$ when the guards are `true` ("blind-choice") and we omit $\sum_{j \in J}$ when J is a singleton. Process $P \parallel Q$ represents the parallel composition of P and Q. Process $(\mathbf{local}\ x)\ \mathbf{in}\ P$ behaves like P, except that all the information on x produced by P can only be seen by P and the information on x produced by other processes cannot be seen by P.

The only move of $\mathbf{next}\ P$ is a unit-delay for the activation of P. We use $\mathbf{next}^n(P)$ as an abbreviation for $\mathbf{next}(\mathbf{next}(\ldots(\mathbf{next}\ P)\ldots))$, where **next** is repeated n times. $\star P$ represents an arbitrary long but finite delay for the activation of P. It can be viewed as $P + \mathbf{next}\ P + \mathbf{next}^2 P$....

Recursion in `ntcc` is defined by means of *processes definitions* of the form

$$p(x_1, .., x_n) \stackrel{\mathtt{def}}{=} A$$

$p(y_1, ..., y_n)$ is an *invocation* and intuitively the body of the process definition (i.e. A) is executed replacing the formal parameter $\boldsymbol{x}$ by the actual parameters $\boldsymbol{y}$. When $|\boldsymbol{x}| = 0$, we shall omit the parenthesis.

To avoid non-terminating sequences of internal reductions (i.e non-terminating computation within a time interval), recursive calls must be guarded in the context of **next** (see [15] for further details). In what follows $\mathcal{D}$ shall denote a set of processes definitions.

3 Operational Semantics

Operationally, the information in the current time unit is represented as a constraint $c \in \mathcal{C}$, so-called *store*. Following standard lines [18], we extend the syntax with a construct $(\mathbf{local}\ x, d)\ \mathbf{in}\ P$ which represents the evolution of a process of the form $(\mathbf{local}\ x)\ \mathbf{in} Q$, where d is the local information (or private store) produced during this evolution. Initially d is "empty", so we regard $(\mathbf{local}\ x)\ \mathbf{in}\ P$ as $(\mathbf{local}\ x, \mathtt{true})\ \mathbf{in}\ P$.

The operational semantics will be given in terms of the reduction relations $\longrightarrow$, $\Longrightarrow \subseteq Proc \times \mathcal{C} \times Proc \times \mathcal{C}$ defined in Table 1. The *internal transition* $\langle P, c\rangle \longrightarrow \langle Q, d\rangle$ should be read as "P with store c reduces, in one internal step, to Q with store d". The *observable transition* $P \stackrel{(c,d)}{\Longrightarrow} Q$ should be read as "P on input c from the environment, reduces in one time unit to Q and outputs d to the environment". Process Q is the process to be executed in the next time unit. Such a reduction is obtained from a sequence of internal reductions starting in P with initial store c and terminating in a process Q' with store d. Crudely speaking, Q is obtained by removing from Q' what was meant to be executed only during the current time interval. In `ntcc` the store d is not automatically transferred to the next time unit. If needed, information in d can be transfered to next time unit by process P.

Let us describe some of the rules for the internal transitions. Rule $TELL$ says that constraint c is added to the current store d. SUM chooses non-deterministically a process P_j for execution if its guard (c_j) can be entailed from the store. In PAR_r and PAR_l, if a process P can evolve, this evolution can take place in the presence of some other process Q running in parallel. Rule LOC is the standard rule for locality (or hiding) (see [18,9]). $CALL$ shows how a process definition is replaced by its body according to the set of process definition in $\mathcal{D}$. Finally, $STAR$ executes P in some time-unit in the future.

Let us now describe the rule for the observable transitions. Rule OBS says that an observable transition from P labeled by (c, d) is obtained by performing a terminating sequence of internal transitions from $\langle P, c\rangle$ to $\langle Q, d\rangle$, for some Q. The process to be executed in the next time interval, $F(Q)$ ("future" of Q), is obtained as follows:

Definition 3. *(**Future Function**). Let $F : Proc \rightharpoonup Proc$ be defined by*

$$F(P) = \begin{cases} \mathbf{skip} & \text{if } P = \mathbf{skip} \\ \mathbf{skip} & \text{if } P = \mathbf{when}\ c\ \mathbf{do}\ Q \\ F(P_1) \parallel F(P_2) & \text{if } P = P_1 \parallel P_2 \\ (\mathbf{local}\ x)\ \mathbf{in}\, F(Q) & \text{if } P = (\mathbf{local}\ x, c)\ \mathbf{in}\, Q \\ Q & \text{if } P = \mathbf{next}\, Q \end{cases}$$

In this paper we are interested in the so called *quiescent input sequences* of a process. Intuitively, those are sequences of constraints on input of which P can run without adding any information, wherefore what we observe is that the input and the output coincide. The set of quiescent sequences of a process P is thus equivalent to the observation of all sequences that P can possibly output under the influence of *arbitrary* environment. We shall refer to the set of quiescent sequences of P as the *strongest postcondition* of P written $sp(P)$, w.r.t the set of sequences of constraints denoted by $\mathcal{C}^*$. In what follows, let s, s_1,etc range over elements in $\mathcal{C}^*$ and $s(i)$ be the *i-th* element in s. We note that, as in Dijkstra's strongest postcondition approach, proving whether P satisfies a given (temporal) property A, in the presence of any environment, reduces to proving whether $sp(P)$ is included in the set of sequences satisfying A [15].

Table 1. Rules for the internal reduction $\longrightarrow$ and the observable reduction $\Longrightarrow$

$$\text{TELL}\ \frac{}{\langle \mathbf{tell}(c), d\rangle \longrightarrow \langle \mathbf{skip}, d \wedge c\rangle} \qquad \text{SUM}\ \frac{d \models c_j \quad j \in J}{\left\langle \sum_{j\in J} \mathbf{when}\ c_j\ \mathbf{do}\ P_j, d\right\rangle \longrightarrow \langle P_j, d\rangle}$$

$$\text{PAR}_\text{r}\ \frac{\langle P, c\rangle \longrightarrow \langle P', d\rangle}{\langle P \parallel Q, c\rangle \longrightarrow \langle P' \parallel Q, d\rangle} \qquad \text{PAR}_\text{l}\ \frac{\langle Q, c\rangle \longrightarrow \langle Q', d\rangle}{\langle P \parallel Q, c\rangle \longrightarrow \langle P \parallel Q', d\rangle}$$

$$\text{CALL}\ \frac{p(x) :- A \in \mathcal{D}}{\langle p(x), d\rangle \longrightarrow \langle A, d\rangle} \qquad \text{STAR}\ \frac{n \geq 0}{\langle \star P, d\rangle \longrightarrow \langle \mathbf{next}^{\,n} P, d\rangle}$$

$$\text{LOC}\ \frac{\langle P, c \wedge (\exists_x d)\rangle \longrightarrow \langle P', c' \wedge (\exists_x d)\rangle}{\langle (\mathbf{local}\ x, c)\ \mathbf{in}\ P, d \wedge \exists_x c\rangle \longrightarrow \langle (\mathbf{local}\ (x, c'))\ \mathbf{in}\ P', d \wedge \exists_x c'\rangle}$$

$$\text{OBS}\ \frac{\langle P, c\rangle \longrightarrow^* \langle Q, d\rangle \not\longrightarrow \quad R \equiv F(Q)}{P \xRightarrow{(c,d)} R}$$

Definition 4 (Observables). *The behavioral observations that can be made of a process are given by the* strongest postcondition (or quiescent) *behavior of* P

$$sp(P) = \{s \mid P \xRightarrow{(s,s)}{}^{*} \ \textit{for some}\ s \in \mathcal{C}^*\}.$$

where $P \xRightarrow{(s,s)}{}^{*} \equiv P \xRightarrow{(s(1),s(1))} P' \xRightarrow{(s(2),s(2))} \ldots$

4 Denotational Semantics

In this section we give a denotational characterization of the strongest postcondition observables of `ntcc` following ideas developed in [15] and [17].

The denotational semantics is defined as a function $[\![\cdot]\!]$ which associates to each process a set of finite sequences of constraints, namely $[\![\cdot]\!] : (\mathit{Proc} \to \mathcal{P}(\mathcal{C}^*)) \to (\mathit{ProcHeads} \to \mathcal{P}(\mathcal{C}^*))$, where *ProcHeads* denotes the set of process names with their formal parameters. The definition of this function is given in Table 2. We use $\exists_x s$ to represent the sequence obtained by applying $\exists_x$ to each constraint in s. We call the functions in the domain $\mathit{ProcHeads} \to \mathcal{P}(\mathcal{C}^*)$ as *Interpretations*. We consider the following order on (infinite) sequences $s \leq s'$ iff $\forall i.s'(i) \models s(i)$. For what concern finite sequences we can order them similarly. Let s and s' be finite sequences, then if s has length less or equal to s' then $s \leq s'$ iff $\forall i = 1, 2, \ldots \mathit{length}(s).s'(i) \models s(i)$.

Intuitively, $[\![P]\!]$ is meant to capture the quiescent sequences of a process P. For instance, all sequence whose first element is stronger than c is quiescent for $\mathbf{tell}(c)$ (D1). Process $\mathbf{next}\, P$ has not influence on the first element of a sequence, thus $d.s$ is quiescent for it if s is quiescent for P (D5). The semantics

for a procedure call $p(x)$ is directly given by the interpretation I provided (D6). A sequence is quiescent for $\star P$ if there is a suffix of it which is quiescent for P (D7). The other rules can be explained analogously.

Table 2. Denotational semantics of `ntcc`

D1	$[\![\mathbf{tell}(c)]\!]_I \;=\; \{d.s \mid d \models c, s \in \mathcal{C}^*\}$
D2	$[\![\sum_{j\in J} \mathbf{when}\ c_j\ \mathbf{do}\ P_j]\!]_J \;=\; \bigcup_{j\in J}\{d.s \mid d \models c_j, d.s \in [\![P_j]\!]_J\}$ $\cup$ $\bigcap_{j\in J}\{d.s \mid d \not\models c_j, d.s \in \mathcal{C}^*\}$
D3	$[\![P \parallel Q]\!]_I \;=\; [\![P]\!]_I \cap [\![Q]\!]_I$
D4	$[\![\mathbf{local}\ x\ \mathbf{in}\ P]\!]_I \;=\; \{s \mid$ there exists $s' \in [\![P]\!]_I$ s.t. $\exists_x s = \exists_x s'\}$
D5	$[\![\mathbf{next}\ P]\!]_I \;=\; \mathcal{C}^1 \cup \{d.s \mid d \in \mathcal{C}, s \in [\![P]\!]_I\}$
D6	$[\![p(x)]\!]_I \;=\; I(p(x))$
D7	$[\![\star P]\!]_I \;=\; \{s.s' \mid s \in \mathcal{C}^*$ and $s' \in [\![P]\!]_I\}$

Formally the semantics is defined as the least fixed-point of the corresponding operator $T_{\mathcal{D}} \in (\mathit{ProcHeads} \to \mathcal{P}(\mathcal{C}^*)) \to (\mathit{ProcHeads} \to \mathcal{P}(\mathcal{C}^*))$

$$T_{\mathcal{D}}(I)(p(x)) = \quad [\![\exists_y (A \parallel d_{xy})]\!]_I \text{ if } p(y) :- A \in \mathcal{D}$$

where d_{xy} is the diagonal element used to represent parameter passing (see [18] for details). The following example illustrates the $T_{\mathcal{D}}$ operator.

Example 1. Consider a control system that must exhibit the control signal *stop* when some component malfunctions (i.e. the environment introduces as stimulus the constraint $failure$). The following (incorrect) program intends to implement such a system:

$$\begin{aligned} &\mathit{control} = \mathbf{when}\ \mathit{failure}\ \mathbf{do\ next}\ \mathit{action} \parallel \mathbf{next}\ \mathit{control} \\ &\mathit{action} = \mathbf{tell}(\mathit{stop}) \end{aligned}$$

Starting from the bottom interpretation $I_\perp$ assigning to every process the empty set (i.e $I_\perp(\mathit{control}) = I_\perp(\mathit{action}) = \emptyset$), $T_{\mathcal{D}}$ is computed as follows:

$$\begin{aligned} T_{\mathcal{D}}(I_\perp)(\mathit{control}) &= [\![\mathbf{when}\ \mathit{failure}\ \mathbf{do\ next}\ \mathit{action} \parallel \mathbf{next}\ \mathit{control}]\!]_{I_\perp} \\ &= \{d_1.d_2.s | d_1 \models \mathit{failure} \Rightarrow d_1.d_2.s \in [\![\mathbf{next}\ \mathit{action}]\!]_{I_\perp}\} \\ &\quad \cap \{\mathcal{C}^1 \cup \{d_1.s | s \in [\![\mathit{control}]\!]_{I_\perp}\}\} \\ &= \{d_1.d_2.s | d_1 \models \mathit{failure} \Rightarrow d_1.d_2.s \in \{\mathcal{C}^1 \cup \emptyset\}\} \cap \mathcal{C}^1 = \mathcal{C}^1 \\ T_{\mathcal{D}}(I_\perp)(\mathit{action}) &= [\![\mathbf{tell}(\mathit{stop})]\!]_{I_\perp} = \{d_1.s | d_1 \models \mathit{stop}\} \end{aligned}$$

Let now $I_1 = T_{\mathcal{D}}(I_\perp)$:

$$\begin{aligned} T_{\mathcal{D}}(I_1)(\mathit{control}) &= \{d_1.d_2.s | d_1 \models \mathit{failure} \Rightarrow d_1.d_2.s \in [\![\mathbf{next}\ \mathit{action}]\!]_{I_1}\} \cap \\ &\quad \big(\mathcal{C}^1 \cup \{d.s | s \in [\![\mathit{control}]\!]_{I_1}\}\big) \\ &= \{d_1.d_2.s | d_1 \models \mathit{failure} \Rightarrow d_1.d_2.s \in \big(\mathcal{C}^1 \cup \{d_1.d_2.s | d_2 \models \mathit{stop}\}\big)\} \\ &\quad \cap \big(\mathcal{C}^1 \cup \mathcal{C}^2\big) = \{d_1.d_2 | d_1 \models \mathit{failure} \Rightarrow d_2 \models \mathit{stop}\} \end{aligned}$$

Table 3. Abstract denotational semantics for `ntcc`

D1	$[\![\mathbf{tell}(c)]\!]^\tau_I \;=\; \tau(\{d.\beta \mid d \models c, \beta \in \mathcal{C}^*\})$
D2	$[\![\sum_{j\in J} \mathbf{when}\ c_j\ \mathbf{do}\ P_j]\!]^\tau_J \;=\; \bigcup_{i\in J} \tau(\{d.\beta \mid d \models c_i, d.\beta \in [\![P_i]\!]^\tau_I\}) \cup \bigcap_{i\in J} \tau(\{d.\beta \mid d \not\models c_i, d.\beta \in \mathcal{C}^*\})$
D3	$[\![P \parallel Q]\!]^\tau_I \;=\; [\![P]\!]^\tau_I \cap [\![Q]\!]^\tau_I$
D4	$[\![\mathbf{local}\ x\ \mathbf{in}\ P]\!]^\tau_I \;=\; \{\beta \mid$ there exists $\beta' \in [\![P]\!]^\tau_I$ s.t. $\exists_x\beta = \exists_x\beta'\}$
D5	$[\![\mathbf{next}\ P]\!]_I \;=\; \tau(\mathcal{C}) \cup \tau(\{d.\beta \mid d \in \mathcal{C}, \beta \in [\![P]\!]_I\})$
D6	$[\![p(x)]\!]^\tau_I \;=\; \tau(I(p(x)))$
D7	$[\![\star P]\!]^\tau_I \;=\; \tau(\{\beta.\beta' \mid \beta \in \mathcal{C}^*, \beta' \in [\![P]\!]^\tau_I\})$

5 Abstract Semantics

In this section, starting from the fixpoint semantics in Section 4, we develop an abstract semantics which approximates the observable behavior of the program and is adequate for modular data-flow analysis.

We will focus our attention now on a special class of abstract interpretations which are obtained from what we call a *sequence abstraction* $\tau : (\mathcal{C}^*, \leq) \to (AS, \lesssim)$. We require that $(AS, \lesssim)$ is noetherian and that τ is surjective and monotone.

We start by choosing as abstract domain $\mathbb{A} := \mathcal{P}(AS)$, ordered by an extension to sets $X \lesssim_S Y$ iff $\forall x \in X \exists y \in Y : (x \lesssim y)$. We will call elements of $\mathbb{A}$ abstract sequences. The concrete domain $\mathbb{E}$ is $\mathcal{P}(\mathcal{C}^*)$, ordered by set inclusion.

Then we can lift τ to a Galois Insertion of $\mathbb{A}$ into $\mathbb{E}$ by defining

$$\alpha(E)(p) := \{\tau(s) \mid s \in E(p)\}$$
$$\gamma(A)(p) := \{s \mid \tau(s) \in A(p)\}$$

for some $ProcHead\ p$. Then, we can lift in the standard way to abstract Interpretations the approximation induced by the above abstraction. The Interpretation $f : ProcHeads \to \mathcal{P}(\mathcal{C}^*)$ can be approximated by $f^\alpha : ProcHeads \to \alpha(\mathcal{P}(\mathcal{C}^*))$.

The only requirement we put on τ is that $\alpha(Sem(\mathcal{D}))$ is finite, where $Sem(\mathcal{D})$ is the (concrete) semantics of $\mathcal{D}$.

Now we can derive the optimal abstract version of $T_\mathcal{D}$ as $T^\alpha_\mathcal{D} := \alpha \circ T_\mathcal{D} \circ \gamma$. By applying the previous definition of α and γ this turns out to be equivalent to the following definition.

Definition 5. *Let τ be a sequence abstraction, $X \in \mathbb{A}$ be an abstract Interpretation. Then,*

$$T^\alpha_\mathcal{D}(X)(p(x)) = \tau([\![\exists_y(A \parallel d_{xy})]\!]_X) \ \textit{if}\ p(y) :- A \in \mathcal{D}$$

where the abstract denotational semantics is defined in Table 3.

Abstract interpretation theory assures that $T^\alpha_\mathcal{D} \uparrow \omega$ is the best correct approximation of $Sem(\mathcal{D})$. Correct means $\alpha(Sem(\mathcal{D})) \sqsubseteq T^\alpha_\mathcal{D} \uparrow \omega$ and best means that it is the minimum w.r.t. $\sqsubseteq$ of all correct approximations.

Now we can define the abstract semantics as the least fixpoint of this (continuous) operator.

Definition 6. *The abstract least fixpoint semantics of a program $\mathcal{D}$ is defined as $\mathcal{F}^\alpha(\mathcal{D}) = T^\alpha_\mathcal{D} \uparrow \omega$.*

By our finiteness assumption on τ we are guaranteed to reach the fixpoint in a finite number of steps, that is, there exists $h \in \mathbb{N}$ s.t. $T^\alpha_\mathcal{D} \uparrow \omega = T^\alpha_\mathcal{D} \uparrow h$.

5.1 A Case Study: The Domain *Sequence*(*k*)

Now we show how to approximate a set of computed sequences by means of a *sequence*(k) cut [20], i.e., by using a sequence abstraction which approximates sequences having a length bigger than k. Sequences are approximated by replacing each constraint at length bigger than k with the constraint `false`.

First of all we define the sequence abstraction $s/_k$ (for $k \geq 0$) as the *sequence*(k) cut of the concrete sequence s. We mean by that $s/_k = s'$, where $s'(i) = s(i)$ for $i \leq k-1$ and $s'(k) =$ `false`. We denote by $S/_k$ the set of sequences cut at length k. The abstract domain $\mathbb{A}$ is thus $\mathcal{P}(S/_k)$ ordered by the extension of ordering $\leq$ to sets, i.e. $X \leq_S Y$ iff $\forall x \in X \, \exists y \in Y : (x \leq y)$ The resulting abstraction α is $\kappa(E)(p) := \{s/_k \mid s \in I(p)\}$. By abuse, we will denote by the same notation $\leq_S$ its standard extension to interpretations. A sequence $s = c_1, \ldots, c_n$ is complete if $c_n \neq false$.

We provide a simple and effective mechanism to compute the abstract fixpoint semantics.

Definition 7. *The effective abstract least fixpoint semantics of a program $\mathcal{D}$, is defined as $\mathcal{F}^\kappa(\mathcal{D}) = T^\kappa_\mathcal{D} \uparrow \omega$.*

Proposition 1 (Correctness). *Let $\mathcal{D}$ be a program and $k > 0$.*

1. *$\mathcal{F}^\kappa(\mathcal{D}) \leq_S \kappa(\mathcal{F}(\mathcal{D})) \leq_S \mathcal{F}(\mathcal{D})$.*
2. *Let Y be Fix^κ $\mathcal{D}$, and let Z be Fix $\mathcal{D}$.*
   For all $s \in \mathcal{F}^\kappa(\mathcal{D})(Y)(p(x))$ that are complete, $s \in \mathcal{F}(\mathcal{D})(Z)(p(x))$.

6 Abstract Diagnosis of `ntcc` Programs

In this section, we recall the framework developed in [1]. We modify it in order to be able to handle the specific features of `ntcc`. Then we show that the main correctness and completeness results can be extended to `ntcc`. We also develop a completely new methodology based on the linear temporal logic (LTL) associated to the calculus [15]. The basic idea is that given a specification in the logic we can generate automatically the sequences which we use for debugging the program. Let us recall the syntax and semantics of this logic (see [15] for further details).

Formulae in the `ntcc` LTL are given by the following grammar:

$$A, B, \ldots : c \mid A \dot{\Rightarrow} A \mid \dot{\neg} A \mid \dot{\exists}_x A \mid \circ A \mid \Diamond A \mid \Box A$$

c is a constraint. $\dot{\Rightarrow}$, $\dot{\neg}$ and $\dot{\exists}_x$ represent the linear-temporal logic implication, negation and existential quantification, respectively. These symbols should not

be confused with their counterpart in the constraint system (i.e $\Rightarrow$, $\neg$ and $\exists$). Symbols $\circ$, $\Box$ and $\Diamond$ denote the temporal operators *next*, *always* and *eventually*.

The interpretation structures of formulae in this logic are infinite sequences of constraints. We say that $\beta \in C^*$ is a model of (or that it satisfies) A, notation $\beta \models A$, if $\langle \beta, 1\rangle \models A$ where:

$$
\begin{array}{ll}
\langle \beta, i\rangle \models c & \text{iff } \beta(i) \models c \\
\langle \beta, i\rangle \models \dot{\neg} A & \text{iff } \langle \beta, i\rangle \not\models \ A \\
\langle \beta, i\rangle \models A_1 \dot{\Rightarrow} A_2 & \text{iff } \langle \beta, i\rangle \models A_1 \ implies \langle \beta, i\rangle \models A_2 \\
\langle \beta, i\rangle \models \circ A & \text{iff } \langle \beta, i+1\rangle \models A \\
\langle \beta, i\rangle \models \Box A & \text{iff } \forall_{j \geq i} \langle \beta, j\rangle \models A \\
\langle \beta, i\rangle \models \Diamond A & \text{iff } \exists_{j \geq i} \ s.t. \langle \beta, j\rangle \models A \\
\langle \beta, i\rangle \models \dot{\exists}_x A & \text{iff exists } \beta' \text{ s.t. } \exists_x \beta = \exists_x \beta'
\end{array}
$$

The LTL is then used to specify (temporal) properties of programs:

Definition 8. *We say that P satisfies some property A written $P \vdash A$, iff $sp(P) \subseteq [\![A]\!]$ where $[\![A]\!]$ is the collection of all models of A, i.e., $[\![A]\!] = \{\beta \models A\}$.*

Program properties which can be of interest are Galois Insertions between the concrete domain (the set of interpretations ordered pointwise) and the abstract domain chosen to model the property. The following Definition extends to abstract diagnosis the definitions given in [19,10,13] for declarative diagnosis. In the following, $\mathcal{I}^\alpha$ is the specification of the intended behavior of a program.

Definition 9. *[1] Let $\mathcal{D}$ be a program and α be a property.*

1. *$\mathcal{D}$ is* partially correct *w.r.t. $\mathcal{I}^\alpha$ if $\mathcal{I}^\alpha \sqsubseteq \alpha(Sem(\mathcal{D}))$.*
2. *$\mathcal{D}$ is* complete *w.r.t. $\mathcal{I}^\alpha$ if $\alpha(Sem(\mathcal{D})) \sqsubseteq \mathcal{I}^\alpha$.*
3. *$\mathcal{D}$ is* totally correct *w.r.t. $\mathcal{I}^\alpha$, if it is partially correct and complete.*

Note that the above definition is given in terms of the abstraction of the concrete semantics $\alpha(Sem(\mathcal{D}))$ and not in terms of the (possibly less precise) abstract semantics $Sem^\alpha(\mathcal{D})$. This means that $\mathcal{I}^\alpha$ is the abstraction of the intended concrete semantics of $\mathcal{D}$. In other words, the specifier can only reason in terms of the properties of the expected concrete semantics without being concerned with (approximate) abstract computations.

The *diagnosis* determines the "basic" symptoms and, in the case of incorrectness, the relevant rule in the program. This is modeled by the definitions of *abstractly incorrect rule* and *abstract uncovered equation*.

Definition 10. *Let r be a procedure definition. Then r is* abstractly incorrect *if $\exists p(x) : -A \in \mathcal{D}. T^\alpha_{\{r\}}(\mathcal{I}^\alpha)(p(x)) \not\sqsubseteq \mathcal{I}^\alpha$.*

Informally, r is abstractly incorrect if it derives a wrong abstract element from the intended semantics.

Definition 11. *Let $\mathcal{D}$ be a program. $\mathcal{D}$ has abstract uncovered elements if $\exists p(x) : -A \in \mathcal{D}. \mathcal{I}^\alpha \not\sqsubseteq T^\alpha_{\mathcal{D}}(\mathcal{I}^\alpha)(p(x))$.*

Informally, a sequence s is uncovered if there are no rules deriving it from the intended semantics. It is worth noting that checking the conditions of Definitions 10 and 11 requires one application of $T^{\alpha}_{\mathcal{D}}$ to $\mathcal{I}^{\alpha}$, while the standard detection based on symptoms [19] would require the construction of $\alpha(Sem(\mathcal{D}))$ and therefore a fixpoint computation.

Now, we want to recall the properties of the diagnosis method. The proofs of the following theorems in this section are extensions of those in [1].

Theorem 1. *[1] If there are no abstractly incorrect rules in $\mathcal{D}$, then $\mathcal{D}$ is partially correct w.r.t. $\mathcal{I}^{\alpha}$.*

Theorem 2. *[1] Let $\mathcal{D}$ be partially correct w.r.t. $\mathcal{I}^{\alpha}$. If $\mathcal{D}$ has abstract uncovered elements then $\mathcal{D}$ is not complete.*

Abstract incorrect rules are in general just a hint about a possible source of errors. If an abstract incorrect rule is detected, one would still have to check on the abstraction of the concrete semantics if there is indeed a bug. This is obviously unfeasible in an automatic way. However we will see that, by adding to the scheme an under-approximation of the intended specification, something worthwhile can still be done.

Real errors can be expressed as incorrect rules according to the following definition.

Definition 12. *Let r be a process definition. Then r is* incorrect *if there exists a sequence s such that $s \in T_{\{r\}}(\mathcal{I})$ and $s \notin \mathcal{I}$.*

Definition 13. *Let $\mathcal{D}$ be a program. Then $\mathcal{D}$ has an* uncovered element *if there exists a sequence s such that $s \in \mathcal{I}$ and $s \notin T_{\mathcal{D}}(\mathcal{I})$.*

The check of Definition 12 (as claimed above) is not effective. This task can be (partially) accomplished by an automatic tool by choosing a suitable under-approximation $\mathcal{I}^c$ of the specification $\mathcal{I}$, $\gamma(\mathcal{I}^c) \subseteq \mathcal{I}$ (hence $\alpha(\mathcal{I}) \sqsubseteq \mathcal{I}^c$), and checking the behavior of an abstractly incorrect rule against it.

Definition 14. *Let r be a process definition. Then r is* provably incorrect *using α if $\exists p(x) : -A \in \mathcal{D}.T^{\alpha}_{\{r\}}(\alpha(\mathcal{I}^c))(p(x)) \not\sqsubseteq \mathcal{I}^{\alpha}$.*

Definition 15. *Let $\mathcal{D}$ be a program. Then $\mathcal{D}$ has* provably uncovered *elements using α if $\exists p(x) : -A \in \mathcal{D}.\alpha(\mathcal{I}^c) \not\sqsubseteq T^{\alpha}_{\mathcal{D}}(\mathcal{I}^{\alpha})(p(x))$.*

The name "provably incorrect using α" is justified by the following theorem.

Theorem 3. *[1] Let r be a program rule (process) and $\mathcal{I}^c$ such that $(\gamma\alpha)(\mathcal{I}^c) = \mathcal{I}^c$. Then if r is provably incorrect using α it is also incorrect.*

By choosing a suitable under-approximation we can refine the check for wrong rules. For all abstractly incorrect rules we check if they are provably incorrect using α. If it so then we report an error, otherwise we can just issue a warning.

As we will see in the following, this property holds (for example) for our case study. By Proposition 1 the condition $(\gamma\alpha)(\mathcal{I}^c) = \mathcal{I}^c$ is trivially satisfied by any

subset of the 'finite' sequences in the over-approximation. We can also consider a finite subset of the sequences in which we do the following change: if a sequence is such that all its elements starting from the length k are equal to **false**, we replace such elements by **true**.

Theorem 4. *[1] Let $\mathcal{D}$ be a program. If $\mathcal{D}$ has a provably uncovered element using α, then $\mathcal{D}$ is not complete.*

Abstract uncovered elements are provably uncovered using α. However, Theorem 4 allows us to catch other incompleteness bugs that cannot be detected by using Theorem 2 since there are provably uncovered elements using α which are not abstractly uncovered.

The diagnosis w.r.t. approximate properties is always effective, because the abstract specification is finite. As one can expect, the results may be weaker than those that can be achieved on concrete domains just because of approximation:

- every incorrectness error is identified by an abstractly incorrect rule. However an abstractly incorrect rule does not always correspond to a bug. Anyway,
- every abstractly incorrect rule which is provably incorrect using α corresponds to an error.
- provably uncovered equations always correspond to incompleteness bugs.
- there exists no sufficient condition for completeness.

6.1 Our Case Study

An efficient debugger can be based on the notion of over-approximation and under-approximation for the intended fixpoint semantics that we have introduced. The basic idea is to consider two sets to verify partial correctness and determine program bugs: $\mathcal{I}^\alpha$ which over-approximates the intended semantics $\mathcal{I}$ (that is, $\mathcal{I} \subseteq \gamma(\mathcal{I}^\alpha)$) and $\mathcal{I}^c$ which under-approximates $\mathcal{I}$ (that is, $\gamma(\mathcal{I}^c) \subseteq \mathcal{I}$).

Let us now derive an efficient debugger by choosing suitable instances of our general framework. Let α be the *sequence*(k) abstraction κ of the program that we have defined in previous section. Thus we choose $\mathcal{I}^\kappa = \mathcal{F}^\kappa(\mathcal{I})$ as an over-approximation of the values of a program. We can consider any of the sets defined in the works of [4,7] as an under-approximation of $\mathcal{I}$. In concrete, we take the finite abstract sequences of $\mathcal{I}^\kappa$ as $\mathcal{I}^c$ but replacing the constraint **false** by **true** after the position κ. This provides a simple albeit useful debugging scheme which is satisfactory in practice.

Another possibility is to generate the sequences for the under and the over-approximation from the sequences that are models for a LTL formula specifying the intended behavior of the program. Let us illustrate the method by using the guiding example.

Example 2. Let first consider what the intended behaviour of the system proposed in Example 1 should be. We require that as soon as the constraint $failure$ can be entailed from the store, the system must exhibit the constraint *stop*. This

specification can be provided by means of another (correct) program or by a formula in the LTL. Let us explore both cases.

Let $control'$ and $action'$ be the intended system:

$$control' = \textbf{when}\ failure\ \textbf{do}\ action\ ||\ \textbf{next}\, control'$$
$$action' = \textbf{tell}(stop)$$

and A be the LTL formula:

$$A = \Box(failure \Rightarrow stop)$$

Sequences generated by A and by the processes $control'$ and $action'$ coincides and can be used to compute the intended behavior $\mathcal{I}$, the corresponding over-approximation $\mathcal{I}^\alpha = \mathcal{I}^\kappa$ and the under-approximation $\mathcal{I}^c$ as was defined previously. The intended behaviour $\mathcal{I}$ is then:

$\mathcal{I}(control') = \{s'.d.s''|d \models failure \Rightarrow d.s'' \in [\![action']\!]_{\mathcal{I}}\}$ for any s' and s''
$\mathcal{I}(action') = \{d.s|d \models stop\}$

According to Definition 14, let us compute $T^\alpha_{\{control\}}(\alpha(\mathcal{I}^c))(control)$:

$$T^\alpha_{\{control\}}(\alpha(\mathcal{I}^c))(control) = \tau(\{d_1...d_k.\textbf{true}^*|d_1 \models failure \Rightarrow d_2 \models stop \\ and\ \forall_{i>1} d_i \models failure \Rightarrow d_i \models stop\})$$

Given $s_1 = failure.stop.\texttt{true}^* \in T^\alpha_{\{control\}}(\alpha(\mathcal{I}^c))(control)$ and $s_2 = failure \wedge stop.\texttt{true}^* \in \mathcal{I}^\alpha$, notice that $s_1 \not\leq s_2$ suggesting that *control* is provably incorrect. The error is due to it delays one time unit the call to the process *action*.

Example 3. In this example we debug a program capturing the behaviour of a RCX-based robot that can go right or left. The movement of the robot is defined by the following rules: 1) If the robot goes right in the current time unit, it must turn left in the next one. 2) After two consecutive time units moving to the left, next decision must be turn right. The program proposed is depicted below:

$$\begin{aligned}
\textbf{GoR} &= \textbf{tell}(a_0 = r)||\textbf{next}\,\textbf{tell}(a_1 = r) \\
\textbf{GoL} &= \textbf{tell}(a_0 = l)||\textbf{next}\,\textbf{tell}(a_1 = l) \\
\textbf{Update} &= \textbf{when}\ a_1 = l\ \textbf{do}\ \textbf{next}\,\textbf{tell}(a_2 = l) + \\
&\quad\ \textbf{when}\ a_1 = r\ \textbf{do}\ \textbf{next}\,\textbf{tell}(a_2 = r) \\
\textbf{Zigzag} &= \textbf{when}\ a_2 \neq r\ \textbf{do}\ \textbf{GoR} + \textbf{when}\ a_2 \neq l\ \textbf{do}\ \textbf{GoL}|| \\
&\quad\ \textbf{Update}||\textbf{next}\,\textbf{Zigzag}
\end{aligned}$$

GoR and **GoL** represent the decision of turning right or left respectively. **Update** keeps track of the second-to-last action and **ZigZag** controls the behaviour of the robot. Variables a_0, a_1 and a_2 represent the current, the previous and the second-to-last decisions respectively.

Now we provide the intended specification for each process:

$$\begin{aligned}
\mathcal{I}(GoR) &= \{s|\forall_i.\ s(i) \models (a_0 = r) \text{ and } s(i+1) \models (a_1 = r)\} \\
\mathcal{I}(GoL) &= \{s|\forall_i.\ s(i) \models (a_0 = l) \text{ and } s(i+1) \models (a_1 = l)\} \\
\mathcal{I}(Update) &= \{s|\forall_i.\ s(i) \models (a_1 = l) \Rightarrow s(i+1) \models (a_2 = l)\} \cup \\
&\quad\ \{s|\forall_i.\ s(i) \models (a_1 = r) \Rightarrow s(i+1) \models (a_2 = r)\}
\end{aligned}$$

Specification of **Zigzag** can be stated by the following LTL formula:

$$A = \Box((a_0 = r) \dot{\Rightarrow} \circ(a_0 = l) \dot{\wedge} ((a_0 = l) \dot{\wedge} \circ(a_0 = l) \dot{\Rightarrow} \circ \circ (a_0 = r)))$$

The property states that always is true that 1)if the current decision is go right, the next decision must be go left and 2)if the robot goes left in two consecutive time units, the next decision must be turn right. $\mathcal{I}(Zigzag)$ can be then viewed as all the possible models of the formula A, i.e. $[\![A]\!]$:

$$\begin{aligned}\mathcal{I}(Zigzag) = &\{s | s(i) \models (a_0 = r) \Rightarrow s(i+1) \models (a_0 = l)\} \cup \\ &\{s | \, (s(i) \models (a_0 = l) \wedge s(i+1) \models (a_0 = l)) \Rightarrow s(i+2) \models (a_0 = r)\}\end{aligned}$$

Let $\mathcal{I}^\alpha$ be the abstraction $\mathcal{I}^\kappa$ representing the over-approximation of $\mathcal{I}$ and $\mathcal{I}^c$ the under-approximation as before. Let us compute a step of $T^\alpha_{\{Zigzag\}}$:

$$\begin{aligned}T^\alpha_{\{Zigzag\}}(\alpha(\mathcal{I}^c))(Zigzag) &= [\![\textbf{when } a_2 \neq r \textbf{ do GoR} + \textbf{when } a_2 \neq l \textbf{ do GoL}|| \\ &\quad \textbf{Update}]\!]_{\alpha(\mathcal{I}^c)} \\ &= (\{d_1.s | d_1 \models (a_2 \neq r) \Rightarrow d_1.s \in [\![GoR]\!]_{\alpha(\mathcal{I}^c)}\} \cup \\ &\quad \{d_1.s | d_1 \models (a_2 \neq l) \Rightarrow d_1.s \in [\![GoL]\!]_{\alpha(\mathcal{I}^c)}\}) \cap \\ &\quad \{d.s | s \in [\![Zigzag]\!]_{\alpha(\mathcal{I}^c)}\} \cap [\![\textbf{Update}]\!]_{\alpha(\mathcal{I}^c)}\end{aligned}$$

Notice that the sequence $d_1.d_2$ where $d_1 \models (a_0 = r)$ and $d_2 \models (a_0 = r)$ is a sequence that can be generated from $T^\alpha_{\{Zigzag\}}(\alpha(\mathcal{I}^c))(Zigzag)$ but $d_1.d_2 \notin \mathcal{I}^\alpha$. Using Definition 14 we can say that *Zigzag* is provably incorrect. The bug in this case, is that guard in the sub-process **when** $a_2 \neq r$ **do GoR** must be $a_1 \neq r$.

7 Conclusions

We have presented a framework for the declarative debugging of `tcc` programs w.r.t. the set of computed constraint sequences. Our method is based on a denotational semantics for `tcc` programs which models the semantics in a compositional manner. Moreover, we have shown that it is possible to use a linear temporal logic for providing a specification of correct programs, and generate automatically from it the sequences which are necessary for debugging.

We follow the idea of considering declarative specifications as programs. The intended specification can then be automatically abstracted and can be used to automatically debug the final program. As future work we are developing a prototype of our debugging system on top of an implementation of an interpreter of the `ntcc`calculus that we developed in [16].

References

1. Alpuente, M., Comini, M., Escobar, S., Falaschi, M., Lucas, S.: Abstract Diagnosis of Functional Programs. In: Leuschel, M.A. (ed.) LOPSTR 2002. LNCS, vol. 2664, pp. 1–16. Springer, Heidelberg (2003)

2. Alpuente, M., Correa, F., Falaschi, M.: A Debugging Scheme for Functional Logic Programs. In: Hanus, M. (ed.) Proc. of WFLP 2001. ENTCS, vol. 64, Elsevier, Amsterdam (2002)
3. Antoy, S., Johnson, S.: TeaBag: A Functional Logic Debugger. In: Kuchen, H. (ed.) Proc. of WFLP 2004, pp. 4–18 (2004)
4. Bueno, F., Deransart, P., Drabent, W., Ferrand, G., Hermenegildo, M., Maluszyński, J., Puebla, G.: On the role of semantic approximations in validation and diagnosis of constraint logic programs. In: Proc. of AADEBUG'97, U. of Linkoping Press, pp. 155–170 (1997)
5. Cameron, M., de la Banda, M.G., Marriott, K., Moulder, P.: Vimer: a visual debugger for mercury. In: Proc. of the 5th PPDP, pp. 56–66. ACM Press, New York (2003)
6. Comini, M., Gori, R., Levi, G.: Assertion based Inductive Verification Methods for Logic Programs. In: Seda, A.K. (ed.) Proceedings of MFCSIT'2000. ENTCS, vol. 40, Elsevier Science Publishers, Amsterdam (2001)
7. Comini, M., Levi, G., Meo, M.C., Vitiello, G.: Proving properties of Logic Programs by Abstract Diagnosis. In: Dam, M. (ed.) Analysis and Verification of Multiple-Agent Languages. LNCS, vol. 1192, pp. 22–50. Springer, Heidelberg (1997)
8. Comini, M., Levi, G., Meo, M.C., Vitiello, G.: Abstract diagnosis. Journal of Logic Programming 39(1-3), 43–93 (1999)
9. de Boer, F.S., Gabbrielli, M., Marchiori, E., Palamidessi, C.: Proving concurrent constraint programs correct. ACM TOPLAS 19(5), 685–725 (1997)
10. Ferrand, G.: Error Diagnosis in Logic Programming, an Adaptation of E.Y.Shapiro's Method. Journal of Logic Programming 4(3), 177–198 (1987)
11. Hanus, M., Josephs, B.: A debugging model for functional logic programs. In: Penjam, J., Bruynooghe, M. (eds.) PLILP 1993. LNCS, vol. 714, pp. 28–43. Springer, Heidelberg (1993)
12. Hanus, M., Koj, J.: An integrated development environment for declarative multi-paradigm programming. In: Proc. of the International Workshop on Logic Programming Environments (WLPE'01), Paphos (Cyprus), pp. 1–14 (2001)
13. Lloyd, J.W.: Declarative error diagnosis. New Generation Computing 5(2), 133–154 (1987)
14. Naish, L., Barbour, T.: Towards a portable lazy functional declarative debugger. Australian Computer Science Communications 18(1), 401–408 (1996)
15. Nielsen, M., Palamidessi, C., Valencia, F.D.: Temporal concurrent constraint programming: Denotation, logic and applications. Nordic Journal of Computing 9(1), 145–188 (2002)
16. Olarte, C., Rueda, C.: A Stochastic Concurrent Constraint Based Framework to Model and Verify Biological Systems. Clei Electronic Journal 9(2) (2005)
17. Saraswat, V., Jagadeesan, R., Gupta, V.: Foundations of timed concurrent constraint programming. In: Proc. of the Ninth Annual IEEE Symposium on Logic in Computer Science, pp. 71–80. IEEE Computer Society Press, Los Alamitos (1994)
18. Saraswat, V.A.: Semantic Foundation of Concurrent Constraint Programming. In: Proc. of 18th ACM POPL, ACM, New York (1991)
19. Shaphiro, E.Y.: Algorithmic Program Debugging. In: ACM Distinguished Dissertation, The MIT Press, Cambridge, Massachusetts (1982)
20. Tamaki, H., Sato, T.: Unfold/Fold Transformations of Logic Programs. In: Tärnlund, S. (ed.) Proc. of 2nd ICLP, pp. 127–139 (1984)

Logic Programs with Abstract Constraint Atoms: The Role of Computations

Lengning Liu[1], Enrico Pontelli[2], Tran Cao Son[2], and Mirosław Truszczyński[1]

[1] Department of Computer Science, University of Kentucky, Lexington, KY 40506, USA
{lliu1,mirek}@cs.uky.edu
[2] Department of Computer Science, New Mexico State University, Las Cruces, NM 88003, USA
{epontell,tson}@cs.nmsu.edu

Abstract. We provide new perspectives on the semantics of logic programs with *constraints*. To this end we introduce several notions of *computation* and propose to use the *results* of computations as answer sets of programs with constraints. We discuss the rationale behind different classes of computations and study the relationships among them and among the corresponding concepts of answer sets. The proposed semantics generalize the answer set semantics for programs with monotone, convex and/or arbitrary constraints described in the literature.

1 Introduction and Motivation

In this paper we study logic programs with *arbitrary* abstract constraints (which we also simply refer to as *constraints*). Programs with constraints generalize normal logic programs, programs with monotone and convex constraints [16,11], and several classes of programs with aggregates (e.g., [2,6,18]). Intuitively, a constraint A represents a *condition* on models of the program containing A. The definition of A includes an *explicit* description of conditions interpretations have to meet in order to satisfy it. The syntax of such programs and one possible semantics have been proposed in [15]. Another semantics has been proposed in [21]. Following the approach proposed in [11], and exploiting analogies to the case of normal logic programs, we introduce several other semantics based on *computations* for programs with constraints. We argue that the *results* of (some types of) computations adequately generalize *answer sets* of programs with constraints.

The notion of an *answer set* of a logic program [8] is the foundation for answer-set programming (ASP) [14,17]. Intuitively, an answer set represents beliefs of an agent, given a logic program encoding its knowledge base. Researchers developed several characterizations of answer sets, providing a reasoner with alternative ways to determine them. The original definition of answer sets [8] naturally leads to a "guess-and-check" approach. We first guess a candidate for an answer set, and then we validate the guess. The validation consists of recomputing the guess starting with the empty set and iterating the *one-step provability operator* [22] for the *Gelfond-Lifschitz* program reduct [8]. Alternatively, we can compute an answer set starting with the empty set. At each step, we include in the set under construction the heads of *some* of the rules applicable at this step; typically, we include all the rules selected during the previous

V. Dahl and I. Niemelä (Eds.): ICLP 2007, LNCS 4670, pp. 286–301, 2007.

steps *plus* some additional ones. Once the process stabilizes, we need to check that the final result does not block any rules chosen to fire earlier [12,13]. In this second approach, we replace the initial non-deterministic step of guessing the answer set with non-deterministic choices of rules to fire at each step of the construction.

Example 1. Let us consider the program P_1 consisting of the following rules:

$$a \leftarrow \mathbf{not}\ b \qquad c \leftarrow a \qquad b \leftarrow \mathbf{not}\ a \qquad d \leftarrow b.$$

This program has two answer sets: $\{a, c\}$ and $\{b, d\}$. In the "guess-and-check" approach, we might guess $\{a, c\}$ as a candidate answer set. To verify the guess, we compute the Gelfond-Lifschitz reduct, consisting of the rules $a \leftarrow$, $c \leftarrow a$ and $d \leftarrow b$. Next, we iterate the one-step provability operator for the reduct to compute its least Herbrand model—which corresponds to $\{a, c\}$. Since it coincides with the initial guess, the guess is validated as an answer set. In this way, we can also validate the guess $\{b, d\}$. However, the validation of $\{a\}$ fails—i.e., $\{a\}$ is not an answer set.

The other approach starts with the empty interpretation, $\emptyset$, which makes two rules applicable: $a \leftarrow \mathbf{not}\ b$ and $b \leftarrow \mathbf{not}\ a$. The algorithm needs to select some of them to "fire", say, it selects $a \leftarrow \mathbf{not}\ b$. The choice results in the new interpretation, $\{a\}$. Two rules are applicable now: $a \leftarrow \mathbf{not}\ b$ and $c \leftarrow a$. The algorithm selects both rules for firing. The interpretation that results is $\{a, c\}$. The same two rules that were applicable in the previous step are applicable now. Thus, there is no possibility to add new elements to the current set. The computation stabilizes at $\{a, c\}$. Since $\{a, c\}$ does not block any of the rules fired in the process, $\{a, c\}$ is an answer set. □

We note that the first approach starts with a tentative answer set of the program, while the second starts with the *empty* interpretation. In the first approach, we guess the entire answer set at once and, from that point on, proceed in a deterministic fashion. In the second approach we construct an answer set *incrementally* making non-deterministic choices along the way. Thus, each approach involves non-determinism. However, in the second approach, the role of non-determinism is generally reduced. In this paper we cast these two approaches in terms of abstract principles related to a notion of *computation*. We then lift these principles to the case of programs with abstract constraints and propose several new semantics for such programs.

The interest in ASP has been fueled by the development of software to compute answer-sets of logic programs, most notably SMODELS and DLV, which allow programmers to tackle complex real-world problems (e.g., [1,10,5]). To facilitate declarative solutions of problems in knowledge representation and reasoning, researchers proposed extensions of the logic programming language, which support constraints and aggregates [17,2,3,4,6,9,18,19]. This development stimulated interest in logic programming formalisms based on *abstract constraint atoms* [15,16]. The concept of an abstract constraint atom is general and encompasses several language-level extensions including aggregates. The introduction of constraints brought forth the question of how to extend the semantics of answer sets to the case of programs with constraints. Researchers proposed several possible approaches [6,3,19,21,4,20]. They *all* agree on specific classes of programs with constraints including normal logic programs (viewed as programs with constraints), programs with *monotone* constraints [16], and programs with *convex* constraints [11]. However, they differ on programs with arbitrary constraints.

What makes the task of defining answer sets for programs with arbitrary constraints difficult and interesting is the *nonmonotonic* behavior of such constraints. For instance, let us consider the constraint $(\{p(1), p(-1)\}, \{\{p(1)\}\})$ (we introduce this notation in Section 3), which can be seen as an encoding of the aggregate $\text{SUM}(\{X \mid p(X)\}) \geq 1$.[1] This aggregate atom is true in the interpretation $\{p(1)\}$ but false in $\{p(1), p(-1)\}$.

The contribution of this paper is the development of a general framework for defining and studying answer sets for logic programs with arbitrary constraints. Our proposals rely on an abstract notion of incremental *computation* and can be traced back to one of the two basic approaches to computing answer sets of normal logic programs that we mentioned above. This notion generalizes an approach developed in [16,11] for the case of programs with monotone and convex constraints. In the paper, we study properties of the notions of answer set we introduce, and relate them to the earlier proposals.

2 Computations in Normal Logic Programs—Principles

We start by motivating the notion of a *computation*, which is central to our paper. To this end, we look at the case of normal logic programs and build on our discussion in Example 1. In particular, we show how to use computations to characterize answer-sets. We represent propositional interpretations as sets of atoms. A program rule whose body is satisfied by an interpretation M is called *M-applicable*. We write $P(M)$ to denote the set of all M-applicable rules in a program P. The *one-step provability operator* assigns to an interpretation M the set of the heads of all rules in $P(M)$. We denote this operator by T_P. Fixpoints of T_P are *supported* models of P. An interpretation M is a *stable model* or, as we will say here, an *answer set* of P if M is the least model of the Gelfond-Lifschitz reduct P^M.

We define computations as sequences $\langle X_i \rangle_{i=0}^{\infty}$ of sets of atoms (propositional interpretations), where X_i represents the status of the computation at step i. In particular, we *require* that $X_0 = \emptyset$. The basic intuition is that a computation, at each step $i \geq 1$, revises its previous status X_{i-1}. We base the revision on a non-deterministic operator, $Concl_P(X)$, that provides the set of revisions of X that can be justified according to a logic program P and a set of atoms X. Formally, a set of atoms Y is *grounded* in a set of atoms X and a program P if

$$Y \subseteq \{a \mid (a \leftarrow body) \in P \text{ and } X \models body\}.$$

We write $Concl_P(X)$ to denote the set of all possible sets Y grounded in X and P. We require that computations satisfy the *principle of revision*:

> ($\mathbf{R}$) *Revision principle*: each successive element in a computation must be grounded in the preceding one and the program, i.e., $X_i \in Concl_P(X_{i-1})$, for every i, $1 \leq i$.

A computation of an answer set of a program, using a method as described in Example 1, produces a monotonically increasing sequence of sets, each being a part of the answer set being built. Thus, at each step not only new atoms are computed, but also all atoms established earlier are recomputed. This suggests the principle of *persistence of beliefs*:

[1] We assume that $1, -1$ are the two available constants.

($\mathbf{P}$) *Persistence of beliefs*: each next element in the computation must contain the previous one (once we "revise an atom in", we keep it), i.e., $X_{i-1} \subseteq X_i$, for every i, $1 \leq i$.

For a sequence $\langle X_i \rangle_{i=0}^{\infty}$ satisfying the principle ($\mathbf{P}$), we define X_∞ to be the *result* of $\langle X_i \rangle_{i=0}^{\infty}$ by setting $X_\infty = \bigcup_{i=0}^{\infty} X_i$. The result of the computation should be an interpretation that could not be revised further. This suggests one more basic principle for computations, the principle of *convergence*:

($\mathbf{C}$) *Convergence*: the computation process continues until it becomes stable (no additional revisions can be made), i.e., $X_\infty = T_P(X_\infty)$, where T_P is the one-step provability operator. In other words, X_∞ is a supported model of P.

Definition 1. *Let P be a normal logic program. A sequence $\langle X_i \rangle_{i=0}^{\infty}$ is a* computation *for P if $\langle X_i \rangle_{i=0}^{\infty}$ satisfies the principles ($\mathbf{R}$), ($\mathbf{P}$) and ($\mathbf{C}$), and $X_0 = \emptyset$.*

Computations are indeed relevant to the task of describing answer sets of normal logic programs. We have the following result.

Proposition 1. *Let P be a normal logic program. If a set of atoms X is an answer set of P then there exists a computation for P, $\langle X_i \rangle_{i=0}^{\infty}$, such that $X = X_\infty$.*

Proof (Sketch). The sequence $\langle X \rangle_{i=0}^{\infty}$ can be obtained from the iterations of the one-step provability operator of the Gelfond-Lifschitz reduct of the program P. □

Proposition 1 implies that the principles ($\mathbf{R}$), ($\mathbf{P}$) and ($\mathbf{C}$) give a notion of computation broad enough to derive any answer set. Is this concept of computation what is needed to precisely characterize answer sets? In other words, does *every* sequence of sets of atoms starting with the $\emptyset$ and satisfying the principles ($\mathbf{R}$), ($\mathbf{P}$) and ($\mathbf{C}$) result in an answer set? This is indeed the case for *positive* normal logic programs, more commonly referred to as Horn programs.

Proposition 2. *Let P be a positive logic program. The result of every computation is equal to the least model of P, that is, to the unique answer set of P.*

However, in the case of arbitrary normal programs, there are computations that do not result in answer sets.

Example 2. Consider the program P_2 containing the two rules $a \leftarrow \mathbf{not}\ a$ and $a \leftarrow a$. The sequence $X_0 = \emptyset$, $X_1 = \{a\}$, $X_2 = \{a\}$, ... satisfies ($\mathbf{R}$), ($\mathbf{P}$) and ($\mathbf{C}$) and so, it is a computation for P. However, $X = \bigcup_{i=0}^{\infty} X_i = \{a\}$ is not an answer set of P_2. □

It follows that the notion of a computation defined by the principles ($\mathbf{R}$), ($\mathbf{P}$) and ($\mathbf{C}$) is too broad to describe exactly the notion of an answer set. Let us come back to Example 2. There, $a \in X_1$ because the body of the first rule is satisfied by the set $\emptyset$. However, the body of the first rule is *not* satisfied in every set X_i for $i \geq 1$. Nevertheless, $a \in X_i$, for $i \geq 2$, since the body of the *second* rule is satisfied by X_{i-1}. Thus, the reason for the presence of a in the next revision changes between the first and second step. This is why that sequence does not result in an answer set, even though it

satisfies the principle $(\mathbf{P})$, which guarantees that atoms once revised in will remain in throughout the rest of the computation.

These considerations suggest that useful classes of computations can be obtained by requiring that not only atoms but also *reasons* for including atoms persist. Intuitively, we would like to associate with each atom included in X_i a rule that supports the inclusion, and this rule should remain applicable from that point on. More formally, we state this principle as follows:

> (**Pr**) *Persistence of Reasons*: for every $a \in X_\infty$ there is a rule $r_a \in P$ (called the *reason* for a) whose head is a and whose body holds in every X_i, $i \geq i_a - 1$, where i_a is the least integer such that $a \in X_{i_a}$.

This principle is exactly what is needed to characterize answer sets of normal logic programs.

Definition 2. *Let P be a normal logic program. A computation $\langle X \rangle_{i=0}^{\infty}$ for P is* persistent *if it satisfies the principle* $(\mathbf{Pr})$.

Proposition 3. *Let P be a normal logic program. A set X is an answer set of P if and only if there is a persistent computation for P such that its result is X.*

We now observe that, in general, the operator $Concl_P$ offers several choices for revising a current interpretation X_{i-1} into X_i during a computation. The question is whether this freedom is necessary or whether we can restrict the principle $(\mathbf{R})$ and still characterize answer sets of normal logic programs.

Example 3. Let P_3 be the normal logic program:

$$a \leftarrow \mathbf{not}\ b \qquad c \leftarrow \mathbf{not}\ b \qquad e \leftarrow a, c \qquad f \leftarrow a, \mathbf{not}\ c.$$

This program has only one answer set $M = \{a, c, e\}$, which corresponds to the computation $\emptyset$, $\{a, c\}$, $\{a, c, e\}$. In this computation, at each step $i = 1, 2$, we take as X_i a greatest element of $Concl_{P_3}(X_{i-1})$, which exists and is given by $T_{P_3}(X_{i-1})$. Thus, the next element of the computation is the result of firing *all applicable* rules.

On the other hand, selecting an element in $Concl_{P_3}(X)$ other than $T_{P_3}(X)$ can result in sequences that cannot be extended to a computation. For example, the sequence $\emptyset$, $\{a\}$, $\{a, f\}$ represents a potential computation since it satisfies the (**R**) and (**P**) principles. Yet, no possible extension of this sequence satisfies the (**C**) principle. □

This example indicates that interesting classes of computations can be obtained by restricting the operator $Concl_P$. Since for every X we have $T_P(X) \in Concl_P(X)$, we could restrict the choice for possible revisions of X based on P to $T_P(X)$. The class of computations obtained under this restriction is a *proper* subset of the class of computations. For instance, the program P_1 from Example 1 does not admit computations that revise X_{i-1} into $X_i = T_P(X_{i-1})$. Thus, the class of such computations is not adequate for the task of characterizing answer sets of normal logic program. We note, though, that they do characterize answer sets for some special classes of logic programs, for instance, for stratified logic programs.

To obtain a general characterization of answer sets by restricting the choices offered by $Concl_P(X)$, we need to modify the operator $T_P(X)$. The first approach to computing answer sets, discussed in the introduction provides a clue: we need to change the notion of satisfiability used in the definition of $T_P(X)$.

Let $M \subseteq At$. We define the satisfiability relation $\models_M$, between sets of atoms and conjunctions of literals, as follows: we say that $S \models_M F$ holds (where $S \subseteq At$ and F is a conjunction of literals) if $S \models F$ and $M \models F$. That is, the satisfaction is based not only on S but also on M (the "context"). We now define the (context-based) one-step provability operator T_P^M as follows:

$$T_P^M(X) = \{a \mid a \leftarrow \mathit{body} \in P,\ X \models_M \mathit{body}\}.$$

We note that $T_P^M(X) \in Concl_P(X)$. Thus, we obtain the following result.

Proposition 4. *Let P be a normal logic program. A sequence $\langle X_i \rangle_{i=0}^{\infty}$ of sets of atoms that satisfies properties* (**P**) *and* (**C**)*, as well as the property $X_i = T_P^M(X_{i-1})$, for $i = 1, 2, \ldots$, is a computation for P.*

If a computation is determined by the satisfiability relation $\models_M$ in the way described in Proposition 4, we call it an M-computation. Not all M-computations define answer sets. Let us consider the program P_2 from Example 2. The sequence $X_0 = \emptyset$, $X_i = \{a\}$, $i = 1, 2, \ldots$, is a $\emptyset$-computation for P_2. But the result of this computation ($\{a\}$) is not an answer set for P_2.

The problem is that M-computations may not to be persistent. In fact, the $\emptyset$-computation described above is not. It turns out that if we impose the condition of persistence on M-computations, their results are answer sets.

Proposition 5. *Let P be a normal logic program and $M \subseteq At$. If an M-computation is persistent then its result is an answer set for P.*

It is also the case that every answer set is the result of some persistent M-computation.

Proposition 6. *Let P be a normal logic program. A set M of atoms is an answer set of P if and only if there exists a persistent N-computation whose result is M.*

A even stronger result can be proved—in which answer sets are characterized by a proper subclass of persistent M-computations. We call an M-computation *self-justified* if its result is M.

Proposition 7. *Let P be a normal logic program. A set $M \subseteq At$ is an answer set of P if and only if P has a self-justifying M-computation (whose result, by the definition, is M).*

One can check that self-justified M-computations are persistent. Moreover, in general, the class of self-justified M-computations is a proper subclass of M-computations. Thus, we can summarize our discussion in this section as follows. Answer sets of normal logic programs can be characterized as the results of persistent computations, persistent M-computations, and self-justified M-computations, with each subsequent class being a proper subclass of the preceding one.

Our goal, in this section, was to recast answer sets of normal logic programs in terms of computations. More specifically, taking two ways of computing answer sets as the departure point, we introduced three characterizations of answer sets in terms of persistent computations. In Sections 4 and 5, we will show how to generalize the classes of computations discussed here to the case of programs with constraints. In this way, we will arrive at concepts of answer sets for programs with constraints that generalize the concept of answer sets for normal logic programs.

3 Programs with Abstract Constraints—Basic Definitions

We recall here some basic definitions concerning programs with constraints [15,16,11]. We fix an infinite set At of propositional variables. A *constraint* is an expression $A = (X, C)$, where $X \subseteq At$ is a *finite* set, and $C \subseteq \mathcal{P}(X)$ ($\mathcal{P}(X)$ denotes the powerset of X). The set X is called the *domain* of $A = (X, C)$, denoted by A_{dom}. Elements of C are called *satisfiers* of A, denoted by A_{sat}. Intuitively, the sets in A_{sat} are precisely those subsets of A_{dom} that *satisfy* the constraint. A constraint A is said to be *monotone* if for every $X \subseteq Y \subseteq A_{dom}$, $X \in A_{sat}$ implies that $Y \in A_{sat}$. A constraint A is said to be *convex* if for all $X \subseteq Y \subseteq Z \subseteq A_{dom}$ such that $X, Z \in A_{sat}$, $Y \in A_{sat}$, too.

Constraints are building blocks of rules and programs. A *rule* is an expression

$$A \leftarrow A_1, \ldots, A_k \tag{1}$$

where A, $A_1, \ldots, A_k$ are constraints. A *constraint program* (or a *program*) is a collection of rules. A program is *monotone* (*convex*) if every constraint occurring in it is monotone (convex).

Given a rule r of the form (1), the constraint A is the *head* of r and the set $\{A_1, \ldots, A_k\}$ of constraints is the *body* of r (sometimes we view the body of a rule as the *conjunction* of its constraints). We denote the head and the body of r by $hd(r)$ and $bd(r)$, respectively. We define the *headset* of r, written $hset(r)$, as the domain of the head of r. That is, $hset(r) = hd(r)_{dom}$.

We view subsets of At as interpretations. We say that M *satisfies* a constraint A, denoted by $M \models A$, if $M \cap A_{dom} \in A_{sat}$. The satisfiability relation extends in the standard way to conjunctions of constraints, rules and programs.

Let $M \subseteq At$ be an interpretation. A rule is *M-applicable* if M satisfies every constraint in $bd(r)$. We denote with $P(M)$ the set of all M-applicable rules in P. Let P be a program. A model M of P is *supported* if $M \subseteq hset(P(M))$.

Let P be a program and M a set of atoms. A set M' is *non-deterministically one-step provable* from M by means of P, if $M' \subseteq hset(P(M))$ and $M' \models hd(r)$ for every rule $r \in P(M)$. The *nondeterministic one-step provability operator* T_P^{nd} for a program P is an operator on $\mathcal{P}(At)$ such that for every $M \subseteq At$, $T_P^{nd}(M)$ consists of all sets that are non-deterministically one-step provable from M by means of P.

For an arbitrary atom $a \in At$, the constraint $(\{a\}, \{\{a\}\})$ is called an *elementary constraint*. Since $(\{a\}, \{\{a\}\})$ has the same models as a, $(\{a\}, \{\{a\}\})$ is often identified with (and denoted by) a. For the same reason, $(\{a\}, \{\emptyset\})$ can be identified with $\mathbf{not}\ a$. Given a normal logic program P, by $C(P)$ we denote the program with constraints obtained from P by replacing every positive atom a in P with the constraint $(\{a\}, \{\{a\}\})$, and replacing every literal $\mathbf{not}\ a$ in P with the constraint $(\{a\}, \{\emptyset\})$.

We note that $C(P)$ is a convex program [11]. One can show that supported models of P coincide with supported models of $C(P)$, and answer sets of P coincide with answer sets of $C(P)$ (according to the definition from [11]). In other words, programs with constraints are sufficient to express normal logic programs. Therefore, in this paper we focus on programs with abstract constraints only.

4 Computations for Programs with Constraints

Our goal is to extend the concept of a computation to programs with constraints. Once we have such a concept in hand, we will use it to define answer sets for programs with constraints. To this end, we will build on the two characterizations of answer sets for the case of normal logic programs, which we developed in Section 2.

In order to define computations of programs with constraints, we consider the principles identified in Sect. 2. The key step is to generalize the revision principle. For normal programs, it was based on sets of atoms grounded in a set of atoms X (current interpretation) and P. We will now extend this concept to programs with constraints.

Definition 3. *Let P be a program with constraints and let $X \subseteq At$ be a set of propositional atoms. A set Y is grounded in X and P if for some program $Q \subseteq P(X)$, $Y \in T_Q^{nd}(X)$. We denote by $Concl_P(X)$ the set of all sets Y grounded in X and P.*

The intuition is the same as before: a set Y is grounded in X and P if it can be justified by means of some rules in P on the basis of X. It follows directly from the definition that if $Q \subseteq P$ then $T_Q^{nd}(X) \subseteq Concl_P(X)$, which generalizes a similar property in the case P is a normal logic program: if $Q \subseteq P$ then $T_Q(X) \in Concl_P(X)$.

With this definition of $Concl_P(X)$, the principle $(\mathbf{R})$ lifts without any changes. The same is true for the principle $(\mathbf{P})$ (which is independent of the choice of the class of programs). The principle $(\mathbf{C})$ is also easy to generalize thanks to its alternative statement in terms of models:

$(\mathbf{C})$ *Convergence*: X_∞ is a supported model of P, i.e., $X_\infty \in T_P^{nd}(X_\infty)$.

Finally, the principle $(\mathbf{Pr})$ generalizes, as well. At a step i of a computation that satisfies $(\mathbf{R})$, we select as X_i an element of $Concl_P(X_{i-1})$. By the definition of $Concl_P(X_{i-1})$, there is a program $P_{i-1} \subseteq P(X_{i-1})$ such that $X_i \in T_{P_{i-1}}^{nd}(X_{i-1})$. Each such program can be viewed as a *reason* for X_i. We can now state the generalized principle $(\mathbf{Pr})$ as follows:

$(\mathbf{Pr})$ *Persistence of Reasons*: There is a sequence of programs $\langle P_i \rangle_{i=0}^{\infty}$ such that for every i, $0 \leq i$, $P_i \subseteq P_{i+1}$, $P_i \subseteq P(X_i)$, $X_{i+1} \in T_{P_i}^{nd}(X_i)$.

Having extended the principles $(\mathbf{R})$, $(\mathbf{P})$, $(\mathbf{C})$ and $(\mathbf{Pr})$ to the class of programs with constraints, we define computations literally extending the earlier definitions.

Definition 4. *Let P be a program with abstract constraints. A sequence $\langle X_i \rangle_{i=0}^{\infty}$ is a* computation *for P if $X_0 = \emptyset$ and the sequence satisfies the principles $(\mathbf{R})$, $(\mathbf{P})$ and $(\mathbf{C})$. A computation is* persistent *if it also satisfies the principle $(\mathbf{Pr})$.*

As before, we have the following containment property.

Proposition 8. *Let P be a program with constraints. The class of persistent computations is a* proper *subset of the class of computations.*

Proof (Sketch). The containment is evident. To show that it is proper, we consider the program $C(P_2)$, where P_2 is the normal logic program from Example 2. The sequence $\emptyset, \{a\}, \ldots$ is a computation but not a persistent computation for $C(P_2)$.

It is also the case, that computations for programs with constraints generalize computations for normal logic programs.

Proposition 9. *Let P be a normal logic program. The class of computations (respectively, persistent computations) of P, according to the definitions in Section 2, coincides with the class of computations (respectively, persistent computations) of the program $C(P)$, according to definitions in Section 3.*

We conclude this section by proposing the first definition of the concept of *answer set* for programs with constraints. To this end, we generalize the characterization of answer sets of normal logic programs in terms of persistent computations discussed in Section 2.

Definition 5. *Let P be a program with constraints. A set X is an* answer set *of P if there is a persistent computation for P, whose result is X.*

Since persistent computations satisfy the principle $(\mathbf{C})$, answer sets of a program P with constraints are supported models of P, generalizing a property of normal logic programs. As a matter of fact, the results of arbitrary computations are supported models (because of $(\mathbf{C})$). However, there are programs such that some of their supported models cannot be reached by a computation. For instance, $\{a\}$ is a supported model of the program $C(P)$, where $P = \{a \leftarrow a\}$, but there is no computation for $C(P)$ with the result $\{a\}$.

Proposition 9 implies that our definition of answer sets for programs with constraints generalizes the definition of answer sets for normal logic programs.

Corollary 1. *Let P be a normal logic program. A set $X \subseteq At$ is an answer set of P if and only if X is an answer set of $C(P)$.*

It is also the case that this concept of answer sets extends that introduced in [16,11] for monotone and convex programs.

Proposition 10. *Let P be a monotone or convex program. Then, a set of atoms $X \subseteq At$ is an answer set of P according to the definition in [16,11] if and only if X is the answer set of P according to Definition 5.*

5 Computations and Quasi-satisfiability Relations

The notion of a computation discussed so far makes use of the non-deterministic operator $Concl_P$ to revise the interpretations occurring along a computation. As we mentioned earlier, the use of $Concl_P$ provides a wide range of choices for revising a state of a computation, essentially considering all the subsets of applicable rules.

We will now study computations, as well as related concepts resulting by relaxing some of the postulates for computations, which can be obtained by narrowing down the set of choices given by $Concl_P(X)$ as possible revisions of X. In the case of normal logic programs, we accomplished this goal by means of an operator T_P^M, based on the satisfiability relation $\models_M$. We will now generalize that idea to the case of programs with constraints.

Definition 6. *A sequence $C = \langle X_i \rangle_{i=0}^{\infty}$ is a* weak computation *for a program with constraints, P, if $X_0 = \emptyset$ and if C satisfies the properties* $(\mathbf{P})$ *and* $(\mathbf{C})$.

Thus, weak computations are sequences that do not rely on a program P when moving from step i to step $i+1$. We will now define a broad class of weak computations that, at least to some degree, restore the role of P as a revision mechanism.

Let $\triangleright$ be a relation between sets of atoms (interpretations) and abstract constraints. We extend the relation $\triangleright$ to the case of conjunctions (sets) of constraints as follows: $X \triangleright A_1, \ldots, A_k$ if $X \triangleright A_i$, for every i, $1 \leq i \leq k$. This relation is intended to represent some concept of satisfiability of constraints and their conjunctions. We will call such relations *quasi-satisfiability* relations. They will later allow us to generalize the relation $\models_M$.

For a quasi-satisfiability relation $\triangleright$, we define

$$P^{\triangleright}(X) = \{r \in P \mid X \triangleright bd(r)\}.$$

In other words, $P^{\triangleright}(X)$ is the set of all rules in P that are applicable with respect to X under the relation $\triangleright$. Next, we define $T_P^{nd;\triangleright}(X)$ to consist of all sets $Y \subseteq hset(P^{\triangleright}(X))$ such that $Y \models hd(r)$, for every $r \in P^{\triangleright}(X)$. In other words $T_P^{nd;\triangleright}$ works similarly to T_P^{nd}, except that rules in $P^{\triangleright}(X)$ are "fired" rather than those in $P(X)$.

Definition 7. *Let $\triangleright$ be a quasi-satisfiability relation. A weak computation $C = \langle X_i \rangle_{i=0}^{\infty}$ is a* $\triangleright$-weak computation *for P if $X_i \in T_P^{nd;\triangleright}(X_{i-1})$, for $i \geq 1$.*

Since we do not impose any particular properties on $\triangleright$, it is not guaranteed that $T_P^{nd;\triangleright}(X) \subseteq Concl_P(X)$. Thus, $\triangleright$-weak computations are not guaranteed to be computations.

We say that a quasi-satisfiability relation $\triangleright$ is a *sub-satisfiability* relation if for every $X \subseteq At$ and every abstract constraint A, $X \triangleright A$ implies $X \models A$.

We note that relations $\models_M$ considered in Section 2 are sub-satisfiability relations (with respect to the standard satisfiability relation $\models$).

Proposition 11. *Let P be a program with constraints. If $\triangleright$ is a sub-satisfiability relation then for every $X \subseteq At$, $T_P^{nd;\triangleright}(X) \subseteq Concl_P(X)$ and every $\triangleright$-weak computation is a computation.*

From now on, if $\triangleright$ is a sub-satisfiability relation, we will write $\triangleright$-*computation* instead of $\triangleright$-weak computation. We will now define another class of answer sets for programs with constraints.

Definition 8. *Let P be a program with constraints and $\triangleright$ a sub-satisfiability relation. Then, M is a* $\triangleright$-answer set *of P if M is the result of a persistent $\triangleright$-computation.*

Since $\triangleright$-computations are computations, we have the following direct consequence of the appropriate definitions.

Proposition 12. *Let P be a program with constraints and $\triangleright$ a sub-satisfiability relation. Then every $\triangleright$-answer set for P is an answer set for P.*

A natural questions arises whether every answer set of a program with constraints is a $\triangleright$-answer set of that program for some sub-satisfiability relation $\triangleright$. Unlike in the case of normal logic programs, it is not the case.

Example 4. Let P consist of three rules:

$$(\{a\},\{\{a\}\}). \quad (\{b\},\{\{b\}\}). \quad (\{c\},\{\{c\}\}) \leftarrow (\{a,b,c\},\{\{a\},\{a,c\},\{a,b,c\}\}).$$

We first note that $X_0 = \emptyset$, $X_1 = \{a\}$, $X_2 = \{a,c\}$, $X_i = \{a,b,c\}$, $i \geq 3$, is a persistent computation for P. Thus, $\{a,b,c\}$ is an answer set or P. However, there is no sub-satisfiability relation $\triangleright$ such that $\{a,b,c\}$ is a $\triangleright$-answer set for P. Indeed, for each such relation $\triangleright$ we have $T_P^{nd;\triangleright}(\emptyset) = \{a,b\}$, and it is impossible to derive c, as the third rule is not applicable with respect to $\{a,b\}$ (and so, also not a member of $P^{\triangleright}(\{a,b\})$). □

Thus, given a program P, the class of $\triangleright$-answer sets is a proper subset of the class of answer sets of P.

The class of $\triangleright$-answer sets forms a generalization of answer sets of normal logic programs given by Proposition 5. We will now propose a way to generalize answer sets of normal logic programs to programs with constraints based on Proposition 7. In our considerations we extend the approach proposed and studied in [21]. Our method requires a fixed mapping f that assigns to each weak computation C a quasi-satisfiability relation $\triangleright_C^f$. For some mappings f it yields models that are not "grounded enough" to be called answer sets. For some other mappings f, however, it does result in classes of models that arguably generalize answer sets of normal logic programs.

Definition 9. *Let P be a program with constraints and f a mapping assigning to each weak computation C a quasi-satisfiability relation $\triangleright_C^f$. A weak computation C is (P,f)* self-justified *if C is a $\triangleright_C^f$-weak computation for P. A set of atoms M is an* f-model of P *if M is the result of a (P,f) self-justified weak computation.*

The definition of an f-model is sound. Since weak computations satisfy the property $(\mathbf{C})$, their results are indeed models of P, in fact, even supported models.

Several interesting classes of models of programs with constraints can be described in terms of f-models by specializing the mapping f.

Supported models. Let C be a weak computation. We define the relation $\triangleright_C^{supp}$ as follows: given a set of atoms X and a constraint A, $X \triangleright_C^{supp} A$ if $X_\infty \models A$. One can show that for every supported model M of P, the sequence $C = \langle \emptyset, M, M, \ldots \rangle$ is a weak computation self-justified with respect to P and $\triangleright_C^{supp}$. Thus, every supported model of P is a $supp$-model of P. As we observed earlier, all f-models are supported models. It follows that supported models of P are precisely $supp$-models of P.

***Mr*-models.** Let C be a weak computation. We define the relation $\triangleright_C^{mr}$ as follows: given a set of atoms X and a constraint A, $X \triangleright_C^{mr} A$ if there is $Y \subseteq X$ such that $Y \models A$ and $X_\infty \models A$. One can show that mr-models of P are precisely the answer sets of P as defined by [15].

The discussion of $supp$-models and mr-models was meant to show the flexibility of our approach. It took us away, however, from the main theme of the paper — generalizations of the concept of an answer set. Indeed, neither $\triangleright_C^{supp}$-weak computations nor $\triangleright_C^{mr}$-weak computations are computations (they do not satisfy the revision principle). Therefore, we do not view their results as "generalized" answer sets but only as some special classes of models.

To specialize the general approach of self-justified weak computations so that it yields extensions of the concept of an answer set, we need to look for mappings f that ensure that self-justified weak computations are computations (satisfy the revision principle) and are persistent.

We already saw that requiring that $\triangleright_C^f$ be a sub-satisfiability relation guarantees that $\triangleright_C^f$-weak computations are indeed computations (referred to, we recall, as $\triangleright_C^f$-computations). We will now seek conditions guaranteeing the persistence of $\triangleright_C^f$-computations.

Under the assumption that $\triangleright$ is a sub-satisfiability relation, $P^{\triangleright}(X) \subseteq P(X)$. This property and the appropriate definitions imply that

$$T_P^{nd;\triangleright}(X) = T_{P^{\triangleright}(X)}^{nd;\triangleright}(X) = T_{P^{\triangleright}(X)}^{nd}(X).$$

Consequently, we can show the following result.

Proposition 13. *Let $\triangleright$ be a sub-satisfiability relation and let $C = \langle X_i \rangle_{i=0}^{\infty}$ be a $\triangleright$-computation. If for every constraint A and every $i = 0, 1, \ldots$, $X_i \triangleright A$ implies that $X_{i+1} \triangleright A$, then C is persistent.*

We will now define the mapping s, which assigns to every weak computation C a relation $\triangleright_C^s$. Namely, for a set of atoms X and a constraint A we define $X \triangleright_C^s A$ if there is i such that $X = X_i$ and $X_j \models A$, for every $j \geq i$. It is clear that $\triangleright_C^s$ is a sub-satisfiability relation. Thus, $\triangleright_C^c$-weak computations are computations ($\triangleright_C^s$-computations, to be precise). Moreover, it follows from Proposition 13 that $\triangleright_C^s$-computations are persistent.

Definition 10. *Let P be a program with constraints. A set of atoms M is a* strong answer set *of P (or, an* s-answer *set for P) if it is an s-model for P, that is, if M is the result of a (P, s) self-justified computation.*

We will now summarize our discussion so far. Taking characterizations of answer sets of normal logic programs as the starting point, we proposed three notions of answer sets for programs with constraints. For normal logic programs, for programs with monotone constraints, and for programs with convex constraints all three concepts coincide with the standard notion of an answer set. For general programs with constraints, these three concepts are different. The following results summarizes the relationships between the three classes of answer sets we introduced so far.

Proposition 14. *Let P be a program with constraints. The class of strong answer sets for P is a proper subset of the class of $\triangleright$-answer-sets for P which, in turn, is a proper subset of the class of answer sets for P.*

Example 5. Consider the program P (remember that a is shorthand for $(\{a\}, \{\{a\}\})$):

$$a \leftarrow (\{a,b,c,d\}, \mathcal{P}(\{a,b,c,d\}))$$
$$b \leftarrow (\{a,b\}, \{\{a\}, \{a,b\}\})$$
$$c \leftarrow (\{a,b,c,d\}, \{\emptyset, \{a\}, \{a,b\}, \{a,b,c,d\}\})$$

where $\mathcal{P}(X)$ is the powerset of X. Let us define $X \triangleright A$ if for every Y, $X \cap A_{dom} \subseteq Y \subseteq A_{dom}$, $Y \in A_{sat}$. Clearly, $\triangleright$ is a sub-satisfiability relation. Furthermore $\emptyset$, $\{a\}$, $\{a,b\}, \{a,b\} \ldots$ is a $\triangleright$-weak computation, say C. On the other hand, we have that $\emptyset \ \triangleright^s_C (\{a,b,c,d\}, \mathcal{P}(\{a,b,c,d\}))$ and $\emptyset \triangleright^s_C (\{a,b,c,d\}, \{\emptyset, \{a\}, \{a,b\}, \{a,b,c,d\}\})$. Thus, any (P,s) self-justified computation will need to have $\{a,c\}$ as its second element and will be unable to reach $\{a,b\}$. This shows that $\{a,b\}$ is not a strong answer set of P. We note that it follows from Example 4 that the second inclusion is proper. □

6 Yet Another Class of Answer Sets

Given a computation C, we defined the relation $\triangleright^s_C$ so that it is the weakest sub-satisfiability relation satisfying the assumptions of Proposition 13. However, in general there may be other ways to define a sub-satisfiability relation $\triangleright_C$ with respect to a given computation C. One such definition was proposed in [21]. Namely, given a weak computation $C = \langle X_i \rangle_{i=0}^{\infty}$, we define $\triangleright^{spt}_C$ as follows: $X \triangleright^{spt}_C A$ if $X \models A$ and for each set Y such that $X \cap A_{dom} \subseteq Y \subseteq X_\infty \cap A_{dom}$ we have that $Y \models A$ (or equivalently, $Y \in A_{sat}$). It is easy to see that $\triangleright^{spt}_C$ is a sub-satisfiability relation. Thus, it defines computations. Secondly, $\triangleright^{spt}_C$-computations are persistent as the relation $\triangleright^{spt}_C$ satisfies the assumptions of Proposition 13. Thus, the mapping spt gives rise to yet another class of answer sets — spt-answer sets. One can show that spt-answer sets capture precisely the semantics for aggregates proposed in [18,19].

We have the following result relating spt-answer sets to other classes of answer sets considered in the previous section.

Proposition 15. *Let P be a program with constraints. If a computation C is a $\triangleright^{spt}_C$-computation then it is also a $\triangleright^s_C$-computation.*

Example 6. Let us consider the program P consisting of two rules

$$(\{b\}, \{\{b\}\}) \leftarrow (\{a\}, \{\{a\}\}) \qquad (\{a\}, \{\{a\}\}) \leftarrow (\{a,b\}, \{\emptyset, \{a\}, \{a,b\}\})$$

The sequence $\emptyset$, $\{a\}$, $\{a,b\}$ is a $\triangleright^s_C$-computation of P. On the other hand, it is not a $\triangleright^{spt}_C$ computation since $\emptyset \not\triangleright^{spt}_C (\{a,b\}, \{\emptyset, \{a\}, \{a,b\}\})$. □

Corollary 2. *Let P be a program with constraints. If M is an spt-answer set of P then M is a strong answer set of P.*

Next, we note that, similarly to our other classes of answer sets, the semantics defined by spt-answer sets also collapses to the standard answer-sets semantics for normal logic programs and to the answer-sets semantics for programs with convex constraints [11].

The approach based on the mapping spt takes a more "skeptical" perspective than the one embodied by the mapping s. To ensure persistence, it requires that *all* possible extensions of the state in which a constraint A holds satisfy A (and not only those actually encountered during the computation). In this way, the relations $\rhd_C^{spt}$ are, in a sense, the strongest relations satisfying the assumptions of Proposition 13. This may be perceived as a problem of this semantics — $\rhd_C^{spt}$-computations are localized to convex subfragments of constraints forming program rules. In other words, they cannot jump over "missing" satisfiers.

7 A Note on the Complexity

In this paper we introduced several classes of answer sets for programs with constraints. We have the following result concerning the computational complexity of the problem concerning the existence of answer sets.

Proposition 16. *Assuming an explicit representation of constraints, given a program with constraints, it is NP-complete to decide whether the program has an answer set (respectively, $\rhd$-answer set, strong answer set, spt-answer set).*

8 Discussion and Conclusions

We grounded our study in four basic principles: revision, persistence of beliefs, persistence of reasons, and convergence. We showed that there are several ways in which the principle of revision can be realized. In a least restrictive approach, we allow any element of $Concl_P(X)$ to be a valid revision of X. This choice defines the class of persistent computations and we take the results of such computations as the definition of the first notion of an answer set. In the case of normal logic programs and programs with convex constraints, computations capture precisely the concept of an answer sets as defined for these classes of programs earlier in [8,11].

More restrictive approaches to the revision principle narrow down choices offered by $Concl_P$ to those offered by $T_P^{nd;\rhd}$, where $\rhd$ is a sub-satisfiability relation. The results of persistent $\rhd$-computations form another class of answer sets, $\rhd$-answer sets, which forms a proper subclass of the previous one. However, in the case of normal logic programs and programs with convex constraints both notions of the answer set coincide.

The final two approaches result from the two specializations of a general schema to define f-models of a program with constraints. The schema is designed to generalize the guess-and-check approach for normal logic programs. It relies on a mapping that assigns to weak computations quasi-satisfiability relations. We demonstrated two mappings, s and spt, for which the resulting weak computations are in fact persistent computations and so, their results can be used as answer sets. The mapping s seems to be more appropriate as it is less restrictive. The mapping spt, on the other extreme of

the spectrum, seems to be too restrictive. As we noted earlier programs that intuitively should have answer sets do not have spt-answer sets.

This work draws attention to the concept of computation, and shows that, for programs with arbitrary constraints, there are many classes of computations of interest. In general, they give rise to different classes of answer set, by formalizing in different ways the negation-as-failure implicitly present in (non-monotone) constraints. Three classes of computations seem especially well suited as the basis for the generalization. Specifically, s-answer sets, $\triangleright$-answer sets and answer sets are viable candidates for the answer set semantics of programs with constraints, as they are grounded in some basic and intuitive principles imposed on computations and on schemata to define computations generalizing those used earlier in the context of normal logic programs. The class of spt-answer sets may be too restrictive. The issue whether any of the three classes identified here has any significant advantage over the other two requires further studies.

We note that some of our methods go beyond generalizations of just answer sets. if we weaken requirements on computations and consider the class of weak computations, the general schema of defining f-models of programs yields characterizations of supported models and mr-answer sets [15] and so also deserves further attention.

References

1. Balduccini, M., Gelfond, M., Nogueira, M.: Answer Set Based Design of Knowledge Systems. Annals of Mathematics and Artificial Intelligence (2006)
2. Dell'Armi, T., et al.: Aggregate Functions in Disjunctive Logic Programming: Semantics, Complexity, and Implementation in DLV. In: IJCAI, pp. 847–852 (2003)
3. Denecker, M., Pelov, N., Bruynooghe, M.: Ultimate well-founded and stable semantics for logic programs with aggregates. In: Codognet, P. (ed.) ICLP 2001. LNCS, vol. 2237, pp. 212–226. Springer, Heidelberg (2001)
4. Elkabani, I., Pontelli, E., Son, T.C.: Smodels with CLP and its applications. In: Demoen, B., Lifschitz, V. (eds.) ICLP 2004. LNCS, vol. 3132, pp. 73–89. Springer, Heidelberg (2004)
5. Erdem, E., Lifschitz, V., Ringe, D.: Temporal phylogenetic networks and logic programming. TPLP 6(5), 539–558 (2006)
6. Faber, W., Leone, N., Pfeifer, G.: Recursive aggregates in disjunctive logic programs: Semantics and complexity. In: Alferes, J.J., Leite, J.A. (eds.) JELIA 2004. LNCS (LNAI), vol. 3229, pp. 200–212. Springer, Heidelberg (2004)
7. Ferraris, P.: Answer sets for propositional theories. In: Logic Programming and Nonmonotonic Reasoning, pp. 119–131. Springer, Heidelberg (2005)
8. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: Joint Int. Conf. and Symp. on Logic Programming, pp. 1070–1080. MIT Press, Cambridge (1988)
9. Gelfond, M.: Representing Knowledge in A-Prolog. In: Computational Logic: Logic Programming and Beyond, pp. 413–451. Springer, Heidelberg (2002)
10. Heljanko, K., Niemelä, I.: Bounded LTL model checking with stable models. Theory and Practice of Logic Programming 3(4,5), 519–550 (2003)
11. Liu, L., Truszczyński, M.: Properties of programs with monotone and convex constraints. In: National Conference on Artificial Intelligence, AAAI/MIT Press, pp. 701–706 (2005)
12. Marek, V.W., Nerode, A., Remmel, J.: Logic Programs, Well-orderings, and Forward Chaining. Annals of Pure and Applied Logic 96, 231–276 (1999)
13. Marek, V.W., Truszczyński, M.: Nonmonotonic Logic; Context-Dependent Reasoning. Springer, Heidelberg (1993)

14. Marek, V.W., Truszczyński, M.: Stable models and an alternative logic programming paradigm. In: The Logic Programming Paradigm, pp. 375–398. Springer, Heidelberg (1999)
15. Marek, V.W., Remmel, J.B.: Set constraints in logic programming. In: Logic Programming and Nonmonotonic Reasoning, pp. 167–179. Springer, Heidelberg (2004)
16. Marek, V.W., Truszczyński, M.: Logic programs with abstract constraint atoms. In: National Conference on Artificial Intelligence (AAAI) AAAI Press / The MIT Press (2004)
17. Niemelä, I.: Logic programming with stable model semantics as a constraint programming paradigm. Annals of Mathematics and Artificial Intelligence 25(3,4), 241–273 (1999)
18. Pelov, N.: Semantic of Logic Programs with Aggregates. PhD thesis, K.U. Leuven (2004)
19. Son, T.C., Pontelli, E.: A Constructive Semantic Characterization of Aggregates in Answer Set Programming. Theory and Practice of Logic Programming (2007)
20. Son, T.C., Pontelli, E., Elkabani, I.: An Unfolding-Based Semantics for Logic Programming with Aggregates. Computing Research Repository (2006) (cs.SE/0605038)
21. Son, T.C., Pontelli, E., Tu, P.H.: Answer Sets for Logic Programs with Arbitrary Abstract Constraint Atoms. Journal of Artificial Intelligence Research (Accepted 2007)
22. van Emden, M.H., Kowalski, R.A.: The Semantics of Predicate Logic as a Programming Language. Journal of the ACM 23(4), 733–742 (1976)

Resource-Oriented Deadlock Analysis

Lee Naish

University of Melbourne, Melbourne 3010, Australia
lee@csse.unimelb.edu.au
http://www.csse.unimelb.edu.au/~lee/

Abstract. We present a method of detecting if deadlocks may occur in concurrent logic programs. Typical deadlock analysis is "process-oriented", being based on possible interleaving of processes. Our method is oriented towards the shared resources (communication channels, locks *et cetera*) and is based on orders in which individual resources are used by different processes. In cases where there are resources used by only a subset of all processes the search space can be dramatically reduced. The method arises very naturally out of the concurrent logic programming paradigm. Analysis of concurrent programs has previously used "coarsification" and "partial order" methods to reduce the search space. Our approach rediscovers and also extends these techniques. Our presentation is based around a logic programming pearl which finds deadlocked computations in a program which solves the dining philosophers problem.

Keywords: Concurrency, deadlock analysis, partial order, dining philosophers, committed choice nondeterminism, floundering, coroutining.

1 Introduction

Concurrent programming is a core area of computer science. Concurrent processes (or threads) may operate on a single CPU, multiple CPUs on a single computer, or over a computer network. Processes often need temporary exclusive access to shared resources such as data storage, communication channels and hardware devices. This requires concurrency control — mechanisms to suspend a process while another process is using a shared resource and resume the process when the use has finished. Implementing concurrent systems is notoriously difficult and one important class of errors is *deadlocks*, where all processes suspend, waiting for each other. Whether a particular system can deadlock is an important question.

In this paper we explore the area from the perspective of concurrent logic programming. By removing the "committed choice" nondeterminism and retaining essentially the same control information, concurrent logic programs can be transformed into pure logic programs which can be run to find all possible deadlocks. Our presentation is based around a program to solve the dining philosophers problem, which we describe initially. We then briefly describe the most basic technique of conventional deadlock analysis. We then introduce concurrent logic

V. Dahl and I. Niemelä (Eds.): ICLP 2007, LNCS 4670, pp. 302–316, 2007.

programming and present our program. The transformation to eliminate committed choice nondeterminism is then described and we present some analysis results. We discuss why the transformation is effective and correct, and compare it with some established deadlock analysis techniques, then conclude.

2 The Dining Philosophers Problem

In the dining philosophers problem [1] there are several philosophers sitting around a table. There is food in the center of the table, each philosopher has a plate, and between each is a fork (or chop stick). In order to eat, a philosopher must acquire both adjacent forks. Each philosopher alternately thinks, becomes hungry, gets one fork then the other (which may require waiting for adjacent philosophers to stop using them), eats, puts back the forks and resumes thinking. The aim is to simulate this situation in software, with a concurrent process for each philosopher, ensuring this cycle can continue indefinitely. Since each fork is shared by two adjacent philosophers, some form of concurrency control is needed.

One naive implementation is for each philosopher to first acquire the fork on their right with some form of atomic operation (using a lock, semaphore, monitor, *et cetera*), then acquire the fork on their left in a similar way. With this algorithm all philosophers may successfully obtain the first fork but then wait indefinitely for the second fork, creating a deadlock. Deadlocks can be avoided by making the acquisition of both forks a single atomic operation, but this requires central concurrency control shared by all philosophers, leading to greater contention. A better method is to number the philosophers consecutively and have odd numbered philosophers first get the fork on their left and even numbered philosophers first get the fork on their right. The coding we will present allows for both "left handed" and "right handed" philosophers and also allows philosophers who alternate between the two. We use a form of lock (one for each fork) as the concurrency control mechanism.

3 Process-Oriented Deadlock Analysis

Whether a deadlock occurs in practice depends not only on the algorithm, but also timing factors during each execution. Simple testing is thus an unreliable way of determining if a deadlock can occur so it is desirable to have specialised tools and techniques. The normal approach is to consider possible interleavings of operations done by the different processes. To reduce the number of interleavings considered some sequences of operations within each process are treated as a single operation (this is called coarsification). For example, the thinking phase of a philosopher may be very complex but because it is independent of the other processes it is not necessary to consider all ways it can be interleaved. It is sufficient to consider all interleavings of "chunks" which include a single atomic operation which can influence or be influenced by other processes. Analysis of imperative languages such as Ada and Java is complex but for an implementation

of the dining philosophers in such a language, analysis may reveal that the lock and unlock operations are the only critical ones.

Even with coarsification, the number of possible interleavings grows very rapidly, particularly with a reasonable number of processes. Later we present a program which will consider computations with a bound on the number of operations for each lock. For nine philosophers and up to eight operations for each lock the number of possible interleavings can be conservatively estimated as follows. First we assume exactly eight operations for each lock and each philosopher. The number of orderings of these operations is (8×9) factorial. However, the order of operations for each philosopher is fixed (rather than having 8! orderings), so the number of possible interleavings is (conservatively) $\frac{(8\times9)!}{(8!)^9} = 2.17 \times 10^{62}$. The search space can be pruned significantly by generating orderings incrementally and noting some prefixes are impossible, thus efficiently eliminating all orderings with that prefix, but even with such pruning the size of the search space is many orders of magnitude larger than we can effectively search. Various techniques have been devised to reduce the size of the search space further and using the logic programming paradigm we can very naturally exploit and extend such techniques — see Section 6. Even with the simple pruning outlined above the space can be searched completely in a reasonable time frame.

4 Concurrent Logic Programming

There are several forms of parallelism or concurrency which can be exploited in logic programs. The one which is relevant here is often referred to as stream and-parallelism. Numerous languages have evolved based on this general idea, including (variations of) Parlog [2], Concurrent Prolog [3], Guarded Horn Clauses [4] and Parallel NU-Prolog (PNU-Prolog) [5]. We will use PNU-Prolog in our examples.

In these languages, several subgoals in the current resolvent can be selected and matched with clauses concurrently. Each subgoal can be thought of as a process, its arguments being internal "state". A call to a tail-recursive procedure is analogous to a process which can change its state as the recursion proceeds. Different processes can communicate via shared variables. Shared lists can be thought of as streams of messages — the list elements can be "produced" (instantiated) by one "process" and obtained (via matching) by another "process". In the code we present the philosopher processes will each produce two streams of messages requesting (exclusive access to) the two adjacent forks.

4.1 Deterministic Code

Unlike Prolog, backtracking is not normally supported, due to the difficulty of implementing it in a distributed way. To avoid the need for backtracking, it is important that procedures can suspend until their input arguments are sufficiently instantiated. Typically they will suspend until just one clause matches (an important class of exceptions is discussed in the next section), and the implementation

must ensure that the matching clause can be found without speculatively binding any variables which may be shared by other subgoals which may be selected concurrently. The suspension mechanism can also be used for concurrency control — even if there is only one (matching) clause, a subgoal may still suspend until some (part of) an argument or arguments are non-variables. It is generally helpful to think of "modes": calls typically suspend rather than instantiate their "input" arguments. The implementation of NU-Prolog was adapted to support parallel execution for a subset of the language which does not require distributed backtracking and supports this style of programming. However, the full language supports (sequential) backtracking search and we use this to find deadlocks.

```
% wait until unlocked and a new lock request message arrives,
% allow entry to lock, recurse with Unlock flag from new message
:- lock1(U, L.Ls) when U and L. % causes process to wait
lock1(unlocked, lock(_I, Locked, Unlocked).Ls) :-
    Locked = locked, % back-communication
    lock1(Unlocked, Ls).

% Start lock1 process, initially unlocked
lock(Ls) :- lock1(unlocked, Ls).
```

Fig. 1. A process which can be used as a lock

Figure 1 gives code which can be used to implement a form of lock. A `lock1` "process" accepts a stream of lock request messages. A process using the lock will instantiate the next element of the message stream to `lock(I, Locked, Unlocked)`, where `Locked` and `Unlocked` are fresh variables, suspend until `Locked` is instantiated, perform whatever critical operations are required, then instantiate `Unlocked` to `unlocked` to release the lock. The `I` argument is for instrumentation purposes only — we use it to identify the process which requested the lock. The `lock1` process instantiates `Locked` to `locked`, but only when the lock becomes available (previous lock operations have been released). Although the stream of messages is an input to the process, the `Locked` components of the messages are outputs, a technique called "back-communication". The `when` declaration for `lock1` ensures the process suspends until a lock request message has been created and the previous one has been unlocked; only then will `Locked` be instantiated. It is possible for several messages to be instantiated, and several processes suspended, while the lock waits for an earlier instance to be unlocked. A top level call to `lock` creates a `lock1` process in an unlocked state. For brevity, we have not included a base case for `lock1` (and other code below) since the dining philosophers are perpetual processes. A more typical application would instantiate the stream to an empty list when the lock is no longer required, and have another clause to simply succeed in that case.

Figure 2 implements a philosopher process. The sequential nature of philosophers first thinking then attempting to get one fork, then the other, then eating

```
% think for a while...
p_think(P, LMs, RMs) :- p_hungry(P, LMs, RMs).

% getting hungry...
p_hungry(P, LMs, RMs) :- p_get_first(P, LMs, RMs).

% issue lock operation for first fork
% (either right or left depending on handedness)
:- p_get_first(p(_, H, _), _, _) when H.
p_get_first(p(I, r, B), LMs, lock(I, Locked, U1).RMs) :-
        p_get_second(p(I, r, B), Locked, U1, LMs, RMs).
p_get_first(p(I, l, B), lock(I, Locked, U1).LMs, RMs) :-
        p_get_second(p(I, l, B), Locked, U1, LMs, RMs).

% wait until we have first fork, issue lock operation for second
:- p_get_second(p(_, H, _), Locked, _, _, _) when H and Locked.
p_get_second(p(I, l, B), locked, U1, LMs, lock(I, Locked, U2).RMs) :-
        p_eat(p(I, l, B), Locked, U1, U2, LMs, RMs).
p_get_second(p(I, r, B), locked, U1, lock(I, Locked, U2).LMs, RMs) :-
        p_eat(p(I, r, B), Locked, U1, U2, LMs, RMs).

% wait until we have second fork, eat!
:- p_eat(_, Locked, _, _, _, _) when Locked.
p_eat(P, locked, U1, U2, LMs, RMs) :-
        p_release_both(P, U1, U2, LMs, RMs).

% release both forks, recomputed handedness, go back to work
p_release_both(P, unlocked, unlocked, LMs, RMs) :-
        next_handedness(P, P1),
        p_think(P1, LMs, RMs).

:- next_handedness(p(I, H, B), _) when H and B.
next_handedness(p(I, l, 0), p(I, l, 0)).
next_handedness(p(I, r, 0), p(I, r, 0)).
next_handedness(p(I, l, 1), p(I, r, 1)).
next_handedness(p(I, r, 1), p(I, l, 1)).
```

Fig. 2. A dining philosopher

then releasing the forks is achieved using a sequence of procedures, each calling the next. A process identifier `P`, of the form `p(I, H, B)`, is passed to each procedure. The first argument, `I` is an integer used for instrumentation purposes (we include it in lock messages). `H` and `B` are used to define the behaviour of the philosopher. `H` is either `l` or `r`, meaning the philosopher is currently left or right handed, respectively (which determines which fork is picked up first). `B` is either 0, resulting in consistent "handedness" or 1, resulting in alternation between left and right handed behaviour. After a philosopher finishes eating `I` remains unchanged but the new handedness is computed (more complex computations

could clearly be used here). Two streams of messages, `LMs` and `RMs`, are used to communicate with lock processes for the left and right forks, respectively. When declarations are used to suspend calls until the locks are obtained (`Locked` is instantiated) and only one clause matches (`P` needs to be instantiated to varying degrees in different procedures). PNU-Prolog has a preprocessor which accepts a form of mode declaration (for example, `:- lazyDet p_get_first(i, o, o)`) which can be used to generate appropriate `when` declarations for the code above (including the lock code). This is easier and less error prone than writing `when` declarations manually but we avoid relying on it here to make this presentation more self-contained.

4.2 Committed Choice Nondeterminism

Each philosopher binds successive elements of the message streams corresponding to adjacent forks. Thus, for each fork there are actually two streams of messages, and these must be merged into a single stream which is passed to a lock process. The way these streams are merged must be nondeterministic, to allow different interleaving of philosopher processes dependent on how long each one thinks for *et cetera*. The `merge` procedure should proceed when either philosopher instantiates their stream. However, since backtracking is not supported, *committed choice* (also called "don't care") nondeterminism is used — even if more than one clause can match, only one is ever chosen.

```
% nondeterministic committed choice merge of two (infinite) streams
% :- eagerDet merge(i, i, o). % specifies modes, committed choice
% merge(A.As, Bs, A.Cs) :- !, merge(As, Bs, Cs).
% merge(As, B.Bs, B.Cs) :- !, merge(As, Bs, Cs).
% The preprocessor translates this to...
:- merge(A, B, C) when A or B.
merge(A, B, C) :- nonvar(A), A = [D|E], !, C = [D|F], merge(E, B, F).
merge(A, B, C) :- nonvar(B), B = [D|E], !, C = [D|F], merge(A, E, F).
```

Fig. 3. Committed choice merge

Figure 3 shows how `merge` can be coded in PNU-Prolog. As with the lock code, base cases could be added. The comments show a mode declaration and high level version of the clauses, similar to how `merge` would be coded in Parlog. The first clause can proceed without instantiating the inputs if the first argument is a nonvariable; similarly for the second clause and second argument. The `when` declaration for `merge` is a *disjunction* of these conditions. The `nonvar` tests are also added to the clauses, thus the control information is used twice. Head unification of input arguments is moved into the body, after these tests, and output unifications are moved after the commit operator (cut) to avoid speculative instantiation. If both inputs are instantiated, most concurrent logic programming languages don't specify which clause will be selected and our deadlock analysis technique reflects this.

In practice, most implementations (including NU-Prolog) would always select the first clause. Whatever clause is selected, it is committed to.

Unlike the other definitions we have given, the presence of `nonvar` and cut make a declarative reading of `merge` difficult. Cuts destroy completeness, eliminating potentially successful derivations and `nonvar` tests cause some derivations to fail, but in a time-dependent way (whether `nonvar` succeeds can depends on when it is called). Most stream and-parallel languages do not use explicit `nonvar` tests, but they are implicit in the implementation (this is also true in PNU-Prolog if the preprocessor is used). In typical stream and-parallel code, there are generally very few of these genuine uses of committed choice nondeterminism (most languages insist on commits in every clause, but most calls suspend until a single clause matches and all arguments are sufficiently instantiated so the commits and implicit `nonvar` tests have no effect).

The PNU-Prolog preprocessor supports two distinct kinds of declarations. LazyDet declarations cause suspension until input arguments are sufficiently instantiated for *all* clauses (no clause would further instantiate these arguments by head unification or any calls before a cut). If any input argument is used in a call before a cut the procedure delays until that input is ground. Cuts then behave like a logical if-then-else and `nonvar` tests are not needed. EagerDet declarations, used for committed choice nondeterminism, cause suspension until input arguments are sufficiently instantiated for *some* clause (hence the condition in the delay declaration is typically a disjunction). Unifications (or calls) using input arguments are preceded by `nonvar` (or `ground`) tests to make sure the wrong clauses are not selected.

Figure 4 shows how several philosophers and locks communicate using `merge`. Calling `main` with the first argument instantiated to a list of appropriate process identifiers will initiate an instance of the dining philosophers problem. With an appropriate environment the incremental instantiation of the message streams to the lock processes could be observed. If the computation deadlocks then the NU-Prolog interactive top level prints a message to that effect and the final state of instantiation of the message streams is displayed.

```
% Just 3 philosophers...
% We pass in an id for each philosopher (which encodes the behaviour).
% The streams of lock messages are returned for instrumentation.
main([P1,P2,P3], [Ms1,Ms2,Ms3]) :-
    lock(Ms1), lock(Ms2), lock(Ms3),
    p_think(P1,Ms1a,Ms2b), p_think(P2,Ms2a,Ms3b), p_think(P3,Ms3a,Ms1b),
    merge(Ms1a,Ms1b,Ms1), merge(Ms2a,Ms2b,Ms2), merge(Ms3a,Ms3b,Ms3).
```

Fig. 4. Three dining philosophers sharing resources

5 Finding Deadlocks

To find deadlocks we propose transforming the original program in three stages. The first, and most important, replaces “don’t care” nondeterminism with “don’t

know" nondeterminism, so all possible derivations are explored rather than just one. The second transformation eliminates some spurious deadlocked derivations introduced by the first transformation. Some Prolog systems have facilities which allow alternative methods of eliminating these derivations. The third transformation allows a fair search to be implemented; other mechanisms for this are also possible.

5.1 Eliminating Committed Choice Nondeterminism

If we want to explore all possible deadlocked derivations then clearly the cuts in `merge` are not a good idea. If we eliminate them the code cannot be run in parallel, but it is still a valid NU-Prolog program which can be run sequentially. Some derivations may terminate with some calls still delayed by the `when` declarations. In sequential Prolog this is normally called *floundering*, but it is identical to deadlock in stream and-parallel logic programming. However, the `nonvar` tests can still prevent us from finding all deadlocked derivations. If the `nonvar` call is selected too early it may fail, whereas if it had been selected later its argument may then have been instantiated and the call would have succeeded. Thus a derivation which deadlocks for *some* timing of calls (computation rule) may not be found. If we simply eliminate the `nonvar` calls then `merge` can instantiate its inputs, resulting in unbounded numbers of spurious derivations which would not have occurred in the original program. A solution to this problem is to (at least initially) *delay* the evaluation rather than fail due to a `nonvar` test. However, delaying just the `nonvar` tests would not avoid any of the spurious derivations being explored; we need to delay the use of the whole clause.

Transformation 1

Let P be a committed choice procedure in the original program with N arguments and M clauses, the i^{th} clause being $P(\bar{A}_i)$`:-`B_i. Each clause is transformed into $P(\bar{V})$`:-`$Q_i(\bar{V})$, where $\bar{V}$ is a sequence of N distinct variables and Q_i is a new procedure name not appearing elsewhere. Each of the M new procedures Q_i is defined with a single clause, $Q_i(\bar{A}_i)$`:-`B_i. We assume the control information on the original procedure is equivalent to that generated by some `eagerDet` declaration. That is, P delays until at least one clause is sufficiently instantiated to proceed without further instantiating input arguments before the cut, and each clause has appropriate `nonvar` (or `ground`) tests to prevent it being selected inappropriately. The control information on each new procedure is that which would be generated by the same `eagerDet` declaration (it delays until the input arguments are sufficiently instantiated and `nonvar` tests are added). Since there is a single clause in each Q_i the cuts are redundant and are removed[1]. The `nonvar` tests are also redundant but are retained because they become useful in

[1] PNU-Prolog actually allows nondeterministic (backtracking) code to be called before a cut. This is not supported in other committed choice languages and for simplicity we ignore it here.

our second transformation. The control information for P remains unchanged. That is, the condition in the `when` declaration is the disjunction of the conditions in the `when` declarations for the new procedures.

The logic of the transformed code is equivalent to the logic of the original code with cuts removed. However, the control information for each clause of P is now used three times: in the `when` declaration and the body (the `nonvar` calls) of new procedure and in the (disjunctive) condition of the `when` declaration for P. Figure 5 contains the transformed code for `merge`.

```
:- merge(A, B, C) when A or B.
merge(A, B, C) :- m1(A, B, C).
merge(A, B, C) :- m2(A, B, C).

:- m1(A, B, C) when A.
m1(A, B, C) :- nonvar(A), A = [D|E], C = [D|F], merge(E, B, F).

:- m2(A, B, C) when B.
m2(A, B, C) :- nonvar(B), B = [D|E], C = [D|F], merge(A, E, F).
```

Fig. 5. Multiple solution merge

Derivations in the original parallel committed choice code and those in the transformed code are related as follows (this is discussed further in Section 6).

1. If a call to P in the original code suspends indefinitely (the inputs never become sufficiently instantiated), the same applies to the transformed code.
2. If a call to P in the original coed proceeds and all clauses could be selected for some timing (the inputs become instantiated enough for all clauses eventually), in the new code P will be called as well as (on backtracking) all new procedures.
3. If a call to P in the original proceeds but only some clauses could ever be selected (the inputs become instantiated enough for only some clauses eventually), in the new code P will be called and the corresponding new procedures will be called but calls to the other new procedures will suspend indefinitely.

Thus, if we ignore derivations in which a new procedure is floundered, the transformed code finds exactly the (successful and) deadlocked derivations of the original code. Although there are still some spurious derivations, each one results in just a single floundered call which binds no variables.

5.2 Eliminating Spurious Derivations

Some Prolog systems which support delays have a method of examining which calls are floundered and this could be used to filter out the unwanted floundered derivations with `m1` or `m2`; the `nonvar` tests could be eliminated, resulting in a

pure logic program. This would allow declarative diagnosis of floundering [6]. Alternatively, the program can be further transformed as follows.

Transformation 2

An extra argument is added to each new procedure introduced by transformation 1, and every procedure which (recursively) calls it. We choose a new variable `W`, not appearing elsewhere in the program, as the additional argument in all cases. If a new procedure has `when` declaration $P(T_1, ..., T_n)$ `when` C, the transformed procedure has `when` declaration $P(T_1, ..., T_n, \texttt{W})$ `when` C `or W`; other `when` declarations remain unchanged.

In addition to transforming the program, we ensure that the top-level goal instantiates `W` after all other computation succeeds or deadlocks. This results in any delayed calls to new procedures "waking up" (proceeding). The `nonvar` tests retained by transformation 1 are called and fail (the condition C of the `when` declaration is not satisfied), eliminating the spurious derivation. Derivations in which there are no new procedures delayed at the point where `W` is instantiated are not affected since no other calls delay on `W`.

5.3 Implementing Fair Search

Since derivations can be of unbounded length, it is important to implement a fair search in order to guarantee finding floundered derivations if they exist. There are numerous methods of implementing a fair search. Depth first iterative deepening is often a very efficient method and is well suited to Prolog. Unfortunately, the branching factor for a single "process" (such as a philosopher) is often close to one, making iterative deepening based on the length of these derivations inefficient. However, processes typically don't have unbounded computation between successive accesses to a shared resource (such as a fork). We have found that imposing a depth bound primarily on just calls to the nondeterministic procedures can be a particularly effective method.

Transformation 3

An extra argument is added to each nondeterministic procedure and every procedure which (recursively) calls it. We choose a new variable `S`, not appearing elsewhere in the program. The new argument is `S` except in the cases defined below. For each cycle C of the call graph which is connected to a nondeterministic procedure (each set of mutually recursive procedures which recursively call a nondeterministic procedure), in at least one procedure in C the new argument in each clause head is `s(S)`.

In the top level goal, `S` is instantiated to a depth bound using successor notation, for example `s(s(0))`. For a given depth, the number of nondeterministic calls is limited. In our Dining Philosophers program there are two cycles in the call graph. One consists of `merge` and `m1`, the other consists of `merge` and `m2`.

```
:- merge(A, B, C, W, S) when A or B.
merge(A, B, C, W, s(S)) :- m1(A, B, C, W, S).
merge(A, B, C, W, s(S)) :- m2(A, B, C, W, S).

:- m1(A, B, C, W, S) when A or W.
m1(A, B, C, W, S) :- nonvar(A), A = [D|E], C = [D|F], merge(E,B,F,W,S).

:- m2(A, B, C, W, S) when B or W.
m2(A, B, C, W, S) :- nonvar(B), B = [D|E], C = [D|F], merge(A,E,F,W,S).
```

Fig. 6. Depth-bounded flounder-preserving multiple solution merge

Thus it is sufficient to use the `s(S)` arguments in `merge` — see Figure 6. Thus the depth bound simply limits the depth of recursion in each call to `merge`, that is, the number of lock and unlock operations on each fork. If a call beyond the depth bound is sufficiently instantiated to proceed, it simply fails when `0` is matched with `s(S)` in a clause head. The only derivations which are not failed are those in which all `merge` calls flounder before reaching this depth.

```
main(S, [P1,P2,P3], [Ms1,Ms2,Ms3]) :-
    lock(Ms1), lock(Ms2), lock(Ms3),
    p_think(P1,Ms1a,Ms2b), p_think(P2,Ms2a,Ms3b), p_think(P3,Ms3a,Ms1b),
    merge(Ms1a,Ms1b,Ms1,W,S), merge(Ms2a,Ms2b,Ms2,W,S),
    merge(Ms3a,Ms3b,Ms3,W,S),
    W = wake.            % resume + fail spurious deadlocks with m1/m2

% generate increasing depths
depth(0).
depth(s(S)) :- depth(S).

% as above, also prints each depth
d(S) :- depth(S), write('Depth limit: '), writeln(S).
```

Fig. 7. Three floundering philosophers with depth limit

By simply using our transformed `merge` procedure and instantiating the wake up variable `W` in the rightmost subgoal, our dining philosophers program will find exactly those derivations which would (succeed or) deadlock in the original program. By instantiating the depth bound to a fixed value or generating increasing depth bounds, we can limit the search or ensure fairness, respectively — see Figure 7 for code with three philosophers. Figure 8 is a sample run, showing how deadlock can occur if the second philosopher gets to eat first then all philosophers pick up their left fork. Using nine philosophers numbered 1–9 where even numbered philosophers are right handed, odd numbered philosophers are left handed and all philosophers alternate their handedness, the program has successfully verified there are three ways deadlock can occur with up to eight lock

or unlock operations per fork. The first is where all even numbered philosophers eat once, then all philosophers pick up their left fork then attempt to pick up their right fork (17 locks obtained, 8 unlocks and 9 locks which suspend, found at depth 3). The other two involve all odd numbered philosophers eating once (19 locks obtained, 10 unlocks and 9 locks which suspend, found at depth 4). Since first and last philosopher share a fork there are two distinct orders to consider and that fork has more operations than others (3 locks obtained, 2 unlocks and one lock which suspends). A complete search up to depth four (up to eight operations for each fork) for nine philosophers takes around 100 to 300 seconds (depending on the behaviour of the philosophers and their arrangement) running NU-Prolog on a 1GHz Intel Pentium III processor. Memory requirements are minimal due to the naive backtracking search.

```
?- d(S), main(S,[p(1,l,1), p(2,r,1), p(3,l,1)], [Ms1, Ms2, Ms3]).
Depth limit: 0
Depth limit: s(0)
Depth limit: s(s(0))
Depth limit: s(s(s(0)))
Warning: Goal floundered.
Ms3 = [lock(2, locked, unlocked), lock(3, locked, _), lock(2, _, _)| _],
Ms2 = [lock(2, locked, unlocked), lock(2, locked, _), lock(1, _, _)| _],
Ms1 = [lock(1, locked, _), lock(3, _, _)| _],
S = s(s(s(0)))
```

Fig. 8. Three floundering philosophers: sample goal

6 Discussion

The ability to solve problems of this complexity is because the size of search space is dramatically reduced. Each call to `merge` in the Prolog program can match with two clauses, so the number of `merge` nodes in the tree at depth four is (conservatively) $2^{(9\times4)} = 6.87 \times 10^{10}$ (not all branches exist because some calls to `m1` and `m2` flounder instead of recursively calling `merge`). Each pair of lock and unlock operations results in a branching factor of two, compared with a branching factor of (around) 9×9 if all (coarsified) interleavings are considered.

The crucial reason for this is that in logic programming with well designed primitives for delaying calls, floundering is independent of the computation rule (the order in which subgoals are selected) [7][8]. For any branch of the search space (which is uniquely determined by the clauses selected in the `merge` calls), every computation rule consistent with the `when` declarations eventually calls the same subgoals and binds the same variables to the same values. This is why we avoid having to consider each execution order (interleaving) and why there is a simple relationship between derivations in the original parallel code and the transformed backtracking version. It essentially gives us optimal coarsification "for free" since deterministic code never leads to additional branches in the search space. It also allows us to significantly reduce the branching factor when

there are timing-dependent operations. Instead of the branching factor being the total number of processes, it is typically the degree of contention for the resource — the number of processes attempting to access the resource. If three philosophers had to share a fork our code would implement a three-way merge and the branching factor would be three instead of two. The search space is determined by the number of orderings of operations on shared resources. For this reason we refer the method as being resource oriented rather than process oriented.

Inter-process communication which does not involve contention, such as one process producing a stream of data and another process consuming it, does not increase the size of the search space. For example, our program could be adapted so philosophers could communicate with each other via an extra argument on the `lock` and `unlocked` terms (like a philosopher attaching a note to a fork as it is put down). Even logic variables can be passed around the ring to set up private communication channels between pairs of philosophers. Potentially, the logical positivists and the post-modernists could each team together and cooperate in complex ways; the existentialists may starve to death. As long as such things are done in an essentially synchronous way, without contention and without the code using committed choice nondeterminism, the search space remains the same.

The order in which time-dependent accesses to resources are performed (for example, the order of lock operations in the output arguments of the `merge` calls) plus the sequence of operations performed by each process determines a *partial order* for all operations. The search space is reduced by considering all such partial orders rather than the much larger number of total orders. Partial order methods of concurrency analysis have been investigated previously [9] and rely on recognising "independence" of operations or state transitions (in fact, coarsification can also be seen as a partial order method). There are two components to independence. One is the commutativity of operations. In pure logic programming essentially the only operation is the procedure call (which incorporates unification), and this is commutative. The only calls which do not commute are those to "impure" committed choice procedures. The other component of independence is related to the control aspect: independent operations do not enable or disable (affect suspension of) each other. In logic programming languages where floundering or deadlock is independent of the computation rule this aspect of independence is not required. There is greater independence and thus our method can examine fewer partial orders in the search space.

In the logic programming paradigm, independence is the natural state of affairs and all dependence on execution order is made explicit using committed choice nondeterminism. Specifically, binding shared variables, such as performing an unlock operation, clearly affect suspension of other processes but do not increase the search space in our method. In our code the order of releasing both forks is not defined — it is not guaranteed to be an atomic operation and could be done in either order, depending on the compiler and possibly runtime factors. With two philosophers, P_1 and P_2, both unlock operations of P_1 potentially affect suspension of P_2 but our analysis method avoids considering both possible

orders. Another advantage of using the logic programming paradigm is that dynamic process creation and termination require no additional complexity. In our program the call to `next_handedness` could spawn a new process so it could be done in parallel with the philosopher thinking. Synchronisation is not required until `p_get_first` is called, but this is all transparent in our technique.

The key requirement for floundering to be independent of the computation rule is that the "callable atom set" is closed under instantiation. That is, if an atom can be called (not suspend), any instance of the atom can also be called. This is a form of monotonicity with respect to callability; a suspended "process" may become callable due to the actions of other processes but this cannot be undone by further actions. In [7] is was noted that some proposed concurrent logic programming languages do not have this property (and our technique does not apply to such languages). Similarly, some concurrency control primitives in imperative languages are not monotonic: a process which is not blocked can become blocked by the action of other processes. However, in cases where the primitives are monotonic (for example, unlock operations), it seems that the definition of independence could be weakened, leading to the smaller search space we have uncovered by our technique. This could add to the arsenal of existing techniques for reducing the search space.

The code we have presented uses the naive depth first search strategy of Prolog but many more sophisticated search algorithms could be used. For example, intelligent backtracking is familiar to logic programmers, the SPIN concurrency analysis system uses depth first search augmented with a hashing method to detect states which are identical on different branches of the search space, trading space for time. Such forms of tabling along with abstract interpretation techniques can be used to make search spaces finite, though some precision may be lost. These methods, although important, are orthogonal to the partial order techniques used to initially reduce size of the search space.

7 Conclusion

Analysis of concurrent imperative programs is very challenging. Potentially, every operation can influence every process and to determine if a deadlock is possible, all interleavings of executions need to be considered. Analysis techniques have been developed which allow detection of some sub-computations which are independent, greatly reducing the size of the search space. In the concurrent logic programming paradigm, independence of sub-computations is the normal state of affairs and even if sub-computations can influence each other during the execution, the end result is independent of the execution order. Timing-dependent operations can be coded using committed choice nondeterminism, but this is the exception rather than the rule. By eliminating committed choice nondeterminism and retaining essentially the same control information, concurrent logic programs can be simply transformed to allow detection of possible deadlocks. By exploiting the independence of execution order in this way, the techniques developed for imperative programs are emulated and improved. The logic of the

underlying problem is neatly exposed, allowing gains in efficiency, and the search can be performed by the built-in mechanisms of Prolog. Because our program and general technique highlight several advantages of the logic programming paradigm, we propose it as a logic programming pearl.

References

1. Dijkstra, E.: Hierarchical ordering of sequential processes. Acta Informatica 1, 115–138 (1971)
2. Gregory, S.: Parallel logic programming in PARLOG: the language and its implementation. Addison-Wesley, Reading (1987)
3. Shapiro, E.: A subset of Concurrent Prolog and its interpreter. In: Shapiro, E. (ed.) Concurrent Prolog: Collected Papers, vol. I, pp. 27–83. MIT Press, London (1987)
4. Ueda, K.: Guarded Horn clauses. PhD thesis, University of Tokyo, Japan (1986)
5. Naish, L.: Parallelizing NU-Prolog. In: Proceedings of the Fifth International Conference/Symposium on Logic Programming, pp. 1546–1564 (1988)
6. Naish, L.: Declarative diagnosis of floundering. Technical report, Department of Computer Science, University of Melbourne (Submitted for publication, 2007)
7. Naish, L.: Coroutining and the construction of terminating logic programs. Australian Computer Science Communications 15(1), 181–190 (1993)
8. Naish, L.: A declarative view of floundering. Technical report, Department of Computer Science, University of Melbourne (Submitted for publication, 2006)
9. Godefroid, P.: Partial-Order Methods for the Verification of Concurrent Systems. LNCS, vol. 1032. Springer, Heidelberg (1996)

Static Region Analysis for Mercury*

Quan Phan and Gerda Janssens

Department of Computer Science, K.U. Leuven
Celestijnenlaan, 200A, B-3001 Heverlee, Belgium
{quan.phan,gerda.janssens}@cs.kuleuven.be

Abstract. Region-based memory management is a form of compile-time memory management, well-known from the functional programming world. This paper describes a static region analysis for the logic programming language Mercury. We use region points-to graphs to model the partitioning of the memory used by a program into separate regions. The algorithm starts with a region points-to analysis that determines the different regions in the program. We then compute the liveness of the regions by using an extended live variable analysis. Finally, a program transformation adds region annotations to the program for region support. These annotations generate data for a region simulator that generates reports on the memory behaviour of region-annotated programs. Our approach obtains good memory consumption for several benchmark programs; for some of them it achieves optimal memory management.

1 Introduction

Memory management is a classic problem in the implementation of programming languages. Manual memory management using explicit constructs, such as *malloc/free* in C, is widely known to be error-prone. Therefore automatic memory management in the form of runtime garbage collection (RTGC) has become an integral part in many modern programming languages. The use of RTGC is even more important for declarative logic programming (LP) languages because these languages have no procedural constructs for memory management, and instant reclaiming based on backtracking is not feasible in many logic programs. While runtime garbage collectors for LP languages can reclaim more than 90% of the heap space of a logic program, they incur execution overhead because collectors often need to temporarily stop the main program.

Recently, there has been increasing interest in compile-time memory management to reduce the overhead and unpredictability of RTGC. This static method generally follows two approaches: compile-time garbage collection (CTGC) and region-based memory management (RBMM). CTGC looks for program points at which allocated memory cells are no longer used and instructs the program to reuse those cells for constructing new terms, reducing the memory footprint and in some cases achieving faster code. In LP, this idea has been used to reuse memory cells locally in the procedures of Mercury programs [9, 10, 8]. The basic idea

* This work is supported by the project GOA/2003/08 and by FWO Vlaanderen.

V. Dahl and I. Niemelä (Eds.): ICLP 2007, LNCS 4670, pp. 317–332, 2007.

of RBMM is to divide the heap memory used by a program into different regions. The dynamically created terms and their subterms have to be distributed over the regions in such a way that, at a certain point in the execution of the program, all terms in a region are dead and the whole region can be removed. RBMM is a topic of intensive research for functional programming languages [16, 1, 4, 15] and more recently also for imperative languages [3, 2]. For LP languages, there has been only one attempt to apply RBMM to Prolog [7, 6, 5]. However, the algorithm for RBMM in [6, 5] was developed for a non-standard implementation of Prolog which would require substantial changes to have it implemented in any standard implementation. The authors of [7] fixed the problem by implementing RBMM in the context of a WAM-based Prolog system. Nevertheless, the work mainly concentrated on the runtime extensions needed by regions to run Prolog programs with RBMM. It reused a type-based region analysis which was developed for the functional programming language SML [4] to work with untyped Prolog, which could lead to poor memory behaviour for large programs in which type inference for LP obtains imprecise type information. The limited research on RBMM in LP, therefore, suggests that it is worthwhile to investigate how a dedicated static region analysis can be developed and implemented in the context of typed logic programming languages.

Mercury is a pure LP language designed with a native, expressive type system, which makes it a natural context for this research. The main contribution of this paper is an algorithm that augments Mercury programs with region annotations. In addition, we have also implemented a region simulator to run the region-annotated Mercury programs and to evaluate the memory performance of several benchmark programs. We find that the algorithm can be practically implemented, and that, with RBMM, the memory savings are very encouraging.

In Section 2, we explain how memory management for Mercury can be based on the use of regions. The whole algorithm is composed of three phases, which are described in Sections 3, 4, and 5, respectively. The concept of the region points-to graph and the points-to analysis are given in Section 3. Section 4 defines the live region analysis which uses the region points-to graph of each procedure to precisely detect the lifetime of the regions. We do the transformation by adding RBMM annotations in Section 5. Section 6 briefly discusses how the presented algorithm can support Mercury programs with backtracking. The prototype implementation of the algorithm and the experimental results are shown in Section 7. Finally, Section 8 concludes.

2 Regions and Mercury

Mercury programs. We assume that the input of our program analysis is a Mercury program that has been transformed by the Melbourne Mercury Compiler (MMC) into *superhomogeneous* form, with the goals reordered and each unification specialised into a *construction* (<=), a *deconstruction* (=>), an *assignment* (:=), or a *test* (==) based on the modes [14]. The *qsort* program in this form then consists of the following procedures as shown in Fig. 1.

```
main(!IO) :-
(1) L<=[2,1,3],
(2) A<=[],
(3) qsort(L,A,S),
(4) io.write(S, !IO),

qsort(L,A,S) :-
(
(1) L=>[],
(2) S:=A
;
(3) L=>[Le|Ls],
(4) split(Le,Ls,L1,L2),
(5) qsort(L2,A,S2),
(6) A1<=[Le|S2],
(7) qsort(L1,A1,S)
).
```

```
split(X,L,L1,L2) :-
(
(1) L=>[],
(2) L1<=[],
(3) L2<=[]
;
(4) L=>[Le|Ls],
    (
    (5) X>=Le
    ->
    (6) split(X,Ls,L11,L2),
    (7) L1<=[Le|L11]
    ;
    (8) split(X,Ls,L1,L21),
    (9) L2<=[Le|L21]
    )
).
```

Fig. 1. *qsort* program in superhomogeneous form

The use of regions: an example. We illustrate the usefulness of distributing terms over regions using the *qsort* example. Memory cells representing a list can be divided into cells for the elements and those for the list skeleton. Observing the memory behaviour of the *qsort* procedure, we see that the output list has a new skeleton built up in the accumulator while its elements are those of the input list. In the *main* predicate, the input list L is no longer used after the call to *qsort*. This means that if the skeleton of the input list, the elements, and the skeleton of the output list (and the accumulator) are stored in three different regions we can safely free the memory occupied by the input list's skeleton by removing its region after the call. Take a closer look inside the *qsort* procedure at (4). The call to *split* creates two new lists with two new skeletons while the elements are also those of the input list. Therefore, if the two new skeletons are stored in regions different from the region of the input list's skeleton, the region can even be removed earlier, namely after this call to *split* inside *qsort*. So, by storing different components of the lists in separate regions we can do timely removal and recover dead memory sooner.

The *qsort* program with region support produced by our analysis is shown in Fig. 2 with the region annotations in bold. In analogy to program variables used to refer to memory cells, in RBMM, we use *region variables* to refer to (i.e., to be bound to) physical regions in memory. Two special instructions *create* and *remove* handle the creation and removal of regions. **create(R)** creates a region, and makes the region variable **R** bound to it. **R** must be unbound before and be bound after the operation. **remove(R)** removes the region to which **R** is currently bound. **R** must be bound before and be unbound after the operation. That region variables can become unbound makes them different from regular Mercury variables. In a procedure definition, $\{\ldots,\mathbf{R_i},\ldots\}$ is the list of *formal region parameters*. At a call site of the procedure the caller will provide the *actual region parameters* in the usual manner of parameter passing. The information about which variables are stored in which regions will be given in Fig. 3.

```
main(!IO) :-
  create(R1), create(R2),
  (1) L<=[2,1,3],
  create(R3),
  (2) A<=[],
  (3) qsort(L,A,S){R1,R2,R3},
  (4) io.write(S, !IO),
  remove(R2), remove(R3).

qsort(L,A,S){R1,R2,R3} :-
(
(1) L=>[],
    remove(R1),
(2) S:=A
;
(3) L=>[Le|Ls],
(4) split(Le,Ls,L1,L2){R2,R1,R2,R4,R5},
(5) qsort(L2,A,S2){R5,R2,R3},
(6) A1<=[Le|S2],
(7) qsort(L1,A1,S){R4,R2,R3}
).

split(X,L,L1,L2){R5,R1,R2,R3,R4} :-
(
(1) L=>[],
    remove(R1),
    create(R3),
(2) L1<=[],
    create(R4),
(3) L2<=[]
;
(4) L=>[Le|Ls],
    (
    (5) X>=Le
    - >
    (6) split(X,Ls,L11,L2){R5,R1,R2,R3,R4},
    (7) L1<=[Le|L11]
    ;
    (8) split(X,Ls,L1,L21){R5,R1,R2,R3,R4},
    (9) L2<=[Le|L21]
    )
).
```

Fig. 2. Region-annotated *qsort* program

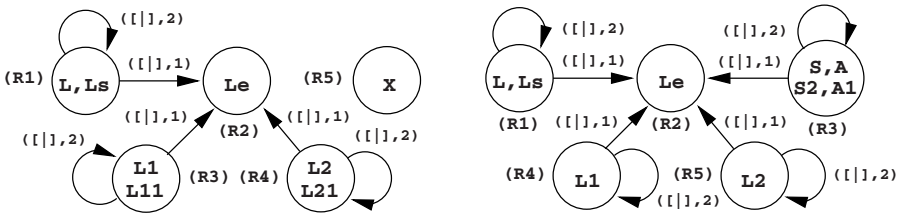
In Fig. 2, assume that in the *main* procedure, the list L's skeleton is in the region (bound to) **R1**, its elements in **R2**, and the skeleton of the accumulator in **R3**. In the *qsort* procedure, the region of the skeleton of the list L passed to *qsort* from *main* is removed in the base case branch of *split* in the call at (4). The two new skeletons of the lists *L1* and *L2* are allocated in two separate regions. These regions are created by the base case branch of *split*, and removed (indirectly) by the calls at (5) and (7). If *L1* and *L2* are empty lists the removals will happen in the base case branch of *qsort*; otherwise, they will happen in the base case branch of *split*. The region of the output list's skeleton is the region of the accumulator, which is created in *main*.

3 Region Points-to Analysis

The goal of this analysis is to build, for each procedure, a region points-to graph that represents the partitioning into regions of the memory used by the procedure. The concept of a region points-to graph was introduced for Java in [2] and we adapted it to Mercury.

Region points-to graph. For a procedure p, a region points-to graph, $G = (N, E)$, consists of a set of nodes, N, representing regions and a set of directed edges, E, representing references between the regions. Each node has an associated set of variables of p which are stored in the region represented by the node. For a node n, its set of variables is denoted by $vars(n)$. The node n_X denotes the node such that $X \in vars(n_X)$. A directed edge $(m, (f, i), n)$ denotes an edge from m to n, which is labelled by the type selector (f, i) and represents the structured relation between the variables in the two nodes. The type selector (f, i) selects the i^{th} argument of the functor f [8].

The points-to graphs of *split* and *qsort* procedures are shown in Fig. 3. For *split*, we see that the skeletons of the lists L and Ls are in the same region and that the list elements are in the region pointed to by the edge labelled by $([], 1)$. The self-edge with the label $([], 2)$ is for the skeleton.



(a) *split* (b) *qsort*

Fig. 3. The points-to graphs of *split* and *qsort*

The region points-to graph $G = (N, E)$ of a procedure p is *compatible* if and only if G satifies the following invariants.

1. If a unification $X := Y$ is in p then X and Y are in the same node.
2. If a unification $X = f(\ldots, Y_i, \ldots)$ (i.e., <= or =>) appears in p then $n_X, n_{Y_i} \in N$, and, for each i, there exists exactly one edge with the label (f, i) from n_X and $(n_X, (f, i), n_{Y_i}) \in E$.
3. If X is a variable bound to a term, Y is a variable bound to a subterm of that term and they are of the same type, then they are in the same node.
4. Every variable of p belongs to exactly one node and the variables in a node have the same type, which is regarded as the type of the node.
5. If p calls a procedure q, then the subgraph of the region points-to graph of p rooted at the nodes of the actual parameters must be a *conservative approximation* of the subgraph of the graph of q rooted at the nodes of the formal parameters of q.

The fifth invariant includes the effects of procedure calls in the region points-to graph of p. According to [2], a region points-to graph $G = (N, E)$ is a conservative approximation of $G' = (N', E')$ if there exists a function $\alpha : N' \to N$ such that $(n, (f, i), m) \in E'$ implies $(\alpha(n), (f, i), \alpha(m)) \in E \ \forall n, m \in N'$.

The reason for the third invariant is to ensure that the terms of recursive type are always stored in a fixed, finite number of regions. For a more thorough development of this design choice the reader is referred to Section 2.3 in [12].

The task of the region points-to analysis then is to produce a compatible graph for each procedure in a program. To update a region points-to graph $G = (N, E)$ the analysis uses two operations *unify* and *edge* defined as follows.

– $k = unify(n, m)$: unifies nodes n and m and returns the new node k.

  - $N \leftarrow N \setminus \{n, m\} \cup \{k\}$, k is a new node and $vars(k) = vars(n) \cup vars(m)$.
  - $E \leftarrow E$ where all appearances of m and n are replaced by k.
- $edge(n, sel, m)$: creates an edge with a label sel from node n to node m.
  - $G \leftarrow (N, E \cup \{(n, sel, m)\})$.

The region points-to analysis is flow-insensitive (i.e., the execution order of the literals in a procedure does not matter), and consists of an intraprocedural analysis and an interprocedural analysis. We will describe them in turn.

3.1 Intraprocedural Analysis

Assume that we are analysing a procedure p. Its region points-to graph $G = (N, E)$ is computed as follows.

1. Each variable in p is assigned to a separate node: for a variable X, n_X becomes a node in N and $vars(n_X) = \{X\}$.
2. The specialized unifications in p are processed one by one as follows:
   - An assignment $X := Y$: apply $unify(n_X, n_Y)$ to ensure the first invariant.
   - A test $X == Y$: do nothing.
   - A deconstruction $X => f(Y_1, \ldots, Y_n)$ or a construction $X <= f(Y_1, \ldots, Y_n)$: create the references from n_X to each of n_{Y_1}, ..., n_{Y_n} by adding the edges $edge(n_X, (f, 1), n_{Y_1})$, ..., $edge(n_X, (f, n), n_{Y_n})$.
3. The rules in Fig. 4 are fired whenever applicable. Rules P1 and P2 are to ensure the second invariant. Rule P3 enforces the third invariant.

$$\frac{\begin{array}{c} k = unify(n, n') \\ (k, sel, m) \in E \\ (k, sel, m') \in E \\ m \neq m' \end{array}}{unify(m, m')} \text{ (P1)} \qquad \frac{\begin{array}{c} edge(n, sel, m) \\ (n, sel, m') \in E \\ m \neq m' \end{array}}{unify(m, m')} \text{ (P2)} \qquad \frac{\begin{array}{c} edge(n, sel, m) \\ (k_1, m) \in E^+ \\ (m, k_2) \in E^+ \\ k_1 \neq k_2 \\ type(k_1) = type(k_2) \end{array}}{unify(k_1, k_2)} \text{ (P3)}$$

E^+ is the reflexive, transitive closure of E; $type(n)$ returns the type of node n.

Fig. 4. Intraprocedural analysis rules

3.2 Interprocedural Analysis

The interprocedural analysis updates the region points-to graph G_p of a procedure p by integrating the relevant parts of the region points-to graphs of the called procedures into G_p at each call site. For a call $q(Y_1, \ldots, Y_n)$, the head of the defining procedure is $q(X_1, \ldots, X_n)$. The analysis is performed as follows.

1. Process each procedure call $q(Y_1, \ldots, Y_n)$ in p: integrate the graph of q, $G_q = (N_q, E_q)$, into the graph of p, $G_p = (N_p, E_p)$ by building the partial α mapping from N_q to N_p as follows:
   (a) Initialize the α mapping with $\alpha(n_{X_1}) = n_{Y_1}, \ldots, \alpha(n_{X_n}) = n_{Y_n}$. For those nodes that have been unified in G_q, the corresponding nodes in G_p should also be unified. This is achieved by applying rule P4 in Fig. 5 to ensure that α is a function.

(b) In the graph G_q, start from each n_{X_i}, follow each edge once and apply rules P5 - P8 in Fig. 5 when applicable. Those rules complete the α mapping and copy the parts of G_q that are relevant to n_{X_i}'s into G_p. When two nodes of G_p are unified (rules P4 and P5) or an edge is added to G_p (rules P7 and P8) we need to apply rules P1 or P3 respectively to G_p in order to maintain the second and the third invariants. (Rule P2 is never applicable because its conditions cannot be satisfied.)

2. Repeat step 1 until there is no change in G_p.

$$\frac{\alpha(n_{X_i}) = n_{Y_i} \quad \alpha(n_{X_j}) = n_{Y_j} \quad n_{X_i} = n_{X_j} \quad n_{Y_i} \neq n_{Y_j}}{unify(n_{Y_i}, n_{Y_j})} \text{ (P4)}$$

$$\frac{(n_q, sel, m_q) \in E_q \quad \alpha(n_q) = n_p \quad (n_p, sel, m_p') \in E_p \quad \alpha(m_q) = m_p \neq m_p'}{unify(m_p, m_p')} \text{ (P5)}$$

$$\frac{(n_q, sel, m_q) \in E_q \quad \alpha(n_q) = n_p \quad (n_p, sel, m_p) \in E_p \quad \alpha(m_q)\ undefined}{\alpha(m_q) = m_p} \text{ (P6)}$$

$$\frac{(n_q, sel, m_q) \in E_q \quad \alpha(n_q) = n_p \quad \nexists k : (n_p, sel, k) \in E_p \quad \alpha(m_q) = m_p}{edge(n_p, sel, m_p)} \text{ (P7)}$$

$$\frac{(n_q, sel, m_q) \in E_q \quad \alpha(n_q) = n_p \quad \nexists k : (n_p, sel, k) \in E_p \quad \alpha(m_q)\ undefined}{m_p : a\ new\ node\ in\ G_p \quad \alpha(m_q) = m_p \quad edge(n_p, sel, m_p)} \text{ (P8)}$$

Fig. 5. Interprocedural analysis rules

When the algorithm terminates, i.e., when no rules can be applied and we reach a fixpoint, the resulting region points-to graphs for the program's procedures will all be compatible. This is because for each procedure, the analysis starts with a points-to graph that satisfies the fourth invariant, and no rules invalidate it. The first invariant is respected by the intraprocedural analysis. If the second invariant does not hold then rules P1 and P2 must be applied and also if the third invariant is invalid then rule P3 is applied. Rule P4 ensures that at a call site an α mapping exists. Then if the fifth invariant does not hold at least one among the rules P5-P8 must be applicable. The termination of the algorithm can be shown by defining a partial order over the set of region points-to graphs and showing that the rules are actually monotonic and that there exists a maximum compatible points-to graph for each procedure. From now on, when we mention the region points-to graph of a procedure it means the compatible one obtained after both intra- and inter-procedural analyses.

For the *qsort* example, the region points-to graphs of *split* and *qsort* after the region points-to analysis are exactly as shown in Fig. 3.

4 Live Region Analysis

Each node in the region points-to graph of a procedure is named by a distinct region variable. A region variable being live means that it is bound to a physical region. During the execution of a program the memory cells holding the values of the variables must be allocated before they are used. Similarly, in RBMM,

regions also need to be created before used, and should be removed when no long in use. The goal of live region analysis is to detect which region variables are live at each program point and to decide which regions are created and removed by each procedure.

We associate a *program point* with every literal in the body of a procedure p. An *execution path* in p is a sequence of program points, such that at runtime the literals associated with these program points are performed in sequence. We use the notions "before" and "after" a program point. Before a program point means the associated literal has not been executed yet, while after a program point means its literal has just been completed. The set of live region variables at a program point is computed via the set of live variables at the program point.

4.1 Live Region Variables at a Program Point

Live variables. A variable is live after a program point in a procedure p if:

- There exists an execution path in p containing the program point that instantiates the variable before or at the program point and uses it after the program point,
- OR it is an output variable of p, which is instantiated before or at the program point.

If we call $pre\_inst(i, P)$ the set of variables instantiated before the program point i in the execution path P, $post\_use(i, P)$ the set of variables used after i in P, $out(i)$ the set of variables instantiated by the goal at i, $out(p)$ the set of output variables of a procedure p then the set of live variables after i is:

$$LV_{after}(i) = \{V \mid \exists P : V \in (pre\_inst(i, P) \cup out(i)) \cap (out(p) \cup post\_use(i, P))\}.$$

If we call $in(i)$ the set of input variables to the literal at i, the set of live variables before i is:

$$LV_{before}(i) = (LV_{after}(i) \setminus out(i)) \cup in(i).$$

The LV_{before} of the first program point of an execution path of a procedure p is defined to be $in(p)$, the set of input variables of the procedure. The LV_{after} of the last program point is defined to be $out(p)$.

Live region variables. A region variable is live at a program point if its node is reachable from a live variable at the program point.

The set of nodes that are reachable from a variable is defined:

$$Reach(X) = \{n_X\} \cup \{m \mid \exists (n_X, m) \in E^*(X)\},$$

in which $E^*(X)$ is defined:

$$E^*(X) = \{(n_X, n_i) \mid \exists (n_X, sel_0, n_1), \ldots, (n_{i-1}, sel_{i-1}, n_i) \in E\}.$$

The live region variables sets before and after a program point i are defined:

$$LR_{before}(i) = \bigcup(Reach(X)) \ \forall X \in LV_{before}(i).$$
$$LR_{after}(i) = \bigcup(Reach(X)) \ \forall X \in LV_{after}(i).$$

4.2 Region Lifetime Across Procedure Boundary

The formal region parameters of a procedure are the region variables that are reachable from its head variables. For a procedure, we could safely require that the regions bound to by its formal region parameters are created and removed only by its callers. But we may achieve better memory use if we can give such a region a later creation or an earlier removal, or both, inside the procedure. To reason about this, for a procedure p, we define:

- $bornR(p)$ is the set of region parameters that are bound to regions which are created inside p, i.e., by p or by one of the procedures it calls.
- $deadR(p)$ is the set of region parameters that are bound to regions which are removed inside p.

When analysing a procedure p, initially, $bornR(p) = outputR(p) \setminus inputR(p)$; $deadR(p) = inputR(p) \setminus outputR(p)$, where $inputR(p)$ and $outputR(p)$ are the sets of region variables reachable from input and output variables, respectively.

The analysis then follows each execution path of p and applies the rules in Fig. 6 to any call q to update the $deadR$ and $bornR$ sets of q. A region variable is eliminated from $deadR(q)$ if it needs to be live after the call to q in p (i.e., the region to which it is bound must not be removed in the call to q) (rule L1); or if the region will be removed more than once by q because of the so-called "region alias", which means the same actual region parameter is used for more than one formal region parameter (rule L2). A region variable is excluded from $bornR(q)$ if the region it is bound to is already live before the call to q (rule L3); or if q will create the region more than once due to region alias (rule L4). When there is a change to those sets of q, q needs to be analysed to propagate the change to its called procedures. Therefore, this analysis requires a fixpoint computation.

$$\frac{\begin{array}{c} r \in LR_{before}(pp(l)) \\ r \in LR_{after}(pp(l)) \end{array} \quad r = \alpha(r') \quad r' \in deadR(q)}{deadR(q) = deadR(q) \setminus \{r'\}} \text{ (L1)}$$

$$\frac{\alpha(r') = r \quad \alpha(r'') = r \quad r' \neq r'' \quad r' \in deadR(q)}{deadR(q) = deadR(q) \setminus \{r'\}} \text{ (L2)}$$

$$\frac{\begin{array}{c} r \in LR_{before}(pp(l)) \\ r = \alpha(r') \end{array} \quad r' \in bornR(q)}{bornR(q) = bornR(q) \setminus \{r'\}} \text{ (L3)}$$

$$\frac{\alpha(r') = r \quad \alpha(r'') = r \quad r' \neq r'' \quad r' \in bornR(q)}{bornR(q) = bornR(q) \setminus \{r'\}} \text{ (L4)}$$

$pp(l)$ returns the program point of the literal l.

Fig. 6. Live region analysis rules

After this analysis, the set of region variables of a procedure, N, is partitioned into four sets: $deadR$, $bornR$, $constantR$, and $localR$, where $localR = N \setminus (inputR \cup outputR)$ and $constantR = N \setminus (deadR \cup bornR \cup localR)$. The first three sets are constituents of the set of region parameters of the procedure. The region parameters in $deadR$ are live before a call to the procedure and the regions they are bound to can safely be removed inside the procedure. Those in $bornR$ are

unbound before a call to the procedure and will get bound inside the procedure. Those in *constantR* escape the procedure scope. The last set, *localR*, contains region variables that are absolutely internal to the procedure.

The algorithm to detect live region variables at each program point is an extension of live variable analysis, which is a standard, well-known program analysis [11]. The analysis in Section 4.2 aims to compute a shortest possible lifetime for a region. Its termination can intuitively be understood from the fact that each procedure uses a finite set of region variables, hence, initially *bornR* and *deadR* sets are also finite, and the analysis just reduces their size.

In the *qsort* program, *split* has three execution paths: $\langle(1),(2),(3)\rangle$, $\langle(4),(5),(6),(7)\rangle$, and $\langle(4),(8),(9)\rangle$; *qsort* has two: $\langle(1),(2)\rangle$ and $\langle(3),(4),(5),(6),(7)\rangle$. The LV and LR sets of *split* are in Table 2(a), of *qsort* in Table 2(b) (see also Fig. 2 and Fig. 3). Note that, in this example, it happens to occur that the set after one program point is always equal to the one before the next point in the same execution path. In general, this is not necessarily the case, consider for example a split point, the set of live region variables after it is the union of all the sets before its next points. The region parameter sets of the two procedures are: $deadR(split) = \{R1\}$, $bornR(split) = \{R3, R4\}$, $constantR(split) = \{R2, R5\}$; $deadR(qsort) = \{R1\}$, $bornR(qsort) = \phi$, $constantR(qsort) = \{R2, R3\}$.

Table 1. Live variable and live region variable sets in *qsort* program

pp	*LV*	*LR*
(1_b)	$\{X, L\}$	$\{R5, R1, R2\}$
$(1_a, 2_b)$	$\{\}$	$\{\}$
$(2_a, 3_b)$	$\{L1\}$	$\{R3, R2\}$
(3_a)	$\{L1, L2\}$	$\{R3, R2, R4\}$
(4_b)	$\{X, L\}$	$\{R5, R1, R2\}$
$(4_a, 5_b)$	$\{X, Le, Ls\}$	$\{R5, R2, R1\}$
$(5_a, 6_b)$	$\{X, Le, Ls\}$	$\{R5, R2, R1\}$
$(6_a, 7_b)$	$\{L2, Le, L11\}$	$\{R4, R2, R3\}$
(7_a)	$\{L1, L2\}$	$\{R3, R2, R4\}$
$(5_a, 8_b)$	$\{X, Le, Ls\}$	$\{R5, R2, R1\}$
$(8_a, 9_b)$	$\{L1, Le, L21\}$	$\{R4, R2, R3\}$
(9_a)	$\{L1, L2\}$	$\{R3, R2, R4\}$

(a) *split*

pp	*LV*	*LR*
(1_b)	$\{L, A\}$	$\{R1, R2, R3\}$
$(1_a, 2_b)$	$\{A\}$	$\{R3, R2\}$
(2_a)	$\{S\}$	$\{R3, R2\}$
(3_b)	$\{L, A\}$	$\{R1, R2, R3\}$
$(3_a, 4_b)$	$\{A, Le, Ls\}$	$\{R3, R2, R1\}$
$(4_a, 5_b)$	$\{A, Le, L1, L2\}$	$\{R3, R2, R4, R5\}$
$(5_a, 6_b)$	$\{Le, L1, S2\}$	$\{R2, R4, R3\}$
$(6_a, 7_b)$	$\{L1, A1\}$	$\{R4, R2, R3\}$
(7_a)	$\{S\}$	$\{R3, R2\}$

(b) *qsort*

5 Program Transformation

The main task of program transformation is to annotate the input program with *create* and *remove* instructions based on the region liveness information. It also annotates a procedure definition with formal region parameters known from the live region analysis. Actual region parameters at a call site can be derived from the formal region parameters of the called procedure and the α mapping at the call site.

Correctness conditions. To be correct, the transformation of a procedure p must ensure the following:

1. If a region variable is live at a program point, it must be bound to a region before that point.
2. If a region variable is not live at a program point, its binding status must be unbound before that point.
3. The regions to which the region parameters in $deadR(p)$ are bound will be removed inside p.
4. The regions to which those in $bornR(p)$ are bound will be created inside p.
5. No creation and removal will occur to those in $constantR(p)$ in the procedure.

Transformation. Each procedure is transformed by following its execution paths and applying the transformation rules in Fig. 7 to each program point so that the correctness conditions are respected. Assume that we are analysing a procedure p. Let l_i be the associated literal at a program point i in p. A literal can be either a specialized unification denoted by *unif* or a call (user-defined or built-ins). We assume that all the specialized unifications as well as calls to builtin operations do not remove or create any regions.

$$\frac{\begin{array}{c} l_i \equiv q(\ldots) \\ r \in LR_{after}(i) \setminus LR_{before}(i) \\ r \in (localR(p) \cup bornR(p)) \\ \nexists r' : r = \alpha(r') \wedge r' \notin bornR(q) \end{array}}{add\ \text{“}create\ r\text{”}\ before\ l_i} \quad \text{(T1)} \qquad \frac{\begin{array}{c} l_i \equiv X <= f(\ldots) \\ r \in Reach(X) \setminus LR_{before}(i) \qquad r \in LR_{after}(i) \\ r \in localR(p) \cup bornR(p) \end{array}}{add\ \text{“}create\ r\text{”}\ before\ l_i} \quad \text{(T2)}$$

$$\frac{\begin{array}{c} l_i \equiv q(\ldots) \\ r \in LR_{before}(i) \setminus LR_{after}(i) \\ r \in localR(p) \cup deadR(p) \cup bornR(p) \\ \nexists r' : r = \alpha(r') \wedge r' \notin deadR(q) \end{array}}{add\ \text{“}remove\ r\text{”}\ after\ l_i} \quad \text{(T3)} \qquad \frac{\begin{array}{c} l_i \equiv unif \\ r \in LR_{before}(i) \setminus LR_{after}(i) \\ r \in localR(p) \cup deadR(p) \cup bornR(p) \end{array}}{add\ \text{“}remove\ r\text{”}\ after\ l_i} \quad \text{(T4)}$$

$$\frac{\begin{array}{c} l_j\ is\ right\ after\ l_i\ in\ an\ execution\ path \\ r \in LR_{after}(i) \setminus LR_{before}(j) \\ r \in localR(p) \cup deadR(p) \cup bornR(p) \end{array}}{add\ \text{“}remove\ r\text{”}\ before\ l_j} \quad \text{(T5)}$$

Fig. 7. Transformation rules

When a region variable first becomes live, namely when it is not live before i but is live after i, a region must be created and the region variable is bound to the region. If the region is created inside l_i, then no annotation is added at i. Otherwise the region is created either by a caller of p or by p itself. The former means that the region should not be created again in p, hence no annotation is added at i. The latter occurs when the region variable belongs to either $bornR(p)$ or $localR(p)$, and in this case we add a *create* instruction before l_i. This is reflected by the transformation rules T1 and T2.

When a region variable ceases to be live, the region it is currently bound to is removed. The first case is when the region variable is live before i but not live after i. If p does not remove the region, it is removed by a caller of p and no annotation is introduced at i. Otherwise, the region is removed inside p. This means the region variable is in one of the $deadR$, $localR$, or $bornR$ sets of p. There are two subcases: if l_i removes the region, then no *remove* instruction

needs to be inserted at i; otherwise if p removes the region itself, we insert a *remove* instruction after l_i. The transformation rules T3 and T4 ensure this effect. While the reason for removing a region to which a region variable that belongs to the first two sets is bound is straightforward, the removal of a region that is bound to by a region variable in $bornR(p)$ is allowed because it is acceptable for p to remove the region after i and re-create it later on. The fact that region variable is in $bornR(p)$ ensures this. The second case is when the region variable is live after i, but not live before some program point j following i in a certain execution path. This can happen when i is a shared point among different execution paths and the region variable is live after i due to an execution path to which j does not belong. A *remove* instruction is added before l_j to remove the region as expressed by the transformation rule T5.

The result of the program transformation of the *qsort* program has been shown in Fig. 2. The addition of the *remove* instructions after the first program points in both *qsort* and *split* procedures results from the application of T4. Two *create* instructions inserted in the *split* procedure are effects of T2.

6 Support for Mercury Programs with Backtracking

The region analysis and transformation presented in Sections 3, 4, and 5 are correct for Mercury programs without backtracking. The region liveness analysis only takes into account forward execution and the transformation assumes that at runtime a program will follow only one execution path. To support backtracking the authors in [5, 7] described an enhanced runtime for Prolog with RBMM. The idea is to make backtracking transparent to the algorithm for non-backtracking programs by using a mechanism to undo changes to the region-based heap memory, restoring it to the previous state at the point to which the program backtracks. We reused this idea, developed the necessary details (which can be found in [13]) in the context of the Melbourne implementation of Mercury [14]. With this support, the algorithm for non-backtracking programs can be used unchanged to correctly support ones with backtracking.

7 Prototype Implementation and Experimental Results

We implemented an RBMM prototype using MMC version 0.12.0. It consists of an analyser that uses the region analysis to generate region-annotated programs and a *region simulator*. The analyser operates as one of the last analyses on the High Level Data Structure representation of the original source code and produces source code with region support. To have a working Mercury system with RBMM the runtime system of Mercury would need to be extended with support for regions, which is out of the scope of this paper. Instead, we developed a region simulator to experiment with region-annotated programs. The simulator can mimic the region operations in a region-annotated Mercury program as if the program were being executed by a working RBMM system. The simulator

provides an interface with non-logical methods for region creation, removal, allocation into a region, and for supporting backtracking (see [13] for more details).

The program transformation emits a program (as valid Mercury code) in which *create* instructions are replaced by calls to the method for region creation in the simulator's interface and *remove* instructions by calls to the region removal method. After each construction a call to the region allocation method is added to collect the number of words allocated. Procedure definitions and calls are extended with region parameters as extra parameters. Calls to backtrack-supporting methods are inserted at suitable places. We also augment the Mercury runtime system to interact with the simulator so that the saving and restoration of region-based memory states can be imitated correctly.

Primitive types are not dealt with specially by the analyser, but they are by the simulator. In particular, when dealing with a list of integers, the integers themselves are not put in a separate region but stored in the first words of the cons cells needed for the list skeleton.

When a Mercury program is executed it puts the terms that are created during execution on the heap. Assume that no RTGC is used, during forward execution the heap grows and only on backtracking instant reclaiming is done. With RBMM the terms will be put into regions and the regions can be freed (and reused) during the forward run of the program. In particular, it should be the case that temporary data is in regions which are freed as soon as the data is no longer needed.

In our experiments, we have included some deterministic programs that use some temporary data: **nrev** reverses a list of 5000 integers, **qsort** sorts a list of 100000 integers and **primes** finds all the primes less than 100; **dnamatch** and **life** are known to be difficult cases for region analysers. Two non-deterministic programs are used: 9-**queens** program that first generates a permutation and then checks, and **crypt** that finds the unique answer to a cryptoarithmetic puzzle[1]. The experiments allow us to measure the memory used by the benchmarks.

Table 2 shows the experimental results obtained by the region simulator for the annotated versions of the benchmarks. The experiments were done on a PC with a 2.8 GHz Pentium 4 CPU, 512 MB of RAM running Debian GNU/Linux 3.1, under a usual load.

Table 2. Experimental results

	nrev	**qsort**	**prime**	**dnamatch**	**rdnamatch**	**life**	**rlife**	**crypt**	**queens**
TR	5,002	200,002	29	2,082,005	2,083,005	50,303	50,403	418	7,689
MR	2	21	3	8	9	102	102	4	4
TW	25,015,000	5,865,744	916	18,537,685	18,541,685	894,336	894,336	3,442	159,234
MW	10,000	200,000	194	4,201,700	113,792	8,208	1,068	62	78
SLR	10,000	200,000	194	4,096,000	64,000	6,486	390	32	60
S (%)	99.96	96.59	78.82	77.33	99.39	99.08	99.88	98.20	99.95
AT (ms)	8	11	9	72	78	107	n/a	61	13
CT (ms)	251	243	223	329	343	361	n/a	307	273

[1] The original and annotated source code of the benchmark programs can be found at http://www.cs.kuleuven.be/∼quan/benchmarks.tar.gz

Our measurements reported in Table 2 are: the total number of regions created during the execution of a benchmark (TR), the maximum number of regions coexisting during its run (MR), the total number of words allocated (TW), the maximum number of words that coexist (MW), and the size in words of the largest region (SLR). Then savings (S) can be computed as (TW-MW)/TW. We also give the times in milliseconds taken by our analysis (AT) and by the usual Mercury compilation of the benchmark (CT). For all benchmarks the analysis time is only a fraction of the compilation time.

The fact that TR is much larger than MR confirms that the data used by a program can indeed be divided into regions that can be freed timely during its execution. If Mercury runs the programs without memory management, it will actually need TW words. When using RBMM only MW are needed. The savings are impressive, about 92% on average. The large savings can partly be explained by the fact that the analysis did a good job of partitioning the heap into regions such that temporary data can be freed in a timely manner. Optimal memory management is achieved for **nrev**, **qsort** and **prime**, as the programs use no more memory than what is needed to store their input data. In a standard LP system, all memory management in **crypt** and **queens** will be handled well by *instant reclaiming* because backtracking happens very frequently. The savings with RBMM in these two benchmarks are with respect to the situation where instant reclaiming is not used therefore seem to be unfair. However, they show that the runtime support for backtracking can also provide the ability of instant reclaiming in the context of RBMM.

In **dnamatch** and **life**, while the savings seem acceptable, the memory management is actually suboptimal. In these programs, there is a region that exists for almost the whole runtime and contains all the temporary and final values of the computation of the programs' output, which make up a significant part of the maximum number of words used. This undesirable performance is due to the fact that the present algorithm fails to split temporary values from the final outputs in these programs. A well-known solution to this problem is to make such programs *region-friendly* by rewriting after studying their RBMM-related behaviour [16, 1, 4, 7]. **rdnamatch** is a region-friendly version of **dnamatch**, achieved by adding an extra predicate to copy the temporary values into regions different from the region of the final output. An orthogonal solution is to enhance the static analysis: **rlife**, a manually adapted version of the region-annotated **life**, illustrates the effect of such a possible improvement. Their data show a large reduction in the maximum number of words needed. The latter solution is more preferable because it is entirely automated, hence freeing logic programmers from caring about memory management tasks. Moreover, the first solution requires extra time and memory for copying, while the second solution requires the same total number of words as before but the region analysis can distribute them more cleverly. We are working further in this direction.

Three of our benchmarks, **nrev**, **qsort**, and **dnamatch**, were also reported in [7] with the same inputs. We achieve optimal memory use in **nrev** and **qsort**, while in [7] the former was reported using maximally double and the latter 1.66

times the memory size of the input list. For **dnamatch**, we gain a saving of 77.33% compared to 33.5% shown in [7]. The reason for the better results is probably that our algorithm can remove a region earlier. For example, in the *qsort* procedure of the *qsort* program, the inference in [7] removes the region of the input list after the call to *split* (so the list and the two sublists are live at that point). In our case, that removal happens at the base case in the call to *split*, before creating the two sublists.

8 Conclusion

In this paper we have developed a region analysis algorithm for the typed logic programming language Mercury. Our approach was inspired by the work in [2]. The analyses in [2] take into account the data flow in a program in order to determine the regions and their lifetimes. Therefore the analyses had to be redefined for Mercury to deal with unification and a control flow which are fundamentally different from object manipulation and control flow in Java. Type information in Mercury has been exploited by the region analysis to achieve a finite region representation for recursive structures. Apart from that our algorithm allows interprocedural creation of regions which was not handled in [2]. The analyser annotates Mercury programs with instructions for RBMM. Experiments with the analyser for several small benchmarks (4-17 procedures, 70-500 lines of code) show that it takes a reasonable amount of time. We also implemented a region simulator which can run the annotated programs produced by the region analyser to imitate the effects of executing Mercury programs in an RBMM system. The memory use results of the benchmarks are positive, in some programs we obtain optimal memory consumption. This indicates that our method could be an interesting alternative memory management technique. It should be noted that the experiments have been done only with small programs and that the lack of the completely implemented runtime supporting for RBMM makes us unable to measure runtime performance and the internal cost of the region allocator and to provide a thorough comparison between our approach and other memory management techniques available in Mercury, such as RTGC and CTGC. We are working on integrating the presented algorithm into the MMC and hope to lift this limitation soon.

We have not provided a formal proof for the whole algorithm and left it as future work. However, the foundations for the correctness of the algorithm have been laid intuitively. Finally, we also would like to investigate some extensions to the presented algorithm: better region partition as mentioned in Section 7 and modular region analysis.

Acknowledgements

We would like to thank Zoltan Somogyi for his helpful comments on various parts of this work.

References

[1] Aiken, A., Fähndrich, M., Levien, R.: Better static memory management: Improving region-based analysis of higher-order languages. In: Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation, pp. 174–185. ACM Press, New York (1995)
[2] Cherem, S., Rugina, R.: Region analysis and transformation for Java programs. In: Proceedings of the 4th International Symposium on Memory Management, pp. 85–96. ACM Press, New York (2004)
[3] Grossman, D., Morrisett, G., Jim, T., Hicks, M., Wang, Y., Cheney, J.: Region-based memory management in Cyclone. In: Proceedings of the ACM Conference on Programming Language Design and Implementation, pp. 282–293. ACM Press, New York (2002)
[4] Henglein, F., Makholm, H., Niss, H.: A direct approach to control-flow sensitive region-based memory management. In: Principles and Practice of Declarative Programming, pp. 175–186. ACM Press, New York (2001)
[5] Makholm, H.: A region-based memory manager for Prolog. In: Proceedings of the 2nd International Symposium on Memory Management, pp. 25–34. ACM Press, New York (2000)
[6] Makholm, H.: Region-based memory management in Prolog. Master's thesis, University of Copenhagen (2000)
[7] Makholm, H., Sagonas, K.: On enabling the WAM with region support. In: Stuckey, P.J. (ed.) ICLP 2002. LNCS, vol. 2401, Springer, Heidelberg (2002)
[8] Mazur, N.: Compile-time garbage collection for the declarative language Mercury. PhD thesis, Department of Computer Science, Katholieke Universiteit Leuven (May 2004)
[9] Mazur, N., Janssens, G., Bruynooghe, M.: A module based analysis for memory reuse in Mercury. In: Palamidessi, C., Moniz Pereira, L., Lloyd, J.W., Dahl, V., Furbach, U., Kerber, M., Lau, K.-K., Sagiv, Y., Stuckey, P.J. (eds.) CL 2000. LNCS(LNAI), vol. 1861, pp. 1255–1269. Springer, Heidelberg (2000)
[10] Mazur, N., Ross, P., Janssens, G., Bruynooghe, M.: Practical aspects for a working compile time garbage collection system for Mercury. In: Codognet, P. (ed.) ICLP 2001. LNCS, vol. 2237, pp. 105–119. Springer, Heidelberg (2001)
[11] Nielson, F., Nielson, H.R., Hankin, C.: The Principles of Program Analysis. Springer, Heidelberg (1999)
[12] Phan, Q., Janssens, G.: Towards region-based memory management for Mercury programs. In: CICLOPS (2006)
[13] Phan, Q., Janssens, G.: A proposal for runtime region support for Mercury programs. Technical Report CW482, Department of Computer Science, Katholieke Universiteit Leuven (2007)
[14] Somogyi, Z., Henderson, F., Conway, T.: The execution algorithm of Mercury, an efficient purely declarative logic programming language. The Journal of Logic Programming 29(1-3), 17–64 (1996)
[15] Tofte, M., Birkedal, L., Elsman, M., Hallenberg, N.: A retrospective on region-based memory management. Higher-Order and Symbolic Computation 17, 245–265 (2004)
[16] Tofte, M., Talpin, J.-P.: Region-based memory management. Information and Computation. 132(2), 109–176 (1997)

Automatic Binding-Related Error Diagnosis in Logic Programs

Paweł Pietrzak[1] and Manuel V. Hermenegildo[1,2]

[1] School of Computer Science, Technical University of Madrid (UPM)
[2] Depts. of CS and ECE, University of New Mexico

Abstract. This paper proposes a diagnosis algorithm for locating a certain kind of errors in logic programs: variable binding errors that result in abstract symptoms during compile-time checking of assertions based on abstract interpretation. The diagnoser analyzes the graph generated by the abstract interpreter, which is a provably safe approximation of the program semantics. The proposed algorithm traverses this graph to find the point where the actual error originates (a reason of the symptom), leading to the point the error has been reported (the symptom). The procedure is fully automatic, not requiring any interaction with the user. A prototype diagnoser has been implemented and preliminary results are encouraging.

1 Introduction

Obtaining a program that satisfies the programmer's intentions is clearly a crucial objective in software development. If the program does not conform to the user's expectations (i.e., if it contains a discrepancy between the program semantics and the specification –a *symptom*) this means that somewhere in the program there is an error which has to be found and corrected. The difficulty in this process comes from the fact that the effects of a given error (the symptoms) propagate from one location in the code to another and manifest themselves far away from the place where the error resides. The process of locating (a piece of code that contains) an error given some symptom is called *diagnosis*. In this paper we address the problem of compile-time automatic diagnosis in untyped (constraint) logic programs, focusing on binding errors, meaning that we locate a variable binding that eventually produces a symptom (making the program erroneous). We aim at designing sound foundations for a practical and useful diagnosis tool that can be used routinely.

Before an error can be diagnosed, its presence has to be detected through a symptom. Our approach relies on the idea of compile-time program verification and error detection based on abstract interpretation [7], as proposed in [13,22,14]. Properties expected by the user concern call and success patterns of program predicates and are given in the form of *assertions* [11,19,21]. An important characteristic of the approach used is that only a small number of assertions may be present in the program, or even no assertions at all. In the latter case the system takes advantage of assertions written for built-in and library predicates to

V. Dahl and I. Niemelä (Eds.): ICLP 2007, LNCS 4670, pp. 333–347, 2007.

detect errors in user programs. Also, the approach is parametric on the abstract domains used, so that a variety of properties can be proved or disproved, based on the set of abstract domains used.

Assertion verification is preceded by static program analysis based on abstract interpretation [7,3,20,15]. The results of the analysis are compared against the assertions. An assertion that can be shown to be false, together with the related program point is called a *symptom*. Such (abstract) symptoms are the starting point of our diagnosis procedure. The static analyzer produces a *program analysis graph* which is essentially a finite representation of all execution paths that may appear at run-time, annotated with the state at the call and exit points of procedures and at each program point. This graph is the fundamental data structure exploited by the diagnoser. The diagnoser traverses the graph from the point of the symptom, against the direction of execution, trying to identify a point (or points) where variable bindings occur which are responsible for the symptom. During the traversal, the abstract operations of the analyzer are executed in order to analyze parts of the graph, and thus come to conclusions regarding corresponding pieces of the code. The proposed procedure is fully automatic, and does not require any user intervention. In the implementation, the initial call to the diagnoser is (optionally) automatically triggered by the assertion checker when a discrepancy with an assertion is detected.

2 Related Work

Locating errors is an inherent part of debugging[1] and has attracted significant attention. One of the best-known diagnosis techniques is *declarative* or *algorithmic debugging*, initially proposed in [24]. In this approach the search for the error takes the form of an interactive session with the user, who is required to answer queries about the intended behavior of the program. A drawback of the approach is that the number of questions posed to the user is typically very large. One way to reduce the number of queries and to simplify them is to add partial formal specifications to the program in the form of assertions [11], but the load on the user remains a problem for the practical takeup of this technique. The algorithmic debugging approach is strongly tied to the declarative semantics while our aim is to develop an approach that works also for impure (constraint) logic programs. An additional difference with our approach is that the declarative debugging session concerns the concrete semantics and a single (test) execution only, whereas we are interested in diagnosing errors at compile-time, and for all possible executions.

When the full (abstract) specification of the program is available the method known as *abstract diagnosis* [6] can be applied. This method however requires, in addition to the full specification, again adherence to the declarative semantics (and also makes the assumption that the specification can be linked with the concrete semantics via a *Galois connection*). Other techniques based on applying

[1] We prefer to reserve the term "debugging" for a process that involves both locating and removing the bug.

a verification condition, such as, e.g., [10], can also be used to locate errors. In [10], in particular, descriptive types are used to approximate the operational semantics. A clause on which the inductive proof fails indicates an error.

In the context of strongly-typed languages, the problem of locating type errors, i.e., understanding why an expression cannot be typed, has received much attention. These diagnosis algorithms try to find a reason for the failure in type unification during type inference. The problem was initially attacked in [26] where the steps of the type inference procedure are recorded and later looked up for inconsistencies. Many researchers (e.g., [2]) have followed this line and proposed various improvements. In our case we are dealing with an untyped (logic) language, and with a general class of properties that goes beyond traditional types. Also, as in the typed languages, the error might be placed far away from the expression reported in the type error message. But because in our case there may be only a few assertions in the program, the error may in fact be propagated further and show up much later, even in different functions or modules, and in a way that does not correspond intuitively with the direction of the execution-time data flow.

Our approach has a strong relationship with *slicing* (see e.g.,[12,23] for slicing in logic programming). Note that in (backward) slicing the goal is to find a piece of program that potentially affects a value of a variable at the point of interest, whatever the value is, whereas we are interested only in values violating the specification. Also, unlike in slicing we do not track dependencies between individual variables, letting an abstract domain and the generic abstract interpreter capture the necessary information.

3 Preliminaries and Notation

We assume that the reader is familiar with logic programming (see, e.g., [1,18]) and abstract interpretation [7]. We will use the standard notions of SLD resolution and SLD derivation with the Prolog computation rule. We use a standard notion of substitution, i.e. mappings from program variables to terms. A substitution will be typically denoted as θ, possibly with sub- or superscripts. We also use $\theta_{|A}$ to denote a projection of θ over variables in an atom A. Let G_k be a *resolvent* of the form $\leftarrow (A_1, \ldots, A_n)\theta_0 \cdots \theta_k$, obtained in the k-th step of the derivation. In step $k+1$ we obtain the resolvent G_{k+1} (denoted $G_k \rightsquigarrow G_{k+1}$) of the form $\leftarrow (B_1, \ldots, B_m, A_2, \ldots, A_n)\theta_0 \cdots \theta_k\theta_{k+1}$ where $B_0 \leftarrow B_1, \ldots, B_m$ is a renamed clause of the program and $\theta_{k+1} = mgu(A_1\theta_0 \cdots \theta_k, B_0)$. Let $\overset{+}{\rightsquigarrow}$ denote a transitive closure of $\rightsquigarrow$. In order to handle program points we annotate every atom A in a derivation by a program-point identifier $\textcircled{p}$, which determines a clause and a position in the clause where A comes from. We write annotated atoms as $A^{\textcircled{p}}$. We say that a program point $\textcircled{p}$ corresponds to a derivation state G iff G is of the form $\leftarrow (A^{\textcircled{p}}, \ldots)\theta$.

Goal-directed abstract interpretation is a technique whose aim is for a given initial call pattern (describing a possibly infinite set of input data), to generate annotations describing (in an abstract way) all possible run-time variable bindings. The annotations are expressed in an *abstract domain* D_α (a lattice

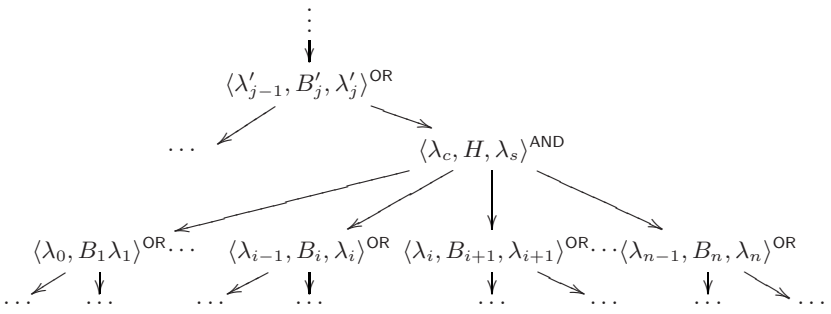


Fig. 1. A fragment of an abstract AND-OR graph

equipped with a partial order $\sqsubseteq$, and standard elements $\top$ and $\bot$, and operations $\sqcup$ and $\sqcap$). In the case of logic programming, the annotations typically take the form of *abstract substitutions* (approximations of concrete substitutions), i.e., mappings from program variables to values in D_α. Abstract domain D_α and concrete domain D are linked to each other by two monotone mappings $\alpha : D \mapsto D_\alpha$ and $\gamma : D_\alpha \mapsto D$, called *abstraction* and *concretization* functions respectively. We do not restrict our attention to any specific abstract domain. However, in the examples we will use for concreteness regular types [9,25].

Throughout this paper we will use the abstract interpretation framework of Bruynooghe [3], which, with variations and optimizations (in our case [20,15]), is the basis of a large portion of the practical analyzers for logic programs. In these frameworks the analyzer produces a program analysis graph called an *abstract* AND-OR *graph*. The abstract AND-OR graph is a finite description of the set of (concrete) AND-OR trees that are conceptually traversed during execution of the program. The abstract graph (see Figure 1) has two sorts of nodes: AND-nodes containing (copies of) heads of clauses, which have body atoms as their children[2] and OR-nodes which are body atoms representing calls. Each node (whether AND- or OR-) is adorned with two substitutions, describing the bindings of variables just before entering and just after exiting the respective piece of program code.

Let $vars(E)$ denote a set of variables that occur in a syntactic object E. We use the notation $\langle \lambda_c, H, \lambda_s \rangle^{\mathsf{AND}}$ to denote an AND-node, where H is the head of a clause, say $H \leftarrow B_1, \ldots, B_n$. λ_c is an abstract substitution that describes the bindings of variables $vars(H)$ before entering the clause, and λ_s keeps the bindings after exiting the clause. The AND-node with the atom H has children, each being an OR-node, written as $\langle \lambda_{i-1}, B_i, \lambda_i \rangle^{\mathsf{OR}}$, for $1 \leq i \leq n$. A substitution λ_i describes bindings of variables just before calling an atom B_{i+1} (or after succeeding in the clause body if $i = n$), and it ranges over all $vars((H \leftarrow B_1, \ldots, B_n))$. We use a function $children(N)$ to denote an ordered set of children of an AND- or OR-node N. We assume that all substitutions in the graph are different from $\bot$, i.e., branches corresponding to sub-computations known to fail

[2] Clearly, children of nodes have to be ordered. Moreover, there might be more than one AND-node per clause.

are not present in the graph. Variables in the program and those in the abstract AND-OR graph are renamed apart. During the analysis phase, when the abstract AND-OR graph is constructed the following abstract operations (functions) are used (following [15]):

- $\mathsf{Aproj}(\lambda, V)$ performs the projection of a substitution λ on a set of variables V.
- $\mathsf{Aextend}(\lambda, V)$ extends the substitution λ to the set of variables V.
- $\mathsf{Aunif}(E_1, E_2, \lambda)$ performs abstract unification of two expressions $E_1 = E_2$ and conjoins the results with λ.
- $\mathsf{Aconj}(\lambda_1, \lambda_2)$ performs the abstract conjunction of two substitutions.
- $\mathsf{Alub}(\lambda_1, \lambda_2)$ performs the abstract disjunction of two substitutions[3].

We recall the notion of *topmost abstract substitution* over the set of variables V (introduced in [5]), which captures the abstract representation of the most general concrete substitution, and which, e.g., for our example type domain can be defined as $\lambda_V^\top \stackrel{def}{=} \{X/\top \mid X \in V\}$. V will be dropped if it is clear from the context.

4 Assertions, Symptoms, and Errors

Our objective is to find a reason for a symptom, where the symptom is understood as a deviation of the program behavior from the user expectations. The expectations have to be expressed as a formal specification. Such specification is commonly written in terms of assertions (e.g., see [11,19,21]). A specific assertion language definition is not required for our results (however, the implementation and experiments are carried out in the Ciao Preprocessor [14] using its assertion language [21]). For simplicity we assume that a (partial) specification is provided in terms of abstract substitutions assigned to program points. We will refer to such substitutions as *program-point assertions* or *expected properties*, interchangeably. We explicitly allow the specification to be partial, i.e., the program can contain just a few or even no assertions, the only assertions then checked being those in libraries. A program-point assertion λ_{Prop} between atoms B_i and B_{i+1} expresses an expected success pattern of B_i or expected call pattern of B_{i+1}. Consequently, λ_{Prop} ranges over $vars(B_i)$ or $vars(B_{i+1})$. Note that we have chosen program point assertions without loss of generality since predicate-level assertions (cf. [21]) can be translated to program point assertions with a simple program transformation. We also assume that the expected properties are renamed along with the clauses, so that they range over the same variables as the copies of the clauses present in the abstract graph. These assumptions simplify the subsequent presentation. Finally, we assume that abstract values are *over*-approximations (a dual approach applies for *under*-approximations).

Now we are in a position to define a notion of *symptom*. Assume that at a program point Ⓢ there is an associated assertion λ_{Prop}. A *symptom* of violating

[3] While we will not need this operation in the paper it is included in order to make the description complete and avoid confusion.

the assertion occurs whenever there is a derivation $D = G_0 \overset{+}{\rightsquigarrow} G_k \rightsquigarrow \cdots$ with a state $G_k = \leftarrow (A^{\textcircled{s}}, \ldots)\theta_0 \cdots \theta_k$ s.t. $(\theta_0 \cdots \theta_k)_{|A} \notin \gamma(\lambda_{Prop})$. In other words the assertion is expected to be satisfied for all variable bindings at ⓢ in any possible execution. We say that D is an *assertion violating derivation*. A symptom of violating the assertion can be signaled by compile-time checking whenever the static analysis produces an abstract substitution λ at ⓢ (for all concrete θ at ⓢ $\theta \in \gamma(\lambda)$), s.t. $\lambda \not\sqsubseteq \lambda_{Prop}$.

We are interested in finding the reason for the symptom. We say that a derivation state $G_l = \leftarrow (B^{\textcircled{e}}, \ldots)\theta_0 \cdots \theta_l$ $(l < k)$, and the corresponding program point ⓔ *contribute* to the symptom at ⓢ iff for any substitution θ there is a sub-derivation D' of an assertion violating derivation D which is of the form $\leftarrow (B^{\textcircled{e}}, \ldots)\theta \overset{+}{\rightsquigarrow} \leftarrow (A^{\textcircled{s}}, \ldots)\theta\theta'_{l+1} \cdots \theta'_k$ s.t. $(\theta\theta'_{l+1} \cdots \theta'_k)_{|A} \notin \gamma(\lambda_{Prop})$. D' differs from the appropriate part of D only in computed substitutions. We assume the same clauses are selected in corresponding steps. By replacing the input substitution $\theta_0 \cdots \theta_l$ by a universally quantified substitution θ we try to determine whether the source of violating the assertion lies between derivation states G_l and G_k or it is instead propagated from states preceding G_l along with the substitution $\theta_0 \cdots \theta_l$. Note that the initial state G_0 always contributes to the symptom. This however is not useful for locating bugs. We define a *binding error* as a derivation state (and the corresponding program point) for which the sub-derivation D' has the shortest possible length. A binding error indicates a program point where a variable binding takes place which eventually leads to the symptom. Note that the actual symptom might be due to some other (non-binding) error which we are not able to detect. Nonetheless, we provide a strong indication that the actual error should be searched between the binding error and the symptom.

Our objective in this paper is to locate binding errors statically. The starting point of the error diagnosis process is compile-time assertion checking based on the output of the abstract interpretation framework. Interestingly, although abstract interpretation in general provides only safe approximations of the properties, in practice it is often possible to definitely prove or disprove an assertion. The latter case occurs when we have $\lambda_a \sqcap \lambda_{Prop} = \bot$. In those cases we say that λ_a is *incompatible* with the expected property λ_{Prop} (we will use this notion later in our diagnosis algorithm) and we have a definite error. However in some cases the system will not be able to prove or disprove a given assertion ($\lambda_a \not\sqsubseteq \lambda_{Prop}$ and $\lambda_a \sqcap \lambda_{Prop} \neq \bot$). In this case the system allows the user to choose (via flags) whether this should be considered an error, a warning, or be ignored. Our practical experience (and we understand it is also that of for example the ASTREE developers [8]) is that these cases are often actual symptoms, even if sometimes they are not and the "false alarm" is simply due to loss of precision in the analysis. Herein we will consider such cases indeed as symptoms, and start diagnosis for them, and will accept that in some cases no real error will be responsible for them, in which case we will guarantee that the diagnosis procedure will never locate such a non-existing error and will simply report that no error could be found.

5 Traversing Abstract AND–OR Graphs

The core of our binding error searching procedure consists of traversing (parts of) an abstract AND-OR graph and performing abstract operations, resulting in abstract substitutions, in a similar way that they are performed during analysis. Therefore, we will need a notion of traversing an abstract AND-OR graph, and, at the same time, replacing existing abstract substitutions with the newly generated ones.

In the following, an abstract AND-OR graph R with all substitutions replaced by $\lambda^\top$ will be denoted by $R^\top$.

Definition 1 (Forward traversal of an abstract AND-OR graph R. Defintion of transition $\Rightarrow_R$)

Let $\Rightarrow_R$ be a transition over nodes of an abstract AND-OR graph R:

(Entry) $\langle \lambda'_{j-1}, B'_j, \lambda'_j \rangle^{\mathsf{OR}} \Rightarrow_R \langle \lambda_c, H, \lambda_s \rangle^{\mathsf{AND}}$ *if* $H \in children(B'_j)$
$\lambda_{add} := \mathsf{Aunif}(B'_j, H, \lambda'_{j-1})$
$\lambda_c := \mathsf{Aproj}(\lambda_{add}, vars(H))$

(Enter Body) $\langle \lambda_c, H, \lambda_s \rangle^{\mathsf{AND}} \Rightarrow_R \langle \lambda_0, B_1, \lambda_1 \rangle^{\mathsf{OR}}$ *if* $B_1 \in children(H)$
$\lambda_0 := \mathsf{Aextend}(\lambda_c, vars(H \leftarrow B_1, \ldots, B_n))$

(Move Right) $\langle \lambda_{i-1}, B_i, \lambda_i \rangle^{\mathsf{OR}} \Rightarrow_R \langle \lambda_i, B_{i+1}, \lambda_{i+1} \rangle^{\mathsf{OR}}$ *if* $\exists H', \{B_i, B_{i+1}\} \subseteq children(H')$

(Exit Body) $\langle \lambda_{n-1}, B_n, \lambda_n \rangle^{\mathsf{OR}} \Rightarrow_R \langle \lambda_c, H, \lambda_s \rangle^{\mathsf{AND}}$, *if* $B_n \in children(B)$ *and* $|children(B)| = n$ *(i.e., B_n is the rightmost child of H)*
$\lambda_s := \mathsf{Aproj}(\lambda_n, vars(H))$

(Exit) [4] $\langle \lambda_c, H, \lambda_s \rangle^{\mathsf{AND}} \Rightarrow_R \langle \lambda'_{j-1}, B'_j, \lambda'_j \rangle^{\mathsf{OR}}$, *if* $H \in children(B'_j)$
$\lambda_{add} := \mathsf{Aunif}(H, B'_j, \lambda_s)$
$\lambda_{ext} := \mathsf{Aextend}(\lambda_{add}, vars(H' \leftarrow B'_1, \ldots, B'_{n'}))$
$\lambda'_j := \mathsf{Aconj}(\lambda_c, \lambda_{ext})$ □

We will omit the subscript R in $\Rightarrow_R$ if it is clear from the context. We extend the $\Rightarrow_R$ relation to a traversal over a finite sequence of nodes $s = [s_1, \ldots, s_n]$, which will be written as $\overset{s}{\Rightarrow}_R$, and defined as follows: $s_1 \overset{s}{\Rightarrow}_R s_n$ iff $\forall_{1 \leq i < n}.s_i \Rightarrow_R s_{i+1}$, The ":" operator will denote concatenation of node sequences. For a given s, $s^\top$ will denote a sequence of nodes identical to s but with all substitutions replaced by $\lambda^\top$.

The $\overset{s}{\Rightarrow}$ relation is a basis for our binding error searching algorithm. Note that $\overset{s}{\Rightarrow}$ mimics the basic operations performed by abstract interpretation, and therefore safely approximates the concrete semantics. In the following an abstract AND–OR graph is called *fresh* if it has been adorned directly by the abstract interpretation process, i.e., no node has been modified by $\overset{s}{\Rightarrow}$ afterward.

Lemma 1. *Let R be a fresh abstract AND-OR graph, resulting from analyzing a program P. Assume an OR-node $N = \langle \lambda_i, A, \lambda_{i+1} \rangle^{\mathsf{OR}}$ in R. Assume also a subderivation $D = \leftarrow (A, \ldots)\theta \overset{+}{\rightsquigarrow} \leftarrow (B, \ldots)\theta\theta'$ which occurs when executing P.*

[4] This operation differs from the corresponding one used in constructing the whole graph (cf. [3,20,15]), as we propagate the success substitution from one clause only.

(i) If $\theta_{|A} \in \gamma(\lambda_i)$ *then there is a sequence of nodes* s *s.t.* $N \overset{s}{\Rightarrow} \langle \lambda'_j, B, \lambda'_{j+1} \rangle^{\mathsf{OR}}$ *and* $\theta\theta'_{|B} \in \gamma(\lambda'_j)$.

(ii) Moreover, there is a corresponding OR*-node* $N^* = \langle \lambda^\top, A, \_ \rangle^{\mathsf{OR}}$ *and a sequence of nodes* $s^\top$ *in* $R^\top$ *s.t.* $N^* \overset{s^\top}{\Rightarrow} \langle \lambda^*_j, B, \_ \rangle^{\mathsf{OR}}$ *and* $\theta\theta'_{|B} \in \gamma(\lambda^*_j)$. *(Obviously, we have* $\lambda'_j \sqsubseteq \lambda^*_j$.*)*

We say that s and $s^\top$ *approximate* a sub-derivation D.

Corollary 1. *Take the assumptions of Lemma 1. Part (ii) holds for all* θ.

Since $\overset{s}{\Rightarrow}$ performs the same sequence of abstract operations as the entire abstract interpretation process but limited to one specific path in the abstract AND-OR graph, it is evident that every step of $\overset{s}{\Rightarrow}$ generates abstract substitutions that are not more general than those produced by the static analyzer. In other words, $\overset{s}{\Rightarrow}$ never loses precision with respect to the full analysis process.

Now we justify applying $\overset{s^\top}{\Rightarrow}$ to locate binding errors.

Proposition 1. *Let* $\ldots, B_i \,@\, B_{i+1}, \ldots$ *be a fragment of a clause body in the program* P *where* λ_{Prop} *is an expected property at point* @. *Let* R *denote a (fresh) abstract* AND-OR *graph obtained by the static analysis of* P. *Assume also that there exists in* $R^\top$ *a sequence of nodes* $s^\top$ *and an* OR*-node with atom* B', *s.t.* $\langle \lambda^\top, B', \lambda^\top \rangle^{\mathsf{OR}} \overset{s^\top}{\Rightarrow} \langle \lambda^*_i, B_{i+1}, \_ \rangle^{\mathsf{OR}}$

If $\lambda^*_i \sqcap \lambda_{Prop} = \bot$, *and if there exists a sub-derivation* D *reaching* @ *and approximated by* $s^\top$ *then* D *contains a symptom at* @. *Moreover a derivation state with* B' *as the leftmost atom and with the corresponding substitution is a binding error related to the symptom.*

Proof: Follows from Corollary 1. □

Notice that even though the $\overset{s}{\Rightarrow}$ relation approximates the concrete semantics it is finer grained in the sense that $\overset{s}{\Rightarrow}$ distinguishes steps taken as one in an SLD derivation. The steps are selecting atoms in a resolvent, entering and exiting clauses. In fact, a starting node of a $\overset{s}{\Rightarrow}$ traversal does not have to be an OR-node, it can be an AND-node as well. This gives us an opportunity to locate some errors more precisely than would be captured by the SLD resolution semantics.

6 An Example

In this section we explain informally, in terms of an example, how the $\overset{s}{\Rightarrow}$ transition is used to locate binding errors. The general idea is to traverse the abstract AND–OR graph starting from the symptom, and moving against the direction of execution, in DFS fashion. This makes it feasible to examine only those nodes which are involved in the (abstract) execution prior the symptom, and therefore only those which may potentially contain an error.

To illustrate the algorithm let us consider the following *slowsort* example, depicted in Figure 2. *slowsort* is a program that is meant to sort a list of numbers by first generating a permutation of the input list and then checking whether

the generated permutation is a sorted list. The predicate `perm/2` generates permutations by removing non-deterministically an element from the list (predicate `del/3`) and then calling itself recursively for the rest of the list. The test that checks if the list is sorted is performed by `sorted/2`. This code can be augmented with an `entry` declaration: `:- entry slowsort(A,B) : list(A,num).` which declares an intended initial call pattern to the top-level/exported predicate (cf. [21]). The `entry` declaration is used by the static analyzer as the starting point of the (top-down) analysis graph.

Observe that the head of the second clause of `perm/2` contains a binding error: the head should look like: `perm(L,[H|L1])`. The error results in a run-time exception ("illegal arithmetic expression") raised when the computation reaches the library predicate `=</2` in the third clause of `sorted/1`, since the second element `Y` of the input list is a list itself, rather than a number, as one would expect. In the Ciao system libraries (which subsume the classical notion of "built-ins") predicates are equipped with assertions specifying their expected call and success patterns. Therefore, the expected value of `Y` is known to the diagnoser (thanks to the modular nature of analysis) without any prior effort from the user. In fact, static assertion checking [22] is able to detect that the value of `Y` is of type $rt21$, defined by the following regular term grammar rules (see, e.g., [9]):

```
slowsort(L,S) :- perm(L,S), sorted(S).

perm([],[]).
% There is a bug here:
perm(L,[H,L1]) :- del(L,H,L2), perm(L2,L1).

del([H|L],H,L).
del([H|L],E,[H|L1]) :- del(L,E,L1).

sorted([]).
sorted([_]).
sorted([X,Y|L]):- X =< Y, sorted([Y|L]).
```

Fig. 2. An erroneous *slowsort* program

$$rt21 \rightarrow [\,]$$
$$rt21 \rightarrow [num, rt21].$$

with the meaning that a term of type $rt21$ is either an empty list or a two-element list, with the first element being a number and the second one being a term of type $rt21$. Therefore the value of `Y` is not compatible with the expected type *arithexpr* (arithmetic expression) which appears in the assertion for `=</2`. Let this point be the starting symptom for our diagnosis session.

A part of the abstract AND-OR graph R generated by the analyzer is depicted in Figure 3. Abstract substitutions have been left out for the sake of readability. The root OR-node ① corresponds to the `entry` declaration. The starting point of the diagnosis is the illegal call to `=</2` in the clause:

```
sorted([X,Y|L]):- X =< Y, sorted([Y|L]).
```

which corresponds to node ⑫. Observe that node ⑬ has no descendants, as the call to `X =< Y` never succeeds, and computations never reach the goal `sorted([Y|L])`. During the diagnosis process we build a sequence of visited

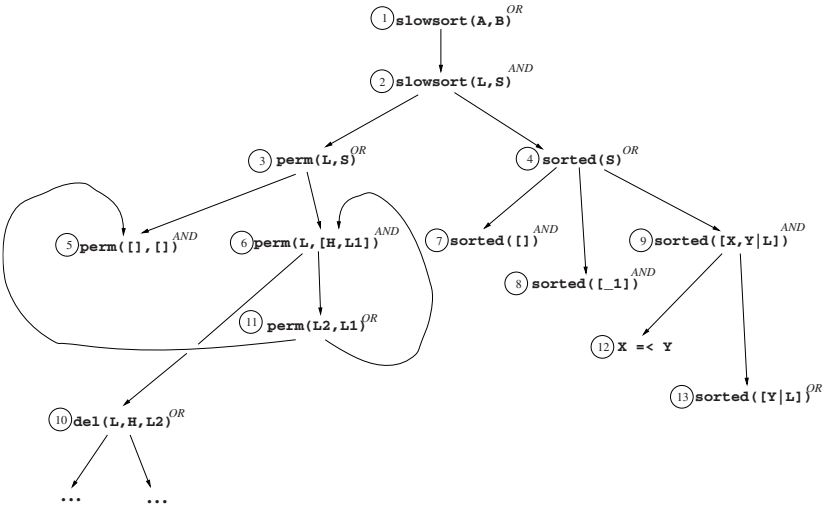


Fig. 3. A part of the abstract AND-OR graph R, an output of the analysis of the *slowsort* example from Figure 2

nodes, moving backwards, as discussed in Section 5. We will traverse the graph $R^{\top}$, starting with the AND-node ⑨, as ⑫ is its child. Let the constructed sequence of nodes be s. Initially s contains one node:

$$N_9 = \langle \{\mathtt{X}/\top, \mathtt{Y}/\top, \mathtt{L}/\top\}, \mathtt{sorted}([\mathtt{X}, \mathtt{Y}|\mathtt{L}]), \{\mathtt{X}/\top, \mathtt{Y}/\top, \mathtt{L}/\top\}\rangle^{\mathsf{AND}}$$

We keep the indexing of N's consistent with Figure 3. At this point we have reached the head of the clause, i.e., we have to find a calling atom, which is a parent OR-node ④. We add the node

$$N_4 = \langle \{\mathtt{S}/\top\}, \mathtt{sorted}(\mathtt{S}), \{\mathtt{S}/\top\}\rangle^{\mathsf{OR}}$$

to the sequence s. We need to determine if entering the clause, i.e., performing (abstract) unification when matching a calling atom and the head of clause, could cause the error symptom. To achieve this we run abstract operations corresponding to entering the clause from node ④ to ⑨ (i.e., we do the traversal $N_4 \stackrel{s}{\Rightarrow} N_9$ with application of rule **Entry** of Definition 1). As a result, variable `Y` is given value $\top$ in ⑫. This is not incompatible with *arithexpr* and we cannot conclude that the node ④ supports our symptom. Thus we keep on traversing the graph towards AND-node ②, and the left sibling of ④, i.e., to OR-node ③:

$$N_3 = \langle \{\mathtt{L}/\top, \mathtt{S}/\top\}, \mathtt{perm}(\mathtt{L}, \mathtt{S}), \{\mathtt{L}/\top, \mathtt{S}/\top\}\rangle^{\mathsf{OR}}$$

We now have $s = [N_3, N_4, N_9]$, which corresponds to succeeding in `perm(L,S)`. Nothing is assumed at this point about the values of `L` and `S`. The diagnoser checks whether erroneous bindings could be propagated through these variables from one of the clauses defining `perm/2`. In order to achieve this a child

(AND-node) of ③ is selected. Assume that ⑥ has been taken first. Now we have to check if the (abstract) unification performed when exiting the clause

```
perm(L,[H,L1]) :- del(L,H,L2), perm(L2,L1).
```

can cause the symptom. Traversal $N_6 \overset{s}{\Rightarrow} N_9$ is performed, using the appropriate rules **Exit, Move Right, Entry** (see Definition 1). As a result, variable `S` obtains a type of two-element list composed of $\top$. Thus, `Y` is given value $\top$ again and nothing can be concluded yet about an error causing the symptom. After performing two more steps we reach ⑥ again and we have $s = [N'_6, N_{11}, N_6, N_3, N_2, N_4, N_9]$, where each node contains related atoms with appropriate top-most abstract substitutions (we rename apart two copies of ⑥). After performing traversal $N'_6 \overset{s}{\Rightarrow} N_9$ the variable `Y` is abstractly bound to the type t defined as: $t \rightarrow [\top, \top]$. This is now incompatible with the expected property *arithexpr* and the diagnoser signals that ⑥ supports the symptom. More precisely, the point of exiting the second clause of `perm/2` after recursive call from the same clause causes the symptom. In other words, we are sure that whatever execution follows the nodes of s it will cause a run-time error at `X =< Y`.

Note that the diagnoser performs non-deterministic choices in OR-nodes. In fact, our system returns a second answer as a potential source of the symptom: the point exiting the first clause of `perm/2` after the recursive call. The corresponding sequence of nodes is $s = [N_5, N_{11}, N_6, N_3, N_2, N_4, N_9]$. This program point appears counterintuitive, and obviously is not an actual error. Nevertheless, observe that writing for example `perm([],2)` instead of `perm([],[])` would make the goal `?- slowsort([1],L)` succeed without the run-time error.

7 Binding Error Searching Algorithm

In this section we present the actual algorithm for finding binding errors. The algorithm makes use of the $\overset{s}{\Rightarrow}$ transition introduced in Section 5, as illustrated in the example of the previous section.

The diagnosis procedure is shown in Algorithm 1. The algorithm takes as input an abstract AND–OR graph R, a clause containing a symptom at the given program point, and an expected property at that point. The set $\mathcal{E}$ of program points supporting the symptom is returned as output. The algorithm consists of the main module (lines 1-3) and two mutually recursive procedures search_AND(A, i, s) and search_OR(A, s), which implement the traversal of the graph against the control flow.

The search_OR(O, s) procedure takes the atom O in the OR–node, and the sequence s of nodes visited so far, i.e., the nodes following O in the control-flow order. Then, for every child (an AND–node) A of O, A is concatenated with s and the algorithm verifies whether the transition $\Rightarrow$ over the new sequence of nodes leads to a violation of the expected property at the program point of the initial symptom (line 25). Note that we collect nodes from $R^\top$ rather than from R. This allows us to differentiate abstract substitutions generated in the analysis phase from those generated during the error location step. I.e., we are able to identify if the problem causing the assertion violation is within

Algorithm 1. The diagnosis algorithm.

Input:
- analysis output in the form of an abstract AND–OR graph R,
- a clause $C^s = H^s \leftarrow B^s_1, \ldots, B^s_{n_s}$,
- an index (program point) $1 \leq i_s \leq n_s$,
- an expected abstract substitution λ_{Prop} at the program point between $B^s_{i_s}$ and $B^s_{i_s+1}$, (or after $B^s_{n_s}$ if $i_s = n_s$).

Output: a set of binding errors $\mathcal{E}$.

1: $\mathcal{E} := \emptyset$, Visited $:= \emptyset$
2: let $A := \langle \lambda^\top, H^s\sigma, \lambda^\top \rangle^{\mathsf{AND}}$ be an AND-node in $R^\top$ corresponding to clause C^s, where σ is a renaming substitution
3: search_AND$(A, i_s, [A])$
4: **procedure** search_AND(A, i, s) {A: AND–node, i: index, s: sequence of nodes}
5: **if** $i = 0$ **then**
6: let O be an OR–node s.t. $A \in$ *children*(O) in $R^\top$
7: $s' := [O] : s$
8: **if** $O \stackrel{s'}{\Rightarrow} \langle \lambda, B^s_{i_s+1}, \lambda^\top \rangle^{\mathsf{AND}}$ and $\lambda \sqcap \lambda_{Prop} = \bot$ **then**
9: $\mathcal{E} := \mathcal{E} \cup \{entry(O, A)\}$
10: **else**
11: let A' be an AND–node in $R^\top$ s.t. O is the j-th child of A'
12: **if** $(O, A') \notin$ Visited **then**
13: Visited := Visited $\cup \{(O, A')\}$
14: search_AND$(A', j-1, [A'] : s')$
15: **end if**
16: **end if**
17: **else if** $(A, O') \notin$ Visited **then**
18: Visited := Visited $\cup \{(A, O')\}$
19: let O' be the i-th child of A
20: search_OR$(O', [O'] : s)$
21: **end if**
22: **procedure** search_OR(O, s) {O: OR–node, s: sequence of nodes}
23: **for all** $A \in$ *children*(O) in $R^\top$ **do**
24: $s' := [A] : s$
25: **if** $A \stackrel{s'}{\Rightarrow} \langle \lambda, B^s_{i_s+1}, \lambda^\top \rangle^{\mathsf{AND}}$ and $\lambda \sqcap \lambda_{Prop} = \bot$ **then**
26: $\mathcal{E} := \mathcal{E} \cup \{exit(A, O)\}$
27: **else**
28: $n := |children(A)|$
29: search_AND(A, n, s')
30: **end if**
31: **end for**

the current sequence of visited nodes (ideally in the first one in the sequence). If we take a node from R the variable bindings that cause the problem might have been placed in the current node or they may have been propagated from the preceding (in control-flow sense) nodes through abstract substitutions. By using $R^\top$ we "isolate" abstract values of the current nodes from the ones in the preceding nodes. If A supports the symptom the term $exit(A, O)$ is added to $\mathcal{E}$

to indicate that the critical binding occurs when the (abstract) execution leaves a clause with head A after completing call O. Otherwise, the search_AND(A, n, s) procedure is called which performs similar actions in an AND–node.

The search_AND(A, i, s) procedure performs one step of backwards traversal of an instance of a clause with head A. The body atom in question is determined by the index i. If the clause body has already been traversed ($i = 0$, line 5), the OR–node O corresponding the call to A is found and the algorithm checks whether entering the clause from O causes the symptom (line 8). If true, then, similarly to the OR–node case, the term $entry(O, A)$ is recorded in $\mathcal{E}$ (line 9). Otherwise, the search continues in the upper AND–node A' (A' is a head and O a body atom of the same clause) with the index value $j - 1$ pointing to the atom just before O in the clause body (line 14). If $i > 0$ when calling search_AND(A, i, s), then the OR–node corresponding to the i-th atom in the body is inspected. Note that when search_AND is called from inside search_OR (line 29) the second argument is set to n, i.e., to the length of the corresponding clause body. The reason for this is that we want to examine the last atom in the body first, in order to find an atom supporting the symptom located as close as possible to the symptom.

As the AND–OR graph may contain cycles, the algorithm keeps track of visited nodes using for that purpose the global variable Visited. Observe, however, that AND–nodes can be visited multiple times during traversal of the graph, and therefore we need to store not only a node but also a node visited in the previous step of the current traversal. That is why the elements of Visited are pairs of nodes rather than individual nodes.

8 Conclusions and Future Work

We have implemented a prototype of the diagnoser in Ciao [4] and integrated it into the Ciao Preprocessor, CiaoPP [14], whose abstract interpretation engine, PLAI [20,15]. The diagnoser makes use of abstract operations of PLAI and its data structures. The diagnoser inherits the parametric nature of the PLAI system, which allows the addition of arbitrary abstract domains as plugins. As a consequence of this the symptoms for which errors can be localized range over the same properties that can be inferred with the different domains available in the system: types/shapes, instantiation modes, pointer aliasing and structure sharing, determinacy, non-failure, etc.

The efficiency of the diagnosis procedure seems to be satisfactory, at least for the relatively small-sized programs that we have tested to date. For the *slowsort* example from Section 6, for example, the diagnosis time, including searching for all the errors, was 9.33 ms., compared to 34.66 ms. taken by the analysis. For standard *quicksort* the diagnosis took 205.97 ms. and analysis 92.98 ms. For another version of *quicksort*, i.e., with a different error, we measured 3457.47 ms. for diagnosis and 333.94 ms for analysis. Diagnosing the same bug starting from two other, different symptoms took 2474.62 ms. and 1185.82 ms. respectively.[5] Further benchmarking of the system is planned as future work.

[5] The tests where run on CPU Pentium 4, 1.8GHz, and 1 GB RAM.

Inevitably, as shown in Section 6, our system may identify several points as sources of an error symptom. Not all of them are actual errors in the sense of the user's expectations, but they are all reasons for the symptom. Also, due to the approximate (but safe) nature of reasoning in abstract interpretation and consequently in our algorithm, we are not guaranteed to find sources for every symptom. In particular, this happens when the abstract value inferred by the analyzer at the point of the symptom is $\top$. This problem can often be overcome by adding more assertions to the program (something which may encourage programmers to write more assertions). In this case the assertions holding expected properties can guide the diagnosis process.

In principle, a similar effect to that achieved by our error location method could be achieved by means of backward analysis [17], and this was indeed the first solution that we considered. However, backwards analysis requires the definition of new and relatively complex operations on the abstract domain. In addition, the abstract domains used must be condensing, which is a property satisfied only by a reduced number of the domains used in practice. Our approach allows using an arbitrary abstract domain, and simplifies the implementation since it reuses the standard operations which are already defined in the system for each domain. We argue that this is a practical advantage, worth taking perhaps some performance penalty.

Acknowledgments. The authors thank the anonymous reviewers for useful comments and suggestions. This work was funded in part by EU IST FET project FP6 IST-15905 *MOBIUS*, MEC project TIN2005-09207-C03 *MERIT-COMVERS*, and CAM project S-0505/TIC/0407 PROMESAS-CM. M. Hermenegildo is also supported in part by the Prince of Asturias Chair in Information Science and Technology at UNM. P. Pietrzak is supported by a MEC "Juan de la Cierva" grant.

References

1. Apt, K.R.: Introduction to Logic Programming. In: van Leeuwen, J. (ed.) Handbook of Theoretical Computer Science. Formal Model and Semantics, vol. B, pp. 495–574. Elsevier, Amsterdam and The MIT Press, Cambridge (1990)
2. Beaven, M., Stansifer, R.: Explaining type errors in polymorphic languages. ACM Letters on Programming Languages 2, 17–30 (1993)
3. Bruynooghe, M.: A Practical Framework for the Abstract Interpretation of Logic Programs. Journal of Logic Programming 10, 91–124 (1991)
4. Bueno, F., Cabeza, D., Carro, M., Hermenegildo, M., López-García, P., Puebla, G. (eds.): The Ciao System. Reference Manual (v1.10). Technical report, School of Computer Science (UPM) (2004), Available at `http://www.ciaohome.org`
5. Bueno, F., Cabeza, D., Hermenegildo, M., Puebla, G.: Global Analysis of Standard Prolog Programs. In: Nielson, H.R. (ed.) ESOP 1996. LNCS, vol. 1058, pp. 108–124. Springer, Heidelberg (1996)
6. Comini, M., Levi, G., Meo, M.C., Vitiello, G.: Abstract diagnosis. Journal of Logic Programming 39(1-3), 43–93 (1999)
7. Cousot, P., Cousot, R.: Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In: Proc. POPL, pp. 238–252 (1977)

8. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: The ASTRÉE Analyser. In: Sagiv, M. (ed.) ESOP 2005. LNCS, vol. 3444, pp. 21–30. Springer, Heidelberg (2005)
9. Dart, P.W., Zobel, J.: A Regular Type Language for Logic Programs. In: Types in Logic Programming, pp. 157–187. MIT Press, Cambridge (1992)
10. Drabent, W., Małuszyński, J., Pietrzak, P.: Using parametric set constraints for locating errors in CLP programs. TPLP 2(4-5), 549–611 (2002)
11. Drabent, W., Nadjm-Tehrani, S., Maluszynski, J.: Algorithmic debugging with assertions. In: Abramson, H., Rogers, M.H. (eds.) Meta-programming in Logic Programming, pp. 501–522. MIT Press, Cambridge (1989)
12. Gyimóthy, T., Paakki, J.: Static Slicing of Logic Programs. In: Proc. AADEBUG'95, IRISA-CNRS, pp. 87–103 (1995)
13. Hermenegildo, M., Puebla, G., Bueno, F.: Using Global Analysis, Partial Specifications, and an Extensible Assertion Language for Program Validation and Debugging. In: The Logic Programming Paradigm: a 25–Year Perspective, pp. 161–192. Springer, Heidelberg (1999)
14. Hermenegildo, M., Puebla, G., Bueno, F., López García, P.: Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). Science of Computer Programming 58(1-2), 115–140 (2005)
15. Hermenegildo, M., Puebla, G., Marriott, K., Stuckey, P.: Incremental Analysis of Constraint Logic Programs. ACM TOPLAS 22(2), 187–223 (2000)
16. Janssens, G., Bruynooghe, M.: Deriving Descriptions of Possible Values of Program Variables by means of Abstract Interpretation. Journal of Logic Programming 13(2 and 3), 205–258 (1992)
17. King, A., Lu, L.: A Backward Analysis for Constraint Logic Programs. TPLP 2(4-5), 32 (2002)
18. Lloyd, J.W.: Foundations of Logic Programming, 2nd edn. Springer, Heidelberg (1987)
19. Le Métayer, D.: Proving properties of programs defined over recursive data structures. In: ACM PEPM, pp. 88–99 (1995)
20. Muthukumar, K., Hermenegildo, M.: Compile-time Derivation of Variable Dependency Using Abstract Interpretation. JLP 13(2/3), 315–347 (1992)
21. Puebla, G., Bueno, F., Hermenegildo, M.: An Assertion Language for Constraint Logic Programs. In: Deransart, P., Małuszyński, J. (eds.) Analysis and Visualization Tools for Constraint Programming. LNCS, vol. 1870, pp. 23–61. Springer, Heidelberg (2000)
22. Puebla, G., Bueno, F., Hermenegildo, M.: Combined Static and Dynamic Assertion-Based Debugging of Constraint Logic Programs. In: Bossi, A. (ed.) LOPSTR 1999. LNCS, vol. 1817, pp. 273–292. Springer, Heidelberg (2000)
23. Schoenig, S., Ducasse, M.: A Backward Slicing Algorithm for Prolog. In: Cousot, R., Schmidt, D.A. (eds.) SAS 1996. LNCS, vol. 1145, pp. 317–331. Springer, Heidelberg (1996)
24. Shapiro, E.: Algorithmic Program Debugging. In: ACM Distiguished Dissertation, MIT Press, Cambridge (1982)
25. Vaucheret, C., Bueno, F.: More Precise yet Efficient Type Inference for Logic Programs. In: Hermenegildo, M.V., Puebla, G. (eds.) SAS 2002. LNCS, vol. 2477, pp. 102–116. Springer, Heidelberg (2002)
26. Wand, M.: Finding the source of type errors. In: Proc. POPL, pp. 38–43 (January 1986)

User-Definable Resource Bounds Analysis for Logic Programs

Jorge Navas[1], Edison Mera[2], Pedro López-García[3], and Manuel V. Hermenegildo[1,3]

[1] University of New Mexico, USA
[2] Complutense University of Madrid, Spain
[3] Technical University of Madrid, Spain

Abstract. We present a static analysis that infers both upper and lower bounds on the usage that a logic program makes of a set of user-definable resources. The inferred bounds will in general be functions of input data sizes. A *resource* in our approach is a quite general, user-defined notion which associates a basic cost function with elementary operations. The analysis then derives the related (upper- and lower-bound) resource usage functions for all predicates in the program. We also present an assertion language which is used to define both such resources and resource-related properties that the system can then check based on the results of the analysis. We have performed some preliminary experiments with some concrete resources such as execution steps, bytes sent or received by an application, number of files left open, number of accesses to a database, number of calls to a procedure, number of asserts/retracts, etc. Applications of our analysis include resource consumption verification and debugging (including for mobile code), resource control in parallel/distributed computing, and resource-oriented specialization.

1 Introduction

It is generally recognized that inferring information about the costs of computations can be useful for a variety of applications. These costs are usually related to execution steps and, sometimes, time or memory. We propose an analyzer which allows automatically inferring both upper and lower bounds on the usage that a logic program makes of *user-definable resources*. Examples of such user-definable resources are bits sent or received by an application over a socket, number of calls to a procedure, number of files left open, number of accesses to a database, number of licenses consumed, monetary units spent, disk space used, etc., as well as the more traditional execution steps, execution time, or memory. We expect the inference of this kind of information to be instrumental in a variety of applications, such as resource usage verification and debugging, certification of resource consumption in mobile code, resource/granularity control in parallel/distributed computing, or resource-oriented specialization.

In our approach a resource is a user-defined notion which associates a basic cost function with elementary operations in the base language and/or to some

V. Dahl and I. Niemelä (Eds.): ICLP 2007, LNCS 4670, pp. 348–363, 2007.

predicates in libraries. In this sense, each *resource* is essentially a user-defined *counter*. The user gives a name (such as, e.g., `bits_received`) to the counter and then defines via assertions how each elementary operation in the program (e.g., unifications, calls to builtins, external calls, etc.) increments or decrements that counter. The use of resources obviously depends in practice on the sizes or values of certain inputs to programs or predicates. Thus, in the assertions describing elementary operations the counters may be incremented or decremented not only by constants but also by amounts that are *functions* of input data sizes or values. Correspondingly, the objective of our method is to statically derive from these elementary assertions and the program text *functions* that yield upper and lower bounds on the amount of those resources that each of the predicates in the program (and the program as a whole) will consume or provide. The input to these functions will also be the sizes or value ranges of the topmost input data to the program or predicate being analyzed.

As mentioned previously, most previous work is specific to the analysis of execution steps and, sometimes, time or memory. The ACE system [16] can automatically extract upper bounds on *execution steps* for a subset of functional programming. The system is based on program transformation. The original program is transformed into a step-counting version and then into a composition of a cost bound and a measure function. In [19] another automatic upper-bound analysis is presented based on an abstract interpretation of a step-counting version. The analysis measures both execution time and execution steps. However, size measures cannot automatically be inferred and the experimental section shows few details about the practicality of the analysis. In [8,7] a semi-automatic analysis is presented which infers upper-bounds on the number of execution steps. These bounds are functions on the sizes or value ranges of input data. This seminal work applies to a large class of logic programs and presents techniques in order to deal with the generation of multiple solutions via backtracking. The authors also show how other specific analyses could be developed, such as for, e.g., time or memory. This approach was later fully automated and extended to inferring upper- and lower-bounds on the number of *execution steps* (which is non-trivial because of the possibility of failure) in [9,12]. Our method builds on this work but generalizes it in order to deal with a much more general class of *user-defined* resources, allowing thus the coverage of an unlimited number of analyses within a single implementation. In [11] a method is presented for automatically extracting cost recurrences from first-order DML programs. The main feature is the use of dependent types to describe a size measure that abstracts from data to data size. In [17], and inspired by [3] and [15], a complexity analysis is presented for Horn clauses, also fully automating the necessary calculations. In [13] a method is presented for modeling problems such as memory management, lock primitive usage, etc., and a type-based method is proposed as solution to the inference problem. In [21] a cost model is presented for inferring cost equations for recursive, polymorphic, and higher-order functional programs. While it is claimed that the approach can be modified in order to infer a reduced set of resources such as execution time, execution steps, or memory, no details are given. Worst case execution time (WCET)

estimation has been studied for imperative languages and for different application domains (see, e.g., [20,4,10] and its references). However these and related methods again concentrate only on execution time. Also, they do not infer cost functions of input data sizes but rather absolute maximum execution times, and they generally require the manual annotation of loop iteration bounds. In [5] a method is presented for reserving resources before their actual use. However, the programmer (or program optimizer) needs to annotate the program with "acquire" and "consume" primitives, as well as provide loop invariants and function pre- and post-conditions. Interesting type-based related work has also been performed in the GRAIL system [1], also oriented towards resource analysis, but it has concentrated mainly on ensuring memory bounds.

In comparison with previous work our approach allows dealing with a class of resources which is open, in the ample sense that such resources are in fact defined by programmers using an assertion language, which we also consider itself an important contribution of our work. Another important contribution of our work (because its impact in the scalability and automation of the analysis) is that our approach allows defining the resource usage of external predicates, which can be used for modular composition. In addition, assertions also allow describing by hand the usage of any predicate for which the automatic analysis infers a value that is not accurate enough, and this can be used to prevent inaccuracies in the automatic inference from propagating. In the following (Sect. 2) we first present the assertion language proposed. Sect. 3 then provides an overview of the approach, Sect. 4 shows how size relationships among program variables are determined, and Sect. 5 and 6 describe how the resource usage functions are inferred. We have implemented the proposed analysis and applied it to a series of example programs. The results are presented in Sect. 7. Finally, Sect. 8 summarizes our conclusions.

2 The Resource Assertion Language

We start by describing the assertion schema. This language is used for describing resources and providing other input to the resource analysis, and is also the language in which the resource analysis produces its output. This assertion language is used additionally to state resource-related specifications (which can then be proved or disproved based on the results of analysis following the scheme of [12], which allows finding bugs, verifying the program, etc.).

The rules for the assertion language grammar are listed in Fig. 1. In this grammar Var corresponds to variables written in the syntax for variables of the underlying logic programming language (i.e., normally non-empty strings of characters which start with a capital letter or underscore). Similarly, Num is any valid number and *Pred_name* any valid name for a predicate in the underlying programming language (normally non-empty strings of characters which start with a lower-case letter or are quoted). *State_prop* corresponds to other *state properties* (such as modes and types), and *Comp_prop* stands for any other valid *computational property* (see [12] and its references).

$\langle program\_assrt \rangle$ ::= **:-** $\langle status\_flag \rangle$ $\langle pred\_assrt \rangle$.
| **:- head_cost**($\langle approx \rangle$,Res_name,Δ^H).
| **:- literal_cost**($\langle approx \rangle$,Res_name,Δ^L).
$\langle status\_flag \rangle$ ::= **trust** | **check** | **true** | ϵ
$\langle pred\_assrt \rangle$ ::= **pred** $\langle pred\_desc \rangle$ $\langle pre\_cond \rangle$ $\langle post\_cond \rangle$ $\langle comp\_cond \rangle$.
$\langle pred\_desc \rangle$ ::= Pred_name | Pred_name($\langle args \rangle$)
$\langle args \rangle$ ::= Var | Var, $\langle args \rangle$
$\langle pre\_cond \rangle$ ::= **:** $\langle state\_props \rangle$ | ϵ
$\langle post\_cond \rangle$ ::= => $\langle state\_props \rangle$ | ϵ
$\langle comp\_cond \rangle$ ::= + $\langle comp\_props \rangle$ | ϵ
$\langle state\_prop \rangle$ ::= **size**(Var,$\langle approx \rangle$,$\langle sz\_metric \rangle$,$\langle arith\_expr \rangle$) | State_prop
$\langle state\_props \rangle$ ::= $\langle state\_prop \rangle$ | $\langle state\_prop \rangle$, $\langle state\_props \rangle$
$\langle comp\_prop \rangle$ ::= **size_metric**(Var,$\langle sz\_metric \rangle$) | $\langle cost \rangle$ | Comp_prop
$\langle comp\_props \rangle$ ::= $\langle comp\_prop \rangle$ | $\langle comp\_prop \rangle$, $\langle comp\_props \rangle$
$\langle cost \rangle$ ::= **cost**($\langle approx \rangle$,Res_name,$\langle arith\_expr \rangle$)
$\langle approx \rangle$ ::= **ub** | **lb** | **oub** | **olb**
$\langle sz\_metric \rangle$ ::= **value** | **length** | **size** | **void**
$\langle arith\_expr \rangle$ ::= $-$ $\langle arith\_expr \rangle$ | $\langle arith\_expr \rangle$! | $\langle quantifier \rangle$ $\langle arith\_expr \rangle$
| $\langle arith\_expr \rangle$ $\langle bin\_op \rangle$ $\langle arith\_expr \rangle$
| $\langle arith\_expr \rangle^{\langle arith\_expr \rangle}$ | log_{Num} $\langle arith\_expr \rangle$
| Num | $\langle sz\_metric \rangle$(Var)
$\langle bin\_op \rangle$ ::= $+$ | $-$ | $*$ | $/$
$\langle quantifier \rangle$ ::= $\sum$ | $\prod$

Fig. 1. Syntax of the Resource Assertion Language

Predicates can be annotated with zero or more **pred** assertions, which state properties of the execution states when the predicate is called (*pre_cond*), properties of the execution states when the predicate terminates execution (*post_cond*), or properties of the whole computation of the predicate, rather than the input-output behavior (*comp_cond*, which herein will be used only for resource-related properties). For brevity, the $\langle state\_props \rangle$ fields can also be written using "star notation" (see the examples). In addition, there may be a set of global **head_cost** and **literal_cost** declarations (one for each resource and approximation direction). The *Res_name* fields determine which resource the assertion refers to. These *Res_name*s are user-provided identifiers which give a name to each particular resource that needs to be tracked. Resources do not need to be declared in any other way –the set of resources that the system is aware of is simply the set of such names that appear in assertions which are in the scope. The $\langle approx \rangle$ fields state whether $\langle arith\_expr \rangle$ is providing an upper bound or a lower bound (with **oub** meaning it is a "big O" expression, i.e., with only the order information, and **olb** meaning it is an Ω asymptotic lower bound).

The first and most fundamental use of assertions in our context is to describe how the execution of some predicates increments or decrements the usage of the resources (counters) defined in the program. The purpose of analysis is then to infer the resource usage of all predicates in the program. The **head_cost**($\langle approx \rangle$, Res_name, Δ^H) declarations are used to describe how predicates in general *update*

the value for those resources that are applicable only to predicate heads (such as counting the number of arguments passed or total execution steps –see Section 5). The definition of Δ^H : *cl_head* $\rightarrow$ *arith_expr* is provided by means a user-defined (or imported) predicate, written in the source language, and which will be called by the analyzer when the clause head is analyzed. This code gets loaded into the compiler in a similar way to, e.g., macro expansion code. The **literal_cost**($\langle approx \rangle$, Res_name, Δ^L) declarations describe how predicate bodies *update* the value of certain resources which are applicable only to body literals (such as, for example, number of unifications). In this case, Δ^L : *body_lit* $\rightarrow$ *arith_expr* is also user- (or library-)provided code which will be executed when the body literals of different predicates are analyzed. The **cost**($\langle approx \rangle$, Res_name, $\langle arith\_expr \rangle$) comp-type properties are included in **pred** assertions and used to provide the actual resource usage functions for each builtin or external (e.g., defined in another language) predicate used in the program. Such assertions have **trust** status (meaning that the analysis will assume this value [12]). As mentioned previously, the aim of the analysis is to derive functions that describe the resource usage (as well as argument size relations) for the rest of the predicates in the program. Note however that it is also possible to provide **pred** assertions for some of those predicates and this can also be used to guide the analysis. In particular, the analysis will compute the most precise expression between the resource usage function provided by the assertion ($\langle cost \rangle$) and the resource usage function inferred by analysis. Additionally, size metric (**size_metric**(Var,$\langle sz\_metric \rangle$)) information can be provided by users if needed (but note that in practice size metrics can often be derived automatically from the inferred types).

Assertions can also be used, via the *pre_cond* and *post_cond* fields, to declare relationships between the data sizes of the inputs and outputs of predicates, which are needed by our analysis, as will be described later. These assertions are also used to label predicate arguments as input or output, as well as to provide types or size (**size**(Var,$\langle approx \rangle$,$\langle sz\_metric \rangle$,$\langle arith\_expr \rangle$)) information. In the same way as with the $\langle cost \rangle$ properties, for user-defined predicates these other assertions can be provided by the user or inferred by analysis. Again, analysis will compute the most precise of the two.

Example 1. The following example shows how a program can be annotated using our assertion language in order to perform resource analysis. Consider a client application in Fig 2 that sends a data buffer through a socket[1] and receives another (possibly transformed) data buffer. Assume that we would like to obtain an upper bound on the number of bits received by the application –a *resource* that we will call `bits_received`.

As we will see, the approach needs to know for each argument in the program the metric and whether it is input or output in order to perform properly the size and resource usage analyses described in Sect. 4 and 5. Input/output and metric information can be required by the language (typed language), given by the user (via assertions), or, as in our implementation, inferred automatically via analysis.

[1] Note that the types of the socket operations must be given to the analysis by other analyses or by user-provided assertions.

```
:- pred main(Opts, IBuf, OBuf)
   : list(gnd) * list(byte) * var.
main([Host,Port],IBuf,OBuf) :-
   connect_socket(Host,Port,Stream),
   exchange_buffer(IBuf,Stream,OBuf),
   close(Stream).
exchange_buffer([],_,[]).
exchange_buffer([B|Bs],Id,[B0|Bs0]) :-
   exchange_byte(B,Id,B0),
   exchange_buffer(Bs,Id,Bs0).
:- trust pred
   connect_socket(Host,Port,Stream)
   :  atm * num * var
   => atm * num * atm
   +  cost(ub,bits_received,0).
```

```
:- trust pred close(Stream)
   :  atm => atm
   +  cost(ub,bits_received,0).
:- trust pred exchange_byte(B,Id,B0)
   :  byte * num * var
   => byte * num *
     {byte(B0), size(B0,ub,bytes,1)}
   +  cost(ub,bits_received,8).

:- head_cost(ub,bits_received,
             head_bits_received).
head_bits_received(_,0).
:- literal_cost(ub,bits_received,
             literal_bits_received).
literal_bits_received(_,0).
```

Fig. 2. A simple client application

3 Overview of the Approach

Our basic approach is as follows. Given a predicate call p, let $\Phi(p, r, n)$ denote the units of resource r consumed or produced during the computation of p for an input of size n. An expression $\mathtt{RU}_{pred}(p,ap,r,n)$ is determined (at compile-time) that approximates $\Phi(p, r, n)$ with approximation ap. Typical size metrics are the actual value of a number, the length of a list or array, the size (number of nodes and fields) of a data structure, etc. We will refer to such $\mathtt{RU}_{pred}(p,ap,r,n)$ expressions as *resource usage bound functions*. Certain program information (such as, for example, input/output modes and size metrics for predicate arguments) is first automatically inferred by other (abstract interpretation-based) analyzers and then provided as input to the size and resource analysis (the techniques involved in inferring this information are beyond the scope of this paper —see, e.g., [12] and its references for some examples). Based on this information, our analysis first finds bounds on the size of input arguments to the calls in the body of the predicate being analyzed, relative to the sizes of the input arguments to this predicate using the inferred metrics. The size of an output argument in a predicate call depends in general on the size of the input arguments in that call. For this reason, for each output argument we infer an expression which yields its size as a function of the input data sizes. To this end, and using the input-output argument information, data dependency graphs are used to set up difference equations whose solution yields size relationships between input and output arguments of predicate calls. This information regarding argument sizes is then used to set up another set of difference equations whose solution provides bound functions on resource usage. Both the size and resource usage difference equations must be solved by a difference equation solver. Although the operation of such solvers is beyond the scope of the paper our implementation does provide a table-based solver which covers a reasonable set of difference equations such as first-order and higher-order linear difference equations in one variable with

constant and polynomial coefficients,[2] divide and conquer difference equations, etc. In addition, the system allows the use of external solvers (such as, e.g. [2], Mathematica, Matlab, etc.). Note also that, since we are computing upper/lower bounds, it suffices to compute upper/lower bounds on the solution of a set of difference equations, rather than an exact solution. This allows obtaining an *approximate* closed form when the exact solution is not possible.

4 Size Analysis

We will give the intuition behind the data dependency-based method for inferring bounds on the sizes of output arguments in the head of a predicate as a function of the sizes of input arguments to the predicate. Besides this, as a result of the size analysis, we have bounds on the size of each input argument to body literals in a clause as a function of the size of the input arguments to the head of that clause. The size of the input arguments to body literals will be used later to infer functions which give bounds on the resource usage of body literals in terms of the sizes of the input arguments to the head. We adopt the approach of [8,7] for the inference of upper bounds on argument sizes and [9] for lower bounds. For the sake of brevity, we will only consider the inference of upper bounds in this paper, and refer the reader to [9] for the inference of lower bounds.

Various metrics are used for the "size" of an argument (e.g., *value*, *length*, *size*,[3]...). For the sake of brevity, we will only consider the *length* metric in this paper, and refer the reader to [8,7,9] for other metrics.

We define the $\texttt{size}(\langle sz\_metric\rangle, t)$ operation which returns the size of a term t under the metric $\langle sz\_metric\rangle$:

$$\texttt{size}(length, t) = \begin{cases} 0 & \text{if } t = [\,] \text{ (the empty list)} \\ 1 + \texttt{size}(length, T) & \text{if } t = [H|T] \\ \bot & \text{otherwise.} \end{cases}$$

Thus, $\texttt{size}(length, [X, Y]) = 2$, and $\texttt{size}(length, [X|Y]) = \bot$.

We also define the $\texttt{diff}(\langle sz\_metric\rangle, t_1, t_2)$ operation, which returns (an upper bound on) the size difference between two terms t_1 and t_2 under the metric $\langle sz\_metric\rangle$.

$$\texttt{diff}(length, t_1, t_2) = \begin{cases} 0 & \text{if } t_1 \equiv t_2 \\ \texttt{diff}(length, t, t_2) - 1 & \text{if } t_1 = [\_|t] \text{ for some term t} \\ \bot & \text{otherwise.} \end{cases}$$

Thus, $\texttt{diff}(length, [a, b|T], T) = -2$.

A directed acyclic graph called *argument dependency graph* is used to represent the data dependency between argument positions in a clause body (and between them and those in the clause head). Each node in the graph denotes an argument position. There is an edge from a node n_1 to a node n_2 if the variable

[2] Note that it is always possible to reduce a system of linear difference equations to a single linear difference equation in one variable.

[3] Metric *void* in an argument means that analysis does not need to infer its size.

bindings generated by n_1 are used to construct the term occurring at n_2. The node n_1 is said to be a *predecessor* of the node n_2, and n_2 a *successor* of n_1.

Using the `size` and `diff` functions and the argument dependency graph we can set up size relations for expressing the size of each argument position in terms of the sizes of its predecessors. Let $\mathtt{sz}(a)$ denote the size of the term occurring at an argument position a, and $@a$ the term occurring at an argument position a. For the sake of simplicity, we will omit the argument $\langle sz\_metric \rangle$ in the `size` and `diff` functions in the rest of the paper.

- *Output arguments.* Let $l_1, \ldots, l_n$ denote the input argument positions of the literal L, and let $\Psi_p^b : \mathcal{N}_{\perp,\infty}^n \mapsto \mathcal{N}_{\perp,\infty}$ be a function that represents the size of the b-th (output) argument position of the predicate p of literal L in terms of the size of its input positions, where $\mathcal{N}_\perp$ denotes the set of natural numbers augmented with the special symbols $\perp$ (denoting "undefined"), and ∞. Assume that a is an output argument position in a clause. Then the following size relation is set up:
$$\mathtt{sz}(a) \leq \Psi_p^a(\mathtt{sz}(l_1), \ldots, \mathtt{sz}(l_n))$$
If L is recursive, then $\Psi_p^a(\mathtt{sz}(l_1), \ldots, \mathtt{sz}(l_n))$ is a symbolic expression. However, if L is non-recursive then the function Ψ_p^a has been recursively computed, and thus we replace $\Psi_p^a(\mathtt{sz}(l_1), \ldots, \mathtt{sz}(l_n))$ by the (explicit) expression resulting from the application of the function Ψ_p^a to $\mathtt{sz}(l_1), \ldots, \mathtt{sz}(l_n)$.
- *Input arguments.* Assume that a is an input argument position in a body literal. Let $\mathtt{predecessors}(a)$ be the set of predecessors of a in the argument dependency graph. We have the following possibilities:
  1. Compute $\mathtt{size}(@a)$. If $\mathtt{size}(@a) \neq \perp$ then set up the size relation: $\mathtt{sz}(a) \leq \mathtt{size}(@a)$.
  2. Otherwise, if $\exists r \in \mathtt{predecessors}(a)$ such that the size metrics corresponding to r and a are the same and $d = \mathtt{diff}(r, a) \neq \perp$, then $\mathtt{sz}(a) \leq \mathtt{sz}(r) + d$.
  3. Otherwise, if $\mathtt{size}(@a)$ can be expanded using the definition of the `size` function, then expand $\mathtt{size}(@a)$ one step and recursively compute $\mathtt{size}(t_i)$ for the appropriate subterms t_i of $@a$. If each of these recursive size computations have a defined result, then use them to compute the size relation for $\mathtt{size}(@a)$. Otherwise, $\mathtt{sz}(a) = \perp$.

Size relations can be propagated to transform a size relation corresponding to an input argument in a body literal or an output argument in the clause head into a function in terms of the sizes of the input arguments of the head. The basic idea here is to repeatedly substitute size relations for body literals into size relations for head arguments. This is the purpose of the normalization algorithm described in [7]. However, for recursive clauses, we need to solve the symbolic expression due to recursive literals into an explicit function first.

Example 2. Consider again the program described in Fig. 2. Here and in the rest of the paper we will denote by *pred_name* the name of a predicate, and by $pred\_name_i^j$ the i-th argument position in the j-th literal with predicate

name *pred_name* in the body of a clause. If there is only one body literal with predicate name *pred_name* in the body of a clause then we omit the superscript j and write simply $pred\_name_i$. Let $head_i$ be the i-th argument position in the clause head. To simplify notation, we will represent `exchange_buffer/3` and `exchange_byte/3` with *ex_buf* and *ex_byt* respectively. Consider the recursive clause of `exchange_buffer/3`. First, the system sets up the size relation for the input/output arguments of the body literals:

$$\begin{array}{rcl}
\mathtt{sz}(ex_byt_1) & \leq & \mathtt{size}(B) = \mathtt{sz}(arg(1, head_1)) \\
\mathtt{sz}(ex_byt_2) & \leq & \mathtt{size}(Id) = \mathtt{sz}(head_2) + \mathtt{diff}(Id, Id) = \mathtt{sz}(head_2) \\
\mathtt{sz}(ex_byt_3) & \leq & \Psi^3_{ex_byt}(\mathtt{sz}(ex_byt_1), \mathtt{sz}(ex_byt_2)) = 1 \\
\mathtt{sz}(ex_buf_1) & \leq & \mathtt{size}(Bs) = \mathtt{sz}(head_1) + \mathtt{diff}([B|Bs], Bs) = \mathtt{sz}(head_1) - 1 \\
\mathtt{sz}(ex_buf_2) & \leq & \mathtt{size}(Id) = \mathtt{sz}(head_2) + \mathtt{diff}(Id, Id) = \mathtt{sz}(head_2) \\
\mathtt{sz}(ex_buf_3) & \leq & \Psi^3_{ex_buf}(\mathtt{sz}(ex_buf_1), \mathtt{sz}(ex_buf_2))
\end{array}$$

Then, system sets up the size relation for the output arguments of the head:

$$\begin{array}{rcl}
\mathtt{sz}(head_3) & \leq & \mathtt{size}([B0|Bs0]) = \mathtt{size}(Bs0) + 1 = \\
& & \mathtt{sz}(ex_buf_3) + \mathtt{diff}(Bs0, Bs0) + 1 = \mathtt{sz}(ex_buf_3) + 1
\end{array}$$

The normalization algorithm is applied to the previous size relation and gives:

$$\begin{array}{rcl}
\mathtt{sz}(head_3) & \leq & \Psi^3_{ex_buf}(\mathtt{sz}(ex_buf_1), \mathtt{sz}(ex_buf_2)) + 1 \\
& \leq & \Psi^3_{ex_buf}(\mathtt{sz}(head_1) - 1, \mathtt{sz}(head_2)) + 1
\end{array}$$

Thus, the system establishes the difference equation for the output argument ($head_3$) in the head (since it belongs to a recursive predicate). Then, it obtains the boundary condition $\Psi^3_{ex\_buf}(0, y) = 0$ from the non-recursive clause, and using it, it obtains a closed form function by calling the difference equation solver (variables x and y represent $\mathtt{sz}(head_1)$ and $\mathtt{sz}(head_2)$ respectively):

$$\begin{array}{rcl}
\Psi^3_{ex_buf}(0, y) & = & 0 \\
\Psi^3_{ex_buf}(x, y) & = & \Psi^3_{ex_buf}(x - 1, y) + 1
\end{array} \qquad \Rightarrow \Psi^3_{ex_buf}(x, y) = \quad x$$

5 Resource Usage Analysis

In order to infer the resource usage functions all predicates in the program are processed in a single traversal of the call graph in reverse topological order. Consider such a predicate `p` defined by clauses $\mathtt{C}_1, ..., \mathtt{C}_m$. Assume that $\overline{n}$ is a tuple such that each element corresponds to the size of an input argument position to predicate `p`. Then, the resource usage (expressed in units of resource r with approximation ap) of a call to `p`, for an input of size $\overline{n}$, can be expressed as:

$$\mathtt{RU}_{\mathtt{pred}}(p, ap, r, \overline{n}) = \bigodot(ap)_{1 \leq i \leq m}\{\mathtt{RU}_{\mathtt{clause}}(C_i, p, ap, r, \overline{n})\} \tag{1}$$

where $\bigodot(ap)$ is a function that takes an approximation identifier ap and returns a function which applies over all $\mathtt{RU}_{\mathtt{clause}}(C_i, p, ap, r, \overline{n})$, for $1 \leq i \leq m$. For example, if ap is the identifier for approximation "upper bound" (ub), then a possible conservative definition for $\bigodot(ap)$ is the $\sum$ function. In this case, and since the number of solutions generated by a predicate that will be demanded is generally not known in advance, a conservative upper bound on the computational cost of

a predicate is obtained by assuming that all solutions are needed, and that all clauses are executed (thus the cost of the predicate is assumed to be the sum of the costs of all of its clauses). However, it is straightforward to take mutual exclusion into account (information which is inferred by CiaoPP [14,12] and is available to our analysis) to obtain a more precise estimate of the cost of a predicate, using the maximum of the costs of mutually exclusive groups of clauses. If ap is the identifier for approximation "lower bounds" (lb), then $\bigodot(ap)$ is the min function.

Let us see now how to compute the resource usage of clauses. Consider a clause $\mathtt{C}$ of predicate p of the form $\mathtt{H} :- \mathtt{L_1}, ..., \mathtt{L_k}$ where $\mathtt{L_j}$, $1 \leq j \leq k$, is a literal (either a predicate call, or an external or builtin predicate), and $\mathtt{H}$ is the clause head. Assume that $\psi_j(\overline{n})$ is a tuple with the sizes of all the input arguments to literal $\mathtt{L_j}$, given as functions of the sizes of the input arguments to the clause head. Note that these $\psi_j(\overline{n})$ size relations have previously been computed during size analysis for all input arguments to literals in the bodies of all clauses.

Then, $\mathtt{RU_{clause}}(C, ap, r, \overline{n})$, the resource usage (expressed in units of resource r with approximation ap) of clause $\mathtt{C}$ of predicate p, is $\mathtt{RU_{clause}}(C, ap, r, \overline{n}) = closed\_form(\mathtt{RU}(p, ap, r, \overline{n}))$. That is, it is expressed as the solved form function of the following expression (which, in general, for recursive clauses yields a difference equation):

$$\mathtt{RU}(p, ap, r, \overline{n}) = \delta(ap, r)(head(C)) + \sum_{j=1}^{lim(ap,C)} (\prod_{l \prec j} \mathtt{Sols_{L_l}}(\overline{n}_l))(\beta(ap, r)(L_j) + \mathtt{RU_{lit}}(L_j, ap, r, \psi_j(\overline{n}))) \quad (2)$$

where $lim(ap, C)$ is a function that takes an approximation identifier ap and a clause C and returns the index of a literal in the clause body. For example, if ap is the identifier for approximation "upper bound" (ub), then $lim(ap, C) = k$ (the index of the last body literal). If ap is the identifier for approximation "lower bounds" (lb), then $lim(ap, C)$ is the index for the rightmost body literal that is guaranteed not to fail. $\delta(ap, r)$ is a function that takes an approximation identifier ap and a resource identifier r and returns a function $\Delta^H : cl\_head \rightarrow arith\_expr$ which takes a clause head and returns an arithmetic resource usage expression $< arith\_expr >$ as defined in Section 2. Thus, $\delta(ap, r)(head(C))$ represents $\Delta^H(head(C))$. On the other hand, $\beta(ap, r)$ is a function that takes an approximation identifier ap and a resource identifier r and returns a function $\Delta^L : body\_lit \rightarrow arith\_expr$ which takes a body literal and returns an arithmetic resource usage expression $< arith\_expr >$ as defined in Section 2. In this case, $\beta(ap, r)(L_j)$ represents $\Delta^L(L_j)$. $\mathtt{Sols_{L_l}}$ is the number of solutions that literal $\mathtt{L}_l$ can generate, where $l \prec j$ denotes that $\mathtt{L}_l$ precedes $\mathtt{L}_j$ in the literal dependency graph for the clause. Section 6 illustrates different definitions of the functions $\delta(ap, r)$ and $\beta(ap, r)$ in order to infer different resources.
$\mathtt{RU_{lit}}(\mathtt{L_j}, ap, r, \psi_j(\overline{n}))$ is:

- If $\mathtt{L_j}$ is recursive (i.e., calls a predicate $\mathtt{q}$ which is in the strongly-connected component of the call graph being analyzed), then $\mathtt{RU_{lit}}(\mathtt{L_j}, ap, r, \psi_j(\overline{n}))$ is replaced by a symbolic expression $\mathtt{RU}(q, ap, r, \psi_j(\overline{n}))$.

– If $\mathtt{L_j}$ is not recursive, assume that it is a call to q (where q can be either a predicate call, or an external or builtin predicate), then q has been already analyzed, i.e., the (closed form) resource usage function for q has been recursively computed as Φ' and $\mathtt{RU_{lit}}(\mathtt{L_j}, ap, r, \overline{n})$ can be expressed explicitly in terms of the function Φ', and it is thus replaced with $\Phi'(\psi_j(\overline{n}))$.

Note that in both cases, if there is a resource usage assertion for q, $cost(ap, r, \Phi)$, then $\mathtt{RU_{lit}}(\mathtt{L_j}, ap, r, \psi_j(\overline{n}))$ is replaced by the most precise (greatest lower bound if *upper bounds* or least upper bound if *lower bounds*) of a) the arithmetic resource usage expression (in closed form) $\Phi(\psi_j(\overline{n}))$ or b) its (closed form) resource usage function inferred previously by the analysis (provided they are not incompatible, in which case an error is flagged).

It can be proved by induction on the number of literals in the body of clause C that:

1. If clause C is not recursive, then, expression (2) results in a closed form function of the sizes of the input argument positions in the clause head;
2. If clause C is simply recursive, then, expression (2) results in a difference equation in terms of the sizes of the input argument positions in the clause head;
3. If clause C is mutually recursive, then expression (2) results in a difference equation which is part of a system of equations for mutually recursive clauses in terms of the sizes of the input argument positions in the clause head.

If these difference equations can be solved (including approximating the solution in the direction of ap) then $\mathtt{RU}(p, ap, r, \overline{n})$ can be expressed in a closed form, which is a function of the sizes of the input argument positions in the head of predicate p (and hence $\mathtt{RU_{clause}}(C, ap, r, \overline{n}) = closed\_form(\mathtt{RU}(p, ap, r, \overline{n}))$). Thus, after the strongly-connected component to which p belongs in the call graph has been analyzed, we have that expression (1) results in a closed form function of the sizes of the input argument positions in the clause head.

Note that our analysis is parameterized by the functions $\delta(ap, r)$ and $\beta(ap, r)$ whose definitions can be given by means of assertions of type **head_cost** and **literal_cost** respectively, as given in Figure 1. These functions make our analysis parametric w.r.t. resources (such as execution steps, bytes sent by an application/socket, number of calls to a procedure, etc.).

6 Defining the Parameters (Functions) of the Analysis

In this section we explain and illustrate with examples how the functions that make our resource analysis parametric, namely, δ (which includes the definition of Δ^H), and β (which includes the definition of Δ^L) are written in practice in our system. For brevity, we assume that we are interested in computing upper bounds on the different resources.

Assume for example that the resource we want to measure is (an upper bound on) the number of resolution steps performed by a program. Then we can define

$\delta(ub, steps) = delta\_one$, and define $delta\_one(H) = 1$ for all H. This is achieved by providing the following `head_cost` assertion and definition of the `delta_one` predicate:

```
:- head_cost(ub, steps, delta_one).
delta_one(_, 1).
```

In order to simplify the process of defining interesting and useful Δ^H and Δ^L functions, our implementation provides a library with predicates that perform (syntactic) operations on clauses, such as, for example, getting the number of arguments in a clause head or body literal, accessing an argument of a clause head or body literal, getting the main functor and arity of a term in a certain position, etc. In this context it is important to remember that the different Δ^H and Δ^L function definitions perform syntactic matching on the program text.

Assume now that the resource we want to measure is the number of argument passings that occur during clause head matching in a program (as an approximation to the number of unifications performed by the program). Then we can define $\delta(ub, num\_args) = delta\_num\_args$, and define $delta\_num\_args(H) = arity(H)$. This is achieved by the following code:

```
:- head_cost(ub, num_args, delta_num_args).
delta_num_args(H, N) :- functor(H, _, N).
```

As another example, if we are interested in decomposing arbitrary unifications performed while unifying a clause head with the literal being solved into simpler steps, we can define a resource num_unifs, define $\delta(ub, num\_unifs) = delta\_num\_unifs$, and define $delta\_num\_unifs(H) =$ *The number of function symbols, constants, and variables in* H.

If, in addition to the number of unifications performed while unifying a clause head, we are also interested in the cost of term creation for the literals in the body of clauses, we can define a resource $terms\_created$, and define a β function $\beta(ub, terms\_created) = beta\_terms\_created$, where $beta\_terms\_created(L) =$ *The number of function symbols, and constants in body literal* L. Note that in this case we define $\delta(ub, terms\_created) = delta\_terms\_created$, and define $delta\_terms\_created(H) = 0$ for all H.

Example 3. Consider the same program defined in Fig. 2 and the size relations computed in *Example* 2. We now show the corresponding resource usage equations for each clause for the resource *bits_received* (denoted by *bits* for brevity) inferred automatically by our system. Although the functions $\delta(ap, r)(H)$ and $\beta(ap, r)(L)$ take as arguments a clause head H and a body literal L respectively, in our examples we will only write the predicate name of H and L for the sake of brevity. Again, to simplify notation we will represent `exchange_buffer/3`, `exchange_byte/3` and `connect_socket/3` with *ex_buf* and *ex_byt* and *con_sock* respectively. Since the program is analyzed in a single traversal of the call graph in reverse topological order, the system starts by analyzing the predicate `exchange_buffer/3`. Note that the resource usage for external predicates (whose code is not available) `connect_socket/3`, `exchange_byte/3` and `close/1` is

already given by "trust" assertions which express that: $\texttt{RU}(con\_sock, \mathbf{ub}, bits, \_) = 0$, $\texttt{RU}(ex\_byt, \mathbf{ub}, bits, \_) = 8$ and $\texttt{RU}(close, \mathbf{ub}, bits, \_) = 0$.

For the recursive clause of `exchange_buffer/3`, the system sets up the following difference equation, where n represents the length of the first argument to this predicate (note that the system infers the "length" size metric for this argument and that $n > 0$):

$$\begin{aligned}\texttt{RU}(ex\_buf, \mathbf{ub}, bits, \langle n\rangle) = \;& \delta(ub, bits)(ex\_buf) + \beta(ub, bits)(ex\_byt) + \\ & \texttt{RU}(ex\_byt, \mathbf{ub}, bits, \_) + \beta(ub, bits)(ex\_buf) + \\ & \texttt{RU}(ex\_buf, \mathbf{ub}, bits, \langle n-1\rangle) = \\ & 0 + 0 + 8 + 0 + \texttt{RU}(ex\_buf, \mathbf{ub}, bits, \langle n-1\rangle)\end{aligned}$$

For the non-recursive clause of `exchange_buffer/3` the system infers: $\texttt{RU}(ex\_buf, \mathbf{ub}, bits, \langle 0\rangle) = 0$ which can be used as boundary condition for solving the previous difference equation, yielding the following closed form resource usage function: $\texttt{RU}(ex\_buf, \mathbf{ub}, bits, \langle n\rangle) = 8 \times n$.

Now, the `main/2` predicate is analyzed, and the system sets up the following expression for its only clause (where k is the length of the input buffer, i.e., the second argument to this predicate):

$$\begin{aligned}\texttt{RU}(main, \mathbf{ub}, bits, \langle k\rangle) = \;& \delta(ub, bits)(main) + \\ & \beta(ub, bits)(con\_sock) + \texttt{RU}(con\_sock, \mathbf{ub}, bits, \_) + \\ & \beta(ub, bits)(ex\_buf) + \texttt{RU}(ex\_buf, \mathbf{ub}, bits, \langle k\rangle) \\ & \beta(ub, bits)(close) + \texttt{RU}(close, \mathbf{ub}, bits, \_) \\ = \;& 0 + 0 + 0 + 0 + 8 \times k + 0 + 0 = 8 \times k\end{aligned}$$

7 Experimental Results

To study the feasibility of the approach we have completed a prototype implementation of the analyzer. It is written in the Ciao language and uses a number of modules and facilities from CiaoPP, the Ciao preprocessor (including difference equation processing). We have also written a Ciao language extension (a "package" in Ciao terminology) which when loaded into a module allows writing the resource-related assertions and declarations proposed herein.[4] We have then used this prototype to analyze a set of representative benchmarks which include definitions of resources using this language and used the system to infer the resource usage bound functions.

The results from the analysis of these benchmarks are shown in Table 1. For brevity, we report only results for upper-bounds analysis. The column labeled **Resource** shows the actual resource for which bounds are being inferred by the analysis for a given benchmark. While any of the resources defined in a given benchmark could then be used in any of the others we show only the results for the most natural or interesting resource for each one of them. We have tried to use a relatively wide range of resources: number of bytes sent by

[4] The system also supports adding resource assertions specifying expected resource usages which the implemented analyzer will then verify or falsify using the results of the implemented analysis.

Table 1. Accuracy and efficiency in milliseconds of the analysis

Program	Resource	Usage Function	Metrics	Time
`client`	"bits received"	$\lambda x.8 \cdot x$	length	186
`color_map`	"unifications"	39066	size	176
`copy_files`	"files left open"	$\lambda x.x$	length	180
`eight_queen`	"queens movements"	19173961	length	304
`eval_polynom`	"FPU usage"	$\lambda x.2.5x$	length	44
`fib`	"arithmetic operations"	$\lambda x.2.17 \cdot 1.61^x + 0.82 \cdot (-0.61)^x - 3$	value	116
`grammar`	"phrases"	24	length/size	227
`hanoi`	"disk movements"	$\lambda x.2^x - 1$	value	100
`insert_stores`	"accesses Stores" "insertions Stores"	$\lambda n, m.n + k$ $\lambda n, m.n$	length	292
`perm`	"WAM instructions"	$\lambda x.(\sum_{i=1}^{x} 18 \cdot x!) + (\sum_{i=1}^{x} 14 \cdot \frac{x!}{i}) + 4 \cdot x!$	length	98
`power_set`	"output elements"	$\lambda x.\frac{1}{2} \cdot 2^{x+1}$	length	119
`qsort`	"lists parallelized"	$\lambda x.4 \cdot 2^x - 2x - 4$	length	144
`send_files`	"bytes read"	$\lambda x, y.x \cdot y$	length / size	179
`subst_exp`	"replacements"	$\lambda x, y.2xy + 2y$	size / length	153
`zebra`	"resolution steps"	30232844295713061	size	292

an application, number of calls to a particular predicate, robot arm movements, number of files left open in a kernel code, number of accesses to a database, etc. The column **Usage Function** shows the actual resource usage function (which depends on the size of the input arguments) inferred by the analysis, given as a lambda term. The column **Metrics** shows the size metric used for the relevant arguments. Finally, the column labeled **Time** shows the resource analysis times in milliseconds, on a medium-loaded Pentium IV Xeon 2.0Ghz with two processors, 4Gb of RAM memory, running Fedora Core 5.0. Note that these times do not include other analyses such as types, modes, etc.

8 Conclusions

We have presented a static analysis that infers upper and lower bounds on the usage that a logic program makes of a quite general notion of user-definable resources. The inferred bounds are in general functions of input data sizes. We have also presented the assertion language which is used to define such resources. The analysis then derives the related (upper- and lower-bound) resource usage functions for all predicates in the program. Our preliminary experimental results are encouraging because they show that interesting resource bound functions can be obtained automatically and in reasonable time, at least for our benchmarks. While clearly further work is needed to assess scalability we are cautiously hopeful in the sense that our approach allows defining via assertions the resource usage of external predicates, which can then be used for modular composition.

These includes also predicates for which the code is not available or which are written in a programming language that is not supported by the analyzer. In addition, assertions also allow describing by hand the usage of any predicate for which the automatic analysis infers a value that is not accurate enough, and this can be used to prevent inaccuracies in the automatic inference from propagating. Our expectation is that the automatic analysis will be able to do the bulk of the work for large applications, even if the cost of some specially complex predicates may still need to be given by the user. In particular, for the examples in Table 1 all results where obtained automatically. Finally, we expect the applications of our analysis to be rather interesting, including resource consumption verification and debugging (including for mobile code), resource control in parallel/distributed computing, and resource-oriented specialization [6,18].

Acknowledgments. This work was funded in part by the Prince of Asturias Chair in Information Science and Technology at UNM, the IST programme of the European Commission, FET project FP6 IST-15905 *MOBIUS*, by Ministry of Education and Science (MEC) project TIN2005-09207-C03 *MERIT-COMVERS* and CAM project S-0505/TIC/0407 PROMESAS.

References

1. Aspinall, D., Beringer, L., Hofmann, M., Loidl, H.-W., Momigliano, A.: A program logic for resource verification. In: Slind, K., Bunker, A., Gopalakrishnan, G.C. (eds.) TPHOLs 2004. LNCS, vol. 3223, pp. 34–49. Springer, Heidelberg (2004)
2. Bagnara, R., Pescetti, A., Zaccagnini, A., Zaffanella, E., Zolo, T.: Purrs: The Parma University's Recurrence Relation Solver, `http://www.cs.unipr.it/purrs/`
3. Basin, D., Ganzinger, H.: Complexity Analysis based on Ordered Resolution. In: 11th. IEEE Symposium on Logic in Computer Science, IEEE Computer Society Press, Los Alamitos (1996)
4. Bate, I., Bernat, G., Puschner, P.: Java virtual-machine support for portable worst-case execution-time analysis. In: 5th IEEE Int'l. Symp. on Object-oriented Real-time Distributed Computing (April 2002)
5. Chander, A., Espinosa, D., Islam, N., Lee, P., Necula, G.C.: Enforcing resource bounds via static verification of dynamic checks. In: Sagiv, M. (ed.) ESOP 2005. LNCS, vol. 3444, pp. 311–325. Springer, Heidelberg (2005)
6. Craig, S.J., Leuschel, M.: Self-tuning resource aware specialisation for Prolog. In: Proc. of PPDP'05, pp. 23–34. ACM Press, New York (2005)
7. Debray, S.K., Lin, N.W.: Cost analysis of logic programs. TOPLAS, 15(5) (1993)
8. Debray, S.K., Lin, N.-W., Hermenegildo, M.: Task Granularity Analysis in Logic Programs. In: Proc. PLDI'90, pp. 174–188. ACM, New York (1990)
9. Debray, S.K., López-García, P., Hermenegildo, M., Lin, N.-W.: Lower Bound Cost Estimation for Logic Programs. In: Proc. ILPS'97, MIT Press, Cambridge (1997)
10. Eisinger, J., Polian, I., Becker, B., Metzner, A., Thesing, S., Wilhelm, R.: Automatic identification of timing anomalies for cycle-accurate worst-case execution time analysis. In: Proc. of DDECS, IEEE Computer Society Press, Los Alamitos (2006)
11. Grobauer, B.: Cost recurrences for DML programs. In: Int'l. Conf. on Functional Programming, pp. 253–264 (2001)

12. Hermenegildo, M., Puebla, G., Bueno, F., López García., P.: Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). Science of Computer Programming 58(1-2), 115–140 (2005)
13. Igarashi, A., Kobayashi, N.: Resource usage analysis. In: Symposium on Principles of Programming Languages, pp. 331–342 (2002)
14. López-García, P., Bueno, F., Hermenegildo, M.: Determinacy Analysis for Logic Programs Using Mode and Type Information. In: Bruynooghe, M. (ed.) Logic Based Program Synthesis and Transformation. LNCS, vol. 3018, pp. 19–35. Springer, Heidelberg (2004)
15. McAllester, D.A.: On the complexity analysis of static analyses. In: Static Analysis Symp., pp. 312–329 (1999)
16. Le Metayer, D.: ACE: An Automatic Complexity Evaluator. ACM Transactions on Programming Languages and Systems 10(2), 248–266 (1988)
17. Nielson, F., Nielson, H.R., Seidl, H.: Automatic complexity analysis. In: European Symposium on Programming, pp. 243–261 (2002)
18. Puebla, G., Ochoa, C.: Poly-Controlled Partial Evaluation. In: Proc. of PPDP'06, pp. 261–271. ACM Press, New York (2006)
19. Rosendhal, M.: Automatic Complexity Analysis. In: Proc. FPCA, ACM, New York (1989)
20. Thiele, L., Wilhelm, R.: Design for time-predictability. In: Perspectives Workshop: Design of Systems with Predictable Behaviour
21. Vasconcelos, P., Hammond, K.: Inferring Cost Equations for Recursive, Polymorphic and Higher-Order Functional Programs. In: Trinder, P., Michaelson, G.J., Peña, R. (eds.) IFL 2003. LNCS, vol. 3145, Springer, Heidelberg (2004)

Automatic Correctness Proofs for Logic Program Transformations

Alberto Pettorossi[1], Maurizio Proietti[2], and Valerio Senni[1]

[1] DISP, University of Roma Tor Vergata, Via del Politecnico 1, I-00133 Roma, Italy
{pettorossi,senni}@disp.uniroma2.it
[2] IASI-CNR, Viale Manzoni 30, I-00185 Roma, Italy
proietti@iasi.rm.cnr.it

Abstract. The many approaches which have been proposed in the literature for proving the correctness of unfold/fold program transformations, consist in associating suitable well-founded orderings with the proof trees of the atoms belonging to the least Herbrand models of the programs. In practice, these orderings are given by 'clause measures', that is, measures associated with the clauses of the programs to be transformed. In the unfold/fold transformation systems proposed so far, clause measures are fixed in advance, independently of the transformations to be proved correct. In this paper we propose a method for the automatic generation of the clause measures which, instead, takes into account the particular program transformation at hand. During the transformation process we construct a system of linear equations and inequations whose unknowns are the clause measures to be found, and the correctness of the transformation is guaranteed by the satisfiability of that system. Through some examples we show that our method is able to establish in a fully automatic way the correctness of program transformations which, by using other methods, are proved correct at the expense of fixing sophisticated clause measures.

1 Introduction

Rule-based program transformation is a program development methodology which consists in deriving from an initial program a final program, via the application of semantics preserving transformation rules [5]. In the field of logic (or functional) programming, program transformation can be regarded as a deductive process. Indeed, programs are logical (or equational, resp.) theories and the transformation rules can be viewed as rules for deducing new formulas from old ones. The logical soundness of the transformation rules easily implies that a transformation is *partially correct*, which means that an atom (or an equation, resp.) is true in the final program only if it is true in the initial program. However, it is usually much harder to prove that a transformation is *totally correct*, which means that an atom (or an equation, resp.) is true in the initial program if and only if it is true in the final program.

In the context of functional programming, it has been pointed out in the seminal paper by Burstall and Darlington [5] that, if the transformation rules

V. Dahl and I. Niemelä (Eds.): ICLP 2007, LNCS 4670, pp. 364–379, 2007.

rewrite the equations of the program at hand by using equations which belong to the same program (like the *folding* and *unfolding* rules), the transformations are always partially correct, but the final program may terminate (w.r.t. a suitable notion of termination) less often than the initial one. Thus, a sufficient condition for total correctness is that the final program obtained by transformation always terminates. This method of proving total correctness is sometimes referred to as *McCarthy's method* [13]. However, the termination condition may be, in practice, very hard to check.

The situation is similar in the case of definite logic programs, where the folding and unfolding rules basically consist in applying equivalences that hold in the least Herbrand model of the initial program. For instance, let us consider the program:

P: $p \leftarrow q$ $\qquad$ $r \leftarrow q$ $\qquad$ $q \leftarrow$

The least Herbrand model of P is $M(P) = \{p, q, r\}$ and $M(P) \models p \leftrightarrow q$. If we replace q by p in $r \leftarrow q$ (that is, we fold $r \leftarrow q$ using $p \leftarrow q$), then we get:

Q: $p \leftarrow q$ $\qquad$ $r \leftarrow p$ $\qquad$ $q \leftarrow$

The transformation of P into Q is totally correct, because $M(P) = M(Q)$. However, if we replace q by p in $p \leftarrow q$ (that is, we fold $p \leftarrow q$ using $p \leftarrow q$ itself), then we get:

R: $p \leftarrow p$ $\qquad$ $r \leftarrow q$ $\qquad$ $q \leftarrow$

and the transformation of P into R is partially correct, because $M(P) \supseteq M(R)$, but it is *not* totally correct, because $M(P) \neq M(R)$. Indeed, program R does not terminate for the goal p.

A lot of work has been devoted to devise methods for proving the total correctness of transformations based on various sets of rules, including the folding and the unfolding rules. These methods have been proposed both in the context of functional programming (see, for instance, [5,10,17]) and in the context of logic programming (see, for instance, [3,4,6,7,8,9,11,14,15,16,19,20,21]).

Some of these methods (such as, [3,5,6,11]) propose sufficient conditions for total correctness which are explicitly based on the preservation of suitable termination properties (such as, termination of call-by-name reduction for functional programs, and universal or existential termination for logic programs).

Other methods, which we may call *implicit methods*, are based on conditions on the sequence of applications of the transformation rules that guarantee that termination is preserved. A notable example of these implicit methods is presented in [9], where integer counters are associated with program clauses. The counters of the initial program are set to 1 and are incremented (or decremented) when an unfolding (or folding, resp.) is applied. A sequence of transformations is totally correct if the counters of the clauses of the final program are all positive.

The method based on counters allows us to prove the total correctness of many transformations. Unfortunately, there are also many simple derivations where the method fails to guarantee the total correctness. For instance, in the transformation from P to Q described above, we would get a value of 0 for the counter of the clause $r \leftarrow p$ in the final program Q, because it has been derived

by applying the folding rule from clause $r \leftarrow p$. Thus, the method does not yield the total correctness of the transformation. In order to overcome the limitations of the basic counter method, some modifications and enhancements have been described in [9,15,16,21], where each clause is given a *measure* which is more complex than an integer counter.

In this paper we present a different approach to the improvement of the basic counter method: instead of fixing *in advance* complex clause measures, for any given transformation we automatically generate, if at all possible, the clause measures that prove its correctness. For reasons of simplicity we assume that clause measures are non-negative integers, also called *weights*, and given a transformation starting from a program P, we look for a weight assignment to the clauses of P that proves that the transformation is totally correct.

Our paper is structured as follows. In Section 2 we present the notion of a *weighted transformation sequence*, that is, a sequence of programs constructed by applying suitable variants of the definition introduction, unfolding, and folding rules. We associate the clauses of the initial program of the sequence with some unknown weights, and during the construction of the sequence, we generate a set of constraints consisting of linear equations and inequations which relate those weights. If the final set of constraints is satisfiable for some assignment to the unknown weights, then the transformation sequence is totally correct. In Section 3 we prove our total correctness result which is based on the *well-founded annotations* method proposed in [14]. In Section 4 we consider transformation sequences constructed by using also the goal replacement rule and we present a method for proving the total correctness of those transformation sequences. Finally, in Section 5 we present a method for proving predicate properties which are needed for applying the goal replacement rule.

2 Weighted Unfold/Fold Transformation Rules

Let us begin by introducing some terminology concerning systems of linear equations and inequations with integer coefficients and non-negative integer solutions.

By $\mathcal{P}_{LIN}$ we denote the set of linear polynomials with integer coefficients. Variables occurring in polynomials are called *unknowns* to distinguish them from logical variables occurring in programs. By $\mathcal{C}_{LIN}$ we denote the set of linear equations and inequations with integer coefficients, that is, $\mathcal{C}_{LIN}$ is the set $\{p_1 = p_2, p_1 < p_2, p_1 \leq p_2 \mid p_1, p_2 \in \mathcal{P}_{LIN}\}$. By $p_1 \geq p_2$ we mean $p_2 \leq p_1$, and by $p_1 > p_2$ we mean $p_2 < p_1$. An element of $\mathcal{C}_{LIN}$ is called a *constraint*. A *valuation* for a set $\{u_1, \ldots, u_r\}$ of unknowns is a mapping $\sigma\colon \{u_1, \ldots, u_r\} \to \mathbb{N}$, where $\mathbb{N}$ is the set of natural numbers. Let $\{u_1, \ldots, u_r\}$ be the set of unknowns occurring in $p \in \mathcal{P}_{LIN}$. Given a valuation σ for (a superset of) $\{u_1, \ldots, u_r\}$, $\sigma(p)$ is the integer obtained by replacing the occurrences of $u_1, \ldots, u_r$ in p by $\sigma(u_1), \ldots, \sigma(u_r)$, respectively, and then computing the value of the resulting arithmetic expression. A valuation σ is a *solution* for the constraint $p_1 = p_2$ if σ is a valuation for a superset of the variables occurring in $p_1 = p_2$ and $\sigma(p_1) = \sigma(p_2)$ holds. Similarly, we define a solution for $p_1 < p_2$ and for $p_1 \leq p_2$. σ is a solution for a finite set $\mathcal{C}$ of constraints

if, for every $c \in \mathcal{C}$, σ is a solution for c. We say that a constraint c is *satisfiable* if there exists a solution for c. Similarly, we say that a set $\mathcal{C}$ of constraints is *satisfiable* if there exists a solution for $\mathcal{C}$. A *weight function* for a set S of clauses is a function $\gamma : S \to \mathcal{P}_{LIN}$. A value of γ is also called a *weight polynomial*.

A *weighted unfold/fold transformation sequence* is a sequence of programs, denoted $P_0 \mapsto P_1 \mapsto \cdots \mapsto P_n$, such that $n \geq 0$ and, for $k = 0, \ldots, n-1$, P_{k+1} is derived from P_k by applying one of the following transformation rules: *weighted definition introduction*, *weighted unfolding*, and *weighted folding*. These rules, which will be defined below, are variants of the familiar rules without weights. For reasons of simplicity, when referring to the transformation rules, we will often omit the qualification 'weighted'. For $k = 0, \ldots, n$, we will define: (i) a weight function $\gamma_k : P_k \to \mathcal{P}_{LIN}$, (ii) a finite set $\mathcal{C}_k$ of constraints, (iii) a set $Defs_k$ of clauses defining the new predicates introduced by the definition introduction rule during the construction of the sequence $P_0 \mapsto P_1 \mapsto \cdots \mapsto P_k$, and (iv) a weight function $\delta_k : P_0 \cup Defs_k \to \mathcal{P}_{LIN}$. The weight function γ_0 for the initial program P_0 is defined as follows: for every clause $C \in P_0$, $\gamma_0(C) = u$, where u is an unknown and, for each pair C and D of distinct clauses in P_0, we have that $\gamma_0(C) \neq \gamma_0(D)$. The initial sets $\mathcal{C}_0$ and $Defs_0$ are, by definition, equal to the empty set and $\delta_0 = \gamma_0$.

For every $k > 0$, we assume that P_0 and P_k have no variables in common. This assumption is not restrictive because we can always rename the variables occurring in a program without affecting its least Herbrand model. Indeed, in the sequel we will feel free to rename variables, whenever needed.

Rule 1 (Weighted Definition Introduction). Let $D_1, \ldots, D_m$, with $m > 0$, be clauses such that, for $i = 1, \ldots, m$, the predicate of the head of D_i does not occur in $P_0 \cup Defs_k$. By *definition introduction* from P_k we derive $P_{k+1} = P_k \cup \{D_1, \ldots, D_m\}$.

We set the following: (1.1) for all C in P_k, $\gamma_{k+1}(C) = \gamma_k(C)$, (1.2) for $i = 1, \ldots, m$, $\gamma_{k+1}(D_i) = u_i$, where u_i is a new unknown, (2) $\mathcal{C}_{k+1} = \mathcal{C}_k$, (3) $Defs_{k+1} = Defs_k \cup \{D_1, \ldots, D_m\}$, (4.1) for all D in $P_0 \cup Defs_k$, $\delta_{k+1}(D) = \delta_k(D)$, and (4.2) for $i = 1, \ldots, m$, $\delta_{k+1}(D_i) = u_i$.

Rule 2 (Weighted Unfolding). Let C: $H \leftarrow G_L \wedge A \wedge G_R$ be a clause in P_k and let C_1: $H_1 \leftarrow G_1$, ..., C_m: $H_m \leftarrow G_m$, with $m \geq 0$, be *all* clauses in P_0 such that, for $i = 1, \ldots, m$, A is unifiable with H_i via a most general unifier ϑ_i. By *unfolding* C w.r.t. A *using* $C_1, \ldots, C_m$, we derive the clauses D_1: $(H \leftarrow G_L \wedge G_1 \wedge G_R)\vartheta_1$, ..., D_m: $(H \leftarrow G_L \wedge G_m \wedge G_R)\vartheta_m$, and from P_k we derive $P_{k+1} = (P_k - \{C\}) \cup \{D_1, \ldots, D_m\}$.

We set the following: (1.1) for all D in $P_k - \{C\}$, $\gamma_{k+1}(D) = \gamma_k(D)$, (1.2) for $i = 1, \ldots, m$, $\gamma_{k+1}(D_i) = \gamma_k(C) + \gamma_0(C_i)$, (2) $\mathcal{C}_{k+1} = \mathcal{C}_k$, (3) $Defs_{k+1} = Defs_k$, and (4) $\delta_{k+1} = \delta_k$.

For a goal (or set of goals) G, by $vars(G)$ we denote the set of variables occurring in G.

Rule 3 (Weighted Folding). Let C_1: $H \leftarrow G_L \wedge G_1 \wedge G_R$, ..., C_m: $H \leftarrow G_L \wedge G_m \wedge G_R$ be clauses in P_k and let D_1: $K \leftarrow B_1$, ..., D_m: $K \leftarrow B_m$

be clauses in $P_0 \cup \mathit{Defs}_k$. Suppose that there exists a substitution ϑ such that the following conditions hold: (i) for $i = 1, \ldots, m$, $G_i = B_i\vartheta$, (ii) there exists no clause in $(P_0 \cup \mathit{Defs}_k) - \{D_1, \ldots, D_m\}$ whose head is unifiable with $K\vartheta$, and (iii) for $i = 1, \ldots, m$ and for every variable U in $\mathit{vars}(B_i) - \mathit{vars}(K)$: (iii.1) $U\vartheta$ is a variable not occurring in $\{H, G_L, G_R\}$, and (iii.2) $U\vartheta$ does not occur in the term $V\vartheta$, for any variable V occurring in B_i and different from U.

By *folding* $C_1, \ldots, C_m$ *using* $D_1, \ldots, D_m$, we derive E: $H \leftarrow G_L \wedge K\vartheta \wedge G_R$, and from P_k we derive $P_{k+1} = (P_k - \{C_1, \ldots, C_m\}) \cup \{E\}$.

We set the following: (1.1) for all C in $P_k - \{C_1, \ldots, C_m\}$, $\gamma_{k+1}(C) = \gamma_k(C)$, (1.2) $\gamma_{k+1}(E) = u$, where u is a new unknown, (2) $\mathcal{C}_{k+1} = \mathcal{C}_k \cup \{u \leq \gamma_k(C_1) - \delta_k(D_1), \ldots, u \leq \gamma_k(C_m) - \delta_k(D_m)\}$, (3) $\mathit{Defs}_{k+1} = \mathit{Defs}_k$, and (4) $\delta_{k+1} = \delta_k$.

The *correctness constraint system* associated with a weighted unfold/fold transformation sequence $P_0 \mapsto \cdots \mapsto P_n$ is the set $\mathcal{C}_{\mathit{final}}$ of constraints defined as follows:

$\mathcal{C}_{\mathit{final}} = \mathcal{C}_n \cup \{\gamma_n(C) \geq 1 \mid C \in P_n\}$.

The following result, which will be proved in Section 3, guarantees the total correctness of weighted unfold/fold transformations. By $M(P)$ we denote the least Herbrand model of program P.

Theorem 1 (Total Correctness of Weighted Unfold/Fold Transformations). *Let $P_0 \mapsto \cdots \mapsto P_n$ be a weighted unfold/fold transformation sequence constructed by using Rules* 1–3*, and let $\mathcal{C}_{\mathit{final}}$ be its associated correctness constraint system. If $\mathcal{C}_{\mathit{final}}$ is satisfiable then $M(P_0 \cup \mathit{Defs}_n) = M(P_n)$.*

Example 1. (*Continuation Passing Style Transformation*) Let us consider the initial program P_0 consisting of the following three clauses whose weight polynomials are the unknowns u_1, u_2, and u_3, respectively (we write weight polynomials on a second column to the right of the corresponding clause):

1. $p \leftarrow$ u_1
2. $p \leftarrow p \wedge q$ u_2
3. $q \leftarrow$ u_3

We want to derive a continuation-passing-style program defining a predicate p_{cont} equivalent to the predicate p defined by the program P_0. In order to do so, we introduce by Rule 1 the following clause 4 with its unknown u_4:

4. $p_{\mathit{cont}} \leftarrow p$ u_4

and also the following three clauses for the unary continuation predicate *cont* with unknowns u_5, u_6, and u_7, respectively:

5. $\mathit{cont}(f_{\mathit{true}}) \leftarrow$ u_5
6. $\mathit{cont}(f_p(X)) \leftarrow p \wedge \mathit{cont}(X)$ u_6
7. $\mathit{cont}(f_q(X)) \leftarrow q \wedge \mathit{cont}(X)$ u_7

where f_{true}, f_p, and f_q are three function symbols corresponding to the three predicates *true*, p, and q, respectively. By folding clause 4 using clause 5 we get the following clause with the unknown u_8 which should satisfy the constraint

$u_8 \leq u_4 - u_5$ (we write constraints on a third column to the right of the corresponding clause):

8. $p_{cont} \leftarrow p \wedge cont(f_{true})$ $\quad u_8 \quad u_8 \leq u_4 - u_5$

By folding clause 8 using clause 6 we get the following clause 9 with unknown u_9 such that $u_9 \leq u_8 - u_6$:

$(*)$ 9. $p_{cont} \leftarrow cont(f_p(f_{true}))$ $\quad u_9 \quad u_9 \leq u_8 - u_6$

By unfolding clause 6 w.r.t. p using clauses 1 and 2, we get:

$(*)$ 10. $cont(f_p(X)) \leftarrow cont(X)$ $\quad u_6 + u_1$

11. $cont(f_p(X)) \leftarrow p \wedge q \wedge cont(X)$ $\quad u_6 + u_2$

Then by folding clause 11 using clause 7 we get:

12. $cont(f_p(X)) \leftarrow p \wedge cont(f_q(X))$ $\quad u_{12} \quad u_{12} \leq u_6 + u_2 - u_7$

and by folding clause 12 using clause 6 we get:

$(*)$ 13. $cont(f_p(X)) \leftarrow cont(f_p(f_q(X)))$ $\quad u_{13} \quad u_{13} \leq u_{12} - u_6$

Finally, by unfolding clause 7 w.r.t. q we get:

$(*)$ 14. $cont(f_q(X)) \leftarrow cont(X)$ $\quad u_7 + u_3$

The final program is made out of clauses 9, 10, 13, and 14, marked with $(*)$, and clauses 1, 2, and 3. The correctness constraint system $\mathcal{C}_{final}$ is made out of the following 11 constraints.

For clauses 9, 10, 13, and 14: $u_9 \geq 1$, $u_6 + u_1 \geq 1$, $u_{13} \geq 1$, $u_7 + u_3 \geq 1$.

For clauses 1, 2, and 3: $u_1 \geq 1$, $u_2 \geq 1$, $u_3 \geq 1$.

For the four folding steps: $u_8 \leq u_4 - u_5$, $u_9 \leq u_8 - u_6$, $u_{12} \leq u_6 + u_2 - u_7$, $u_{13} \leq u_{12} - u_6$.

This system $\mathcal{C}_{final}$ of constraints is satisfiable and thus, the transformation from program P_0 to the final program is totally correct.

3 Proving Correctness Via Weighted Programs

In order to prove that a weighted unfold/fold transformation sequence $P_0 \mapsto \cdots \mapsto P_n$ is *totally correct* (see Theorem 1), we specialize the method based on *well-founded annotations* proposed in [14]. In particular, with each program P_k in the transformation sequence, we associate a *weighted program* $\overline{P}_k$ by adding an integer argument $n(\geq 0)$, called a *weight*, to each atom $p(\boldsymbol{t})$ occurring in P_k. Here and in the sequel, $\boldsymbol{t}$ denotes a generic m-tuple of terms $t_1, \ldots, t_m$, for some $m \geq 0$. Informally, $p(\boldsymbol{t}, n)$ holds in $\overline{P}_k$ if $p(\boldsymbol{t})$ 'has a proof of weight at least n' in P_k. We will show that if the correctness constraint system $\mathcal{C}_{final}$ is satisfiable, then it is possible to derive from $\overline{P}_0$ a weighted program $\overline{P}_n$ where the weight arguments determine, for every clause $\overline{C}$ in $\overline{P}_n$, a well-founded ordering between the head of $\overline{C}$ and every atom in the body $\overline{C}$. Hence $\overline{P}_n$ terminates for all ground goals (even if P_n need not) and the immediate consequence operator $T_{\overline{P}}$ has a unique fixpoint [2]. Thus, as proved in [14], the total correctness of the transformation sequence follows from the *unique fixpoint principle* (see Corollary 1).

Our transformation rules can be regarded as rules for replacing a set of clauses by an equivalent one. Let us introduce the notions of implication and equivalence between sets of clauses according to [14].

Definition 1. Let I be an Herbrand interpretation and let Γ_1 and Γ_2 be two sets of clauses. We write $I \models \Gamma_1 \Rightarrow \Gamma_2$ if for every ground instance $H \leftarrow G_2$ of a clause in Γ_2 such that $I \models G_2$ there exists a ground instance $H \leftarrow G_1$ of a clause in Γ_1 such that $I \models G_1$. We write $I \models \Gamma_1 \Leftarrow \Gamma_2$ if $I \models \Gamma_2 \Rightarrow \Gamma_1$, and we write $I \models \Gamma_1 \Leftrightarrow \Gamma_2$ if ($I \models \Gamma_1 \Rightarrow \Gamma_2$ and $I \models \Gamma_1 \Leftarrow \Gamma_2$).

For all Herbrand interpretations I and sets of clauses Γ_1, Γ_2, and Γ_3 the following properties hold:

Reflexivity: $I \models \Gamma_1 \Rightarrow \Gamma_1$
Transitivity: if $I \models \Gamma_1 \Rightarrow \Gamma_2$ and $I \models \Gamma_2 \Rightarrow \Gamma_3$ then $I \models \Gamma_1 \Rightarrow \Gamma_3$
Monotonicity: if $I \models \Gamma_1 \Rightarrow \Gamma_2$ then $I \models \Gamma_1 \cup \Gamma_3 \Rightarrow \Gamma_2 \cup \Gamma_3$.

Given a program P, we denote its associated *immediate consequence operator* by T_P [1,12]. We denote the least and greatest fixpoint of T_P by $lfp(T_P)$ and $gfp(T_P)$, respectively. Recall that $M(P) = lfp(T_P)$.

Now let us consider the transformation of a program P into a program Q consisting in the replacement of a set Γ_1 of clauses in P by a new set Γ_2 of clauses. The following result, proved in [14], expresses the *partial correctness* of the transformation of P into Q.

Theorem 2 (Partial Correctness). *Given two programs P and Q, such that*: *(i) for some sets Γ_1 and Γ_2 of clauses, $Q = (P - \Gamma_1) \cup \Gamma_2$, and (ii) $M(P) \models \Gamma_1 \Rightarrow \Gamma_2$. Then $M(P) \supseteq M(Q)$.*

In order to establish a sufficient condition for the total correctness of the transformation of P into Q, that is, $M(P) = M(Q)$, we consider programs whose associated immediate consequence operators have unique fixpoints.

Definition 2 (Univocal Program). *A program P is said to be* univocal *if T_P has a unique fixpoint, that is, $lfp(T_P) = gfp(T_P)$.*

The following theorem is proved in [14].

Theorem 3 (Conservativity). *Given two programs P and Q, such that*: *(i) for some sets Γ_1 and Γ_2 of clauses, $Q = (P - \Gamma_1) \cup \Gamma_2$, and (ii) $M(P) \models \Gamma_1 \Leftarrow \Gamma_2$, and (iii) Q is univocal. Then $M(P) \subseteq M(Q)$.*

As a straightforward consequence of Theorems 2 and 3 we get the following.

Corollary 1 (Total Correctness Via Unique Fixpoint). *Given two programs P and Q such that*: *(i) for some sets Γ_1, Γ_2 of clauses, $Q = (P - \Gamma_1) \cup \Gamma_2$, (ii) $M(P) \models \Gamma_1 \Leftrightarrow \Gamma_2$, and (iii) Q is univocal. Then $M(P) = M(Q)$.*

Corollary 1 cannot be directly applied to prove the total correctness of a transformation sequence generated by applying the unfolding and folding rules, because

the programs derived by these rules need not be univocal. To overcome this difficulty we introduce the notion of weighted program.

Given a clause C of the form $p_0(\boldsymbol{t_0}) \leftarrow p_1(\boldsymbol{t_1}) \wedge \ldots \wedge p_m(\boldsymbol{t_m})$, where $\boldsymbol{t_0}, \boldsymbol{t_1}, \ldots, \boldsymbol{t_m}$ are tuples of terms, a *weighted clause*, denoted $\overline{C}(w)$, associated with C is a clause of the form:

$\overline{C}(w)$: $p_0(\boldsymbol{t_0}, N_0) \leftarrow N_0 \geq N_1 + \cdots + N_m + w \wedge p_1(\boldsymbol{t_1}, N_1) \wedge \ldots \wedge p_m(\boldsymbol{t_m}, N_m)$

where w is a natural number called the *weight* of $\overline{C}(w)$. Clause $\overline{C}(w)$ is denoted by $\overline{C}$ when we do not need refer to the weight w. A *weighted program* is a set of weighted clauses. Given a program $P = \{C_1, \ldots, C_r\}$, by $\overline{P}$ we denote a weighted program of the form $\{\overline{C}_1, \ldots, \overline{C}_r\}$. Given a weight function γ and a valuation σ, by $\overline{C}(\gamma, \sigma)$ we denote the weighted clause $\overline{C}(\sigma(\gamma(C)))$ and by $\overline{P}(\gamma, \sigma)$ we denote the weighted program $\{\overline{C}_1(\gamma, \sigma), \ldots, \overline{C}_r(\gamma, \sigma)\}$.

For reasons of conciseness, we do not formally define here when a formula of the form $N_0 \geq N_1 + \cdots + N_m + w$ (see clause $\overline{C}(w)$ above) holds in an interpretation, and we simply say that for every Herbrand interpretation I and ground terms $n_0, n_1, \ldots, n_m, w$, we have that $I \models n_0 \geq n_1 + \cdots + n_m + w$ holds iff $n_0, n_1, \ldots, n_m, w$ are (terms representing) natural numbers such that n_0 is greater than or equal to $n_1 + \cdots + n_m + w$.

The following lemma (proved in [14]) establishes the relationship between the semantics of a program P and the semantics of any weighted program $\overline{P}$ associated with P.

Lemma 1. *Let P be a program. For every ground atom $p(\boldsymbol{t})$, $p(\boldsymbol{t}) \in M(P)$ iff there exists $n \in \mathbb{N}$ such that $p(\boldsymbol{t}, n) \in M(\overline{P})$.*

By erasing weights from clauses we preserve clause implications, in the sense stated by the following lemma (proved in [14]).

Lemma 2. *Let P be a program, and Γ_1 and Γ_2 be any two sets of clauses. If $M(\overline{P}) \models \overline{\Gamma}_1 \Rightarrow \overline{\Gamma}_2$ then $M(P) \models \Gamma_1 \Rightarrow \Gamma_2$.*

A weighted program $\overline{P}$ is said to be *decreasing* if every clause in $\overline{P}$ has a positive weight.

Lemma 3. *Every decreasing program is univocal.*

Now, we have the following result, which is a consequence of Lemmata 1, 3, and Theorems 2 and 3. Unlike Corollary 1, this result can be used to prove the total correctness of the transformation of program P into program Q also in the case where Q is not univocal.

Theorem 4 (Total Correctness Via Weights). *Let P and Q be programs such that: (i) $M(P) \models P \Rightarrow Q$, (ii) $M(\overline{P}) \models \overline{P} \Leftarrow \overline{Q}$, and (iii) $\overline{Q}$ is decreasing. Then $M(P) = M(Q)$.*

By Theorem 4, in order to prove Theorem 1, that is, the total correctness of weighted unfold/fold transformations, it is enough to show that, given a weighted unfold/fold transformation sequence $P_0 \mapsto \cdots \mapsto P_n$, we have that:

(P1) $M(P_0 \cup Defs_n) \models P_0 \cup Defs_n \Rightarrow P_n$

and there exist suitable weighted programs $\overline{P}_0 \cup \overline{Defs}_n$ and $\overline{P}_n$, associated with $P_0 \cup Defs_n$ and P_n, respectively, such that:

(P2) $M(\overline{P}_0 \cup \overline{Defs}_n) \models \overline{P}_0 \cup \overline{Defs}_n \Leftarrow \overline{P}_n$, and

(P3) $\overline{P}_n$ is decreasing.

The suitable weighted programs $\overline{P}_0 \cup \overline{Defs}_n$ and $\overline{P}_n$ are constructed as we now indicate by using the hypothesis that the correctness constraint system $\mathcal{C}_{final}$ associated with the transformation sequence, is satisfiable. Let σ be a solution for $\mathcal{C}_{final}$. For every $k = 0, \ldots, n$ and for every clause $C \in P_k$, we take $\overline{C} = \overline{C}(\gamma_k, \sigma)$, where γ_k is the weight function associated with P_k. For $C \in Defs_k$ we take $\overline{C} = \overline{C}(\delta_k, \sigma)$. Thus, $\overline{P}_k = \overline{P}_k(\gamma_k, \sigma)$ and $\overline{Defs}_k = \overline{Defs}_k(\delta_k, \sigma)$.

In order to prove Theorem 1 we need the following two lemmata.

Lemma 4. *Let $P_0 \mapsto \cdots \mapsto P_k$ be a weighted unfold/fold transformation sequence. Let C be a clause in P_k, and let $D_1, \ldots, D_m$ be the clauses derived by* unfolding *C w.r.t. an atom in its body, as described in Rule* 2. *Then*:

$M(\overline{P}_0 \cup \overline{Defs}_n) \models \{\overline{C}\} \Leftrightarrow \{\overline{D}_1, \ldots, \overline{D}_m\}$

Lemma 5. *Let $P_0 \mapsto \cdots \mapsto P_k$ be a weighted unfold/fold transformation sequence. Let $C_1, \ldots, C_m$ be clauses in P_k, $D_1, \ldots, D_m$ be clauses in $P_0 \cup Defs_k$, and E be the clause derived by* folding *$C_1, \ldots, C_m$ using $D_1, \ldots, D_m$, as described in Rule* 3. *Then*:

(*i*) $M(P_0 \cup Defs_n) \models \{C_1, \ldots, C_m\} \Rightarrow \{E\}$

(*ii*) $M(\overline{P}_0 \cup \overline{Defs}_n) \models \{\overline{C}_1, \ldots, \overline{C}_m\} \Leftarrow \{\overline{E}\}$

We are now able to prove Theorem 1. For a weighted unfold/fold transformation sequence $P_0 \mapsto \cdots \mapsto P_n$, the following properties hold:

(R1) $M(P_0 \cup Defs_n) \models P_k \cup (Defs_n - Defs_k) \Rightarrow P_{k+1} \cup (Defs_n - Defs_{k+1})$, and

(R2) $M(\overline{P}_0 \cup \overline{Defs}_n) \models \overline{P}_k \cup (\overline{Defs}_n - \overline{Defs}_k) \Leftarrow \overline{P}_{k+1} \cup (\overline{Defs}_n - \overline{Defs}_{k+1})$.

Indeed, Properties (R1) and (R2) can be proved by reasoning by cases on the transformation rule applied to derive P_{k+1} from P_k, as follows. If P_{k+1} is derived from P_k by applying the definition introduction rule then $P_k \cup (Defs_n - Defs_k) = P_{k+1} \cup (Defs_n - Defs_{k+1})$ and, therefore, Properties (R1) and (R2) are trivially true. If P_{k+1} is derived from P_k by applying the unfolding rule, then $P_{k+1} = (P_k - \{C\}) \cup \{D_1, \ldots, D_m\}$ and $Defs_k = Defs_{k+1}$. Hence, Properties (R1) and (R2) follow from Lemma 2, Lemma 4 and from the monotonicity of $\Rightarrow$. If P_{k+1} is derived from P_k by applying the folding rule, then $P_{k+1} = (P_k - \{C_1, \ldots, C_m\}) \cup \{E\}$ and $Defs_k = Defs_{k+1}$. Hence, Properties (R1) and (R2) follow from Points (i) and (ii) of Lemma 5 and the monotonicity of $\Rightarrow$.

By the transitivity of $\Rightarrow$ and by Properties (R1) and (R2), we get Properties (P1) and (P2). Moreover, since σ is a solution for $\mathcal{C}_{final}$ and $\overline{P}_n = \overline{P}_n(\gamma_n, \sigma)$, Property (P3) holds. Thus, by Theorem 4, $M(P_0 \cup Defs_n) = M(P_n)$.

4 Weighted Goal Replacement

In this section we extend the notion of a weighted unfold/fold transformation sequence $P_0 \mapsto P_1 \mapsto \cdots \mapsto P_n$ by assuming that P_{k+1} is derived from P_k by applying, besides the definition introduction, unfolding, and folding rules, also the goal replacement rule defined as Rule 4 below. The goal replacement rule consists in replacing a goal G_1 occurring in the body of a clause of P_k, by a new goal G_2 such that G_1 and G_2 are equivalent in $M(P_0 \cup \mathit{Defs}_k)$. Some conditions are also needed in order to update the value of the weight function and the associated constraints. To define these conditions we introduce the notion of *weighted replacement law* (see Definition 3), which in turn is based on the notion of weighted program introduced in Section 3.

In Definition 3 below we will use the following notation. Given a goal $G : p_1(\boldsymbol{t_1}) \wedge \ldots \wedge p_m(\boldsymbol{t_m})$, a variable N, and a natural number w, by $\overline{G}[N, w]$ we denote the formula $\exists N_1 \ldots \exists N_m (N \geq N_1 + \cdots + N_m + w \wedge p_1(\boldsymbol{t_1}, N_1) \wedge \ldots \wedge p_m(\boldsymbol{t_m}, N_m))$. Given a set $X = \{X_1, \ldots, X_m\}$ of variables, we will use '$\exists X$' as a shorthand for '$\exists X_1 \ldots \exists X_m$' and '$\forall X$' as a shorthand for '$\forall X_1 \ldots \forall X_m$'.

Definition 3 (Weighted Replacement Law). Let P be a program, γ be a weight function for P, and $\mathcal{C}$ be a finite set of constraints. Let G_1 and G_2 be goals, u_1 and u_2 be unknowns, and $X \subseteq \mathit{vars}(G_1) \cup \mathit{vars}(G_2)$ be a set of variables. We say that the *weighted replacement law* $(G_1, u_1) \Rightarrow_X (G_2, u_2)$ holds with respect to the triple $\langle P, \gamma, \mathcal{C} \rangle$, and we write $\langle P, \gamma, \mathcal{C} \rangle \models (G_1, u_1) \Rightarrow_X (G_2, u_2)$, if the following conditions hold:

(i) $M(P) \models \forall X \, (\exists Y \, G_1 \leftarrow \exists Z \, G_2)$, and

(ii) for every solution σ for $\mathcal{C}$,

$M(\overline{P}) \models \forall X \forall U \, (\exists Y \, (\overline{G}_1[U, \sigma(u_1)]) \rightarrow \exists Z \, (\overline{G}_2[U, \sigma(u_2)]))$

where: (1) $\overline{P}$ is the weighted program $\overline{P}(\gamma, \sigma)$, (2) U is a variable, (3) $Y = \mathit{vars}(G_1) - X$, and (4) $Z = \mathit{vars}(G_2) - X$.

By using Lemma 2 it can be shown that, if $\mathcal{C}$ is satisfiable, then Condition (ii) of Definition 3 implies $M(P) \models \forall X \, (\exists Y \, G_1 \rightarrow \exists Z \, G_2)$ and, therefore, if $\langle P, \gamma, \mathcal{C} \rangle \models (G_1, u_1) \Rightarrow_X (G_2, u_2)$ and $\mathcal{C}$ is satisfiable, we also have that $M(P) \models \forall X \, (\exists Y \, G_1 \leftrightarrow \exists Z \, G_2)$.

Example 2. (*Associativity of List Concatenation*) Let us consider the following program *Append* for list concatenation. To the right of each clause we indicate the corresponding unknown.

1. $a([\,], L, L)$ $\qquad u_1$
2. $a([H|T], L, [H|R]) \leftarrow a(T, L, R)$ $\qquad u_2$

The following replacement law expresses the associativity of list concatenation:

Law (α): $(a(L_1, L_2, M) \wedge a(M, L_3, L), \; w_1)$
$\Rightarrow_{\{L_1, L_2, L_3, L\}} \quad (a(L_2, L_3, R) \wedge a(L_1, R, L), \; w_2)$

where w_1 and w_2 are new unknowns. In Example 4 below we will show that Law (α) holds w.r.t. $\langle \mathit{Append}, \gamma, \mathcal{C} \rangle$, where $\mathcal{C}$ is the set of constraints $\{u_1 \geq 1, \; u_2 \geq 1, \; w_1 \geq w_2, \; w_2 + u_1 \geq 1\}$.

In Section 5 we will present a method, called *weighted unfold/fold proof method*, for generating a suitable set $\mathcal{C}$ of constraints such that $\langle P, \gamma, \mathcal{C}\rangle \models (G_1, u_1) \Rightarrow_X (G_2, u_2)$ holds. Now we introduce the Weighted Goal Replacement Rule, which is a variant of the rule without weights (see, for instance, [20]).

Rule 4 (Weighted Goal Replacement). Let C: $H \leftarrow G_L \wedge G_1 \wedge G_R$ be a clause in program P_k and let $\mathcal{C}$ be a set of constraints such that the weighted replacement law $\lambda : (G_1, u_1) \Rightarrow_X (G_2, u_2)$ holds w.r.t. $\langle P_0 \cup \mathit{Defs}_k, \delta_k, \mathcal{C}\rangle$, where $X = \mathit{vars}(\{H, G_L, G_R\}) \cap \mathit{vars}(\{G_1, G_2\})$.

By applying the replacement law λ, from C we derive $D : H \leftarrow G_L \wedge G_2 \wedge G_R$, and from P_k we derive $P_{k+1} = (P_k - \{C\}) \cup \{D\}$. We set the following: (1.1) for all E in $P_k - \{C\}$, $\gamma_{k+1}(E) = \gamma_k(E)$, (1.2) $\gamma_{k+1}(D) = \gamma_k(C) - u_1 + u_2$, (2) $\mathcal{C}_{k+1} = \mathcal{C}_k \cup \mathcal{C}$, (3) $\mathit{Defs}_{k+1} = \mathit{Defs}_k$, and (4) $\delta_{k+1} = \delta_k$.

The proof of the following result is similar to the one of Theorem 1.

Theorem 5 (Total Correctness of Weighted Unfold/Fold/Replacement Transformations). *Let $P_0 \mapsto \cdots \mapsto P_n$ be a weighted unfold/fold transformation sequence constructed by using Rules 1–4, and let $\mathcal{C}_{final}$ be its associated correctness constraint system. If $\mathcal{C}_{final}$ is satisfiable then $M(P_0 \cup \mathit{Defs}_n) = M(P_n)$.*

Example 3. (*List Reversal*) Let *Reverse* be a program for list reversal consisting of the clauses of *Append* (see Example 2) together with the following two clauses (to the right of the clauses we write the corresponding weight polynomials):

3. $r([\,],[\,]) \leftarrow$ u_3
4. $r([H|T], L) \leftarrow r(T, R) \wedge a(R, [H], L)$ u_4

We will transform the *Reverse* program into a program that uses an accumulator [5]. In order to do so, we introduce by Rule 1 the following clause:

5. $g(L_1, L_2, A) \leftarrow r(L_1, R) \wedge a(R, A, L_2)$ u_5

We apply the unfolding rule twice starting from clause 5 and we get:

6. $g([\,], L, L) \leftarrow$ $u_5 + u_3 + u_1$
7. $g([H|T], L, A) \leftarrow r(T, R) \wedge a(R, [H], S) \wedge a(S, A, L)$ $u_5 + u_4$

By applying the replacement law (α), from clause 7 we derive:

8. $g([H|T],L,A) \leftarrow r(T,R) \wedge a([H],A,S) \wedge a(R,S,L)$ $u_5 + u_4 - w_1 + w_2$

together with the constraints (see Example 2): $u_1 \geq 1$, $u_2 \geq 1$, $w_1 \geq w_2$, $w_2 + u_1 \geq 1$. By two applications of the unfolding rule, from clause 8 we get:

9. $g([H|T], L, A) \leftarrow r(T, R) \wedge a(R, [H|A], L)$ $u_5 + u_4 - w_1 + w_2 + u_2 + u_1$

By folding clause 9 using clause 5 we get:

10. $g([H|T], L, A) \leftarrow g(T, L, [H|A])$ u_6

together with the constraint $u_6 \leq u_4 - w_1 + w_2 + u_2 + u_1$.
Finally, by folding clause 4 using clause 5 we get:

11. $r([H|T], L) \leftarrow g(T, L, [H])$ u_7

together with the constraint $u_7 \leq u_4 - u_5$.

The final program consists of clauses 1, 2, 3, 11, 6, and 10. The correctness constraint system associated with the transformation sequence is as follows.

For clauses 1, 2, and 3: $u_1 \geq 1,\ u_2 \geq 1,\ u_3 \geq 1.$
For clauses 11, 6, and 10: $u_7 \geq 1,\ u_5 + u_3 + u_1 \geq 1,\ u_6 \geq 1.$
For the goal replacement: $u_1 \geq 1,\ u_2 \geq 1,\ w_1 \geq w_2,\ w_2 + u_1 \geq 1.$
For the two folding steps: $u_6 \leq u_4 - w_1 + w_2 + u_2 + u_1,\ u_7 \leq u_4 - u_5.$

This set of constraints is satisfiable and, therefore, the transformation sequence is totally correct.

5 The Weighted Unfold/Fold Proof Method

In this section we present the unfold/fold method for proving the replacement laws to be used in Rule 4. In order to do so, we introduce the notions of: (i) *syntactic equivalence*, (ii) *symmetric folding*, and (iii) *symmetric goal replacement*.

A *predicate renaming* is a bijective mapping $\rho : \mathit{Preds}_1 \rightarrow \mathit{Preds}_2$, where Preds_1 and Preds_2 are two sets of predicate symbols. Given a formula (or a set of formulas) F, by $\mathit{preds}(F)$ we denote the set of predicate symbols occurring in F. Suppose that $\mathit{preds}(F) \subseteq \mathit{Preds}_1$, then by $\rho(F)$ we denote the formula obtained from F by replacing every predicate symbol p by $\rho(p)$. Two programs Q and R are *syntactically equivalent* if there exists a predicate renaming $\rho : \mathit{preds}(Q) \rightarrow \mathit{preds}(R)$, such that $R = \rho(Q)$, modulo variable renaming.

An application of the folding rule by which from program P_k we derive program P_{k+1}, is said to be *symmetric* if $\mathcal{C}_{k+1}$ is set to $\mathcal{C}_k \cup \{u = \gamma_k(C_1) - \delta_k(D_1), \ldots, u = \gamma_k(C_m) - \delta_k(D_m)\}$ (see Point 2 of Rule 3).

Given a program P, a weight function γ, and a set $\mathcal{C}$ of constraints, we say that the replacement law $(G_1, u_1) \Rightarrow_X (G_2, u_2)$ holds *symmetrically* w.r.t. $\langle P, \gamma, \mathcal{C}\rangle$, and we write $\langle P, \gamma, \mathcal{C}\rangle \models (G_1, u_1) \Leftrightarrow_X (G_2, u_2)$, if the following condition holds:

(ii*) for every solution σ for $\mathcal{C}$,
$M(\overline{P}) \models \forall X \forall U\, (\exists Y\, (\overline{G}_1[U, \sigma(u_1)]) \leftrightarrow \exists Z\, (\overline{G}_2[U, \sigma(u_2)]))$

where $\overline{P}$, U, Y, and Z are defined as in Definition 3. Note that, by Lemma 2, Condition (ii*) implies Condition (i) of Definition 3. An application of the *goal replacement rule* is *symmetric* if it consists in applying a replacement law that holds symmetrically w.r.t. $\langle P_0 \cup \mathit{Defs}_k, \delta_k, \mathcal{C}\rangle$. A weighted unfold/fold transformation sequence is said to be *symmetric* if it is constructed by applications of the definition and unfolding rules and by symmetric applications of the folding and goal replacement rules.

Now we are ready to present the weighted unfold/fold proof method, which is itself based on weighted unfold/fold transformations.

The Weighted Unfold/Fold Proof Method. Let us consider a program P, a weight function γ for P, and a replacement law $(G_1, u_1) \Rightarrow_X (G_2, u_2)$. Suppose that X is the set of variables $\{X_1, \ldots, X_m\}$ and let $\boldsymbol{X}$ denote the sequence $X_1, \ldots, X_m$.

Step 1. First we introduce two new predicates $\mathit{new}1$ and $\mathit{new}2$ defined by the following two clauses: D_1: $\mathit{new}1(\boldsymbol{X}) \leftarrow G_1$ and D_2: $\mathit{new}2(\boldsymbol{X}) \leftarrow G_2$, associated with the unknowns u_1 and u_2, respectively.

Step 2. Then we construct two weighted unfold/fold transformation sequences of the forms: $P \cup \{D_1\} \mapsto \cdots \mapsto Q$ and $P \cup \{D_2\} \mapsto \cdots \mapsto R$, such that the following three conditions hold:

(1) For $i = 1, 2$, the weight function associated with the initial program $P \cup \{D_i\}$ is γ_0^i defined as: $\gamma_0^i(C) = \gamma(C)$ if $C \in P$, and $\gamma_0^i(D_i) = u_i$;
(2) The final programs Q and R are syntactically equivalent; and
(3) The transformation sequence $P \cup \{D_2\} \mapsto \cdots \mapsto R$ is symmetric.

Step 3. Finally, we construct a set $\mathcal{C}$ of constraints as follows. Let γ_Q and γ_R be the weight functions associated with Q and R, respectively. Let $\mathcal{C}_Q$ and $\mathcal{C}_R$ be the correctness constraint systems associated with the transformation sequences $P \cup \{D_1\} \mapsto \cdots \mapsto Q$ and $P \cup \{D_2\} \mapsto \cdots \mapsto R$, respectively, and let ρ be the predicate renaming such that $\rho(Q) = R$. Suppose that both $\mathcal{C}_Q$ and $\mathcal{C}_R$ are satisfiable.

(3.1) Let the set $\mathcal{C}$ be $\{\gamma_Q(C) \geq \gamma_R(\rho(C)) \mid C \in Q\} \cup \mathcal{C}_Q \cup \mathcal{C}_R$. Then we infer:

$\langle P, \gamma, \mathcal{C} \rangle \vdash_{UF} (G_1, u_1) \Rightarrow_X (G_2, u_2)$

(3.2) Suppose that the transformation sequence $P \cup \{D_1\} \mapsto \cdots \mapsto Q$ is symmetric and let the set $\mathcal{C}$ be $\{\gamma_Q(C) = \gamma_R(\rho(C)) \mid C \in Q\} \cup \mathcal{C}_Q \cup \mathcal{C}_R$. Then we infer:

$\langle P, \gamma, \mathcal{C} \rangle \vdash_{UF} (G_1, u_1) \Leftrightarrow_X (G_2, u_2)$

It can be shown that the unfold/fold proof method is sound.

Theorem 6 (Soundness of the Unfold/Fold Proof Method)
If $\langle P, \gamma, \mathcal{C} \rangle \vdash_{UF} (G_1, u_1) \Rightarrow_X (G_2, u_2)$ *then* $\langle P, \gamma, \mathcal{C} \rangle \models (G_1, u_1) \Rightarrow_X (G_2, u_2)$.
If $\langle P, \gamma, \mathcal{C} \rangle \vdash_{UF} (G_1, u_1) \Leftrightarrow_X (G_2, u_2)$ *then* $\langle P, \gamma, \mathcal{C} \rangle \models (G_1, u_1) \Leftrightarrow_X (G_2, u_2)$.

Example 4. (*An Unfold/Fold Proof*) Let us consider again the program *Append* and the replacement law (α), expressing the associativity of list concatenation, presented in Example 2. By applying the unfold/fold proof method we will generate a set $\mathcal{C}$ of constraints such that law (α) holds w.r.t. $\langle Append, \gamma, \mathcal{C} \rangle$.

Step 1. We start off by introducing the following two clauses:

D_1. $new_1(L_1, L_2, L_3, L) \leftarrow a(L_1, L_2, M) \wedge a(M, L_3, L)$ $\quad w_1$
D_2. $new_2(L_1, L_2, L_3, L) \leftarrow a(L_2, L_3, R) \wedge a(L_1, R, L)$ $\quad w_2$

First, let us construct a transformation sequence starting from $Append \cup \{D_1\}$. By two applications of the unfolding rule, from clause D_1 we derive:

E_1. $new1([\,], L_2, L_3, L) \leftarrow a(L2, L3, L)$ $\quad w_1 + u_1$
E_2. $new1([H|T], L_2, L_3, [H|R]) \leftarrow a(T, L_2, M) \wedge a(M, L_3, R)$ $\quad w_1 + 2u_2$

By folding clause E_2 using clause D_1 we derive:

E_3. $new1([H|T], L_2, L_3, [H|R]) \leftarrow new1(T, L_2, L_3, R)$ $\quad u_8$

together with the constraint $u_8 \leq 2u_2$.

Now, let us construct a transformation sequence starting from $Append \cup \{D_2\}$. By unfolding clause D_2 w.r.t. $a(L_1, R, L)$ in its body we get:

F_1. $new2([\,], L_2, L_3, L) \leftarrow a(L_2, L_3, L)$ $\quad w_2 + u_1$
F_2. $new2([H|T], L_2, L_3, [H|R]) \leftarrow a(L_2, L_3, M) \wedge a(T, M, R)$ $\quad w_2 + u_2$

By a symmetric application of the folding rule using clause D_2, from clause F_2 we get:

F_3. $new2([H|T], L_2, L_3, [H|R]) \leftarrow new2(T, L_2, L_3, R)$ $\quad u_9$

together with the constraint $u_9 = u_2$.

The final programs $Append \cup \{E_1, E_3\}$ and $Append \cup \{F_1, F_3\}$ are syntactically equivalent via the predicate renaming ρ such that $\rho(new1) = new2$. The transformation sequence $Append \cup \{D_2\} \mapsto \cdots \mapsto Append \cup \{F_1, F_3\}$ is symmetric.

Step 3. The correctness constraint system associated with the transformation sequence $Append \cup \{D_1\} \mapsto \cdots \mapsto Append \cup \{E_1, E_3\}$ is the following:

$\mathcal{C}_1$: $\{u_1 \geq 1,\ u_2 \geq 1,\ w_1 + u_1 \geq 1,\ u_8 \geq 1,\ u_8 \leq 2u_2\}$

The correctness constraint system associated with the transformation sequence $Append \cup \{D_2\} \mapsto \cdots \mapsto Append \cup \{F_1, F_3\}$ is the following:

$\mathcal{C}_2$: $\{u_1 \geq 1,\ u_2 \geq 1,\ w_2 + u_1 \geq 1,\ u_9 \geq 1,\ u_9 = u_2\}$

Both $\mathcal{C}_1$ and $\mathcal{C}_2$ are satisfiable and thus, we infer:

$\langle Append, \gamma, \mathcal{C}_{12} \rangle \vdash_{UF}$

$(a(L_1, L_2, M) \wedge a(M, L_3, L),\ w_1) \Rightarrow_{\{L_1, L_2, L_3, L\}} (a(L_2, L_3, R) \wedge a(L_1, R, L),\ w_2)$

where $\mathcal{C}_{12}$ is the set $\{w_1 + u_1 \geq w_2 + u_1,\ u_8 \geq u_9\} \cup \mathcal{C}_1 \cup \mathcal{C}_2$.

Notice that the constraints $w_1 + u_1 \geq w_2 + u_1$ and $u_8 \geq u_9$ are determined by the two pairs of syntactically equivalent clauses (E_1, F_1) and (E_3, F_3), respectively. By eliminating the unknowns u_8 and u_9, which occur in the proof of law (α) only, and by performing some simple simplifications we get, as anticipated, the following set $\mathcal{C}$ of constraints: $\{u_1 \geq 1,\ u_2 \geq 1,\ w_1 \geq w_2,\ w_2 + u_1 \geq 1\}$.

6 Conclusions

We have presented a method for proving the correctness of rule-based logic program transformations in an automatic way. Given a transformation sequence, constructed by using the unfold, fold, and goal replacement transformation rules, we associate some unknown natural numbers, called weights, with the clauses of the programs in the transformation sequence and we also construct a set of linear constraints that these weights must satisfy to guarantee the total correctness of the transformation sequence. Thus, the correctness of the transformation sequence can be proven in an automatic way by checking that the corresponding set of constraints is satisfiable over the natural numbers. However, it can be shown that our method is incomplete and, more in general, it can be shown that there exists no algorithmic method for checking whether or not any unfold/fold transformation sequence is totally correct.

As already mentioned in the Introduction, our method is related to the many methods given in the literature for proving the correctness of program transformation by showing that suitable conditions on the transformation sequence hold (see, for instance, [4,8,9,16,20,21], for the case of definite logic programs). Among these methods, the one presented in [16] is the most general and it makes use of *clause measures* to express complex conditions on the transformation sequence.

The main novelty of our method with respect to [16] is that in [16] clause measures are fixed in advance, independently of the specific transformation sequence under consideration, while by the method proposed in this paper we automatically generate specific clause measures for each transformation sequence to be proved correct.

Thus, in principle, our method is more powerful than the one presented in [16]. For a more accurate comparison between the two methods, we did some practical experiments. We implemented our method in the MAP transformation system (`http://www.iasi.cnr.it/~proietti/system.html`) and we worked out some transformation examples taken from the literature. Our system runs on SICStus Prolog (v. 3.12.5) and for the satisfiability of the sets of constraints over the natural numbers it uses the *clpq* SICStus library.

By using our system we did the transformation examples presented in this paper (see Examples 1, 3, and 4) and the following examples taken from the literature: (i) the *Adjacent* program which checks whether or not two elements have adjacent occurrences in a list [9], (ii) the *Equal Frontier* program which checks whether or not the frontiers of two binary trees are equal [5,21], (iii) a program for solving the N-queens problem [18], (iv) the *In_Correct_Position* program taken from [8], and (v) the program that encodes a liveness property of an n-bit shift register [16]. Even in the most complex derivation we carried out, that is, the *Equal Frontier* example taken from [21], consisting of 86 transformation steps, the system checked the total correctness of the transformation within milliseconds. For making that derivation we also had to apply several replacement laws which were proved correct by using the unfold/fold proof method described in Section 5.

Acknowledgements

We thank the anonymous referees for constructive comments.

References

1. Apt, K.R.: Introduction to logic programming. In: van Leeuwen, J. (ed.) Handbook of Theoretical Computer Science, pp. 493–576. Elsevier, Amsterdam (1990)
2. Bezem, M.: Characterizing termination of logic programs with level mappings. In: Proc. of NACLP, Cleveland, Ohio (USA), pp. 69–80. MIT Press, Cambridge (1989)
3. Bossi, A., Cocco, N.: Preserving universal termination through unfold/fold. In: Rodríguez-Artalejo, M., Levi, G. (eds.) ALP 1994. LNCS, vol. 850, pp. 269–286. Springer, Heidelberg (1994)
4. Bossi, A., Cocco, N., Etalle, S.: On safe folding. In: Bruynooghe, M., Wirsing, M. (eds.) PLILP 1992. LNCS, vol. 631, pp. 172–186. Springer, Heidelberg (1992)
5. Burstall, R.M., Darlington, J.: A transformation system for developing recursive programs. Journal of the ACM 24(1), 44–67 (1977)
6. Cook, J., Gallagher, J.P.: A transformation system for definite programs based on termination analysis. In: Fribourg, L., Turini, F. (eds.) LOPSTR 1994 and META 1994. LNCS, vol. 883, pp. 51–68. Springer, Heidelberg (1994)

7. Etalle, S., Gabbrielli, M.: Transformations of CLP modules. Theoretical Computer Science 166, 101–146 (1996)
8. Gergatsoulis, M., Katzouraki, M.: Unfold/fold transformations for definite clause programs. In: Penjam, J. (ed.) PLILP 1994. LNCS, vol. 844, pp. 340–354. Springer, Heidelberg (1994)
9. Kanamori, T., Fujita, H.: Unfold/fold transformation of logic programs with counters. Technical Report 179, ICOT, Tokyo, Japan (1986)
10. Kott, L.: About transformation system: A theoretical study. In: 3ème Colloque International sur la Programmation, Paris (France), Dunod, pp. 232–247 (1978)
11. Lau, K.-K., Ornaghi, M., Pettorossi, A., Proietti, M.: Correctness of logic program transformation based on existential termination. In: Lloyd, J.W. (ed.) Proceedings of ILPS '95, pp. 480–494. MIT Press, Cambridge (1995)
12. Lloyd, J.W.: Foundations of Logic Programming, 2nd edn. Springer, Heidelberg (1987)
13. McCarthy, J.: Towards a mathematical science of computation. In: Proceedings of IFIP 1962, Amsterdam, pp. 21–28. North Holland (1963)
14. Pettorossi, A., Proietti, M.: A theory of totally correct logic program transformations. In: Proceedings of PEPM '04, pp. 159–168. ACM Press, New York (2004)
15. Roychoudhury, A., Narayan Kumar, K., Ramakrishnan, C.R., Ramakrishnan, I.V.: Beyond Tamaki-Sato style unfold/fold transformations for normal logic programs. Int. Journal on Foundations of Computer Science 13(3), 387–403 (2002)
16. Roychoudhury, A., Narayan Kumar, K., Ramakrishnan, C.R., Ramakrishnan, I.V.: An unfold/fold transformation framework for definite logic programs. ACM Transactions on Programming Languages and Systems 26, 264–509 (2004)
17. Sands, D.: Total correctness by local improvement in the transformation of functional programs. ACM Toplas 18(2), 175–234 (1996)
18. Sato, T., Tamaki, H.: Examples of logic program transformation and synthesis. Unpublished manuscript (1985)
19. Seki, H.: Unfold/fold transformation of stratified programs. Theoretical Computer Science 86, 107–139 (1991)
20. Tamaki, H., Sato, T.: Unfold/fold transformation of logic programs. In: Proceedings of ICLP '84, Uppsala, Sweden, pp. 127–138. Uppsala University (1984)
21. Tamaki, H., Sato, T.: A generalized correctness proof of the unfold/fold logic program transformation. Technical Report 86-4, Ibaraki University, Japan (1986)

Core TuLiP
Logic Programming for Trust Management*

Marcin Czenko[1] and Sandro Etalle[1,2]

[1] Department of Computer Science
University of Twente, The Netherlands
[2] University of Trento, Italy
{marcin.czenko,sandro.etalle}@utwente.nl

Abstract. We propose CoreTuLiP - the core of a trust management language based on Logic Programming. CoreTuLiP is based on a subset of moded logic programming, but enjoys the features of TM languages such as RT; in particular clauses are issued by different authorities and stored in a distributed manner. We present a lookup and inference algorithm which we prove to be correct and complete w.r.t. the declarative semantics. CoreTuLiP enjoys uniform syntax and the well-established semantics and is expressive enough to model scenarios which are hard to deal with in RT.

1 Introduction

Trust management (TM) [6,8,13,15] is an approach to access control in decentralised distributed systems where access control decisions are based on policy statements issued by multiple principals, and stored in a distributed manner. Policy statements are often digitally signed to ensure their authenticity and integrity; such statements are sometimes called *credentials* or *certificates*, and – in many cases – can be represented as datalog clauses. Indeed, like in logic programming, in TM credentials often have to be combined together to provide an authorisation *proof* (e.g. a proof that a given user has indeed access to a given resource).

One of the features of TM advocated in e.g. [16] (w.r.t. classical decentralised access control, but also w.r.t. logic programming) is distributed storage: in practice credentials may or may not be stored by the authority who *issues* them; therefore one of the prominent problems of TM is that of guaranteeing that – under reasonable circumstances – if there exists a proof of a certain (authorisation) statement, then it is also possible to *find* the credentials needed to construct the proof itself.

To date, one of the most successful TM systems is the RT family, defined by Li, Winsborough and Mitchell [15,16]. This family of languages enjoys a well-defined LP-based declarative semantics, a syntax similar to that of SDSI [8], and offers the possibility of storing credentials either by the *issuer* (the authority issuing them) and/or by the *subject* (the entity the credential "refers to"). The location where the credential is stored is determined by the so-called *type* of the credential. Li et al. show that if all

* This work was supported by the projects: Freeband I-Share, EU-NoE-ARTIST2, and EU-IST-IP-SERENITY (contract N 27587). Permanent address of both authors: University of Twente.

V. Dahl and I. Niemelä (Eds.): ICLP 2007, LNCS 4670, pp. 380–394, 2007.

credentials are *well-typed* then there exists a terminating credential chain discovery algorithm which ines whether a given statement is valid in the present state.

Although the RT family is successful in achieving its goals, we believe that it presents drawbacks which are worth investigating and improving. In particular, RT syntax is inflexible to the extent that to accommodate natural things such as separation of duty etc., one has to resort to a number of rather artificial extensions (RT_1 until RT^D, and RT^T), which are difficult to grasp and use. Secondly, it cannot be linked naturally with external languages. Finally, while it enjoys a declarative reading, this reading does not reflect the crucial type information.

One could speculate that to solve these problems one should simply translate RT into Logic Programming, and then use the latter to specify and prove authorisation statements. This is however inaccurate, as this translation would lose one of the essential elements that make RT a *trust management* language, in particular the information concerning where credentials should be stored and how they can be found when needed.

In this paper we present *CoreTuLiP*, which is the stripped-down version of the TuLiP (Trust management system based on Logic Programming) system we are developing at the University of Twente in the context of the I-Share project [12]. CoreTuLiP is basically a subset of (function-free) moded logic programming, with the essential additional feature that the clauses are not stored at a central authority, but are distributed across the different principals involved in the system. The mode information determines *where* a clause will be stored and a form of *well-modedness* is used to guarantee that, as the computation progresses, enough information is available to *find* the clauses needed to build a proof of the query being evaluated. Since credentials are distributed, CoreTuLiP is not amenable to SLD resolution, and requires a mix of top-down and bottom up reasoning. Here, we present a terminating algorithm which is able to answer well-moded queries, together with the soundness and completeness result. Finally, we show that RT_0, the core language of the RT family, is basically equivalent to a subset of CoreTuLiP. Doing so, we prove that it is possible to define a true trust management language which is as expressive as RT_0 also in terms of credential distribution without giving up the established LP formalism.

CoreTuLiP, is also based on LP, but has a more flexible underlying syntax than RT, and can easily accommodate extensions. For instance, it allows one to express thresholds and separation of duties, which require special additions to RT_0.

This paper is structured as follows: in Sect. 2 we introduce the basics of moded Logic Programming. In Sect. 3 we introduce CoreTuLiP. In Sect. 4 we present the Lookup and Inference AlgoRithm, and we show that it is sound and complete w.r.t. the standard LP semantics. In Sect. 5 we compare RT_0 with CoreTuLiP. Finally, in Sect. 6 we present the related work and then we conclude the paper and propose future research in Sect. 7.

2 Preliminaries on Logic Programs

The reader is assumed to be familiar with the terminology and the basic results of the semantics of logic programs [1,17]. Here, we refer to *function-free* (Datalog-like) logic programs and we adopt the notation of [1]. We denote atoms by $A, B, H, \ldots$, queries by $\mathbf{A}, \mathbf{B}, \mathbf{C}, \ldots$ (following [1], queries are simply conjunctions of atoms, possibly empty), clauses by $c, d, \ldots$, and programs by P. The empty query is denoted by $\Box$. For any

syntactic object (e.g. atom, clause, query) o, we denote by $Var(o)$ the set of variables occurring in o. Given a *substitution* $\sigma = \{x_1/t_1, ..., x_n/t_n\}$ we say that $\{x_1, \ldots, x_n\}$ is its *domain* (denoted by $Dom(\sigma)$) and that $Var(\{t_1, ..., t_n\})$ is its *range* (denoted by $Ran(\sigma)$). Further, we denote by $Var(\sigma) = Dom(\sigma) \cup Ran(\sigma)$. If $t_1, ..., t_n$ is a permutation of $x_1, ..., x_n$ then we say that σ is a *renaming*. The *composition* of substitutions is denoted by juxtaposition ($\theta\sigma(X) = \sigma(\theta(X))$). We say that a syntactic object (e.g. an atom) o is an *instance* of o' iff for some σ, $o = o'\sigma$; o is called a *variant* of o', written $o \approx o'$ iff o and o' are instances of each other. A substitution θ is a *unifier* of objects o and o' iff $o\theta = o'\theta$. We denote by $mgu(o, o')$ any *most general unifier* (*mgu*, in short) of o and o'. Computations are sequences of derivation steps. The non-empty query $q : \mathbf{A}, B, \mathbf{C}$ and a clause $c : H \leftarrow \mathbf{B}$ (renamed apart w.r.t. q) yield the resolvent $(\mathbf{A}, \mathbf{B}, \mathbf{C})\theta$, provided that $\theta = mgu(B, H)$. A *derivation step* is denoted by $\mathbf{A}, B, \mathbf{C} \overset{\theta}{\Longrightarrow}_{P,c} (\mathbf{A}, \mathbf{B}, \mathbf{C})\theta$. c is called its *input clause*, and B is called the *selected atom* of q. A derivation is obtained by iterating derivation steps. A maximal sequence $\delta := \mathbf{B}_0 \overset{\theta_1}{\Longrightarrow}_{P,c_1} \mathbf{B}_1 \overset{\theta_2}{\Longrightarrow}_{P,c_2} \cdots \mathbf{B}_n \overset{\theta_{n+1}}{\Longrightarrow}_{P,c_{n+1}} \mathbf{B}_{n+1} \cdots$ of derivation steps is called an *SLD derivation of* $P \cup \{\mathbf{B}_0\}$ provided that for every step the standardisation apart condition holds, i.e. the input clause employed at each step is variable disjoint from the initial query $\mathbf{B}_0$, and from the substitutions and the input clauses used at earlier steps. If δ is maximal and ends with the empty query ($\mathbf{B}_n = \square$) then the restriction of θ to the variables of $\mathbf{B}$ is called its *computed answer substitution* (*c.a.s.*, for short).

Moded Programs. Informally speaking, a *mode* indicates how the arguments of a relation should be used, i.e. which are the input and which are the output positions of each atom, and allows one to derive properties such as absence of run-time errors for Prolog built-ins and absence of floundering for programs with negation [3]. Most compilers encourage the user to specify a mode declaration.

Definition 1 (Mode). *Consider an n-ary predicate symbol p. By a* mode *for p we mean a function m_p from $\{1, \ldots, n\}$ to $\{In, Out\}$.*

If $m_p(i) = In$ (resp. *Out*), we say that i is an *input* (resp. *output*) *position of* p (with respect to m_p). We assume that each predicate symbol has a *unique mode* associated to it; multiple modes may be obtained by simply renaming the predicates. We use the notation $(X_1, \ldots, X_n)$ to indicate the mode m in which $m(i) = X_i$. For instance, (In, Out) indicates the mode in which the first (resp. second) position is an input (resp. output) position. To benefit from the advantage of modes, programs are required to be *well-moded* [3]: they have to respect some correctness conditions relating the input arguments to the output ones. We denote by $In(A)$ (resp. $Out(A)$) the sequence of terms filling in the input (resp. output) positions of A, and by $VarIn(A)$ (resp. $VarOut(A)$) the set of variables occupying the input (resp. output) positions of A.

Definition 2 (Well-Moded). *A clause $H \leftarrow B_1, \ldots, B_n$ is* well-moded *if* $\forall\, i \in [1, n]$

$$VarIn(B_i) \subseteq \bigcup_{j=1}^{i-1} VarOut(B_j) \cup VarIn(H), \text{ and}$$
$$VarOut(H) \subseteq \bigcup_{j=1}^{n} VarOut(B_j) \cup VarIn(H).$$

A query $\mathbf{A}$ is well-moded *iff the clause $H \leftarrow \mathbf{A}$ is well-moded, where H is any (dummy) atom of zero arity. A program is* well-moded *if all of its clauses are well-moded.*

Note that the first atom of a well-moded query is ground in its input positions and a variant of a well-moded clause is well-moded. The following Lemma, due to [2], shows the "persistence" of the notion of well-modedness.

Lemma 1. *An SLD-resolvent of a well-moded query and a well-moded clause that is variable-disjoint with it, is well-moded.* □

3 Core TuLiP

We now introduce CoreTuLiP, which in first approximation is a variant of moded LP. In CoreTuLiP, there are two disjoint types of predicates: (user-defined) *credential predicates* and built-in *constraint predicates*. In CoreTuLiP,

- credential predicates have arity two;
- in an atom with a credential predicate, we call the term filling in the first argument position the *issuer*, and the one filling in the second argument position the *recipient*.
- a *credential* is a clause defining credential predicate. Then, the *issuer* of a credential is the term filling in the first argument position of the head.

(the relations with the notions of issuer and subject used in RT are discussed in Sect. 5). In the full version we are going to have credential predicates with more arguments and user defined predicates as well. These additions are, however, immaterial for this paper.

Example 1. To access a project document at the University of Twente (UT) one must be either a project member and a Ph.D. student at the UT or at one of the partner universities, or be approved by two different assistant professors from the UT. John and Jeroen are assistant professors at the UT. John states that a project member from one of the partner universities can access the document if she is approved by at least one project member who is also an associate professor at that university. Jeroen approves anyone who is also approved by a project leader at the UT. Sandro is a project leader at the UT.

(c_1) $\mathit{access\_document}(\mathit{ut}, X) :-$
$\mathit{project\_member}(\mathit{ut}, X),$
$\mathit{prof}(\mathit{ut}, A_1),$
$\mathit{prof}(\mathit{ut}, A_2),$
$A_1 \neq A_2,$
$\mathit{approve\_access}(A_1, X),$
$\mathit{approve\_access}(A_2, X).$

(c_2) $\mathit{access\_document}(\mathit{ut}, X) :-$
$\mathit{phd\_student}(P, X),$
$\mathit{project\_partner}(\mathit{ut}, P),$
$\mathit{project\_member}(P, X).$

(c_3) $\mathit{approve\_access}(\mathit{john}, X) :-$
$\mathit{approve\_access}(A, X),$
$\mathit{associate\_prof}(P, A),$
$\mathit{project\_partner}(\mathit{ut}, P),$
$\mathit{project\_member}(P, A),$
$\mathit{project\_member}(P, X).$

(c_4) $\mathit{approve\_access}(\mathit{jeroen}, X) :-$
$\mathit{approve\_access}(L, X),$
$\mathit{project\_leader}(\mathit{ut}, L).$

(c_5) $\mathit{project\_leader}(\mathit{ut}, \mathit{sandro}).$

(c_6) $\mathit{phd\_student}(\mathit{ut}, \mathit{marcin}).$

(c_7) $\mathit{approve\_access}(\mathit{sandro}, \mathit{rico}).$
$\mathit{approve\_access}(\mathit{jeffrey}, \mathit{rico}).$

(c_8) $\mathit{associate\_prof}(\mathit{tud}, \mathit{jeffrey}).$

(c_9) $\mathit{project\_member}(\mathit{ut}, \mathit{john}).$
$\mathit{project\_member}(\mathit{ut}, \mathit{charles}).$
$\mathit{prof}(\mathit{ut}, \mathit{john}).$
$\mathit{prof}(\mathit{ut}, \mathit{jeroen}).$
$\mathit{project\_partner}(\mathit{ut}, \mathit{ut}).$

(c_{10}) $\mathit{project\_partner}(\mathit{ut}, \mathit{tud}).$
$\mathit{project\_member}(\mathit{tud}, \mathit{jeffrey}).$
$\mathit{project\_member}(\mathit{tud}, \mathit{rico}).$

In TM, credentials are always *issued* by some authority (for the sake of simplicity here we identify authorities with the set of ground terms). In Example 1, the credential *prof*(*ut*, *john*) is *issued* by *ut* (University of Twente), and has *john* as the recipient. With this credential, *ut* states that *john* is one of the professors at the University of Twente. In a practical setting, this credential is signed by *ut*, and *ut* and *john* are placeholders for the implementation dependent identifiers (like public keys or URIs). Under these assumptions, it is natural to expect that the issuer of a credential should be a ground term.

Definition 3. *Let* $cl : H \leftarrow B_1, \ldots, B_n$ *be a clause. We say that* cl *is* well-formed *if it is well-moded and* issuer(H) *is a ground term.*

Modes and Decentralised Storage. Credential predicates have three *legal modes*: (In, In), (In, Out), and (Out, In). The reason why the mode (Out, Out) is considered illegal is that it would allow queries with completely uninstantiated arguments like *prof*(X, Y), in which neither the issuer nor the recipient is specified. Unlike in LP, such queries cannot be answered in a TM system because the system does not know where to look for relevant credentials, which could be issued and stored by any authority. By requiring that at least one of the arguments be input, and that the credentials be *traceable* (see below) we will be able to find the credentials we need to construct the proofs we need. For constraint predicates the only legal mode is the one "all-input" $(In, \ldots, In)$.

A peculiar feature of trust management systems is that credentials are stored in a distributed way. For instance, in Example 1, the credential *prof*(*ut*, *john*) which is issued by *ut* could be stored by either *ut* or *john*. Storing it by *john* has the advantage that *john* does not have to fetch the credential at *ut* every time he needs it, which in a highly distributed system may be costly. We call the *depositary* of a credential the authority where the credential is stored. In CoreTuLiP, it is the mode of the credential's head which determines its depositary (here, we allow only one mode per relation symbol, so credentials will be stored at one place only; by allowing multiple modes we lift this limitation in the extended system). Returning to Example 1, if *mode*(*prof*) is either (In, In) or (In, Out), then the credential *prof*(*ut*, *john*) will be stored at *ut*, otherwise (if the mode is (Out, In)), *john* will store it. Storing the credential at some other place would make it unfindable. The definition below generalises this concept.

Definition 4 (Traceable, Depositary). *We say that a clause* $cl : H \leftarrow B_1, \ldots, B_n$ *is* traceable *if it is well-formed and one of the following conditions holds:*

1. mode$(H) \in \{(In, In), (In, Out)\}$ *– in this case* issuer(H) *is the* depositary *of the rule,*
2. mode$(H) = (Out, In)$*, and* recipient$(H)(= In(H))$ *contains a ground term - in this case* recipient(H) *is the* depositary *of the rule,*
3. mode$(H) = (Out, In)$ *and* recipient(H) *contains a variable. In this case we require that there exists a prefix* $B_1, \ldots, B_k$ *of the body such that*
   - mode$(B_1) = \ldots =$ mode$(B_k) = (Out, In)$,
   - $In(H) = In(B_1)$,
   - $In(B_{i+1}) = Out(B_i)$*, and is a variable, for* $i \in [1, k-1]$,
   - $Out(B_k)$ *contains a ground term,*

In this case, we say that issuer$(B_k)(= Out(B_k))$ *is the* depositary *of the rule.*

The third case is complex, but it has the advantage of permitting the storage of a credential at a third party (neither the issuer, nor the recipient). Consider again Example 1, with the following mode assignments: *access_document*:(In, In), *project_member*:(In, In), *prof*:(In, Out), and the remaining predicates are moded (Out, In). In this case credentials $c_1 - c_4$ and c_9 are stored at *ut*. Credential c_5 is stored by *sandro*, c_6 by *marcin*, c_7 by *rico*, and c_8 by *jeffrey*. *tud* stores all the credentials c_{10}.

We can now introduce the concept of a state.

Definition 5. *A* state $\mathcal{P}$ *is a finite collection of pairs* (a, P_a) *where* P_a *is a collection of traceable credentials and* a *is the depositary of these credentials.*

The declarative semantics of a state is simply given in terms of logic programming as follows (where for simplicity we assume that all constraints are user-defined)

Definition 6. *Let* $\mathcal{P}$ *be the state* $\{(a_1, P_1), \ldots, (a_n, P_n)\}$*, and* A *be an atom*

- *We denote by* $P(\mathcal{P})$ *the set of clauses* $P_1 \cup \cdots \cup P_n$*. We call* $P(\mathcal{P})$ *the LP-counterpart of state* $\mathcal{P}$*.*
- *We say that* A *is true in state* $\mathcal{P}$ *iff* $P(\mathcal{P}) \cup \mathcal{C} \models A$*, where* $\mathcal{C}$ *is a first order theory determining the meaning of credential predicates.*

4 The Lookup and Inference AlgoRithm (LIAR)

The goal of an authorisation system is to check whether a fact is true in a given state. Since the state $\mathcal{P}$ can be very large and distributed across different agents, it is essential to have an algorithm which takes care of computing whether a given query is true in $\mathcal{P}$ without having to collect the entire $P(\mathcal{P})$. An extra difficulty comes from the fact that clauses might easily be mutually recursive, and that cases 2 and 3 of Definition 4 make it impossible to follow a straightforward top-down reasoning. In this section we present a suitable algorithm. Before we proceed we need the following definitions.

Definition 7 (Connected). *We say that two atoms* A *and* B *that have mode* (Out, In) *are* connected *if* recipient(A) *is ground and* recipient(A) = recipient(B).

Let A be an atom and S be a set of atoms. We adopt the following conventions:

(i) We write $A \mathrel{\widetilde{\in}} S$ iff $\exists A' \in S$, such that $A' \approx A$ (i.e. A' is a renaming of A).
(ii) We write $A \mathrel{\widetilde{\notin}} S$ iff $\nexists A' \in S$ such that $A' \approx A$.
(iii) We write $A \overset{\theta}{\hookrightarrow} S$ iff $\exists A' \mathrel{\widetilde{\in}} S$ standardised apart w.r.t. A such that $\gamma = mgu(A, A')$ and $A\theta \approx A\gamma$.

Definition 8. *Let* A *be an atomic well-moded query. We define the Lookup and Inference AlgoRithm (LIAR) which given a state* $\mathcal{P}$ *and a query* A *as an input returns the (possibly empty) sets of atoms* FACTSTACK *and* GOALSTACK*. The algorithm is reported in Fig. 1.*

INPUT : A. /* A is the initial atomic query */
Init:
CLSTACK : $\{A \leftarrow A\}$;
FACTSTACK = GOALSTACK = $\boxed{\text{VISITED}} = \emptyset$;
SATISFIED = **FALSE** ;
REPEAT
Phase 1 (Top-down resolution):
CHOOSE:
c : $H \leftarrow \mathbf{B}, C, \mathbf{D} \in$ CLSTACK and
$\mathbf{B}' \subseteq$ FACTSTACK, such that the following conditions hold:
(i) $\mathbf{B}$ and $\mathbf{B}'$ unify with mgu θ,
(ii) $C\theta$ is well-moded,
(iii) $C\theta \overset{\sim}{\notin}$ GOALSTACK,
$\boxed{\text{(iv) } \textbf{IF } mode(C) = (Out, In) \textbf{ THEN } recipient(C\theta) \notin \text{VISITED } \textbf{ENDIF}}$
ADD $C\theta$ to GOALSTACK;
IF *mode*$(C) \in \{(In, Out), (In, In)\}$ **THEN**
FETCH at *issuer*$(C\theta)$ all clauses $\{c_1, \ldots, c_n\}$ whose head unifies with $C\theta$ with mgus $\{\gamma_1, \ldots, \gamma_n\}$ respectively ;
FOR EACH $c_i\gamma_i \in \{c_1\gamma_1, \ldots, c_n\gamma_n\}$**DO**
IF $c_i\gamma_i \overset{\sim}{\notin}$ CLSTACK **THEN** ADD $c_i\gamma_i$ to CLSTACK **ENDIF**
END FOR EACH
ELSEIF *mode*$(C) = (Out, In)$ **THEN**
FETCH all clauses $\{c_1, \ldots, c_n\}$ stored at *recipient*$(C\theta)$ whose head has mode (Out, In) ;
$\boxed{\text{ADD } \textit{recipient}(C\theta) \text{ to VISITED;}}$
FOR EACH $c_i \in \{c_1, \ldots, c_n\}$ **DO**
IF $c_i \overset{\sim}{\notin}$ CLSTACK **THEN** ADD c_i to CLSTACK **ENDIF**
END FOR EACH
ENDIF
Phase 2 (Bottom-up model-building):
REPEAT
CHOOSE: $H \leftarrow \mathbf{B} \in$ CLSTACK and $\mathbf{B}' \subseteq$ FACTSTACK, such that $\mathbf{B}$ and $\mathbf{B}'$ unify with mgu θ ;
IF $H\theta \notin$ FACTSTACK **THEN** ADD $H\theta$ to FACTSTACK **ENDIF**;
IF *mode*$(H) = (Out, In)$ $\boxed{\textit{AND issuer}(H\theta) \notin \text{VISITED}}$ **THEN**
ADD to CLSTACK the clause:
$dummy(X, \textit{issuer}(H\theta)) \leftarrow dummy(X, \textit{issuer}(H\theta))$
ENDIF
UNTIL nothing can be added to FACTSTACK;
IF A is ground and $A \in$ FACTSTACK **THEN** SATISFIED = **TRUE ENDIF**
UNTIL SATISFIED **OR** nothing can be added to FACTSTACK and CLSTACK;
OUTPUT = FACTSTACK;

Fig. 1. The Lookup and Inference AlgoRithm (LIAR). We assume that *dummy* is a reserved predicate symbol, with mode (Out, In). Statements in boxes are optional and included only for optimisation purposes.

The algorithm maintains three *stacks*: CLSTACK contains the set of clauses collected so far, FACTSTACK contains the set of atomic logical consequences inferred from CLSTACK, and GOALSTACK contains the set of atomic goals already processed (to handle loops). Additionally, the VISITED stack contains the set of entities that have been visited during the processing. Initially, CLSTACK contains a single clause constructed from the initial atomic query A; the other stacks are empty. The algorithm is divided in two phases. Phase 1 contains the credential discovery. First, it selects a *new* well-moded atom $C\theta$ from the body of a clause in CLSTACK and then, depending on its mode, it fetches the *new* credentials from either *issuer*$(C\theta)$ or *recipient*$(C\theta)$. The fetched credentials are then added to the CLSTACK. Notice that, when *mode*$(C) = (Out, In)$, *all* clauses whose head has mode (Out, In) must be fetched from *recipient*$(C\theta)$, and not only the clauses whose head unifies with $C\theta$. This is because in this case one does not know which credentials may be needed to prove $C\theta$, yet. To overcome this problem, the algorithm overestimates and fetches all credentials with the right mode being stored at *recipient*$(C\theta)$. In Phase 2, the model of the set of clauses in the CLSTACK is build bottom-up. Newly inferred facts are added to the FACTSTACK. For the facts having mode (Out, In), the algorithm adds a *dummy* clause to CLSTACK, so that the "subject traceable" chains can be discovered properly. The algorithm extends naturally to queries containing more than one atom. The following results show that LIAR algorithm is sound and complete w.r.t. the standard LP semantics, i.e. the centralised algorithm based on the SLD resolution. All proofs are reported in [9]. We need the following lemma.

Lemma 2. *Let $\mathcal{P}$ be a state and* FACTSTACK *be the result of the algorithm execution for some well-moded query. Let A be an atom in* FACTSTACK. *Then A is ground.*

The soundness result is rather straightforward.

Theorem 1 (soundness). *Let $\mathcal{P}$ be a state and* FACTSTACK *be the result of executing LIAR on $\mathcal{P}$ and a well-moded query. Then $\forall A \in$ FACTSTACK, $P(\mathcal{P}) \models A$.*

Proof. It is easy to see that, by construction, if an atom A is added to FACTSTACK, then CLSTACK $\models A$. Since $\forall c \in$ CLSTACK c is an instance of a clause $c' \in P(\mathcal{P})$, it follows that $P(\mathcal{P}) \models A$. □

The following completeness result guarantees among other things that – after executing LIAR on a state $\mathcal{P}$ and some well-moded query – for any goal $A \in$ GOALSTACK it holds that if there exists a successful SLD derivation of A in $P(\mathcal{P})$ with c.a.s. θ then $A \overset{\theta}{\hookrightarrow}$ FACTSTACK.

Theorem 2 (completeness). *Let $\mathcal{P}$ be a state and then* FACTSTACK, GOALSTACK *be the result of executing LIAR on $\mathcal{P}$ and a given well-moded goal.*
Then $\forall C \in$ GOALSTACK, if $\exists$ a successful SLD derivation $\delta : C \xrightarrow{\theta}_{P(\mathcal{P})} \square$ then $C \overset{\theta}{\hookrightarrow}$ FACTSTACK.

5 CoreTuLiP vs. RT_0

In this section we are going to compare CoreTuLiP with the well-established RT_0 trust management language. We are going to show that – in most respects – CoreTuLiP is

at least as expressive as RT_0. To this end, we first present a slightly simplified (yet expressively equivalent) version of RT_0 as given in [16]: A *principal* is a uniquely identified individual or process. A principal can define a *role*, which is indicated by a principal's name followed by a *role name*, separated by a dot. For instance $a.r$, and $alice.pictures$ are roles. For the sake of uniformity, we depart from [16] in using names starting with a lowercase letter (sometimes with subscripts) to indicate role names. A role denotes a set of principals – the members of the role. RT_0 allows a principal to issue four kinds of statements:

- *Simple Member*: $a.r \leftarrow d$. "a asserts that d is a member of $a.r$."
- *Simple Inclusion*: $a.r \leftarrow b.r_1$. "a asserts that $a.r$ includes (all members of) $b.r_1$."
- *Linking Inclusion*: $a.r \leftarrow a.r_1.r_2$. "a asserts that $a.r$ includes $b.r_2$ for every b that is a member of $a.r_1$."
- *Intersection Inclusion*: $a.r \leftarrow b_1.r_1 \cap b_2.r_2$. "$a$ asserts that $a.r$ includes every principal who is a member of both $b_1.r_1$ and $b_2.r_2$."

An *RT_0 policy* (indicated by $\mathcal{S}$) is a set of RT_0 statements. Its semantics is defined by translating it into a *semantic program*, *SP*($\mathcal{S}$), which is a Prolog program with only one ternary predicate m. Given an RT_0 statement c, the *semantic program* of c, *SP*(c), is defined as follows:

$$\begin{aligned}
\textit{SP}(a.r \leftarrow d)) &= m(a, r, d).\\
\textit{SP}((a.r \leftarrow b.r_1)) &= m(a, r, X) :\!- m(b, r_1, X).\\
\textit{SP}((a.r \leftarrow a.r_1.r_2)) &= m(a, r, X) :\!- m(a, r_1, Y), m(Y, r_2, X).\\
\textit{SP}((a.r \leftarrow b_1.r_1 \cap b_2.r_2)) &= m(a, r, X) :\!- m(b_1, r_1, X), m(b_2, r_2, X).
\end{aligned}$$

SP extends to the set of statements in the obvious way: $\textit{SP}(\mathcal{S}) = \{\textit{SP}(c) \mid c \in \mathcal{S}\}$. Intuitively, $m(a, r, d)$ indicates that d is a member of the role $a.r$. Finally, given an RT_0 policy $\mathcal{S}$, the semantics of a role $a.r$ is defined in terms of atoms entailed by the semantic program: $[\![a.r]\!]_{SP(\mathcal{S})} = \{d \mid \textit{SP}(\mathcal{S}) \models m(a, r, d)\}$.

The Type System of RT_0. To ensure traceability, RT_0 comes with a type system [16]. In the original presentation, each role name has two types: an issuer-side type and a subject-side type. Here – also for the sake of simplicity – we assume that each role has just one of the following three type values: *issuer-traces-all (ITA)*, *issuer-traces-def (ITD)*, and *subject-traces-all (STA)*. To extend the results we present here to the full version (i.e. including all possible combinations of RT_0 types), we need to extend Core TuLiP in a straightforward way by allowing predicates with multiple modes.

Concerning storage, if a role name r is issuer-traces-all or issuer-traces-def, then principal a has to store all the credentials defining $a.r$. When a role name r is subject-traces-all then a credential of the form $a.r \leftarrow e$, must be stored by every subject of this credential. The successful discovery requires that each credential in the policy $\mathcal{S}$ be *well-typed*. For the sake of simplicity, we use the following definition of well-typed credentials (equivalent to [16]).

Definition 9. *Let c be an RT_0 credential. We say that c is* well-typed *iff the combination of type value assignments appears as a valid entry in Table 1.*

Table 1. Well-Typed RT_0 credentials

r \ $a.r \leftarrow b.r_1$	r_1: ITA	ITD	STA
ITA	OK		
ITD	OK	OK	OK
STA			OK

r \ $a.r \leftarrow a.r_1.r_2$	r_1: ITA			ITD			STA		
	r_2: ITA	ITD	STA	ITA	ITD	STA	ITA	ITD	STA
ITA	OK								
ITD	OK	OK	OK			OK			OK
STA									OK

r \ $a.r \leftarrow b_1.r_1 \cap b_2.r_2$	r_1: ITA			ITD			STA		
	r_2: ITA	ITD	STA	ITA	ITD	STA	ITA	ITD	STA
ITA	OK	OK	OK	OK			OK		
ITD	OK	OK	OK	OK	OK	OK	OK	OK	OK
STA			OK			OK	OK	OK	OK

For example, take the credential $c : a.r \leftarrow a.r_1.r_2$ and assume first that $type(r)$ and $type(r_1)$ is ITD, and $type(r_2)$ is STA. Then, after checking with Table 1, we see that c is well-typed w.r.t. this type value assignment. On the other hand, if $type(r) = type(r_1) = type(r_2) =$ ITD, then c is not well-typed as there is no valid entry for this type value assignment in Table 1. Note that simple member credentials (of the form $a.r \leftarrow b$) are always well-typed.

Three Sorts *of Goals.* If the set of credentials $\mathcal{S}$ is *well-typed* then there exists a terminating algorithm supporting three *sorts* of goals.

1. "given $a.r$, list all principals in $[\![a.r]\!]_{SP(\mathcal{S})}$"; this goal can be answered provided that r is issuer-traces-all.
2. "given $a.r$ and b, check if b is a member of $[\![a.r]\!]_{SP(\mathcal{S})}$"; this goal can be answered in all cases.
3. "given b, list all roles $a.r$ such that b is a member of $[\![a.r]\!]_{SP(\mathcal{S})}$"; this goal can be answered only partially: given b, the system is able to find all *subject traceable* roles $a.r$ (i.e. the roles $a.r$ where $type(r) =$ STA) such that b is a member of $[\![a.r]\!]_{SP(\mathcal{S})}$.

5.1 Translating RT_0 into CoreTuLiP

We now demonstrate that CoreTuLiP is – in most cases – more expressive than RT_0 by showing that an arbitrary RT_0 policy can be translated in a straightforward way into an equivalent CoreTuLiP state. First, we define a mapping T from RT_0 to CoreTuLiP.

Definition 10. *Let c be an RT_0 credential. Then $T(c)$ is defined as follows:*

$$T(a.r \leftarrow d) = r(a, d).$$

$$T(a.r \leftarrow b.r_1) = r(a, X) :\!- r_1(b, X).$$

$$T(a.r \leftarrow a.r_1.r_2) = \begin{cases} r(a, X) :\!- r_2(Y, X), r_1(a, Y). \textit{ if } type(r_1) \neq \textit{ITA}, \\ r(a, X) :\!- r_1(a, Y), r_2(Y, X). \textit{ otherwise.} \end{cases}$$

$$T(a.r \leftarrow b_1.r_1 \cap b_2.r_2) = \begin{cases} r(a, X) :\!- r_2(b_2, X), r_1(b_1, X). \textit{ if } type(r_1) \neq \textit{ITA}, \\ r(a, X) :\!- r_1(b_1, X), r_2(b_2, X). \textit{ otherwise.} \end{cases}$$

Concerning the mode of the predicates, we have: if the RT$_0$ type of r is ITA then the mode of r in $T(c)$ is (In, Out), if the RT$_0$ type of r is ITD then the mode of r in $T(c)$ is (In, In), and if the RT$_0$ type of r is STA then the mode of r in $T(c)$ is (Out, In). □

The following theorem shows that, from the view point of the declarative semantics, $\mathcal{S}$ and $T(\mathcal{S})$ are equivalent (recall that m is the fixed predicate symbol used in *SP*($\mathcal{S}$)). The proof can be found in [9].

Theorem 3. *Let $\mathcal{S}$ be an RT$_0$ policy. Then* $\mathrm{SP}(\mathcal{S}) \models m(a, r, d)$ *iff* $T(\mathcal{S}) \models r(a, d)$.

This shows that each RT$_0$ policy can be translated into a declaratively equivalent CoreTuLiP state. Now, to prove the full equivalence we still have to prove two things, namely that (a) if an RT$_0$ credential is stored at principal a then its corresponding CoreTuLiP statement is stored at a as well, and that (b) the CoreTuLiP system can answer the same goals the RT$_0$ system can. We start with the following proposition proven in [9].

Proposition 1. *Let c be an RT$_0$ credential.*

(a) If c is stored at a then $T(c)$ is also stored at a.
(b) If c is a well-typed then $T(c)$ is traceable.

At last, we have to show how RT$_0$ goals can be transformed into (legal, i.e. well-moded) CoreTuLiP queries. Since RT does not have a formal notation to express goals we have to be a bit verbose.

Remark 1 (Translating RT$_0$ goals). Let $\mathcal{S}$ be a well-typed RT$_0$ policy and $T(\mathcal{S})$ its CoreTuLiP equivalent. Let us consider the different sorts of goals supported by RT$_0$.

Sort 1: the general goal of this sort is "given $a.r$, list all principals in $[\![a.r]\!]_{SP(\mathcal{S})}$". This is translated into the query $r(a, X)$. This goal can be answered in RT$_0$ only if the role r has type ITA. But in this case the mode of r in $T(\mathcal{S})$ is (In, Out), and the query $r(a, X)$ is well-moded w.r.t. it. Therefore, we can conclude that goals of sort 1 can be safely expressed in CoreTuLiP.

Sort 2: the general goal of this sort is "given $a.r$ and b, check if b is a member of $[\![a.r]\!]_{SP(\mathcal{S})}$". This is translated into the query $r(a, b)$, which being ground is always well-moded. Therefore, we can conclude that goals of sort 2 can be safely expressed in CoreTuLiP.

Sort 3: the general goal of this sort is "given b, list all $a.r$ such that b is a member of $[\![a.r]\!]_{SP(\mathcal{S})}$". Such goals have no corresponding CoreTuLiP translation. The technical reason behind this limitation is purely of syntactic nature: the translation would be higher-order query ($X(Y, b)$, where X and Y are variables). There are two reasons why we believe that this limitation of CoreTuLiP w.r.t. RT$_0$ is hardly relevant in practice: first, RT$_0$ allows to express the query, but it is not able to give a complete answer in any case: it can only find all such $a.r$ which are also *subject traceable*. Secondly (also because RT$_0$ is not able to provide a full answer), this kind of goal is not used in practice on their own, but only as a subgoal of the goals of Sort 2. □

The syntactic inability of CoreTuLiP to express goals of Sort 3 is actually a conscious design choice we made to keep the syntax manageable (to express goals of this sort we

would need a "polymorphic" mode system in which the actual mode of an atom does not only depend on its predicate symbol but also on some of its arguments). Actually, our LIAR algorithm would be able to answer such queries as well.

Summarising, Theorem 3, Proposition 1, and Remark 1 allow us to say that CoreTuLiP is at least as expressive as RT_0, with the small exception of the goals of Sort 3. In proving this, we have made the restrictive assumption that an RT_0 role name has just one of the following three types: ITA (issuer-traces-all), ITD (issuer-traces-def), or STA (subject-traces-all). The extension to the full version (i.e. including all possible combinations of RT_0 types) can be done in a straightforward way by extending Core TuLiP so that it allows predicates with multiple modes.

5.2 A Flexible Syntax

As we said already, CoreTuLiP is simply the core language of the TM system we are developing. The full language will allow credentials with more than two arguments and user defined predicates. Nevertheless, CoreTuLiP is already expressive enough to express complex policies (like thresholds or separation of duty) that in RT require the adoption of special operators (which are present in more expressive members of the RT family RT_1, RT_2, RT^T, or RT^D). Consider for instance the following statement taken from [15] "a says that an entity is a member of $a.r$ if one member of $a.r_1$ and two *different* members of $a.r_2$ all say so". This policy cannot be expressed in RT_0, and to express this in RT one needs to use the so-called *manifold roles*, which extend the notion of roles by allowing role members to be *collections* of entities (rather than just principals). This is done in RT^T by defining the operators $\odot$ and $\otimes$. A *type-5* credential of the form $a.r \leftarrow b_1.r_1 \odot b_2.r_2$ says that $\{s_1 \cup s_2\}$ is a member of $a.r$ if s_1 is a member of $b_1.r_1$ and s_2 is a member of $b_2.r_2$. A *type-6* credential $a.r \leftarrow b_1.r_1 \otimes b_2.r_2$ has a similar meaning, but it additionally requires that $s_1 \cap s_2 = \emptyset$. With these two additional types, one can express the above statement using the following three credentials:

$$\begin{aligned} a.r &\leftarrow a.r_4.r \\ a.r_4 &\leftarrow a.r_1 \odot a.r_3 \\ a.r_3 &\leftarrow a.r_2 \otimes a.r_2 \end{aligned}$$

In CoreTuLiP, on the other hand, this policy can be expressed quite naturally with the following one line:

$$r(a, X) :- r_1(a, Y),\ r(Y, X),\ r_2(a, Z_1),\ r_2(a, Z_2),\ Z_1 \neq Z_2,\ r(Z_1, X),\ r(Z_2, X).$$

Notably, to express this, we don't have to use manifold-like structures which are, in our opinion, rather hard to grasp.

6 Related Work

The term "trust management" was first coined by Blaze, Feigenbaum, and Lacy [7] as an answer to the inadequacy of traditional authorisation mechanisms based on identity-based public-key systems like X.509. X.509 strongly depends on a centralised hierarchy

of certificate authorities (CA), which significantly restricts the possible application area in the distributed heterogeneous systems. In their work, Blaze et al. identify various requirements that a trust management system should satisfy and they introduce *PolicyMaker* - a trust management engine fulfilling these requirements. A direct descendant of PolicyMaker is KeyNote [5]. The reader can find a comparison of PolicyMaker and KeyNote in [6].

Clark et al. developed SDSI/SPKI [8] that combines local namespaces and the corresponding *name certificates* of SDSI [20] with *authorisation certificates* of SPKI [10]. They present a TM language that they prove to be expressive enough for many access control scenarios. All these TM systems avoid the problem of *credential storage*. In all these approaches it is assumed that all required credentials can be found when needed and as such they are not well-prepared to face the reality of the today's highly distributed world. In [21] Winsborough and Li identify the features a "good" language for credentials should have, one of those being the support for distributed storage. The RT family of Trust Management Languages [16,15] is the first in which the problem of credential discovery is given an extensive treatment. In particular, in [16], a type system is introduced in order to restrict the number of possible credential storage options. In Sect. 5 we compare the type system of RT_0 with our approach to the credential discovery problem.

Li, Grosof, and Feigenbaum in [14] developed a logic-based language, called Delegation Logic (DL), to represent policies, credentials, and requests for distributed authorisations. Monotonic version of Delegation Logic – called D1LP – is based on Logic Programming language Datalog. D1LP extends Datalog with constructs that feature delegation depth and a wide variety of complex principals.

Bertino et al. [4] introduce an XML-based trust negotiation language $\mathcal{X}$-TNL used for expressing credentials and disclosure policies in the *Trust-*$\mathcal{X}$ system. Though Trust-$\mathcal{X}$ policies are formalised using logic rules, it also does not directly support credential discovery.

The well-known *eXtensible Access Control Markup Language* (XACML) [19] supports the distributed policies and also provides a profile for role based access control (RBAC). However, in XACML, it is the responsibility of the *Policy Decision Point* (PDP) – an entity handling access requests – to know where to look for the missing attribute values in the request.

7 Conclusions

In this paper we introduce CoreTuLiP, a true Trust Management Language which enjoys the advantages of LP syntax and of its declarative semantics. CoreTuLiP forms the basis for the TuLiP TM language we are developing at the UT (which will include user-defined predicates and will enjoy most features of moded logic programs, including interface with other languages, debugging facilities, etc.). The main purpose of CoreTuLiP is to provide a theoretical basis for the further developments.

CoreTuLiP enjoys the advantages of trust management languages: for instance, statements may be issued by multiple authorities and be stored by authorities different from the issuing one. To deal with the problem of finding the credentials which are needed for

a proof, we define the notion of traceable credentials and present a Lookup and Inference AlgoRithm (LIAR), which we show to be correct and complete w.r.t. the standard declarative semantics. We also compare CoreTuLiP with RT_0 and show that each RT_0 credential and goal translates in a straightforward way into Core TuLiP (with the small exception of the goals of Sort 3).

The theoretical relevance of this paper is that we show that it is possible to define a true TM language without leaving the well-established LP formalism. The practical relevance lies in the much greater flexibility, extendibility and accessibility that LP language enjoys with respect to – for instance – RT. As we have discussed, to accommodate various needs, the language RT_0 has developed a relatively large number of extensions, which are in our opinion often hard to grasp. We thought that this was the price we had to pay to have a true TM language, but CoreTuLiP shows that this can be done otherwise.

Future Work. CoreTuLiP can be extended in several directions. First we plan to investigate extending CoreTuLiP to support non-stratified negation. This is connected to our previous work on $RT_{\ominus}$[18], where we extend RT_0 with *negation-in-context*. Secondly, we are going to add support for integrity constraints for TM systems [11]. We also plan to provide an implementation for CoreTuLiP, possibly supporting the XACML standard.

Acknowledgements. We would like to thank Pieter Hartel and Jeroen Doumen from the University of Twente for their feedback and valuable comments to this paper.

References

1. Apt, K.R.: From Logic Programming to Prolog. Prentice-Hall, Englewood Cliffs (1997)
2. Apt, K.R., Luitjes, I.: Verification of Logic Programs with Delay Declarations. In: Alagar, V.S., Nivat, M. (eds.) AMAST 1995. LNCS, vol. 936, pp. 66–90. Springer, Heidelberg (1995)
3. Apt, K.R., Marchiori, E.: Reasoning about Prolog programs: from Modes through Types to Assertions. Formal Aspects of Computing 6(6A), 743–765 (1994)
4. Bertino, E., Ferrari, E., Squicciarini, A.C.: Trust-χ: A Peer-to-Peer Framework for Trust Establishment. IEEE Trans. Knowl. Data Eng. 16(7), 827–842 (2004)
5. Blaze, M., Feigenbaum, J., Ioannidis, J., Keromytis, A.: The KeyNote Trust-Management System, Version 2. IETF RFC 2704 (September 1999)
6. Blaze, M., Feigenbaum, J., Ioannidis, J., Keromytis, A.: The Role of Trust Management in Distributed Systems Security. In: Vitek, J., Jensen, C. (eds.) Secure Internet Programming. LNCS, vol. 1603, pp. 185–210. Springer, Heidelberg (1999)
7. Blaze, M., Feigenbaum, J., Lacy, J.: Decentralized Trust Management. In: Proc. of the 17th IEEE Symposium on Security and Privacy, pp. 164–173. IEEE Computer Society Press, Los Alamitos (1996)
8. Clarke, D., Elien, J.E., Ellison, C., Fredette, M., Morcos, A., Rivest, R.L.: Certificate Chain Discovery in SPKI/SDSI. Journal of Computer Security 9(4), 285–322 (2001)
9. Czenko, M.R., Etalle, S.: Core TuLiP. Technical Report TR-CTIT-07-22, Centre for Telematics and Information Technology. University of Twente, Enschede (March 2007)
10. Ellison, C., Frantz, B., Lampson, B., Rivest, R., Thomas, B., Ylonen, T.: SPKI Certificate Theory. IETF RFC 2693 (September 1999)

11. Etalle, S., Winsborough, W.H.: Integrity Constraints in Trust Management – Extended Abstract. In: Ahn, G-J. (ed.) Proc. 10th ACM Symp. on Access Control Models and Technologies (SACMAT), pp. 1–10. ACM Press, New York (2005), Extended version available at CoRR: http://arxiv.org/abs/cs.CR/0503061
12. Freeband Communication. I-Share: Sharing Resources in Virtual Communities for Storage, Communications, and Processing of Multimedia Data. URL: http://www.freeband.nl/project.cfm?language=en\&id=520
13. Jim, T.: SD3: A Trust Management System with Certified Evaluation. In: Proceedings of the 2001 IEEE Symposium on Security and Privacy, pp. 106–115. IEEE Computer Society Press, Los Alamitos (2001)
14. Li, N., Grosof, B., Feigenbaum, J.: Delegation Logic: A Logic-based Approach to Distributed Authorization. ACM Transactions on Information and System Security (TISSEC) 6(1), 128–171 (2003)
15. Li, N., Mitchell, J., Winsborough, W.: Design of A Role-based Trust-management Framework. In: Proc. 2002 IEEE Symposium on Security and Privacy, pp. 114–130. IEEE Computer Society Press, Los Alamitos (2002)
16. Li, N., Winsborough, W., Mitchell, J.: Distributed Credential Chain Discovery in Trust Management. Journal of Computer Security 11(1), 35–86 (2003)
17. Lloyd, J.W.: Foundations of Logic Programming, 2nd edn. Springer, Heidelberg (1993)
18. Czenko, M., Tran, H., Doumen, J., Etalle, S., Hartel, P., den Hartog, J.: Nonmonotonic Trust Management for P2P Applications. In: Proc. of the 1st International Workshop on Security and Trust Management. Electronic Notes in Theoretical Computer Science, vol. 157, pp. 113–130. Elsevier, Amsterdam (2006)
19. OASIS.: eXtensible Access Control Markup Language (XACML) Version 2.0 (February 2005), URL: http://www.oasis.org
20. Rivest, R., Lampson, B.: SDSI - A Simple Distributed Security Infrastructure (October 1996), Available at http://theory.lcs.mit.edu/~rivest/sdsi11.html
21. Winsborough, W.H., Li, N.: Towards Practical Automated Trust Negotiation. In: POLICY, pp. 92–103. IEEE Computer Society Press, Los Alamitos (2002)

Demand-Driven Indexing of Prolog Clauses*

Vítor Santos Costa[1], Konstantinos Sagonas[2], and Ricardo Lopes

[1] LIACC- DCC/FCUP, University of Porto, Portugal
[2] National Technical University of Athens, Greece

Abstract. As logic programming applications grow in size, Prolog systems need to efficiently access larger and larger data sets and the need for any- and multi-argument indexing becomes more and more profound. Static generation of multi-argument indexing is one alternative, but applications often rely on features that are inherently dynamic which makes static techniques inapplicable or inaccurate. Another alternative is to employ dynamic schemes for flexible demand-driven indexing of Prolog clauses. We propose such schemes and discuss issues that need to be addressed for their efficient implementation in the context of WAM-based Prolog systems. We have implemented demand-driven indexing in two different Prolog systems and have been able to obtain non-negligible performance speedups: from a few percent up to orders of magnitude. Given these results, we see very little reason for Prolog systems not to incorporate some form of dynamic indexing based on actual demand. In fact, we see demand-driven indexing as only the first step towards effective runtime optimization of Prolog programs.

1 Introduction

The WAM [1] has mostly been a blessing but occasionally also a curse for Prolog systems. Its ingenious design has allowed implementors to get byte code compilers with decent performance — it is not a fluke that most Prolog systems are still based on the WAM. On the other hand, *because* the WAM gives good performance in many cases, implementors have not incorporated in their systems many features that drastically depart from WAM's basic characteristics. For example, first argument indexing is sufficient for many Prolog applications. However, it is clearly sub-optimal for applications accessing large data sets; for a long time now, the database community has recognized that good indexing is the basis for fast query processing.

As logic programming applications grow in size, Prolog systems need to efficiently access larger and larger data sets and the need for any- and multi-argument indexing becomes more and more profound. Static generation of multi-argument indexing is one alternative. The problem is that this alternative is often unattractive because it may drastically increase the size of the generated byte code and do so unnecessarily. Static analysis can partly address this concern, but in applications that rely on features which are inherently dynamic (e.g., generating hypotheses for inductive logic programming data sets during runtime) static analysis is inapplicable or grossly inaccurate. Another alternative, which has not been investigated so far, is to do flexible indexing on demand during program execution.

* Dedicated to the memory of our friend, colleague and co-author Ricardo Lopes. We miss you!

V. Dahl and I. Niemelä (Eds.): ICLP 2007, LNCS 4670, pp. 395–409, 2007.

This is precisely what we advocate with this paper. More specifically, we present a small extension to the WAM that allows for flexible indexing of Prolog clauses during runtime based on actual demand. For static predicates, the scheme we propose is partly guided by the compiler; for dynamic code, besides being demand-driven by queries, the method needs to cater for code updates during runtime. Where our schemes radically depart from current practice is that they generate new byte code during runtime, in effect doing a form of just-in-time compilation. In our experience these schemes pay off. We have implemented Demand-Driven Indexing in two different Prolog systems (YAP and XXX) and have obtained non-trivial speedups, ranging from a few percent to orders of magnitude, across a wide range of applications. Given these results, we see very little reason for Prolog systems not to incorporate some form of indexing based on actual demand from queries. In fact, we see Demand-Driven Indexing as only the first step towards effective runtime optimization of Prolog programs.

Organization. After commenting on the state of the art and related work concerning indexing in Prolog systems (Sect. 2) we briefly review indexing in the WAM (Sect. 3). We then present Demand-Driven Indexing schemes for static (Sect. 4) and dynamic (Sect. 5) predicates, their implementation in two Prolog systems (Sect. 6) and the performance benefits they bring (Sect. 7). The paper ends with some concluding remarks.

2 State of the Art and Related Work

Many Prolog systems still only support indexing on the main functor symbol of the first argument. Some others, such as YAP version 4, can look inside some compound terms [2]. SICStus Prolog supports *shallow backtracking* [3]; choice points are fully populated only when it is certain that execution will enter the clause body. While shallow backtracking avoids some of the performance problems of unnecessary choice point creation, it does not offer the full benefits that indexing can provide. Other systems such as BIM-Prolog [4], SWI-Prolog [5] and XSB [6] allow for user-controlled multi-argument indexing. Notably, ilProlog [7] uses compile-time heuristics and generates code for multi-argument indexing automatically. In all these systems, this support comes with various implementation restrictions. For example, in SWI-Prolog at most four arguments can be indexed; in XSB the compiler does not offer multi-argument indexing and the predicates need to be asserted instead; we know of no system where multi-argument indexing looks inside compound terms. More importantly, requiring users to specify arguments to index on is neither user-friendly nor guarantees good performance results.

Recognizing the need for better indexing, researchers have proposed more flexible indexing mechanisms for Prolog. For example, Hickey and Mudambi proposed *switching trees* [8], which rely on the presence of mode information. Similar proposals were put forward by Van Roy, Demoen and Willems who investigated indexing on several arguments in the form of a *selection tree* [9] and by Zhou et al. who implemented a *matching tree* oriented abstract machine for Prolog [10]. For static predicates, the XSB compiler offers support for *unification factoring* [11]; for asserted code, XSB can represent databases of facts using *tries* [12] which provide left-to-right multi-argument

indexing. However, in XSB none of these mechanisms is used automatically; instead the user has to specify appropriate directives.

Long ago, Kliger and Shapiro argued that such tree-based indexing schemes are not cost effective for the compilation of Prolog programs [13]. Some of their arguments make sense for certain applications, but, as we shall show, in general they underestimate the benefits of indexing on EDB predicates. Nevertheless, it is true that unless the modes of predicates are known we run the risk of doing indexing on output arguments, whose only effect is an unnecessary increase in compilation times and, more importantly, in code size. In a programming language such as Mercury [14] where modes are known the compiler can of course avoid this risk; indeed in Mercury modes (and types) are used to guide the compiler generate good indexing tables. However, the situation is different for a language like Prolog. Getting accurate information about the set of all possible modes of predicates requires a global static analyzer in the compiler — and most Prolog systems do not come with one. More importantly, it requires a lot of discipline from the programmer (e.g., that applications use the module system religiously and never bypass it). As a result, most Prolog systems currently do not provide the type of indexing that applications require. Even in systems such as Ciao [15], which do come with a built-in static analyzer and more or less force such a discipline on the programmer, mode information is not used for multi-argument indexing.

The situation is actually worse for certain types of Prolog applications. For example, consider applications in the area of inductive logic programming. These applications on the one hand have high demands for effective indexing since they need to efficiently access big datasets and on the other they are unfit for static analysis since queries are often ad hoc and generated only during runtime as new hypotheses are formed or refined. Our thesis is that the abstract machine should be able to adapt automatically to the runtime requirements of such or, even better, of all applications by employing increasingly aggressive forms of dynamic compilation. As a concrete example of what this means in practice, in this paper we will attack the problem of satisfying the indexing needs of applications during runtime. Naturally, we will base our technique on the existing support for indexing that the WAM provides, but we will extend this support with the technique of Demand-Driven Indexing that we describe in the next sections.

3 Indexing in the WAM

To make the paper relatively self-contained we review the indexing instructions of the WAM and their use. In the WAM, the first level of dispatching involves a test on the type of the argument. The switch_on_term instruction checks the tag of the dereferenced value in the first argument register and implements a four-way branch where one branch is for the dereferenced register being an unbound variable, one for being atomic, one for (non-empty) list, and one for structure. In any case, control goes to a bucket of clauses. In the buckets for constants and structures the second level of dispatching involves the value of the register. The switch_on_constant and switch_on_structure instructions implement this dispatching: typically with a fail instruction when the bucket is empty, with a jump instruction for only one clause, with a sequential scan when the number of clauses is small, and with a hash table lookup when the number of clauses

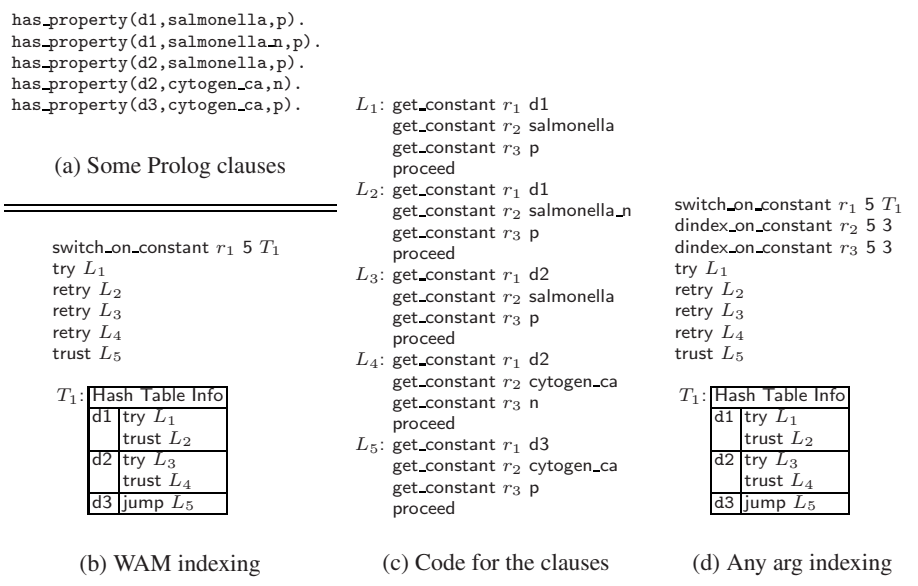


Fig. 1. Part of the Carcinogenesis dataset and WAM code that a byte code compiler generates

exceeds a threshold. For this reason the switch_on_constant and switch_on_structure instructions take as arguments the hash table T and the number of clauses N the table contains. In each bucket of this hash table and also in the bucket for the variable case of switch_on_term the code sequentially backtracks through the clauses using a try-retry-trust chain of instructions. The try instruction sets up a choice point, the retry instructions (if any) update certain fields of this choice point, and the trust instruction removes it.

The WAM has additional indexing instructions (try_me_else and friends) that allow indexing to be interspersed with the code of clauses. We will not consider them here. This is not a problem since the above scheme handles all programs. Also, we will feel free to do some minor modifications and optimizations when this simplifies things.

Let's see an example. Consider the Prolog code shown in Fig. 1(a), a fragment of the machine learning dataset *Carcinogenesis*. These clauses get compiled to the WAM code shown in Fig. 1(c). The first argument indexing code that a Prolog compiler generates is shown in Fig. 1(b). This code is typically placed before the code for the clauses and the switch_on_constant is the entry point of the predicate. Note that compared with vanilla WAM this instruction has an extra argument: the register on the value of which we index (r_1). This extra argument will allow us to go beyond first argument indexing. Another departure from the WAM is that if this argument register contains an unbound variable instead of a constant then execution will continue with the next instruction; in effect we have merged part of the functionality of switch_on_term into the switch_on_constant instruction. This small change in the behavior of switch_on_constant will allow us to get Demand-Driven Indexing. Let's see how.

4 Demand-Driven Indexing of Static Predicates

For static predicates the compiler has complete information about all clauses and shapes of their head arguments. It is both desirable and possible to take advantage of this information at compile time and so we treat the case of static predicates separately. We will do so with schemes of increasing effectiveness and implementation complexity.

4.1 A Simple WAM Extension for Any Argument Indexing

Let us initially consider the case where the predicates to index consist only of Datalog facts. This is commonly the case for all extensional database predicates where indexing is most effective and called for.

Refer to the example in Fig. 1. The indexing code of Fig. 1(b) incurs a small cost for a call where the first argument is a variable (namely, executing the switch_on_constant instruction) but the instruction pays off for calls where the first argument is bound. On the other hand, for calls where the first argument is a free variable and some other argument is bound, a choice point will be created, the try-retry-trust chain will be used, and execution will go through the code of all clauses. This is clearly inefficient, more so for larger data sets. We can do much better with the relatively simple scheme shown in Fig. 1(d). Immediately after the switch_on_constant instruction, we can statically generate dindex_on_constant (demand indexing) instructions, one for each remaining argument. Recall that the entry point of the predicate is the switch_on_constant instruction. The dindex_on_constant r_i N A instruction works as follows:

- if the argument r_i is a free variable, execution continues with the next instruction;
- otherwise, Demand-Driven Indexing kicks in as follows. The abstract machine scans the WAM code of the clauses and creates an index table for the values of the corresponding argument. It can do so because the instruction takes as arguments the number of clauses N to index and the arity A of the predicate. (In our example, the numbers 5 and 3.) For Datalog facts, this information is sufficient. Because the WAM byte code for the clauses has a very regular structure, the index table can be created very quickly. Upon its creation, the dindex_on_constant instruction gets transformed to a switch_on_constant. Again this is straightforward because of the two instructions have similar layouts in memory. Execution of the abstract machine then continues with the switch_on_constant instruction.

Figure 2 shows the index table T_2 which is created for our example and how the indexing code looks after the execution of a call with mode `(out,in,?)`. Note that the dindex_on_constant instruction for argument register r_2 has been appropriately patched. The call that triggered Demand-Driven Indexing and subsequent calls of the same mode will use table T_2. The index for the second argument has been created.

The main advantage of this scheme is its simplicity. The compiled code (Fig. 1(d)) is not significantly bigger than the code which a WAM-based compiler would generate (Fig. 1(b)) and, if Demand-Driven Indexing turns out unnecessary during runtime (e.g. execution encounters only open calls or with only the first argument bound), the extra overhead is minimal: the execution of some dindex_on_constant instructions for the open call only. In short, this is a simple scheme that allows for indexing on *any single* argument. At least for big sets of Datalog facts, we see little reason not to use it.

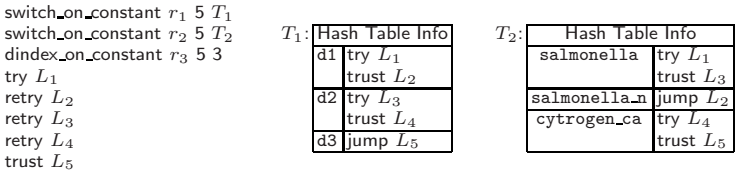


Fig. 2. WAM code after demand-driven indexing for argument 2; T_2 is generated dynamically

Optimizations. Because we are dealing with static code, there are opportunities for some easy optimizations. Suppose we statically determine that there will never be any calls with `in` mode for some arguments or that these arguments are not discriminating enough.[1] Then we can avoid generating dindex_on_constant instructions for them. Also, suppose we know that some arguments are most likely than others to be used in the `in` mode. Then we can simply place the dindex_on_constant instructions for them before the instructions for other arguments. This is possible since all indexing instructions take the argument register number as an argument; their order does not matter.

4.2 From Any Argument Indexing to Multi-argument Indexing

The scheme of the previous section gives us only single argument indexing. However, all the infrastructure we need is already in place. We can use it to obtain any fixed-order multi-argument Demand-Driven Indexing in a straightforward way.

Note that the compiler knows exactly the set of clauses that need to be tried for each query with a specific symbol in the first argument. For multi-argument Demand-Driven Indexing, instead of generating for each hash bucket only try-retry-trust instructions, the compiler can prepend appropriate demand indexing instructions. We illustrate this on our running example. The table T_1 contains four dindex_on_constant instructions: two for each of the remaining two arguments of hash buckets with more than one alternative. For hash buckets with none or only one alternative (e.g., for `d3`'s bucket) there is obviously no need to resort to Demand-Driven Indexing for the remaining arguments. Figure 3 shows the state of the hash tables after the execution of queries `has_property(C,salmonella,T)`, which creates T_2, and `has_property(d2,P,n)` which creates the T_3 table and transforms the dindex_on_constant instruction for `d2` and register r_3 to the appropriate switch_on_constant instruction.

Implementation issues. In the dindex_on_constant instructions of Fig. 3 notice the integer 2 which denotes the number of clauses that the instruction will index. Using this number an index table of appropriate size will be created, such as T_3. To fill this table we need information about the clauses to index and the symbols to hash on. The clauses can be obtained by scanning the labels of the try-retry-trust instructions following dindex_on_constant; the symbols by looking at appropriate byte code offsets (based on the argument register number) from these labels. In our running example, the symbols can be obtained by looking at the second argument of the get_constant instruction whose argument register is r_2. In the loaded bytecode, assuming the argument

[1] In our example, suppose the third argument of `has_property/3` was the atom `p` throughout.

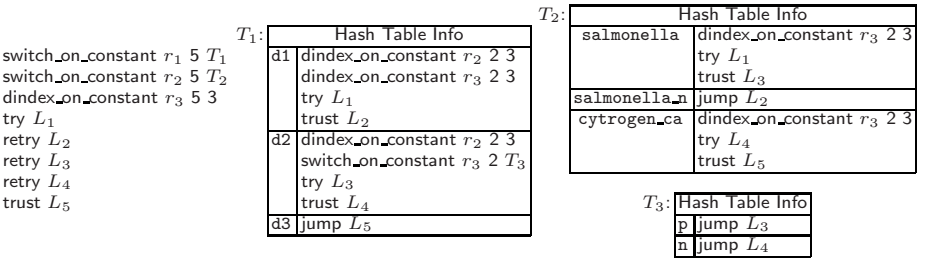


Fig. 3. Demand-Driven Indexing for all arguments; T_1 is static; T_2 and T_3 are created dynamically

register is represented in one byte, these symbols are found $sizeof(\mathsf{get\_constant}) + sizeof(opcode) + 1$ bytes away from the clause label; see Fig. 1(c). Thus, multi-argument Demand-Driven Indexing is easy to get and the creation of index tables can be extremely fast when indexing Datalog facts.

4.3 Beyond Datalog and Other Implementation Issues

Indexing on demand clauses with function symbols is not significantly more difficult. The scheme we have described is applicable but requires the following extensions:

1. Besides dindex_on_constant we also need dindex_on_term and dindex_on_structure instructions. These are the Demand-Driven Indexing counterparts of the WAM's switch_on_term and switch_on_structure.
2. Because the byte code for the clause heads does not necessarily have a regular structure, the abstract machine needs to be able to "walk" the byte code instructions and recover the symbols on which indexing will be based. Writing such a code walking procedure is not hard.
3. Indexing on a position that contains unconstrained variables for some clauses is tricky. The WAM needs to group clauses in this case and without special treatment creates two choice points for this argument (one for the variables and one per each group of clauses). However, this issue and how to deal with it is well-known by now. Possible solutions to it are described in a paper by Carlsson [16] and can be readily adapted to Demand-Driven Indexing. Alternatively, in a simple implementation, we can skip Demand-Driven Indexing for positions with variables in some clauses.

Before describing Demand-Driven Indexing more formally, we remark on the following design decisions whose rationale may not be immediately obvious:

- By default, only table T_1 is generated at compile time (as in the WAM) and the additional index tables $T_2, T_3, \ldots$ are generated dynamically. This is because we do not want to increase compiled code size unnecessarily (i.e., when there is no demand for these indices).
- On the other hand, we generate dindex_on_* instructions at compile time for the head arguments.[2] This does not noticeably increase the generated byte code but it

[2] The dindex_on_* instructions for T_1 can be generated either by the compiler or the loader.

greatly simplifies code loading. Notice that a nice property of the scheme we have described is that the loaded byte code can be patched *without* the need to move any instructions.
- Finally, one may wonder why the dindex_on_* instructions create the dynamic index tables with an additional code walking pass instead of piggy-backing on the pass which examines all clauses via the main try-retry-trust chain. Main reasons are: 1) in many cases the code walking can be selective and guided by offsets and 2) by first creating the index table and then using it we speed up the execution of the queries and often avoid unnecessary choice point creations.

Note that all these decisions are orthogonal to the main idea and are under compiler control. For example, if analysis determines that some argument sequences will never demand indexing we can simply avoid generation of dindex_on_* instructions for them. Similarly, if some argument sequences will definitely demand indexing we can speed up execution by generating the appropriate tables at compile time instead of dynamically.

4.4 Demand-Driven Index Construction and Its Properties

The idea behind Demand-Driven Indexing can be captured in a single sentence: *we can generate every index we need during program execution when this index is demanded*. Subsequent uses of these indices can speed up execution considerably more than the time it takes to construct them (more on this below) so this runtime action makes sense.

Let p/k be a predicate with n clauses. At a high level, its indices form a tree whose root is the entry point of the predicate. For simplicity, assume that the root node of the tree and the interior nodes corresponding to the index table for the first argument have been constructed at compile time. Leaves of this tree are the nodes containing the code for the clauses of the predicate and each clause is identified by a unique label $L_i, 1 \leq i \leq n$. Execution always starts at the first instruction of the root node and follows Algorithm 1. The algorithm might look complicated but is actually quite simple. Each non-leaf node contains a sequence of byte code instructions with groups of the form $\langle I_1, \ldots, I_m, T_1, \ldots, T_l \rangle, 0 \leq m \leq k, 1 \leq l \leq n$ where each of the I instructions, if any, is either a switch_on_* or a dindex_on_* instruction and each of the T instructions either forms a sequence of try-retry-trust instructions (if $l > 1$) or is a jump instruction (if $l = 1$). Step 2.2 dynamically constructs an index table $\mathcal{T}$ whose buckets are the newly created interior nodes in the tree. Each bucket associated with a single clause contains a jump to the label of that clause. Each bucket associated with many clauses starts with the I instructions which are yet to be visited and continues with a try-retry-trust chain pointing to the clauses. When the index construction is done, the instruction mutates to a switch_on_* WAM instruction.

Complexity properties. Index construction during runtime does not change the complexity of query execution. First, note that each demanded index table will be constructed at most once. Also, a dindex_on_* instruction will be encountered only in cases where execution would examine all clauses in the try-retry-trust chain.[3] The construction visits these clauses *once* and then creates the index table in time linear in the number of clauses as one pass over the list of $\langle c, L \rangle$ pairs suffices. After index construction,

[3] This statement is possibly not valid in the presence of Prolog cuts.

Algorithm 1. Actions of the abstract machine with Demand-Driven Indexing

1. if the current instruction I is a switch_on_*, try, retry, trust or jump, act as in the WAM;
2. if the current instruction I is a dindex_on_* with arguments r, l, and k (r is a register) then
   - 2.1 if register r contains a variable, the action is a goto the next instruction in the node;
   - 2.2 if register r contains a value v, the action is to dynamically construct the index:
     - 2.2.1 collect the subsequent instructions in a list $\mathcal{I}$ until the next instruction is a try;
     - 2.2.2 for each label L in the try-retry-trust chain inspect the code of the clause with label L to find the symbol c associated with register r in the clause; (This step creates a list of $\langle c, L \rangle$ pairs.)
     - 2.2.3 create an index table $\mathcal{T}$ out of these pairs as follows:
       - if I is a dindex_on_constant or a dindex_on_structure then create an index table for the symbols in the list of pairs; each entry of the table is identified by a symbol c and contains:
         - the instruction jump L_c if L_c is the only label associated with c;
         - the sequence of instructions obtained by appending to $\mathcal{I}$ a try-retry-trust chain for the sequence of labels $L'_1, \ldots, L'_l$ that are associated with c
       - if I is a dindex_on_term then
         - partition the sequence of labels $\mathcal{L}$ in the list of pairs into sequences of labels $\mathcal{L}_c, \mathcal{L}_l$ and $\mathcal{L}_s$ for constants, lists and structures, respectively;
         - for each of the four sequences $\mathcal{L}, \mathcal{L}_c, \mathcal{L}_l, \mathcal{L}_s$ of labels create code:
           - the instruction fail if the sequence is empty;
           - the instruction jump L if L is the only label in the sequence;
           - the sequence of instructions obtained by appending to $\mathcal{I}$ a try-retry-trust chain for the current sequence of labels;
     - 2.2.4 transform the dindex_on_* r, l, k instruction to a switch_on_* $r, l, \mathcal{T}$ instruction;
     - 2.2.5 continue execution with this instruction.

execution will visit a subset of these clauses as the index table will be consulted. Thus, in cases where Demand-Driven Indexing is not effective, execution of a query will at most double due to dynamic index construction. In fact, this worst case is pessimistic and unlikely in practice. On the other hand, Demand-Driven Indexing can change the complexity of query evaluation from $O(n)$ to $O(1)$ where n is the number of clauses.

4.5 More Implementation Choices

The observant reader has no doubt noticed that Algorithm 1 provides multi-argument indexing but only for the main functor symbol. For clauses with compound terms that require indexing in their sub-terms we can either employ a program transformation such as *unification factoring* [11] at compile time or modify the algorithm to consider index positions inside compound terms. This is relatively easy to do but requires support from the register allocator (passing the sub-terms of compound terms in appropriate registers) and/or a new set of instructions. Due to space limitations we omit further details.

Algorithm 1 relies on a procedure that inspects the code of a clause and collects the symbols associated with some particular index position (step 2.2.2). If we are satisfied with looking only at clause heads, this procedure needs to understand only the structure of get and unify instructions. Thus, it is easy to write. At the cost of increased implementation complexity, this step can of course take into account other information that

may exist in the body of the clause (e.g., type tests such as `var(X)`, `atom(X)`, aliasing constraints such as `X = Y`, numeric constraints such as `X > 0`, etc.).

A reasonable concern for Demand-Driven Indexing is increased memory consumption. In our experience, this does not seem to be a problem in practice since most applications do not have demand for indexing on many argument combinations. In applications where it does become a problem or when running in an environment with limited memory, we can easily put a bound on the size of index tables, either globally or for each predicate separately. For example, the dindex_on_* instructions can either become inactive when this limit is reached, or better yet we can recover the space of some tables. To do so, we can employ any standard recycling algorithm (e.g., LRU) and reclaim the memory of index tables that are no longer in use. This is easy to do by reverting the corresponding switch_on_* instructions back to dindex_on_* instructions. If the indices are demanded again at a time when memory is available, they can simply be regenerated.

5 Demand-Driven Indexing of Dynamic Predicates

We have so far lived in the comfortable world of static predicates, where the set of clauses to index is fixed and the compiler can take advantage of this knowledge. Dynamic code introduces several complications:

- We need mechanisms to update multiple indices when new clauses are asserted or retracted. In particular, we need the ability to expand and possibly shrink multiple code chunks after code updates.
- We do not know a priori which are the best index positions and cannot determine whether indexing on some arguments is avoidable.
- Supporting the logical update (LU) semantics of ISO Prolog becomes harder.

We briefly discuss possible ways of addressing these issues. However, note that Prolog systems typically provide indexing for dynamic predicates and thus already deal in some way or another with these issues; Demand-Driven Indexing makes the problems more involved but not fundamentally different than with only first argument indexing.

The first complication suggests that we should allocate memory for dynamic indices in separate chunks, so that these can be expanded and deallocated independently. Indeed, this is what we do. Regarding the second complication, in the absence of any other information, the only alternative is to generate indices for all arguments. As optimizations, we can avoid indexing predicates with only one clause and exclude arguments where some clause has a variable.

Under LU semantics, calls to dynamic predicates execute in a "snapshot" of the corresponding predicate. Each call sees the clauses that existed at the time when the call was made, even if some of the clauses were later retracted or new clauses were asserted. If several calls are alive in the stack, several snapshots will be alive at the same time. The standard solution to this problem is to use time stamps to tell which clauses are *live* for which calls. This solution complicates freeing index tables because: (1) an index table holds references to clauses, and (2) the table may be in use (i.e., may be accessible from the execution stacks). An index table thus is killed in several steps:

1. Detach the index table from the indexing tree.
2. Recursively *kill* every child of the current table; if a table is killed so are its children.

3. Wait until the table is not in use, that is, it is not pointed to from anywhere.
4. Walk the table and release any references it may hold.
5. Physically recover space.

6 Implementation in XXX and in YAP

The implementation of Demand-Driven Indexing in XXX follows a variant of the scheme presented in Sect. 4. The compiler uses heuristics to determine the best argument to index on (i.e., this argument is not necessarily the first) and employs switch_on_* instructions for this task. It also statically generates dindex_on_constant instructions for other arguments that are good candidates for Demand-Driven Indexing. Currently, an argument is considered a good candidate if it has only constants or only structure symbols in all clauses. Thus, XXX uses only dindex_on_constant and dindex_on_structure instructions, never a dindex_on_term. Also, XXX does not perform Demand-Driven Indexing inside structure symbols. For dynamic predicates, Demand-Driven Indexing is employed only if they consist of Datalog facts; if a clause which is not a Datalog fact is asserted, all dynamically created index tables for the predicate are simply removed and the dindex_on_constant instruction becomes a noop. All this is done automatically, but the user can disable Demand-Driven Indexing in compiled code using an option.

YAP implements Demand-Driven Indexing since version 5. The current implementation supports static code, dynamic code, and the internal database. It differs from the algorithm presented in Sect. 4 in that *all indexing code is generated on demand*. Thus, YAP cannot assume that a dindex_on_* instruction is followed by a try-retry-trust chain. Instead, by default YAP has to search the whole predicate for clauses that match the current position in the indexing code. Doing so for every index expansion was found to be very inefficient for larger relations: in such cases YAP will maintain a list of matching clauses at each dindex_on_* node. Indexing dynamic predicates in YAP follows very much the same algorithm as static indexing: the key idea is that most nodes in the index tree must be allocated separately so that they can grow or shrink independently. YAP can index arguments where some clauses have unconstrained variables, but only for static predicates, as in dynamic code this would complicate support for LU semantics.

YAP uses the term JITI (Just-In-Time Indexing) to refer to Demand-Driven Indexing. In the next section we will take the liberty to use this term as a convenient abbreviation.

7 Performance Evaluation

We evaluate JITI on a set of benchmarks and applications. Throughout, we compare performance of JITI with first argument indexing. For the benchmarks of Sect. 7.1 and 7.2 which involve both systems, we used a 2.4 GHz P4-based laptop with 512 MB of memory. For the benchmarks of Sect. 7.3 which involve YAP 5.1.2 only, we used a 8-node cluster, where each node is a dual-core AMD 2600+ machine with 2GB of memory.

7.1 Performance of Demand-Driven Indexing When Ineffective

In some programs, Demand-Driven Indexing does not trigger[4] or might trigger but have no effect other than an overhead due to runtime index construction. We therefore wanted

[4] In XXX only; even 1st argument indexing is generated on demand when JITI is used in YAP.

Table 1. Performance of some benchmarks with 1st vs. demand-driven indexing (times in msecs)

(a) When JITI is ineffective

Benchmark	YAP 1st	YAP JITI	XXX 1st	XXX JITI
tc_l_io (8000)	13	14	4	4
tc_r_io (2000)	1445	1469	614	615
tc_d_io (400)	3208	3260	2338	2300
tc_l_oo (2000)	3935	3987	2026	2105
tc_r_oo (2000)	2841	2952	1502	1512
tc_d_oo (400)	3735	3805	4976	4978
compress	3614	3595	2875	2848

(b) When JITI is effective

	YAP 1st	YAP JITI	YAP **ratio**	XXX 1st	XXX JITI	XXX **ratio**
sg_cyl	2,864	24	119×	2,390	28	85×
muta	30,057	16,782	1.79×	26,314	21,574	1.22×
pta	5,131	188	27×	4,442	279	16×
tea	1,478,813	54,616	27×	—	—	—

to measure this overhead. As both systems support tabling, we decided to use tabling benchmarks because they are small and easy to understand, and because they are a bad case for JITI in the following sense: tabling avoids generating repetitive queries and the benchmarks operate over extensional database (EDB) predicates of size approximately equal to the size of the program. We used **compress**, a tabled program that solves a puzzle from an ICLP Prolog programming competition. The other benchmarks are different variants of tabled left, right and doubly recursive transitive closure over an EDB predicate forming a chain of size shown in Table 1(a) in parentheses. For each variant of transitive closure, we issue two queries: one with mode `(in,out)` and one with mode `(out,out)`. For YAP, indices on the first argument and try-retry-trust chains are built on all benchmarks under Demand-Driven Indexing. For XXX, Demand-Driven Indexing triggers on no benchmark but the dindex_on_constant instructions are executed for the three **tc_?_oo** benchmarks. As can be seen in Table 1(a), Demand-Driven Indexing, even when ineffective, incurs a runtime overhead that is at the level of noise and goes mostly unnoticed. We also note that our aim here is *not* to compare the two systems, so the **YAP** and **XXX** columns should be read separately.

7.2 Performance of Demand-Driven Indexing When Effective

On the other hand, when Demand-Driven Indexing is effective, it can significantly improve runtime performance. We use the following programs and applications:

sg_cyl. The same generation DB benchmark on a $24 \times 24 \times 2$ cylinder. We issue the open query.

muta. A computationally intensive application where most predicates are defined intentionally.

pta. A tabled logic program implementing Andersen's points-to analysis. A medium-sized imperative program is encoded as a set of facts (about 16,000) and properties of interest are encoded using rules. Program properties are then determined by the closure of these rules.

tea. Another implementation of Andersen's points-to analysis. The analyzed program, the `javac` benchmark, is encoded in a file of 411,696 facts (62,759,581 bytes in total). Its compilation exceeds the limits of the XXX compiler (w/o JITI). So we run this benchmark only in YAP.

As can be seen in Table 1(b), Demand-Driven Indexing significantly improves the performance of these applications. In **muta**, which spends most of its time in recursive predicates, the speed up is only 79% in YAP and 22% in XXX. The remaining benchmarks execute several times (from 16 up to 119) faster. It is important to realize that *these speedups are obtained automatically*, i.e., without any programmer intervention or by using any compiler directives, in all these applications.

7.3 Performance of Demand-Driven Indexing on ILP Applications

The need for Demand-Driven Indexing was originally noticed in inductive logic programming applications. These applications tend to issue ad hoc queries during execution and thus their indexing requirements cannot be determined at compile time. On the other hand, they operate on lots of data, so memory consumption is a reasonable concern. We evaluate JITI's time and space performance on some learning tasks using the Aleph system [17] and the datasets of Fig. 4 which issue simple queries in an extensional database. Several of these datasets are standard in the ILP literature.

Time performance. We compare times for 10 runs of the saturation/refinement cycle of the ILP system; see Table 2(a).The **Mesh** and **Pyrimidines** applications are the only ones that do not benefit much from indexing in the database; they do benefit through from indexing in the dynamic representation of the search space, as their running times improve somewhat with Demand-Driven Indexing.

The **BreastCancer** and **GeneExpression** applications use unstructured data. The speedup here is mostly from multiple argument indexing. **BreastCancer** is particularly interesting. It consists of 40 binary relations with 65k elements each, where the first argument is the key. We know that most calls have the first argument bound, hence indexing was not expected to matter much. Instead, the results show Demand-Driven Indexing to improve running time by more than an order of magnitude. This suggests that even a small percentage of badly indexed calls can end up dominating runtime.

IE-Protein_Extraction and **Thermolysin** are example applications that manipulate structured data. **IE-Protein_Extraction** is the largest dataset we consider, and indexing is absolutely critical. The speedup is not just impressive; it is simply not possible to run the application in reasonable time with only first argument indexing. **Thermolysin** is smaller and performs some computation per query, but even so, Demand-Driven Indexing improves its performance by an order of magnitude. The remaining benchmarks improve from one to more than two orders of magnitude.

Space performance. Table 2(b) shows memory usage when using Demand-Driven Indexing. The table presents data obtained at a point near the end of execution; memory usage should be at the maximum. These applications use a mixture of static and dynamic predicates and we show their memory usage separately. On static predicates, memory usage varies widely, from only 10% to the worst case, **Carcinogenesis**, where the index tables take more space than the original program. Hash tables dominate usage in **IE-Protein_Extraction** and **Susi**, whereas try-retry-trust chains dominate in **Breast-Cancer**. In most other cases no single component dominates memory usage. Memory usage for dynamic predicates is shown in the last two columns; this data is mostly used to store the search space. Observe that there is a much lower overhead in this case. A

Table 2. Time and space performance of JITI on Inductive Logic Programming datasets

(a) Time (in seconds) (b) Memory usage (in KB)

Benchmark	Time			Static code		Dynamic code	
	1st	JITI	**ratio**	Clauses	Index	Clauses	Index
BreastCancer	1,450	88	16×	60,940	46,887	630	14
Carcinogenesis	17,705	192	92×	1,801	2,678	13,512	942
Choline	14,766	1,397	11×	666	174	3,172	174
GeneExpression	193,283	7,483	26×	46,726	22,629	116,463	9,015
IE-Protein_Extraction	1,677,146	2,909	577×	146,033	129,333	53,423	1,531
Mesh	4	3	1.3×	802	161	2,149	109
Pyrimidines	487,545	253,235	1.9×	774	218	25,840	12,291
Susi	105,091	307	342×	5,007	2,509	4,497	759
Thermolysin	50,279	5,213	10×	2,317	929	116,129	7,064

GeneExpression learns rules for yeast gene activity given a database of genes, their interactions, and micro-array gene expression data;

BreastCancer processes real-life patient reports towards predicting whether an abnormality may be malignant;

IE-Protein_Extraction processes information extraction from paper abstracts to search proteins;

Susi learns from shopping patterns;

Mesh learns rules for finite-methods mesh design;

Carcinogenesis, Choline, Pyrimidines try to predict chemical properties of compounds and store them as tables, given their chemical composition and major properties;

Thermolysin also manipulates chemical compounds but learns from the 3D-structure of a molecule's conformations.

Fig. 4. Description of the ILP datasets used in the performance comparison of Table 2

more detailed analysis shows that most space is occupied by the hash tables and by internal nodes of the tree, and that relatively little space is occupied by `try-retry-trust` chains, suggesting that Demand-Driven Indexing is behaving well in practice.

8 Concluding Remarks

Motivated by the needs of applications in the areas of inductive logic programming, program analysis, deductive databases, etc. to access large datasets efficiently, we have described a novel but also simple idea: *indexing Prolog clauses on demand during program execution*. Given the impressive speedups this idea can provide for many LP applications, we are a bit surprised similar techniques have not been explored before. In general, Prolog systems have been reluctant to perform code optimizations during runtime and our feeling is that LP implementation has been left a bit behind. We hold that this should change. Indeed, we see Demand-Driven Indexing as only a first, very successful, step towards effective runtime optimization of logic programs.

As presented, Demand-Driven Indexing is a hybrid technique: index generation occurs during runtime but is partly guided by the compiler, because we want to combine

it with compile-time WAM-style indexing. More flexible schemes are of course possible. For example, index generation can be fully dynamic (as in YAP), combined with user declarations, or driven by static analysis to be even more selective or go beyond fixed-order indexing. Last, observe that Demand-Driven Indexing fully respects Prolog semantics. Better performance can be achieved in the context of one solution computations, or in the context of tabling where order of clauses and solutions does not matter and repeated solutions are discarded.

Acknowledgments. Vítor Santos Costa was partially supported by CNPq and would like to acknowledge support received while visiting at UW-Madison and the support of the YAP user community. This work has been partially supported by MYDDAS (POSC/EIA/59154/2004) and by funds granted to LIACC through the Programa de Financiamento Plurianual, Fundação para a Ciência e Tecnologia and Programa POSC.

References

1. Warren, D.H.D.: An abstract Prolog instruction set. Tech. Note 309, SRI International (1983)
2. Santos Costa, V., Damas, L., Reis, R., Azevedo, R.: YAP User's Manual (2002)
3. Carlsson, M.: On the efficiency of optimising shallow backtracking in compiled Prolog. In: Levi, G., Martelli, M. (eds.) Proceedings of the Sixth ICLP, pp. 3–15. MIT Press, Cambridge (1989)
4. Demoen, B., Mariën, A., Callebaut, A.: Indexing in Prolog. In: Lusk, E.L., Overbeek, R.A. (eds.) Proceedings of NACLP, pp. 1001–1012. MIT Press, Cambridge (1989)
5. Wielemaker, J.: SWI-Prolog 5.1: Reference Manual. SWI, University of Amsterdam, Roetersstraat 15, 1018 WB Amsterdam, The Netherlands (1997-2003)
6. Sagonas, K.F., Swift, T., Warren, D.S., Freire, J., Rao, P.: The XSB Programmer's Manual. State University of New York at Stony Brook (1997)
7. Troncon, R., Janssens, G., Demoen, B., Vandecasteele, H.: Fast frequent quering with lazy control flow compilation. Theory and Practice of Logic Programming (2007) (to appear)
8. Hickey, T., Mudambi, S.: Global compilation of Prolog. JLP 7(3), 193–230 (1989)
9. Van Roy, P., Demoen, B., Willems, Y.D.: Improving the execution speed of compiled Prolog with modes, clause selection and determinism. In: Ehrig, H., Levi, G., Montanari, U. (eds.) CAAP 1987 and TAPSOFT 1987. LNCS, vol. 249, pp. 111–125. Springer, Heidelberg (1987)
10. Zhou, N.F., Takagi, T., Kazuo, U.: A matching tree oriented abstract machine for Prolog. In: Warren, D.H.D., Szeredi, P. (eds.) ICLP90, pp. 158–173. MIT Press, Cambridge (1990)
11. Dawson, S., Ramakrishnan, C.R., Ramakrishnan, I.V., Sagonas, K., Skiena, S., Swift, T., Warren, D.S.: Unification factoring for the efficient execution of logic programs. In: Conference Record of POPL'95, pp. 247–258. ACM Press, New York (1995)
12. Ramakrishnan, I.V., Rao, P., Sagonas, K., Swift, T., Warren, D.S.: Efficient access mechanisms for tabled logic programs. Journal of Logic Programming 38(1), 31–54 (1999)
13. Kliger, S., Shapiro, E.: A decision tree compilation algorithm for FCP(—,:?). In: Proceedings of the Fifth ICSLP, pp. 1315–1336. MIT Press, Cambridge (1988)
14. Somogyi, Z., Henderson, F., Conway, T.: The execution algorithm of Mercury, an efficient purely declarative logic programming language. JLP 26(1-3), 17–64 (1996)
15. Hermenegildo, M.V., Puebla, G., Bueno, F., López-García, P.: Integrated program debugging, verification, and optimization using abstract interpretation (and the Ciao system preprocessor). Science of Computer Programming 58(1–2), 115–140 (2005)
16. Carlsson, M.: Freeze, indexing, and other implementation issues in the WAM. In: Lassez, J.L. (ed.) Proceedings of the Fourth ICLP, pp. 40–58. MIT Press, Cambridge (1987)
17. Srinivasan, A.: The Aleph Manual (2001)

Design, Implementation, and Evaluation of a Dynamic Compilation Framework for the YAP System

Anderson Faustino da Silva and Vítor Santos Costa

COPPE - Systems Engineering
Federal University of Rio de Janeiro
Caixa Postal: 68511 – 21941-972 Rio de Janeiro, RJ, Brazil
{faustino,vitor}@cos.ufrj.br

Abstract. We propose dynamic compilation for Prolog, in the style of Just-In-Time compilers. Our approach adapts to the actual characteristics of the target program by **(i)** compiling only the parts of the program that are executed frequently, and **(ii)** adapting to actual call patterns. This allows aggressive optimization of the parts of the program that are really executed, and better informed heuristics to drive these optimizations. Our compiler does need to support all features in the language, only what is deemed important to performance. Complex execution patterns, such as the ones caused by error handling, may be left to the interpreter. On the other hand, compilation is now part of the run-time, and thus incurs run-time overheads. We have implemented dynamic compilation for YAP system. Our initial results suggest that dynamic compilation achieves very substantial performance improvements over the original interpreter, and that it can approach and even out-perform state-of-the-art native code systems. We believe that we have shown that dynamic compilation is worthwhile and fits naturally with Prolog execution.

1 Introduction

Most Prolog systems rely on interpreters of Warren's Abstract Machine, or WAM [1]. Towards better performance, previous research on high-performance Prolog implementations proposed the design of native-code compilers [2,3,4]. Such systems can be complex and hard to maintain, as they need to cope with all the complexity in logic program execution. Thus, global analysis and/or user annotations have so far been considered essential for high performance [5,4]. In fact, the Mercury Project took this argument to what seems to be its logical conclusion: the Mercury designers argued that issues such as mode declarations and determinism are a fundamental problem with Prolog, and therefore proposed a new language where such declarations are always required.

In this approach we propose an alternative approach to the efficient implementation of Prolog. Our goal is to build a system that would be easy to maintain while achieving good performance. In order to do so, arguably we need to know what are the most important components of programs (the so-called *hot-spots*)

V. Dahl and I. Niemelä (Eds.): ICLP 2007, LNCS 4670, pp. 410–424, 2007.

and how they are called. Compiling such program fragments should be the key to good performance; only compiling them should be the key to simplicity.

Our approach is based on *dynamic compilation*, an idea that has become the fundamental principle in the compilation of high performance virtual machines, such as Java Virtual Machine. Dynamic compilers, and namely dynamic compilers for Java [6,7,8], only compile code after the code is first called. The advantage is that compilation can benefit from *dynamic information*, such as the instantiation of virtual methods in an object oriented language. The drawback is that the compilation is now included in the program's execution time. Researchs showed that best performance is achieved when the compiler is very selective about which parts of the program it decides to compile and when and how it decides to compile them. More specifically, it should compile some parts only if the extra time spent in compilation can be amortized by the performance gain expected from the compiled code. Once such *hot-spots or hot-regions* are detected, the dynamic compiler must be very aggressive in identifying good opportunities for optimizations that can achieve good performance.

We believe that *dynamic approach* naturally fits the Prolog language. Techniques such *Type or mode feedback*, that read type or mode information from runtime data, can alleviate the need for extensive global analysis. Hot-spot selection may reduce compile-time overhead and reduce complexity by avoiding complex structures that are hard to analyze and to generate good code for. To prove our thesis, we developed an dynamic compilation framework for the YAP System [9,10], YAPc. YAPc combines the YAP interpreter with a dynamic compiler that performs runtime feedback-directed optimizations. A low-overhead, continuously operating sampling profiler identifies program hot regions for optimization. Our experimental results show that this approach provides significant advantages in terms of performance, besides addressing the issue of maintainability.

This paper makes the following contributions:

- System architecture: we present an architecture for a simple, but efficient dynamic compilation framework for the YAP System with a mixed mode interpreter. We believe that our architecture can be easy to maintain. Experimental data is presented for both performance and code quality.
- Profiling techniques: we present a program profiling mechanism that combines two different techniques. One is a continuously operating, lightweight sampling profiler for detecting program hot, and the other is a type feedback profiler that collects detailed information for the hots gathered by the first profiler.
- Compilation techniques: we present compilation techniques that decrease both the runtime pauses runtime and overhead of native calls.

The rest of this paper is organized as follows. The next section summarizes related work, comparing our system to prior systems. Section 3 describes the overall architecture of our system. Section 4 describes our compilation scheme. Section 5 presents some results to show the effectiveness of our framework. Finally, we conclude in section 6.

2 Related Work

Native-code compilation is an old idea in Prolog. The first Prolog compiler, the DEC-10 compiler, generated native code [11]. Other native code compilers include BIM-Prolog [12], and the work by Komatsu et al [13], who demonstrated that specialized hardware is not essential for high-performance execution of Prolog.

Aquarius [5] and Parma [14] were particularly influential in the development of Prolog compilation technology. Both use a fine-grained instruction set to allow more effective optimizations. The systems showed great performance, as measure on a number of standard (small) benchmarks. Both systems relied on global analysis to obtain type, mode, and reference-chain length information [15]. Such information can enable aggressive optimizations, especially when analysis is able to derive types that have only an operational meaning, such as dereference (reference chains), trailing, and aliasing-related types (uninitialized variables). On the other hand, it significantly increases compile time and is not guaranteed to generate precise information.

Unfortunately, Aquarius and Parma were never widely used. Other native compilers, such as the SICStus native code compiler [16], or the WAMCC [17] and its successor, GNU-Prolog [18], do not perform such analysis, which arguably limits the performance benefits they can offer. Recent work in CIAO [4] again demonstrated the usefulness of doing type analysis and computing mode information to improve compiled code. The new CIAO compiler uses this information to remove type and mode checks, and to use specialized versions of critical builtins. The initial results are very promising, as their system benefits from substantial progress in global analysis.

One alternative to global analysis is to ask the user for type, mode and even determinacy declarations. Mercury *requires* such declarations for every procedure. As such, Mercury can use this information at all levels in its compiler. Moreover, because it is strongly moded, Mercury can avoid some of the complexity associated with mode verification and dereferencing in Prolog. Performance results do show higher performance than Prolog [19]. On the other hand, Mercury sacrifices some of the expressiveness in Prolog. Languages such as HAL [20] try to bridge the two approaches by using the constraint framework, and introducing Herbrand domains.

Maybe the work closest to us is partial translation in BinProlog [21], which also compiles selected sequences of instructions to native code. Partial translation was originally developed in the context of *binarization*, an algorithm where one can generate long sequences of straight-line code to create continuations. Such code sequences are amenable to optimization. One difference with our approach is that we do not necessarily guarantee *compositionality*, instead we trust the semantics of Prolog. A second difference is that we use dynamic translation, whereas BinProlog relies on instruction sequence length to choose which sequences to compile top `C`.

2.1 Dynamic Optimization Systems

Just-in-time (JIT) or dynamic compilation stems from work in the object-oriented languages. Arguably, the seminal work in JIT technology is the Self-93 [22] compiler. Self-93 pioneered on-line profile-directed adaptive compilation systems. Before execution, Self-93 compiles its program with a simple compiler. At run-time, a much more sophisticated compiler is called if a method is deemed *critical*. The goal of this second compiler was to avoid long compilation pauses and to improve responsiveness for interactive programs. Note that compilation could take advantage of type feedback information for receiver class distributions by profiling the initial code.

Hank [23] described the problems of using a function-based compilation strategy. The work proposed region formation to perform an aggressive inlining pass first, followed by a partitioning phase that created new regions based on heuristics from offline profile results. Way [24] improved this region formation algorithm to make it scalable by combining region selection and the inlining process, thus reducing the compilation time and memory requirements considerably.

The notion of mixed execution of interpreted and compiled code was considered as a continuous compiler or smart JIT approach by Plezbert and Cytron [25], and the study of three-mode execution using an interpreter, a fast non-optimizing compiler, and a fully optimizing compiler was reported by Agesen and Detlefs [26]. In both of these papers, it was proven that there is a performance advantage by using an interpreter in the system for balancing the compilation cost and resulting code quality.

Arguably, most of the recent work in JIT compilation has taken place within the Java community. Jalapeño [8] is an example research Java Virtual Machine that implements a multilevel recompilation scheme using a baseline compiler and a optimizing compiler. The optimizing compiler compiles entire methods. HotSpot [27] is a popular JVM product implementing an adaptive optimization system. Two Sun JIT compiler are available, both compile from Java bytecode. The Client compiler is a simpler compiler and is optimized for speed, whereas the Server compiler is more aggressive, and designed for long-running programs. Following SELF-83, the HotSpot Server Compiler [7] employs the *uncommon trap mechanism* to avoid code generation for uncommon cases. In this mechanism, the system falls back to the interpreter at a safe point if an uncommon path is actually taken; hence, only common paths need to be compiled.

3 System Architecture

We propose dynamic compilation system towards a robust system that could achieve close to the best possible performance. The overall architecture of our system is as depicted in figure 1. Notice we compile a clause at a time. All clauses in the program are managed by the *clause manager*. Clauses are first compile for interpretation, but the code is set up to to also *count* clause informations. The *interpreter* will use these counters to decide on whether it should make a

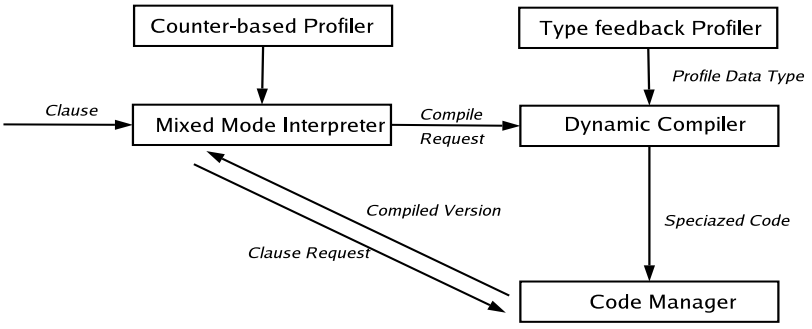


Fig. 1. System architecture of our dynamic system

compilation request. The compiler will then take advantage of mode information, as we shall see. We describe each of the major components of the system next.

3.1 Mixed Mode Interpreter

Most clauses in a Prolog program are not frequently called. Arguably, compiling all such clauses is inefficient both in terms of compilation time and code space. *Mixed mode interpretation* (MMI) allows the efficient mixed execution of interpreted and compiled code by sharing the execution stack between compiled code and the emulator.

Initially, every clause is interpreted by the MMI. Each clause receives a *counter* that is decremented at every call. When the count reaches zero, it is known that the clause has been invoked frequently, and JIT compilation is triggered. Originally, we envisioned counters as a first step, to be used only until a better solution was found. However, the simple counter-based approach seems to work well so far [28,29].

3.2 Compilation Control

The compilation controller receives as input a clause that a clause with a counter set to zero and decides whether the clause should be compiled. Currently, the controller uses a simple rule to make this decision: it only compiles clauses whose size exceeds a pre-defined threshold. The assumption underlying this rule is that small clauses don't need to be optimized, because these will not improve performance.

3.3 Type-Feedback-Based Profiler

Compilation should have representation-level type information on the current arguments. Ideally, we should know the types and modes of clause usage. Of course, the interpreter cannot give us that. On the other hand, it can give us the current types and modes *for the current call*. If we assume these types and

modes are the most often used ones, and this is likely to be the case, then the compiler can generate code specialised for these types and modes, and only for them. In other words, and in the spirit of the uncommon trap mechanism, the compiler will still need to *test* whether the call is being called with the current code. But, if these tests succeed, it can generate specialized code. If these tests fail, compiler will need to fall back to the interpreter.

Our system therefore first inspects the internal state of the system, inferring the current types of the input arguments. It then stores this information in a way that is usable by the compiler.

3.4 The Compiler

We implemented a new compiler for the YAP system. The compiler is written in `C` and supports a simple pointer-based language that fits naturally WAM description code.

The dynamic compiler first parses its input, a set of instructions of the WAM-derived YAAM abstract machine [30], into an abstract syntax tree (AST), thus reducing instruction granularity. Note that the full AST for a clause can be quite complex. Next, the parser uses the *type-feedback* to specialize the control-flow graph and to decide at which points it should perform inlining. The parser concludes by transforming the AST into an intermediate low level representation. The *instruction selector* next finds the appropriate machine instructions to implement the low level representation. Following this step, the *register allocator* allocates real machine registers. After real registers are allocated, the *peephole optimizer* performs low-level optimizations on the code, and finally the compiler generates native code. When the compiler finishes the compilation process, the native code is installed into run-time system, and scheduled for execution.

Our compiler implements several optimizations such as region selection, inlining, recursive-call elimination, coalescing, and peepholing. Some of these will be described in more detail the next section.

The bulk of the compiler effort lies in between the two halves of a traditional compiler. This "middle half" of the compiler performs the representation-level type analysis and region selection that bridges the semantic gap between the high-level polymorphic program input to the compiler and the lower-level monomorphic version of the program suitable for the optimizations performed by a traditional compiler back-end.

3.5 Exception Handling

We have shown that control will return back to the emulator if a clause is called with unexpected modes. In general, we call such situations *exceptions*: they can appear in the compiled code if arguments are called with an unexpected instantiation or unexpected types. YAPc benefits from the referential transparency of Prolog to address this problem. If we compile pure code, then if the Prolog code partially succeeded for G with bindings θ, then it should also succeed for a call $G\theta$. This means that the native code running G, should obtain the same results

as the interpreter running $G\theta$. Therefore, it should be sufficient to reexecute the code from scratch in the interpreter.

In the practice, there are two technical problems. The first problem is often found in JIT systems, and results from the fact that the compiler may maintain data at a state that is inconsistent with the external environment. In our case, this is easily shown by pondering on how the WAM reuses registers. Consider the recursive clause in `append/3`:

```
append([X|L], T, [X|NL]) :- append(L, T, NL).
```

Most WAM compilers will generate code of the form:

```
get_list   A1
unify_var  X4
unify_var  X1
get_list   A3
unify_val  X4
unify_var  X3
```

notice that the code destroys the contents of register `A1` before starting unification with `A3`. If the code needs to return to the interpreter (ie, because `A3` was called instantiated, but was expected to be free), the interpreter would find `A1` set to the tail of the list and `A3` set to the original, resulting in wrong execution. The compiler should take care so that the interpreter always finds consistent arguments.

There are at least two solutions for this problem. First, the system can save the registers before calling the compiled code, and if it an exception occurs restore these registers back. Whatever their state, they will be correct. Alternatively, compiled code could try to guarantee that the external registers are consistent at points of a possible exception. Such points are called *safe points*. In this case, a simple strategy is only to write in the WAM argument registers at the very end of the code, when we are sure no exceptions can occur. These solutions are similar to what is used in JITs [7,6].

In the above example, the register `A1` could be kept in the stack and recovered if the `A3` test failed. Alternatively, the register `X1` might not be aliased with `A1` until the return to interpreted code. The first solution is simpler to implement, but the second solution decreases the overhead in the interface of the two systems. Currently, YAPc implements the first solution: we will experiment with the second approach in future work.

The second problem arises from meta-logical instructions, such as cut and `var/1`. These instructions can break referential transparency. The program:

```
a(X) :- var(X), !, X=2, exception.
a(X).
```

may not work correctly if we allow bindings to X. The problem is that the first (compiled) call to `a(X)` would set $X = 2$. At the second (interpreted) call, the interpreter would use $a(2)$. This call will fail the `var/1` meta-test, backtrack to the second clause, and succeed with an unexpected result.

A first solution is never to compile code containing cuts or meta-predicates unless we can prove that the code is called with sufficient instantiation. This can be improved by creating a *safe point* at every cut, that is, by making sure all the registers needed by the interpreter are correctly set before a cut.

4 The Compiler

In this section, we discuss a number of issues on how clauses are compiled in some more detail. First, we do not always compile the full clause. Second, we discuss the region selection algorithm. Last, if a clause is called in a variety of ways we may have a number of compiled versions of the same original clause.

4.1 Partial Compilation

In practice, we only completely compile clauses if they are *facts*, ie, if they terminate with a *procceed*. Otherwise, compilation terminates at the first instruction before an *execute* or a *call*. The compiler therefore only partially compiles the clause.

This decision was taken on the following grounds:

Compilation Simplicity: This is the simplest straight-line fragment of code we can compile independently of other code.

Interface Simplicity: This reduces to the very minimum the actual interface between compiler and interpreter. If we were to compile to native code `call` instruction, the emulator would need to know that the environment was created by native code.

There is an exception to this rule. Consider again the `append/3` example

```
append([], L, L).
append([X|L], T, [X|NL]) :- append(L, T, NL).
```

In this case, a simple tail-recursive procedure, we know the procedure is calling itself. It therefore makes sense not to stop at the `call` instruction. Instead, YAPc follows the indexing code and compiles the loop as a single unit.

A more ambitious approach will be to inline the clause called for the instruction execute/call (in the style of the VAM [31]). However, as it will be seen in section 6, so far partial compilation has shown itself to be a good choice in keeping the system light and with good performance.

4.2 Region-Based Compilation

Programs often contain rarely or never executed paths. Compiling such paths can cause adverse effects that reduce the effectiveness of code generated. For example, an optimizing compiler can spend a lot of time applying aggressive optimizations in rarely executed code. This can increase both compilation time (a major problem in dynamic compilation) and code size, with little or no benefit.

If we can eliminate from the compilation target the fragments that are rarely or never executed, we can focus the optimization efforts only on non-rare paths, making the optimization process both faster and more effective.

In our work, we implement a region-based compilation technique. We define *region* to be the the straight-line code segment that corresponds to the current mode for the current goal. All other portions of code are removed. Our algorithm works as follows.

First, we assume each clause is represented as a control flow graph (CFG) with a single entry block and a single exit block. The algorithm begins basic blocks as either *rare* or non-rare by employing both heuristics and dynamic profile results. We use the following heuristics:

- A block that performs type checking is rare.
- A block that processes errors is rare.
- A block that contains unresolved or uninitialized cell references is rare.
- A block that ends with a return instruction is non-rare.

If the dynamic profile information shows that a block is never executed, then we mark the block as rare. Otherwise, the block is marked as non-rare. Dynamic profiling information has priority over the static heuristics.

An iteration phase propagates this information through backward data flow until it converges for all of the basic blocks. After this, the compiler traverses the basic blocks to determine the transitions from non-rare blocks to rare blocks, and marks those locations as rare block entry points. After performing live analysis to find the set of live variables at each rare block entry point, the compiler generates a new region exit basic block for each entry point and replaces the original entry block by redirecting its incoming control flow edge to the new block. All the rare blocks that originally existed following rare block entry points are no longer reachable from the top of the method and thus eliminated in the succeeding control flow cleanup phase.

During region selection, the compiler also inlines functions at the clause's executable paths. Currently, the compiler only performs inlining of unification and dereferencing as such functions are key to performance.

4.3 Multiple Version Code Management

Compiled clauses are registered at a runtime system called the code manager, which controls and manages all compiled codes by associating them with their corresponding clause structures and with information about the specialization context. This means all future invocations to this clause through indirect clause lookup will be directed to the new version of the code, instead of the existing interpreted code.

The system maintains an list of specialized code for each clause. This is necessary because call patterns can change during execution. If so, the clause can deserve more compilation. In order to decide which version should be run, the code manager must be able to inspect the runtime first.

Currently, the system does not need a mechanism for on-stack replacement [32], the technique of dynamically rewriting stack frames from one version to another, as Prolog procedures are usually quite small, so we do not lose much if we postpone the invocation of compiled code until the next call.

5 Evaluation

This section presents some experimental results showing the effectiveness of our dynamic system. We outline our experimental methodology first, describe the benchmarks used for the evaluation, and then present and discuss our performance results.

5.1 Benchmarking Methodology

All the performance results presented in this section were obtained using a Pentium IV 2.4 MHz uni-processor with 1 GB memory, running Linux, and using the latest version of YAP Prolog System [33]. The benchmarks we chose for evaluating our JIT compiler are shown in table 1. These benchmarks are widely used by the Prolog community [14,34].

Table 1. Benchmarks description

Program	**Description**
Append	Inserts a list of 50 integers.
Crypt	Solves a cryptoarithmetic puzzle.
Deriv	Symbolically differentiates four functions of a single variable.
Fib	Calcules the 100'is number of Fibonacci's series.
Hanoi	Hanoi tower.
Merge Sort	Mergesorts a list of 50 integers using difference lists.
MMatrix	Multiples two matrices of 10x10 integers.
Naive Reverse	Reverses a list of 50 elements using the naive algorithm.
Poly	Symbolically raises 1+x+y+z to the tenth power.
Prime	Finds all primes up to 100.
Quick Sort	Quicksorts a list of 50 integers using difference lists.
Queens	Finds all safe placements of 8 queens on a 8x8 chessboard.
Query	Finds countries with approximately equal population density.
Tak	An artificial benchmark heavily recursive and does lots of simple integer arithmetic.
Zebra	Solves a puzzle.

The following configuration and parameters were used throughout the experiments.

– Each benchmark was invoked 100000 times, therefore the execution time for each benchmark is $100000 \times \textit{benchmark time}$.

- The execution time shown is the average among five rounds.
- The threshold in the mixed mode interpreter to initiate dynamic compilation was set to 1000.
- In *No Opt* mode the following optimizations were disabled: *region selection*, *inlining*, *coalescing*, and *peepholing*.
- In *Full Opt* mode all optimizations were enabled; this is the default in our compiler.
- Garbage collection was disabled throughout.

5.2 Dynamic Compilation

In order step to evaluate total performance we compare YAPc against the standard emulator. Table 2 summarizes the results. The second column contains the execution times (in seconds) of programs running in interpreted mode. The third column corresponds the execution times using our dynamic framework without optimizations. The fourth column corresponds the execution times using our dynamic framework with all optimizations. The other columns show the speedup.

Table 2. Execution performance

Program	**Execution Time**			**Speedup**	
	Interpreter Only	**Mixed Mode**		*No Opt*	*Full Opt*
		No Opt	*Full Opt*		
Append	0.18	0.07	0.05	2.57	3.50
Crypt	29.04	10.02	8.25	2.89	3.52
Deriv	0.99	0.55	0.31	1.80	3.13
Fib	2.39	1.52	0.84	1.57	2.85
Hanoi	39.25	15.32	10.78	2.56	3.64
Merge Sort	8.34	3.78	1.49	2.21	5.60
MMatrix	21.55	11.56	7.38	1.86	2.92
Naive Reverse	3.57	1.23	0.83	2.90	4.28
Poly	333.52	123.25	44.77	2.71	7.45
Prime	13.66	7.11	3.50	1.92	3.90
Quick Sort	3.77	1.96	1.47	1.92	2.57
Queens	14.83	10.88	5.60	1.36	2.65
Query	14.50	6.95	3.48	2.09	4.17
Tak	1079.89	520.12	326.25	2.08	3.31
Zebra	272.08	120.58	45.73	2.26	5.95

As we expected, mixed mode execution is faster than interpreted-only. The YAP Prolog System is between 2 to 7 times slower than mixed mode execution with all optimizations on these benchmarks. The smallest difference is for the Quick Sort program: mixed mode is only 2.57 times faster than interpreted-only. Poly has the major difference: 7.45 times faster than in interpreted-only. Optimizations are crucial to performance, without optimizations we gain only a factor of two. This is for two reasons:

1. Emulator overhead: it is very short, besides our system compiles only clauses (some clauses).
2. Much longer compilation times: region selection speeds up the compilation time by a factor between 5 to 17 times.

We found our dynamic compiler to be very fast. Compilation time varies from 0.0021 to 0.0431 seconds. Compilation speed depends on the characteristics of the program. If the compiler process a few clause only, this decreases compilation time, and results in very compact code. Longer compiler time result from two factors: the complexity of YAAM instructions, this is the case of Tak, and the need to compile a large number clauses, this is the case of Poly. In this last case, nine clauses were compiled.

Since translation is taking place during program execution, the compilation time is now part of the execution time, and thus an *overhead*. On the other hand, the compilation time is very small, resulting in a low overhead, that does not decrease the performance of runtime system.

The performance obtained by using dynamic compilation is much better, and performing well for all programs. Best speedups are obtained in programs with a large number of clause. We believe our results were possible because we benefit from the runtime information from the MMI, allowing more aggressive optimization of the parts of the program that are actually executed often.

5.3 Code Quality

Table 3 shows the statistics for code quality when running the benchmarks with the same configuration described in methodology, with and without region selection. The second to fifth rows are code statistics, showing the total number of bytes, the total number of assembly instructions, the total number of spills, and the total number of fetches. In this table, bold statistics were collected with our technique on, and the others with our technique off.

The native code we generate is very compact, Append and Naive Reverse are the best cases. For these, all temporaries had been kept in machine registers, and we did not need to represent registers in memory. Other programs do suffer register spilling.

Both region-based selection and type-feedback profiler compact a considerable portion of code. This occurs because our techniques removes several type checks, and eliminates some rare paths. Using region selection, code size decreases from 2 to 6 times. And the compiler needs few instructions to represent the code specialized for the running program, ranging from 64 to 1138 instructions. As expected in a register-limited architecture, our compiler does needs to represent temporaries in memory. The exceptions are for Append, Naive Reverse.

Type-feedback is key to our good results: it speeds up compilation time (between 2.25 to 17.22 fold) and results in much more compact code, with less register pressure.

Table 3. Code quality characteristics

Program	**Native Code**			
	Bytes	Instructions	Spills	Fetches
Append	1623(**224**)	498(**64**)	37(**0**)	106(**0**)
Crypt	6091(**1218**)	2030(**427**)	112(**26**)	256(**52**)
Deriv	5644(**1044**)	1652(**407**)	98(**20**)	189(**48**)
Fib	1845(**606**)	899(**238**)	38(**16**)	86(**42**)
Hanoi	1596(**828**)	460(**225**)	33(**19**)	60(**19**)
Merge Sort	3008(**1524**)	756(**402**)	89(**42**)	135(**59**)
MMatrix	5325(**1026**)	1365(**321**)	125(**28**)	725(**54**)
Naive Reverse	2265(**464**)	679(**126**)	58(**0**)	140(**0**)
Poly	12984(**7792**)	5984(**1012**)	296(**62**)	848(**95**)
Prime	6292(**966**)	1972(**346**)	138(**39**)	456(**66**)
Quick Sort	5657(**1723**)	1743(**520**)	142(**23**)	348(**72**)
Queens	6920(**4265**)	2088(**892**)	105(**85**)	289(**125**)
Query	3510(**2133**)	1044(**602**)	80(**52**)	189(**78**)
Tak	2340(**1427**)	696(**415**)	44(**31**)	116(**55**)
Zebra	8293(**4476**)	2273(**1138**)	142(**64**)	434(**177**)

6 Conclusions

We have described the design and implementation of YAPc, a new dynamic compilation framework for Prolog. Our work was motivated by good results achieved by the dynamic compilation of indexing code [35]. YAPc addresses the problem of clause compilation and is based on the idea of Hot-Spot compilation: we only compile often used regions in the program, and use the *uncommon trap mechanism* to address unexpected modes. Region selection relies on a simple counter mechanism to select critical procedures, and on *type or mode feedback* to select the main paths in those procedures.

Performance results show that the system can achieve quite good performance as compared to interpreted-only approach. Our results show significant speedups on a number of benchmarks over a state-of-the-art interpreter. In fact, in some cases YAPc did obtain similar or even superior performance to the CIAO compiler and to Mercury [10]. Moreover, results show low compilation overheads. Therefore, we believe that our work shows that dynamic compilation can be used to execute logic programs efficiently.

We believe our dynamic framework will be a major contribution to the future of the YAP Prolog System, and hope it will the ideas will be useful throughout the logic programming community. In the future, we will continue to improve the system. One major area of ongoing research is how to dynamically change compilation strategies as we go along, given full execution data. Ideally, it should be possible to re-recompile clauses, as execution goes along. A second area is how we could benefit from global analysis data, if such data is available, to achieve the best of both worlds. A third area is to better integrate compilation of several

distinct clauses as a single unit. Last, it would make sense to cache compiled code segments so that they can be reused in later runs.

Acknowledgements. Anderson Silva thanks Claudio Amorim, Marcelo Lobosco and the LCP team for their support throughout this work. Vítor Santos Costa was partially supported by CNPq. This work has been partially supported by funds granted to LIACC through the Programa de Financiamento Plurianual, Fundação para a Ciência e Tecnologia and Programa POSIC. We thank the referees for their very insightful comments.

References

1. Warren, D.H.D.: An Abstract Prolog Instruction Set. Technical Report 4776, Artificial Intelligence Center, SRI International (1983)
2. Haygood, R.C.: Native code compilation in SICStus Prolog. In: Van Hentenryck, P. (ed.) Proceedings of the Eleventh International Conference on Logic Programming, MIT Press, Cambridge (1994)
3. Diaz, D., Codognet, P.: The GNU Prolog System and its Implementation. In: SAC (2), pp. 728–732 (2000)
4. Morales, J., Carro, M., Hermenegildo, M.: Improved Compilation of Prolog to C Using Moded Types and Determinism Information. In: Proceedings of the Colloquium on Implementation of Constraint and Logic Programming Systems, pp. 197–212 (2003)
5. Van Roy, P.: Can Logic Programming Execute as Fast as Imperative Programming? PhD thesis, University of California, Berkeley, California, USA (1990)
6. Suganuma, T., Ogasawara, T.: Overview of the IBM Java Just-in-Time Compiler. IBM Systems Journal 39(1), 66–76 (2000)
7. Paleczny, M., Vich, C., Click, C.: The Java HotSpot Server Compiler. In: Proceedings of the Java Virtual Machine Research and Technology Symposium, pp. 1–12 (2001)
8. Alpern, B., Attanasio, C.R.: The Jalapeño Virtual Machine. IBM Systems Journal 39(1), 211–238 (2000)
9. Silva, A., Santos Costa, V.: Design of the YAPc Compiler: An Optimizing Compiler for Logic Programming Languages. In: Proceedings on 10th Brazilian Symposium on Programming Languages, Itatiaia-RJ, Brasil (2006)
10. Silva, A.: Design and Implementation of the YAPc Compiler: An Optimizing Compiler for Logic Programming Language. PhD thesis, Federal University of Rio de Janeiro - Brazil (2006)
11. Warren, D.H.D.: Implementing Prolog - Compiling Predicate Logic Programs. Technical Report 39-40, Department of Artificial Intelligence, University of Edinburgh (1977)
12. Mariën, A.: Improving the Compilation of Prolog in the Framework of the Warren Abstract Machine. PhD thesis, Katholiek Universiteit Leuven (1993)
13. Tamura, N.: Knowledge-Based Optimization in Prolog Compiler. In: Proceedings of the Computer Society Fall Joint Conference, pp. 237–240 (1986)
14. Taylor, A.: Parma - Bridging the Performance GAP Between Imperative and Logic Programming. Journal of Logic Programming 29(1-3), 5–16 (1996)
15. Cousot, P., Cousot, R.: Abstract Interpretation and Application to Logic Programs. Journal of Logic Programming 13(2-3), 103–179 (1992)

16. Haygood, R.C.: Native Code Compilation in SICStus Prolog. MIT Press, Cambridge (1994)
17. Codognet, P., Diaz, D.: Compiling Prolog to C. In: Proceedings of the International Conference on Logic Programming, pp. 317–331 (1995)
18. Diaz, D., Codognet, P.: Design and Implementation of the GNU Prolog System. Journal of Functional and Logic Programming 13(4), 451–490 (2001)
19. Henderson, F., Somogyi, Z.: Compiling Mercury to High-Level C Code. In: Proceedings of Computational Complexity. pp. 197–212 (2002)
20. Demoen, B., de la Banda, M.G., Harvey, W., Marriott, K., Stuckey, P.J.: Herbrand Constraint Solving in HAL. In: Proceedings of ICLP99, pp. 260–274. MIT Press, Cambridge (1999)
21. Tarau, P., DeBosschere, K., Demoen, B.: Partial Translation: Towards a Portable and Efficient Prolog Implementation Technology. Journal of Logic Programming 29(1-3), 65–84 (1996)
22. Chambers, C.: The Design and Implementation of the Self Compiler: an Optimizing Compiler for Objetct-Oriented Programming Languages. PhD thesis, Department of Computer Science, Stanford University (1992)
23. Rank, R.E., Hwu, W., Ran, B.R.: Region-Based Compilation: An Introduction and Motivation. In: Proceedings of the International Conference on Microarchitecture, pp. 158–168 (1995)
24. Way, T., Breech, B., Pollock, L.: Evaluation of a Region-Based Partial Inlining Algorithm for an ILP Optimizing Compiler. In: Proceedings of the Conference on Parallel Architecture and Compilation Technique, pp. 24–36 (2000)
25. Plezbert, M.P., Cytron, R.K.: Does Just-In-Time = Better Late Than Never? In: Proceedings of the Symposium on Principles of Programming Language, pp. 120–131 (1997)
26. Agesen, O., Detlefs, D.: Mixed-mode Bytecode Execution. Technical Report SMLI TR-2000-87, Sun Microsystems (2000)
27. MicroSystems, S.: The Java HotSpot Virtual Machine. Technical report, Sun Developer Network Community (2003)
28. Silva, A., Santos Costa, V.: An Experimental Evaluation of JAVA JIT Technology. In: Proceedings on 9th Brazilian Symposium on Programming Languages, Recife, Brasil (2005)
29. Silva, A., Santos Costa, V.: Our Experiences with Optimizations in Sun's Java Just-in-time Compilers. In: Proceedings on 10th Brazilian Symposium on Programming Languages, Itatiaia-RJ, Brasil (2006)
30. Lopes, R.N.: Execução de Prolog com Alto Desempenho. Master's thesis, Universidade do Porto, Porto, Portugal (1996)
31. Krall, A.: The vienna abstract machine. The Journal of Logic Programming 1-3 (1996)
32. Fink, S., Qian, F.: Design, Implementation and Evaluation of Adaptive Recompilation With On-Stack-Replacement. In: Proceeding of the International Symposium on Code Generation and Optimization, pp. 421–252 (2003)
33. Santos Costa, V.: Optimising bytecode emulation for prolog. In: Nadathur, G. (ed.) PPDP 1999. LNCS, vol. 1702, pp. 261–267. Springer, Heidelberg (1999)
34. Van Roy, P., Despain, A.: High Performance Logic Programming with the Aquarius Prolog Compiler. IEEE Computer Magazine 39(1), 54–68 (1992)
35. Santos Costa, V., Sagonas, K., Lopes, R.: Demand-Driven Indexing of Prolog Clauses. In: Proceedings of the International Conference of Logic Programming, pp. 40–58 (2007)

Declarative Debugging of Missing Answers in Constraint Functional-Logic Programming

Rafael Caballero, Mario Rodríguez Artalejo,
and Rafael del Vado Vírseda*

Dep. Sistemas Informáticos y Computación, Univ. Complutense de Madrid
{rafa,mario,rdelvado}@sip.ucm.es

It is well known that constraint logic and functional-logic programming languages have many advantages, and there is a growing trend to develop and incorporate effective tools to this class of declarative languages. In particular, *debugging tools* are a practical need for diagnosing the causes of erroneous computations. Recently [1], we have presented a prototype tool for the declarative diagnosis of wrong computed answers in $CFLP(\mathcal{D})$, a new generic scheme for lazy Constraint Functional-Logic Programming which can be instantiated by any constraint domain $\mathcal{D}$ given as parameter [2]. The declarative diagnosis of *missing answers* is another well-known debugging problem in constraint logic programming [4]. This poster summarizes an approach to this problem in $CFLP(\mathcal{D})$. From a programmer's viewpoint, a tool for diagnosing missing answers can be used to experiment wether the program rules for certain functions are sufficient or not for computing certain expected answers. For example, consider a $CFLP(\mathcal{H})$-program fragment written in $\mathcal{TOY}$ [3], where strict equality and disequality constraints are used for generating family relationships based on the basic family facts shown in Fig. 1.

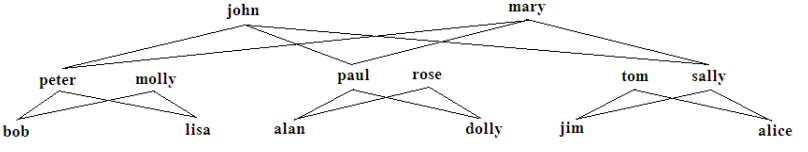


Fig. 1. Missing family relationships

```
type person = [char]

motherOf, fatherOf, sonOf, daughterOf, brotherOf, sisterOf :: person -> person
motherOf  X = Y  <== maleChildOf Z Y == X // femaleChildOf Z Y == X
brotherOf X = Y  <== maleChildOf (fatherOf X) (motherOf X) == Y, Y /= X
sonOf     X = maleChildOf X Y // maleChildOf Y X
% Analogously for the other basic family relationships
```

* The authors have been partially supported by the Spanish National Projects MERIT-FORMS (TIN2005-09207-C03-03) and PROMESAS-CAM (S-0505/TIC/0407).

V. Dahl and I. Niemelä (Eds.): ICLP 2007, LNCS 4670, pp. 425–427, 2007.

```
basicFamilyRelation :: person -> person
basicFamilyRelation = motherOf // fatherOf // sonOf // ... // brotherOf // sisterOf
familyRelation :: person -> person
familyRelation = basicFamilyRelation // basicFamilyRelation . basicFamilyRelation
(//) :: A -> A -> A          (.) :: (B -> C) -> (A -> B) -> (A -> C)
X // Y = X                   (F . G) X = F (G X)
X // Y = Y
```

In order to find a family relationship between *alice* and *alan*, a user can inspect the computed answers for the goal `familyRelation == R, R "alice" == "alan"`, involving higher-order patterns. Different diagnosis for missing answers are possible. For instance, the answer `R -> sonOf . brotherOf . motherOf` (i.e., *alan* is son of a brother of the mother of *alice*) may be missed due to an incomplete definition of `familyRelations`, which could be extended by adding the new rule `familyRelation = familyRelation . basicFamilyRelation`; while other answers such as `R -> cousinOf` or `R -> sonOf . uncleOf` may be missed due to an incomplete definition of `basicFamilyRelation`, which could be extended like this: `basicFamilyRelation = ... // cousinOf // uncleOf`.

Declarative programming paradigms such as the $CFLP(\mathcal{D})$ scheme involve complex operational details such as constraint solving, lazy evaluation of possibly higher-order and non-deterministic functions, logical variables etc. Therefore, *declarative debugging* is a better option than debugging techniques which rely on the inspection of low-level computation traces. Declarative debugging requires suitable trees to represent computations. Inspired by [1] and [4], we propose so-called *Negative Proof Trees* (shortly, NPTs) as computation trees for declarative debugging of missing answers in $CFLP(\mathcal{D})$. NPTs represent logical proofs in a *Constrained Negative Proof Calculus*. The root of a NPT has attached an *answer collection statement* of the form $G \Rightarrow \bigvee_{i \in I} S_i$, where G is an initial goal and S_i are all the observed computed answers. Internal nodes have attached answer collection statements $f\overline{t}_n \rightarrow t \;\square\; S \Rightarrow \bigvee_{i \in I} S_i$; these correspond to all the computed answers S_i for an intermediate goal $f\overline{t}_n \rightarrow t \;\square\; S$, asking for results of the function call $f\overline{t}_n$ which match t and satisfy the constraints within S. NPTs are built in such a way that the validity of the collection statement at each node follows from the collection statements at their children, under the assumption that the function definition relating the parent node to the children nodes is complete. As usual, declarative diagnosis proceeds by finding a *buggy node* which is invalid but such that all its children are valid. Every such buggy node points to an incomplete function definition. The search for a buggy node can be implemented with the help of an external *oracle* (usually the user with some semiautomatic support) who has a reliable declarative knowledge of the valid collection statements, the so-called *intended interpretation*. A prototype under development is available at `http://gpd.sip.ucm.es/rafa/missing`.

References

1. Caballero, R., Rodríguez-Artalejo, M., del Vado-Vírseda, R.: Declarative Diagnosis of Wrong Answers in Constraint Functional-Logic Programming. In: Etalle, S., Truszczyński, M. (eds.) ICLP 2006. LNCS, vol. 4079, pp. 421–422. Springer, Heidelberg (2006)

2. López-Fraguas, F.J., Rodríguez-Artalejo, M., del Vado-Vírseda, R.: A New Generic Scheme for Functional Logic Programming with Constraints. Journal of Higher-Order and Symbolic Computation 20(1/2), 73–122 (2007)
3. López-Fraguas, F.J., Sánchez-Hernández, J.: TOY: A Multiparadigm Declarative System. In: Narendran, P., Rusinowitch, M. (eds.) RTA 1999. LNCS, vol. 1631, pp. 244–247. Springer, Heidelberg (1999), `http://toy.sourceforge.net`
4. Tessier, A., Ferrand, G.: Declarative Diagnosis in the CLP Scheme. In: Deransart, P., Małuszyński, J. (eds.) Analysis and Visualization Tools for Constraint Programming. LNCS, vol. 1870, pp. 151–174. Springer, Heidelberg (2000)

Tightly Integrated Probabilistic Description Logic Programs for the Semantic Web*

Andrea Calì[1] and Thomas Lukasiewicz[2,3]

[1] Facoltà di Scienze e Tecnologie Informatiche, Libera Università di Bolzano, Italy
cali@inf.unibz.it
[2] Dipartimento di Informatica e Sistemistica, Sapienza Università di Roma, Italy
lukasiewicz@dis.uniroma1.it
[3] Institut für Informationssysteme, Technische Universität Wien, Austria
lukasiewicz@kr.tuwien.ac.at

Abstract. We present a novel approach to probabilistic description logic programs for the Semantic Web, where a tight integration of disjunctive logic programs under the answer set semantics with description logics is generalized by probabilistic uncertainty. The approach has a number of nice features. In particular, it allows for a natural probabilistic data integration and for a natural representation of ontology mappings under probabilistic uncertainty and inconsistency. It also provides a natural integration of a situation-calculus based language for reasoning about actions with both description logics and probabilistic uncertainty.

1 Overview

The *Semantic Web* aims at an extension of the current Web by standards and technologies that help machines to understand the information on the Web so that they can support richer discovery, data integration, navigation, and automation of tasks. The Semantic Web consists of several hierarchical layers, where the *Ontology layer*, in form of the *OWL Web Ontology Language*, is currently the highest layer of sufficient maturity. OWL consists of the three sublanguages *OWL Lite*, *OWL DL*, and *OWL Full*, where OWL Lite and OWL DL are essentially very expressive description logics with an RDF syntax. As a next step in the development of the Semantic Web, one aims especially at sophisticated reasoning capabilities for the *Rules*, *Logic*, and *Proof layers*.

In particular, there is a large body of work on integrating rules and ontologies, which is a key aspect of the Semantic Web. Significant research efforts focus on hybrid integrations of rules and ontologies, called *description logic programs* (or *dl-programs*), which are of the form $KB = (L, P)$, where L is a description logic knowledge base, and P is a finite set of rules involving either queries to L in a loose integration [2] or concepts and roles from L as unary resp. binary predicates in a tight integration [3].

Other works explore formalisms for *uncertainty reasoning in the Semantic Web*. In particular, the work [4] extends the loosely integrated dl-programs of [2] by probabilistic uncertainty as in Poole's independent choice logic (ICL) [5]. The ICL is a powerful representation and reasoning formalism for single- and also multi-agent systems,

* This work has been partially supported by the project P18146-N04 of the FWF, a Heisenberg Professorship of the DFG, and the STREP FET project TONES (FP6-7603) of the EU.

V. Dahl and I. Niemelä (Eds.): ICLP 2007, LNCS 4670, pp. 428–429, 2007.

which combines logic and probability, and which can represent a number of important uncertainty formalisms, in particular, influence diagrams, Bayesian networks, Markov decision processes, normal form games, and Pearl's causal models.

In this paper [1], we present *tightly integrated probabilistic disjunctive description logic programs* (or simply *probabilistic dl-programs*) *under the answer set semantics*, which are a tight integration of disjunctive logic programs under the answer set semantics, expressive description logics, and probabilistic uncertainty. To our knowledge, this is the first such approach. The main contributions of this paper are as follows:

- We present a novel approach to probabilistic dl-programs, which is based on the approach to disjunctive dl-programs under the answer set semantics of [3]. The latter is a tight integration that assumes no structural separation between the vocabularies of the description logic and the logic program components. In the same spirit as [4], this approach is developed as a combination of dl-programs with probabilistic uncertainty as in the ICL. However, rather than being based on a loose integration of rules and ontologies, it is based on a tight integration. Furthermore, rather than being based on normal dl-programs, it is based on disjunctive dl-programs.
- We present an approach to probabilistic data integration for the Semantic Web, which is based on the novel approach to probabilistic dl-programs, where probabilistic uncertainty over possible worlds may be used as trust, error, or mapping probabilities. This application takes inspiration from a number of recent probabilistic data integration approaches in the database and web community.
- Since the ICL is actually a formalism for reasoning about actions in dynamic systems, our approach to probabilistic dl-programs also provides a natural way of combining a language for reasoning about actions with both description logics and probabilistic uncertainty, especially towards Web Services.
- We show that consistency checking and query processing in probabilistic dl-programs are decidable resp. computable, and that they can be reduced to consistency checking and cautious/brave reasoning in tightly integrated disjunctive dl-programs. This directly reveals algorithms for solving the former two problems.
- We also analyze the complexity of consistency checking and query processing in probabilistic dl-programs in special cases, which turn out to be complete for the classes $\mathrm{NEXP}^{\mathrm{NP}}$ and $\mathrm{co\text{-}NEXP}^{\mathrm{NP}}$, respectively. Moreover, in the special case of stratified normal probabilistic dl-programs relative to the description logic *DL-Lite*, these two problems can be solved in polynomial time in the data complexity.

References

1. Calì, A., Lukasiewicz, T.: Tightly integrated probabilistic description logic programs. Report INFSYS RR-1843-07-05, Institut für Informationssysteme, TU Wien (March 2007)
2. Eiter, T., Lukasiewicz, T., Schindlauer, R., Tompits, H.: Combining answer set programming with description logics for the Semantic Web. In: Proc. KR-2004, pp. 141–151 (2004)
3. Lukasiewicz, T.: A novel combination of answer set programming with description logics for the Semantic Web. In: Franconi, E., Kifer, M., May, W. (eds.) ESWC 2007, LNCS, vol. 4519, pp. 384–398. Springer, Heidelberg (2007)
4. Lukasiewicz, T.: Probabilistic description logic programs. Int. J. Approx. Reason. (2007)
5. Poole, D.: The independent choice logic for modelling multiple agents under uncertainty. Artif. Intell. 94(1–2), 7–56 (1997)

View Updating Through Active Integrity Constraints

Luciano Caroprese, Irina Trubitsyna, and Ester Zumpano

DEIS, University of Calabria, 87030 Rende, Italy
{caroprese,irina,zumpano}@deis.unical.it

Motivation. Current database systems are often large and complex and the case that a user or an application has full access to the entire database is rare. It is more likely to occur that access is granted via windows of the entire systems, called *views*. A view, usually virtual, is defined by giving a query on the whole database and at any point the content of the view is just the outcome of this query. Applications query a base relation or a view in the same way. Therefore, querying a view does not represent a serious conceptual problem. In contrast, the issue of *view updating* is problematic and of paramount importance: it refers to the problem of translating an update request against a view into an update request involving the base of data. Over the years, a substantial amount of research has been devoted to the various issues surrounding view updating and not surprisingly a wide selection of approaches to the view update problem has evolved. See [2,3] for surveys of methods for view updating.

The basic problem underlying view updating is that a translation from a view update into corresponding updates over the extensional database does not always exist or several translations could be performed in order to satisfy the update request. The complexity of the view update problem arises even in the case of a simple update operation, such as inserting a tuple in a view. Current commercial DBMS, e.g. Access, MySql, Oracle, SQLServer accept an update against a view, and propagate it to the stored relation, only in the simple case in which the view is defined from one database relation, and reject any update request against a view if this is defined by joining more than one relation. This rigid behavior ensures the acceptance of an update request if and only if a unique translation exists and solves, albeit drastically, the ambiguity of more translations.

Approach. Our work focuses on view updating in the presence of *existentially derived predicates*[1] and *non-flat integrity constraints* [2] and proposes a logic framework that translates a view updating into an update of the underlying database. Specifically, given a deductive database consisting of a set of base facts, a set of integrity constraints and a set of deductive rules and given an update request, consisting of a set of insert and delete operations of base and derived facts, it allows us to determine how the update request can be translated into a minimal set of updates of the stored base facts, while ensuring integrity constraint maintenance and performing "smaller change". The benefits of this proposal, evident in the presence of existential derived predicates, will be intuitively introduced by a few examples.

[1] An existential derived predicate is defined by a deductive rule containing variables in the body that do not occur in the head of the rule. Note that this situation is likely to occur in many real cases, e.g the simple case of a database view defined as a projection of a base relation.

[2] A flat integrity constraint is defined only in terms of base predicates, i.e. its definition does not contain view.

V. Dahl and I. Niemelä (Eds.): ICLP 2007, LNCS 4670, pp. 430–431, 2007.

Example 1. Consider the update request $+P(a)$ asking for the *insertion of the fact P(a)* and the deductive database consisting of the fact $Q(a, b)$ and the view $P(X) \leftarrow Q(X, Y), R(X, Y, Z)$. The request, not allowed by commercial DBMS, could be translated, as proposed in [4], in the translations: $\{+R(a, b, val_i)\}$, for each possible value of val_i, and $\{+Q(a, val_i), +R(a, val_i, val_j)\}$ for each possible value of val_j and each possible value of $val_i \neq b$. However, this seems us to be a "bigger change" to the database that is not strictly necessary for performing the desired update. The solution, proposed in this paper, retrieves in this case, the unique translation $\{+R(a, b, \perp)\}$. The existential variable Z is fixed to the value $\perp$ (the NULL value), that suffices, in the absence of any additional information specifying Z, to perform the desired view update. Intuitively, $Q(a, b)$, thought of as a 'trustable' fact, as it belongs to the extensional database, is used to 'justify' the construction of the translation. □

Our proposal is a contribution to support view updating, consisting of insertion and deletion operations in deductive database, preventing the anomalies previous approaches suffer from. In fact, as shown before, existing approaches satisfy the update request by generating as many translations as the different values that can be assigned to the existential variables, whereas, intuitively, the approach proposed in this paper, that could be defined *cautiously liberal*, limits the wide range of translations to those that are "justified" or validated by the deductive database.

Example 2. Consider the deductive database, obtained by extending the Example 1:

$Q(a, b).$ $\quad\quad$ $P(X) \leftarrow Q(X, Y), R(X, Y, Z)$

$S(a, b, c).$ $\quad\quad$ $R(X, Y, Z) \leftarrow S(X, Y, Z), T(X, Y, Z)$

and the update request $+P(a)$. In this case our approach retrieves as unique repair $\mathcal{R} = \{+T(a, b, c)\}$. The proposed strategy implements a process that takes advantage of the initial knowledge, i.e. the set of extensional facts and the set of intensional facts derived through views of the deductive database. In this specific case, it recognizes that due to the insertion of $T(a, b, c)$ the intensional fact $R(a, b, c)$ can be derived and, consequently, the update request asking for the insertion of the fact $P(a)$ can be justified. □

The novelty of our proposal consists in the definition of a formal declarative semantics for view updating that allows the identification, among the set of all possible repairs, of the subset of *justified repairs*, i.e. the repairs whose actions are "justified" by the database or by other updates. Given a deductive database and an update request, the computation of justified repairs is performed by rewriting the update request and the deductive database in the form of active integrity constraints, whose semantics, given in terms of stable models, has been formally defined in [1].

References

1. Caroprese, L., Greco, S., Sirangelo, C., Zumpano, E.: Declarative Semantics of Production Rules for Integrity Maintenance. In: Etalle, S., Truszczyński, M. (eds.) ICLP 2006. LNCS, vol. 4079, pp. 26–40. Springer, Heidelberg (2006)
2. Fraternali, P., Paraboschi, S.: A Review of Repairing Tenchniques for Integrity Maintenance, RIDS, pp. 333–346 (1993)
3. Mayol, E., Teniente, E.: A Survey of Current Methods for Integrity Constraints Maintenance and View Updating. ER Workshop, pp. 62–73 (1999)
4. Teniente, E., Olivé, A.: Updating knowledge bases while maintaining their consistency. VLDB Journal 4(2), 193–421 (1995)

Prosper: A Framework for Extending Prolog Applications with a Web Interface

Levente Hunyadi

Budapest University of Technology and Economics
Department of Computer Science and Information Theory
1117 Budapest, Magyar Tudósok körútja 2., Hungary
Tel.: +36 1 463-2585 Fax: +36 1 463-3147
hunyadi@users.sourceforge.net

Abstract. Clear separation of presentation and code-behind, declarative use of visual control elements and a supportive background framework to automate recurring tasks are fundamental to rapid web application development. This poster presents a framework that facilitates extending Prolog applications with a web front-end. The framework relies on Prolog to the greatest possible extent, supports code re-use, and integrates easily into existing web server solutions.

Keywords: web integration; application development framework.

When developing web applications, separating application logic (what the program does) and presentation (how results are displayed) is of paramount importance. This approach leads to a logic focused closely on the task at hand and a replaceable presentation layer that wraps all web-related issues. Prosper augments regular Prolog modules that encapsulate application logic with a presentation layer that facilitates X(HTML) content generation.

Prosper [1] has a two-layered architecture (Figure 1). The lower layer, *Prolog Web Container*, maintains a direct persistent connection to the web server through the *FastCGI protocol* [2], which caters for flexibility and easy integration with an existing web server. In addition, Prolog Web Container parses HTTP request and generates HTTP response headers and content (similarly to the PiLLoW [3] library), maintains a worker thread pool and assigns jobs to threads. The primary task of the container is to isolate the communication protocol and provide a natural view of request and session data for the programmer as well as balancing incoming request load.

Prolog Server Pages, built on top of the container, defines an XML-based document model. The conventional XML document model is extended with *special elements* belonging to a dedicated namespace each of which realizes a *transformation rule*. A transformation rule can be thought of as a Prolog predicate that defines a mapping between a source and a target XML subtree. For instance, a simple iteration rule repeats its content a specified number of times. The exact behavior of the transformation is specified by means of XML attributes. Attribute values may bind to Prolog predicates or may use the so-called *expression language*. Expression language can be seen as an extension to the `is/2` predicate to include basic atom manipulation, request context variables and user-defined Prolog functions. Prolog Server Pages also offer once-assignable local variables valid inside the server page document to propagate computed values.

V. Dahl and I. Niemelä (Eds.): ICLP 2007, LNCS 4670, pp. 432–433, 2007.

Prosper includes a predefined set of special elements implementing the most common transformation rules such as conditionals and iteration constructs. However, the set of transformation rules is not restricted. Relying on the *extension infrastructure*, the user may create new modules that contain hook predicates registered for steps associated with reply generation. Modules correspond to XML namespaces and exported hook predicate names to element names in server page documents. Special elements and their implementor hook predicates are declared in a *configuration file*.

The document model allows declarative, XML-based definition of visual appearance without the use of a foreign language interface (as opposed to e.g. PrologBeans [6]). Complementing the visual part of Prolog Server Pages, *logic modules* give real power to the architecture. Logic modules (which are conventional Prolog modules) provide the code-behind that incorporates application logic and are not interleaved with the presentation layer (unlike [4] and [5]). Server pages can reference code-behind in a variety of ways: assign server page variables based on application logic, test for the satisfiability of predicates and formulate conditions using the return value of functions, thereby affecting visual layout.

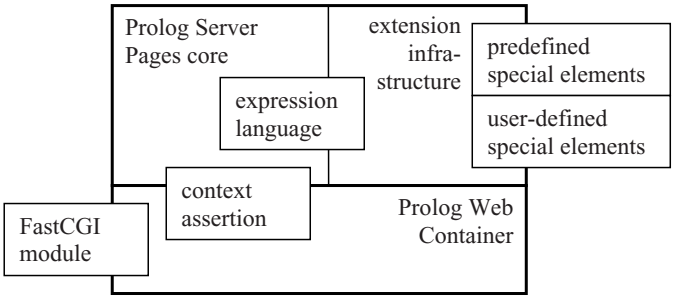


Fig. 1. The architecture of the proposed framework

References

1. Project page of Prosper, module Prosper in CVS pserver:anonymous@prospear.cvs.sourceforge.net:/cvsroot/prospear, `http://sourceforge.net/projects/prospear`
2. Brown, M.R.: FastCGI specification, Document Version: 1.0, Open Market, Inc. (April 29, 1996)
3. Cabeza, D., Hermenegildo, M.: The PiLLoW Web Programming Library, Reference Manual, The CLIP Group, School of Computer Science, Technical University of Madrid, (January 5, 2001), `http://www.clip.dia.fi.upm.es/Software/pillow/pillow.html`
4. Johnston, B.: Prolog Server Pages, `http://www.benjaminjohnston.com.au/template.prolog?t=psp`
5. Di Nuzzo, M.: Prolog Server Pages: A server-side scripting language based on Prolog, Version 0.2, (April 2006), `http://www.prologonlinereference.org/psp.psp`
6. PrologBeans and PrologBeans .NET for SICStus Prolog, `http://www.sics.se/sicstus/docs/latest/html/sicstus/PrologBeans.html`

Web Sites Verification: An Abductive Logic Programming Tool

P. Mancarella[1], G. Terreni[1], and F. Toni[2]

[1] Dipartimento di Informatica, Università di Pisa, Italy
[2] Department of Computing, Imperial College London, UK
{paolo,terreni}@di.unipi.it, ft@doc.ic.ac.uk

Abstract. We present the CIFFWEB system, an innovative tool for the verification of web sites, relying upon abductive logic programming. The system allows the user to define rules that a web site should fulfill by using (a fragment of) the query language Xcerpt. The rules are translated into abductive logic programs with constraints and their fulfillment is checked through the CIFF abductive proof procedure.

1 Web Checking Rules

The exponential growth of the WWW raises the question of maintaining and repairing automatically web pages at both structural and data level. Our *web checking rules* are characterized by using (a fragment of) the query language Xcerpt [2] for expressing complex queries in a natural syntax. The following is an example of a XML page **[P1]** about a theater company (left) and a rule expressing that *no director should occur twice in the director list* **[R1]** (right).

```
%%% directorindex.xml
<dirlist>
 <dir>dir1</dir>
 <dir>dir2</dir>
 <dir>dir2</dir>
</dirlist>
```

```
GOAL all error [var D,"double director"]
FROM
in {resource{"file:directorindex.xml"},
    dirlist {{
    dir {{var D}}, dir {{var D}} }}
} END
```

Web checking rules are specified by a *condition part* (starting with `FROM`) and an *error part* (starting with `GOAL`). The intuitive meaning of a rule is that for each instance in the XML `resource`(s) matching the condition part, an error needs to be returned. Due to lack of space, the example does not cover the whole expressiveness of our rules: in particular the absence of XML data (`without` construct) and (arithmetical) constraints over variables can also be expressed.

However, to the best of our knowledge, Xcerpt lacks of both a clear semantics for negation constructs (`without`) and a concrete tool for evaluating Xcerpt queries. Hence, we map web checking rules into *abductive logic programs with constraints* (ALPCs) that can be fed as input to the CIFF System 4.0, an implementation of the general-purpose CIFF abductive proof procedure [3] which is sound with respect to the 3-valued completion semantics. Using CIFF for determining the fulfillment of the rules, we inherit its formal properties, thus obtaining a sound concrete tool for web sites verification.

V. Dahl and I. Niemelä (Eds.): ICLP 2007, LNCS 4670, pp. 434–435, 2007.

2 Mapping Rules to Abductive Logic Programs

The CIFF proof procedure gets as input an ALPC $\langle P, A, IC\rangle_{\Re}$ where P is a normal logic program (with constraints), A is a set of *abducible predicates* and IC is a set of *integrity constraints* of the form $L_1 \wedge \cdots \wedge L_m \rightarrow H$ where each L_i is a literal and H is a disjunction of atoms. Constraint atoms are evaluated wrt an underlying structure $\Re$, as in constraint logic programming [3].

Since CIFF is not designed to handle directly XML resources, we translate XML pages into sets of atoms of the form `pg_el(ID,Tag,IDF)`. Each XML `Tag` element is associated to both a unique `ID` and to its father's id (`IDF`) in order to represent the page structure. The same is done for the data inside a XML tag. Similarly, we translate Xcerpt rules into ALPCs, where abducibles are associated with *errors*. In the absence of negation (`without` construct) each web checking rule is translated into a single integrity constraint whose head is an abducible atom `abd_err(Args,Msg)` representing the error. As an example, the translation of both **[P1]** and **[R1]** is the following.

```
[P1]: pg_el(1,dirlist,_). pg_el(2,dir,1). data_el(3,'dir1',2).
      pg_el(4,dir,1). data_el(5,'dir2',4).
      pg_el(6,dir,1). data_el(7,'dir2',6).

[R1]: [pg_el(ID1,dirlist,_), pg_el(ID2,dir,ID1), data_el(ID3,D,ID2),
       pg_el(ID4,dir,ID1), data_el(ID5,D,ID4), ID2 #\= ID4]
      implies   [abd_err([D],"double director")].
```

Running the CIFF System 4.0 with the above input, the abductive answer `[abd_err(['dir2'],'double director')]` is correctly produced[1].

3 Conclusions

The CIFFWEB tool shows how abductive logic programming can be exploited for verifying properties' fulfillment of web sites. The main advantages of this approach are the expressiveness, a clear formal semantics and a concrete computational counterpart. There are many issues to be addressed yet. In particular abductive reasoning can also be exploited for web sites repairing. A (more complex) formalization of the repairing task is work in progress. We are also working on expressiveness extensions to the framework, a user-friendly GUI, and medium-large size experiments. In the literature there is limited work on these topics. The GVERDI-R system [1] is the closest work.

References

1. Alpuente, M., Ballis, D., Falaschi, M.: A rewriting-based framework for web sites verification. Electronic Notes in Theoretical Computer Science 124, 41–61 (2005)
2. Bry, F., Schaffert, S.: The XML query language Xcerpt: Design principles, examples, and semantics (2002)
3. Endriss, U., Mancarella, P., Sadri, F., Terreni, G., Toni, F.: The CIFF proof procedure for abductive logic programming with constraints. In: Proc. JELIA (2004)

[1] The CIFF System 4.0 and the full version of this paper with all the technical details are both available at `www.di.unipi.it/~terreni/research.php`

Visual Logic Programming Method Based on Structural Analysis and Design Technique

Alexei A. Morozov

Institute of Radio Engineering and Electronics RAS
Mokhovaya 11, Moscow, Russia, 125009
morozov@cplire.ru
http://www.cplire.ru/Lab144/

The convergence of logic and visual languages is recognized as a promising approach for both the logic programming automation and enhancement of visual modeling/programming methods. The distinction of our approach is in that we unite the Actor Prolog concurrent object-oriented logic language [1,2,3] with the Structural Analysis and Design Technique (SADT) diagrams to obtain new issues in the following areas:

1. Functional modeling of complex systems;
2. Rapid prototyping of artificial intelligence applications;
3. Graphic user interface management.

The SADT diagrams (also called the IDEF∅ diagrams) are a variety of functional diagrams and are widely applied for analysis and design of complex systems. Note that the main idea of our approach is in use of the logic language for analysis/animation of the SADT diagrams, but not in use of the visual notation for automation of programming. That is why we have employed SADT, but not the UML language that has much more specialized destination.

Our method of visual programming/modeling includes the following scheme of logic program design:

1. Standard SADT diagrammers are used to develop a graphic description of the software system. SADT description is a hierarchy of blocks that receive and pass data flows (see an example of SADT diagram on Fig. 1).
2. Each elementary block of the SADT model is put into correspondence with a logic description in the form of a certain class of Actor Prolog. The source text in Actor Prolog can be written by a programmer or taken from the library of reusable modules (components).
3. The graphic description of the software system is automatically translated into the text in Actor Prolog. The syntactic means of Actor Prolog make possible to implement the block-hierarchical structure and links between blocks of the diagrams in the form of communicating processes.
4. Assembling the automatically created text and descriptions of elementary blocks, we obtain a ready-to-use program in Actor Prolog.

Experimentation with visual programming has shown that the diagrams can be conveniently used not only as a visual programming language but simply as a graphic user interface. At present, visual programming system automatically

V. Dahl and I. Niemelä (Eds.): ICLP 2007, LNCS 4670, pp. 436–437, 2007.

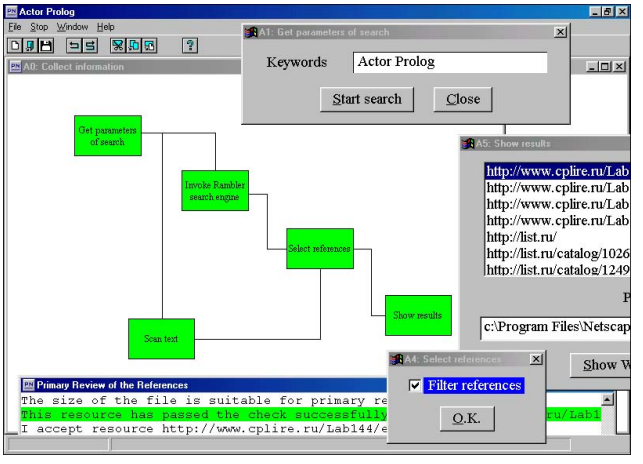


Fig. 1. An example of user interface based on SADT diagram

creates visual interface of a logic program on the basis of source SADT diagrams (see example of user interface on Fig. 1).

The individual blocks of visual user interface based on the SADT diagrams are implemented by using parallel processes of Actor Prolog. As a rule, each elementary block of a diagram has its own dialog box that can be opened by the click of a mouse. The color of the block changes automatically depending on the state of the corresponding process. A user may interact with the blocks of the diagram in any order. Repeated alteration of any parameters entered earlier is also possible.

We applied the visual programming method for logic programming of Internet agents [4]. It was shown that the tools for visual programming based on SADT and the object-oriented logic approach substantially simplify and accelerate creation of complex logic programs.

This work was supported by RFBR, project no. 06-07-89302.

References

1. Morozov, A.: Actor Prolog: an object-oriented language with the classical declarative semantics. In: Sagonas, K., Tarau, P., (eds.): Proc. of the IDL'99 Int. Workshop, Paris, France, pp. 39–53 (1999), Available at `http://www.cplire.ru/Lab144/paris.pdf`
2. Morozov, A.A., Obukhov, Y.V.: An approach to logic programming of intelligent agents for searching and recognizing information on the Internet. Pattern Recognition and Image Analysis 11(3), 570–582 (2001), Available at `http://www.cplire.ru/Lab144/pria570m.pdf`
3. Morozov, A.: Getting Started in Actor Prolog. IRE RAS, (2002), Available at `http://www.cplire.ru/Lab144/start/`
4. Morozov, A.: Development and application of logical actors mathematical apparatus for logic programming of Web agents. In: Palamidessi, C. (ed.) ICLP 2003. LNCS, vol. 2916, pp. 494–495. Springer, Heidelberg (2003)

Approximating Horn Knowledge Bases in Regular Description Logics to Have PTIME Data Complexity

Linh Anh Nguyen

Institute of Informatics, University of Warsaw
ul. Banacha 2, 02-097 Warsaw, Poland
nguyen@mimuw.edu.pl

This work is a continuation of our previous works [4,5]. We assume that the reader is familiar with description logics (DLs). A knowledge base in a description logic is a tuple $(\mathcal{R},\mathcal{T},\mathcal{A})$ consisting of an RBox $\mathcal{R}$ of assertions about roles, a TBox $\mathcal{T}$ of global assumptions about concepts, and an ABox $\mathcal{A}$ of facts about individuals (objects) and roles. The instance checking problem in a DL is to check whether a given individual a is an instance of a concept C w.r.t. a knowledge base $(\mathcal{R},\mathcal{T},\mathcal{A})$, written as $(\mathcal{R},\mathcal{T},\mathcal{A}) \models C(a)$. This problem in DLs including the basic description logic $\mathcal{ALC}$ (with $\mathcal{R}=\emptyset$) is EXPTIME-hard. From the point of view of deductive databases, $\mathcal{A}$ is assumed to be much larger than $\mathcal{R}$ and $\mathcal{T}$, and it makes sense to consider the data complexity, which is measured when the query consisting of $\mathcal{R}$, $\mathcal{T}$, C, a is fixed while $\mathcal{A}$ varies as input data. It is desirable to find and study fragments of DLs with PTIME data complexity. Several authors have recently introduced a number of Horn fragments of DLs with PTIME data complexity [2,1,3]. The most expressive fragment from those is Horn-$\mathcal{SHIQ}$ introduced by Hustadt et al. [3]. It assumes, however, that the constructor $\forall R.C$ does not occur in bodies of program clauses and goals. The data complexity of the "general Horn fragment of $\mathcal{ALC}$" is coNP-hard [6]. So, to obtain PTIME data complexity one has to adopt some restrictions for the "general Horn fragments of DLs". The goal is to find as less restrictive conditions as possible.

A RBox is a finite set of assertions of the form $R_{s_1} \circ \ldots \circ R_{s_k} \sqsubseteq R_t$, where R_{s_1}, …, R_{s_k}, R_t are role names. A regular RBox is an RBox whose set of corresponding grammar rules $t \rightarrow s_1 \ldots s_k$ forms a grammar such that the set of words derivable from any symbol s using the grammar is a regular language specified by a finite automaton. We assume that the corresponding finite automata specifying $\mathcal{R}$ are given when $\mathcal{R}$ is considered. By $\mathcal{R}eg$ we denote $\mathcal{ALC}$ extended with regular RBoxes. We extend the language of $\mathcal{ALC}$ and $\mathcal{R}eg$ with the concept constructor $\forall\exists$, which creates a concept $\forall\exists R_t.C$ from a role name R_t and a concept C. Let $\mathsf{Sem}_1(\forall\exists R_t.C) = \{\forall R_t.C, \exists R_t.\top\}$ and $\mathsf{Sem}_{2,\mathcal{R}}(\forall\exists R_t.C) = \{\forall R_t.C\} \cup \{\forall R_{s_1} \ldots \forall R_{s_{i-1}} \exists R_{s_i}.\top \mid R_{s_1} \circ \cdots \circ R_{s_k} \sqsubseteq R_t$ is a consequence of $\mathcal{R}$ and $1 \leq i \leq k\}$. Then, for a model $I = \langle \Delta^I, \cdot^I \rangle$ of an RBox $\mathcal{R}$, define $x \in \Delta^I$ to be an instance of a concept C in I w.r.t. $\mathfrak{s} \in \{\mathsf{Sem}_1, \mathsf{Sem}_{2,\mathcal{R}}\}$, write $I, x \models_{\mathfrak{s}} C$, in the usual way if C is not of the form $\forall\exists R_t.D$, and that $I, x \models_{\mathfrak{s}} \forall\exists R_t.D$ if $I, x \models_{\mathfrak{s}} D'$ for every $D' \in \mathfrak{s}(\forall\exists R_t.D)$. We write $I \models_{\mathfrak{s}} C(a)$ for $I, a^I \models_{\mathfrak{s}} C$.

A *positive concept* is a concept (in the extended language) without the constructors $\bot$, $\neg$, $\sqsubseteq$, $\doteq$. A *deterministic positive concept* is a positive concept which does not contain the constructor $\forall$ (but may contain $\exists$ and $\forall\exists$).

V. Dahl and I. Niemelä (Eds.): ICLP 2007, LNCS 4670, pp. 438–439, 2007.

A *positive logic program* is a finite set of *program clauses* formed using the following BNF grammar, in which C denotes a positive concept without $\forall\exists$:

$$D ::= \top \mid A \mid C \sqsubseteq D \mid D \sqcap D \mid \forall R_t.D \mid \exists R_t.D$$

A *deterministic positive logic program* is a finite set of *deterministic program clauses* formed using the above BNF grammar with C being a deterministic positive concept.

Given a positive logic program $\mathcal{T}'$, the deterministic version of $\mathcal{T}'$ is the deterministic positive logic program $\mathcal{T}$ obtained from $\mathcal{T}'$ by replacing every concept of the form $\forall R.C$ in bodies of program clauses of $\mathcal{T}'$ by $\forall\exists R.C$.

In the full version [6] of this paper, we present an algorithm that, given $\mathfrak{s} \in \{\mathsf{Sem}_1, \mathsf{Sem}_{2,\mathcal{R}}\}$ and a knowledge base $(\mathcal{R}, \mathcal{T}, \mathcal{A})$ consisting of a regular RBox $\mathcal{R}$, a deterministic positive logic program $\mathcal{T}$, and an ABox $\mathcal{A}$, constructs a finite "least $\mathfrak{s}$-pseudo-model" I of $(\mathcal{R}, \mathcal{T}, \mathcal{A})$. For $\mathfrak{s} = \mathsf{Sem}_{2,\mathcal{R}}$, the least $\mathfrak{s}$-pseudo-model I has the property that, $I \models_{\mathsf{Sem}_{2,\mathcal{R}}} C(a)$ iff $(\mathcal{R}, \mathcal{T}, \mathcal{A}) \models_{\mathsf{Sem}_{2,\mathcal{R}}} C(a)$, for every deterministic positive concept C. For $\mathfrak{s} = \mathsf{Sem}_1$, we have only the one-way assertion: $I \models_{\mathsf{Sem}_1} C(a)$ implies $(\mathcal{R}, \mathcal{T}, \mathcal{A}) \models_{\mathsf{Sem}_1} C(a)$, for every positive concept C.

Given a Horn knowledge base $(\mathcal{R}, \mathcal{T}', \mathcal{A})$ and a positive concept C, where $\mathcal{R}$ is a regular RBox and $\mathcal{T}$ is a positive logic program, we propose the approximation of checking $(\mathcal{R}, \mathcal{T}', \mathcal{A}) \models C(a)$ by checking whether $I \models_{\mathsf{Sem}_1} C(a)$, where $\mathcal{T}$ is the deterministic version of $\mathcal{T}'$ and I is the least Sem_1-pseudo-model of $(\mathcal{R}, \mathcal{T}, \mathcal{A})$ constructed by the algorithm given in [6]. The approximation is correct in the sense that $I \models_{\mathsf{Sem}_1} C(a)$ implies $(\mathcal{R}, \mathcal{T}', \mathcal{A}) \models C(a)$. It is a good approximation because that:

- First, Sem_1, which interprets $\forall\exists R.D$ in premises as $\forall R.D \sqcap \exists R.\top$, is highly "acceptable". For example, $\forall\exists child.good\_person \sqsubseteq happy\_parent$ w.r.t. Sem_1 is more "acceptable" than the clause $\forall child.good\_person \sqsubseteq happy\_parent$.
- Second, $(\mathcal{R}, \mathcal{T}, \mathcal{A}) \models_{\mathsf{Sem}_{2,\mathcal{R}}} C(a)$ implies $I_1 \models_{\mathsf{Sem}_1} C(a)$ if C is a deterministic positive concept. At least, when the constructors $\forall R.D$ and $\forall\exists R.D$ are disallowed in C and bodies of program clauses of $\mathcal{T}'$ (as assumed for Horn-$\mathcal{SHIQ}$ [3]), the approximation is exact, i.e. $(\mathcal{R}, \mathcal{T}', \mathcal{A}) \models C(a)$ iff $I \models_{\mathsf{Sem}_1} C(a)$.
- Third, I is constructed in polynomial time in the size of $\mathcal{A}$, and checking $I \models_{\mathsf{Sem}_1} C(a)$ can be done in polynomial time in the size of I and C. That is, the approximation has PTIME data complexity.

References

1. Calvanese, D., De Giacomo, G., Lembo, D., Lenzerini, M., Rosati, R.: Data complexity of query answering in description logics. In: Proc. of KR'2006, pp. 260–270. AAAI Press, Stanford (2006)
2. Grosof, B.N., Horrocks, I., Volz, R., Decker, S.: Description logic programs: combining logic programs with description logic. In: Proceedings of WWW'2003, pp. 48–57 (2003)
3. Hustadt, U., Motik, B., Sattler, U.: Data complexity of reasoning in very expressive description logics. In: Proceedings of IJCAI-05, pp. 466–471. Professional Book Center (2005)
4. Nguyen, L.A: A bottom-up method for the deterministic Horn fragment of the description logic $\mathcal{ALC}$. In: Fisher, M., van der Hoek, W., Konev, B., Lisitsa, A. (eds.) JELIA 2006. LNCS (LNAI), vol. 4160, pp. 346–358. Springer, Heidelberg (2006)
5. Nguyen, L.A.: On the deterministic Horn fragment of test-free PDL. In: Advances in Modal Logic - vol. 6, pp. 373-392. King's College Publications (2006)
6. Nguyen, L.A.: The full version of this paper, `http://www.mimuw.edu.pl/~nguyen/RG.pdf`

A Linear Transformation from Prioritized Circumscription to Disjunctive Logic Programming*

Emilia Oikarinen** and Tomi Janhunen

Helsinki University of Technology, P.O. Box 5400, FI-02015 TKK, Finland

Introduction

The stable model semantics of *disjunctive logic programs* (DLPs) is based on *minimal models* which assign atoms false by default. While this feature is highly useful—leading to concise problem encodings—it occasionally renders knowledge representation with disjunctive rules difficult. Reiter-style *minimal diagnoses* [1] provide a good example in this respect. This problem can be alleviated by a more refined control of minimization provided by *parallel circumscription* [2] which allows certain atoms to *vary* or to have *fixed* truth values. The scheme of *prioritized circumscription* [3,2] generalizes this setting with priority classes for atoms being minimized. Our aim is to bring these enhancements of minimality to the realm of disjunctive logic programming. We strive for a translation-based approach where varying and fixed atoms, as well as priority classes are effectively removed from representations by transformations. We have already addressed parallel circumscription and provided a *linear* and *faithful* but *non-modular* translation [4]. Here we present a similar transformation for prioritized circumscription, extend our implementation [5], and report preliminary experiments.

Circumscription. In the sequel, we consider the two forms of circumscription in the propositional case. Given a theory Π represented as a *positive* DLP, the purpose of a prioritized circumscription $\mathrm{Circ}(\Pi, P_1 > \ldots > P_k, V, F)$ [1] of Π is to falsify atoms in each set P_i, with a decreasing level of priority $0 < i \leq k$, as far as possible. Meanwhile the truth values of atoms in V may *vary* freely and the truth values of atoms in F are kept *fixed*. These objectives can be captured using a notion of minimality as follows.

Definition 1. *A model $M \models \Pi$ of a positive DLP Π is $\langle P_1 > \ldots > P_k, V, F\rangle$-minimal iff there is no $N \models \Pi$ such that (i) $N \cap P_1 \subset M \cap P_1$, or $N \cap (P_1 \cup \ldots \cup P_{i-1}) = M \cap (P_1 \cup \ldots \cup P_{i-1})$ and $N \cap P_i \subset M \cap P_i$ for some $1 < i \leq k$; and (ii) $N \cap F = M \cap F$.*

The parallel circumscription $\mathrm{Circ}(\Pi, P, V, F)$ of Π is obtained as a special case of Definition 1 ($k = 1$). In addition, the $\langle P_1 > \ldots > P_k, V, F\rangle$-minimality of $M \models \Pi$ is captured by the unsatisfiability of a translation $\mathrm{Tr}_{\mathrm{UNSAT}}(\Pi, P_1 > \ldots > P_k, F, M) =$

$$\{(A \setminus F) \leftarrow (B \setminus F) \mid A \leftarrow B \in \Pi,\, M \not\models \bigvee(A \cap F), \text{ and } M \models B \cap F\} \cup \{e_0 \leftarrow\} \cup \bigcup_{i=1}^{k} \{e_i \leftarrow (P_i \cap M) \cup \{e_{i-1}\}\} \cup \{\bot \leftarrow e_k\} \cup \bigcup_{i=1}^{k} \{\bot \leftarrow a, e_{i-1} \mid a \in P_i \setminus M\}. \quad (1)$$

* This research has been partially funded by the Academy of Finland under project "*Advanced Constraint Programming Techniques for Large Structured Problems*" (#211025).

** Support from Helsinki Graduate School in Computer Science and Engineering, Nokia Foundation, Finnish Foundation of Technology, and Finnish Cultural Foundation is acknowledged.

[1] The sets of atoms $P_1, \ldots, P_k$, V, and F are mutually disjoint and cover all atoms of Π.

[2] Here e_0 and $e_1, \ldots, e_k$, which correspond to priority classes $P_1, \ldots, P_k$, are new atoms.

V. Dahl and I. Niemelä (Eds.): ICLP 2007, LNCS 4670, pp. 440–441, 2007.

Translation-Based Approach. In [4], we present a transformation that captures the models of a parallel circumscription $\mathrm{Circ}(\Pi, P, V, F)$ with the stable models of its translation. Prioritized circumscription is handled by translating it to parallel circumscription using Lifschitz' (quadratic) scheme [5]. Here we extend the method from [4] for prioritized circumscription. The translation $\mathrm{Tr}_{\mathrm{circ2dlp}}(\Pi, P_1 > \ldots > P_k, V, F)$ consists of two DLP *modules* (cf. DLP-*functions* in [6]). The module $\mathrm{Tr}_{\mathrm{gen}}(\Pi)$ generates a model candidate for $\mathrm{Circ}(\Pi, P_1 > \ldots > P_k, V, F)$ roughly in the same way as in [4]. The module $\mathrm{Tr}_{\mathrm{min}}(\Pi)$ encodes the test for $\langle P_1 > \cdots > P_k, V, F\rangle$-minimality as an unsatisfiability check based on (1). When the two modules are joined as a single DLP, model candidates created by $\mathrm{Tr}_{\mathrm{gen}}(\Pi)$ are passed as input to $\mathrm{Tr}_{\mathrm{min}}(\Pi)$ for testing $\langle P_1 > \ldots > P_k, V, F\rangle$-minimality. As a consequence, the $\langle P_1 > \cdots > P_k, V, F\rangle$-minimal models M of a positive DLP Π and the stable models N of $\mathrm{Tr}_{\mathrm{circ2dlp}}(\Pi, P_1 > \ldots > P_k, V, F)$ end up in a bijective correspondence such that $M = N \cap \mathrm{At}(\Pi)$.

Experiments. Our translator CIRC2DLP (v2.1)[3] implements $\mathrm{Tr}_{\mathrm{circ2dlp}}(\cdot)$. We use the problem of finding Reiter-style minimal diagnoses for digital circuits as the benchmark. For $k > 1$ priority classes for minimization, the performance of CIRC2DLP significantly improves the quadratic translation from [5]. On smaller instances the running times of CIRC2DLP (using DLV as back-end) and CIRCUM2 are very similar, but the memory consumption of CIRCUM2 becomes soon a bottleneck (over 512MB) as instances grow.

Discussion. The translation $\mathrm{Tr}_{\mathrm{circ2dlp}}(\cdot)$ improves its predecessors [4,5] and it has a distinctive set of properties: (i) arbitrary propositional theories Π subject to prioritized circumscription are covered, (ii) the translation $\mathrm{Tr}_{\mathrm{circ2dlp}}(\Pi, P_1 > \ldots > P_k, V, F)$ can be produced in linear time and space before computing any models, (iii) the minimal models of $\mathrm{Circ}(\Pi, P_1 > \ldots > P_k, V, F)$ and the stable models of its translation are in a bijective relationship, (iv) the signature $\mathrm{At}(\Pi)$ is preserved, and (v) there is no need for incremental updating. All previous transformations lack some of the features (i)–(v). In particular, those involving *characteristic clauses* and *loop formulas*, and the one underlying CIRCUM2, are worst-case exponential. Our first experiments indicate that CIRC2DLP combined with a disjunctive solver compares favorably with CIRCUM2.

References

1. Reiter, R.: A theory of diagnosis from first principles. Artif. Intell. 32(1), 57–95 (1987)
2. Lifschitz, V.: Computing circumscription. In: IJCAI'85, Los Angeles, CA, USA, pp. 121–127. Morgan Kaufmann, San Francisco (1985)
3. McCarthy, J.: Applications of circumscription to formalizing commonsense knowledge. Artif. Intell. 28, 89–116 (1986)
4. Janhunen, T., Oikarinen, E.: Capturing parallel circumscription with disjunctive logic programs. In: Alferes, J.J., Leite, J.A. (eds.) JELIA 2004. LNCS (LNAI), vol. 3229, pp. 134–146. Springer, Heidelberg (2004)
5. Oikarinen, E., Janhunen, T.: CIRC2DLP–Translating circumscription into disjunctive logic programming. In: Baral, C., Greco, G., Leone, N., Terracina, G. (eds.) LPNMR 2005. LNCS (LNAI), vol. 3662, pp. 405–409. Springer, Heidelberg (2005)
6. Janhunen, T., Oikarinen, E., Tompits, H., Woltran, S.: Modularity aspects of disjunctive stable models. In: Baral, C., Brewka, G., Schlipf, J. (eds.) LPNMR 2007. LNCS (LNAI), vol. 4483, pp. 175–187. Springer, Heidelberg (2007)

[3] See `http://www.tcs.hut.fi/Software/circ2dlp/` for binaries and benchmarks.

Representation and Execution of a Graph Grammar in Prolog

Girish Keshav Palshikar
Tata Research Development and Design Centre (TRDDC)
gk.palshikar@tcs.com

Structured diagrams - e.g., ER diagrams - consist of finite types of symbols interconnected according to well-defined rules. Graph grammars [CEEe95, Hab92] are a well-known formalism used to specify the syntax of structured diagrams. Many graph grammar formalisms have been defined, which differ mainly in their graph rewrite mechanism. Graph grammars are well understood in algebraic framework; but there is little work in linking them to logic programming [CMR+91, CMR+91, CR93, Sch93].

We propose a graph grammar formalism called *Graph Grammar in Prolog (GGP)*. We show how GGP can be embedded in Prolog, akin to how DCG embeds context-free string grammars in Prolog. Like DCG, we use Prolog to provide an executable semantics to the GGP formalism. We illustrate the formalism by defining syntax of a simple class of graphs.

An *edge terminal* is a Prolog term of the form `edge(vertex(U,A), vertex(V,B))` which states that there is a directed edge between two vertices having IDs `U`, `V` and labels `A`, `B` respectively. The vertex IDs or labels can be Prolog variables or constants. A *non-terminal* (also called an N*-term*) is a Prolog term of the form `name(L)`, where `name` is an atom and `L` is a Prolog list, typically consisting of only Prolog variables. A GGP production has the form `Head ==> Body.` where `Head` is an N-term and `Body` is a sequence of terminals and N-terms (and also the usual Prolog predicates).

The following GGP productions define the syntax for star graphs, whose center is labeled with `a` and all other vertices are labeled with `b`. Only one non-terminal `star` is used. The second GGP production is recursive: N-term `star` occurs as head as well as in the body. The N-terms and terminals can freely share Prolog variables; e.g., the head `star` and the N-term `star` in the body in second production share the Prolog variables `S0`.

```
star([S0]) ==> edge(vertex(S0,a),vertex(S1,b)).
star([S0]) ==> edge(vertex(S0,a),vertex(S1,b)), star([S0]).
```

A declarative reading of these GGP productions is as follows. Prolog variable `S0` identifies the ID of the centre vertex of the star graph. The first production says that a graph consisting of two vertices `S0, S1`, labeled with `a` and `b` respectively and having a single directed edge from `S0` to `S1` is a star graph. The second production says that a graph is a star graph with centre vertex `S0` labeled with `a`, if it includes vertices `S0, S1` labeled with `a` and `b` and a directed edge from `S0` to `S1` and the graph obtained by deleting this `(S0, S1)` edge (without deleting the vertices `S0` and `S1`) is a star graph.

V. Dahl and I. Niemelä (Eds.): ICLP 2007, LNCS 4670, pp. 442–443, 2007.

A GGP parses a given graph (represented as unordered sets of vertices and edges) and reports whether or not the sets obeys the syntax specified by the GGP productions. Each step in the GGP derivation removes one edge from the given set of edges. The derivation terminates when all edges are removed. Prolog variables in N-terms get instantiated during the derivation. As in DCG, each GGP production is translated into a corresponding Prolog rule. Above GGP productions are translated into the following Prolog code.

```
star(A,B,[C],D,E) :- check_edge(A,B,vertex(C,a),vertex(F,b),D,E).
star(A,B,[C],D,E) :- check_edge(A,B,vertex(C,a),vertex(F,b),G,H),
                     star(G,H,[C],D,E).
```

We use the Prolog interpreter to check whether the above (translated) GGP parses a directed 5-vertex star graph where the central vertex having ID 1 is labeled with `a` and other vertices (having IDs 2, 3, 4, 5) are labeled with `b`. Actual vertex IDs and the order of vertices and edges in the lists are immaterial.

```
?- star([v(1,a),v(2,b),v(3,b),v(4,b),v(5,b)],
        [e(1,2),e(1,3),e(1,4),e(1,5)], X,_,[]).
X = [1]
```

We have also developed a formalism called *simple graph grammar (SGG)*, which can be viewed as both a string grammar (with a string rewrite mechanism) and a graph grammar (with a graph rewriting mechanism). GGP is an embedding of SGG in Prolog. We have used GGP to define syntax of various structured diagrams: ER diagram, message sequence chart and statecharts. We are working on using GGP as a graph transformation and graph database query mechanism.

Acknowledgments. Sincere thanks to Dr. Manasee Palshikar and colleagues in TRDDC.

References

[CEEe95] Cuny, J., Engels, G., Ehrig, H., Rozenberg, G. (eds.): 5th Int. Work. Graph Grammars and Their Application to Computer Science. LNCS, vol. 1073. Springer, Heidelberg (1996)

[CMR+91] Corradini, A., Montanari, U., Rossi, F., Ehrig, H., Lowe, M.: Graph grammars and logic programming. In: Ehrig, H., Kreowski, H.-J., Rozenberg, G. (eds.) Proc. 4th Int. Work. Graph Grammars and Their Application to Computer Science. LNCS, vol. 532, pp. 221–237. Springer, Heidelberg (1991)

[CR93] Corradini, A., Rossi, F.: On the expressive power of hyperedge-replacement jungle rewriting for term rewriting systems and logic programming. Theoretical Computer Science, 109 (1993)

[Hab92] Habel, A.: Hyperedge Replacements: Grammars and Languages. Springer, Heidelberg (1992)

[Sch93] Schurr, A.: Logic-based structure rewriting systems. In: Dagstuhl Seminar on Graph Transformations in Computer Science, pp. 341–357 (1993)

On Applying Program Transformation to Implement Suspension-Based Tabling in Prolog (Extended Abstract)

Ricardo Rocha, Cláudio Silva, and Ricardo Lopes*

DCC-FC & LIACC
University of Porto, Portugal
ricroc@ncc.up.pt, ccaldas@dcc.online.pt, rslopes@ncc.up.pt

Tabling is a technique of resolution that overcomes some limitations of traditional Prolog systems in dealing with redundant sub-computations and recursion. We can distinguish two main categories of tabling mechanisms: *suspension-based tabling mechanisms* and *linear tabling mechanisms*. Suspension-based tabling mechanisms need to preserve the state of suspended tabled subgoals in order to ensure that all answers are correctly computed. A tabled evaluation can be seen as a sequence of sub-computations that suspend and later resume. On the other hand, linear tabling mechanisms use iterative computations of tabled subgoals to compute fix-points. The main idea of linear tabling is to maintain a single execution tree where tabled subgoals always extend the current computation without requiring suspension and resumption of sub-computations.

A common approach used to include tabling support into existing Prolog systems is to modify and extend the low-level engine. Although this approach is ideal for run-time efficiency, it is not easily portable to other Prolog systems as engine level modifications are rather complex and time consuming and require changing important components of the system such as the compiler, the code generator, and the data structures that support Prolog execution. A different approach to incorporate tabled evaluation into existing Prolog systems is to apply source level transformations to a tabled program. The transformed program then uses external tabling primitives that provide direct control over the search strategy to implement tabled evaluation. This idea was first explored by Fan and Dietrich [1] that implemented a form of linear tabling using source level program transformation and tabling primitives implemented as Prolog built-ins.

In this work, we present a suspension-based tabling mechanism based on program transformation, but we use the C language interface, available in most Prolog systems, to implement the tabling primitives. In particular, we use the C interface of the Yap Prolog system to build external Prolog modules implementing the support for tabled evaluation. We can distinguish two main modules in our implementation: the module that implements the specific control primitives and the module that implements the table space data structures. The table space was implemented using *tries* [2]. To implement our mechanism, that we named *tabled evaluation with continuation calls*, we followed a *local scheduling*

* This work has been partially supported by Myddas (POSC/EIA/59154/2004) and by funds granted to LIACC through the Programa de Financiamento Plurianual, Fundação para a Ciência e Tecnologia and Programa POSC.

V. Dahl and I. Niemelä (Eds.): ICLP 2007, LNCS 4670, pp. 444–445, 2007.

strategy [3]. Suspension is implemented by leaving a *continuation call* for the current computation in the table entry corresponding to the variant call being suspended. During this process and as further new answers are found, they are stored in their tables and returned to all variant calls by calling the previously stored continuation calls. To implement the program transformation step, we have extended the original program transformation module of Ramesh and Chen [4] to include the tabling primitives for our approach.

To evaluate the impact of our approach, we ran it against the state-of-the-art YapTab system, that implements tabling support at the low-level engine. YapTab also implements a suspension-based mechanism, uses tries to implement the table space and is implemented on top of Yap. This was thus a first and fair comparison between the approach of supporting tabling at the low-level engine and the approach of supporting tabling by applying source level transformations coupled with tabling primitives. As expected, YapTab outperformed our mechanism in all programs tested. On average, YapTab was about 7.50 faster than the continuation calls mechanism. Best performance was achieved for left recursive tabled predicates with the recursive clause first, with an average overhead between 2 and 3. The results obtained also suggested that there is a cost in the execution time that is proportional to the number of redundant answers, variant calls and continuation calls executed during an evaluation. In particular, the number of continuation calls seems to be the most relevant factor that contributes to this cost because continuation calls are not compiled, they are constructed and called in run-time using the C language interface.

Considering that Yap and YapTab are respectively two of the fastest Prolog and tabling engines currently available, the results obtained are very interesting and very promising. We thus argue that our approach is a good alternative to incorporate tabling into any Prolog system. It requires neither advanced knowledge of the implementation details of tabling nor time consuming or complex modifications to the low-level engine. Moreover, both source level transformations and tabling primitives can be easily ported to other Prolog systems with a C language interface. Currently, we are already working with the Ciao group to include our implementation as a module of the Ciao Prolog system.

References

1. Fan, C., Dietrich, S.: Extension Table Built-Ins for Prolog. Software Practice and Experience 22, 573–597 (1992)
2. Ramakrishnan, I.V., Rao, P., Sagonas, K., Swift, T., Warren, D.S.: Efficient Access Mechanisms for Tabled Logic Programs. Journal of Logic Programming 38, 31–54 (1999)
3. Freire, J., Swift, T., Warren, D.S.: Beyond Depth-First: Improving Tabled Logic Programs through Alternative Scheduling Strategies. In: Kuchen, H., Swierstra, S.D. (eds.) PLILP 1996. LNCS, vol. 1140, pp. 243–258. Springer, Heidelberg (1996)
4. Ramesh, R., Chen, W.: Implementation of Tabled Evaluation with Delaying in Prolog. IEEE Transactions on Knowledge and Data Engineering 9, 559–574 (1997)

Aggregates in Constraint Handling Rules
(Extended Abstract)

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

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

Introduction

Constraint Handling Rules (CHR) [2,3,4] is a general-purpose programming language based on committed-choice, multi-headed, guarded multiset rewrite rules. As the head of each CHR rule only considers a fixed number of constraints, any form of aggregation over unbounded parts of the constraint store necessarily requires explicit encoding, using auxiliary constraints and rules.

Example 1. Suppose the constraints `account(AccountId,ClientId,Balance)` and `client(ClientId)` constitute a simplified representation of the accounts and clients of a bank. Consider the business rule *"A client whose accumulated sum of account balances is $25,000 or more is a platinum client"*. A common approach to encode this rule in CHR is the introduction of an auxiliary constraint:

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

However, the explicit maintenance of such an auxiliary constraint is an inherently cross-cutting concern which requires invasive modifications to many rules, spread throughout the entire program, e.g.:

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

□

Aggregates. We propose an extension of CHR with aggregate expressions in the heads of rules. Aggregates accumulate information over possibly unbounded parts of the constraint store. We provide a wide range of predefined aggregates, including all aggregates commonly found in related paradigms such as database query languages [1] (i.e. `min`, `max`, `sum`, `count` and `avg`) and production rule systems (i.e. `not`, `exists` and `forall`). The complete list of predefined aggregates, together with a number of example uses, can be found in [5].

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

V. Dahl and I. Niemelä (Eds.): ICLP 2007, LNCS 4670, pp. 446–448, 2007.

Example 2. The auxiliary constraint in Example 1 can be avoided using `sum`:

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

No further program changes are required to implement the business rule. □

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

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

The `aggregate/7` construct is expressive enough to specify any aggregate. All predefined aggregates are implemented by it. For instance, `sum(T,G,R)` is defined as `aggregate(=(0),plus,minus,=,T,G,R)`, where '`=(0)`' indicates unification with zero, and `plus/3` and `minus/3` are two straightforward Prolog predicates computing the sum, respectively the difference of the first two arguments.

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

Complex aggregate goals. We allow arbitrary conjunctions of CHR constraints and guards in the aggregate goal `Goal`: for example, the expression `count((platinum(C),account(_,C,_)), N)` counts the number of accounts owned by platinum clients. We also allow *nested aggregates*, i.e. aggregates inside the goal of another aggregate. For example, the client `C` with the largest total balance `S` is given by `argmax(S, (client(C), sum(B,account(_,C,B),S)))`.

Implementation. We have developed a prototype implementation [6] based on a source-to-source transformation to regular CHR (extended with some low-level compiler pragma's). The implementation allows aggregate computation using either an on-demand or an incremental strategy. Case studies indicate that aggregates can significantly reduce the program size and improve the readability, while the run-time overhead is an acceptable constant factor.

References

1. ISO/IEC 9075: Information technology – Database languages – SQL (2003)
2. Duck, G.J., Stuckey, P.J., de la Banda, M.G., Holzbaur, C.: The refined operational semantics of Constraint Handling Rules. In: Demoen, B., Lifschitz, V. (eds.) ICLP 2004. LNCS, vol. 3132, pp. 90–104. Springer, Heidelberg (2004)
3. Frühwirth, T.: Theory and practice of Constraint Handling Rules. Journal of Logic 37(1–3), 95–138 (1998)

4. Schrijvers, T.: Analyses, Optimizations and Extensions of Constraint Handling Rules. PhD thesis, Leuven, K.U. Leuven, Belgium (June 2005)
5. Sneyers, J., Van Weert, P., Schrijvers, T.: Aggregates in CHR. Submitted to 4th Workshop on Constraint Handling Rules (2007)
6. Van Weert, P., Sneyers, J., Demoen, B.: Aggregates for CHR through program transformation. Submitted to 17th Intl. Symposium on Logic-Based Program Synthesis and Transformation (2007)

Computing Fuzzy Answer Sets Using DLVHEX

Davy Van Nieuwenborgh[1], Martine De Cock[2], and Dirk Vermeir[1]

[1] Vrije Universiteit Brussel, VUB
{dvnieuwe,dvermeir}@vub.ac.be
[2] Universiteit Gent, UGent
martine.decock@ugent.be

Abstract. In this poster, we show how the fuzzy answer set semantics, i.e. a combination of answer set programming and fuzzy logic, can be mapped onto the semantics for HEX-programs.

We refer the reader to [4] for an in-depth introduction to the framework of fuzzy answer set programming, while [2] is a good starting point for the concept of HEX-programs.

The mapping from the FASP semantics to HEX-programs we present is based on the well-known guess-and-check methodology in answer set programming [3]. Intuitively, the guess part of our encoding will be responsible for computing x-consistent fuzzy y-answer sets, while the checking part of the encoding will check the foundedness condition of fuzzy models.

To handle the fuzzy operations on the lattice under consideration, e.g. negation or implication, we will use external atoms. Our translation uses, among others, the following external functions: $\&not\_gt[X,Y](), \&not\_geq[X,Y](), \&negator[V](NV), \&t\_norm[X_1,\ldots,X_k](X), \&implicator[X,Y](Z)$ and $\&support[X,Z](Y)$. Note that we use $\&not\_gt$ and $\&not\_geq$ to handle $\not>_{\mathcal{L}}$ and $\not\geq_{\mathcal{L}}$, where $\mathcal{L}$ is the lattice under consideration, as the properties $x \not> y \Leftrightarrow x \leq y$ and $x \not\geq y \Leftrightarrow x < y$ do not hold for non-total lattices.

The first program in our translation will compute, for a program P, the x-consistent fuzzy y-models, with $x, y \in \mathcal{L}$. This program $P^{x,y}_{guess}$, with $x, y \in \mathcal{L}$, contains the following rules.

$$
\begin{aligned}
\{fi(l,v_1) \vee \cdots \vee fi(l,v_n) &\leftarrow \mid l \in Lit_P\} \\
fi(\bot, 0_{\mathcal{L}}) &\leftarrow \\
nfi(L,NV) &\leftarrow fi(L,V), \&negator[V](NV) \\
consistent(X) &\leftarrow \&consistency[fi](X) \\
&\leftarrow consistent(X), \&not\_geq[X,x]() \\
body(r,X,a) &\leftarrow fi(b_1,X_1),\ldots,fi(b_k,X_k), nfi(b_{k+1},X_{k+1}),\ldots,nfi(b_m,X_m), \\
&\quad \&t\_norm[X_1,\ldots,X_m](X) \\
body(r,X,a) &\leftarrow fi(b_1,X_1),\ldots,fi(b_k,X_k), nfi(b_{k+1},X_{k+1}),\ldots,nfi(b_m,X_m), \\
&\quad \&t\_norm[X_1,\ldots,X_m](X) \\
rule(R,Z) &\leftarrow body(R,X,H), fi(H,Y), \&implicator[X,Y](Z) \\
aggregation(X) &\leftarrow \&aggregate[rule](X) \\
&\leftarrow aggregation(X), \&not\_geq[X,y]()
\end{aligned}
$$

V. Dahl and I. Niemelä (Eds.): ICLP 2007, LNCS 4670, pp. 449–450, 2007.

To check, for a program P, whether an x-consistent fuzzy y-model M of P computed by $P^{x,y}_{guess}$ is also an x-consistent fuzzy y-answer set of P, we need additional rules, denoted by P_{check}, that check M for unfounded sets. To accomplish this, we will use the saturation technique that is used in some complexity proofs of the answer set semantics for disjunctive programs [1] and in the work of automated integration of guess and check programs in answer set programming [3].

Intuitively, we will compute a set of literals X. If X is such that it has no literal in common with the fuzzy interpretation contained in fi, it cannot be used to show that fi is not unfounded-free. Therefore, we consider X always as founded in this case and we will saturate the answer set accordingly. On the other hand, when X has at least one literal in common with the fuzzy interpretation contained in fi, then, if X is founded, we will saturate the answer set. Thus, because of the minimality of answer sets, if a set X exists that is unfounded w.r.t. fi, no saturation will occur and we will have a minimal answer set for the HEX-program. As a result, saturated answer sets of the HEX-program $P^{x,y}_{guess} \cup P_{check}$ will correspond to x-consistent fuzzy y-answer sets of P. This program P_{check} contains the following rules.

$$
\begin{aligned}
sup(R, Y) &\leftarrow body(R, X, H), rule(R, Z), \&support[X, Z](Y)\\
\{x(l) \vee x(l') &\leftarrow \mid l \in Lit_P\}\\
founded &\leftarrow x(L), body(R, B, L), \&no\_body\_intersection[R, x](),\\
&\qquad fi(L, V), sup(R, S), \&not\_gt[V, S](), B \neq 0_{\mathcal{L}}\\
&\qquad founded \leftarrow \&no\_intersection[fi, x]()\\
founded \vee unfounded &\leftarrow\\
unfounded &\leftarrow founded\\
&\qquad \{x(l) \leftarrow founded \ ; \ x(l') \leftarrow founded \mid l \in Lit_P\}
\end{aligned}
$$

The following theorem confirms that the combination of $P^{x,y}_{guess}$ and P_{check} can be used to retrieve the x-consistent fuzzy y-answer sets of P.

Theorem 1. *Let P be a program, let I be a fuzzy interpretation and take $x, y \in \mathcal{L}$. I is an x-consistent fuzzy y-answer set of P iff there exists an answer set I' of $P^{x,y}_{guess} \cup P_{check}$ such that $\{fi(l, I(l)) \mid l \in Lit_P\} \subseteq I'$; and $\{founded, unfounded\} \subseteq I'$.*

In future work, we intend to implement an automated version of the translation and integrate it with the DLVHEX system.

References

1. Eiter, T., Gottlob, G.: On the computational cost of disjunctive logic programming: Propositional case. Annals of Mathematics and Artificial Intelligence 15(3-4), 289–323 (1995)
2. Eiter, T., Ianni, G., Schindlauer, R., Tompits, H.: A uniform integration of higher-order reasoning and external evaluations in answer-set programming. In: Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI-05), pp. 90–96 (2005)
3. Eiter, T., Polleres, A.: Towards automated integration of guess and check programs in answer set programming: a meta-interpreter and applications. Theory and Practice of Logic Programming 6(1-2), 23–60 (2006)
4. Van Nieuwenborgh, D., De Cock, M., Vermeir, D.: An introduction to fuzzy answer set programming. Accepted for publication in AMAI

The Use of a Logic Programming Language in the Animation of Z Specifications

Margaret M. West

University of Huddersfield, Queensway, HD1 4DH, UK
M.M.West@hud.ac.uk
http://scom.hud.ac.uk/scommmw

1 Extended Abstract

Animation of a formal specification involves its execution and this paper is concerned with Z specifications and their *correct* animation. Since Z is based on typed set theory the logic programming language Gödel [2] was chosen as the execution language. *Abstract Approximation* was suggested in [1] to provide a formal framework and some proof rules for the correct animation of Z. We describe here how the correctness criteria are applied to our method of *structure simulation* [3].

The Gödel programming language is defined as a theory in first order logic and an implementation must be sound with respect to this semantics. Gödel is strongly typed so that the sorts in Gödel can model the types of Z, and also a set data type is supported. A small example is provided to give a 'flavour' of the simulation. A file system involves a single given set of file identifiers [*FileId*]. A global variable $MaxFiles : \mathbb{N}$ defines the maximum files of the system, where $MaxFiles > 0$. The schema *FileSys*[1] has components *Files*, *Count* where *Files* is a finite subset of the set *FileId*, and *Count* is the number of files in *Files*.

$$FileSys \mathrel{\widehat=} [Files : \mathbb{F} FileId;\ Count : 0 \mathinner{..} MaxFiles \mid \#Files = Count]$$

The given sets of the specification are declared as one of the base types (in Gödel) and some constants of the base types introduced to model the environment (for example `F1, F2, F3`). Other base types include schema and variable names and a 'binding type' `BindVar`, used to facilitate the binding formation of schemas. In order to be able to refer to variable names globally we have made them into constants in Gödel, hence upper case. These link to (lower case) variables via a 'binding function'. We require more than one kind of binding function to allow for different types in the second argument. Thus `Bind1(Files,files)` has a set of file ids as a second argument, `Bind2(Count, count )` has an integer as a second argument. In order to make a set or list of these, the return value of each is the same, `BindVar`. The code, query and result which follows is from Gödel module `Demo` and demonstrates the simulation of *FileSys*. Although not required here, a set library models functions, relations etc. to augment the set

[1] The horizontal form of schema definition is shown.

V. Dahl and I. Niemelä (Eds.): ICLP 2007, LNCS 4670, pp. 451–452, 2007.

operations already provided by Gödel. In more complex examples, it is necessary to instantiate some variable bindings. In contrast the query shown has variables uninstantiated.

```
% Schema for state FileSys - head contains a schema binding
SchemaType( [ Bind1(Files, files ), Bind2(Count, count )], FileSys)
  <- setFID = {x : IsFileId(x) } & %% schema predicate
    files Subset setFID & count In {y : 0 =< y =< 10} &
    Card(files, count).
%% Query - in this simple example all bindings are generated.
[Demo] <- SchemaType(b, FileSys).
b = [Bind1(Files,{}),Bind2(Count,0)] ? % first schema binding
b = [Bind1(Files,{F1}),Bind2(Count,1)] ?  %% second  schema binding
%% ... there are 8 possibilities in total.
```

In abstract approximation, the interpretation of Z syntactical objects in both the execution language (in our case Gödel) and in Z are compared. For correctness, the interpretation in the Gödel domain must always underestimate the interpretation in the Z domain. The approximation expresses the underlying concept of 'safeness' in abstract approximation, that a computation in Gödel should never provide more information than the result obtained by the evaluation of an expression in Z. The criteria involve a structural induction rule and takes place in the following order: base types, predicate expressions, declarations, schemas (see the report [4]).

For example *predicates* evaluate to either *true*, *false* or $\perp$ and at the same time the environment may be enhanced via resolution. Consider the state schema *FileSys* and the declaration $Files : \mathbb{F}\, FileId$. This is equivalent to the evaluation of the predicate $Files \subseteq FileId$. Before the computation, *Files* is uninstantiated ($= \perp$) and $FileId = \{F1, F2, F3\}$. We denote by $\mathcal{P}_Z[\![p]\!]\rho_Z$ the interpretation of syntactic predicates p in Z in an environment ρ_Z (and similarly for Gödel). Both interpretations evaluate to *true* and both enhance the environment in 8 possible ways. In this case the Gödel interpretation equals the Z. If it had not been possible to find values to satisfy the predicate then in both interpretations it would have evaluated to *false* and the environment is unaltered. If the program had floundered the predicate would have evaluated to $\perp$ and underestimated the equivalent Z interpretation.

References

1. Breuer, P.T., Bowen, J.: Towards Correct Executable Semantics for Z. In: Z User Workshop, Cambridge, June 1994, pp. 185–209. Springer, Heidelberg (1994)
2. Hill, P.M., Lloyd, J.W.: The Gödel Programming Language. MIT Press (1994)
3. West, M.M.: Types and Sets in Gödel and Z. In: Bowen, J.P., Hinchey, M.G. (eds.) ZUM 1995. LNCS, vol. 967, pp. 389–407. Springer, Heidelberg (1995)
4. West, M.M.: Report: Correctness criteria for the animation of Z specifications via a logic programming language. Technical report, University of Huddersfield (2007)

A Stronger Notion of Equivalence for Logic Programs

Ka-Shu Wong

University of New South Wales
and National ICT Australia
Sydney, NSW 2052, Australia
kswong@cse.unsw.edu.au

Abstract. Several different notions of equivalence have been proposed for logic programs with answer set semantics, most notably strong equivalence. However, strong equivalence is not preserved by certain logic program operators such as the strong and weak forgetting operators of Zhang and Foo. We propose a stronger notion of equivalence, called T-equivalence, which is preserved by these operators. We give a syntactic definition and provide a model-theoretic characterisation of T-equivalence. We show that strong and weak forgetting does preserve T-equivalence and using this, arrive at a model-theoretic definition of the strong and weak forgetting operators.

Introduction

The answer set semantics for logic programs is a dialect of logic programming with negation-by-failure, which allows it to handle both defaults as well as incomplete information [1]. Several different notions of equivalence have been proposed for logic programs with answer set semantics, the most widely studied being *strong equivalence* [2]: Two logic programs P and Q are strongly equivalent, if by adding any set of rules R to both P and Q, the resulting programs $P \cup R$ and $Q \cup R$ have the same answer sets.

However, for some applications strong equivalence is not strong enough. One case of this is with the strong and weak forgetting operators of Zhang and Foo [3]. There are strongly equivalent logic programs which do not remain strongly equivalent after the same forgetting operators are applied to both.

T-Equivalence. We address this problem by introducing a stronger notion of equivalence, which we call *T-equivalence*. It is defined syntactically using three inference rules to define a consequence relation, then saying that logic programs P and Q are T-equivalent iff the set of consequences of P and Q are the same.

Definition 1. *Let P be a logic program. Define the consequence relation $\vdash$ on logic program rules by the following inference rules:*

(C0) *$P \vdash r$ for every $r \in P$ and every r of the form $a \leftarrow a$ where a is an atom.*
(C1) *If $P \vdash a \leftarrow B, not\ C$ and X, Y are sets of atoms, then $P \vdash a \leftarrow B, X, not\ C, not\ Y$.*

V. Dahl and I. Niemelä (Eds.): ICLP 2007, LNCS 4670, pp. 453–454, 2007.

(C2) *If* $P \vdash a \leftarrow B_1, x, not\ C_1$ *and* $P \vdash x \leftarrow B_2, not\ C_2$*, then* $P \vdash a \leftarrow B_1, B_2, not\ C_1, not\ C_2$.

There is a model-theoretic characterisation of T-equivalence similar to the characterisation of strong equivalence by SE-models [4] and the characterisation of uniform equivalence by by UE-models [5]:

Definition 2. *Let* P *be a logic program, and let* X, Y *be sets of atoms. We say the pair* (X, Y) *is a T-model of* P *if* $X \models P^Y$*. Let* $M(P)$ *denote the set of all T-models of* P*.*

Theorem 1. *Let* P, Q *be logic programs. Then* P *and* Q *are T-equivalent iff* $M(P) = M(Q)$*.*

Forgetting. In the logic program context, *forgetting* is the modification of a program to remove certain facts while preserving as much information as possible. We consider the *strong forgetting* and *weak forgetting* operators defined by Zhang and Foo [3], which forgets sets of atoms from a logic program.

We define three postulates which encode desirable properties for an equivalence relation $\sim$ on logic programs in relation to a forgetting operator F. Let P be a logic program, and S a (possibly empty) ordered sequence of atoms. We write $F(P, S)$ for the program obtained from P by applying the forgetting operation with each element of S in order. The postulates are: **(1)** $F(P, S) \sim F(P, \pi(S))$ for any permutation π, **(2)** $P \sim Q \Rightarrow F(P, S) \sim F(Q, S)$, and **(3)** $P \sim Q \Rightarrow P$ strongly equivalent to Q.

We have the following result in relation to strong and weak forgetting:

Theorem 2. *For both strong forgetting and weak forgetting, T-equivalence satisfies postulates (1)–(3).*

This result allows us to construct model-theoretic versions of strong and weak forgetting in terms of T-models. For space reasons, the definitions of these operators are omitted here. These definitions, plus the proofs of the results presented here, can be found in the extended version of this paper [6].

References

1. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: Proceedings of the Fifth International Conference on Logic Programming pp. 1070–1080 (1988)
2. Lifschitz, V., Pearce, D., Valverde, A.: Strongly equivalent logic programs. Computational Logic 2(4), 526–541 (2001)
3. Zhang, Y., Foo, N.Y.: Solving logic program conflict through strong and weak forgettings. Articifial Intelligence 170(8-9), 739–778 (2006)
4. Turner, H.: Strong equivalence made easy: nested expressions and weight constraints. Theory and Practice of Logic Programming 3(4-5), 609–622 (2003)
5. Eiter, T., Fink, M.: Uniform equivalence of logic programs under the stable model semantics. In: Palamidessi, C. (ed.) ICLP 2003. LNCS, vol. 2916, pp. 224–238. Springer, Heidelberg (2003)
6. Wong, K.S.: A stronger notion of equivalence for logic programs. Technical Report UNSW-CSE-TR-0713, University of New South Wales (2007)

A Register-Free Abstract Prolog Machine with Jumbo Instructions

Neng-Fa Zhou

CUNY Brooklyn College & Graduate Center

1 Introduction

The majority of current Prolog systems are based on the WAM, in which registers are used to pass procedure arguments and store temporary data. In this paper, we present a stack machine for Prolog, named *TOAM Jr.*, which departs from the TOAM adopted in early versions of B-Prolog in that it employs no registers for temporary data and it offers variable-size instructions for encoding unification and procedure calls. TOAM Jr. is suitable for fast bytecode interpretation: the omission of registers facilitates instruction merging and the use of jumbo instructions results in more compact code and execution of fewer instructions than the use of fine-grained instructions. TOAM Jr. is employed in B-Prolog version 7.0. Benchmarking shows that TOAM Jr. helps significantly improve the performance: the execution speed is increased by 48% on a Windows PC and 77% on a Linux machine. Despite the overhead on standard Prolog programs caused by the adoption of a spaghetti stack to support event handling and constraint solving, B-Prolog version 7.0 compares favorably well with the state-of-the-art WAM-based Prolog systems.

2 The TOAM Jr. Instruction Set

TOAM Jr. inherites the memory architecture from the TOAM. There is a frame for each procedure call, which stores arguments, machine status registers, and local variables. A different set of machine status registers needs to be saved for each different procedure type, and hence frames for different types of procedures have different structures.

Instructions are classified into the following categories: *control* (`allocate`, `return`, `fork`, `cut`, and `fail`), *branch* (`jmpn_constant`, `switch_on_cons`, `jmpn_struct`, and `hash`), *move* (`move_list` and `move_struct`), *unify* (`unify_constant`, `unify_value`, `unify_list`, and `unify_struct`) and *call* (`call` and `last_call`). The following shows an example. An operand in the form of $y(Loc)$ denotes a frame slot where a positive offset refers to an argument and a negative offset refers to a local variable. A tagged operand can be an uninitialized frame slot $v(i)$, an initialized frame slot $u(i)$, or a constant $c(a)$. A singleton variable is denoted as $v(0)$. The binary literal '0b11101' is the layout bit vector for the last call, which indicates that all the arguments except for the second one (`Y`) are misplaced and need to be rearranged if the current frame is reused.

V. Dahl and I. Niemelä (Eds.): ICLP 2007, LNCS 4670, pp. 455–457, 2007.

```
% p(X,Y,Z):-S=f(X,Y),q(S),r(Z,Y,X,W,9).
p/3: allocate_det(3,5)                  % arity=3; frame size=5
     move_struct(y(-1),f/2,u(3),u(2))   % S=f(X,Y)
     call(q/1,u(-1))                    % q(S)
     last_call(0b11101,r/5,u(1),u(2),u(3),v(0),c(9))
```

TOAM Jr. offers specialized instructions that carry the numbers and types of operands in their opcodes, and also merged instructions each of which combines two or more base instructions.

3 Experimental Results

Table 1 compares B-Prolog version 7.0 (BP7.0) with version 6.9 (BP6.9) and three fast WAM implementations (Yap 5.1.1, SICStus 4.0, and hProlog 2.7.14) on CPU time on a Linux machine (3.8GHz CPU and 2G RAM). B-Prolog outperforms SICStus and Yap but is about 10% slower than hProlog. Benchmarking on a Windows XP machine reveals that BP7.0 is on average 48% faster than BP6.9, 12% faster than SICStus, and 27% faster than Yap. No Windows version of hProlog was available.

Table 1. Comparison on CPU times

program	BP7.0	BP6.9	Yap	Sics	hProlog
boyer	1	1.69	1.28	1.87	1.04
browse	1	1.72	1.45	1.55	0.74
chat_parser	1	1.56	1.36	1.82	1.13
crypt	1	1.56	1.65	1.94	0.92
meta_qsort	1	1.54	1.21	1.22	0.70
nreverse	1	3.31	1.17	1.03	0.62
poly_10	1	1.63	1.81	1.59	0.87
queens_8	1	1.68	1.29	1.60	0.75
reducer	1	1.62	1.43	1.73	0.98
sendmore	1	1.96	1.81	2.33	1.13
tak	1	1.70	3.25	2.42	1.09
zebra	1	1.30	0.83	1.07	0.91
mean	1	1.77	1.55	1.68	0.91

The speedup of BP7.0 over BP6.9 is mainly attributed to the use of specialized jumbo instructions, which would be more difficult if registers were existent. Other factors affect the performance too. In B-Prolog, the top bit of a word is reserved for tagging and, because of this, pointers have to be repaired after being untagged on Linux machines. Repairing pointers imposes about 5% speed overhead. hProlog uses the lowest three bits for tags and therefore requires no repairing of pointers. The B-Prolog and hProlog compilers are able

to detect the determinacy of a key predicate in `tak`. This is probably the main reason why they run faster on `tak` than the other two systems. B-Prolog has suspension frames stored on the stack (so called spaghetti stack), which facilitates context switching for action rules but incurs certain overhead on Prolog programs: even for a predicate that is made up of only one fact, the return instruction needs to check if the current frame is reusable since any predicate can be interrupted.

Advanced Techniques for Answer Set Programming

Martin Gebser

Institut für Informatik, Universität Potsdam, August-Bebel-Str. 89, D-14482 Potsdam, Germany
gebser@cs.uni-potsdam.de

Abstract. Theoretical foundations and practical realizations of answer set solving techniques are vital research issues. The challenge lies in combining the elevated modeling capacities of answer set programming with advanced solving strategies for combinatorial search problems. My research shall contribute to bridging the gap between high-level knowledge representation and efficient low-level reasoning, in order to bring out the best of both worlds.

Introduction and Motivation. Answer Set Programming (ASP; [1]) is a declarative branch of logic programming in which combinatorial search problems are encoded as logic programs whose answer sets then correspond to problem solutions. The first ASP solvers, namely, `dlv` [2] and `smodels` [3], were based on the Davis-Putnam-Logemann-Loveland procedure (DPLL). In the last decade, a new search algorithm called Conflict-Driven Clause Learning (CDCL; [4]) replaced DPLL as the basic procedure for state-of-the-art Boolean Satisfiability (SAT) solvers. While CDCL rests upon solid proof-theoretic fundaments, there currently is no canonical proof system for ASP, in the sense that different ASP solvers are usually not perceived as instances of a common proof-theoretic framework. In my opinion, the development of semantic and proof-theoretic foundations able to explain existing ASP solving strategies and further allowing the investigation of what is not yet implemented might have an impact that is at least twofold. First, focusing on essential mechanisms rather than on specific "heuristic" aspects certainly enhances the understanding of ASP solving approaches. Based on this, the secondary benefit, hopefully, will be a wider variety of more powerful ASP solvers than currently available. The availability of highly efficient solvers should then establish ASP as the primary framework for applications where its superior modeling capacities give it an edge on SAT and similar formalisms. My work shall contribute to the enhancement of ASP solving in order to further improve the practicability of ASP as a framework for knowledge representation and reasoning.

Research Summary. My research started with the question what would clause learning look like in an ASP solver. In the process of writing my diploma thesis [5] on it, I discovered that the major difficulties lay within the specifications of ASP solving algorithms. Indeed, after some failed attempts, I finally found that the approaches of ASP solvers were conceptionally much simpler than what the literature suggested. The first application of these findings was lookahead in the ASP solver `nomore++` [6]. Along the same lines, we noticed that the Lin-Zhao theorem could be strengthened [7]. Both works were further extended last year. First, we introduced a tableau system [8] explaining the inference patterns of all ASP solvers that do not substantially extend the language of input programs. A major result was that branching on rule bodies has a significant impact on proof complexity, that is, there are classes of logic programs yielding an exponential

V. Dahl and I. Niemelä (Eds.): ICLP 2007, LNCS 4670, pp. 458–459, 2007.

separation between our approach and the one of `dlv` and `smodels`; empirical evidence can be found in [9]. Second, we focused on the relationship between unfounded sets and loops. In [10], we introduced the notion of an elementary set, which allowed us to generalize the results in [7] to disjunctive programs, and in [11], we proposed algorithms for localizing the handling of unfounded sets in ASP solvers. Based on these works, we recently identified the class of Head-Elementary-set-Free (HEF) programs [12] that is more general than the class of Head-Cycle-Free (HCF) programs. Still, verifying the stability of models is tractable for HEF programs, and minimal nonempty unfounded sets can be computed efficiently, properties that may be exploited in the future to improve disjunctive ASP solvers. My other recent works consist of contributing to the development of ASP systems, namely, the conflict-driven ASP solver `clasp` [13,14,15], the grounder `gringo` [16], and the debugging support tool `spock` [17].

References

1. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press, Cambridge (2003)
2. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV system for knowledge representation and reasoning. ACM TOCL 7(3), 499–562 (2006)
3. Simons, P., Niemelä, I., Soininen, T.: Extending and implementing the stable model semantics. Artificial Intelligence 138(1-2), 181–234 (2002)
4. Mitchell, D.: A SAT solver primer. Bulletin of the EATCS 85, 112–133 (2005)
5. Gebser, M.: Backjumping and learning in answer set programming. Diploma thesis (2005)
6. Anger, C., Gebser, M., Linke, T., Neumann, A., Schaub, T.: The nomore++ approach to answer set solving. In: Sutcliffe, G., Voronkov, A. (eds.) LPAR 2005. LNCS (LNAI), vol. 3835, pp. 95–109. Springer, Heidelberg (2005)
7. Gebser, M., Schaub, T.: Loops: Relevant or redundant? In: Baral, C., Greco, G., Leone, N., Terracina, G. (eds.) LPNMR 2005. LNCS (LNAI), vol. 3662, pp. 53–65. Springer, Heidelberg (2005)
8. Gebser, M., Schaub, T.: Tableau calculi for answer set programming. In: Etalle, S., Truszczyński, M. (eds.) ICLP 2006. LNCS, vol. 4079, pp. 11–25. Springer, Heidelberg (2006)
9. Anger, C., Gebser, M., Janhunen, T., Schaub, T.: What's a head without a body? In: Proceedings of ECAI'06, pp. 769–770. IOS Press, Amsterdam (2006)
10. Gebser, M., Lee, J., Lierler, Y.: Elementary sets for logic programs. In: Proceedings of AAAI'06, AAAI Press, Stanford (2006)
11. Anger, C., Gebser, M., Schaub, T.: Approaching the core of unfounded sets. In: Proceedings of NMR'06, pp. 58–66 (2006)
12. Gebser, M., Lee, J., Lierler, Y.: Head-elementary-set-free logic programs. [18], pp. 149–161
13. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Conflict-driven answer set solving. In: Proceedings of IJCAI'07. AAAI Press/MIT Press, pp. 386–392 (2007)
14. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: clasp: A conflict-driven answer set solver. [18], pp. 260–265
15. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Conflict-driven answer set enumeration. [18], pp. 136–148
16. Gebser, M., Schaub, T., Thiele, S.: Gringo: A new grounder for answer set programming. [18], pp. 266–271
17. Brain, M., Gebser, M., Pührer, J., Schaub, T., Tompits, H., Woltran, S.: Debugging ASP programs by means of ASP. [18], pp. 31–43
18. Baral, C., Brewka, G., Schlipf, J. (eds.): LPNMR 2007. LNCS, vol. 4483. Springer, Heidelberg (2007)

A Games Semantics of ASP

Jonty Needham and Marina De Vos

University of Bath
{jmn20,mdv}@cs.bath.ac.uk

Abstract. We present a games semantics of $AnsProlog$with single headed clauses. Using the semantics, we derive tools for the assistance of verification of programs and tools based on those programs, a well as proposing a solver that grounds on the fly.

1 Introduction

Answer set programming, otherwise knows as ASP or $AnsProlog$is a relatively new formalism in logic programming. $AnsProlog$programs consist of headed clauses which allow negation in the bodies of the rules. See [1] for more details.

We aim to use games semantics to build a fully abstract denotational semantics of $AnsProlog$. In this abstract, we cover very briefly our work so far which at the moment consists of a correct denotational semantics from which we have derived some algorithmic tools. At present there is no mathematical model of ASP other than that which we propose here.

The work on algorithmic games semantics was pioneered in [3] where the intensional nature of games semantics was exploited to build tools for program analysis. We have shown that the debugging algorithm developed in [2] is indeed correct using our semantics. Similarly, we have shown that it is possible to construct a solver using the semantics which does grounding only when it is essential to the computation of the answer set, bypassing much unneeded grounding.

2 Preliminary Results

Games semantics has been around for many years in one form or another, starting with Lorenzen's paper describing intuitionistic reasoning using interaction[4]. Games semantics became much more mathematically formal with the introduction of the arena, see [3] for more details.

Traditionally, games correspond to the types of a language with winning strategies denoting terms. The games are defined by a set of moves, a labelling function on those moves stating which player plays each move and whether it's a question or answer, and a justification relation which states which moves can be played when.

Alongside this we have such accessibility rules such as that of *polite dialogue*. This means each player must listen to the other and answer them or ask a question that is clearly justified by the statement preceding it, but we do not have space here to include all the rules. Lastly, winning for a player involves having the last move.

V. Dahl and I. Niemelä (Eds.): ICLP 2007, LNCS 4670, pp. 460–461, 2007.

We have chosen to denote the game to be derived from the Herbrand base, with consistent programs over that base being the winning strategies. The game-play continues as a dialogue between Proponent and Opponent; P's aim is to win by establishing the truth (or unprovability) of each of the atoms, and O aims to win by revealing inconsistencies in P's strategy, putting him in a position of being unable to make a move.

Player	Move	Justification
O	$a?$	Opening
P	$a \leftarrow b?$	query evidence for a
O	$b?$	Query validity of P's evidence
P	$b \leftarrow?$	Evidence for b
O	$\top$	answer to P's request
P	b	answer to O's query about b
O	$\top$	answers P's query about $a \leftarrow b$
P	a	hence winning the game

Fig. 1. Play for the program Π

The game for a Herbrand base, $[\![HB_\Pi]\!]$ has as its set of moves $M_{HB} = \{a?|a \in HB_\Pi\} \cup \{a| \ a \in HB_\Pi\} \cup \{a \leftarrow B?| \ a \in HB_\Pi, \ B \subset HB_\Pi\} \cup \{\top, \bot\}$, of which moves of the form $a?$ are O-questions, $a \leftarrow?B$ are P questions, a, **not** a are P ans and $\top, \bot$ are O answers.

Consider the program $\Pi = \{a \leftarrow b; b \leftarrow\}$.The play for this program continues as in Figure 1. The answer set is obtained directly from P's positive answers.

We have already obtained the correctness of the semantics using the normal induction technique, and we have used the dialogue style developed to prove the correctness of the debugging algorithm in [2]

3 Future Work

Constructing a solver based on this semantics is a project in which we have already made some progress; there seems to be major savings in grounding computation.

Obtaining a fully abstract model of $AnsProlog$is the major goal; that is a model that is not only denotationally correct but also complete, and then use it to derive semantics for related models such as equilibrium logic.

Above all we want to develop a series of tools for verification and development of $AnsProlog$programs in a similar vein to [3].

References

[1] Baral, C.: Knowledge Representation, reasoning and declarative problem solving with Answer Sets. Cambridge University Press, Cambridge (2003)

[2] Brain, M., De Vos, M.: Debugging logic programs under the answer set semantics. ASP, pp. 141–152 (2005)

[3] Ghica, D.R., McCusker, G.: Reasoning about idealised algol using regular languages. In: Welzl, E., Montanari, U., Rolim, J.D.P. (eds.) ICALP 2000. LNCS, vol. 1853, Springer, Heidelberg (2000)

[4] Lorenzen, P.: Ein dialogisches konstruktivitatskriterium. In Symp. Foundations of Mathematics, volume Infinitistic Methods, pp. 193–200, Warsaw, PWN (1961)

Modular Answer Set Programming

Emilia Oikarinen

Helsinki University of Technology (TKK), P.O. Box 5400, FI-02015 TKK, Finland

Introduction

In answer set programming (ASP) a problem is solved declaratively by writing down a logic program the answer sets of which correspond to the solutions of the problem, and computing the answer sets of the program using a special purpose search engine. The growing interest towards ASP is mostly due to efficient search engines, such as SMODELS and DLV. Consequently, a variety of interesting applications of ASP has emerged, e.g., in planning, product configuration, and computer aided verification. Despite the declarative nature of ASP the development of programs resembles that of programs in conventional programming, i.e., a programmer often develops a series of programs for a particular problem, e.g., when optimizing execution time and space. This gives rise to a meta-level problem of verifying whether subsequent programs are equivalent. To solve the equivalence verification problem a translation-based approach has been proposed and extended further [1,2,3]. The idea is to combine logic programs P and Q into logic programs $\mathrm{EQT}(P, Q)$ and $\mathrm{EQT}(Q, P)$ which have no answer sets iff P and Q are equivalent, which allows the use of the same ASP solver for the equivalence verification task as for the search of answer sets. The translation-based method treats programs as integral entities which limits its usefulness, e.g., if a small local change is made in a large program. The same line of thinking applies to current ASP methodology in general. ASP programs are typically seen as integral entities, and there is a lack of mechanisms available in modern programming languages that ease program development by allowing re-use of code or breaking programs into smaller pieces.

Objectives. The aim of this research is to obtain a deeper understanding of program development within ASP. We want to make program development in ASP easier and, more generally, make ASP methodology more accessible for specialists in other fields than computer science. We want to develop ASP into a more *module-oriented direction* in which ASP programs consist of modules that are combined through suitable *interfaces*. Modularization of ASP is a way to structure and ease the program development process, and to make ASP an even more attractive approach for solving hard combinatorial problems, e.g., in the areas of Semantic Web and bioinformatics. Modularity has been studied extensively in *conventional logic programming* [4], but there are only a few approaches for incorporating modularity into ASP, see e.g. [5,6].

Current stage of research. In [7,8,9,10] we propose a simple and intuitive notion for ASP modules that interact through an input/output interface. This is achieved by accommodating program modules proposed by Gaifman and Shapiro [11] in conventional logic programming to the context of ASP. One of the main results is a module theorem showing that the ASP module system is compositional with respect to the semantics. Furthermore, we introduce an equivalence relation (modular equivalence) that is a proper congruence relation for composition of modules, and show that deciding modular equivalence is **coNP**-complete for modules that have well established

V. Dahl and I. Niemelä (Eds.): ICLP 2007, LNCS 4670, pp. 462–463, 2007.

communication interface. This allows us to extend the earlier translation-based method for verification of so-called visible equivalence [3] to cover the verification of modular equivalence. Based on an experimental evaluation, modularization of the verification of (modular) equivalence seems to be a good idea, especially when the common context shared by the modules is large and the number of submodules stays reasonable.

Open issues. A crucial step further is to extend the concept of modularity to the *non-ground case*, i.e., to consider program modules involving *variables*. Understanding the relations between ASP and other constraint programming approaches is essential, when considering a (more) general view on *module-oriented constraint programming*. There is a need for a well-defined *module interface* that gives support for other constraint programming approaches, such as *propositional satisfiability*, *constraint programming*, and *circumscription*, in addition to ASP. We have already studied the relation between *circumscription* and *ASP* in [12,13] introducing a translation-based method for embedding *parallel* circumscription into ASP in and later extending the method to the case of *prioritized* circumscription. The translation-based method allows the use of efficient ASP solvers for computing circumscription without developing a special purpose solver for the task.

References

1. Janhunen, T., Oikarinen, E.: Testing the equivalence of logic programs under stable model semantics. In: Flesca, S., Greco, S., Leone, N., Ianni, G. (eds.) JELIA 2002. LNCS (LNAI), vol. 2424, pp. 493–504. Springer, Heidelberg (2002)
2. Oikarinen, E., Janhunen, T.: Verifying the equivalence of logic programs in the disjunctive case. In: Lifschitz, V., Niemelä, I. (eds.) Logic Programming and Nonmonotonic Reasoning. LNCS (LNAI), vol. 2923, pp. 180–193. Springer, Heidelberg (2003)
3. Janhunen, T., Oikarinen, E.: Automated verification of weak equivalence within the smodels system. Theory and Practice of Logic Programming (to appear)
4. Bugliesi, M., Lamma, E., Mello, P.: Modularity in logic programming. Journal of Logic Programming 19/20, 443–502 (1994)
5. Eiter, T., Gottlob, G., Veith, H.: Modular logic programming and generalized quantifiers. In: Fuhrbach, U., Dix, J., Nerode, A. (eds.) LPNMR 1997. LNCS, vol. 1265, pp. 290–309. Springer, Heidelberg (1997)
6. Tari, L., Baral, C., Anwar, S.: A language for modular answer set programming: Application to ACC tournament scheduling. In: ASP'05, CEUR-WS.org (2005)
7. Oikarinen, E.: Modular answer set programming. Research Report A106, Helsinki University of Technology, Laboratory for Theoretical Computer Science, Espoo, Finland (2006)
8. Oikarinen, E., Janhunen, T.: Modular equivalence for normal logic programs. In: ECAI'06, pp. 412–416. IOS Press, Amsterdam (2006)
9. Oikarinen, E.: Modularity in smodels programs. In: Baral, C., Brewka, G., Schlipf, J. (eds.) LPNMR 2007, LNCS (LNAI), vol. 4483, pp. 321–326. Springer, Heidelberg (2007)
10. Janhunen, T., Oikarinen, E., Tompits, H., Woltran, S.: Modularity aspects of disjunctive stable models. In: Baral, C., Brewka, G., Schlipf, J. (eds.) LPNMR 2007, LNCS (LNAI), vol. 4483, pp. 175–187. Springer, Heidelberg (2007)
11. Gaifman, H., Shapiro, E.Y.: Fully abstract compositional semantics for logic programs. In: POPL'89, pp. 134–142. ACM Press, New York (1989)
12. Janhunen, T., Oikarinen, E.: Capturing parallel circumscription with disjunctive logic programs. In: Alferes, J.J., Leite, J.A. (eds.) JELIA 2004. LNCS (LNAI), vol. 3229, pp. 134–146. Springer, Heidelberg (2004)
13. Oikarinen, E., Janhunen, T.: A linear transformation from prioritized circumscription to disjunctive logic programming. In: ICLP'07 (to appear)

Universal Timed Concurrent Constraint Programming

Carlos Olarte[1,3], Catuscia Palamidessi[1], and Frank Valencia[2]

[1] INRIA Futurs, LIX, École Polytechnique, France
[2] CNRS LIX, École Polytechnique, France
[3] Department of Computer Science, Javeriana University Cali, Colombia
{carlos.olarte,catuscia,frank.valencia}@lix.polytechnique.fr

Abstract. In this doctoral work we aim at developing a rich timed concurrent constraint (tcc) based language with strong ties to logic. The new calculus called *Universal Timed Concurrent Constraint* (utcc) increases the expressiveness of tcc languages allowing infinite behaviour and mobility. We introduce a constructor of the form (**abs** $x, c)P$ (*Abstraction* in P) that can be viewed as a dual operator of the hidden operator **local** x **in** P. i.e. the later can be viewed as an existential quantification on the variable x and the former as an universal quantification of x, executing $P[t/x]$ for all t s.t. the current store entails $c[t/x]$. As a compelling application, we applied this calculus to verify security protocols.

1 Introduction

Concurrent Constraint Programming (ccp) [3] is a well-established and mature model for concurrency with several reasoning techniques and strong ties to logic. ccp agents can alternatively be viewed as logic formulae, algebraic terms and computational processes. ccp is based on a monotonic shared-memory model and parametric in an information system. Processes interact by communicating through the shared store posting new constraints (**tell**(c) operator) or testing the structure of the store (**ask** c **then** P) for synchronisation purposes.

Timed Concurrent Constraint (tcc) [2] is a temporal extension of ccp aimed at specifying reactive systems. In tcc time is conceptually divided into discrete intervals and computation occurs in bursts of activity. When an stimulus (i.e. a constraint) is received from the environment, a tcc process is executed with that constraint as the initial store. When the resting point is reached, the environment can observe the store produced and a residual process is computed to be executed in the next time interval. As is shown in [2], tcc programs can be compiled into finite state automata.

Motivated in models for the analysis of security protocols where it is necessary to deal with the unbounded capabilities of the spy, in this doctoral work we are interested in increasing the expressiveness of tcc by adding two distinguished capabilities: (1) ability to express infinite behavior and (2) mobility. (1) will allow us to model complex systems such as those emerging e.g. in systemic biology and security and (2) will lead us to a name passing discipline in the tcc model. We have demonstrated that this new language is Turing complete.

V. Dahl and I. Niemelä (Eds.): ICLP 2007, LNCS 4670, pp. 464–465, 2007.

2 An Universal Binder (Abstractions)

utcc is a derived language from tcc adding a new construct for process abstraction. This construct takes the form (**abs** x, c) P where intuitively $P[t/x]$ is executed for every possible term t s.t. the current store can entail the constraint $c[t/x]$. This operator is dual w.r.t. the hiding operator (**local** x , c)P where the former can be viewed as *forall* x *s.t.* $c(x)$ *do* P and the latter as *there exists* x *s.t.* $c(x)$ *and* P.

Formalising this new construct has challenging technical problems. In tcc, operational semantics requires that processes quiesce in a finite number of internal reductions to guarantee *instantaneous responses*[2]. Nevertheless, abstractions can easily generate infinite behaviour within a time unit. For example, consider the ability of composing messages posted in the network, i.e. given two messages m_1 and m_2, the spy can build a new compounded message $\{m_1, m_2\}$. An abstraction modelling this fact could be (**abs** x, $out(x)$)(**abs** y, $out(y)$)$out(\{x, y\})$ where *out* is an uninterpreted predicate in the constraint system. Given the output of the messages m_1 and m_2, this process generates a new one ($out(\{m_1, m_2\})$) and with this, a new reduction can take place producing $out(\{m_1, \{m_1, m_2\}\})$ and so on. Thus the resting point will never be reached.

Inspired in works such as [1], we propose a symbolic semantics for utcc able to compute in a single symbolic step a possible infinite number of internal reductions in the operational semantics. The key point in this approach is to find a constraint representing the possible infinite number of constraints generated by reductions in the operational semantics.

We believe that utcc has much to offer to the concurrency theory community. In particular, to reason about security protocols. The underlying assumptions of utcc are reminiscent of those process calculi used for security. The protocols can be represented in a declarative way and reasoned about using the techniques utcc enjoys. Namely, operational, symbolic and denotational semantics. Furthermore, utcc allows for verification of reachability properties using a proof system based on Linear Temporal Logic.

References

1. Boreale, M.: Symbolic trace analysis of cryptographic protocols. In: Orejas, F., Spirakis, P.G., van Leeuwen, J. (eds.) ICALP 2001. LNCS, vol. 2076, Springer, Heidelberg (2001)
2. Saraswat, V., Jagadeesan, R., Gupta, V.: Foundations of timed concurrent constraint programming. In: Abramsky, S., (ed.): Proceedings of the 9th Annual IEEE Symp. on Logic in Computer Science, LICS (1994)
3. Saraswat, V.A., Rinard, M., Panangadnen, P.: Semantic foundation of Concurrent Constraint Programming. In: Proc. of 18th Annual ACM Symp. on Principles of Programming Languages, ACM Press, New York (1991)

Extension and Implementation of CHR
(Research Summary)

Peter Van Weert*

Department of Computer Science, Katholieke Universiteit Leuven,
Celestijnenlaan 200A, 3001 Heverlee, Belgium
Peter.VanWeert@cs.kuleuven.be

Introduction

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

CHR aims at supporting a very high-level, declarative programming style. Declarative programs simply describe the problem, leaving the choice of the algorithm to solve it, and all implementation details, to the underlying language. Efficiently implementing declarative languages therefore presents many interesting challenges. The end-user though, enjoys considerably reduced development times, and vastly improved understandability, maintainability and robustness.

Practice, however, shows that CHR's conciseness and expressiveness is frequently lacking, in particular when used as a general purpose language. Many programming idioms can only be realized in CHR using tedious auxiliary constraints and rules. Moreover, these auxiliary constructs often cross-cut the entire program, obfuscating its readability. These error-prone and cumbersomely repetitive solutions clearly impair the advantages of declarative programming.

The main goal of my research is to improve the usability of the CHR language, by *extending* the language with expressive, declarative language features, and by developing, *implementing*, and evaluating new and existing program analyses and compilation techniques for (extended) CHR programs.

Extending CHR. We extended the CHR language with *negation as absence* [12, 13], allowing CHR rules to test for the *absence* of constraints. We defined a formal operational semantics, and realized a prototype implementation in SWI-Prolog. We showed the declarative advantages of negation as absence, and evaluated the issues of integrating it with the refined operational semantics of CHR [3].

Recently, we generalized negation as absence to a much more powerful language feature, called *aggregates* [8, 9]. The proposed framework, implemented using source-to-source transformations to regular CHR, supports nested aggregate expressions, efficient incremental aggregate computation and application-tailored user-defined aggregates. Case studies clearly demonstrate the gained

* Research Assistant of the Research Foundation – Flanders (FWO-Vlaanderen).

V. Dahl and I. Niemelä (Eds.): ICLP 2007, LNCS 4670, pp. 466–468, 2007.

expressiveness and conciseness, and that the desired runtime complexity is attainable with an acceptable constant time overhead.

Implementing CHR. I developed a state-of-the-art CHR compiler and runtime system for Java, called K.U.Leuven JCHR [11]. It features a statically typed declarative syntax, a tight integration with the object-oriented host-language, extensive static analysis, and a compilation to highly optimized code. It outperforms other CHR systems by up to several orders of magnitude.

In future work, I will develop a first efficient parallel CHR system[1]. Leveraging the full power of current and future multicore processors demands highly concurrent software [10]. Writing concurrent programs, however, is notoriously difficult. The abstract of [7] reads: "For concurrent programming to become mainstream, we must discard threads as a programming model. Nondeterminism should be judiciously and carefully introduced where needed, and it should be explicit in programs." Clearly, the inherently parallel CHR language provides a valid solution [5]. The parallelization of forward chaining rule-based systems has never been fully successful in the past [2]. I believe the key to effective parallel matching is *lazy matching*, especially when combined with other forms of parallellism available in CHR programs, such as parallel rule firing and parallel search. Important problems are still to be researched in all aspects of the system, from language features and semantics, to analysis, implementation, and optimization.

References

1. The CHR Home Page, `http://www.cs.kuleuven.be/~dtai/projects/CHR/`
2. Amaral, J.N., Ghosh, J.: Speeding up production systems: concurrent matching to parallel rule firing. In: Chapter 7 of Parallel Processing for AI, Elsevier, Amsterdam (1994)
3. G. J. Duck, P. J. Stuckey, de la Banda, M.G. Holzbaur, C.: The refined operational semantics of CHR. In 27th Intl. Conf. on Logic Programming (2004)
4. Frühwirth, T.: Theory and practice of Constraint Handling Rules. Journal of Logic Programming 37(1–3), 95–138 (1998)
5. Frühwirth, T.: Parallelizing union-find in Constraint Handling Rules using confluence. In: Gabbrielli, M., Gupta, G. (eds.) ICLP 2005. LNCS, vol. 3668, Springer, Heidelberg (2005)
6. Lam, E., Sulzmann, M.: A concurrent CHR implementation in Haskell with STM. In Workshop on Declarative Aspects of Multicore Programming, Nice (2007)
7. Lee, E.A.: The problem with threads. Computer 39(5), 33–42 (2006)
8. Sneyers, J., Van Weert, P., Schrijvers, P., Demoen, B.: Aggregates in CHR. Technical Report CW 481, Leuven, K.U. Dept. of Computer Science, Belgium (2007)
9. Sneyers, J., Van Weert, P., Schrijvers, T., Demoen, B.: Aggregates in CHR – extended abstract. In 23rd Intl. Conf. on Logic Programming, Porto (2007)
10. Sutter, H.: The free lunch is over: A fundamental turn toward concurrency in software. Dr. Dobb's Journal 30(3), 16–20 (2005)

[1] Some preliminary, semi-naive implementations already exist [6], but these cannot compete with fully optimized sequential compilers.

11. Van Weert, P., Schrijvers, T., Demoen, B.: Leuven K.U. JCHR: user-friendly, flexible and efficient CHR for Java. In Second Workshop on Constraint Handling Rules, pp. 47–62, Sitges, Spain (October 2005)
12. Van Weert, P., Sneyers, J., et al.: To CHR$^{\neg}$ or not to CHR$^{\neg}$: Extending CHR with negation as absence. Technical Report CW 446, Leuven K.U. (May 2006)
13. Van Weert, P., Sneyers, J., Schrijvers, T., Demoen, B.: Extending CHR with negation as absence. In Third Workshop on CHR, pp. 125–139, Venice (2006)

Author Index